

ZUF 2 BSY

René Bernhardsgürtler, 01.05.2013

Betriebssystem / Computer

Beispiel für Race Condition

Bankkonto

- Thread 1** überweist: Betrag X von Konto A nach B
→ A = A - X dann B = B + X, mit X = 20 €
- Thread 2** berechnet Kontostand S = A + B
- Threads können jederzeit unterbrochen werden

Zeit	Thread 1	Thread 2	Konto A	Konto B	A+B
T-1	...		50	30	80
T	A = A - X		30	30	60
T+1		S = A + B	30	30	60
T+2		...	30	30	60
...			30	30	60
T+n	B = B + X		30	50	80

Race Condition, wenn...

- ...Prozesse und /oder Threads gemeinsam Daten lesen und schreiben.
- ...das Resultat von der Ausführungsreihenfolge der beteiligten Prozesse resp. Threads abhängt.

Der kritische Abschnitt

- Ein Task greift auf gemeinsame Daten resp. Ressourcen zu:**

Dieser Prozess (Thread) befindet sich in seinem kritischen Abschnitt

- Aufenthalt in kritischen Abschnitt muss gegenseitig ausschliessend sein**

Zu jedem Zeitpunkt befindet sich höchstens ein Prozess (Thread) im kritischen Abschnitt

- Zutritt zum kritischen Abschnitt** Generell:
• nur über "Erlaubnis"

Lösung:

Entry Section: Sicherstellen, dass nur einer arbeitet
Working section: Einer arbeitet
Exit Section: Wieder allen freigeben

Anforderungen des Kritischen Abschnitts (=KA):

1. Nur ein Prozess gleichzeitig
2. Muss unabhängig von Prozessgeschwindigkeiten oder Anz. CPUs sein
3. Prozesse ausserhalb des KA dürfen andere nicht beeinflussen (sonst müssten sie in KA)
4. Prozesse, die in KA wollen, dürfen nicht verhungern können
5. Wenn KA frei: Ein anfragender Prozess muss umgehend rein dürfen
6. KA darf nur endlich lange von Prozess besetzt werden

Software: Lösungsansatz 1

Optimizer könnte das wegoptimieren, indem er die Variable nur einmal liest, denn sie wird im Block nie verändert.
Man könnte das Schlüsselwort 'volatile' vorsetzen (gibt auch bei java).

Task 1

```
while (true) {
    while (dran != 1) {
        kritischerAbschnitt();
        dran = 2;
        nichtkritikAbschnitt();
    }
}
```

Task 2

```
while (true) {
    while (dran != 2) {
        kritischerAbschnitt();
        dran = 1;
        nichtkritikAbschnitt();
    }
}
```

Initialisierung: dran = 1

Schreiben ist atomar! Aber ein Inkrement, lesen + schreiben, könnte unterbrochen werden!

Problem: Busy Wait!

Heisst auch Peterson's Algorithm

Hardwareunterstützung

Interrupt ausschalten

- nur auf Uniprozessoren möglich (sinnvoll)
 - Performanceeinbusse: kein Multiprogramming
- macht auf Multiprozessoren keinen Sinn
- wird in der Kernelprogrammierung angewendet
 - Kernel dürfen nur an sicheren Stellen unterbrochen werden
 - Unterbrüche i.A. aber nur kurz
- Beispiel

```
...
InterruptOFF();
kritischerAbschnitt();
InterruptON();
...
```

Hardwareunterstützung: TestAndSet

Idee

- Zutritt zu kritischem Abschnitt
 - **"Anfragen und Sperren"** darf nicht unterbrochen werden
 - sonst inkonsistente Sichten möglich
- Lösung
 - "Sperrvariable" gleichzeitig lesen und setzen
- Pseudocode

```
bool TestAndSet(bool &gesperrt) {
    if (*gesperrt == false)
        return (*gesperrt = true);
    else
        return (*gesperrt);
}
```

atomar

Hardwareunterstützung: Spin Lock

Implementation x86

- globale Variable
 - locked mit 0 initialisiert → offen
- Lock schliessen

```
spinLock: MOV AX, 1
           XCHG AX, locked;
           CMP AX, 0
           JNE try_lock
           RET
```

- Lock freigeben: unkritisch (atomar)

```
releaseLock: MOV locked, 0
              RET
```



Semaphor → Betriebssystemunterstützung

Semaphor (der, Mz.: die Semaphore)

- Semaphor: Ampel, Signal
 - 1965 von Dijkstra vorgeschlagen
- Semaphor S ist eine Integervariable
- Verwaltung durch zwei elementare Operationen

Operationen auf Semaphore

- down(S)** Semaphor schliessen, auch P(S)
- up(S)** Semaphor öffnen, auch V(S)

Vorteile

- kein **busy wait**
- wartende Prozesse
 - werden in Queue eingereiht
 - blockieren bis kritischer Abschnitt freigegeben wird

Der Semaphor ist eigentlich eine Datenstruktur

```
typedef struct {
    int count; //Wert des Semaphors
    ProList queue; //Liste wartender Prozesse
} semaphore;
```

semaphor S;

Dijkstras Bezeichnungen für Semaphoroperationen

- P(S)** probieren → testen
- V(S)** verhögen → erhöhen (freigeben)

Oft werden auch folgende Bezeichnungen verwendet

- semWait(S)** (down)
- semSignal(S)** (up)

Semaphore: Funktionsweise

Es werden beide Instruktionen ausgeführt

```
down(semaphore S){
    if (S.count == 0)
        append(P, S.queue);
    else
        S.count--;
}
```

// append calling process P to queue
// block calling process
// decrement counter, continue

```
up(semaphore S){
    if (test(S.queue) != empty)
        P = remove(S.queue);
    append(P, ready.queue);
    else
        S.count++;
}
```

// remove a process P from queue
// place P on ready queue
// increment counter, continue

S.count mit positivem Wert initialisiert (≥ 0)

Semaphore: Beispiel zu Mutex

Beispiel

- mehrere Tasks sollen die gemeinsame Variable z so aufwärtszählen, dass als Ausgabe "0 1 2 3 ..." resultiert
- Lösung
 - Initialisierung: S = 1, z = -1
 - jeder Task implementiert einen kritischen Abschnitt resp. Mutex

```
...
down(S);
z = z + 1;
printf("Count: %d\n", z);
up(S);
```

Der Codeausschnitt realisiert einen kritischen Abschnitt

- S = 1: binärer Semaphor -> nur ein Prozess im kritischen Abschnitt
- Jeder Prozess hat exklusiven Zugriff auf Variable z

Diskussion

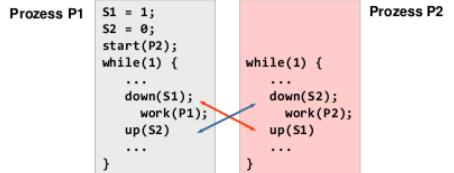
- wieso muss printf() zwischen down() und up() liegen ?
- es kann keine Aussage gemacht werden, welcher Prozess welche Zahl ausgibt !
 - Prozess treffen in *zufälliger* Reihenfolge auf down() -> FIFO Queue
 - Prozess am Kopf der Queue wird geweckt
- es ist möglich, dass der gleiche Prozess mehrmals hintereinander auf die offene Semaphore trifft, abhängig davon, was die anderen Prozesse machen

Ohne Synchronisation ist nicht gewährleistet, dass der gleiche Wert nur einmal ausgegeben ist

Semaphore: Beispiel zu Synchronisation

■ Reihenfolge von zwei Prozessen garantieren

- Prozesse P₁ und P₂ alternierend ausführen
- P₁ Hauptprozess
 - initialisiert Semaphore S₁ und S₂
 - startet Prozess P₂
- down(S_x) und up(S_x) jeweils über beide Prozesse verteilt



Aufgabe: Die Semaphore soll sich so verhalten:
P1 > P1 > P2 > P1 > P1 > P2 ...

- Man könnte bei P1 einfach 2x work(P1) eintragen.
- Man könnte S2 auf 2 initialisierten und nach dem Ausführen von S1 zweieup(S2) machen, dann wäre die 'kleine Queue' wieder zwei Schritte von der Ausführung entfernt.

