BESTURINGSSYSTEMEN EXAM 2023-2024

Prof F. Piessens - Prof J. Van Bulck

Liam Luys

KU Leuven

Veel succes met het examen!
Ik heb geprobeerd een samenvatting te maken van de cursustekst.
Geen garantie dat alles er in staat en/of juist is, doe zeker de boeken ook open.
Foutje gevonden? Laat het zeker weten, dan pas ik het aan!

Les 1 - Inleiding

dinsdag 26 september 2023 11:06

Liam Luys – r0934449 - 2e BA Informatica
Besturingssystemen

KU LEUVEN

Vakinhoud

Hoorcolleges

Elke week hoorcollege met leerstof
Uitleg, met illustraties, volgen gehele cursustekst met extra's

Kennis voor examen = cursustekst

Homeworks

Op eigen initiatief oefeningen voor het vak

Title	Assignment directory	Chapters	Submission deadline
Setup environment	setup	-	No submission
Simple shell	simple-shell	4,5	23/10/2023
Scheduling simulations	sched-sim	7,8,9	No submission
Working with helgrind	intro-helgrind	26,27	No submission
Concurrent queues	concurrent-queues	28,29	27/11/2023
Concurrent accounts	concurrent-accounts	30,31,32	11/12/2023

Practica

Verplichte oefenmomenten op punten.
Voorbereiding op elke sessie

Oefenzitting	Datum sessie	Deadline permanente evaluatie	Voorbereiding
OS Interfaces	4 oktober	10/10 23h59 - 1 week	xv6 boek H1
System calls	11 oktober	24/10 23h59 - 2 weeks	xv6 boek H2
Virtual Memory	25 oktober	7/11 23h59 - 2 weeks	xv6 boek H3
Traps	8 november	14/11 23h59 - 1 week	xv6 boek H4 + H5
Locks	15 november	28/11 23h59 - 2 weeks	xv6 boek H6 + H7
File systems	29 november	12/12 23h59 - 2 weeks	xv6 boek H8
TBA	6 december	-	-

2. Introduction to Operating Systems

Weten hoe computer werkt

- Is onmogelijk door complexiteit
- Basics kan je leren begrijpen

Complexiteit

- Kunnen we begrijpen door abstracties
    - Enkel essentiële eigenschappen voor het specifieke doel

- Computer is abstracte machine

- Abstracties in lagen
    - Implementatie van abstracties dmv andere abstracties
    - Compiler, omzetten hoog niveau naar computerniveau, …

Besturingssysteem
- Zorgt voor manier om programma's uit te voeren en te interageren met elkaar
- Geeft een abstractie die een computer eenvoudiger maakt om te gebruiken

- 3 Onderdelen
    - Virtualization / Virtualisatie
        - Hoe de illusie creëren dat programma alle resources heeft?

    - Concurrency / Gelijktijdigheid
        - Hoe synchroniseren tussen programma's die gelijktijdig uitvoeren?

    - Persistence / Behouden gegevens
        - Hoe gegevens bewaren bij herstarten/falen van computer?

- Biedt instructies interface (API) waarmee programma's kunnen interageren
    - Een standard library van system calls

Virtualisatie
- Hoe meerdere processen resources aanbieden en controle behouden?
- Virtualiseren van CPU
    - Indruk geven dat er zeer veel CPU's zijn met maar één hardware CPU

- Virtualiseren van Geheugen
    - Geheugen (= Array) individueel voor elk proces virtualiseren
    - Elk proces een eigen virtuele adresruimte

Gelijktijdigheid
- Multi-threaded programma's
- Hoe meerdere threads die hetzelfde geheugen aanspreken succesvol laten werken?
- Wanneer gaat het fout?

Persistence
- Hoe data persistent opslaan zonder gegevensverlies bij herstart/falen computer?
- Hardware I/O en software combineren
- File System
    - Software die de disk regelt
    - Deel van OS

Design Goals OS
- Performance
- Minimaliseren van overhead
- Beveiliging en isolatie
- Efficiënt en betrouwbaar

Nagekeken en vervolledigd op 21/12/2023

I - VIRTUALISATIE

4. The Abstraction: The Process

4.1 The Abstraction: A Process

Proces
- Programma in uitvoering
- Bevat gegevens over programma
  - Geheugengebruik, registerinhoud, …
  - Adress Space, Program Counter PC, Instruction Pointer IP, …

- Wordt geregeld via API van besturingssysteem
- Kan in verschillende toestanden zijn

Virtualisatie CPU
- Computer heeft beperkt aantal CPU's, maar programma's willen elk een CPU voor zich
- Fysische CPU opgedeeld in meerdere virtuele CPU's
- Proces moet indruk krijgen dat het de hele tijd bezig is

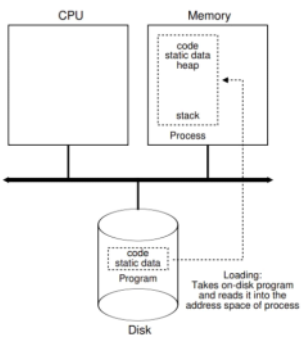
Verdeling computer resources (CPU, geheugen, …) over processen
- Time Sharing
  - Resources om de beurt aan elk proces geven (context-switch)

- Space Sharing
  - Resources opdelen en elk proces een deel geven

Noodzakelijke Componenten
- Mechanisme
  - Hoe kunnen we met de CPU switchen tussen processen?

- Policies
  - Welk proces krijgt de CPU en voor hoe lang?

Proces Aanmaak
- Hoe kan OS programma uitvoeren?
  - Data laden (load) in address space van proces
    - Eagerly: hele code overzetten voor de uitvoer van het programma
    - Lazily: gedeeltes van de code overzetten en als nodig andere delen overzetten



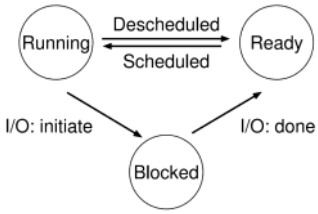
- Geheugen reserveren voor run-time stack
  - Gebruikt voor lokale variabelen, functieparameters, returnwaardes

- Geheugen reserveren voor heap
  - Dynamische allocatie (malloc)

- Starten met uitvoering van main-functie van programma

Proces Toestanden

- Running = Proces dat werkelijk in uitvoering is door de CPU (kan er maar 1 zijn per CPU)
- Ready = Proces dat wacht op de CPU
- Blocked = Proces kan OS niet gebruiken en wacht op andere operaties, bv wacht op gebruikersinput I/O



Overgangen door Process Scheduler

- Final / Zombie = Proces uitgevoerd maar nog niet opgeruimd, andere processen kunnen return nog bekijken

Process List

- Datastructuur binnen OS
- Houdt alle bestaande processen bij

- **<u>Process Control Blocks</u>**
  - Info per proces bijhouden
  - Registers, status, Proces Identifier PID, …

- Wordt gebruikt bij Context Switches

```
enum procstate { UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

```
struct context {
  uint64 ra;
  uint64 sp;

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};
```

5. Process API

Welke interfaces gebruikt OS voor aanmaak en beheer van processen?

Basiselementen API

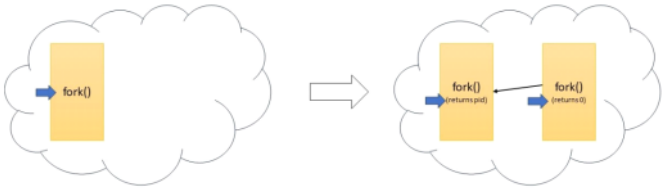
- Fork
- Wait
- Exec

Fork System Call

- Maakt exacte kopie van bestaand proces
    - Genereerd een nieuw proces met andere adress space en PID

- Enige verschil is return-waarde van de fork-call
    - Bij nieuw duplicaat een 0, bij ouder (origineel proces) een int > 0

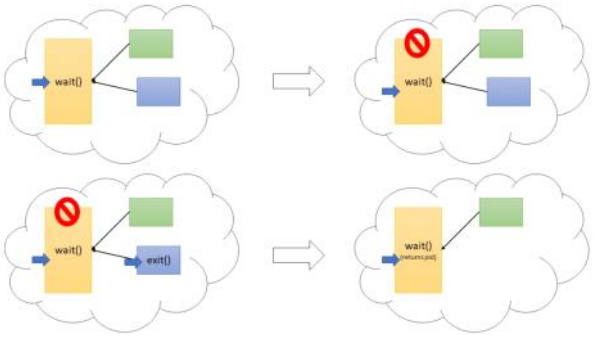
- Processen gaan afzonderlijk met kind en ouder verder
    - Niet deterministische volgorde! (CPU Scheduler)
        - Problemen met concurrency



Wait System Call

- Wachten tot 1 van de kinderen klaar is met uitvoering (exit)
- Maakt uitvoer van fork wel deterministisch (parent wacht tot kind)
- Proces gaat in "blocked" toestand tot een kind klaar is, dan naar running

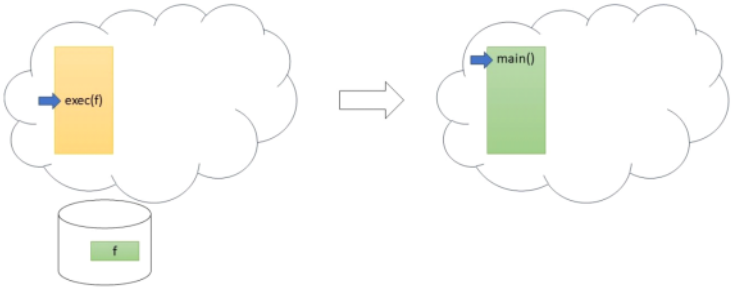
- Waitpid: Als je op specifiek kind wil wachten



Exec System Call

- Laadt een programma van gegeven adres
- Zet het programma in de plaats van originele code en voert het uit
- Keert niet terug bij succesvolle uitvoering

- Maakt geen nieuw proces aan!



Waarom fork en exec elementen gesplitst?

- Zo heeft OS de macht om nog tussen te komen voor de uitvoering van het kind (voor de exec, na de fork)

- I/O Redirection
    - Invoer en uitvoerregeling aanpassen (bv bestand als input, output naar bestand/nieuw programma)

Invoer/Uitvoer via terminal regelen

- Als invoer een bestand gebruiken: command < file.txt
- Als uitvoer naar een bestand: command > output.txt

OS Shell

- Programma waarmee gebruikers interageren met OS

- Grafische User Interface Shell GUI
    - Als je programma opent zie je een venster waarmee je kan interageren

- Command Line Interface Shell CLI
    - Werken in terminal, programma's starten door naam in te

I/O Redirection Implementatie

- File Descriptor
    - Int die verwijst naar een I/O geïmplementeerd in de Kernel
    - 0 = Standaard Input, 1 = Standaard Output, 2 = Standaard Error Output
    - File descriptor allocatie op laagst beschikbare nummer

Andere Operaties

- Veel mogelijkheden om interactie met proces mogelijk te maken
- Kill System Call: Kan signalen sturen naar andere processen
- Signal System Call: Kan signalen opvangen vanuit Kill verzonden
- Toegangscontrole: Wie heeft toegang tot API calls?
    - Superuser heeft toegang tot alles

6. Limited Direct Execution

Hoe efficiënt en veilig CPU virtualisatie zonder controleverlies?

Vragen bij virtualisatie

- Performance: hoe implementeren zonder overhead?
- Control: hoe de controle als OS blijven behouden?

Limited Direct Execution

- Direct Execution
    - Code direct op de hardware, geen interpreter/vertolker

OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute **call** main()	
	Run main()
	Execute **return** from main
Free memory of process	
Remove from process list	

- Limited Direct Execution
  - Programma direct op CPU uitvoeren
  - Beperkingen opleggen om veiligheid te bewaren en time sharing mogelijk te maken

Zonder limitatie heeft OS geen controle en zou het een gewone library zijn

Probleem 1: Beperken wat processen kunnen doen - Restricted Operations

Wat als proces verboden operaties uitvoert, I/O bewerkt of data buiten zijn address space wil lezen?

Doel Restricted Operations

- Proces toegang geven tot I/O instructies, maar geen volledige controle geven
- Hardware en OS laten samenwerken

Twee Toestanden Processor

- User Mode
  - Geen geprivilegieerde instructies toegestaan *(exception bij poging)*
  - Beveiliging tegen misbruik
  - Beperkt: Directe toegang tot apparaten, hele schijf wissen, willekeurig geheugengebruik, …

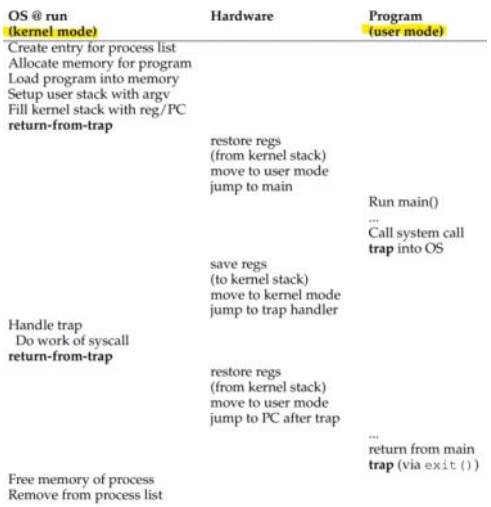
- Kernel Mode
  - Alles mogelijk

Wil proces een geprivilegieerde instructie uitvoeren?

- Moet aan OS vragen via system call!
- Overschakeling tussen modes
  - Trap: Vanuit usermode naar Kernelmode gaan
  - Return from trap

- Belangrijk om data bij te houden overheen trap/return from trap
  - User registers op Hardware Kernel Stack, Kernel registers via Software in geheugen…

- Hoe weet kernel welke instructie uit te voeren?
  - User mode kan geen adres geven om naar te springen in kernel (veiligheid!)
  - Trap Table bij boot van systeem
  - System Call nummer meegegeven op stack
  - Kernel voert in trap handler de system call uit



OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC **return-from-trap**		
	restore regs (from kernel stack) move to user mode jump to main	
		Run main() … Call system call **trap** into OS
	save regs (to kernel stack) move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	restore regs (from kernel stack) move to user mode jump to PC after trap	
		… return from main **trap** (via `exit()`)
Free memory of process Remove from process list		

Fases Limited Direct Execution

- Bij boot de trap table opstellen en CPU pointer onthouden
- Kernel tijdens uitvoering vóór return to trap

- Proces aanmaken op process list, geheugen alloceren, …

- Indien einde user proces (main)
  - Via EXIT() naar OS gaan

Andere Veiligheidsaspecten
- User Input controle
  - Write naar adres in kernel mag niet -> Moet gecontroleerd

Probleem 2: Behouden controle van CPU

Hoe behouden we controle over CPU als we processen er toegang tot geven?

Probleem
- Indien proces wordt uitgevoerd op CPU is OS niet in uitvoering
  - Kan niet tussenkomen en heeft geen controle

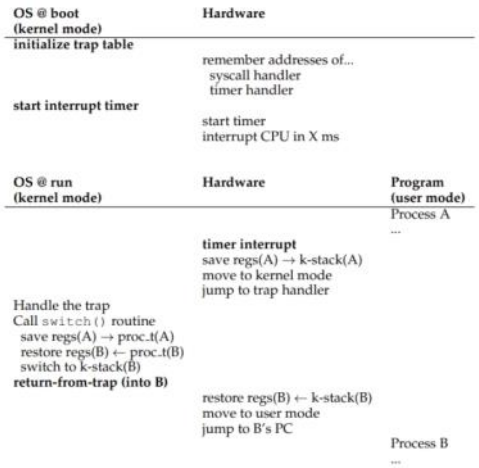
Oplossingen

Cooperative Approach
- Wachten op een system call vanuit programma
  - Bij illegale instructies (delen door 0, geen toegang)
  - In de code zelf ingesteld
    - System calls naar OS
    - Via YIELD() syscall, geeft controle terug aan OS maar doet verder niets

- Probleem
  - Wat als proces nooit system call uitvoert?
  - Wat met oneindige loops?

Non-Cooperative Approach
- OS neemt zelf controle over
- Onderbrekingsmechanisme
  - Timer Interrupt die system call afdwingt (Trap)
  - Interrupt Handler die OS runt
  - Scheduler bepaalt wat gebeurt: proces blijven uitvoeren/overschakelen naar ander proces

- OS kan blijven beslissen om proces te onderbreken en heeft controle over de CPU



OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember addresses of... syscall handler timer handler	
start interrupt timer		
	start timer interrupt CPU in X ms	
OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A ...
	timer interrupt save regs(A) → k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call `switch()` routine save regs(A) → proc_t(A) restore regs(B) ← proc_t(B) switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) ← k-stack(B) move to user mode jump to B's PC	
		Process B ...

Context Switch

- Belangrijk om info bij te houden bij overgang van user naar kernel (en omgekeerd)
- Registers user programma opslaan op kernel stack
- Registers kernel opslaan in struct proc_t
- Geïmplementeerd met assembly code

- Twee mogelijkheden
    - Timer Interrupt: Registers impliciet bijgehouden door hardware

    - Switch van processen: Registers expliciet bijgehouden door software (op de stack)

```
        # Context switch
        #
        #   void swtch(struct context *old, struct context *new);
        #
        # Save current registers in old. Load from new.

