GhK

Pointer Swizzling: Unterbrechung erwünscht!

Lutz Wegner
Universität Gesamthochschule Kassel
FB Mathematik-Informatik
D-34109 Kassel

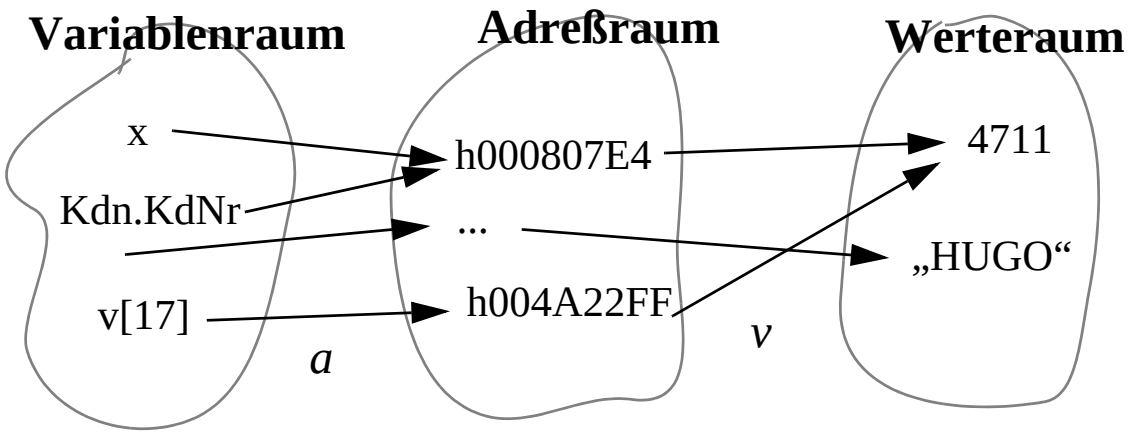
Inhalt

Pointer Swizzling: Modebegriff oder wichtige Technologie?

- etymologische Herkunft des Begriffes ist unbekannt
- Versuch einer Definition:
  „Pointer Swizzling: Technik der für ein Anwendungsprogramm oder eine Datenbank transparenten Wandlung von Plattenadressen in Hauptspeicheradressen (pointer).“
- bekanntgeworden durch OO-DBMS ObjectStore der Firma Object Design Inc.

Bisher

Compiler + Linker stellen eine Abbildung her zwischen Programmvariablen, bzw. Konstanten, und den Adressen, unter denen die Werte dieser Variablen und Konstanten zur Laufzeit gespeichert sind.



Zur Laufzeit besitzen die Variablen Werte, deren Lebensdauer entweder von der Lebensdauer der Prozeduren abhängt, in denen sie vereinbart wurden (*static allocation*, lokale Vereinbarungen) oder die als dynamische Variablen (*dynamic allocation*, Heap-Variable) nur bis zum Programmende existieren → sog. *volatile objects*, transiente Objekte.

Sollen Werte das Programmende überleben, müssen sie *persistent* gespeichert werden, d.h. in eine Datenbank oder Datei geschrieben werden.

Klassische Konzepte der persistenten Datenhaltung:

Explizites Laden/Speichern (read/write) von Werten

- Spezielle Größeneinheit des Transfers, ggf. stets gleiche Größe (records, Sätze)
- häufig Übersetzung der Werte, z.B. Integer aus Zweierkomplement nach ASCII-Folge
- ggf. Transfer über eingestreute SQL-Befehle und DB-Cursor, daher unterschiedliche Namensräume, Typen, usw.
- assoziative Anfragen auf Datenbank, aber elementweise Bearbeitung mit Ortsadressierung in Programmiersprache
- erhebliche Differenz (10^5) zwischen HS- und Plattenzugriff

→ **sog. impedance mismatch**

Neu: Persistente Programmiersprachen

Grundidee: Persistenz (= Fähigkeit als Wert über das Programmende hinaus zu leben) ist orthogonal zum Typ der Variablen, d.h. jede Variable (jedes Objekt) kann persistent gemacht werden.