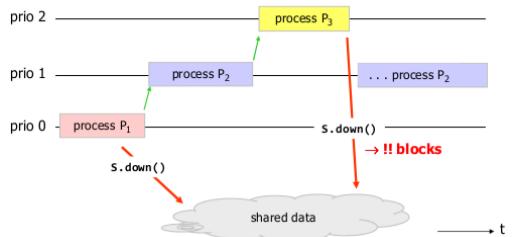
```

S1 = 2;           //P2
S2 = 0;           //P1
// P1
while(1) {
    down(S2);
    down(S2);
    work(P2);
    up(S1);
    up(S1);
    up(S2);
}

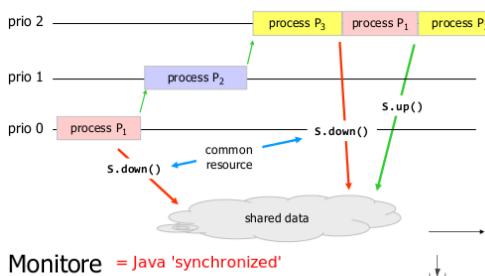
```

Probleme mit Semaphoren: Kompliziert => Fehleranfällig, Anwendung muss in allen Prozessen korrekt sein, Böswilliger Code kann die Zusammenarbeit schädigen.

Problem: Priority Inversion



Lösung: priority inheritance

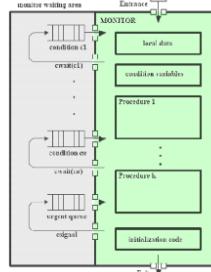


■ Monitor

- ein Softwaremodul
- Daten, Prozeduren und eine Initialisierungssequenz

■ Funktionsweise

- Zugriff auf lokale Variablen nur über Monitormethoden
- Prozess tritt in Monitor ein durch Aufruf einer dieser Methoden
- nur ein Prozess kann gleichzeitig Code innerhalb des Monitors ausführen, alle anderen müssen warten → mutual exclusion



Monitore und Java => siehste!?

■ Java Objekte

- jedes Objekt ist ein Monitor
- garantieren, dass sich nur ein Thread innerhalb des Monitors aufhält
- Eingänge in den Monitor sind **synchronized** Methoden

```

public class KontoMonitor {
    double A,B;
    public synchronized double sum() {
        return A+B;
    }
    public synchronized void xfer(double X) {
        A = A - X;
        B = B + X;
    }
}

```

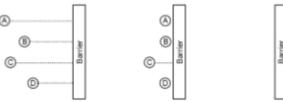
Synchronisation vs. Mutex

■ Mutex

- gegenseitiger Ausschluss bei Zugriff auf gemeinsame Daten resp. Ressourcen → Konzept des kritischen Abschnitts
- Sperren und Entsperren durch gleichen Task
- Synchronisation bezüglich Zugriff auf kritische Ressource

■ Synchronisation

- Synchronisation garantiert Reihenfolge der Verarbeitung
- Sperren und Entsperren durch verschiedene Tasks
- Beispiele
 - Sequenz
 - Barrier



Synchronisation und Mutex

■ Producer / Consumer Problem

- endlicher gemeinsamer Buffer
- Zugriffsteuerung
 - Synchronisation: counting Semaphore empty und full
 - Mutex: binary Semaphore mutex

Producer <pre> while(1) { item = produce(); down(available); down(mutex); insertItem(item); up(mutex); up(used); } </pre>	Consumer <pre> while(1) { down(used); down(mutex); item = getItem(); up(mutex); up(empty); consume(item); } </pre>
-------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------

Deadlocks

1. **Mutual exclusion**: mindestens eine Ressource ist exklusiv reserviert
2. **Hold and wait**: mindestens ein Task hat eine Ressource exklusiv reserviert und wartet auf weitere Ressourcen
3. **No preemption**: reservierte Ressourcen können dem Task nicht entzogen werden (freiwillige Rückgabe nur, wenn Aufgabe gelöst)
4. **Circular wait**: geschlossene „Kette“ von Tasks existiert, in der jeder Prozess mindestens eine Ressource reserviert hat, die auch von einem Nachfolger in der Kette benötigt wird

Deadlocks: Ressourcengrafen

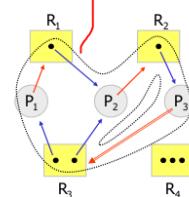
Immer wenn ein Roter Pfeil hin und ein Blauer weg, kann es einen Deadlock geben (so ein Kreis).

■ Situation

- Prozess P_i fordert zusätzlich Ressource R_j an

■ Zwei minimale Zyklen

P₁ → R₁ → P₂ → R₂ → P₃ → R₃ → P₁
P₂ → R₂ → P₃ → R₃ → P₂



■ Deadlock existiert

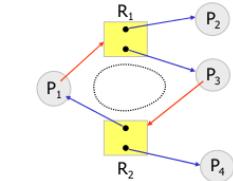
Deadlock existiert:

- Prozess P₂ wartet auf Ressource R₂, die von Prozess P₃ gehalten wird
- Prozesse P₃ wartet auf R₃, die von den Prozessen P₁ und P₂ gehalten wird
- Prozess P wartet auf Ressource R, die von Prozess P gehalten wird

■ Zyklus aber kein Deadlock

■ Zyklus

P₁ → R₁ → P₃ → R₂ → P₁



■ Kein Deadlock

- P₄ kann R₂ freigeben
- P₃ kann dann R₂ allozieren
- P₂ kann R₁ freigeben
- P₁ kann dann R₁ allozieren

Prevention

■ 4 Bedingungen

- mutual exclusion
 - lässt sich nicht grundsätzlich vermeiden
- hold and wait
 - alle benötigen zusammen Ressourcen anfordern
 - vor neuer Anforderung zuerst alle Ressourcen zurückgeben
- no preemption
 - Ressourcen, deren Zustand leicht sicherbar sind
- circular wait
 - Ressourcen nummerieren und alle zusammen in aufsteigender Reihenfolge allozieren

■ Problem

- ineffiziente Ressourcennutzung, serielle Verarbeitung
=> Unwichtig

Avoidance

■ Vermeiden

- die ersten drei Bedingungen bis auf circular wait zulassen
- überprüfen, ob ein Deadlock eintreten könnte
 - Wissen um zukünftige Resourcenanforderung notwendig
 - wenn Deadlockgefahr besteht
 - Prozess nicht starten
 - werden keine zusätzlichen Ressourcen zugesprochen
- Definition "sicherer Zustand"
 - Ressourcenzuweisung führt nicht zu Deadlock

■ Vorgehen

- z.B. Bankers Algorithmus, Dijkstra (siehe Stallings)

Detection

- Betriebssystem muss überprüfen**
 - ob ein Deadlock aufgetreten ist
 - Deadlock auflösen und lauffähigen Zustand wiederherstellen
- Erkennen**
 - z.B. Verfahren von Coffman (siehe Stallings 5th ed., Brause)
- Wann Überprüfung**
 - wenn Prozess auf Ressource warten muss (eher zu aufwendig)
 - z.B. jede halbe Stunde oder jede Stunde, etc.
 - wenn CPU Auslastung unter bestimmten Wert (z.B. 40%) sinkt

Detection: recovery strategies

Deadlock lösen

- alle beteiligten Prozess stoppen
- alle beteiligten Prozess auf Checkpoint zurücksetzen
- beteiligte Prozess der Reihe nach stoppen, bis Deadlock gelöst
- den beteiligten Prozessen Ressourcen wegnehmen, bis Deadlock gelöst ist

Strategie zur Wahl von Prozessen bei 3. und 4.

- Prozess, der am wenigsten CPU konsumiert hat
- Prozess, der am wenigsten Output produziert hat
- Prozess, der am wenigsten Ressourcen alloziert hat
- Prozess mit kleinsten Priorität
- Prozess, der die längste geschätzte Rechenzeit hat

Deadlock Strategie

Alle Strategien haben Vor- und Nachteile

- Kombination der Verfahren verwenden
- Vorschlag
 - Ressourcen in Klassen einteilen
 - Klassen linear anordnen
 - pro Klasse den besten Algorithmus verwenden

Dennoch die meisten Betriebssystem tun nichts!

