Betriebssysteme

Lernzusammenfassung

Lukas Bach
4. März 2017

zum Modul Betriebssysteme
am Karlsruher Institut für Technologie

4. März 2017

Inhaltsverzeichnis

1 Übersicht

Wofür Betriebssysteme?

- Abstraktion für Programme
- Schutz vor Benutzern/Programmen

Was ist ein Betriebssystem

- Ressourcen Verwalter
- Kontrollprogramm

Geschichte

- 1. Generation: Vakuumröhren und Stechkarten
- 2. Generation: Transistoren und Stapelverarbeitung
- 3. Generation: Integrierte Schaltkreise, Multiprogrammierung
- 4. Generation: Personale Computer
- 5. Generation: Mobile Computer

Modes of Execution 4 Ringe (0-3, Ring 1-2 sind Gerätetreiber), bzw. zwei Modi:

- User Mode (Ring 3)
- Kernel Mode (Ring 0)

OS Abstraktionen

OS-Abstraktion	HW Ressource
Prozesse, Threads	CPU
Adress Space	Hauptspeicher
Dateien	Disk, CD...

Caching Speicherzugriffe werden nach Lokalität gecached:

- Örtliche Lokalität (z.B. Arrays)
- Zeitliche Lokalität (z.B. Schleifen)

Speicher Latenzen Beispiel: Intel Sandy Bridge

- Register: 1 CPU Zyklus
- L1 Cache: 4 CPU Zyklen
- L2 Cache: 12 CPU Zyklen
- L3 Cache/LLC: 28 CPU Zyklen
- DDR3-12800U: 28 CPU Zyklen + 50ns

Cache Hit Arten

- Compulsory Miss: Erstreferenz
- Capacity Miss: Platzproblem
- Conflict Miss: Kollision durch Platzierungsstrategie

Gerätezugriff

- Port-Mapped I/O: Spezielle CPU-Anweisungen für port-mapped Register und Speicher sowie Gerätedatentransfer.
- Memory-Mapped I/O: RAM und Gerätespeicher teilen den Adress-Space.

2 Betriebssystemskonzepte

Aufruf des Betriebssystems durch eine der drei Möglichkeiten:

- System Calls
- Interrupts
- Exceptions

Systemcalls vs APIs syscalls werden meist indirekt durch Bibliotheken wie Win32 API oder POSIX API aufgerufen.

trap Alle Systemcalls werden initialisiert

durch trap, was einheitlich zum Betriebssystem und in den Kernelmodus wechselt und dort den Systemcall vom system call dispatcher abarbeitet.

Interrupts werden von Geräten als OS-Benachrichtigung verwendet. Beispiele: Timer-Interrupt, Interrupt durch Network-Interface-Card.

Beim Interrupt, der CPU sichert den zuvor laufenden Prozess (instruction pointer, stack pointer, status word) prüft den interrupt vector, wechselt zur interrupt service routine.

Interrupts sind asynchron.

Exceptions unterbrechen den Ablauf bei Abnormalitäten (div 0, write auf read-only Bereich...). Der Kernel untersucht das Problem, löst es und wechselt zurück oder killt den verursachenden Prozess.

Exceptions sind synchron.

Adress Space Isoliere Programme und Personen, gegen Datenklau und Bugreplikation zwischen Programmen und für Ressourcenfairness. Jeder Prozess hat scheinbar einen eigenen Speicher.

Page Faults entstehen, wenn ein Prozess auf Adressen zugreift, die nicht im RAM liegen. Ausgelöst durch die MMU, wird sie als Exception durch das OS aufgelöst. Oder: falls ein Prozess auf read-only Bereiche schreiben will, Kernel Speicher erreichen will oder instruction pointer auf executable disable memory setzen will.

Prozesse sind ausgeführte Instanzen von Programmen. Verknüpft mit einem Process Control Block (PCB), der Informationen über allokierte Ressourcen hält, und mit einem Virtual Adress Space (AS)

Adress Space Layout AS ist in Bereiche ausgelegt, zwischen denen man nicht zugreifen darf (Page Fault/Segmentation fault, Auflösung durch Prozesstötung). Das Layout ist wie folgt aufgebaut:

- Stack ↓: Funktionshistorie und lokale Variablen
- (freier Platz)
- Data ↑: Konstanten, statische+globale Variablen, strings
- Text: Programmcode

Threads bestehen aus:

- Instructios pointer register (IP)
- Stack pointer
- Program Status Word (PSW) speichert Flaggen zur Ausführungshistorie
- ...

Wenn zwischen Threads gewechselt wird, wird Scheduling angewandt.

Scheduler entscheidet welcher Job als nächstes läuft (policy)

Dispatcher führt den Prozesswechsel durch (mechanism)

I/O Geräte erlangen Performance-Verbesserungen durch OS-Verfahren wie:

- Buffering: Daten bei Übertragung zwischenspeichern
- Caching: Datenteile in schnellerem Speicher zwischenspeichern
- Spooling: Ausgabe eines Jobs mit Eingabe anderer Jobs überlappen

3 Prozesse

CPU Utilisierung $= 1 - p^n$ bei n Prozessen, die den Bruchteil p ihrer Zeit mit I/O-Warten verbringen.

Concurrency vs. Parallelism Beides wird vom CPU verwendet um Multiprogrammierung zu implementieren.

- Konkurrenz/Pseudoparallelismus: Mehrere Prozesse auf einer CPU
- Parallelismus: Mehrere Prozesse gleichzeitig auf dedizierten CPUs

80:20 Regel 80% vom Prozessspeicher ist idle, 20% ist active working set.

Adress Space Segmentarten

- Daten- und Codesegmente fester Größe
  - BSS Segment (Block Started by Symbol): Statisch allokierte uninitialisierte Variablen
  - Data Segment: Größenfeste initialisierte Variablen wie globale
  - Read-only Data Segment: Konstanten und Strings
- Stack Segment: Typische Stapelstruktur mit Stack Pointer (SP)
- Heap Segment: Dynamische Speicherallokation für Datenstrukturen beliebiger Größe
  1. Allokiere großen Speicherteil (evtl später Restteile befreien)
  2. Dynamische Partitionierung dessen in kleinere Allokationen (komplett im User Space). malloc, free

Typisches Prozess Adress Space Layout

- OS: Kerneladressen, für Prozess nicht erreichbar
- Stack ↓: Lokale Variablen, Funktionsparameter, Return-Adressen
- Heap ↑: Dynamisch allokierte Daten
- BSS: Statische uninitialisierte lokale Variablen
- Data: Initialisierte Daten, globale Variablen
- RO-Data: Read-only Daten, Strings
- Text: Programm in Maschinencode

C Bauprozess

- C Präprozessor: Auflösen von Makros und Headern
- C Compiler: C-Dateien zu Outputdateien
- Linker: Kombiniere alle Outputdateien und Bibliotheken zu einer Executable.
- (Loader): Lädt Programm, initialisiert Adress Space, startet Programm.

4 Prozess API

Application Binary Interface (ABI) standardisiert Interface zwischen Programmen/Modulen/OS.