- geht zurück auf M.P.Atkinson et al. 1983 (PS-Algol)
- deutlich mehr als der alte Versuch, eine Programmiersprache mit einer DDL zu verheiraten (Pascal/R und andere)
- heute häufig als Erweiterung von C++ realisiert, z.B. persistente Programmiersprache E (Richardson, Carey, Schuh, DeWitt, ... ab 1989)
- Im Idealfall Illusion des einstufigen Speichers

→ abgesehen von der Tatsache, daß ein DBMS über die persistente Speicherung hinaus noch weitere Dienste leistet (ACID Transaktionen), **eine sehr attraktive Idee!**

Transparente Persistenz - wie kann das gehen?

Temporäre Persistenz gibt es bei virtuellem Speicher

Read-only Persistenz gibt es beim Programmcode

→ **memory mapped files**

Ausnützen der unter UNIX verfügbaren
`shmat` (shared memory areas) und
`mmap` (memory map) Dienste: eine Datei wird in den Prozeß-Adreßraum abgebildet, bzw. ein Adreßsegment wird in eine (ggf. anonyme) Datei abgebildet.

mmap oder Systemprogrammierer sind die Hohepriester eines sehr niedrigen Kultes!

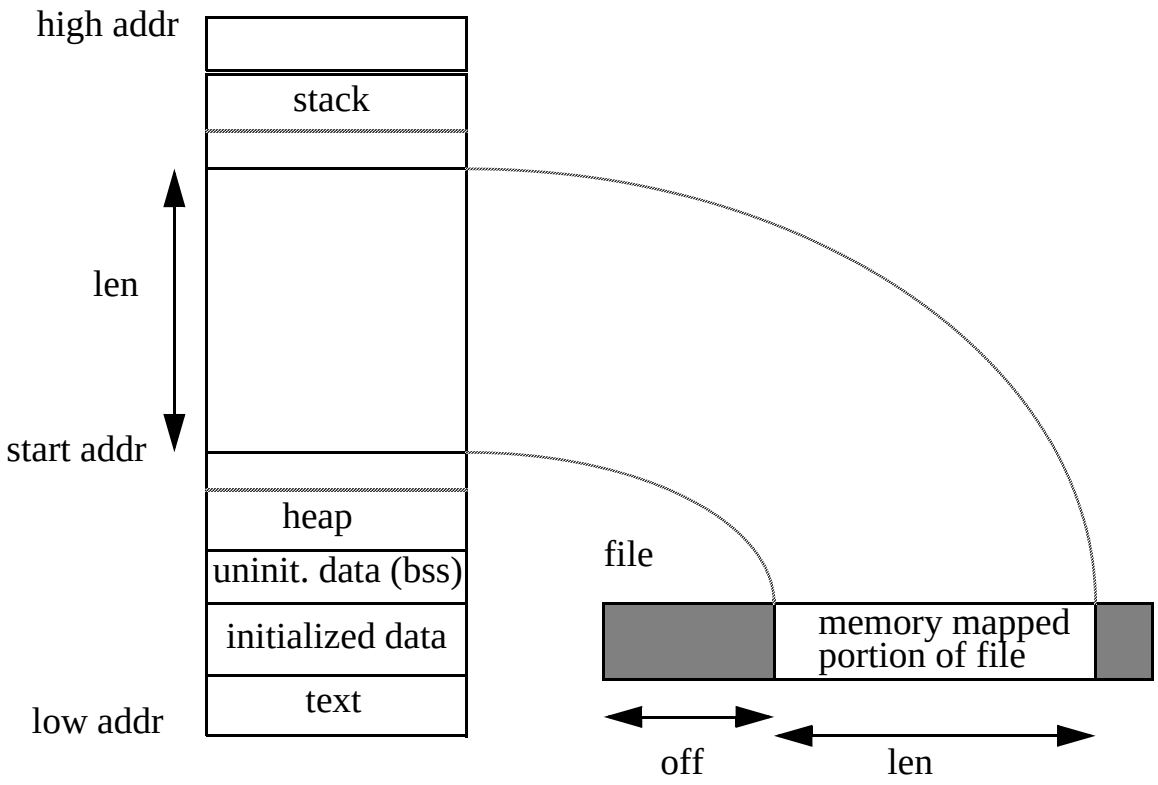
`caddr_t mmap(caddr_t` *addr*, `size_t` *len*, `int` *prot*, `int` *flag*, `int` *filedes*, `off_t` *off*);

liefert Startadresse der abgebildeten Datei wenn OK, sonst -1

addr HS-Startadresse, üblich ist 0 (System entscheidet)

len Länge in Bytes, sollte Mehrfaches von Seiten sein

off Startadresse in Datei, oft 0



`fork` vererbt Mapping, `exec` bewirkt Freigabe der Abbildung.