.globl swtch
swtch:
        sd ra, 0(a0)
        sd sp, 8(a0)
        sd s0, 16(a0)
        sd s1, 24(a0)
        sd s2, 32(a0)
        sd s3, 40(a0)
        sd s4, 48(a0)
        sd s5, 56(a0)
        sd s6, 64(a0)
        sd s7, 72(a0)
        sd s8, 80(a0)
        sd s9, 88(a0)
        sd s10, 96(a0)
        sd s11, 104(a0)

        ld ra, 0(a1)
        ld sp, 8(a1)
        ld s0, 16(a1)
        ld s1, 24(a1)
        ld s2, 32(a1)
        ld s3, 40(a1)
        ld s4, 48(a1)
        ld s5, 56(a1)
        ld s6, 64(a1)
        ld s7, 72(a1)
        ld s8, 80(a1)
        ld s9, 88(a1)
        ld s10, 96(a1)
        ld s11, 104(a1)

        ret
```

Meerdere Interrupts tegelijk?

- Wat als tijdens een interrupt een andere interrupt zich voordoet?

- Mogelijke Oplossingen
    - Interrupts uitschakelen tijdens interrupt
    - Werken met locking en gelijktijdigheid

Nagekeken en vervolledigd op 21/12/2023

7. Scheduling: Introduction

Hoe ontwikkelen van een planningsstrategie voor processen?

Scheduling Policies / Disciplines
- Strategie om processen te plannen en keuzes te maken
- Veel verschillende policies mogelijk

Jobs
- Processen die worden uitgevoerd op het systeem

Workload
- Collectie van jobs
- Scheduling Policy bepaalt hoe de jobs uit te voeren

Metric = Manier om dingen te vergelijken

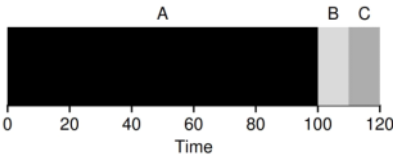
Wanneer is policy beter?
- Performantie: Zo snel mogelijk schakelen
- Eerlijkheid: Komt elke job aan bod?

Turnaround Time / Doorlooptijd
- Tijd nodig om job uit te voeren
- $T_{turnaround} = T_{completion} - T_{arrival}$

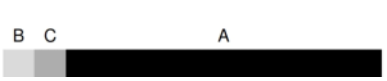
Response Time / Reactietijd
- Hoe lang duurt het vanaf de aankomst voordat de CPU de eerste keer het proces behandelt
- $T_{response} = T_{firstrun} - T_{arrival}$

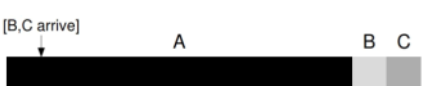
Keuze policy hangt af van verwachtingen
- Doorlooptijd
- Responstijd
- Eerlijkheid

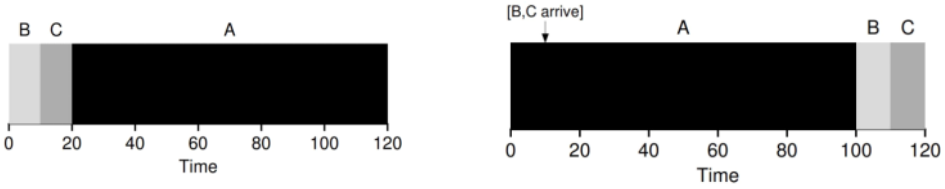
Scheduling Policies Turnaround Time
- FIFO: First in first out
  - Werkt goed bij gelijke jobs
  - Werkt niet goed bij langere en kortere jobs
  - Convoy Effect
    - Korte jobs komen in een wachtrij na lange jobs en de gemiddelde turnaround stijgt hard



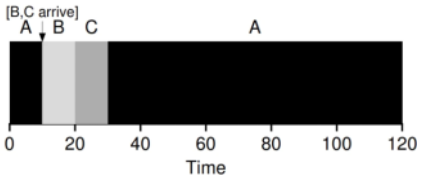
- SJF: Shortest Job First
  - Kortere jobs eerst behandelen
  - Werkt niet optimaal bij verschillende starttijden, als alle jobs op zelfde moment zouden komen wel!
  - Niet eerlijk, niet garantie dat elke job behandelt zal worden





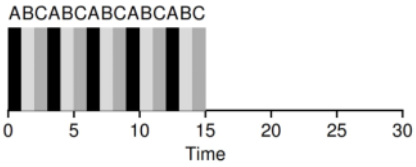


- STCF: Shortest Time to Complete First
  - Job behandelen die het eerst wordt afgerond
  - Preemetive Policy: We nemen aan dat we afwisselend kunnen uitvoeren en niet perse in 1 keer moeten doen
  - Optimaal als je kijkt naar CPU doorlooptijd (zonder I/O)



Scheduling Policies Response Time

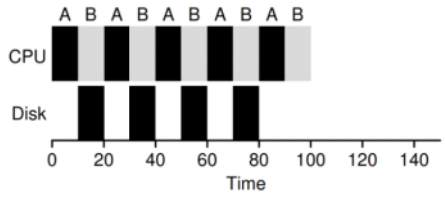
- Round Robin Scheduling RR
  - Schakelen tussen alle jobs op time slices
  - Kortere time slice
    - Beter voor tijd, te kort slecht door overheersende kost context switches
  - Langere time slice
    - Beter voor complexiteit, te lang slecht voor response time
  - Time Slice moet veelvoud zijn van interrupt timer
  - Slecht voor doorlooptijd metriek



Er is geen beste policy! Sommige goed voor bepaalde metriek en slecht voor andere

Toevoegen van I/O

- Wat als job moet wachten op I/O?
- Als job in blocked status, kan scheduler terwijl een andere job uitvoeren
- Als I/O klaar is wordt interrupt verzonden naar OS
- Overlap in processen die wachten op I/O en uitvoeren op CPU



Algemeen Probleem

- OS weet vaak niet hoelang een job zal duren
- Kan niet in de toekomst kijken
- Oplossing: Multi-level Feedback Queue

8. Multi-Level Feedback Queue

Hoe bepalen we goede response- en doorlooptijd indien we het gedrag van de job niet kunnen voorspellen?

Multi-Level Feedback Queue

- Goede Doorlooptijd: Hoe doen zonder lengte van jobs te kennen?
- Goede Responsetijd: Hoe doen zonder negatief effect op doorlooptijd?

MLFQ Werking

- Verschillende processen ingedeeld in rijen met prioriteiten
- Nieuwe jobs starten in hoogste prioriteit (gelijkaardig aan STCF)
- Jobs dalen in prioriteit indien langer duurt
- In gelijke prioriteit gebruik maken van Round Robin

MLFQ Basisregels

- Als $Prioriteit(A) > Prioriteit(B) \rightarrow$ A runt en B niet
- Als $Prioriteit(A) = Prioriteit(B) \rightarrow$ A & B runnen door Round Robin
- Een job die binnenkomt start op hoogste prioriteit
- Indien job gehele time slice gebruikt daalt het in prioriteit
- Als job CPU teruggeeft voor einde time slice blijft het in prioriteit

Problemen bij MLFQ?

- Starvation / Niet eerlijk
  - Langere processen zouden nooit uitgevoerd kunnen worden door korte jobs
  - Oplossing: Prioriteitsboost

- Gaming the scheduler
  - Valsspelen, programma schrijven zodat je telkens in hoogste prioriteit blijft
  - Oplossing: Prioriteitsdaling op basis van totale tijd

- Jobs kunnen veranderen
  - Indien job wel interactief wordt en op lage prioriteit staat verhoogt de prioriteit niet
  - Wordt anders behandelt dan andere interactieve jobs

Aangepaste Regels MLFQ

- Na een bepaalde tijd S krijgen alle jobs de hoogste prioriteit
  - S een voo-doo constante: te groot, lange jobs komen niet aan bod. Te klein, geen eerlijke verdeling

- Eens job totaal boven een tijdsverbruik zit, laten zakken in prioriteit
  - Vervanging van time slices --> Voorkomen van Gaming Scheduler

Instellen MLFQ

- Hoe bepaal je de parameters
  - Aantal queues, grootte van time slice, grootte van tijdssloten, …

- Veel variaties mogelijk

9. Proportional Share

Hoe kunnen we zo eerlijk mogelijk de CPU verdelen?

Proportional Share Scheduler

- Niet gebaseerd op doorloop- en responsetijd
- Zorgen dat elke job een bepaald **<u>percentage</u>** van de CPU tijd krijgt

Proportional Share Schedulers

Lottery Scheduling

Lottery Scheduling
- Rekening houden met gewicht, tickets toekennen aan jobs --> hoe meer tickets, hoe meer kans
- Random ticket kiezen, proces van gewonnen ticket krijgt CPU
- Elke time slice nieuwe loterij

Ticket Mechanisme
- Ticket Currency
  - Gebruiker met bepaalde hoeveelheid tickets kan deze verdelen onder eigen jobs

- Ticket Transfer
  - Proces zet tickets over naar ander proces

Implementatie Lottery Scheduling
- Eenvoudig, belangrijk een goede random factor voor loterij
- Gelinkte lijst met elke knoop een job en aantal tickets

- Hoe tickets toekennen aan jobs?
  - Aan user overlaten? Niet handig!
  - Onbeantwoorde vraag…

Stride Scheduling

Stride Scheduling
- Geen gebruik maken van random factor, deterministisch beslissen
- Processen hebben een stride en pass waarde
  - Bij uitvoering proces na elke time slice de pass-value verhogen met stride

- Telkens proces nemen dat de kleinste pass-waarde heeft --> Kleinste stride komt vaakste voor

Linux Complete Fair Scheduler

Linux CFS
- Round Robin Scheduler met toekenning van gewichten en dynamische time slices
- Bepaald time slice op basis van parameters $sched\_latency$ en $\min\_granularity$
  - $Time\ slice = \frac{sched\_latency}{\#\ runable\ processes}$ $en\ minstens\ \min\_granularity$

- Kan omgaan met meerdere CPU's

Er bestaat geen 'beste' scheduler!

Nagekeken en vervolledigd op 22/12/2023

Les 4 - Address Spacing

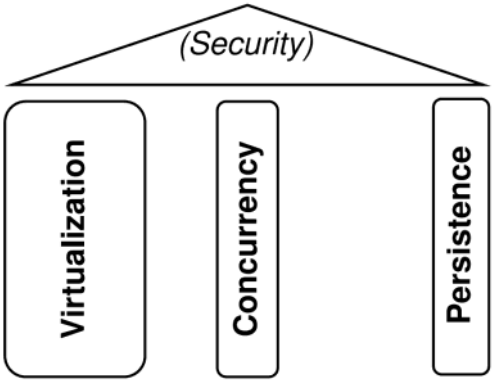
Liam Luys – r0934449 - 2e BA Informatica
Besturingssystemen

maandag 16 oktober 2023 8:32

The Big Picture

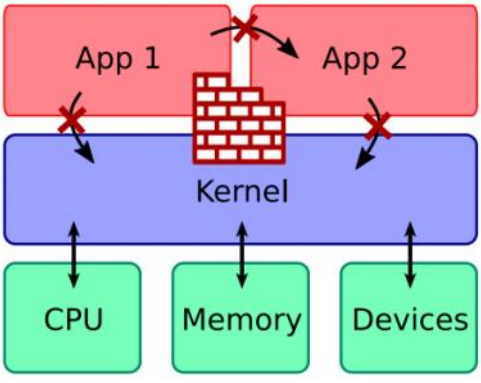
Drie Kernbegrippen Besturingssysteem
- Virtualisatie (CPU / Memory)
- Persistentie
- Concurrency

Overkoepelende Factoren
- Beveiliging
- Performantie/Snelheid
- …



Waarom geheugen Virtualiseren?
- Isolatie
  - Processen mogen het geheugen van elkaar of het OS niet kunnen betreden

- Transparancy
  - Programma kan 1 keer compileren en runnen op verschillende machines



(CC-BY-SA adapted from Wikimedia)

Policy/Mechanism
- Policy: Wat wil je bereiken?
  - Geheugenvirtualisatie, tijdsverdeling, …

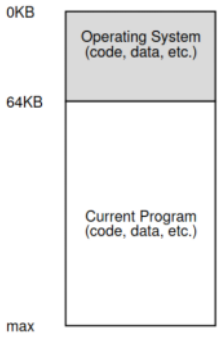
- Mechanism: Hoe ga je het bereiken?
  - Hardware, Limited Direct Execution LDE, segmentation/paging, …

Kunnen uitleggen voor examen, url in ppt optioneel

13. The Abstraction: Address Spaces

Early Days: Uniprogramming

Waar oorsprong van besturingssystemen?

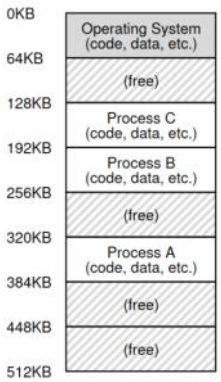


Eerste besturingssysteem
- Werd gezien als library
- Enkel één proces per keer
- Geen isolatie proces/OS
- Geen abstracties: al het geheugen gebruikt

- Vb: MS DOS, …

Multiprogramming & Time Sharing
- Wegens dure apparaten meerdere mensen/processen op 1 computer
- Meerdere processen waartussen de OS switcht (multiprogramming)

Problemen
- Mogelijkheid voor meer dan 1 applicatie te runnen
- Beveiliging & Isolatie
- Abstractieprobleem



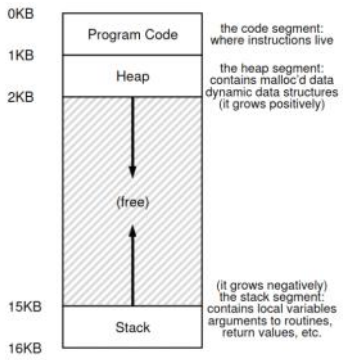
Uitdaging
- Geheugen bij elke interrupt/context switch opslaan en herstellen (zoals CPU registers)
  - Waar bewaar je? Harde schijf?
  - Lastig en duur!

- Oplossing
  - Geheugen verdelen over meerdere processen die de CPU delen
  - Hoe beveiligen? Programma's mogen niet aan elkaar/OS geheugen

The Address Space Abstraction

Hoe kan je illusie opzetten zodat programma denkt het enige te zijn, maar in realiteit het geheugen verdelen over meerdere processen?