Funktionsaufruf Der Aufrufer (caller) macht:

1. Sichert Status vom local scope
2. Sichert Parameter für Subroutine

3. Überträgt Kontrollfluss

Die aufgerufene Funktion (called function) macht dann:

5. Baut local scope auf (lokale Variablen)
6. Läuft ihren Code
7. Sichert Rückgabewert für Caller
8. Springt zum caller zurück (IP)

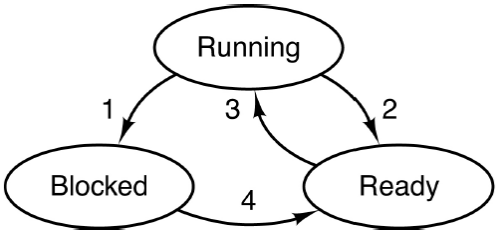
Parameterübertragung bei Funktionsaufrufen 6 können über CPU Register übertragen werden, sonstige über Hauptspeicher (Heap/Stack). Definiert durch ABI.

Prozessentstehung durch einer der Möglichkeiten:

- Booting
- Durch syscall, aufgerufen durch laufenden Prozess
- Benutzeranfrage
- Batch-Job (Cron)

Argumente beim Prozessaufruf argc ist die Anzahl an Argumenten, argv enthält einen Vektor von Argumente, envp enhält Enviroment Vector Pointers.

Prozess Status



1. Prozess blockiert für Input
2. Scheduler wählt anderen Prozess
3. Scheduler wählt diesen Prozess
4. Input wird verfügbar

Prozessterminierung aus einerm der vier Gründe:

- Normale freiwillige Terminierung
- Freiwillige Terminierung durch Fehler
- Unfreiwillige Terminierung durch fatalen Fehler
- Unfreiwillige Terminierung durch anderen Prozess (Kill)

Zombie (process stub) ist ein terminierter Prozess, der darauf wartet, dass ein Exit Status abgeholt wird, sodass er deallokiert werden kann.

5 Threads

Grundlegendes Prozesse können mehrere Threads haben. Manche Daten sind Thread-lokal, macnhe Thread-global (aber Prozess lokal)

In Linux: Prozesse haben geteilte Ressourcen und Adress Space Regions.

POSIX Thread API Jeder PThread hat eine id, (TID), einen Satz an Registern (inkl. IP, SP) und einen Stapelbereich mit dem Ausführungszustand.

- Pthreat_create
- Pthreat_exit
- Pthread_join
- Pthread_yield

Geteilte Daten Jeder Prozess hat code, data, files. Jeder Thread teilt sich diese, und hat zusätzlich registers und stack.

PCB vs. TCB Process Control Block vs. Thread Control Block:

PCB	TCB
Adress Space	Instruction Pointer
Global Variables	Registers
Open files	Stack
Child processes	State
Pending alarms	

Thread Model definiert wie die Thread-Mechanik implementiert ist (Kernel-gestützt oder nicht). Das OS weiß immer zumindest von einem Thread per Prozess.

- Many-to-One Model: User-Level Threads (ULT), komplett im User-Space implementiert.

  Die Thread-Stacks werden dabei im Heap simuliert.

  + Schnelleres Thread Management
  + Flexible scheduling policy
  + Geringe Systemressourcen
  + Unabhängig davon ob OS Threads unterstützt
  - Nicht parallel
  - Ganzer Prozess blockiert sobald ein Thread blockiert
  - OS-Teile wie Scheduler müssen re-implementiert werden.

- One-to-One Model: Kernel-Level-Threads (KLT), Kernel ist sich der Threads bewusst und steuert sie.

  Thread-Stacks sind untereinander mit jeweils Platz zum wachsen organisiert.

  + Echter Parallelismus
  + Threads blockieren unabhängig voneinander
  - OS organisiert alle Threads im System
  - Syscalls für Threadmanagement notwendig (aufwendig)
  - Scheduling fest durchs OS vorgegeben

- M-to-N Model: (Hybrid Thread Model), Kombination von beidem.

  + Flexibles Scheduling
  + Effiziente Ausführung
  - Schwer zu debuggen
  - Schwer zu implementieren (blocking, #KLTs)

Upcall wird beim hybriden Thread-Modell vom System an einen Prozess gesendet, wenn er merkt dass ein Thread blockieren wird. Der Prozess kann dann einen anderen Thread wählen. Ein späterer Upcall benachrichtigt den Thread dass die Blockade fertig ist.

6 Dispatching und Scheduling

Scheduling Problem Man hat k Jobs (Prozesse oder Threads) und n CPUs mit $k > n \geq 1 CPU$. Welche Jobs sollen von welchen CPUs erledigt werden?

Dispatcher vs Scheduler Siehe oben.

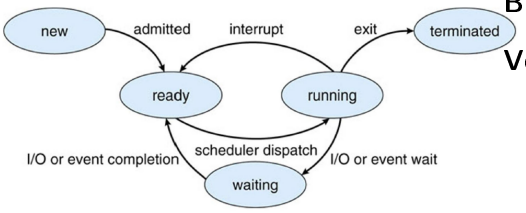
CPU switch Gründe Der Dispatcher

(Kernel) wechselt zu anderen Prozessen wenn:

- Cooperative Multitasking: Der laufende Prozess gibt freiwillig ab (yield).
- Preemtive Scheduling: Der Kernel wird in Zeitintervallen aufgerufen und benutzt Timer-Interrupts um zu wechseln.

PCB hält die Programmstatus, die bei Prozesswechsel gesichert werden müssen.

Prozessstatus Erweiterung des vorigen Modells:



Scheduler-Arten

- Short-term scheduler (CPU scheduler) für Multiprogrammierung
- Long-term scheduler (Job scheduler) für regelmäßige Jobs.

Process Scheduling Queues

- Job queue: Alle Prozesse im System
- Ready queue: Prozesse im Hauptspeicher (ready oder waiting)
- Device queues: Auf I/O wartende Prozesse

Scheduling Policies Kategorien

Alle Kategorien haben zusätzlich als Ziel: Fairness, Balance.

- Batch Scheduling: Irresponsiv, in der Industrie.

  Ziele: Throughput, Turnaround Time, CPU Utilization
- Interactive Scheduling: Optimierung der Antwortzeit, Preemption notwendig.

  Ziele: Waiting time, Response time.
- Real-Time Scheduling: Zeitgarantien, Rechenzeitvorraussichten notwendig.

  Ziele: Meeting Deadlines, Predictability.

Turnaround Time Zeit von Jobeinreichung bis Erledigung.

Burst Time todo

Verschiede Scheduling-Arten

- First-Come, First-Served (FCFS): Convoy Effect (kurze Jobs müssen ewig auf lange warten)
- Shortest-Job-First (SJF): Optimale durchschnittliche Turn-Around-Zeiten. Problem: Job-Dauern im voraus unbekannt. Lösung: Länge des nächsten CPU-Bursts vorhersagen, Prozess mit kürzestem Burst als nächstes nehmen.

  t_n = echte Länge n-ten CPU-Bursts

  τ_n = Vorhersage n-ten CPU-Bursts

  $\tau_{n+1} = \alpha t_N + (1-\alpha)\tau_n$ für $\alpha \in [0,1]$
- Preemtive Shortest-Job-First (PSJF): SJF, aber periodische Scheduling-Neuentscheidungen. Bei jeder Zeiteinheit wird Job mit kürzester Verbleib-Zeit gescheduled.
- Round Robin (RR): Gegen Starvation: In Zeitabschnitten (time quantum), unterbreche Prozesse die bis dahin nicht blockiert haben, setze

diese ans Ende der Schlange und nehme nächsten Thread dran.

Höhere durchschnittliche Turn-Around-Time als SJF, aber bessere Antwortzeiten.