prot Schutz des Bereichs, ggf. ODER-Verknüpfung von

prot	Beschreibung
PROT_READ	Bereich kann gelesen werden
PROT_WRITE	Bereich kann beschrieben werden
PROT_EXEC	Bereich kann ausgeführt werden
PROT_NONE	kein Zugriff auf Bereich

Zitat AIX 3.2:

Note: The operating system generates a SIGSEGV signal if a program attempts an access that exceeds the access permission given to a memory region. For example, if the PROT_WRITE flag is not specified and a program attempts a write access, a SIGSEGV signal results.

flag MAP_SHARED
Speicheroperation (store) wirkt wie write-Op. Änderungen schlagen durch
MAP_PRIVATE
copy-on-write file (Schreiboperationen auf eine Seite führen zu einer Kopie der Seite im Paging-Bereich; Durchschlagen auf Datei muß explizit angefordert werden (fsync, msync), sonst gehen Änderungen nach Programmende verloren

filedes „file descriptor“, Datei muß vorher geöffnet werden, -1 für anonyme Datei

Schutz einzelner Seiten: Schlüssel zum Pointer Swizzling

Zitat AIX 3.2

```
#include <sys/types.h>
#include <sys/mman.h>
int mprotect (caddr_t addr,
              size_t len, int prot)
```

Description
The `mprotect` subroutine modifies the access protection of a mapped file region or anonymous memory region created by the `mmap` subroutine.

→ Seiten eines mit `mmap` im HS angelegten Bereichs können selektiv geschützt werden, auch ohne daß die Seite bereits physisch im HS oder im swapspace existiert. Zugriff kann Schutzverletzung auslösen: `SIGSEGV` - segmentation violation

→ I/O mittels `mmap`-Datei ist sehr schnell, geht aber nicht für *special files*, also Geräte, I/O-Puffer, etc.

Weil Sie schon immer mal einen Interrupt-Handler für UNIX schreiben wollten ...

Mit `signal(...)`, bzw.

```
int sigaction(int SignalNr,
        struct sigaction *Action,
        struct sigaction *OAction)
```