Address Space
- Per proces bijhouden van gegevens
  - Data Proces
  - Code Proces
  - Stack & Heap



Virtualisatie Geheugen
- Per proces het geheugen voorstellen als groot en startend vanaf 0
- Abstractie van fysieke toestand naar Virtuele toestand (Address Translation)

Waarom nooit programma op fysiek adres 0 starten?
- Null-Pointer bugs in programma's
- Het adres 0 aanspreken door een fout

Goals bij Address Spaces

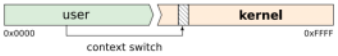
- Transparantie: Programma zelf denkt dat deze gewoon al het geheugen heeft
- Efficiëntie: Tijd- als ruimte-efficiëntie zo goed mogelijk maken
- Beveiliging: Isolatie tussen processen/OS bewaren

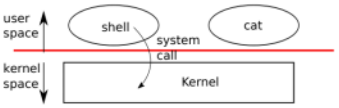
Kernel Address Space Protection

Hoe beschermen we besturingssysteem van andere programma's?

Mogelijkheden

- User vs Kernel Mode
- Kernel mapped als inaccessible in address space
- Controlled Kernel invocation via system calls and traps



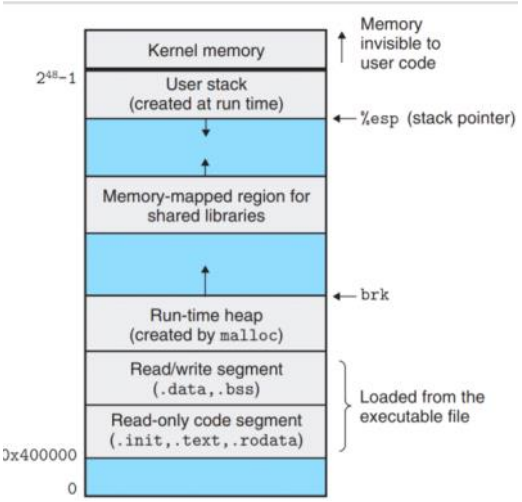


User / Kernel Mode

- Computer voert een gebruikersprogramma of een kernel uit
- Hardware support: een bit in geheugen geeft aan in welke mode je zit
- In gebruikersmode kan je niet aan de kernel
- Controle via system calls en traps

14. Memory API

Hoe geheugen alloceren en beheren in een besturingssysteem?



Types geheugen bij uitvoering

- Stack
- Heap

Stack

- Lokale variabelen
- Automatisch Geheugen
  - Compiler alloceert en geeft automatisch geheugen van de stack vrij

Heap

- Expliciete allocaties en vrijgeven door library calls in de gebruikerscode
  - $malloc(size)$
    - Geeft pointer naar een block op de heap
  - $free(pointer)$
    - Ongebruikt geheugen vrijmaken, voorkomen van memory leak

Support van Besturingssysteem

- $(s)brk() \rightarrow$ heap vergroten/verkleinen
  - Wordt gebruikt door $malloc()$ in library
- $mmap() \rightarrow$ Maakt anonieme geheugenregio binnen programma
  - Regio als swap space
- **Heap Memory API stelt contract voor tussen user en libc!**

Pitfalls / Belangrijke Punten van Heap API

- Niet vergeten heap-geheugen te alloceren
- Genoeg ruimte alloceren
- Memory leaks vermijden
- Nooit use-after-free
- Nooit meerdere keren free na elkaar
- Enkel geldige malloc-pointers oproepen bij free

15. Mechanism: Address Translation

Hoe efficiënt aan geheugenvirtualisatie doen en controle behouden over geheugen?

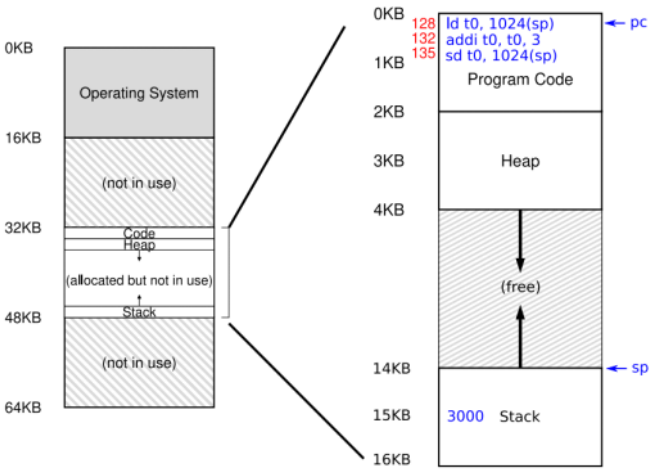
Limited Direct Excution

- We willen processen direct op hardware laten uitvoeren, maar zitten met virtuele adressen tov fysische adressen
- Hardware-Based Address Translation
    - Elk virtueel adres omzetten in fysiek adres waar het wordt opgeslagen

- OS moet geheugen beheren
    - Bijhouden welk geheugen vrij/bezet is
    - Bijhouden hoe geheugen gebruikt wordt

Assumpties Address Space

- Aaneensluitend in fysieke geheugen *(Paging)*
- Niet enorm groot, kleiner dan fysiek geheugen *(Swapping)*
- Elke address space even groot *(Segmentation)*



Dynamic Relocation

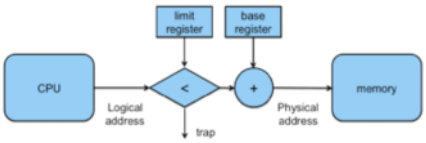
Hoe virtuele adressen van 0 starten en ergens opslaan in fysieke geheugen?

Memory Management Unit MMU

- Twee hardware registers per CPU
    - Base
    - Bounds / Limit

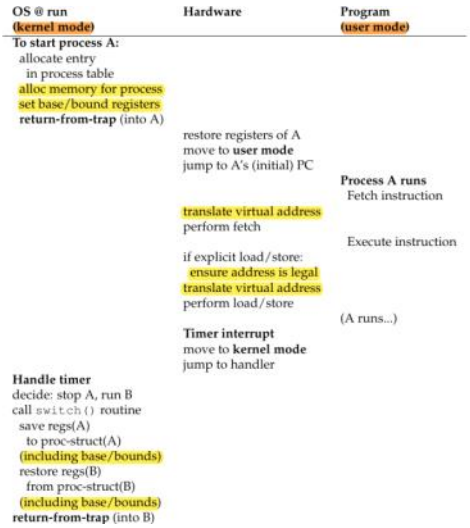
- Translatie door bij elk virtueel adres de base op te tellen
    - $Fysical\ address = Virtual\ address + base$

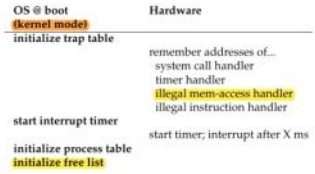
- Bij elke oproep nakijken of address binnen [0, bounds] ligt
    - Trap indien fout



Hardware Support

- 2 CPU Modes
- Processor Status Word
- Memory Management Unit
    - Base & Bounds
    - Exceptions / Exception Handler





16. Segmentation

Hoe een grote address space kunnen ondersteunen? Hoe geen fysiek geheugen verspillen?

Segmentation

- Address Space opdelen in logische segmenten *(vb code, stack, heap)*
- Base Bounds paar bijhouden per segment
- OS kan verschillende segmenten op andere plaatsen in fysiek geheugen plaatsen

Segmentation Fault

- Door memory acces op illegal address
- Ook gebruikt indien geen segmentation aanwezig is op machine

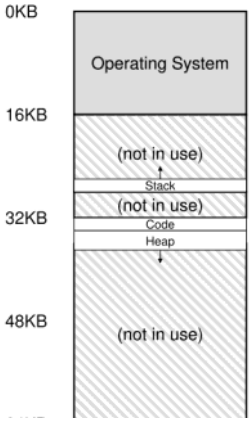
Interne Fragmentatie

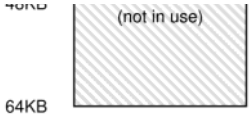
Interne fragmentatie

- Hoe geen fysiek geheugen verspillen door sparse virtuele adresssen?
- Proces vraagt geheugen maar heeft niet (alles) nodig

Oplossing

- Per segment enkele registers bijhouden
- Geen verspilling van ongebruikt geheugen
- Mogelijkheid toevoeging permissies en delen (enkel bekijken, ook schrijven, …)
- OS kan segmenten verplaatsen in het fysieke geheugen

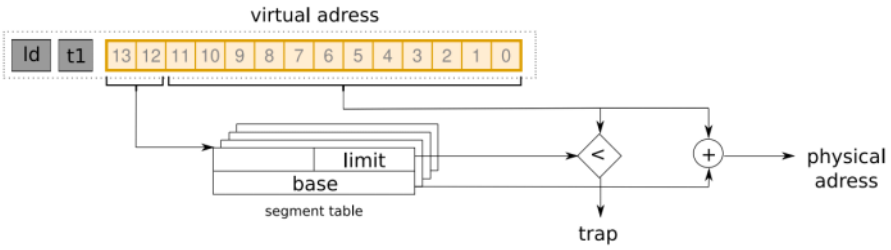




Naar welk segment verwijst adres?
- Hoe weet hardware de offset in segment en naar welk segment een adres verwijst?
- Expliciet Bijhouden
    - Structuur van virtueel adres aanpassen en die info meegeven via bits

- Impliciet Bijhouden
    - Kijken naar structuur adres om segment te bepalen

Problemen Expliciete structuur
- Door bitsvoorstelling van segmenten kan je ongebruikte segmenten hebben
- De bits nemen plaats in van virtueel adres, dus je beperkt adresruimte



Support Sharing
- Om geheugen te besparen bestanden delen
- Protection Bits (hardware support)

Segmentation Types
- Coarse-Grained
    - Grote segmenten binnen virtuele address space

- Fine-Grained
    - Meerdere kleinere segmenten

Segment Table
- Verzameling van base-bounds van segmenten in een tabel

OS Support voor Segmentation
- Segment registers bijhouden en herstellen bij context switch
- Wanneer segment wijzigt van grootte (malloc/free)
- Vrije ruimte beheren

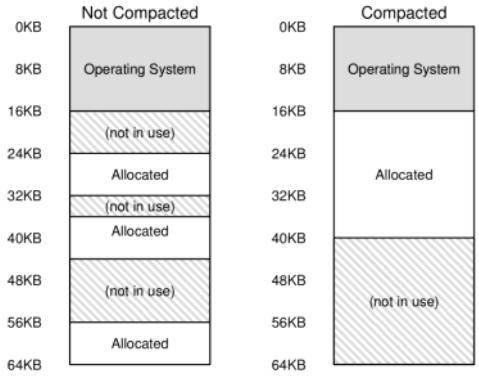
Externe Fragmentatie

Externe Fragmentatie
- Fysieke geheugen snel vol door vele kleinere ongebruikte geheugenstukken
- Nieuwe segmenten kunnen niet meer in fysieke geheugen

Oplossing
- Compact fysiek geheugen door herschikking / de-fragmentatie
    - Dure operatie

- Freelist algoritme gebruiken
    - Probeert grote stukken vrij te houden voor allocatie
    - Veel algoritmes mogelijk
        - Best-fit, worst-fit, first-fit, buddy algorithm, …

- Zal nooit volledig verdwijnen, blijft een probleem



Algemene problemen Segmentation

- Interne/Externe fragmentatie
- Segmentation niet flexibel genoeg voor sparse address space
- Als model van gebruik van address space niet exact overeenkomt met de onderliggende segmentatie werkt de segmentatie niet volledig

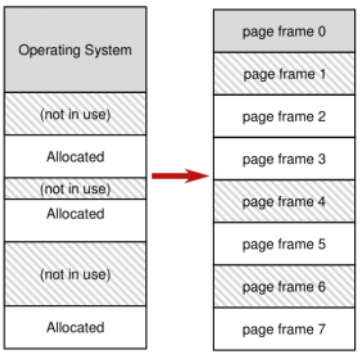
Nagekeken en vervolledigd op 23/12/2023

18. Paging - Introduction

Fragmentatie
- Intern: Ongebruikt geheugen in gealloceerde ruimte
- Extern: Variabele stukken ongebruikt geheugen
    - Zorgt voor kleine delen geheugen die niet-aangesloten zijn en niet in gebruik zijn

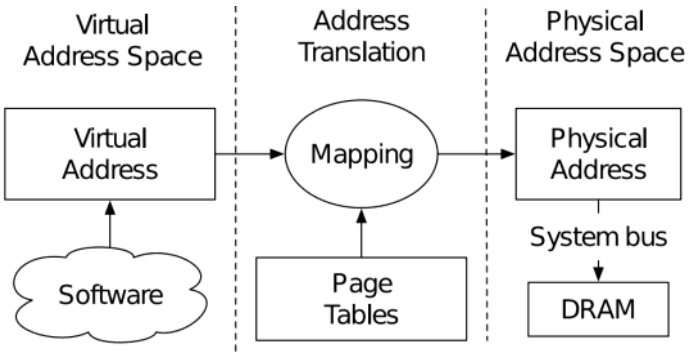
Paging Mechanisme
- Fysiek geheugen opdelen in page frames
- Virtueel geheugen opdelen in pages
- Page Table met mapping/translatie van pages naar frames
    - Bijgehouden in geheugen, ingesteld door OS

- Applicatie krijgt bepaalde hoeveelheid page frames
- Geen externe fragmentatie meer mogelijk

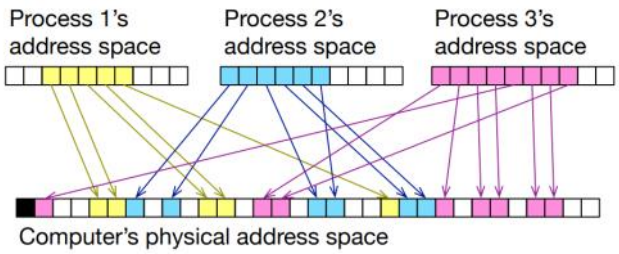


Memory Virtualisation: Page Tables

Hoe op basis van pages toch nog idee creëren dat applicatie een aaneengesloten geheugen krijgt?



Virtuele Adressen vs Fysieke Adressen
- Proces krijgt virtueel geheugen dat wordt gemapped naar (een deel van) het fysieke geheugen (DRAM)



- Grootte van virtuele pagina en fysieke pagina moet gelijk zijn! Anders is mapping niet mogelijk

Onderdelen Virtueel Adres
- VPN: Virtual Page Number
- Offset: adres binnen page frame

Page Table
- Bestaat uit Page Table Entries PTE
    - Koppeling PPN en VPN

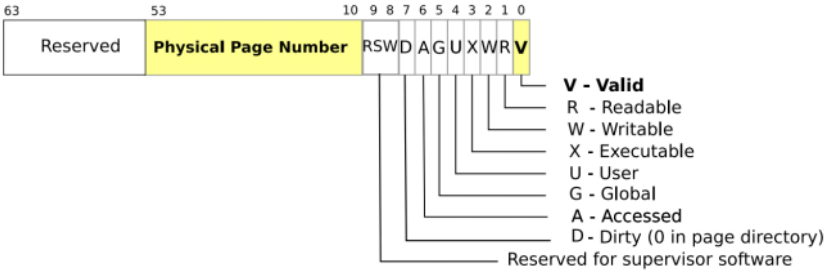
- PPN: Physical Page Number = PFN: Physical Frame Number

- Zijn vaak zeer groot
  - Voor elk adres een PTE
  - Opgeslagen ergens in geheugen

Oplossing
- Page Table in kernel geheugen zetten
- Pointer PTBR (Page Table Base Register) naar start van table in fysiek geheugen

Opbouw Page Table

Opbouw PTE



Opbouw PTE
- Valid Bit
  - Geeft aan of translatie geldig is
  - Nodig voor sparse geheugen en ongebruikte tussenruimtes

- Protecion Bits
  - Read, Write, Execute

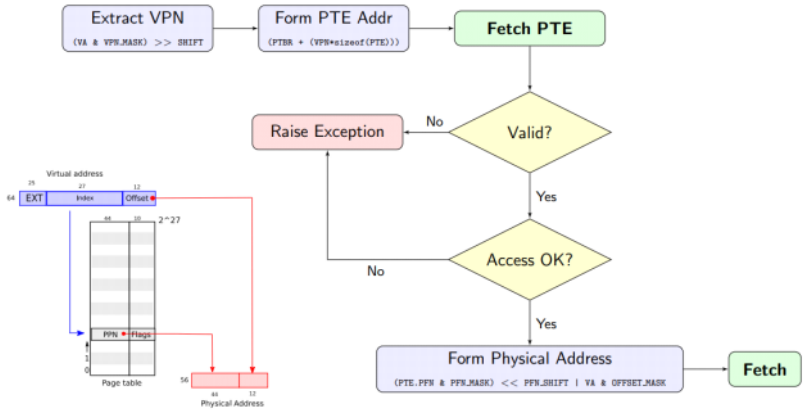
- Dirty Bit
  - Is page gewijzigd na brengen in geheugen?

- Reference / Accessed Bit
  - Is page al bekeken?

- User/Supervisor Mode
  - In welke toestand?

Linear Page Table Translation

Linear Page Table
- Eenvoudige voorstelling van Page Table dmv array
- Index de VPN
- Inhoud PTE array[VPN] geeft PFN



Locatie Page Table

- Ruimte
  - Linear Page Tables worden snel heel groot (te groot voor DRAM)
  - Opslaan in beveiligde kernel geheugen
  - Page-Table base register PTBR naar fysieke geheugen

- Sparse
  - Veel virtuele adressen zijn ongebruikt (linear pagetable niet performant)
  - Oplossing door betere structuur: Multi-Level Pagetable

Problemen als Page Table in geheugen zit

- Elke translatie heeft nu 2 geheugenaccessen nodig!
  - PTE ophalen in Page Table
  - Effectieve Adres uit geheugen halen

- Snelheid
  - Oplossen door caching (TLB)

- Ruimte
  - Oplossen door datastructuur (Multi-level Page Tables)

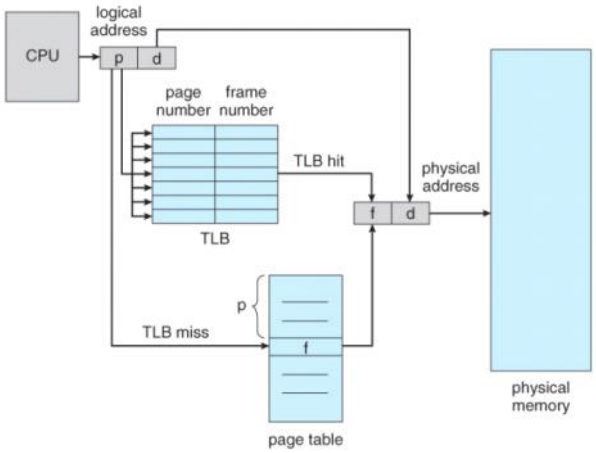
```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT

// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))

// Fetch the PTE
PTE = AccessMemory(PTEAddr)

// Check if process can access the page
if (PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False)
    RaiseException(PROTECTION_FAULT)
else
    // Access is OK: form physical address and fetch it
    offset = VirtualAddress & OFFSET_MASK
    PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
    Register = AccessMemory(PhysAddr)
```

19. Paging - Faster Translations (TLBs)

Hoe de address translation sneller maken en geheugentoegangen beperken?

Page Tables sneller maken

- Caching invoeren om geheugentoegangen te beperken
- Cache/Buffer onderdeel van MMU



Translation Lookaside Buffer TLB

- Omgeving Pages / Spatial Locality
  - Binnen page meestal meer dan 1 byte nodig

- Huidige Page / Temporal Locality
  - Huidige Page hoge kans om nogmaals gebruikt te worden

- Zeer snel maar klein

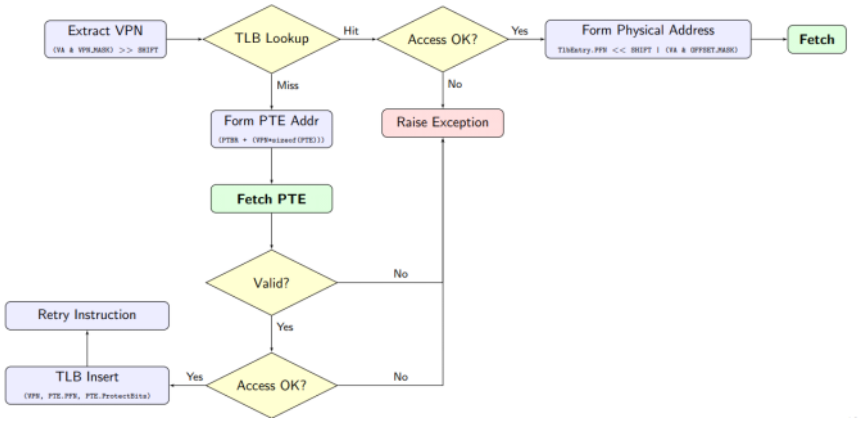
<u>Zonder TLB zou processor enkel aan de snelheid van geheugen kunnen werken.</u>

Opbouw (Hardware-Managed) TLB

- Fully-Associative
  - Translatie kan overal in de TLB zitten, hardware zoekt

- Bestaande uit VPN, PFN, Andere bits (valid, protections, dirty, …)

MMU Flowchart

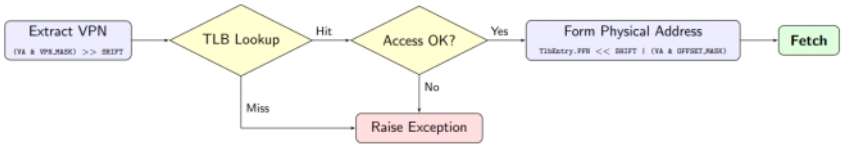


Context Switch en TLB

- TLB bevat proces-specifieke inhoud (PTEs specifiek per proces)
- Tijdens context switch moet dus gegarandeerd alles vervangen worden in TLB

Software-Managed TLB

- Een TLB miss overlaten aan OS
- Wordt niet meer gebruikt



TLB Invalidation

Hoe zorgen dat TLB altijd consistent is met de huidige proces Page Table?

Mogelijke Oplossingen

- Flush bij context switch
  - TLB volledig leegmaken, alle valid bits naar 0 zetten
  - Is duur bij vele context switches!

- Sharing TLB Table
  - Gelijke TLB houden over context switch
  - Toevoegen van **address space identifier ASID** veld
    - TLB entry koppelen aan address space (bepaald proces)

VPN	PFN	valid	prot	ASID
10	101	1	r-x	1
—	—	0	—	—
50	101	1	r-x	2
—	—	0	—	—

Replacement Policy

Vervangen van TLB inhoud

- Verschillende policies mogelijk
  - Least Recently Used LRU
  - Random

20. Paging - Smaller Tables

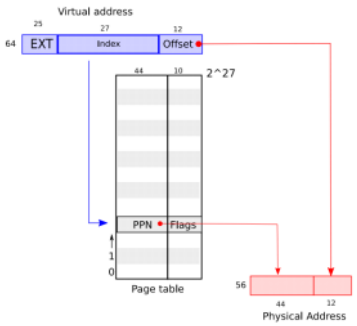
Hoe grote Page Table bijhouden in geheugen wetend dat er veel van die table niet wordt gebruikt (sparse)?

Opslaan van ongebruikte virtuele pages in fysieke frames is zinloos

- Hoe mechanisme opbouwen die dit kan beheren?

Idee 1: Grotere Pages

Voordeel: Minder Pages, minder ruimte nodig
Nadeel: Interne Fragmentatie, binnen één page wordt veel ruimte niet gebruikt



Idee 2: Hybride Segmentatie

(wordt niet gebruikt)

BEKIJK LESOPNAME!!!!

Hybrid Segmentation

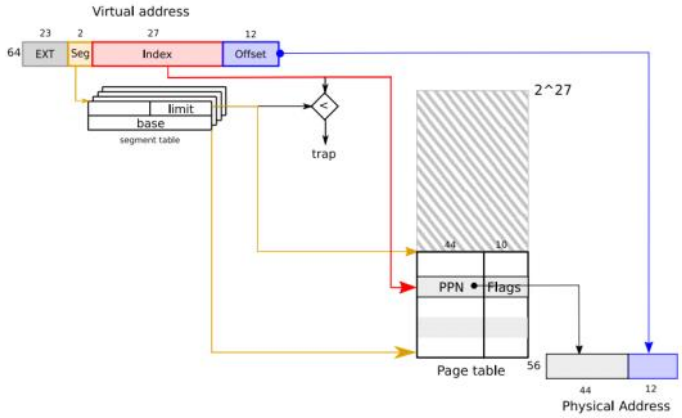
- Combinatie van pages en segmentatie
- Per segment een page-table
  - Bovenste virtueel adres stelt segment voor
  - Base/Limit beschrijven continue fysieke page-table memory

Zorgen dat niet-gebruikt fysiek geheugen niet moet worden vrijgehouden

Problemen

- Interne Fragmentatie
  - Page Table kan terug sparse zijn

- Externe Fragmentatie
  - Page Table heeft variabele grootte en moet aaneengesloten in geheugen zitten



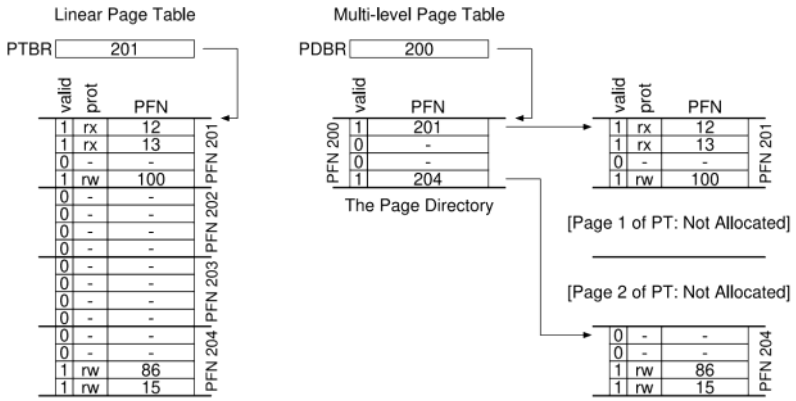
Idee 3: Paging op Page Table

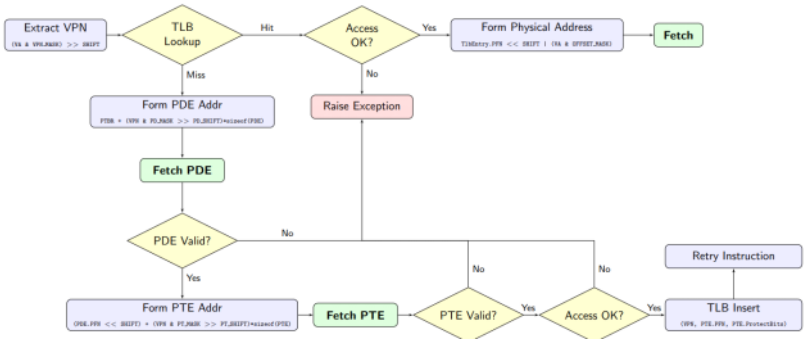
Multi-Level Page Tables

- Page Table verdelen in verschillende stukken/pages met vaste grootte
  - Geen externe fragmentatie

- Top-level Page Directory page bestaande uit onderdelen van originele page table
  - mapt second-level page-table pages met als bladeren PTEs
  - Wordt eerder zoekboom ipv tabel

Page Directory Entry PDE

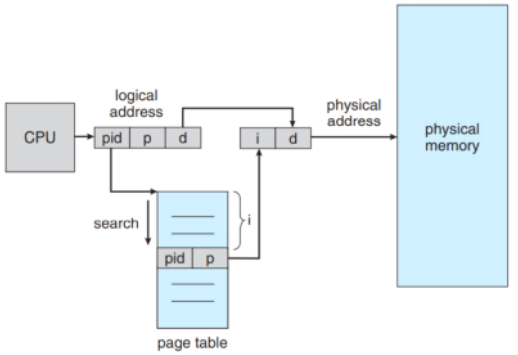




Andere Mogelijkheden

Inverted Page Table

- Ipv voor elk proces een page table, 1 page table met fysieke adressen en binnen entry welke virtuele adressen ernaar gemapped zijn
- Wordt zoekprobleem (lineaire tijd)
    - Is te duur, opgelost met hashtabel



Swapping to Disk / Demand Paging

- Zie volgend hoofdstuk

Samenvatting

EXT

- Bits worden niet meegenomen, genegeerd
- Willekeurige waardes of vaste waarde om aan te geven dat ze niet worden gebruikt

Page Table

- Per proces aangemaakt

- Zit in geheugen
    - Is vrij groot
    - Nadelig, dure operaties
    - Gebruiken van buffer --> TLB

Translation Lookaside Buffer

- Zorgt voor snelheid
- Moet samenwerken met OS zodat data in cache gelijk blijft aan effectieve data in geheugen

Multi Level Page Tables

- Zorgt voor geheugenbesparing
- Ongebruikte pages niet bijhouden
- Werken adhv zoekboom door verschillende lookup tables om mapping te vinden

21. Beyond Physical Memory - Mechanism

Hoe de illusie geven van onbeperkt geheugen met een klein fysiek geheugen en gebruik van grotere trage apparaten?

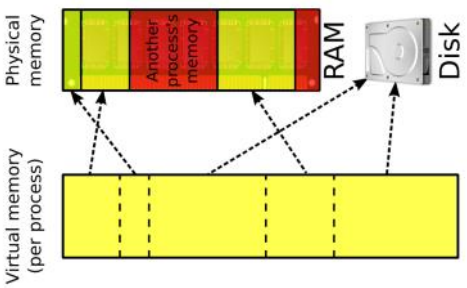
Harde schijf

- Trager dan geheugen
- Veel groter

Swapping

- Ongebruikte pages opslaan op harde schijf ipv op DRAM
  - Swap space gereserveerd op schijf
  - OS moet adres van pages in swap space bijhouden

- Proces mag niet zelf rechtstreeks naar harde schijf
  - Hardware genereert pagefault als virtual page niet in DRAM zit
  - OS Fault handler moet oplossen en page in DRAM zetten



Present Bit

- Extra hardware-mechanisme om swapping mogelijk te maken
- Onderdeel van PTE
- Indien actief staat de page in het geheugen, indien niet staat deze op de disk/schijf

- Indien poging om page te accessen die niet in geheugen zit komt een **pagefault**

Wat bij TLB Miss?

- Staat present bit op 0?
  - Zoek naar page in geheugen
  - Zet page in TLB
  - Herbegin en zoek terug in TLB

- Staat present bit op 1?
  - Adres in PTE stelt plaats voor op disk/schijf
  - Meld page-fault aan OS
  - Zoek bestand op de schijf en breng over naar geheugen
  - Zet present bit op 0
  - Herbegin zoeken: kijk in TLB, indien niet in TLB eerst overbrengen naar TLB, enz…

Page Fault Exception

Hoe geeft programma fouten weer?

Programmmafout / Systeemfout

- OS neemt over en handelt deze fout af
- Page-fault handler

Mechanism

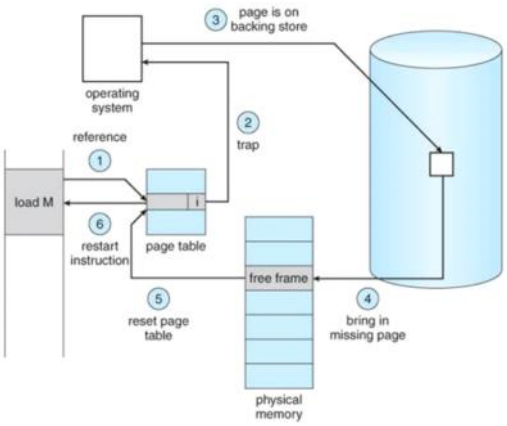
- MMU hardware triggert exception/trap van PTE als
  - Valid Bit V=0
  - Present Bit P=0
  - Protection bits die toegang afwijzen

Policy

- OS Page-fault handler beslist wat te doen
  - Scause/stval registers
  - Illegal access *(Proces heeft geen toegang)*
    - SIGSEGV signaal sturen naar proces
  - Legal access *(Proces heeft toegang, maar page zit niet in DRAM)*
    - Gevraagde page overbrengen van disk naar geheugen en proces verderzetten

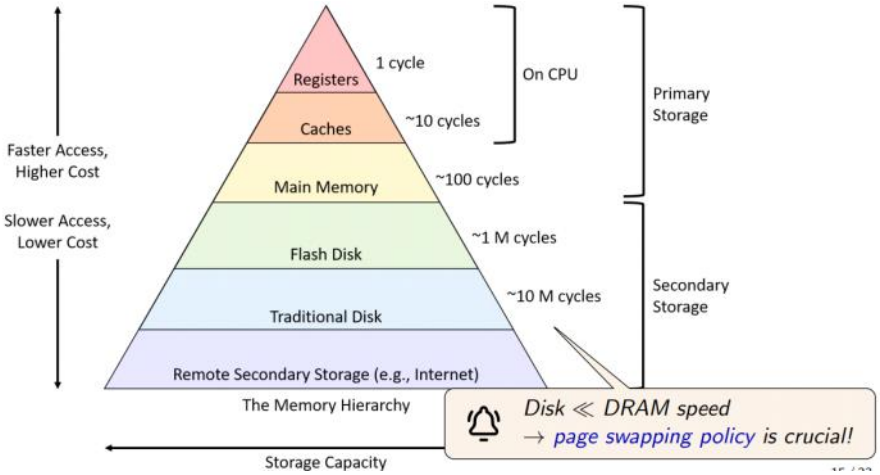
Soorten Onderbrekingen Proces

- Interrupt
  - Wacht tot huidige instructie is afgerond en gaat dan naar OS, gaat verder bij volgende instructie
- Exception
  - Stopt direct de uitvoering en gaat naar OS, herstart erna de uitvoering



Memory Pressure

Hoe omgaan met te weinig geheugen en swap space? Wat is de beste replacement policy?



Probleem

- Wat als geheugen vol zit en een swap page moet worden opgehaald?
- Plaats maken (page out) door page te vervangen
    - Page-Replacement Policy

22. Beyond Physical Memory: Policies

Hoe OS laten beslissen welke page(s) vervangen uit het geheugen?

Replacement Policies

Cache Management

- Systeem met swapping kan je zien als een cache van pages van de disk
- Average Memory Access Time AMAT
    - $AMAT = T_M + \left(P_{Miss} \times T_D\right)$ $met\ T_M\ kost\ van\ memoryaccess, T_D\ kost\ van\ diskaccess$
- Hoe minder cache misses, hoe beter!

Optimale Policy

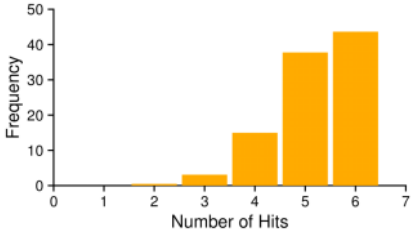
- Page eruit halen die langst niet gebruikt gaat worden
- In toekomst kijken kan niet maar zou optimaal zijn
- Hit Rate zo hoog mogelijk
    - $Hit\ Rate = \frac{Hits}{Hits + Misses}$

Queue/FIFO Policy

- Wachtrij maken van pages
- Zeer eenvoudig maar niet effectief
- Gevaar van belangrijke page uit te halen
  Hit Rate 36.4%

Random Policy

- Alles hangt van random keuze af
- Gevaar van belangrijke page uit te halen



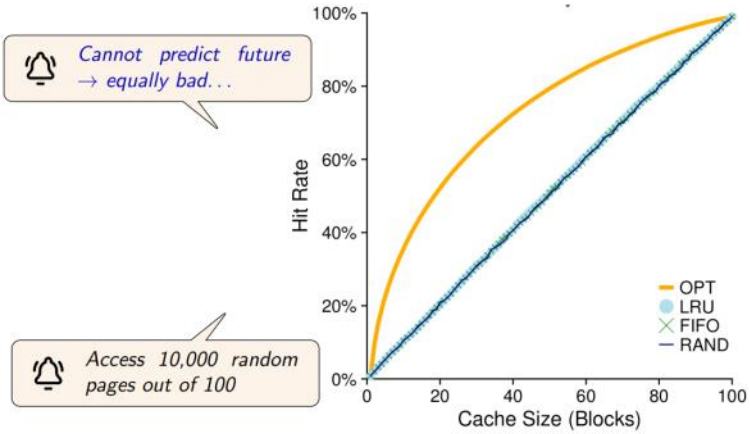
LRU Policy

- Least Recently Used, vervang pagina die het langst niet werd gebruikt
- Is realistisch vaak het beste algoritme

- Werkt goed voor locality
- Werkt slecht bij grote loop van hergebruikte pagina's
  Hit Rate 54.5%

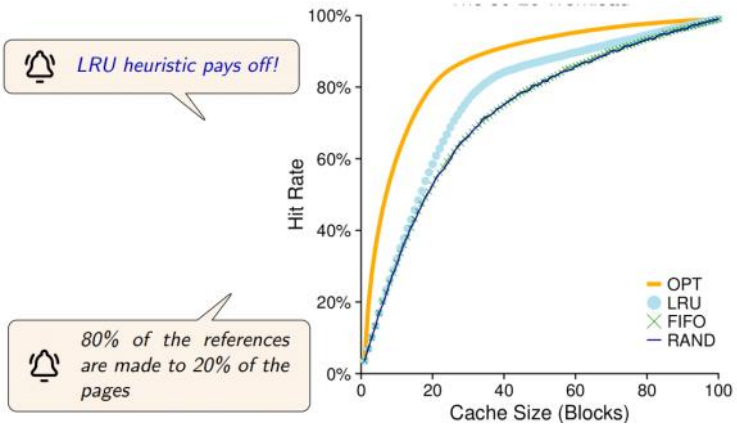
Geen Workload

Geen locality, random keuze van page



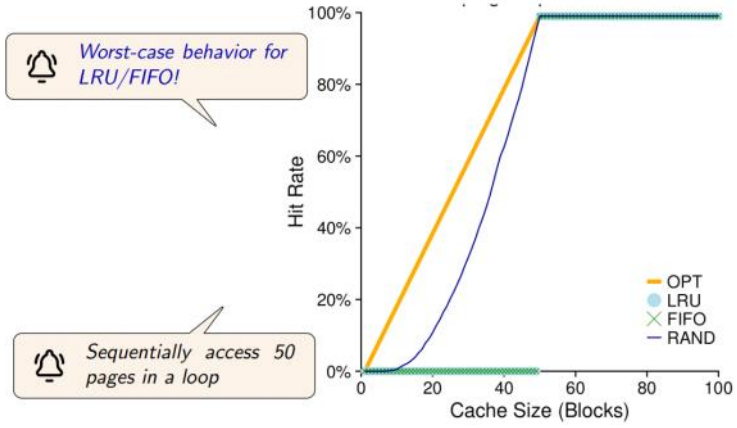
80-20 Workload

Wel locality: 80% van de keuzes worden gemaakt uit 20% van de pages



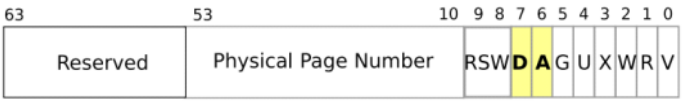
Looping-Sequential Workload

Telkens dezelfde pages van 1 tot 50 accessen



Replacements in realiteit

Hoe de LRU replacement policy gebruiken in realiteit?

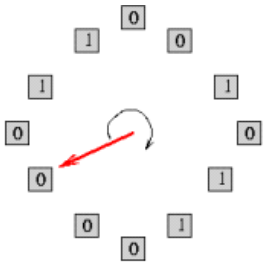


Bits binnen Page Table Entry

- D: Dirty
- A: Accessed
- G: Global -> Gebruikt voor kernel pages
- U: User Space
- X: Executeable / W: Writable / R: Readable
- V: Valid

Gebruik voor replacement

- Bij read/write zet hardware de accessed-bit op 1
    - Enkel OS kan deze terug op 0 zetten
- Accessed-bit gebruiken om LRU te implementeren



Clock Algorithm

- OS kijkt in cirkel of accessed bit op 1 of 0 staat
- Indien op 0 deze page vervangen, indien op 1 kijken naar de volgende
- Uitgebreid met kijken naar modified-bit
    - Indien aangepast zou de aanpassing moeten weggeschreven worden naar disk -> Dure operatie

Wanneer best pages vanuit harde schijf in geheugen brengen?

Policies

- Demand Paging
    - Enkel overzetten na een page fault
    - mmap geheugen
- Lazy Allocation
    - Wacht tot het gebruik van de page
    - sbrk heap
- Copy on Write
    - Identieke fysieke pages delen als read-only (bij parent en child na fork)
    - fork

Trashing

Wat als geheugen vol zit en er meer geheugen wordt gevraagd dan beschikbaar?

Trashing

- Systeem constant aan paging doen
- Admission Control
- Out-of-Memory Killer

Nagekeken en vervolledigd op 24/12/2023

II - CONCURRENCY

26. Concurrency: An Introduction

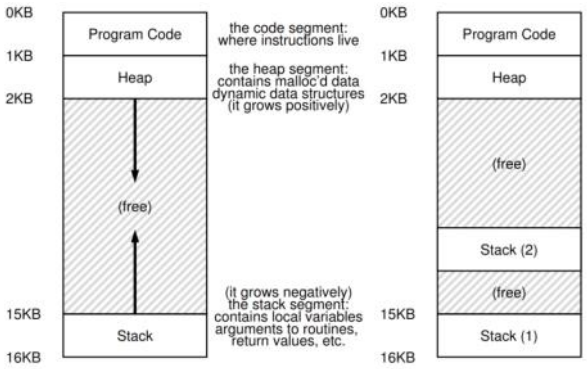
Concurrency / Gelijktijdigheid
- Gelijktijdig uitvoeren van meerdere instructiestromen/processen
- Meerdere fysieke CPU's

Virtualisatie
- Geen echte gelijktijdigheid, nog steeds afwisseling
- Meerdere virtuele CPU's op éénzelfde fysische CPU

Threads / Uitvoeringsdraad
- Abstractie van een enkele sequentiële control flow binnen programma
- Kan je voorstellen als combinatie van CPU en stack
- Elke functie start met 1 main-thread, elke thread eindigt als zijn functie returnt

Multiple Threads
- Meerdere programma's worden tegelijk uitgevoerd
- Elke thread een CPU en stack maar dezelfde address space
- Hoe meer threads, hoe meer overhead door contextswitch
- Als je meer threads maakt dan fysische CPU's gaat je snelheid niet meer verbeteren

Multiple Threads op 1 CPU
- Context switch nodig, gelijkaardig aan processen
- Process Control Block PCB slaat data/registers processen op
- Thread Control Block TCB slaat data/registers threads op
- Page Table blijft gelijk bij context switch threads



Waarom Multi-threading?
- Performantie verbeteren
  - Parallel uitvoering van threads op meerdere fysische CPU's
  - Programma's sneller uitvoeren
    - I/O overlappen met berekeningen binnen 1 proces
    - Intensieve berekening verdelen over meerdere CPU's: **Parallelization**
- Nadelen
  - Moeilijk om te begrijpen hoe het exact werkt

        - Scheduler beslist welke thread eerst wordt uitgevoerd, maar geen garanties over

    - In welke vorm is de gelijktijdigheid?
    - Is veel moeilijker om te programmeren

Probleem bij gelijktijdigheid

- Wat als meerdere threads werken met dezelfde gegevens en elkaar hierdoor negatief beïnvloeden?
- Interleavings
    - Output van programma's arbitrair interleaven?
    - Instructies van 2 threads arbitrair interleaven?
    - Machine-instructies arbitrair interleaven?

- Vb uit les: getal lezen, 1 optellen bij getal en terug in geheugen plaatsen. Bij 1000en herhalingen krijg je niet het verwachte resultaat

OS	Thread 1	Thread 2	(after instruction) PC	eax	counter
	before critical section		100	0	50
	`mov 8049a1c,%eax`		105	**50**	50
	`add $0x1,%eax`		108	**51**	50
interrupt					
save T1					
restore T2			100	0	50
		`mov 8049a1c,%eax`	105	**50**	50
		`add $0x1,%eax`	108	**51**	50
		`mov %eax,8049a1c`	113	51	**51**
interrupt					
save T2					
restore T1			108	51	51
	`mov %eax,8049a1c`		113	51	**51**

Race Condition / Data Race

- Resultaat hangt af van timing en uitvoering van de code
- Niet deterministisch
- Meerdere threads willen met dezelfde structuren iets doen (race)

Critical Section

- Stuk code dat een gedeelde variabele gebruikt en niet door meer dan 1 thread tegelijk mag uitgevoerd worden
- Wil mutual exclusion

Oplossingen Voor Probleem?

- Instructie die in 1 keer de taak uitvoert van de kritische sectie
    - Verwijderd mogelijkheid van data race
    - Uitvoering atomically, kan niet onderbroken worden tijdens de instructie

- Synchronisatie Primitieven
    - Toegangen van kritische secties controleren en synchroniseren

27. Thread API

Hoe kunnen we threads aanmaken en beheren via OS?

API zelf niet vanbuiten kennen, wel de concepten volledig begrijpen en code kunnen lezen.
Herkennen van API methode en weten wat die doet. Als je code krijgt weten wat die doet.

Thread Creation

Nieuwe thread aanmaken, attributen meegeven en startfunctie met argumenten meegeven