- Virtual Round Robin: Gegen Unfairness gegenüber I/O-Prozessen (benutzen Bruchteile ihrer Quanten). Setze Jobs, die ihr Quantum in andere Schlange, speichere deren verbleibende Quantenzeit, gebe diesen Priorität solange ihr Quantum nicht aufgebraucht ist. Danach zurück in die normale Warteschleife.
- (Strict) Priority Scheduling (SPS): Jeder Prozess bekommt eine Prioritätsnummer. Prozess mit höchster Priorität (niedrigste Nummer) darf als nächstes rechnen.

  Falls nicht strikt: Aging (Mit der Zeit reduzieren sich Prioritäten)
- Multi-Level Feedback Queue Scheduling (MLFB): Höhere Priorität für I/O-bound Jobs, niedrigere für CPU-bound Jobs, diese dürfen aber länger (Ziel: weniger Kontext-Switching).

  Mehrere Queues mit SPS, mit jeder nächst-niedrigeren Priorität verdoppelt sich das Zeitquantum dafür. Wenn Prozesse mehrfach ihr Quantum nicht aufbrauchen, kommen sie in eine Queue höherer Priorität (und umgekehrt).
- Lottery Scheduling: Prozesse erhalten Tickets, hohe Priorität = mehr Tickets.

  Scheduler wählt dann zufällige Ticketnummer, und der Prozess mit diesem Ticket erhält das Quantum.

  Ticket Donation, falls sie auf andere Prozesse warten (stärker als Priority Donation).

Priority Donation Prozess B wartet auf Ergebnis von Prozess A, dieser hat niedrigere Priorität. Während B wartet, erhält A Priorität von B.

Boundedness Prozesse können charakterisiert werden:

- CPU-bound: Mehr Berechnungen, wenige lange CPU-Bursts
- I/O-bound: Mehr I/O-Tätigkeiten, viele kurze CPU-Bursts

7 Prozesskommunikation, Synchronisierung, Deadlocks

IPC Inter Process Communication

Direkte/Indirekte Nachrichten

- Direkte Nachrichten: Methoden send(P, message) und receive(Q, message) ermöglichen Kommunikation zwischen Prozessen P,Q.
- Indirekte Nachrichten: Nachrichten laufen über Mailboxes, die mit ids identifiziert werden. Mailboxes müssen dann von Prozessen geteilt werden.

Synchronisierung

- Blocking: Synchron. Bei einer Aktion wird auf die Nachricht gewartet.
- Non-Blocking: Asynchron. Nachrichtenaktion wird gestartet und dann weitergemacht.

Buffering Nachrichten werden in Warteschlange verschiedener Kapazitäten

gesetzt.

- Zero Capacity: Sender muss warten.
- Bounded Capacity: Begrenzte Anzahl und Länge an Nachrichten. Asynchron bis Schlange voll, dann synchron.
- Unbounded Capacity: Immer asynchron, möglicher Speicherüberlauf.

IPC durch geteilten Speicher Geteilter Bereich im Adress Space eines der Prozesse. Probleme durch Cache Koheränz, Wettläufe.

Sequentielle Konsistenz (SC) Annahme, dass Prozesse in sequentieller Reihenfolge ablaufen (Keine Realität).

x86 Speicherkonsistenz Verschiedene Befehle, um Speicherkonsistenz zu garantieren:

- mvnt überspringt Caches
- lock Präfix macht Speicherzugriffe atomar
- fence Befehle verhindert Code-Umsortierung (lfence: r, sfence: w, mfence: rw)

Erwünschte Eigenschaften von kritischen Sektionen

- Mutual Exclusion: Maximal ein Thread ist gleichzeitig in CS
- Progress: Kein Thread außerhalb der CS darf andere Threads vom betreten der CS hindern.
- Bounded Waiting: Wenn A in kritischem Abschnitt ist, soll B nicht unbegrenzt oft Eintritt versuchen.

  Auf Single-Core-Rechnern kann in kritischen Sektionen ein DNI (Don't Interrupt Bit) gesetzt werden.

Lock Variablen um zu speichern, ob in einem kritischen Abschnitt jemand ist. Auf diese muss atomar zugegriffen werden (x86 xchg um atomar Speicher und Register tauschen). Dann Implementation als Spinlock. Aber bounded waiting keine obere Grenze ⇒ Busy Waiting.

Vor-/Nachteile Spinlock:

+ Schnell wenn kurze Warte-Zeiten
- Bei langer Wartezeit Ressourcenverschwendung

Semaphore gegen Busy Waiting mit Methoden *wait*() und *signal*(). Semaphoren mit max. einem Thread in kritischen Abschnitten nennt man Mutex. Starke Semaphoren wecken Threads streng in Einschlafreihenfolge auf, schwache in zufälliger Reihenfolge.

Vor-/Nachteile Semaphore:

+ Effizient bei langen Warte-Zeiten
- Syscall-Overhead bei jeder Operation

Futex Fast User Space Mutex, Mischung aus beidem: Versuche in User-space Spinlock zu gelangen, sonst syscall um Thread schlafen zu legen.

POSIX Thread Synchronisierung durch zum Beispiel:

- pthread_mutex_t: Normale Mutex.
- pthread_cond_t: Bedinungsvariablen (wie zählende Semaphore, aber einfachere Semantik).
- pthread_rwlock_t: Reader-Writer-Locks.

Producer-Consumer-Porblem Buffer (z.B. LIFO) wird zwischen Produzent und Konsument geteilt. Wenn

Buffer leer ist, muss Konsument schlafen bis Produzent neues Element dazufügt, analog wenn der Buffer voll ist.

Gutes Beispielproblem für Wettläufe um Zählvariable.

- Schwere Lösung: Mutex und 2 Zählsemaphoren
- Einfacher: Condition Variable (CV)

1^{st} **Readers-Writers-Problem** Readers Preference. Reader sollte nicht warten wenn andere Reader schon da sind (Writer können verhungern).

2^{nd} **Readers-Writers-Problem** Writers Preference. Writr sollten nicht unnötig lange warten (Leser könnten verhungern).

3^{rd} **Readers-Writers-Problem** Bounded Waiting. Kein Thread sollte verhungern.

Dining Philosophers Problem Zyklisches Vorgehen von 5 Philosophen: Denken, hungrig werden, Stab greifen, andren Stab greifen, essen, Stäbe zurücklegen. Alles ohne Kommunikation und ohne "atomisches" greifen nach beiden Stäbchen.

- mutex_t chopstick[5]
- 4 Philosophen an Tisch für 5 (Deadlock Avoidance)
- Ungerade Philiosophen nehmen linke Stäbchen zuerst, gerade die rechten (Deadlock Prevention).

Deadlock Bedinungen alle sind für Deadlocks notwendig:

1. Gegenseitiger Ausschluss (Mutex): Begrenzte Ressourcen.
2. Hold and Wait: Auf Ressource warten, während man schon eine andere hält.
3. No Preemption: Ressourcen werden nur freiwillig zurückgegeben.
4. Zirkuläres Warten

Gutes Beispiel: Foliensatz 08, Folie 24.

Deadlock Maßnahmen

- Prevention: Verhindere im Voraus, dass sie auftreten können. Möglichkeiten für die jeweiligen Bedinungen:
  1. Mehr Ressourcen (teilen oder virtualisieren)
  2. 2-Phasen-Locking
  3. Virtualisieren für Preemption.
  4. Ressourcen mit Reihenfolge.
- Avoidance: Gegenmaßnahmen kurz vor dem Auftreten von Deadlocks. Möglichkeiten:
  - Bei jeder Ressourcenanfrage, prüfen ob System in sicherem Zustand bleibt (RAG, mehr unten).
- Detection: Im Nachhinein auflösen. Möglichkeiten:
  - WFG (mehr unten) aufrecht erhalten. Periodisch Algorithmus WFG nach Zyklen durchsuchen.
  - Recovery: Process Termination. Alle deadlocked Prozesse abbrechen, einer nach dem anderen bis Deadlock aufgelöst ist.
  - Recovery: Ressource Preemption. todo

RAG Ressource Allocation Graph. Prozesse sind runde Knoten, Ressour-

cen eckige. Instanzen von Ressourcen sind Punkte im eckigen Knoten.