Wenn Sie aber ein kritisches Echtzeitsystem entwickeln müssen

- z.B. Deadlock beim Airbus im Landeanflug ... 20m über ...

Interprocess Communication Kommunikationsmodelle

Versuch einer Klassifizierung

- Message Passing
 - Austausch von Nachrichten
- Shared Objects
 - gemeinsamer Zugriff auf Objekte
- Object Streams
 - serieller Strom von Objekten
 - Objekte: ein oder mehrere Bytes

Message Passing

Sehr häufig verwendetes Verfahren

- zwischen Prozessen auf einem Rechner
- in verteilten Rechnersystemen
- Synchronisation implizit

Grundfunktionen

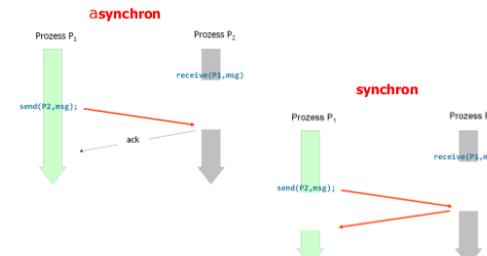
- `send(dest, message)`
- `receive(src, message)`
- beide Funktionen können blockieren

Auch geeignet für

- reine Prozess-Synchronisation
- Implementation eines Mutex

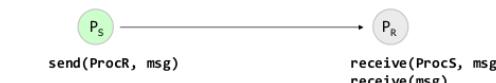
Mutex = mutual exclusion = Gegenseitiger Ausschluss

Synchrone / Asynchrone Kommunikation



Message Passing: Adressierung

Direkte Adressierung



Indirekte Adressierung



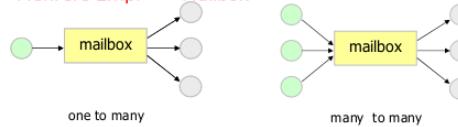
Mailbox, Port: Im Wesentlichen eine Queue.

Message Passing: indirekte Adressierung

Genau 1 Empf => Port



Mehrere Empf => Mailbox



Message Passing: Mailbox

Wem gehört das Port resp. die Mailbox?

Port

- gehört normalerweise dem Empfänger-Prozess
- wird von ihm erzeugt**
- wenn Prozess stirbt, wird Port zerstört

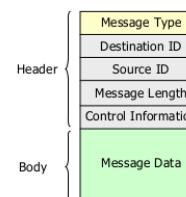
Mailbox wird i.d.R. beim OS angefordert

- Mailboxen werden i.A. beim Betriebssystem angefordert
 - Mailbox gehört meist dem Betriebssystem
 - z.B. Unix Message Queues**
- Mailbox kann aber auch Prozess gehören
 - wird zerstört wenn Prozess stirbt

Message Passing: Nachrichten

Aufbau von Nachrichten (Datenstruktur)

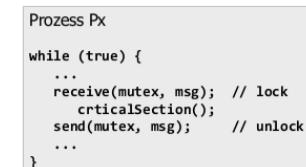
- Header¹⁾
 - Nachrichtentyp
 - Ziel- und Quellenadresse
 - Länge
 - Steuerinformation
 - Aktion bei voller Mailbox
 - Sequenznummer
 - Priorität
- Body
 - Daten



Mutex mit Message Passing

Beteiligte Prozesse haben gemeinsame Mailbox

- Lesen blockiert
- es wird maximal eine Nachricht abgelegt
- Hauptprogramm initialisiert Mailbox
- Mailbox heißt mutex



Shared Memory vs. Message Passing

Man muss selbst dafür sorgen =

Shared Memory

- muss beim Betriebssystem angefordert werden
- keine implizite Synchronisation** beim Zugriff
 - Synchronisation z.B. mit Semaphoren
- Zugriff sehr schnell: Speicherzugriff
- nur in Shared Memory Systemen verfügbar

Message Passing

- Mailboxes müssen beim Betriebssystem angefordert werden
- implizite Synchronisation**
 - einfach zu handhaben
- langsamer als Shared Memory
- in verteilten Umgebungen verfügbar
 - MPI: Message Passing Interface

Unix/Linux: Shared Memory, Semaphore, Messages

speziell auch für nicht verwandte Prozesse

Posix IPC Ressourcen

- erzeugen und/oder öffnen
 - `shm_open()`, `mq_open()`, `sem_open()`
 - "Zugriff" über Name, z.B.
 - `fd = shm_open("/tmp_shmr1", O_CREAT | O_RDWR, 0700);`
- unterschiedliche Rückgabewerte
 - Shared Memory: File Deskriptor
 - Semaphor: Semaphor Objekt, Pointer
 - Message Queue: Message Queue Objekt, kein Pointer

Message Queues (Posix)

Message Queue erzeugen

```
mqd_t q1;
q1 = mq_open("/tmp/que1", O_CREAT | O_RDWR, 0700, NULL);
```

System Calls

- send Message

```
rv = mq_send(q1, buf, len_s, prio);
```

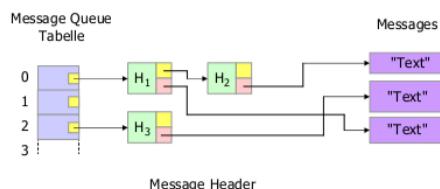
- receive Message → älteste Message mit höchster Priorität

```
rv = mq_receive(q1, buf, len_r, NULL);
```

... Message Queues (Linux)

Message Queues

- Organisation der Datenstrukturen im Kernel



Signale (POSIX)

CTRL-C → SIGINT

Ähnlich wie HW-Interrupts

- keine Priorisierung
- Signalverarbeitung sobald Prozess in den Zustand **Running** geht
- Signal ruft entweder
 - Defaultaktion (meist Prozess terminieren, z.T. ignorieren)
 - oder Signal Handler (Benutzer-Prozedur)
- blockierte Signale
 - mehrfaches Eintreffen der Signale → nur einmal gespeichert
- Rückkehr aus Handler
 - für Prozess wie normale Rückkehr aus Prozedur

Beispiel

- Shell Befehl kill PID sendet **SIGKILL** an Prozess mit Prozessidentifikation PID → Prozess terminiert

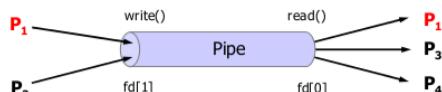
Pipes (Posix)

Pipe

- FIFO-Buffer mit fester Größe
- vom Kernel verwaltet
- Datenaustausch zwischen verwandten Prozessen
- halfduplex

Zugriff synchronisiert

- Reihenfolge nicht garantiert



Zwei Typen von Pipes

- Pipes und Named Pipes resp. Ffos

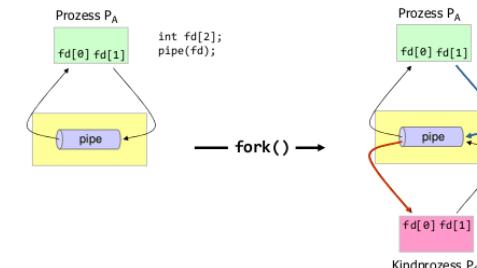
unnamed Pipes

- ein Paar von Filedeskriptoren: einer zum Lesen, einer zum Schreiben
- kann nur von verwandten Prozessen verwendet werden

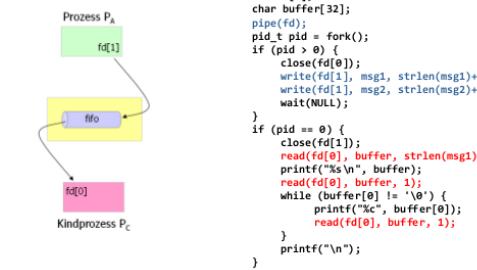
named Pipes

- ein File vom Typ FIFO wird erzeugt
- Zugriff für beliebige Prozesse, weil File

Einrichten einer Pipe



Beispiel (vereinfacht)



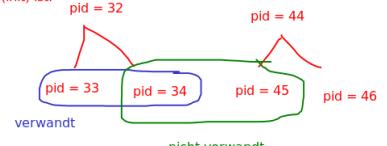
31

... Named Pipe / Fifo (auch für nicht verwandte Prozesse)