```
#include <pthread.h>
int
pthread_create(pthread_t *thread,
               const pthread_attr_t *attr,
               void *(*start_routine)(void*),
               void *arg);
```

Thread Completion

Wachten tot de thread klaar is met de uitvoering

```
int pthread_join(pthread_t thread,
                 void **value_ptr);
```

Opletten bij threads

- Nooit pointer van variabele teruggeven die op thread-stack werd aangemaakt

Locks

Hoe mutual exclusion garanderen voor kritische secties?

Communicatie tussen Threads

- Via globale variabele, alle threads hebben toegang tot globale variabelen van hoofdproces
- Threads delen dezelfde address space

Data Race

- Meerdere threads werken met dezelfde variabelen, beide gebruiken ze en minstens 1 van de twee schrijft ernaar
    - Geeft sowieso problemen --> Je kan volgorde van threads niet garanderen, wisselende uitvoering

- Oplossing: Locks

Lock Methode

- Je krijgt garantie dat er nooit overlapping zal zijn, altijd volledige uitvoering van de code tussen lock en unlock
- In beide threads hetzelfde lock-object gebruiken
- Sluit interleaving uit

- Verschillende lock-objecten kunnen wel overlappen!

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Problemen Locks

- Gebrek aan duidelijke initialisatie
    - Opgelost door toevoegen van API mogelijkheden

- Gebrek aan nakijken van error-codes bij oproepen van (un)lock
    - Wrappers gebruiken die checken op errors of foutcodes
    - $Pthread\ vs\ pthread$

Condition Variabele

- Lock moet steeds opstaan! (vermijden race condition)
- 2 Routines
    - Wait
    - Signal

- Rij van meerdere threads, thread kan zichzelf erin zetten met wait(), andere thread kan thread eruit halen/wakker

maken met signal()

Spinlock

- Baseren op globale variabele en in een while lus blijven wachten tot die aangepast wordt door andere thread
- NOOIT doen!
    - CPU verspilling door spin
    - Moeilijke manier om synchronisatie te bekomen

Voorbeeld Slecht --> Goed Programma

Worker thread example (take 1)

```
job_t* task = NULL;

void* worker(void* dummy) {
    job_t* current = NULL;
    while (1) {
        if (task == NULL)
            continue;
        current = task;
        task = NULL;
        compute(current);
    }
    return NULL;
}
void compute_par(job_t j) {
    task = malloc(sizeof(job_t));
    *task = j;
}
int main(int argc, char const *argv[]) {
    ...
    pthread_t w;
    pthread_create(&w,NULL,&worker,NULL);
    ...
    while (1) {
        // read job j from input
        compute_par(j);
    }
}
```

Worker thread example (take 2, mutexes)

```
job_t* task = NULL;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void* worker(void* dummy) {
    job_t* current = NULL;
    while (1) {
        pthread_mutex_lock(&m);
        if (task == NULL) {
            pthread_mutex_unlock(&m);
            continue;
        };
        current = task;
        task = NULL;
        pthread_mutex_unlock(&m);
        compute(current);
    }
    return NULL;
}
void compute_par(job_t j) {
    pthread_mutex_lock(&m);
    task = malloc(sizeof(job_t));
    *task = j;
    pthread_mutex_unlock(&m);
}
```

Worker thread example (take 3, condition var)

```
job_t* task = NULL;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void* worker(void* dummy) {
    job_t* current = NULL;
    while (1) {
        pthread_mutex_lock(&m);
        while (task == NULL) pthread_cond_wait(&c,&m);
        current = task;
        task = NULL;
        pthread_mutex_unlock(&m);
        compute(current);
    }
    return NULL;
}
void compute_par(job_t j) {
    pthread_mutex_lock(&m);
    task = malloc(sizeof(job_t));
    *task = j;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
}
```

Worker thread example (final take)

```
job_t* task = NULL;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void* worker(void* dummy) {
    job_t* current = NULL;
    while (1) {
        pthread_mutex_lock(&m);
        while (task == NULL) pthread_cond_wait(&c,&m);
        current = task;
        task = NULL;
        pthread_cond_signal(&c);
        pthread_mutex_unlock(&m);
        compute(current);
    }
    return NULL;
}
void compute_par(job_t j) {
    pthread_mutex_lock(&m);
    while (task != NULL) pthread_cond_wait(&c,&m);
    task = malloc(sizeof(job_t));
    *task = j;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
}
```

Nagekeken en vervolledigd op 25/12/2023

Les 8 - Locks

maandag 13 november 2023 8:35

Liam Luys – r0934449 - 2e BA Informatica
Besturingssystemen

KU LEUVEN

Locks and Condition Variables

Kunnen redeneren! Mindset ontwikkelen om te kunnen redeneren over implementaties.
Op examen krijg je implementatie en moet je kunnen zeggen of die correct/fout is

Multi-Threaded Programming
- Niet makkelijk!

Bij werken met meerdere threads die toegang hebben tot dezelfde datastructuren
- De datastructuren moeten "correct" zijn
  - Bij overlap van threads mag er niets fout gaan binnen data
  - Bij vb van optellen in les was dit niet correct, door overlap in algoritmes kreeg je geen verwacht resultaat

- Hardware moet correct zijn
  - Single-Threaded: Lezen van een adres geeft de waarde die er het laatst is op geschreven
  - Multi-Threaded: Hardware geeft zeer kleine garanties

- Zonder synchronisatie is het fout
  - De dag van vandaag kan hardware geen garanties geven
  - Je moet werken met synchronisatie-mechanismen

Out of Order Execution
- CPU kan instructies veranderen van volgorde indien het geen invloed heeft binnen die thread
- Zorgt ervoor dat het bij multi-threads mis kan gaan

28. Locks

Gebruik van Locks
- Rond kritische secties
- Coarse-Grained: Dezelfde lock plaatsen rond kritische secties
- Fine-Grained: Verschillende data(structuren) beschermen met verschillende locks

Building a Lock

Hoe efficiënt een lock ontwikkelen?

Extra Support nodig
- Hardware
  - Nieuwe instructies

- OS
  - Interpretatiemogelijkheden

Eigenschappen Locks
- Mutual Exclusion
  - Maximum 1 thread tegelijk in een kritische sectie

- Fairness
  - Heeft elke thread kans op de lock? Zonder starvation (verhongering)

- Performance
    - Zijn er time overheads?

Building with hardware

Poging 1: Controleren van Interrupts

Bij gebruik van 1 CPU
- Probleem doet zich enkel voor bij interrupts tussen meerdere virtuele CPU's
- Lock implementeren zodat interrupts niet meer mogelijk zijn

Nadelen
- Thread zou geprivilegieerde instructie mogen oproepen (interrupts uitzetten)
- Niet geldig voor meerdere CPU's
- Als interrupts niet meer mogelijk zijn --> Oneindige loop van programma mogelijk

Poging 2: Loads/Store in gedeelde variabele

Gebruiken van gedeelde boolean
- Geeft aan of de lock open of dicht is
- In andere threads wachten tot lock open is, zelf de lock dicht zetten

```
typedef struct __lock_t { int flag; } lock_t;
void init(lock_t *mutex) {
    mutex->flag = 0;            // 0 -> lock is available, 1 -> held
}
void lock(lock_t *mutex) {
    while (mutex->flag == 1) ; // spin-wait until available
    mutex->flag = 1;            // now SET it to mark unavailable!
}
void unlock(lock_t *mutex) {
    mutex->flag = 0;            // mark available again
}
```

Nadelen
- Fout mogelijk in uitvoering
    - Interleavings mogelijk bij wachten/opzetten van lock
    - Threads die fout lopen zeker mogelijk --> Wordt niet gebruikt!

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Correcte en efficiënte locks niet mogelijk zonder hardware support!

Poging 3: Hardware Support

Hardware Support om doel te bereiken? (Hardware Primitives)
- Test-And-Set Instructie
    - Hardware Instructie die tegelijk leest en schrijft
    - Gebeurt **atomically**
    - Lost probleem van poging 2 op

```
int TestAndSet(int *old_ptr, int new) {
  int old = *old_ptr; // fetch old value at old_ptr
  *old_ptr = new;     // store 'new' into old_ptr
  return old;         // return the old value
}
typedef struct __lock_t {
    int flag;
} lock_t;
void init(lock_t *lock) {
    lock->flag = 0; // 0: lock is available, 1: lock is held
}
void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1) ; // spin-wait
}
void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

- Compare & Swap
    - Nakijken of pointerwaarde gelijk is aan gegeven waarde en indien zo wisselen

- Load-linked en Store-Conditional

```
int LoadLinked(int *ptr) {
    return *ptr;
}
int StoreConditional(int *ptr, int value) {
    if (no update to *ptr since LoadLinked to this address) {
        *ptr = value;
    return 1;       // success!
    } else {
        return 0;   // failed to update
    }
}
```

```
void lock(lock_t *lock) {
    while (1) {
        while (LoadLinked(&lock->flag) == 1)
            ; // spin until it's zero
        if (StoreConditional(&lock->flag, 1) == 1)
            return; // if set-it-to-1 was a success: all done
    }               // otherwise: try it all over again
}
```

- Fetch-And-Add
    - Automatisch waarde verhogen en tegelijk de oude waarde teruggeven
    - Wordt gebruikt in Ticket Lock
    - Enige eerlijke spinlock!

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;
void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}
void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn) ; // spin
}
void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

Evaluatie van spinlocks

- Ze zijn correct (via wiskundige analyses bewezen)

- Performantie
  - Hangt af van aantal threads per CPU en de reden voor de lock
  - Worst Case: Veel threads op 1 CPU die allemaal de lock willen
  - Best Case: Elke CPU 1 thread en korte kritieke secties

- Fairness
  - Geen enkel van bovenstaande implementaties zijn fair (buiten laatste)
  - Als meerdere CPU's de lock willen hangt volgorde af van toeval en CPU, niet 'eerlijk'
  - Zou je kunnen oplossen met wachtrij/tickets (zoals bij de slager)

Spin Locks = Locks met while loop
Sleep Locks = Locks die volledig stilliggen tussen processen

Building with OS support

Wat te doen indien interrupt in kritieke sectie en threads die voor altijd spinnen?

★ ZIE IN BOEK/OPNAME en volledig begrijpen!

Poging 1: Yielding the CPU

```
void init() {
    flag = 0;
}
void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}
void unlock() {
    flag = 0;
}
```

Basisidee

- CPU tijdelijk iets anders laten doen ipv spinnen
- $yield()$ call naar OS die CPU mag gebruiken voor andere thread
  - Schakelt status van thread van "Running" naar "Ready"

- Nadelen
  - Nog steeds grote kost van context-switch en oproepen van lock
  - Starvation probleem niet opgelost

Poging 2: Sleeping Lock with park/unpark

We willen zelf kunnen beheren welke thread de lock mag acquiren ipv het over te laten aan het toeval van de scheduler

Implementatie

- Park: Laat thread slapen
- Unpark: Maar thread wakker
- Setpark: Geeft aan dat het gaat parken
- Queue gebruiken om bij te houden welke threads de lock willen

```c
typedef struct __lock_t {
  int flag;
  int guard;
  queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
  m->flag = 0;
  m->guard = 0;
  queue_init(m->q);
}
void lock(lock_t *m) {
  while (TestAndSet(&m->guard, 1) == 1)
    ; //acquire guard lock by spinning
  if (m->flag == 0) {
    m->flag = 1; // lock is acquired
    m->guard = 0;
  } else {
    queue_add(m->q, gettid());
    // avoid wakeup/waiting race !
    setpark();
    m->guard = 0;
    park();
  }
}
void unlock(lock_t *m) {
  while (TestAndSet(&m->guard, 1) == 1)
    ; //acquire guard lock by spinning
  if (queue_empty(m->q))
    // let go of lock; no one wants it
    m->flag = 0;
  else
    // hold lock (for next thread!)
    unpark(queue_remove(m->q));
  m->guard = 0;
}
```

Poging 3: Futex Support in Linux

```c
void mutex_lock (int *mutex) {
  int v;
  /* Bit 31 was clear, we got the mutex (the fastpath) */
  if (atomic_bit_test_set (mutex, 31) == 0) return;
  atomic_increment (mutex);
  while (1) {
    if (atomic_bit_test_set (mutex, 31) == 0) {
      atomic_decrement (mutex);
      return;
    }
    /* We have to wait. First check the futex value is truly negative (locked). */
    v = *mutex;
    if (v >= 0) continue;
    futex_wait (mutex, v);
  }
}
void mutex_unlock (int *mutex) {
 /* Adding 0x80000000 to counter results in 0 if and
 only if there are not other interested threads */
 if (atomic_add_zero (mutex, 0x80000000))
 return;
 /* There are other threads waiting for this mutex, wake one of them up. */
 futex_wake (mutex);
 }
```

Futex-Based Locks in Linux

- $futex\_wait(adres, waarde)$
- $futex\_wake(adres)$

Implementaties begrijpen

★ Waarom tweede variabele in futex_wait? Wat als de waarde er niet zou staan?

29. Lock-Based Concurrent Data Structures

Hoe locks toevoegen/implementeren in datastructuren?

Concurrent Counter

Counter Datastructuur

- Simpele datastructuur
- Optellen/aftrekken/initialiseren en opvragen
- Hoe thread-safe maken voor gelijktijdigheid?

Lock Implementaties

Big Lock (Coarse Grained)

- Bij het oproepen en net voor return van functie de lock op- en afzetten
- Je zet een lock rond alle code

- Voordeel
  - Gemakkelijk en werkt. Het maakt de code thread-safe

- Nadeel
  - Performance niet ideaal

- Twee threads kunnen nooit gelijktijdig iets doen op de datastructuur
- Kan leiden tot Contention op de Big Lock

```c
struct node
{
  int value;
  struct node* next;
};

typedef struct {
  struct node* list;
  pthread_mutex_t lock;
} list;

list empty() {
  pthread_mutex_t lock;
  pthread_mutex_init(&lock,NULL);
  list result = {NULL,lock};
  return result;
};
```

```c
int insert(list *l, int v) {
  struct node *n = malloc(sizeof(struct node));
  if (n !=0) return -1;
  pthread_mutex_lock(&l->lock);
  n->value = v;
  n->next = l->list ;
  l->list = n;
  pthread_mutex_unlock(&l->lock);
  return 0;
}
bool contains(list *l, int v) {
  pthread_mutex_lock(&l->lock);
  struct node *p = l->list;
  while (p != NULL) {
    if (p->value == v) break;
    p = p->next;
  }
  pthread_mutex_unlock(&l->lock);
  return p!=NULL;
}
```

Fine Grained

- Locks zetten op delen van de code
- Verschillende locks gebruiken voor verschillende code
- Kleinere kritische secties

- Verbetering van performance

Scalable Counting

Approximate Counter

- Werken met lokale counters voor elke CPU en 1 globale counter
- Lock voor elke lokale en voor de globale counter
- Binnen CPU tellen op de lokale counter, bij bepaalde waarde/treshold S overbrengen naar globale counter

- Hoe kleiner S, hoe meer de counter lijkt op de non-scalable counter (lagere performantie)
- Hoe groter S, hoe meer scalable maar hoe verder de globale counter kan zijn van de effectieve waarde

Concurrent Linked List

Linked List

- Gelinkte lijst met basisoperaties
- Initialiseren, insert en lookup

Locks implementeren

- Letten op errors, als malloc faalt moet voor die return ook een unlock!
  - In elk geval/mogelijk uitvoeringspad moeten de locks correct worden afgehandeld!

- Big Lock
- Enkel rond de kritische secties
  - Enkel rond kritische secties (bv niet rond malloc) verhoogt performantie en maakt soms eenvoudiger

Scaling Linked List

Hand-over-hand Locking / Lock Coupling

- Ipv één lock voor de hele list, voor elke node een aparte lock
- Zorgt voor zeer hoge concurrency, maar ook veel overhead
- Wordt niet vaak gebruikt

Concurrent Queues

Queues
- Wachtrij met basisoperaties
- Initialiseren, enqueue, dequeue

Locks Implementeren
- Big Lock
- Scott-implementatie
    - 2 locks voor head en tail, mogelijke gelijktijdigheid tussen enqueue en dequeue
    - Biedt vaak niet genoeg functies voor alle operaties --> Condition Variables zijn oplossing

Concurrent Hash Tables

Hash Table
- Simpele versie, vaste grootte

Locks Implementeren
- Big Lock
- Lock gebruiken per hash bucket
    - Zeer goede Performantie en Concurrency

Meer Concurrency is niet altijd betere performance!