Kanten von Ressource zu Prozess: Ressource ist Prozess zugeordnet.

Kante von Prozess zu Ressource: Prozess verlangt Ressource.

WFG Wait-For Graph. Knoten sind Prozesse, Kanten sind "warte auf" Beziehungen. Ressourcen nicht modelliert. Zyklen sind Deadlocks.

8 Speicherverwaltung

Swapping Datenwechsel zwischen Hauptspeicher und Hintergrundspeicher:

- roll-out: Programmzustand wird im Hintergrundspeicher ausgelagert.
- roll-in: Im Hauptspeicher wird dieser durch anderen Programmzustand ersetzt.

  \+ Prozesse sind nativ voneinander beschützt (nur Kernel noch nicht)

  \- Langsam (Übertragungszeit)

  \- Kein Parallelismus (immer nur ein Prozess)

  \- Bei Overlays muss das Programm manuell paritioniert werden

Static Relocation Fester Offset wird auf Adressen addiert, sodass jeder Prozess eigene Region hat. Aber undynamisch, kein Schutz.

Overlay Programm ist zu groß für den Hauptspeicher.

Erwünschte Eigenschaften für geteilten physischen Speicher:

- Schutz: Prozesse können nicht auf Speicher anderer Prozesse zugreifen.
- Transparenz: Prozesse brauchen keine konkreten physischen Adressen und können große zusammenhängende Speicherblöcke verwenden.
- Ressourcen Grenzenfreiheit: Unbegrenzter virtueller Speicher für Prozesse.

MMU Memory Management Unit, wie in Technischer Informatik.

Base und Limit Register speichern den Start und die Größe eines Speicherblocks eines Prozesses. MMU prüft kommende virtuelle Adressen auf Korrektheit und verwendet sie dann als physische Adressen. MMU kann so OS-Zugriff verhindern.

\+ Einfachste Weise

\+ Sehr schnell (2 Vergleiche)

\- Undynamisch

\- Geteilter Speicher?

Segmentierung Virtuelle Adressen: <segment #, offset>. Jeder Prozess hat eine Segmenttabelle, pro Eintrag: Base, Limit, Protection (für geteilten Speicher).

MMU hat 2 Register: Segment-table base register (STBR, Ort der Segmenttabelle des Prozesses) und Segment-table length register (STLR, Anzahl der Segmente des Prozesses)

\+ Geteilter Speicher gut möglich

\+ Einfache Platzierung

+ Prozess muss nicht ganz im Speicher liegen

- Segmente müssen aneinanderhängend im Speicher liegen

- Fragmentierung des Hauptspeichers

Segmentation Fault Segmentnummer ist invalide (STLR).

Paging Physischer Speicher in feste Frames aufgeteilt (Größe $(2Bytes)^n$). Virtueller Speicher ist in gleichgroße Pages aufgeteilt. OS hält page table mit dem Mapping (virtual page numbers (vpn) $\mapsto$ page frame numbers (pfn)).

Present Bit in der page table definiert, ob ein Frame mit einer Page belegt ist.

page faults entstehen, wenn eine page nicht geladen ist. MMU weist dann das OS an, diese zu laden.

Virtuelle Adressen: <page#, offset>

Hierarchische page table Meistens werden nicht alle virtuelle page nummern gebraucht, daher: page tables mappen auf andere page tables. Hierarchie:

1. Page map level 4 (PML4)
2. Page ory pointers table (PDPT) $\mapsto$ 1 GiB page
3. Page directory (PD) $\mapsto$ 2 MiB page
4. Page table entry (PTE) $\mapsto$ 4KiB page

Lineare page table normale page table.

Linear invertierte Page Table Frames werden auf Pages gemappt, und nicht umgekehrt. Dadurch eine einzige Tabelle ohne Hierarchie für alle Prozesse, weniger Speicherverbrauch.

+ Weniger Overhead für page table meta data.

- Mehr Zeitkosten für Suche bei Pagereferenzen (lineare Suche).

Gehashte invertierte Page Table Vor der invertierten Page Tabelle steht noch eine Hash anchor table. Vorgehen, wenn virtuelle page nummer umgerechnet werden muss:

- Pagenumber wird gehasht
- Bei gehashtem Index wird in der Hash Anchor Table die Adresse in der invertierten page table nachgeschlagen
- In der invertierten page table werden die Ergebnisse durchsucht, bis der richtige Eintrag gefunden ist
- Richtiger Eintrag liefert die physische Framenummer.

  Jeder Page table entry speichert dabei einen index auf den nächsten PTE.

  Dadurch müssen maximal ein paar wenige Eintrage durchsucht werden.

Page Table Entry besteht aus:

- Valid Bit aka Present Bit
- Page frame number: Falls page im Speicher, wo liegt sie?
- Write Bit: Darf die page beschrieben werden?
- Caching: Soll die page (mit welcher policy) gecached werden?
- Accessed Bit: Von der MMU gesetzt bei Zugriff, vom OS abgeholt und zurückgesetzt.

- Dirty Bit: Wie Accessed Bit, aber gesetzt bei Änderung statt Zugriff.

Externe Fragmentierung Kann durch Kompaktierung behoben werden, aber sehr teuer. Nicht bei Paging.

Interne Fragmentierung Nur bei Paging. Unbenutzer Rest am Ende von Pages. Durchschnittlich größer bei größeren Pages.

Translation Lookaside Buffer (TLB) mappt <vpn> $\mapsto$ <pfn, protection>.

Gewöhnlich: 4-fach bis voll assoziativ, 64-2K Einträge, 95-99% Hitrate.

- Software-managed: OS erhält TLB miss exceptions und entscheiden, welche Einträge überschrieben werden. Mips.
- Hardware-managed: Überschrieb durch policy definiert, OS bekommt davon nichts mit. x86-x64, ARM.

Adress Space Identifier (ASI). TLB mappt jetzt von <vpn, ASID> $\mapsto$ <pfn, protection>, sodass bei Wechsel von Adress Space nicht die ganze TLB geflusht werden muss.

TLB Reach aka TLB Coverage = TLB Größe $\times$ Pagegröße. Speichermenge, die mit TLB Hits erreichbar ist.

8.1 Page Faults

Page Fault Latency

$$\begin{aligned} p &= \text{Page Fault Rate} \\ EAT &= \text{Effective Access Time} \\ &= (1-p) \cdot \text{memory access} \\ &\quad + p(\text{page fault overhead} \\ &\quad + \text{page fault service time} \\ &\quad + \text{restart overhead}) \end{aligned}$$

Lokale und globale Allokation

- GLobale: Alle Frames stehen für Ersetzung zur Verfügung. Kein Schutz
- Lokale: Nur Frames des "faulting process" stehen zur Verfügung.
  - Gleichwertige Allok.: Jeder Prozess bekommt gleichviele Frames.
  - Proportionale Allok.: Entsprechend der Prozessgröße wird allokiert.

    $\#\text{Frames}_{Prozess} = \frac{\text{Prozessgröße}}{\text{Größe aller Prozesse}} \cdot \#\text{Frames}$
  - Priority Allocation: Allokation proportional zur Priorität statt zur Größe.