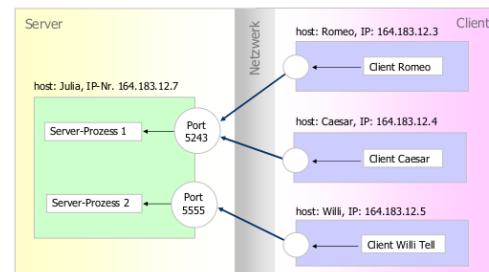
Beispiel (vereinfacht)

```
int main(void) {
    char *msg1 = "A first message";
    char *msg2 = "Let's try a second time";
    char *fifo = "/tmp/myFifo";
    char buffer[32];
    mkfifo(fifo, 0770);
    int fd = open(fifo, O_RDWR);
    pid_t pid = fork();
    if (pid == 0) {
        write(fd, msg1, strlen(msg1)+1);
        write(fd, msg2, strlen(msg2)+1);
        wait(NULL);
        close(fd);
        unlink(fifo);
    }
    if (pid == 0) {
        int fd = open(fifo, O_RDONLY);
        read(fd, buffer, strlen(msg1)+1);
        printf("%s\n", buffer);
        read(fd, buffer, 1);
        while (buffer[0] != '\0') {
            printf("%c", buffer[0]);
            read(fd, buffer, 1);
        }
        printf("\n");
    }
}
```

Prozesse sind verwandt, wenn sie denselben Elternprozess haben und dieser nicht 1 (init) ist.



TCP/IP - Sockets



Typen:

- TCP/IP Sockets: Netzwerk
- Unix Domain Sockets: Lokal
- vor allem Client-Server Kommunikation
- vielfältig und flexibel (dafür auch komplizierter)
- verbindungsorientierte und -lose Kommunikation

ZHAW, BSc, M. Thaler April 13

Systems Calls für das Aufsetzen von Sockets

- socket()**: erzeugt Socket, nicht initialisiert
- bind()**: Socket wird an Adresse (IP Adresse, Port Nummer) gebunden INADDR_ANY: nur Port Nummer von Bedeutung
- listen()**: Initialisiert Queue für Server, für Zwischenpufferung von Verbindungswünschen durch Clients
- accept()**: baut Verbindung mit Client aus Queue auf
- connect()**: Client fordert Verbindung zu Server an resp. baut auf

Systems Calls für den Datenaustausch

- read()**: lesen von verbindungsorientiertem Socket
- recv()**: ditto
- write()**: schreiben auf verbindungsorientierten Socket
- send()**: ditto
- recvfrom()**: Daten von verbindungslosem Socket empfangen
- sendto()**: Daten über verbindungsloses Socket senden

Port Nummern

- 1-255 reserviert: wohlbekannt, für Standardanwendungen
- 1-2023 privilegiert: Superuser Prozesse für unixspezifische Anwendungen
- 1024-5000 kurzlebig: automatische Vergabe an Benutzerprozesse, die auf bestimmte Nummer angewiesen sind
- 5001-65535 benutzerdefiniert: können von nichtprivilegierten Serverprozessen beansprucht werden und an Clients weitergegeben werden

Semaphore (Posix)

Semaphor erzeugen und initialisieren

- named Semaphore

```
sem_t s1;
s1 = sem_open("/tmp_sem1", O_CREAT|O_RDWR, 0700, value);
```

- unnamed Semaphore

```
sem_t s1;
rv = sem_open(&s1, pshared, 0700, value);
```

System Calls

- Semaphor schließen

```
rv = sem_wait(&s1);
```

- Semaphor öffnen

```
rv = sem_post(&s1);
```

Memory Management

Was gehört zu Memory Management ?

Relocation

- verschieben eines Prozesses an beliebigen Ort im Speicher

Protection

- verhindern, dass sich Prozesse gegenseitig beeinflussen

Sharing

- gemeinsam Speicherbereiche zur Verfügung stellen

Logical Organisation

- logischer Adressraum, lineare Folge von Bytes (Words)

Physical Organisation

- physische Realisierung mit Haupt- und Sekundärspeicher

Logische/physikalische Adressen

Logische Adresse

- Referenz auf Speicherplatz, unabhängig von Speicherorganisation

Physikalische Adresse

- Referenz auf physikalischen Speicherplatz

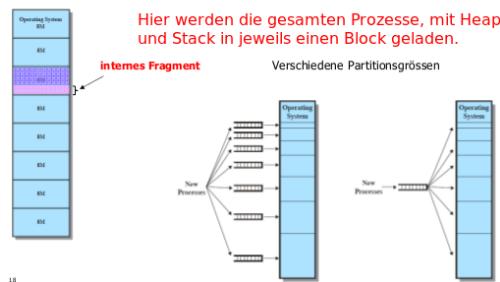
Compiler

- erzeugen Code mit relativen Adressen



Fixed Partitioning

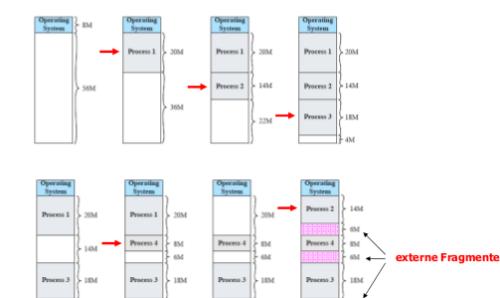
Gleiche Partitionsgrößen



Gleiche Partitionen

- jedes Programm, egal wie gross, belegt eine Partition
- Partition nicht vollständig gefüllt: wird internal fragmentation genannt
- Nutzung des Hauptspeichers ineffizient

Dynamic Partitioning



Grösse und Anzahl der Partitionen variabel

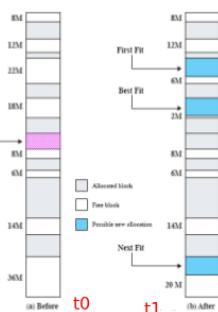
- jedem Prozess soviel Speicher zuweisen, wie er benötigt
- Problem: externe Fragmentierung
 - im Hauptspeicher bilden sich mit der Zeit Löcher
 - Grund: Prozesse werden ausgelagert und können nicht immer durch gleich grosse Prozesse ersetzt werden

- compaction** Algorithmus notwendig, um Prozesse zu verschieben

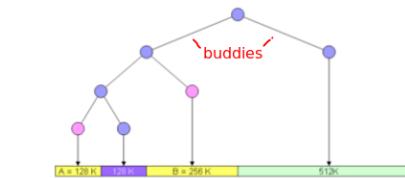
Placement Algorithmen

Gesucht: Block mit 16MByte

- welcher freie Block soll alloziert werden?



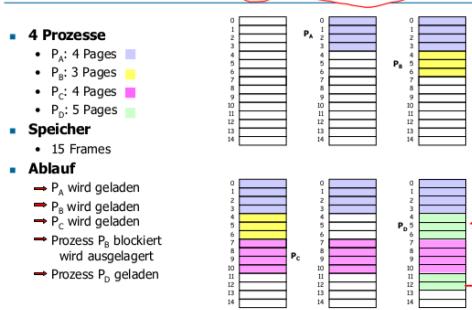
- es werden solange Blöcke halbiert, bis ein Block minimaler Grösse zur Verfügung steht
- Daten zum den Freien Blöcken lassen sich mit einem Binärbaum darstellen
- effiziente Algorithmen verfügbar (siehe Lit., z.B. Stallings)



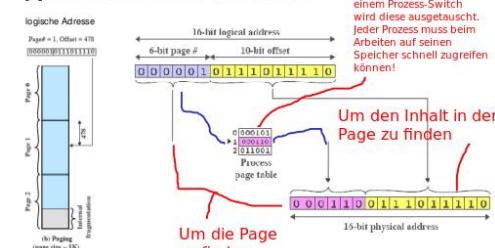
Paging

- Prozesse in Blöcke gleicher Länge aufteilen → Pages
- Speicher auch in gleich grosse Blöcke aufteilen → Frames
- beliebige Zuweisung: Pages ↔ Frames
 - Konsequenz: Prozess müssen nicht zusammenhängend im physikalischen Speicher stehen

Paging: Beispiel



Wie wird eine logische Adresse in eine physikalische Adresse übersetzt?