30. Condition Variables

Hoe wachten op een bepaalde conditie voor verdere uitvoering.

Implementatie Concurrency
- Locks
    - Zorgen voor het beveiligen en thread-correct maken dmv kritische secties te beveiligen

- Condition Variables
    - Een thread wil nakijken of een bepaalde conditie voldaan is voor deze verdergaat met uitvoeren

Defenitions & Routines

vb: Ouder wil wachten tot kind klaar is om dan pas verder uit te voeren

```
void *child(void *arg) {
    printf("child\n");
    // XXX how to indicate we are done?
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    // XXX how to wait for child?
    printf("parent: end\n");
    return 0;
}
```

Spin Waiting
- Globale variabele gebruiken en in child deze op 1 zetten, in ouder spinnen tot de variabele op 1 staat
- Zeer inefficiënt
    - Verspilling CPU

- Beter manier om ouder te laten slapen en wakker te maken indien kind klaar is

```
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    Pthread_create(&p, NULL, child, NULL);
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    return 0;
}
```

Condition Variable

- Expliciete queue waarin threads zichzelf kunnen plaatsen
- Indien thread klaar is maakt die één/meerdere van de threads in de queue wakker om verder te laten uitvoeren

- 2 Operaties
  - $wait(\square)$ → Wanneer thread zichzelf in slaap wil zetten
  - $signal(\square)$ → Wanneer thread iets gewijzigd heeft en slapende thread wil wakkermaken

Wait-operatie

- 2 Parameters
  - Conditie
  - Mutex
    - Verondersteld dat de lock opstaat
    - Wait zet de lock af en zet de thread in slaapmodus

Signal-operatie

- Zet de lock terug op van threads die worden wakkergemaakt

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

void *child(void *arg) {
    printf("child\n");
    pthread_mutex_lock(&m);
    done = 1;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
    return NULL;
}
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    pthread_create(&p, NULL, child, NULL);
    pthread_mutex_lock(&m);
    while (done == 0)
        pthread_cond_wait(&c, &m); // releases lock when going to sleep
    pthread_mutex_unlock(&m);
    printf("parent: end\n");
}
```

Alle elementen zijn belangrijk!

Done-Variabele

- Zorgt ervoor dat de ouder weet dat het kind is uitgevoerd
- Als het kind direct zou uitvoeren bij aanmaak en eerder signalt dan de ouder wait oproept zou je anders oneindig wachten op signal die nooit komt

Locking van mutex

- Indien je niet zou locken zouden er dingen kunnen misgaan

- Als net na de while voor het slapen het kind begint uit te voeren, done op 1 zet en signalt gebeurt niets. Daarna gaat ouder verder en gaat slapen.
  Er wordt nooit meer gesignalt en slapen oneindig lang

- Lock houden als je signal of wait oproept!

While gebruiken bij nakijken van conditie
- Zie hieronder

The Producer/Consumer Problem - Bounded Buffer Problem

Producers & Consumers
- Producers zetten data in een buffer, consumers halen deze eruit en gebruiken de data

```
int buffer;
int count = 0; // initially, empty

void put(int value) {
    assert(count == 0);
    count = 1;
    buffer = value;
}

int get() {
    assert(count == 1);
    count = 0;
    return buffer;
}
```

```
1  void *producer(void *arg) {
2      int i;
3      int loops = (int) arg;
4      for (i = 0; i < loops; i++) {
5          put(i);
6      }
7  }
8
9  void *consumer(void *arg) {
10     while (1) {
11         int tmp = get();
12         printf("%d\n", tmp);
13     }
14 }
```

Hoe zorgen voor Concurrency en thread-correcte uitvoering?
- Invoeren van locks niet genoeg
- Gebruik maken van condition variables

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);             // p1
        if (count == 1)                         // p2
            pthread_cond_wait(&cond, &mutex);   // p3
        put(i);                                 // p4
        pthread_cond_signal(&cond);             // p5
        pthread_mutex_unlock(&mutex);           // p6
    }
}
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);             // c1
        if (count == 0)                         // c2
            pthread_cond_wait(&cond, &mutex);   // c3
        int tmp = get();                        // c4
        pthread_cond_signal(&cond);             // c5
        pthread_mutex_unlock(&mutex);           // c6
        printf("%d\n", tmp);
    }
}
```

2 Problemen indien meerdere consumers (voor 1 producer en 1 consumer werkt het wel)

Probleem 1: IF-statement

Probleem
- Indien producer de thread signalt en ondertussen een andere consumer kijkt of er iets in de buffer zit, dit uit de buffer haalt en gebruikt, loopt de originele consumer gewoon verder

- Het is nodig om na het wakkerworden nog eens te kijken of de buffer gevuld is

  Signal geeft een hint aan dat er dingen gewijzigd kunnen zijn, maar niet perse dat het bij de uitvoering nog zo is

```c
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);             // p1
        while (count == 1)                      // p2 NOTE change to while
            pthread_cond_wait(&cond, &mutex);   // p3
        put(i);                                 // p4
        pthread_cond_signal(&cond);             // p5
        pthread_mutex_unlock(&mutex);           // p6
    }
}
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);             // c1
        while (count == 0)                      // c2 NOTE change to while
            pthread_cond_wait(&cond, &mutex);   // c3
        int tmp = get();                        // c4
        pthread_cond_signal(&cond);             // c5
        pthread_mutex_unlock(&mutex);           // c6
        printf("%d\n", tmp);
    }
}
```

Probleem 2: Condition Variable

Stel 1 producer en 2 consumers. Producer vult buffer, signalt naar 1 consumer en wil nog verder vullen maar buffer zit vol en producer gaat slapen. Nu 1 consumer wakker, de producer en andere consumer slapen en wachten op dezelfde conditie.

Consumer runt en signalt naar één thread om wakker te maken.

- Als hij producer wakker maakt geen probleem
- Als de andere consumer wordt wakkergemaakt wel probleem!
    - Consumer wordt wakker, checkt de buffer en gaat terug slapen
    - Gevolg: Alle consumers en de producer slapen

<u>Een consumer moet enkel producers wakkermaken en omgekeerd</u>

Mogelijke Oplossingen: 2 condition variables gebruiken (hier het geval), alle threads wakkermaken (covering condition)

```c
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while (count == 1)
            pthread_cond_wait(&empty, &mutex);
        put(i);
        pthread_cond_signal(&fill);
        pthread_mutex_unlock(&mutex);
    }
}
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while (count == 0)
            pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        pthread_cond_signal(&empty);
        pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Implementaties Locks

- Spin-lock
- Sleep Locks
    - OS support nodig
    - Nuttig bij vele virtuele CPU's of lange kritische secties

<u>Wait functie essentieel om samen de lock vrij te geven en te wachten in één atomaire stap</u>

Nagekeken en vervolledigd op 02/01/2024

31. Semaphores

Hoe Semaphores gebruiken ipv locks en condition variables?

Synchronisatie Primitieven
- Locks
- Condition Variables

- Semaphores / Barriers

Semaphores / Barriers kan je implementeren met locks en condition variables

Code kunnen schrijven die synchronisatie afdwingt.

Semaphores
- Beveiliging voor toegang voor specifiek aantal gelijktijdige gebruikers
  - $sem\_init(\square)$ → specifieer het aantal toegelaten gelijktijdige gebruikers

- Objecten met int-waarde die bewerkt kan worden met 2 operaties
  - $sem\_wait(\square)$ → verlaag waarde met 1 en wacht als waarde negatief of 0 is
  - $sem\_post(\square)$ → verhoog waarde met 1 en als er threads wachten, maak één wakker

- Initialisatie door $sem\_init(s, shared, value)$ met s semaphore, shared binnen één proces op 0 en value 0 in begin

Binary Semaphore
- Implementatie van lock door semaphore
- Semaphore met startwaarde 1
- Biedt dezelfde functionaliteiten als locks

Semaphores voor Ordening
- Implementatie van condition variable door semaphore
- Semaphore met startwaarde 0

The Producer/Consumer Problem

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    Sem_wait(&empty);
    Sem_wait(&mutex);
    put(i);
    Sem_post(&mutex);
    Sem_post(&full);
  }
  return NULL;
}
```

```
1  void *consumer(void *arg) {
2    int i;
3    for (i = 0; i < loops; i++) {
4      Sem_wait(&full);
5      Sem_wait(&mutex);
6      int tmp = get();
7      Sem_post(&mutex);
8      Sem_post(&empty);
9      printf("%d\n", tmp);
10   }
11   return NULL;
12 }
13 int main(int argc, char *argv[]) {
14   // ...
15   Sem_init(&empty, max); // max are empty
16   Sem_init(&full, 0);    // 0 are full
17   Sem_init(&mutex, 1);   // mutex
18   // ...
19 }
```

Toevoeging van mutex-semaphore
- Als lock indien er tegelijk put/get wordt opgeroepen

- Zie boek voor hele uitleg p 397-400
- Mutual Exclusion

- Moet rond de get/put staan, niet aan buitenzijde
    - Indien de lock erbuiten staat en de consumer wacht op de producer staat de mutex-lock op en kan de producer de lock niet nemen. Dat geval wachten ze beide op dezelfde lock --> DEADLOCK

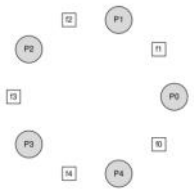
    - Oplossing om mutex lock enkel rond put/get operatie te zetten

Andere Synchronisatieproblemen / Toepassingen

Reader-Writer Locks
- Bestanden mogen door meerdere tegelijk gelezen worden, maar bewerken moet exclusieve toegang zijn
- Als een reader lock is opgezet mogen meerdere readers de lock nemen
- Als een thread een write lock wil moet die wachten tot alle readers klaar zijn
    - Niet altijd eerlijke oplossing
    - Beter om geen read-locks meer toe te laten zodra iemand een write lock heeft aangevraagd

Dining Philosophers Problem
- 5 mensen zitten aan ronde tafel, tussen elk paar mensen ligt 1 vork
- Elke persoon heeft een tijd waar hij denkt (geen vork nodig) en eet (2 vorken nodig)

- Semaphore voor elke vork
    - Locks opzetten voor iedereen aan tafel. Eerst wachten op de linkse, dan op de rechtse
    - Bij één persoon omgekeerd werken om deadlock te vermijden



Pseudo-code for philosopher:

```
while (1) {
    think();
    get_forks(p);
    eat();
    put_forks(p);
}
```

Thread Throttling
- Hoe voorkomen dat te veel threads tegelijk uitvoeren en het systeem platleggen?
    - Een limiet invoeren en met semaphore controle houden

Implementatie Semaphores door Locks/Condition Variables

```
typedef struct __Zem_t {
    int value;
    pthread_cond_t  cond;
    pthread_mutex_t lock;
} Zem_t;
void Zem_init(Zem_t *z, int value) { // only one thread can call this
    z->value = value;
    Cond_init(&z->cond);
    Mutex_init(&z->lock);
}
void Zem_wait(Zem_t *z) {
    Mutex_lock(&z->lock);
    while (z->value <= 0) Cond_wait(&z->cond, &z->lock);
    z->value--;
    Mutex_unlock(&z->lock);
}
void Zem_post(Zem_t *z) {
    Mutex_lock(&z->lock);
    z->value++;
    Cond_signal(&z->cond);
    Mutex_unlock(&z->lock);
}
```

32. Common Concurrency Problems

Hoe omgaan met mogelijke concurrency bugs bij het implementeren?

Types Concurrency Bugs

- Dead Locks
- Data Races
    - Twee threads in hetzelfde geheugen waarvan minstens 1 schrijft
- Atomicy Violations
    - Code wordt verondersteld atomisch te lopen maar wordt niet afgedwongen
- Ordering Violations
    - Code wordt in volgorde verondersteld maar de volgorde wordt niet afgedwongen

Atomicy Violation Bugs

- Code wordt verondersteld atomisch te zijn maar is het niet
- Gewone if-statement die waarde bekijkt en hierna er iets mee doet, …
- Oplossing
    - Locks gebruiken rond kritieke secties

Ordering Violation Bug

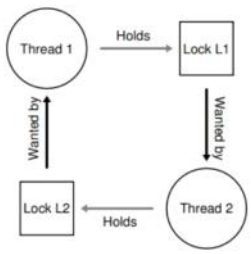
- Code wordt in volgorde verondersteld maar de volgorde wordt niet afgedwongen
- Oplossing
    - Condition Variables gebruiken

Deadlock Bug

- Meerdere threads wachten op elkaar om verder uit te voeren

```
Thread 1:                    Thread 2:
pthread_mutex_lock(L1);      pthread_mutex_lock(L2);
pthread_mutex_lock(L2);      pthread_mutex_lock(L1);
```

Figure 32.6: **Simple Deadlock (deadlock.c)**



- Oplossing
    - Goed opletten bij schrijven van locking!

Wanneer komt deadlock voor?

- Mutual Exclusion
    - Threads hebben exclusieve controle
- Hold-and-wait
    - Threads houden de resources en wachten op volgende

- No preemtion
    - Resources kunnen niet afgenomen worden van threads

- Circular Wait
    - Threads wachten op elkaars resources om verder te gaan

Als één van deze niet voorkomt is een deadlock niet mogelijk

Voorkomen deadlock

- Circular Wait
    - Zorgen voor een totale ordening van lock acquires zou perfect zijn maar is vaak zeer complex
    - Zorgen voor partiele ordening van lock is een goed begin

- Hold-and-wait
    - Voorkomen door alle locks atomair gelijk te acquiren
    - Slechte oplossing voor encapsulatie, conocurrency, …

- No Preemtion
    - Ordenen van locks door toevoegen van checks "trylock"
    - Livelock kan ontstaan: Threads blijven dezelfde code runnen en in die manier wachten op elkaar

- Mutual Exclusion
    - Geen gebruik maken van locks, andere oplossingen
    - Speciale hardware-instructies *(Compare-and-swap)*
    - Livelock vaak nog wel mogelijk

- Voorkomen via scheduling
    - Scheduler laten zoeken wanneer deadlock zou kunnen ontstaan en hierop tegenwerken

- Detecteren en oplossen
    - Reboot van systeem
    - Is een goede oplossing als de kans op deadlock zo klein is

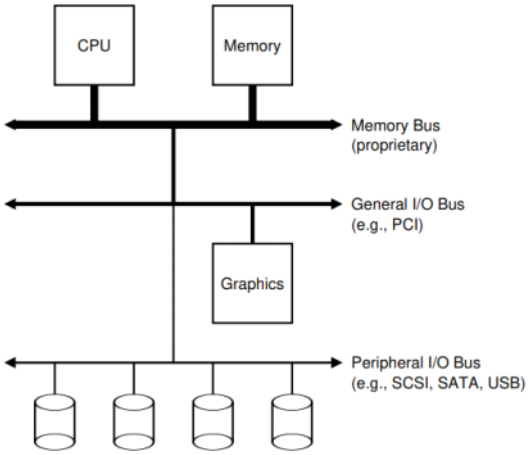
Nagekeken en vervolledigd op 02/01/2024

III - PERSISTENCE

36. I/O Devices

Hoe wordt I/O efficiënt geïntegreerd in systemen?

System Architecture



Soorten apparaten

- Snelle Apparaten
  - Northbridge
  - Geheugen, graphics
- Trage Apparaten
  - Southbridge
  - HDD, USB, …

Verbindingen

- Memory Bus
- General I/O Bus
- Peripheral I/O Bus

Snelheid van verschillende I/O devices verschilt heel erg!

Moderne Structuur

- Zie handboek pagina 447

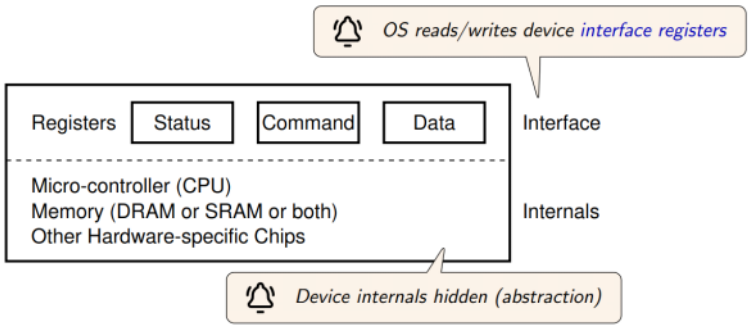
Canonical Device Model

Onderdelen van device

- Hardware Interface
  - Manier om operaties te besturen en het apparaat te bedienen
- Interne Structuur
  - Implementatie voor abstractie van systeem
  - Gebruiken van chips/simpele CPU's of firmware (software in hardware apparaat)

Communicatie I/O en apparaat?

- Via interface registers
  - Statusregister: Huidige status van apparaat lezen
  - Commandoregister: Apparaat bepaalde taak laten uitvoeren
  - Dataregister: Data doorgeven of ontvangen van apparaat



Device Driver Support

Standaardprotocol Interactie

- Kijken naar status en wachten tot vrij *(Polling)*
- Data schrijven naar dataregister *(Evt met hulp van CPU -> Programmed I/O)*
- Commando in commandoregister
- Kijken naar status en wachten tot commando uitgevoerd is

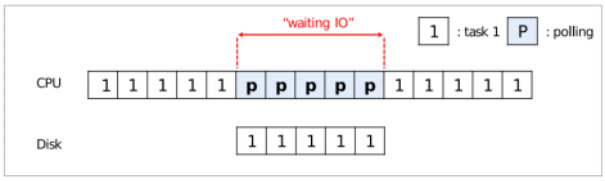
```
// 1. wait until device is not busy
while (*STATUS == BUSY) ;

// 2. Write data to DATA register
*DATA = data;

// 3. write command to COMMAND register
*COMMAND = cmd;

// 4. wait until device is done with request
while (*STATUS == BUSY) ;
```

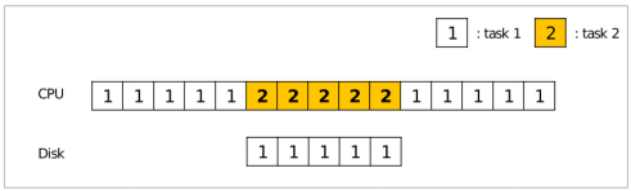
Probleem Standaardprotocol

- Busy Waiting / Polling
  - In while loop wachten tot iets veranderd = Tijdverlies
  - Oplossen door interrupts

Interrupts

- Vervanging van polling, ipv wachten tot de device klaar is de device zelf een interrupt laten sturen als het klaar is
- OS behandelt interrupt via Interrupt Handler/Interrupt service routine
- Maken overlap mogelijk tussen berekeningen en I/O door context switches





Nadelen van Interrupts

- Niet altijd mogelijk/goed
  - Bij trage apparaten/ I/O Devices zeker mogelijk, maar indien snelle apparaten zou er continu context switches aangevraagd worden waardoor er geen vooruitgang meer wordt geboekt
  - Systeem stopt met werken en crashet -> LIVELOCK
- Rekening houden met de kost van context switches

Verbeteringen

- Interrupts enkel gebruiken bij tragere apparaten
- Hybride manier als je apparaat soms traag is en soms snel
  - Voor even polling, daarna interrupt

Waarom niet alle interrupts op alle processoren?