Pareto Prinzip 10% des Speichers erhält 90% der Referenzen. Ziel: Halte diese 10% im Hauptspeicher.

Trashing Jedes Mal, wenn eine Page geladen wird, wird eine andere, bald benötigte Page rausgeworfen.

Effekt: Geringe CPU-Nutzung, OS denkt mehr Multiprogrammierung wird gebraucht. $\Rightarrow$ Schlechter Effekt.

Gründe:

- Zugriffsmuster ohne zeitliche Lokalität
- Jeder Prozess passt alleine in den Speicher, aber nicht alle Prozesse
- Speicher ist zu klein, um Working Set eines Prozesses zu halten

- Page Replacement Policy funktioniert nicht gut

Working-Set Model Zeitdelta Δ mit begrenzt vielen Page-Referenzen.

WSS_i: Working Set von Prozess P_i definiert die Anzahl an Pages, die im letzten Zeitdelta referenziert wurden. Falls zu klein: Lokalität nicht gut genug erfasst. Zu groß: Enthält mehrere Lokalitäten. Unendlich: Lokalität über das ganze Programm.

Wenn $D := \sum WSS_i > m$: Trashing.

Ersetzen der Page, die am spätesten wieder referenziert wird: Halte n-bit History und Referenzbit für jede Page, bei jedem periodischen Timer-Interrupt: Shifte Referenzbit in die History, setze Referenzbit zurück. Falls History = 0: Verwerfe Page.

Page Fetch Policies

- Demand-Paging: Nur die Pages transferieren, die Page-faults erzeugen.
  + Nur Notwendiges wird transferiert
  + Weniger Speicherverbrauch pro prozess
  - Viele initiale Page-faults
  - Mehr I/O ⇒ Mehr I/O Overhead
- Pre-Paging: Spekulatives Paging. Bei jedem page-fault: Spekuliere was noch geladen werden soll.
  + Weniger I/O Overhead durch Einlesen ganzer Chunks
  - Verschwendete I/O Bandbreite falls unnötig eingelesen
  - Kann Working Set anderer Prozesse zerstören bei Page-stealing.

8.2 Page Replacement Policies

Naive Page Replacement Opfer-Page wird verworfen, Modifikationen werden zurückgeschrieben falls dirty, andernfalls schreibe in swap partition. AS-Mapping für die Page wird verworfen, Page table entry wird invalidiert und Cache wird geflusht. Dann wird neue Page gemappt.

Durchläuft zwei I/O-Operationen (swap-out old page, swap-in new page), daher verschiedene Strategien.

FIFO Älteste Page wird verworfen.

Belady's Anomalie Mit FIFO, für jede Zahl n an Frames kann man ein Referenzpermutation finden die schlechter läuft mit $n + 1$ Frames.

LRU Least Recently Used, Versuche Orakel zu approximieren. Annahme: Pages, die am weitesten in der Vergangenheit referenziert wurden, werden am weitesten in der Zukunft referenziert.

Implementationen:

Zyklischer Zähler MMU schreibt CPU Zeitstempel bei jedem Zugriff auf PTE. Bei Page-fault werden alle Einträge gescannt und ältester verworfen.

+ Billig da HW

- Speicheraufwand durch Scan

Stack Doppelt-verkettete Liste aller Pages. Referenzierte Listen

werden ans Ende gehängt.

+ Opfer in $O(1)$
- 6 Pointeränderungen pro Zugriff

Clock Page Replacement LRU Approximation, aka second change page replacement.

MMU setzt Referenzbit in PTE. Pages werden in zirkulärer FIFO Liste gehalten. Bei Opfersuche: Gehe Liste in FIFO Reihenfolge durch, falls Referenzbit = 0: Page ist das Opfer, falls Referenzbit = 1: Setze auf 0 und suche weiter.

Sonstige Strategien

Random eviction Zufälliges Opfer

LFU/MFU n-Bit Referenzcounter statt Referenzbit, dann ersetze Page mit niedrigstem/höchstem Count. Nicht häufig verwendet.

9 Caching

Im Allgemeinen weitgehend wie in Technische Informatik.

Cache hit policies

- Write-through: Speicher wird immer aktualisiert, Schreiben kann langsam sein.
- Write-back: Speicher ist zeitweise inkonsistent.

Cache miss policies

- Write-allocate: Zu überschreibende Daten werden aus dem Speicher in den Cache geladen, danach je nach write-hit policy.
- Write-to-memory: Änderung nur im Hauptspeicher.

Hashfunktionen

- Modulo Hashing:Gewohntes Hashing, jede n-te Zeile gerät in dieselbe Cachezeile.
- Arbitrary cut-out: Zusammenhängende Blöcke im Speicher mappen auf eine Cachezeile. Ineffizient, daher nicht verwendet.

Cache Design Parameter

- Kleiner Cache ⇒ Satz-assoziativ mit großen Sätzen
- Örtliche Lokalität ⇒ Lange Cache Zeilen
- Zeitliche Lokalität ⇒ Write-back policy

Virtuelly Indexed, Virtually Tagged

Virtuelle Adressen gehen von CPU zu Virtuellem Cache zu MMU zu Hauptspeicher.

Operationen, die im Cache berücksichtigt werden müssen:

- Context Switch: Cache wird invalidiert (und zurückgeschrieben bei write-back).
- fork: Kind kopiert sich den Vater-Adressspace. Cache muss das berücksichtigen.
- exec: Cache wird invalidiert, aber nicht zurückgeschrieben (wird im Speicher sowieso überschrieben)
- exit: Flush cache

- brk/sbrk (Heapgröße ändern): Wachsen ist kein Problem, bei Schrumpfen müssen selektiv Caches invalidiert werden.

  Virtually Indexed, Virtually Tagged I/O: Wie oben, aber nach dem Hauptspeicher steht I/O als weitere Stufe.

- Buffered I/O: Kein Problem.
- Unbuffered I/O:
  - Write: Information kann noch im Cache sein, daher davor write back.
  - Read: Cache muss invalidiert werden.
  - Außerdem Probleme falls I/O Region nicht an Cachezeilenlänge ausgerichtet ist.

Ambiguity Problem Identische virtuelle Adressen zeigen auf verschiedene physische Adressen zu verschiedenen Zeitpunkten. Nicht bei Physically Tagged.

Alias Problem Verschiedene virtuelle Adressen zeigen auf dieselbe physische Adresse.

Entsteht insbesondere durch shared memory (Lösung: Disallow, do not cache, ...)

Virtually Indexed, Physically Tagged Oft bei L1. Kein Cache-Flush bei Kontextwechsel (allgemein meistens kein Cache Flush).