- Jede logische Adresse (N-Bits) besteht aus zwei Feldern

- einer Pagenummer
 - k höherwertige Bits der Adresse
 - Offset vom Beginn der page
- Pagegröße = 2^m → immer Zweierpotenz
- Page Größe = 1KB

Page-tabelle

- jeder Prozess hat eine eigene Pagetabelle
- enthält für jede Pagenummer die entsprechende Framenummer
- die Pagenummer ist dient als Referenz (Adresse) zum Lesen der Framenummer

Anmerkungen

- die letzte Page (Frame) ist nicht unbedingt "voll" -> interne Fragmentierung
- Paging ist für Anwender transparent

Daten zum Beispiel

- logische Adresse 16 Bit
- Pagenummer 6 Bit -> maximal 64 Pages
- Offset 10 Bit -> page size = 1KB
- Einträge in der Tabelle <= 64

Segmentation

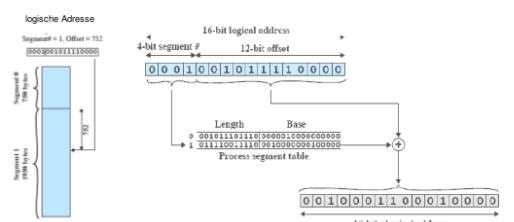
Anwenderprogramme aus mehreren Segmenten

- Programm
- Daten
- Stack

Segmente

- verschiedene Grösse
- zusammenhängender Adressraum
- können im Speicher an beliebigem Orte plaziert werden
- keine interne Fragmentierung, dafür extern
- für Programmierer sichtbar

Wie wird eine logische Adresse in eine physikalische Adresse übersetzt?

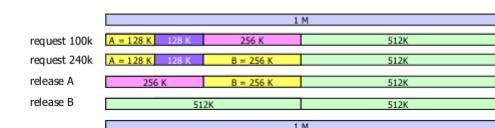


Buddy System

sehr schnell! linux verwendet das (oft für buffer).

Kompromiss

- zwischen fixed und dynamic Partitioning
- schneller Allokations- und Deallokationsalgorithmus
- Unix Kernel verwendet modifiziertes Buddy System



Paging vs. Segmentation

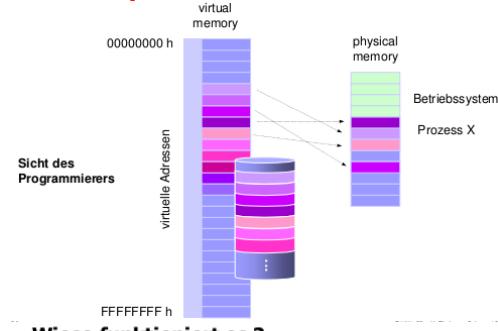
Paging

- für Programmierer transparent
- kleine, interne Fragmentierung
- Adressübersetzung: Tabellen-Lookup
- Schutz auf Ebene von logischen Segmenten

Segmentation

- für den Programmierer sichtbar
- externe Fragmentierung
- Adressübersetzung: Tabellen-Lookup und Addition
- Schutz von physikalischen Segmenten

Virtual Memory



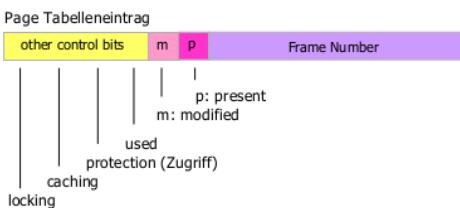
Wieso funktioniert es?

- Virtual Memory basiert auf **Lokalitätprinzip**

Virtual Memory: Adressübersetzung

Page Table aufwendiger

- zusätzliche Einträge notwendig



Page Table Organisation

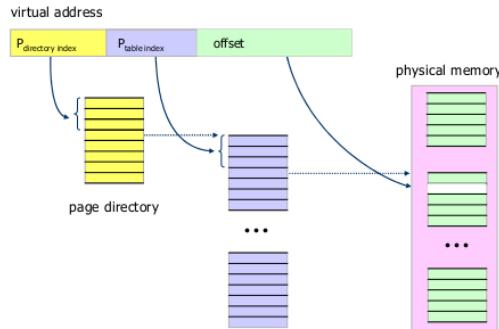
Problem mit Page Tabellen

- Page Tabellen werden gross
- Beispiel: adressen 32 Bit
page size 4 KByte
page table 2²⁰ Einträge
(64 Bit)
(4 KByte)
(2³² Einträge)

Mögliche Abhilfen

- Page Table im virtuellen Speicher
- Multilevel Organisation
 - page directory: hierarchische Organisation
- Hashed Page Table
 - Zugriff auf Tabelleneintrag über Hashwert
- Inverted Page Table
 - pro Page Frame ein Eintrag

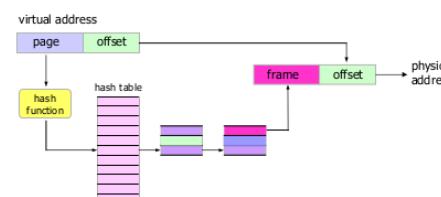
Multilevel Organisation



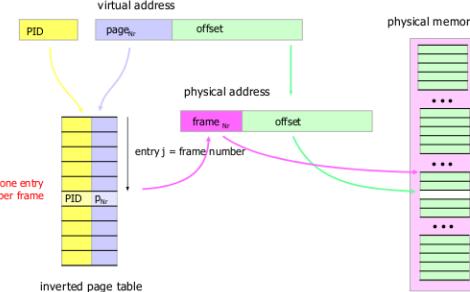
Hashed Page Table

Page Nummer wird als Hash Wert verwendet

- geeignet für Invertierte Page Tabelle
- Adressräume > 32 Bit
- verschiedene Page Nummern → gleicher Hash Wert



Konzept: Invertierte Page Tabelle



TLB: Translation Lookaside Buffer

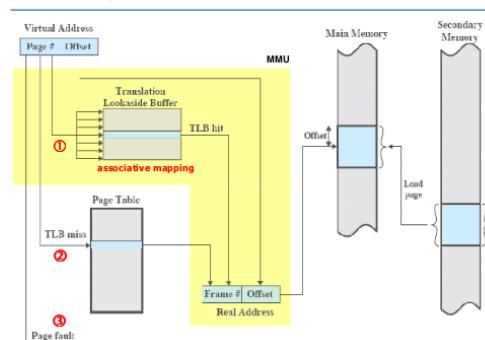
Problem

- jede Speicherreferenz benötigt bis zu zwei Speicherzugriffe
 - Frame Nummer aus Page Tabelle
 - Zugriff auf Daten
- bei Hash Tabellen zusätzliche Zugriffe
 - gleicher Hash Wert für verschiedene Pages
- invertierte Page Tabellen
 - zusätzlicher Zugriff auf Prozess ID

Abhilfe

- spezieller Hardware Cache für Page Tabellen
- enthält kürzlich verwendete Page Tabellen Einträge
- pro Prozess aufgesetzt
- wird TLB (Translation Lookaside Buffer) genannt

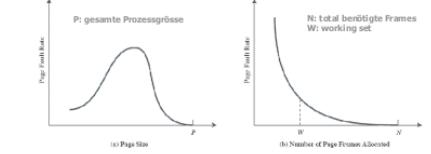
TLB: Zugriff



Page Size

Genaue Angaben schwierig

- z.T. widersprüchliche Anforderungen



	PowerPC	UltraSPARC	Pentium	ITanium	AMD64	IBM 370/XA, 370/ESA	IBM AS/400	VAX	DEC Alpha	MIPS	4 KBytes
virtual memory	4 KBytes	8 Kbytes bis	4 Kbytes oder	4 Kbytes bis	4 KBytes bis	4 MBytes	256 MBytes	512 Bytes	512 Bytes	512 Bytes	512 Bytes
physical memory					64 KBytes						