- Time Interrupts op elke processor
  - Zo behoudt OS de controle

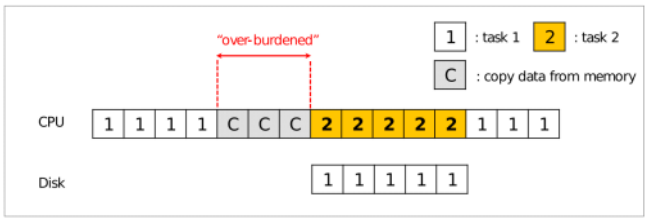
- Andere interrupts vaak voor maar één processor

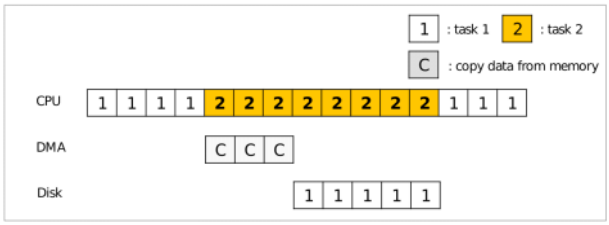
Data Verplaatsen met DMA

Data verplaatsen

- Programmed I/O
  - De CPU helpt data te verplaatsen van geheugen naar device

- Direct Memory Access DMA
  - Device in systeem dat de data kan verplaatsen in opdracht van OS





Methods of Device Interaction

Hoe communiceren hardware en OS met devices?

Communicatie Methodes

- Expliciete I/O instructies
  - Poorten van I/O aanspreken
  - Instructies moeten geprivilegieerd zijn

- Memory-Mapped I/O (MMIO)
  - Hardware maakt registers beschikbaar alsof het geheugen is
  - Fysieke adressen die gemapped worden naar devices

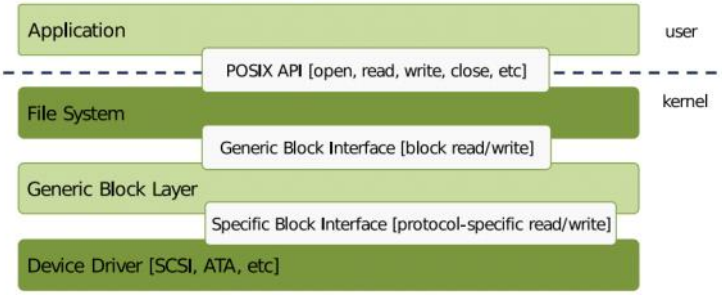
Beide kunnen voorkomen in hetzelfde systeem

Fitting Device Drivers in OS

Hoe specifieke device drivers in algemene OS brengen?

Abstractie

- OS opdelen in verschillende lagen / API Layers
- Op lager OS level specifieke Device Drivers die exact weten hoe device werkt



File System blijft gelijk voor alle devices, stuurt zijn block read/write naar generic block layer die het verder verwerkt

Nadelen van encapsulatie

- Niet altijd handig voor alle devices
- Uitgebreide device die met simpele device driver moet werken -> Veel functionaliteit verloren
  - Vb error reporting bij SCSI devices

<u>Meer dan 70% van Linux Kernel Code is van Device Drivers</u>

39. Files and Directories

Hoe moet OS omgaan met persistent Devices?

Abstracties Gezien

- The Process: Virtualisatie CPU
- Address Space: Virtualisatie Memory

- Persistent Storage

Files and Directories

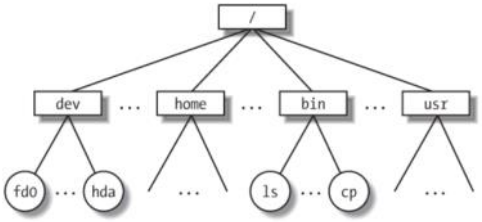
File / Bestand

- Array van Bytes
- Applicaties geven betekenis aan bytes

- Opgebouwd uit
  - Unieke naam: inode nummer

Directory

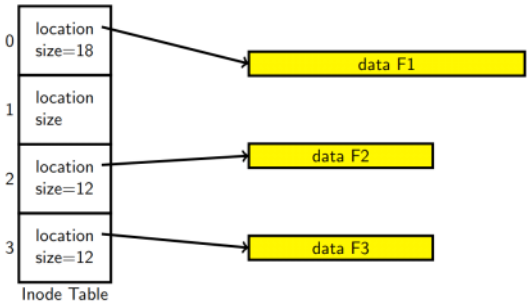
- Lijst / Mapping van namen naar inode nummers
- Start op root-directory



Bestand uniek identificeren in een file system (3 onafhankelijke manieren)

- Inode nummer en file-system id
  - Unieke id per bestand
  - Inode Table
  - Niet gebruiksvriendelijk!

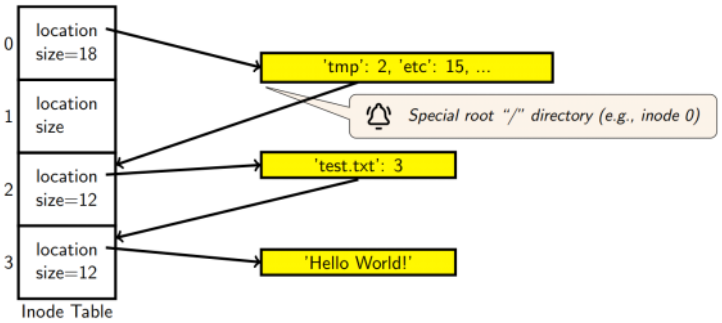
    Gebruikt door Systeem



- Path Name
  - Gebruiken van namen van bestanden en structuur *"/bin/ls"*
  - Via lookups in inode table het bestand vinden
  - Niet efficiënt!

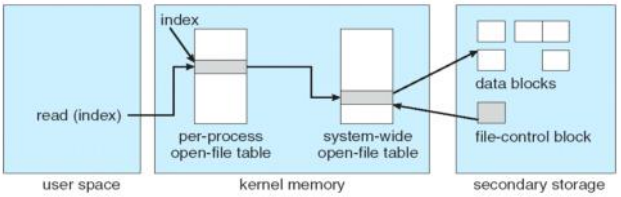
- Gebruiken van namen van bestanden en structuur *"/bin/ls"*
- Via lookups in inode table het bestand vinden
- Niet efficiënt!

Gebruikt door Mens



- File Descriptor
  - Via index en pointers naar bestand gaan
  - Nummer opslaan in per-proces open-file table
  - Waarom mapping van index naar pointers?
    - Veiligheid! Zorgen dat user niet aan andere adressen kan
    - OS behoudt controle
  - Elk proces een eigen file descriptor

Gebruikt door Proces




```
struct file {
    ...
    struct inode *ip;
    uint offset;
};

// Global file table
struct {
    struct spinlock lock;
    struct file file[NFILE];
} ftable;

// Per-process state
struct proc {
    ...
    struct file *ofile[NOFILE]; // Open files
    ...
}
```

code/fs/xv6.fd.table.c

Bestand openen
- File wordt in per-proces open-file table gezet
- In open-file table is een file descriptor aanwezig naar system-wide open-file table

File Control Block FCB
- Datastructuur die de status van een geopende file bijhoudt *(permissies, datum, gebruikers)*
- Zit in het geheugen
- Kopie ervan zit in de open-file tables

Directory is ook een file!
- Voor besturingssysteem geen verschil
- Directory file met speciale betekenis voor de bytes: bestanden binnen directory

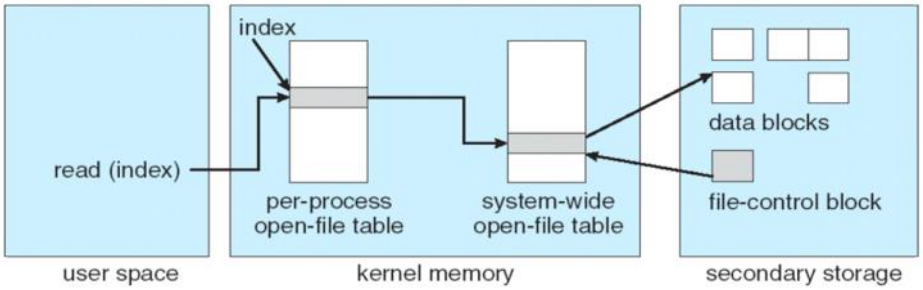
The File System Interface

Creating Files

Creating Files
- Via OPEN-systemcall
- Gaan in directoryboom, vertaalt inode, controleert permissies, alloceert in open-file tabel en geeft file descriptor terug

File Descriptor
- Getal per proces dat gebruikt wordt om bestanden te beheren
- Eens file geopend is, kan je via file descriptor lezen en schrijven
- Capability: geeft je de mogelijkheid om operaties op files uit te voeren
- Mapping van index naar pointers
  - Veiligheid! Zorgen dat user niet aan andere adressen kan
  - OS behoudt controle

- Wordt beheert door OS
    - Houdt array bij met files die geopend zijn per proces in open-file table



Example: Tracing system calls with `strace`

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)
read(3, "hello\n", 4096)
write(1, "hello\n", 6)
hello
read(3, "", 4096)
close(3)
```
code/fs/strace.txt

1. **Open** file `./foo` for reading
   → *Walk directory tree; translate inode; check permissions; allocate entry in open-file table and return file descriptor 3*
2. **Read** up to 4096 bytes from open file 3
   → *Index in process's file-descriptor table; retrieve inode; fetch data from disk; write to user-space memory buffer; return length (6)*
3. **Write** 6 bytes from user-space buffer `hello\n` to open file 1 (i.e., `stdout`)
4. Read up to 4096 bytes from open file 3
   → *Index in process's file-descriptor table; [...] return EOF (end of file)*
5. **Close** file 3

17 /

Reading/Writing Files

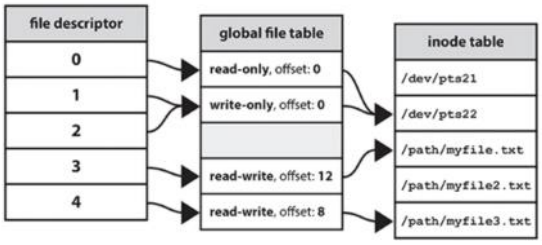
Lezen en printen van File

- File openen
- Lezen in file met READ-systemcall
- Schrijven van inhoudt naar File descriptor 1 (standaard output)
- File sluiten

Elk proces heeft standaard 3 files geopend

Standaard File Descriptors

- 0: Standaard Input stdin
- 1: Standaard Output stdout
- 2: Standaard Error stderr



Niet-sequentieel lezen en schrijven

- Niet perse van begin tot einde bestand lezen/schrijven
- Gebruiken van File Offset

File Offset

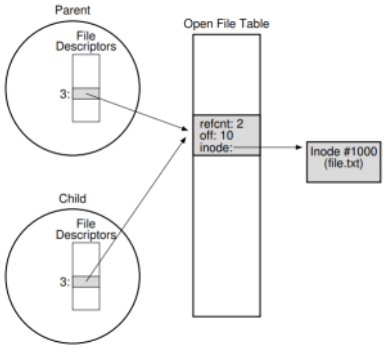
- Geeft aan waar de volgende read/write operatie zal starten
- Opgeslagen in open file table

- Wordt geüpdatet door
    - Impliciet: Read/Write van N bytes -> N toevoegen aan offset
    - Expliciet: lseek -> zet/verhoog de offset naar een bepaalde gegeven waarde

Shared File Table Entries: fork() And dup()

Fork Systemcall

- Nieuwe open-file tabel met file descriptors die verwijst naar gelijke pointers
- Gebruik maken van Reference Count
    - Aantal verwijzende processen naar File Table Entry



Dup Systemcall

- Nieuwe File Descriptor maken die verwijst naar dezelfde open files als de huidige descriptor
- Nuttig als je werkt met obv output redirection of andere UNIX shell operaties

File & Directory Operations

Writing Immediately With fsync()

Gewone Write-Operatie

- Je vraagt File System om in de toekomst aan te passen
- Kan in buffer belanden en ogenblikken later uitgevoerd worden

Fsync-Operatie

- Zorgt dat alle niet-uitgevoerde write-operaties direct uitgevoerd worden

Andere operaties op Files en Directories

Hernoemen van Files

- Via vm command in console
- RENAME-systemcall in atomaire stap
- Gebruikt bij updaten van file door editors
    - Tijdelijke file aanmaken en deze hernoemen naar originele naam

Metadata

- Data over de file
- Via stat-command bekijken
- Opgeslagen in inode

File verwijderen

- Via rm-command
- Met UNLINK-systemcall

Directory Maken

- Je kan geen directory zelf bewerken, enkel de inhoud ervan

- Via mkdir-command
- Bij aanmaak directory bestaan 2 entrys
    - Eén naar zichzelf, één naar de parent

Directory Lezen
- Via ls-command
- Opendir, Readdir en Closedir systemcalls

Directory Verwijderen
- Via rmdir-command
- Gevaarlijk, je kan veel data verwijderen dmv 1 commando
    - Hierdoor kan je enkel lege directories verwijderen

Link / Unlink

Waarom files verwijderen via unlink?
- Manieren om entries toe te voegen en te verwijderen in de file system tree

Link / Hard Link
- Nieuwe entry aanmaken
- Parameters een oude en nieuwe padnaam
- Je maakt nieuwe manier aan om te verwijzen naar dezelfde file
- Via ln-commando

- Verhoogt reference count
- Maakt nieuwe verwijzing naar hetzelfde inode-nummer

Maken van File
- Structuur opzetten (inode) die informatie over de file bijhoudt
- Naam (leesbaar) koppelen aan de file en in een directory zetten

Unlink
- Verwijdert de link tussen leesbare naam en het inode nummer
- Verminderd de reference count

- Indien reference count 0 wordt zal filesystem ook echt de file verwijderen

Limieten van Hard Links
- Je kan geen hard links leggen tussen directories
    - Gevaar voor lussen in file system tree

- Kan enkel in zelfde file system
    - Inodes zijn filesystem-specifiek

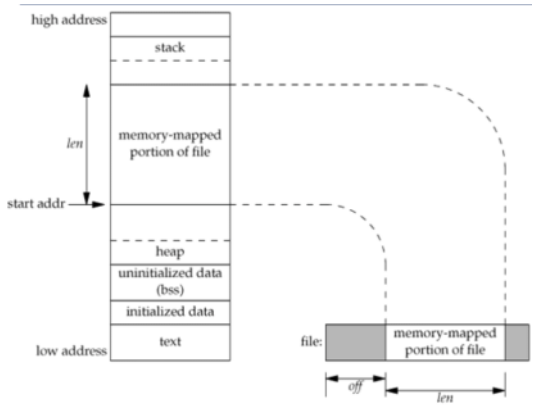
Symbolic Link / Soft Link
- Via ls-command met -s flag
- Is zelf een file, houdt de padnaam van de gelinkte file bij

- Dangling Reference
    - Als je verwijzing verwijderd is maar de link nog bestaat

Memory-Mapped Files

Memory Mapped Files
- Open files mappen naar virtuele geheugen van proces

- Via mmap system call Linux
- Voordelig bij meerdere sequentiële lees-en schrijfoperaties



Hoe en welke blokken uit disk in geheugen brengen?
- Werken met selection policy
  - Demand Paging
  - Lazy Allocation
  - Copy On Write

Permission Bits And Access Control Lists

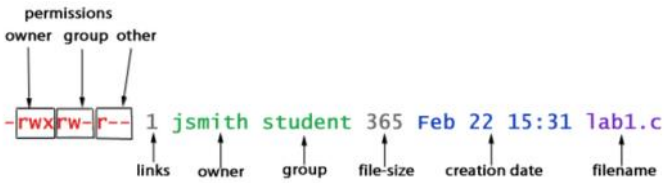
Type Files
- Gewone File (-)
- Directory (d)
- Symbolische Link (l)

Hoe zorgen voor toegangsbeheer bij gedeelde bestanden?
- UNIX Permission Bits
- Access Control Lists

Permission Bits
- Op 3 Levels
  - Owner
  - Group
  - Other

- 3 Bits
  - Read
  - Write
  - Execute

- Kan je bewerken via chmod



Access Control Lists ACL
- Per directory
- Geven exact weer wie welke toegang heeft

Nagekeken en vervolledigd op 03/01/2024

40. File System Implementatie

Welke structuren worden gebruikt om data en metadata op te slaan?

Disk I/O

- Sectoren: Kleinst adresseerbare opslageenheden
- Meestal 512 Bytes

Filesystem

- Is volledig softwareconcept
  - Hardware is 'domme' array, al de rest is software en interactie met de hardware
- 2 Grotere concepten
  - Datastructuren
  - Access Methods / Toegangsmethoden: Mapping van operaties naar datastructuren

Doel Filesystem?

- Hiërarchische bestanden
  - Verdeeld in sectoren
- Metadata per bestand
- Overhead
  - Storage Overhead: Metadata + Interne fragmentatie
  - Access Overhead: Aantal metadata accesses + Externe fragmentatie

Overall Organization

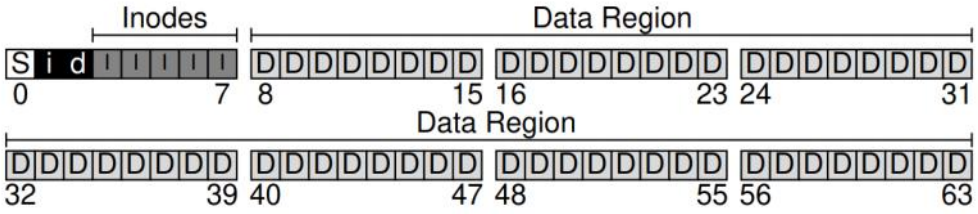
Organisatie Harde Schrijf/Disk

- Verdeelt in blokken
  - Typisch 4 KB/Blok (even groot als pages)
  - Binnen blok zitten verschillende sectoren
- Opgeldeelt in
  - Data Region
    - Blokken voor user data
  - Metadata Region

Metadata Region Opbouw

- Inodes Table
  - Per bestand een inode
  - Metadata: eigenaar, reference count, wijzigingstijd, aanmaak, locatie op disk,… *(geen filenames!)*
- Bitmap
  - Allocatiestructuur die bijhoudt welke inodes/datablokken in gebruik zijn en welke niet
  - Data Bitmap
  - Inode Bitmap
- Superblok
  - Geeft informatie weer van het file system zelf
  - Hoe moet OS het file system interpreteren?

- Opbouw
  - Aantal Inodes, aantal blokken
  - Locatie Inode-tabel
  - Magic Number: identificatie file system



Mounting
- OS leest superblock voor initialisatie van on-disk structuren
- File system van externe input uitlezen en invoegen in bestaand file system
- USB uitlezen, dat systeem komt op je computer in je eigen file system onder "/Apparaten/USB"

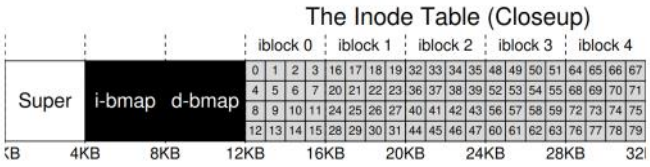
Mount Table
- Houdt info bij over elke mounted volume
- Niet zo belangrijk

Eigenschappen Metadata
- Waarom niet de naam opslaan in metadata?
  - Files kunnen zelfde namen hebben, zitten niet in de metadata maar wel in de directory

The Inode

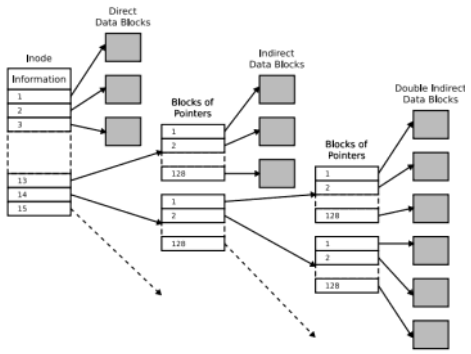
Index Node
- Metadata van bestand
- Inhoud
  - Alles behalve de file name
  - i-nummer (low-level name)
  - Pointer naar de data

$$sector = \left(\frac{inumber \cdot sizeof(inode\_t) + inodeTableStartAddr}{sectorSize}\right)$$



Hoe verwijzen naar data van inode?
- Indexed
  - Via pointers bijhouden in inode waar data staat op de disk (verwijzen naar blok)
  - Is niet handig voor grotere bestanden

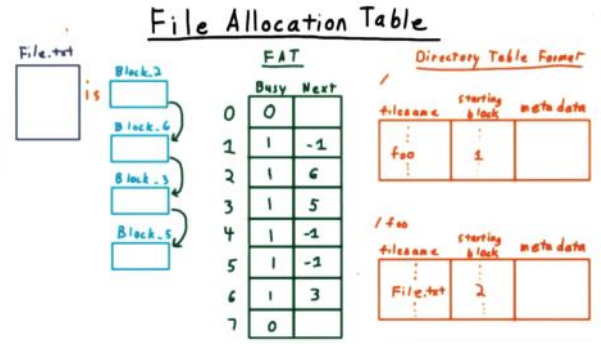
- Multi-Level Index
  - Ipv direct pointer een indirect pointer. Verwijzen naar een blok die verwijzingen bevat
  - Boomstructuur (ongebalanceerd) van pointers
  - Je kan bestanden zeer groot maken
    - Indirect pointers, Double-indirect pointers, Triple-indirect pointers



- Contiguous
  - Eén pointer bijhouden naar begin van file in inode, file in één keer op schijf (datablokken volgen elkaar op)
  - Fragmentatie slecht
  - Eens file is aangemaakt kan die niet meer groeien
  - Sequential/Random access zit goed
  - Metadata is enkel de ene pointer

- Linked Blocks
  - Blocks met data bijhouden via linked list
  - Fragmentatie zit goed
  - Prestatie niet goed, als je einde wil vinden moet je hele lijst doorlopen

- File Allocation Table
  - Tabel met block pointers
  - Per file bijhouden



Directory Organization

Directory Organization

- Directory = Mapping van filenames naar inumbers
- 2 Speciale directories
  - . = Huidige Directory
  - .. = Parent van huidige Directory

- Bijhouden van inumber, record-length, string-length en name
  - Record Length om over ongebruikte/verwijderde directory bytes te springen

- Lineaire Lookup Table