Physically Indexed, Phyically Tagged Transparent für den CPU. ...TODO

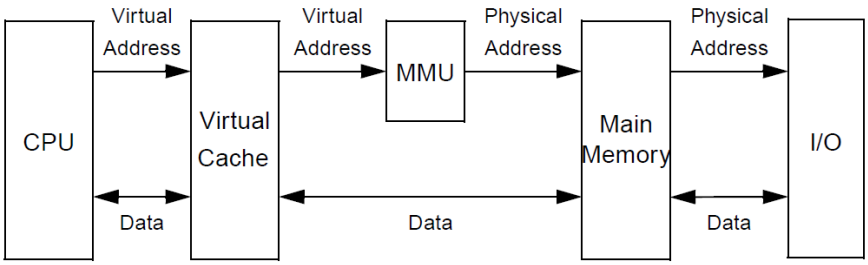


Abbildung 1: Virtually Indexed Virtually Cached mit I/O

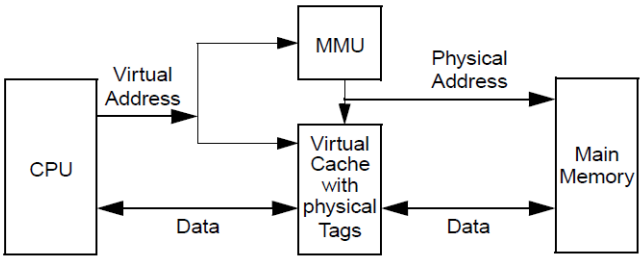


Abbildung 2: Virtually Indexed Physically Cached

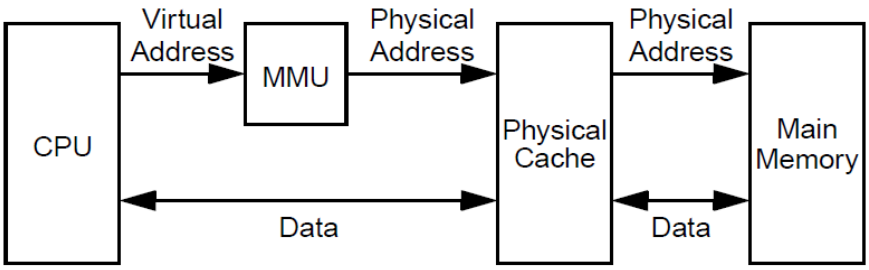


Abbildung 3: Physically Indexed Physically Cached

10 Speicherallokation

Dynamic Memory Allocation Zu beliebiger Zeit wird Speicher beliebiger Größe allokiert (Befehle allocate und free können ohne Restriktionen verwendet werden)

Datenstrukturen

- Bitmap: Speicher ist in Allokationseinheiten fester Größe aufgeteilt. Bitmap speichert, ob allokiert (1) oder frei (0). Zusätzliche Datenstruktur benötigt für Allokationslänge.
- List: Ein Listeneintrag pro allokierter Bereich (Speicher für Liste, Allokationslängen mit drin) oder pro unallokierter Bereich (braucht Datenstruktur für Allokationslängen, findet freien Speicher mit geringem Overhead), oder beides.

Fragmentierung durch Dynamic Memory Allocation. Im allgemeinen durch: Unterschiedliche Lebenszeiten von Objekten (Sonst einfach Stack), unterschiedlichen Objektgrößen, Unfähigkeit ältere Allokationen zu bewegen.

Alles der Fall bei DMA.

Strategien

- Best Fit, Worst Fit: Klar. Problem bei Best Fit: Sawdust, ist aber besser als Worst-Fit.
- First Fit: Fast so gut wie Best-Fit.
  - LIFO First Fit: Billig und schnell.
  - Next Fit: First Fit, aber merke wo letzter Platz gefunden wurde und suche ab dort. Schlechte Cache-Lokalität.

Buddy-Allocator Allokiert Speicher mit 2^n Größe, Chunks sind natürlich ausgerichtet.

Falls kein ausreichend kleiner Block frei ist, suche größeren und teile ihn in zwei gleich große Buddies. Wenn zwei Muddies frei sind, vereine sie.

Allokationsmuster von Programmen:

- Ramps: Daten werden monoton linear akkumuliert.
- Peaks: Viele Daten werden schnell allokiert, kurz benutzt, dann wieder befreit.
- Plateaus: Viele Daten werden allokiert, lange gehalten, dann wieder befreit.

Slab Allokcator Slab besteht aus mehreren Pages kontinuierlichen Speichers. Ein Cache besteht aus ≥ 1 Slabs. Jeder Cache speichert eine Arrt von Objekt (feste Größe).

Kernel kann so oft Speicher für wenige spezifische Datenobjekte fester Größe allokieren.

11 Sekundärspeicher

11.1 Magnetspeicher und Flashspeicher

Magnetspeicher Platten sind in konzentrische Tracks aufgeteilt. Ein Stack of Tracks (auf mehreren Platten übereinander liegende Tracks) heißt Zylinder. Ein Track besteht aus mehreren Sektoren.

Meistens nur ein aktiver Kopf.

Zugriffszeit aus Suchzeit (Kopf bewegen) und Rotationsverzögerung.

Suche:

- Speedup: Arm (mit dem Kopf) wird beschleunigt
- Coast: Bei maximaler Geschwindigkeit weiter bewegen
- Slowdown: Arm wird nahe des Ziels gestoppt
- Settle: Kopf wird genau auf dem Track angepasst.

Sektoren werden vom Disk Interface als logische Schnittstelle angeboten, die auf physische Sektoren gemappt werden.

Disk Interface bietet auch Disk Cache für Tracks (Read-ahead), ggf. auch Write-caching und Command Queuing.

Flash Speicher Begrenzte Zahl an Überschreibungen (dagegen flash translation layer FTL um Operationen aufzuteilen), begrenzte Haltbarkeit ohne Benutzung.

Single-level Cell (hält mehr Überschreibungen aus 10x, langsameres Schreiben) vs Multi-level Cell.

NAND Flash 2112-byte pages: 2048b data + 64b Metadata und Fehlerkorrektur.

Blöcke bestehen aus 64 (SLC) oder 128 (MLC) Pages und sind aufgeteilt in 2-4 Planes.

Eine Page wird aufeinmal gelesen in $25\mu s$ + Zeit um Daten vom Chip zu holen.

Zum Schreiben muss der ganze Block gelöscht werden (teuer), dann Daten in internen Buffer geladen werden, dann Schreiben.

SSD Perfomance Operationen

- Spare Block: SSD Controller hält einen vorgelöschten Block bereit. Bei Re-write wird dieser Block beschrieben und der zu überschreibende Block wird zur Löschung markiert (die dann im idle geschieht).
- trim Command: OS meldet dem SSD Controller unbenutzte Blöcke. Diese müssen bei SSD block rewrite nicht gesichert werden und dienen als spare blocks.

Flash Performance Charakteristika: Foliensatz 15, Folie 19.

11.2 RAID

RAID 0 Keine Redundanz, reine Datenaufteilung. Höhere Bandbreite durch Modulo-Striping.

RAID 1 Spiegelung: Volle Redundanz.

RAID 2 Redundanz durch Hamming Code. Selten implementiert.

RAID 3 Byte-interleaved Parity, Gleichpositionierter Teil jeder Festplatte wird an gleicher Stelle in Paritätsplatte gesichert.

RAID 4 Block-interleaved Parity. Ähnlich wie RAID3. Updates benötigen 2 Reads (alter Block und Paritätsblock) und 2 Writes (neuer Block und Parität), daher kann Paritätsdisk ein Flaschenhals sein.

RAID 5 Block-level distributed parity. Analog zu RAID 4, aber Paritätsblöcke werden auf allen Platten verteilt. Dadurch keine Flaschenhalsplatte, aber trotzdem langsamerer Schreibvorgang.

Komplizierte Rekonstruktion nach Ausfall.

RAID (0+1) Stripes von unabhängigen Festplatten werden gespiegelt.

RAID (1+0) Ein Streifen von mehreren Einheiten aus gespiegelten Festplatten.

11.3 Tertiärspeicher

Tertiärspeicher meist durch entfernbare Medien.

Optische Disks DVD, CD. Wiederbeschreibbar. Zum Schreiben heißt ein Laser phase-change Materialund bringt es zu formlosem kristallischem Zustand. Oder: Write Once, Read Many Times (WORM)

Magnetband Gute Leserate, sobald Daten unter Kopf liegen.