Memory Management Software Replacement Policy: Optimal

Ersetzt Page, die am spätesten referenziert wird

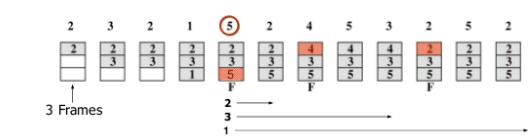
- nicht implementierbar

Aber

- beweisbar, dass minimale Anzahl Page Faults
- gut für Vergleiche

Beispiel mit 3 Frames

- Page Referenzen: 2 3 2 1 5 2 4 5 3 2 5 2



Replacement Policy: Least Recently Used

Ersetzt die am längsten nicht referenzierte Page

- Lokalitätprinzip: wird wahrscheinlich nicht mehr referenziert
- fast so gut wie optimal

Aber

- Implementation aufwendig
 - Time Stamp oder Usage Counter notwendig

Beispiel



Replacement Policy: FIFO

Pages Frames in zirkulärem Buffer angeordnet

- älteste Page wird ersetzt: Policy FIFO
- Problem: auch oft referenzierte Pages werden ausgelagert

Aber

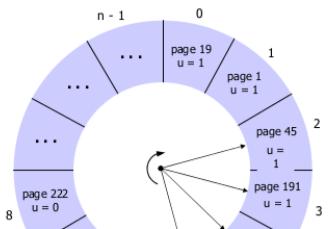
- Implementation sehr einfach
- schlechte Resultate

Beispiel



Replacement Policy: clock

Für Page 727 einen Frame suchen



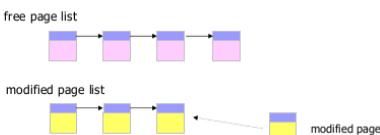
Page Buffering

Bis jetzt

- Frames durch neue Pages ersetzt

Page Buffering: VAX

- Replacement Policy: FIFO
- nicht alle Frames stehen für den Gebrauch zur Verfügung
- BS unterhält zwei Listen: Free Page List und Modified Page List



Load Control: process suspension

Welche Prozesse sollen suspendiert werden, wenn zu viele Prozesse im Speicher?

Sechs Möglichkeiten

- Prozess mit tiefster Priorität
- Prozess mit (vielen) Page Faults
- der am längsten nicht aktivierte Prozess
- Prozess mit kleinstem Resident Set
- größter Prozess
- Prozess mit dem längsten Verarbeitungsfenster

Prozess mit tiefster Priorität

- implementiert eine Scheduling Policy Entscheidung (hat nichts mit Performance zu tun)

Prozess mit (vielen) Page Faults

- große Wahrscheinlichkeit, dass
 - Working Set nicht resident ist
 - Prozess blockiert wegen Page Fault

der am längsten nicht aktivierte Prozess

- hat mit grosser Wahrscheinlichkeit den Working Set nicht vollständig im Speicher
- Prozess mit kleinstem Resident Set

Prozess mit kleinstem Resident Set

- erfordert am wenigsten Aufwand, den Prozesse aus- und wieder einzulagern
- größter Prozess

gibt am meisten Frames frei

- mit grosser Wahrscheinlichkeit muss kein zusätzlicher Prozess ausgelagert werden

Prozess mit dem längsten Verarbeitungsfenster

- implementiert etwas, wie ein "SJF" Scheduling Policy

Implementationsaspekte

I/O Page locking

- Prozess wartet auf Eingabedaten (System Call)
 - Page mit I/O-Buffer kann aus System ausgelagert werden
 - z.B. durch Replacement Strategie mit global scope
 - oder Prozess wird geswapped
- Lösungen
 - Pages mit I/O Buffer in Speicher "locken"
 - I/O Buffer im Kernel verwalten und Daten später in den User Space kopieren

Shared Memory

- ähnliche Probleme wie mit I/O Pages
- aber: mehrere Prozesse sind involviert

Instruktionen

- Instruktion, die Page Fault erzeugt, muss wiederholt werden
 - Problem bei variabel langen Instruktionen
 - Betriebssystem muss wissen, wo Instruktion beginnt
 - ev. müssen zusätzlich Autoinkrement und -Dekrements rückgängig gemacht werden
 - falls keine Hardwareunterstützung
 - zusätzlicher Overhead bei Page Replacement

Paging Daemon

- Page Replacement kann effizient durch Hintergrundprozess (Daemon) ausgeführt werden
 - z.B. periodisch oder während Lastpausen

Betriebssystemaufgaben mit Paging

- Prozesserzeugung
 - aufsetzen und initialisieren der Page Table und Swap Area
- Prozessausführung resp. Prozessaktivierung
 - MMU rücksetzen und Inhalt des TLB entfernen
 - entsprechende Page Table aktivieren
- Page Fault
 - bestimmen welche Page geladen werden muss
 - Neustart der Instruktion
- Prozessterminierung
 - Page Table und Swap Area entfernen
 - Shared Pages beibehalten (Eintrag für Prozess entfernen)

FileSystems

Begriffe

Field

- kleinste speicherbare Einheit: Byte, Word, String, etc.

Record

- eine Sammlung von zusammengehörigen Feldern, z.B. 512 Bytes, Personalien (Geburtsjahr, Name, Vorname, etc.), etc.

File

- eine Menge von gleichen oder ähnlichen Records
- eine Menge von gleichen Feldern: z.B. Unix → Bytes
- etc.

Database (Datenbank)

- eine Sammlung von zusammengehörigen Daten, besteht i.A. aus mehreren Files

... das File Konzept

Funktionen resp. Methoden

- erzeugen / öffnen `create() / open()`
- löschen / schliessen `delete() / close()`
- lesen / schreiben `read() / write()`
- positionieren (Record) `seek()`

File öffnen → Filename

- Filename: relativ zu aktuellem Verzeichnis oder absoluter Pfad
- gibt `file descriptor` resp. `file handle` zurück

```
int fd = open("./tmp.txt", O_RDWR | O_CREAT);
```

Zugriff → File Handle

- immer über Descriptor oder Handle

```
read(fd, buffer, 32);
```

File Attribute

Beispiel Unix / Linux

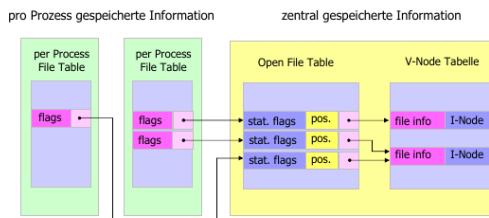
ls -al LateFile

Protection	Besitzer	Datum, Zeit	Links	Grösse	Name (Typ)
-rw-rw-r--	1 tha tha	531 April 1 00:01			LateFile

Informationen zu Files

Wie werden Informationen zu Files verwaltet?

- z.B. beim Unix/Linux File System (vereinfacht)



Notwendige File Typen (für Betriebssystem)

Directories (Verzeichnisse)

- Informations zu Verzeichnissen
- Referenzen auf Dateien und Verzeichnisse
- System Files: Änderungen nur durch BS

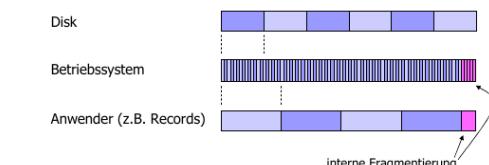
Regular Files (Dateien)

- Benutzerdaten
 - Programme (ausführbare Files extra markiert: Unix/Linux)
 - Daten
 - etc.

File Organisation

Interne File Strukturen

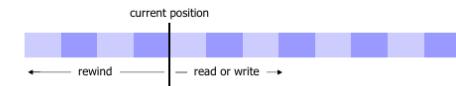
- Disk (physikalisch)
 - Betriebssystem
 - Anwender
- Blöcke, meist mit 512 Bytes
oft Sequenz von Bytes, z.B. Unix, Windows
Sequenz von Bytes oder Menge von Records



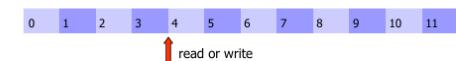
Zugriffsmethoden

Zwei Zugriffsmethoden auf Records

- sequentieller Zugriff: **sequential access**



- direkter Zugriff **random access (direct access)**



Record = Byte oder eine beliebige Struktur.