```
inum | reclen | strlen | name
   5      12        2     .
   2      12        3     ..
  12      12        4     foo
  13      12        4     bar
  24      36       28     foobar_is_a_pretty_longname
```

Waar worden directories bewaart?

- Worden vaak behandelt als speciale file
- Hebben inode en staan in inode table

Free Space Management

- Bijhouden welke inodes en data blocks nog vrij zijn
- Veel mogelijkheden
    - Bitmaps
    - Free lists
    - B-trees

- Pre-allocation
    - Proberen om contiguous files op te slaan bij aanmaak/toekenning geheugen

File-System Access Paths: Reading & Writing

Hoe system-calls mappen op file-system structuren?

Reading A File From Disk

- File openen, lezen en terug sluiten
    - Open
        - Dmv File path de inode zoeken
        - Opbouwen en starten van root directory (standaard inumber 2)
        - In directory via inode zoeken naar data regio en volgende directory volgens path
        - Recursief herhalen tot file gevonden
        - Indien inode gevonden, het inumber kopiëren naar geheugen
        - Permissies controleren

Example: `open("/foo/bar", O_RDONLY);`

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
		read							
			read		read				
				read		read			

    - Read
        - Eerste leesoperatie met offset 0 (tenzij lseek gebruikt), eerste blok lezen
        - Inode updaten met laatste-toegang tijd
        - File descriptor updaten met offset

Example: `3× read(fd, &buf, BLOCK_SIZE);`

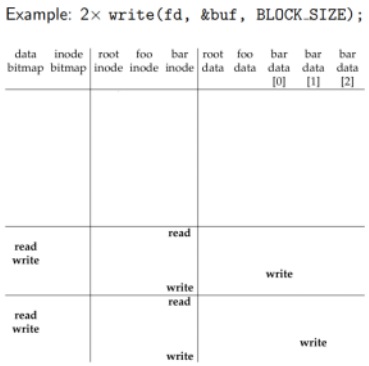
data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
				read					
				write			read		
				read					
				write				read	
				read					
				write					read

    - Close
        - File descriptor dealloceren

Writing A File To Disk

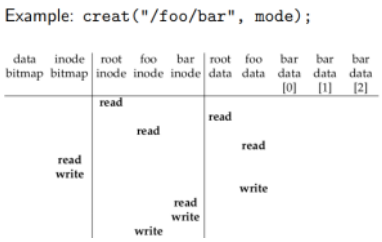
- File Openen, schrijven, terug sluiten

- Write
  - Data lezen van bitmap om lege blok te selecteren
  - Bitmap bewerken
  - Schrijf user data in nieuwe block
  - Lees inode om block pointer te zoeken
  - Bewerk inode met de nieuwe locatie van de block



Creating A File On Disk

- Doorzoek directories of file al bestaat
- Lees inode bitmap
- Bewerk inode bitmap
- Initialiseren nieuwe inode
- Link de naam met de inode in de directory data
- Lees de directory inode
- Update de directory inode



Eigenschappen

- Hoe langer de path name, hoe meer werk
  - Telkens minstens 2 accesses per directory-level: één om inode te lezen en één om data te lezen

- Eén system call komt overeen met meerdere schrijf- en leesoperaties op disk. Complex en niet zo eenvoudig als het lijkt

Caching and Buffering

Hoe hoge kosten van operaties in filesystem beperken en efficiënter werken?

Oplossing

- Belangrijke blokken bijhouden in snel geheugen d.m.v. caching

Caching

- Static Partitioning
  - Fixed-size Cache
  - Belangrijke blokken in geheugen laten staan, algoritme als LRU laten beslissen welke blokken
  - Maar wat indien niet wordt gebruikt -> Enkel verspilling van geheugen

- Dynamic Partitioning

    - Unified Page Cache
    - Ongebruikt geheugen gebruiken voor caching

Buffering

- Soms handig om buffering te doen
- Write-buffering
    - Writes uitstellen en kleinere batch aanpassingen samen uitvoeren
    - Zorgt voor minder I/O operaties

- Write delay
    - Wordt gebruikt in moderne systemen
    - 15-30 seconden wachten om te schrijven naar disk
    - Trade of tussen betere performance, en data kwijt zijn bij een crash

- Sommige applicaties moeten directe operaties kunnen uitvoeren
    - Cache en buffering omzeilen
    - Directe I/O interface

Bestand openen op Computer

Wat gebeurt er als je een bestand opent in bv Word?

1. Fork nieuw proces/Thread

2. Open() systemcall
    a. Open bestand
    b. Controle op access rights

3. System-wide Open-file Table nakijken en locatie bestand opvragen
    a. Indien beschikbaar: File toevoegen in per-process open file table, count verhogen in wide-open table
    b. Indien niet beschikbaar: zoek in memory mount table, zoek bestand in directory en zet in open-file tables
    Per-proces open-file table bevat copy van FCB, file pointer, file open count, disk location, access rights

4. Open() returnt pointer naar per-process file table

5. Reading file
    a. Kijken waar volgende block van file komt te staan

6. Indien schrijven file
    a. Exclusive Lock aanvragen

7. Close
    a. Verwijderen van per-process file table
    b. System-wide open-file table count variabele verminderen

Nagekeken en vervolledigd op 03/01/2024

42. Crash Consistency: FSCK and Journalling

Wat kan er misgaan in filesystem? Hoe gegevens bewaren bij crash of uitval systeem?

Workload in uitvoering

- vb: Block toevoegen aan bestaande file
  - Update data bitmap
  - Schrijf naar nieuwe datablock
  - Update inode metadata

- 3 schijfoperaties die telkens moeten gebeuren

Crash Consistency Problem

- Uitval tussen instructies
- Cache en buffer
  - Write-buffering die nog niet werden uitgevoerd

- Hoe omgaan met crash en vermijden van inconsistente toestand?

Crash Scenarios Workload

- Enkel datablok is geschreven
  - Metadata disc blijft in consistente state

- Inode is geüpdatet, maar pointer nog niet gealloceerd
  - Metadata inconsistent
  - Inode verwijst naar pointer waar geen data op geschreven werd

- Bitmap geüpdatet, maar inode niet ingesteld
  - Metadata inconsistent
  - Ruimte wordt gezien als gebruikt maar pointer/allocatie niet gebeurd -> Space Leak

- Inode en bitmap in orde, data niet gealloceerd
  - Metadata consistent
  - Data is garbage en niet de code die we verwachten!

- Inode in orde en datablock gealloceerd
  - Metadata inconsistent

- Bitmap en datablock in orde
  - Metadata inconsistent

Oplossingen Crash Consistency Problem

- File System Checker FSCK
- Journalling

Solution 1: File System Checker

File System Checker

- Runt in beginfase van de boot
- Wil de inconsistenties nagaan

- Gaat door hele os --> Duurt lang

Basisidee FSCK
- Laat het misgaan
- Fouten vinden we en lossen we inconsistenties later op

Werking FSCK
- Uitvoering voor file system mounted en beschikbaar is
- Superblock
    - FSCK kijkt of superblock oké is, size vergelijken met aantal blocks, …

- Free Blocks
    - FSCK gaat door inodes, indirect blocks, maakt overzicht van welke blokken gealloceerd zijn
    - Gebruikt dat overzicht om bitmap op te bouwen en te kijken of bitmap en inode consistent is

- Inode State
    - Elke inode nakijken, kijken of er een geldig veld aanwezig is, …

- Inode Links
    - Reference Count / Link count nakijken voor elke inode
    - Indien gealloceerde inode werd gevonden maar geen verwijzing ernaartoe -> Lost+Found

- Duplicates
    - Kijken naar duplicaten in pointers

- Bad Blocks
    - Als pointer naar iets wijst buiten toegelaten intervallen

- Directory Checks
    - Content van directories nakijken en inodes checken die er in staan

Probleem FSCK
- Veel te traag
    - Duurt lang om alles te controleren

- Performantie niet ideaal
    - Zeker niet voor grotere disks -> Andere oplossing nodig!

Solution 2: Journaling

Journaling
- Gebruik maken van logging
- Schrijfoperaties duurder maken om reboot makkelijker te maken
- Indien crash kijken in log waar je verder moet gaan
    - Niet meer nodig om hele schijf na te kijken
    - Grote tijdswinst tijdens boot

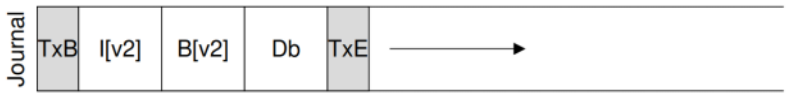
Data Logging
- Voor elke write-operatie log schrijven

- Soorten Logs

    - Physical Logging
        - De inhoud en datablokken zelf in de log opnemen

    - Logical Logging
        - Enkel loggen wat gebeurt "datablok Db toevoegen aan X" zonder de inhoud te loggen
        - Neemt minder plaats in

- Transactiebegin TxB en transactie einde TxE
    - Bevatten TID Transaction Identifier



Checkpointing: Eens de log is opgeslagen kunnen we beginnen aan overwrite

Protocol 1

Operaties bij Write

- Journal Write
    - Schrijf alle transacties en data/metadata updates naar de log

- Checkpoint
    - Schrijf de data en metadata updates naar de echte locatie in het file system

Crash tijdens Journal Write?

- Oplossen door
    - Eén voor één de blokken (TxB, I[v2], B[v2], Db, TxE) schrijven -> Langzaam
    - Alles tegelijk schrijven (atomair) -> Onveilig wegens opdeling door scheduler van write-operatie

- Moeilijk probleem!

Journal Write Opdeling

- Eerst alle blokken behalve TxE tegelijk in journal schrijven
- Daarna TxE schrijven

Protocol 2

Aangepast Protocol

- Journal Write
    - Schrijf alle transacties en data/metadata updates naar de log behalve TxE

- Journal Commit
    - Schrijf commit block TxE naar journal

- Checkpoint
    - Schrijf de data en metadata updates naar de echte locatie in het file system

Problemen Protocol

- Als Journal log vol zit
    - Moeilijke recovery en geen commit meer mogelijk van transacties
    - Lijst wordt nooit leeggemaakt

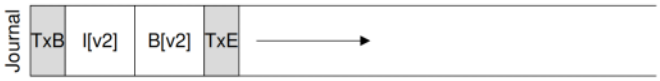
- Data wordt 2 keer naar disk geschreven
    - Eén keer in de log, andere keer naar finale plaats

Circular Journal Log

- Meerdere transacties in Journal sorteren van jong naar oud
- In Journal superblock overzicht bijhouden van transacties

Ordered Journalling / Metadata Journalling

- Geen user data overbrengen naar Journal, enkel de metadata in de log
- Ordening belangrijk!
  - Datablok moet geschreven worden voor de transactie
  - Anders zou log fout verwijzen naar garbage



Protocol 3

Metadata Journaling Protocol

- Data Write
  - Schrijf data naar finale locatie in file system

- Journal Write
  - Schrijf alle transacties en data/metadata updates naar de log behalve TxE

- Journal Commit
  - Schrijf commit block TxE naar journal

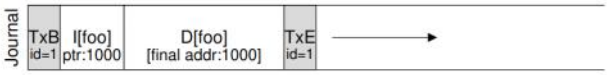
- Checkpoint
  - Schrijf de data en metadata updates naar de echte locatie in het file system

- Free
  - Markeer de transitie als free in de journal superblock

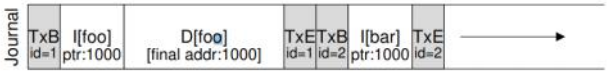
Block Reuse/Hergebruik

Probleem

1. Update directory `foo`:



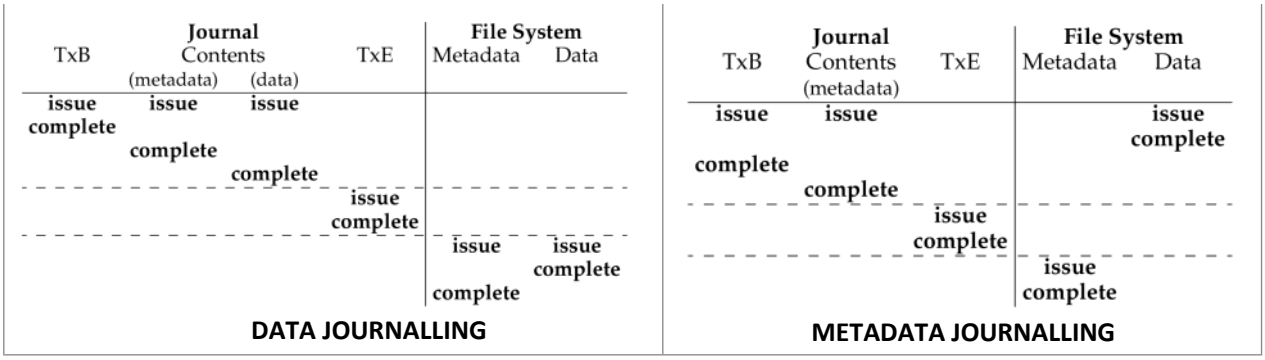
2. Directory `foo` is deleted, block 1000 is freed up for reuse
3. User create a new file `bar`, reusing block 1000 for data (which is *not* journaled!):



4. Crash! → Replay entire log, including the write of directory data in block 1000; *thus overwriting user data of `bar` with old directory contents!*

Mogelijke Oplossingen

- Nooit block hergebruiken voordat het checkpointed uit de journal
- Revoke Record
  - Indien data wordt verwijderd een revoke record op jounal plaatsen
  - Bij recovery zoeken naar revoke records en data ervan nooit overschrijven



<u>We kijken naar consistente staat van disk, niet naar gebruikersgemak!</u>
<u>We gooien soms gegevens weg van gebruiker om disk consistent te houden</u>

Examen

5 vragen

- 1e vraag
  - Aantal termen gegeven, definieer beknopt en correct de term
  - Leg in bepaalde plaats een term uit

- 4 overige vragen
  - Elke vraag over groot deel in cursus
  - Gelijkaardig aan homeworks+-

Nagekeken en vervolledigd op 03/01/2024