Haltbarkeit: 30 Jahre, 16000 end to end passes. Sehr billig und groß, aber langsamer beliebiger Zugriff (bis 80s).

Stacker: Bibliothek mit ein paar Tapes. Silo: Bibliothek mit tausenden von Tapes.

Wie normale Platten von OS behandelt, aber als “raw storage medium”.

Hierarchische Speicherverwaltung HSM, wird oft in Supercomputern und großen Datencentern verwendet. Häufig benutzte Dateien bleiben auf Disks, ältere inaktive Dateien werden auf Jukebox von Tapes archiviert.

12 Dateisysteme

Diskabstraktionen

- Physische Disk
- Logische Disk = Partition
- Logisches Volume = Mehrere Partitionen
- Logische Datei (file block, record, byte#)

Arten an Dateien Definiert durch deren Entitäten.

Byte Sequenz (maximale Flexibilität. Unstrukturierte generische Dateien (Plain)), Record Sequenz (meist Records fester Größe. Strukturierte Datei), Baum/Tree (oft Records variabler Größe).

Dateitypen werden kodiert in der Dateiendung (Win), FS interne Datenstruktur (Unix), im Inhalt (Unix durch Datenpräfix).

Interaktion mit Dateisystem über FS API. Dateiname wird an File Directory Service geschickt, dieser löst dateiname auf, steuert Zugriff und Sharing. fid wird zurück geliefert, diese wird an den File Storage Service geliefert, welche das Ergebnis liefert.

Abstrakte Dateioperationen create, write, read, reposition, delete, truncate, open(F_i), close(F_i).

Management geöffneter Dateien Zu haltende Informationen: file pointer pro Prozess, access rights , file-open count (wie oft ist die Datei offen, zum Steuern von Datenlöschung sobald letzter Prozess die Datei schließt), disk location.

Dateizugriff früher streng sequentiell, heute beliebig.

Plain Files Sequenz von Bytes, eventuell mit Platz dazwischen. Da Disks nur Blöcke ansprechen können, müssen diese oder ganze Dateien im Hauptspeicher gebuffert werden.

Structured File Records gleicher Größe (oder nicht, dann mit zusätzlichem Längenfeld), eventuell mit Key-feld für Sortierung der Records.

Verzeichnis Ziele: Naming, Grouping, Efficiency.

Operationen auf Verzeichnissen Datei erstellen, Datei löschem, Datei umbenennen, Dateisystem traversieren, Verzeichnis auflisten, Datei suchen.

In Unix: opendir, closedir, readdir, mkdir, rmdir.

Grundlegende Arten

- Single-Level Directory: Ein Verzeichnis für alle Benutzer. Naming Problem, Grouping Problem.
- Two-Level Directory: Ein Verzeichnis pro Benutzer. Selber Dateiname für verschiedene Benutzer. Effiziente Suche, kein Grouping.
- Tree-Structured Directories

Link Zugriff auf Dateien ohne Navigation.

- UNIX hard link: Datei wird erst gelöscht, wenn letzter Hardlink gelöscht ist. Invalide Links sind nicht möglich.
- Symbolischer Link: Keine feste Regeln (Verknüpfung).

File Sharing In Multi-User Systemen.

Probleme:

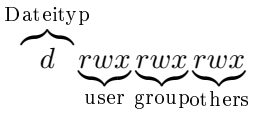
- Wie effizient dieselbe Datei erreichen?
- Wie Zugriffsrechte festlegen?

- Wie zeitgleich statt findende Zugriffe managen?

Zugriffsrechte

- None: Benutzer könnte gar nicht von der Datei wissen.
- Knowledge: Benutzer sieht nur die Datei Existenz und Besitzer.
- Execution: Benutzer kann ein Programm laden und ausführen, aber nicht kopieren.
- Reading: Benutzer kann Datei lesen inklusive ausführen und kopieren.
- Appending: Benutzer kann Daten dazufügen, aber nicht bearbeiten oder entfernen.
- Updating: Benutzer kann Inhalt bearbeiten, löschen und Datei erstellen.
- Changing Protection: Benutzer kann Zugriffsrechte Anderer ändern.
- Deletion: Benutzer kann die Datei löschen.
- Owner: Alles.

Unix Access Rights



Jedes Tripel rwx gibt an, ob gelesen (r), geschrieben (w) und/oder ausgeführt (x) werden kann. Falls eins davon nicht geht, wird das Zeichen durch ein $-$ ersetzt.

Ausführen bei Verzeichnissen heißt, in sie wechseln zu können.

Der Dateityp $\in \{d \mapsto \text{directory}, - \mapsto \text{regular files}, k \mapsto \text{block files}, ...\}$.

UNIX ACLs Access Control List. An Dateien gebundene Liste, die alle Benutzer und deren individuellen Zugriffsberechtigungen zur Datei speichert.

Gleichzeitige Dateizugriffe wird durch manche OSs durch Mechaniken unterstützt.

Programme können Dateien oder Records locken (flock()), exklusiv (writer lock) oder geteilt (mehrere Leser) und Mandatory (Zugriff durch Locks festgelegt) oder Advisory (Prozesse sehen Locks und entscheiden selbst).

12.1 Implementierung

Diskaufteilung Sektor 0 der Disk = MBR, dann Partitionstabelle, dann Partitionen. Sektor 0 einer Partition = Volume Boot Record.

Typischer Dateikontrollblock File permissions, file dates, file owner/-group/ACL, file size, file data blocks/pointers.

Allocation Policies

- Preallocation: Maximale Größe der Datei muss vorher bekannt sein. Oft schwierig, oft Überschätzung.
- Dynamic allocation: Allokiere in Teilen, sobald notwendig.

Fragmentierungsgröße Extremfälle: Grö-

ße der Datei oder kleinster Diskblock.

Konsequenzen: Zusammenhang (Lokalität), Kleinere Fragmente ⇒ Komplizierteres Management, Feste Größe ⇒ Einfachere Allokation, Variabel-große Fragmentierung ⇒ Keine interne, aber externe Fragmentierung.

FAT File Allocation Table

Speicherallokation von Dateien auf eine der drei Möglichkeiten:

- Contiguous Allocation: zusammenhängende logische Blöcke werden für jede Datei allokiert. Problem durch Fragmentierung bei Dateilöschung. Auflösung durch periodische Kompaktierung.
- Chained Allocation: Jede Datei speichert die Daten als verkettete Liste, jeder Datenblock zeigt auf den Nächsten, letzter Block enhält einen NIL-Pointer (-1). FAT zeigt auf den ersten Block.

  Keine externe Fragmentierung, nur für sequentielle Dateien sinnvoll, keine Lokalität. Beim Lesen eines einzelnen Records sehr viel Aufwand.
- Linked List Allocation within RAM: Wie Chained, aber Liste im RAM. Löst die meisten Probleme (schnelle Traversierung), aber zu groß für moderne Systeme.
- Indexed Allocation: FAT speichert Blocknummer von Indexblock, dieser speichert Index für jeden weiteren Datenblock der Datei.

  One-Level Indextable in diesem Block, manchmal auch n-Level Indextable.

  Blöcke können darin mit Längen indexiert werden, sodass ein Zeiger im Index auf mehrere zusammenhängende Blöcke zeigt.