Zugriffsrechte

Zugriffsschutz basiert meist auf Benutzeridentität

- Zugriff durch mehrere Benutzer
 - Liste mit Zugriffsrechten für Benutzer

- traditionell drei Typen von Anwenderrechten

- Besitzer (Unix: `user`)
- Gruppe (Unix: `group`)
- Alle (Unix: `other`)



- heute auch

- ACL: Access Control Lists, z.B. NTFS, Linux (2.6)**

ACLs: Konsequenzen

- Verzeichniseintrag hat keine feste Länge
- aufwendig, weil Anzahl Benutzer nicht im voraus bekannt
- ! Wartung: Benutzer wechselt z.B. Abteilung → Zugriffsrechte?

File Management: Journaling

Journaling

- verhindert Inkonsistenzen des File Systems bei Crashes
- sämtliche Änderungen am File System mit Transaktionen
- moderne Betriebssysteme unterstützen Journaling
 - Linux: ext3, reiserfs
 - Windows: NTFS

z.B. ext3 Journaling-Modi

- ext3 unterstützt 3 Modi
 - ordered** nur Metadaten (Info zu Files, aber nicht Inhalt) werden im Journal gespeichert
 - writeback** nur Metadaten, ordered, aber nicht synchron
 - journal** auch Daten im Journal eingetragen wegen 2-fachem Speichern deutlich langsamer

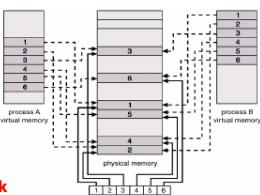
File Management: Memory Mapped Files

Memory Mapped Files

- File wird in das Virtual Memory des Prozesses abgebildet
- Filezugriff = Speicherzugriff
- File wird mit "demand paging" in phys. Speicher gelesen

Vorteile

- einfacher Zugriff
 - read, write, etc.**
- Speicherzugriff
- File Sharing
 - schnell
 - einfach
 - ! Konsistenzsemantik**



Naming

Filezugriff über Namen

- jedes File muss einen eindeutigen Namen haben
- baumartige Verzeichnisstrukturen unterstützen diese Forderung:
 - das File ist eindeutig durch seinen Pfadnamen definiert
 - statt Filenamne wird oft path verwendet
- z.B. Unix: /home/tha/work/proc/prak2/test.cc

Pfadnamen mühsam ...

→ pro Anwender resp. Prozess: ein Working-Directory

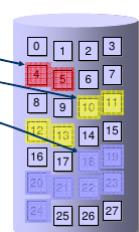
Working Directory

```
$ cd /home/tha/work/proc/prak2 → working directory
$ cat ./test.cc resp. cat test.cc
```

Contiguous Allocation

Alle Blöcke zusammenhängend hintereinander speichern

- Directory Eintrag
File count 4 Start 2 Länge 4
mail 10 mail 4
a.exe 18 a.exe 7

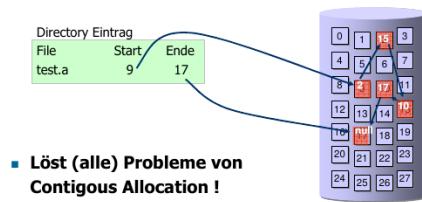


Unterstützt

- sequential access
- direct (random) access

Linked Allocation

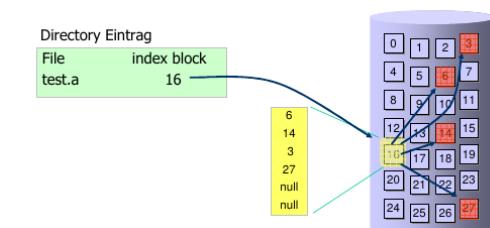
Die Blöcke eines Files bilden eine verkettete Liste



Löst (alle) Probleme von Contiguous Allocation!

Indexed Allocation

Speichert die Blockzeiger in einem Index Block



Allocation: Performance

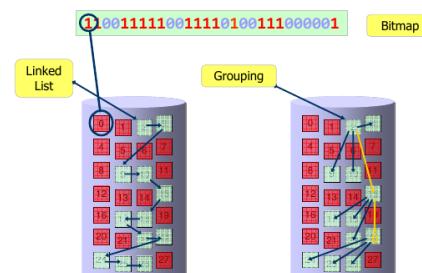
Aussage nicht ganz einfach, hängt von Benutzung des Filesystems ab

- dominiert Sequential Access?
 - eher Linked Allocation
- dominiert Direct (Random) Access?
 - eher Contiguous Allocation oder Indexed Allocation (Aufwand)

Oft anzutreffen

- mixed allocation
 - Contiguous Allocation für kleine Files
 - Indexed Allocation für grosse Files
- Clustering
 - kleine Files automatisch Contiguous Allocation
- CPU-Leistung lässt heute eher aufwendige Algorithmen zu

Free Space Management



Freie Blöcke auf dem Disk müssen verwaltet werden
BS führt Liste mit freien Blöcken

- Free Space List (nicht notwendigerweise eine Liste)

Lösungsmöglichkeiten

- Bit Vector resp. Bit-Map
 - jeder Block entspricht einem Bit in der Liste, z.B. MacOS
 - effizient wenn im Speicher
 - aber grosser Speicherbedarf: 6 GB Disk mit 512Byte Blöcke: ~1.5 MByte Speicher
- Linked List
 - freie Blöcke als Linked List
 - Zeiger auf ersten Block im Speicher
 - aufwendig die Free List abzusuchen (jedoch nicht häufig)
- Grouping
 - im ersten freien Block die Adressen von N freien Blöcken gespeichert
 - letzte im Block Adresse zeigt auf nächsten Block mit freien Blöcken
 - z.B. Unix

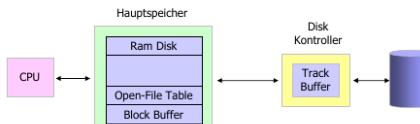
Effizienz und Performance

Effiziente Diskausnutzung

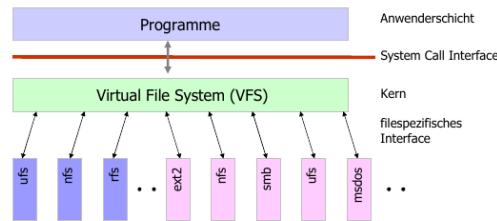
- durch Allocation und Verzeichnis-Algorithmen bestimmt

Performance

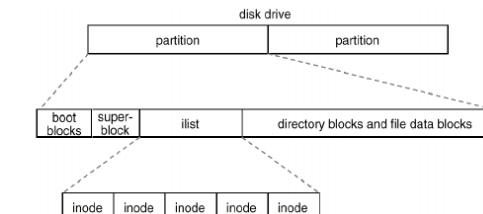
- Disks sind relativ langsam: effizienter Zugriff ein Muss
 - on-board cache: ganze Tracks zwischenspeichern
 - Hauptspeicher: disk cache Kontrolle durch Betriebssystem ram disk Kontrolle durch Anwender



Unix Filesystem: Benutzersicht



Grobstruktur des Filesystems



FileSysteme Slides bis Folie 50/65.

Resident Set Management

Fragestellungen

- aus welcher Menge einen neuen Frames wählen?
- wie viele Frames einem Prozess zuweisen?

Resident Set Size / Replacement Scope

- die Anzahl zugewiesener Frames / wo neuen Frame holen

local replacement	global replacement
fixed allocation	• nicht möglich
variable allocation	• fixe Anzahl Frames pro Prozess • zu ersetzende Pages gehören dem aktuellen Prozess
Virtual Memory	• Anzahl zugewiesener Frames ändert von Zeit zu Zeit • zu ersetzende Pages gehört dem aktuellen Prozess

- nicht alle Pages eines Prozesses müssen / können im Hauptspeicher stehen
- bis jetzt beantwortet: welche Page aus einer vorgegebenen Menge ersetzen
- neue Fragestellungen

- was ist die vorgegebene Menge (replacement scope)?
 - local replacement nur zum Prozess gehörende Frames
 - global replacement gesamthaft verfügbare Frames
- wie viele Frames einem Prozess zuweisen?
 - fixed allocation feste Anzahl Frames zuweisen
 - variable allocation Anzahl Frames ändert mit Zeit

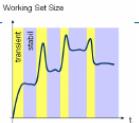
Linux: Variable Allocation / Global Scope

Windows 2k: Variable Allocation / Local Scope

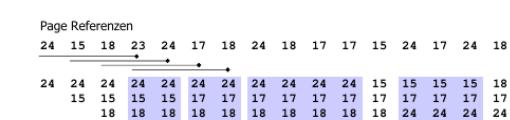
Working Set

Working Set: $W(t, \Delta T)$

- referenzierte Frames innerhalb eines Zeitfensters ΔT
- $W(t, \Delta T)$ konstant während gewisser Zeit



Beispiel: $W(t, \Delta T)$ für $\Delta T = 4$



Resident Set Anzahl aktuell zugewiesener Frames.

Working Set In währnd DeltaT zugewiesene Frames.

Wenn voll: Das am längste nicht mehr gebrauchte raus.

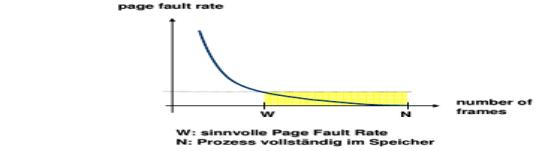
Wichtig: Wenn DeltaT = 2, dann muss eines, dass zweimal nicht mehr gebraucht wurde, entfernt werden, auch wenn der Speicherplatz für nichts sonst gebraucht wird! **Das Hinzufügen ist Nummer 1!**

Page Fault Frequency

Working Set Size

- bestimmen über Page Fault Rate

Page Fault Frequency



IO

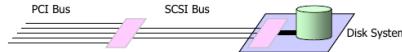
Grosse Gerätevielfalt, aber nur wenige Konzepte

Anschluss der Geräte über

- **Ports** ein Gerät, Datenstrom, z.B. serielle Schnittstelle
- **Busse** mehrere Geräte, mehrere Drähte und ein Protokoll
- **Daisy Chain** Geräte in Serie angeschlossen, arbeitet wie Bus

I/O-Controller

- Steuerelektronik für Port, Bus oder Gerät



Typische Port Konfiguration: 4 Register

- Statusregister
- Kontrollregister
- DataIn
- DataOut

CPU

I/O

Daten Register

Status/Control Register

Adressen

Control

Interface Logik zu externem Gerät

Daten

Status

Control

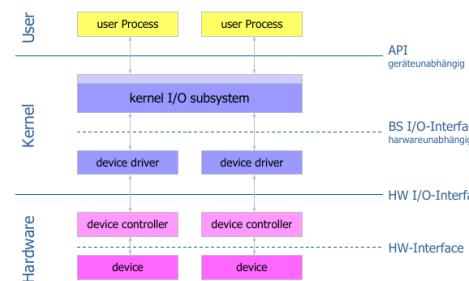
Interaktion zwischen Prozess(or) und Geräten

- Polling busy wait (synchronous)
- Interrupt Interrupt Handler (asynchronous)
- DMA Datentransfer im Hintergrund (asynchronous)

Ziele von I/O Software

- Geräteunabhängigkeit gewährleisten
 - Abstraktion, z.B. Disk Drive
- einheitliche Namensgebung
 - Unix/Linux: alles ist ein File
- Fehlerbehandlung
 - möglichst nahe an HW
- asynchroner- / synchroner I/O
 - Blocking System Calls bei asynchronem I/O
- Buffering
 - Vorverarbeitung, Echtzeitprobleme
- Sharing
 - gemeinsam genutzte Geräte, z.B. Disks

I/O-Architektur

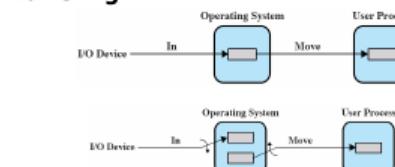


- Architektur: Schichtenmodell
- Wichtigste Komponente: **Device Drivers**
 - standardisierte Schnittstelle zum Kernel I/O-Subsystem
 - intern an Gerätedetails angepasst
 - Initialisierung
 - Lesen und Schreiben der Daten
- Keine vereinheitlichten Schnittstellen für Device Driver in Betriebssystemen
 - jedes Betriebssystem benötigt andere Treiber
- Logical I/O
 - Gerät als logische Ressource betrachtet, Schnittstelle zu Benutzerprozess (API): read/write, put/get
- Device I/O
 - angeforderte Operationen in entsprechende Sequenzen von I/O Instruktionen umwandeln, Buffering
- Scheduling and Control
 - steuert Ablauf der Interaktion zwischen Gerät und Software, Interrupt Handling, I/O Status, etc.
- API
 - bei den meisten Betriebssystemen und Geräten
 - Zugriff wie auf Files: open(), read(), write(), close()

Der Kernel stellt i.A. folgende Services zur Verfügung

- I/O-Scheduling
- Buffering
- Caching, Spooling, Reservation
- Fehlerbehandlung

Buffering



I/O-Geräte: Charakteristiken (1)

Charakter (Byte) Stream vs. Block

- Stream
 - Datenübertragung Byte um Byte, z.B. serielle Schnittstelle
 - charakteristisches Verhalten: spontan erzeugter Input
- Block
 - Datenübertragung als Block von Bytes, z.B. Disk
 - charakteristisches Verhalten: random access

Netzwerkgeräte

- Zugriff meist über Sockets

Sequentiell vs. Random Access

- Datenübertragung (Gerät) bestimmt Reihenfolge, z.B. RS-232
- Anwendung bestimmt, welche Daten gelesen werden sollen, z.B. Diskblock (block device)

Blocking vs. Non-Blocking I/O

Physikalische Aktionen von I/O-Geräten i.A. asynchron

- nicht vorhersehbare, zeitvariable Ausführungszeiten

Trotzdem meistens Blocking Systems Calls

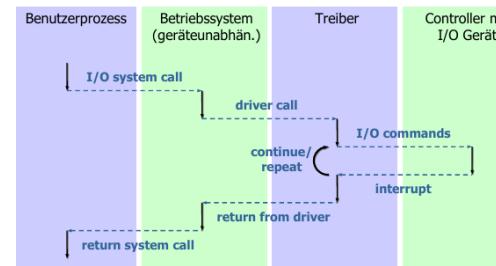
- einfacher zu verstehen

Typische Non-Blocking Operationen

- API zu Maus und Keyboard (interaktive Applikationen)
- Video Interface
 - Daten vom Disk lesen, gleichzeitig dekomprimieren und auf Display darstellen
 - Realisierung: Buffering und Multithreading (Kernel Threads)

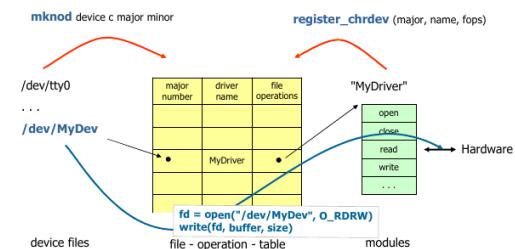
IO: Treiber

Linux: Ablauf Treiberaufruf



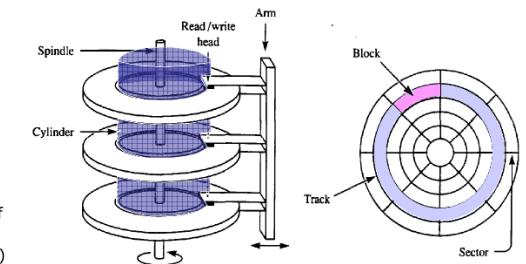
Linux: Modul ↔ Device (Treiber)

Character Device



IO: Disk

Aufbau eines Disks



Diskzugriff: Leistungsdaten (Mittelwerte)

Seek Time: t_s

- Definition: 1/3 der maximalen Positionierstrecke + mechanische Einschwingzeit
- mittlere Positionierzeit, typ. 7-12 ms

Rotational Latency: t_d

- Definition: Zeit für 1/2 Umdrehung
- Formel: $t_d = \frac{1}{2} t_R = \frac{1}{2} \cdot 60 / \text{rpm}$

Data Transfer Time: t_T

- Definition: $t_T = t_d \cdot b / N$
- b: zu transferierender Bytes, N: Bytes pro Track

Mittlere Zugriffszeit: $t_A = t_s + t_d + t_T$