  Zusammenfassung:

characteristic	contiguous	chained	indexed	
preallocation?	necessary	possible	possible	
fixed or variable size fragment?	variable	fixed	fixed	variable
fragment size	large	small	small	medium
allocation frequency	once	low to high	high	low
time to allocate	medium	long	short	medium
file allocation table size	one entry	one entry	large	medium

Abbildung 4: Dateispeicherallokations Möglichkeiten Vergleich

Implementation Verzeichnisse Einfaches Verzeichnis: Einträge fester Größe direkt im Verzeichnisobjekt. UNIX: Verzeichnis mit Links zu i-nodes.

Problem: Wo werden Lange Dateinahemn gespeichert? Bei jeder Datei, oder in einem Heap mit Pointer dorthin?

Lineare Suche im Verzeichnis (Lookup) Ineffizient für große Verzeichnisse. Platzeffizient, solange Kompaktion verwendet wird.

Gehashte Suche im Verzeichnis Dateiname wird auf ein inode gehasht. Dateiname und Metainfos haben variable Größe, Erstellung/Löschung startet Speicherallokation/Klärung.

Schnell und einfach, ineffizient für große Verzeichnisse.

UNIX FS Struktur Datei öffnen: file descriptor fid wird erstellt, Index für Prozessspezifische open-file-table, welche auf Systemweite Dateitabelle zeigt. inode-Table puffert Daten und liefert sie.

Logische Partition besteht aus boot block, super block (FS Charakteristiken, Blockallokationsinfos), inode table und data blocks.

Link-Counter in Dateien zählen, wieviele Hardlinks darauf zeigen.

Buffering Diskblöcke werden in Hauptspeicher gepuffert, Zugriff durch Hashtable (Auflösung durch verkettete Listen). Ersetzungsstrategie: LRU.

Freier Puffer-Management durch doppelt-verkettete Liste, die den gehashten Puffer durchläuft.

UNIX Buffer Cache

+ Geringerer Disktraffix
+ Gute Hitraten
- Write-behind policy führt zu Datenverlust bei Crashes, FS Inkosistenz
- Immer zwei Kopien: disk→

Ext2fs Davor weitgehend BSD Fast File System (ffs).

In ffs, Disk ist in Dateiblöcke von 8Kb allokiert, diese fragmentiert in 1Kb Fragmente für kleine Dateien oder partiell gefüllte Blöcke (Dateienden). Ext2fs benutzt keine Fragmentierung, Standardblockgröße ist 1Kb (oder 2, 4Kb).

Journaling File Systems melden jedes Update dem FS als Transaction, diese werden in Log geschrieben, ab dann gilt es als committet.

Wenn das FS modifiziert wurde, wird die Transaktion gelöscht. Bei Crash müssen verbleibende Transaktionen noch durchgeführt werden.

13 I/O

Bestandteile von I/O Management

- Abstraktion von physischen Details
- Einheitliche Benennung (unabhängig von HW)
- Serialisierung von gleichzeitigen I/O-Operationen

- Schutz vor unerlaubten Zugriffen
- Buffering
- Error Behandlung bei sporadischen Gerätefehlern
- Virtualisierung physischer Geräte durch Speicher- und Zeitmultiplexing

I/O Gerätcharakteristika

- Block Geräte: read, write, seek, raw/fs mapped/mem mapped
- Charakter Geräte: get, put. Z.B. Tastaturen, Mäuse, Serielles. Unter Bibliotheken.
- Netzwerk Geräte: Manchmal wie Block, manchmal wie Charakter, manchmal eigenes Interface (Socket Interface).

Typische Komponenten Controller, Port, Bus

Geräteadresse zeigt auf Statusregister, Kontrollregister, Data-in Register oder Data-out Register.

Memory-Mapped I/O

- Signle-bus: CPU, Speicher und I/O sind über einen Bus verbunden.
- Dual-bus: Wie Single, aber CPU ließt über zusätzlichen Hoher-Bandbreiten-Bus direkt vom Speicher.

Techniken:

- Programmed I/O: Thread wartet aktiv auf I/O (Prozessor ist okkupiert), Kernelthread pollt I/O-Status.
- Interrupt-driven I/O: I/O-Befehl startet, Prozessor macht Anderes, I/O sendet Interrupt bei Vollendung.
    1. Sichere Register
    2. Baue Kontext (AS) für interrupt service procedure auf
    3. Baue Stack dafür auf
    4. Maskiere Interrupt Controller $\Rightarrow$ re-enable andere Interrupts.
    5. Interrupt Service Routine läuft
    6. Eventuell Prozess höherer Priorität aufwecken
    7. Ursprüngliche Prozessregister wiederherstellen
    8. Anderen Prozess läuft
- Direct Memory Access (DMA): DMA Modul kontrolliert Datenfluss zwischen Hauptspeicher und I/O. Prozessor wird interrupted wenn ganzer Block transferiert wurde, CPU wird bypassed um Daten direkt zwischen I/O und Speicher zu übertragen.
    1. Gerätetreiber wird benachrichtigt Daten zu Buffer zu laden
    2. Gerätetreiber beauftragt Diskcontroller, c Bytes von Disk zu Buffer zu laden.
    3. Diskcontroller startet DMA Transfer
    4. Diskcontroller sendet jedes Byte an DMA Controller
    5. DMA Controller sendet Bytes an Buffer, erhöht Speicheradresse und dekrementiert c bis $c = 0$.
    6. Wenn $c = 0$, DMA sendet Interrupt an CPU, dass Transfer fertig ist.

Struktur eines Kernel I/Os

- Kernel
- Kernel I/O Subsystem

- Gerätetreiber
- Gerätecontroller (HW)
- Geräte (HW)

Kernel I/O Subsystem

- Scheduling
- Buffering: Gegen inkonsistente Gerätegeschwindigkeit/Gerätetransfergrößen, Aufrechterhaltung von Copy-Semantiken.
- Errorbehandlung und Auflösung
- Schutz vor Benutzerprozesse
- Spooling: Präpuffern von Aufträgen, bis Gerät frei ist.
- Geräte Reservierung

Reentrant Code Code, der von mehr als einem Thread gleichzeitig ausgeführt werden kann. Bei Gerätetreibern der Fall.

Device Status Table Liste aller Geräte und derer Status.

Stream Kommunikationschannel zwischen Benutzerprozess und Gerät. Besteht aus Stream-Head-Interface für Benutzerprozess, Stream-End-Interface für Gerät und ≥ 0 Stream Modulen dazwischen.

Jedes davon hat eine Lese- und eine Schreib-Warteschlange.

I/O Buffers sodass Threads währenddessen weiter arbeiten können und Pages geladen bleiben können.

Arten: Block-orientiert (Disk, Tapes), Stream-oientiert (Terminals, Drucker, Eingabe, sonstige).

Ohne Buffering: Prozess ließt ein Byte/Wort auf einmal, damit sehr großer Overhead.

- User Level Buffering: Task spezifiziert Puffer im Speicher, dessen Füllen von der Interrupt Service Routine erledigt werden kann.

  Probleme: Puffer auf Disk gepaged?
- Single Buffer: Benutzerprozess kann einen Block laden, während der nächste eingelesen wird. OS steuert Vergabe Systembuffers an Benutzerprozesse.

  Speed-Up: $\frac{T+P}{max(T,P)+C}$ mit:

  T = Gerättransferzeit, C = System- zu Benutzerbuffer-Transferzeit, P =Pufferverarbeitungszeit.
- Double Buffer: 2 Systempuffer, Benutzerprozess agiert mit einem während das OS den anderen leert/füllt.

  Speed-Up: $\frac{T+P}{max(T,P+C)}$.
- Circular Buffering: Situation: Viele Schreibvorgänge zwischen langen Rechenphasen, daher möglichst mehr als ein Block read-ahead.