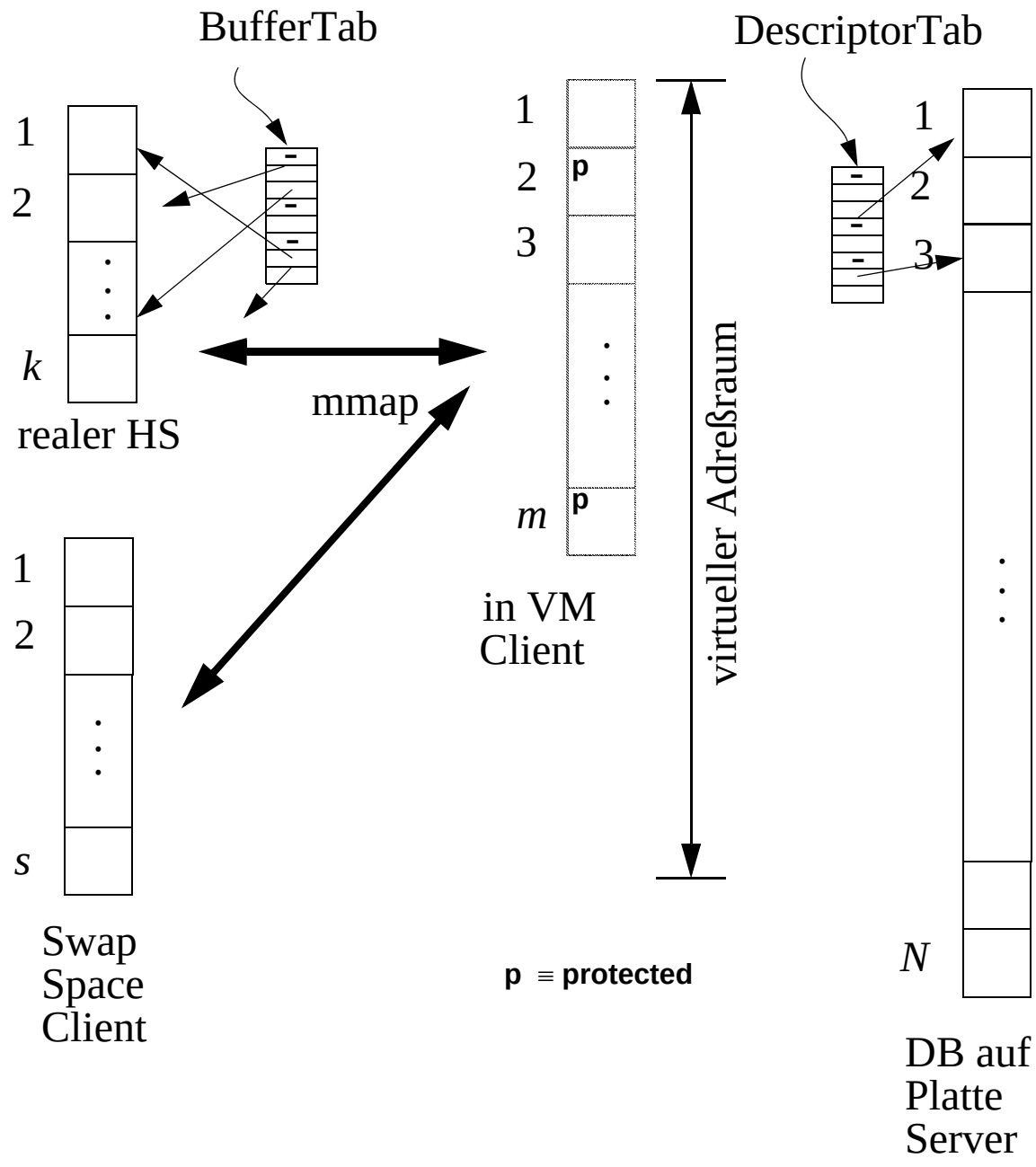
läßt sich eine neue Funktion - der selbstgestrickte Signalbediener - an *SignalNr* binden.

Bei einem ausgelösten Signal erfolgt auf Maschinenbefehlsebene eine Kontextumschaltung bei der alle Register, Instruktionszähler und Stapelzeiger gerettet werden, die Unterbrechung wird mit der angegebenen *Action* behandelt und kehrt danach zu der unterbrochenen Instruktion zurück, die in der Regel wiederholt wird (restart of instruction).

→ **Dies genau ist das sagenumwobene hardware-unterstützte Pointer Swizzling**!

- Baue einen Objektbaum auf (in den Seiten oder durch Kopieren)
- Zeiger auf noch nicht aufbereitete Objekte verweisen in geschützte Seiten; der Versuch der Dereferenzierung löst das `SIGSEGV` Signal aus
- eine eigene Routine repariert die Seiten (Zeiger werden aufbereitet), das Schutzflag wird gelöscht, die Dereferenzierung neu gestartet.

mmap + mprotect + signal = HW-gestütztes Swizzling



Vorgehensweise:

- beim Client mittels `mmap` einen anonymen Bereich (mapped file) anlegen
- Bereich muß Mehrfaches einer Seite sein und sollte zunächst der Größe des für den Puffer vorgesehenen realen HS entsprechen.
- Wurzelseite (und ggf. weitere Seiten) des Objektbaumes aus DB von Server zu Client übertragen und ab 1. Seite des Bereichs ablegen.
- Abbildung VM → DB-Seiten in DescriptorTab festhalten, ggf. auch Umkehrung (meist in Hashtabelle)
- Alle persistenten Pointer (OIDs, Plattenadressen) in den geladenen Seiten übersetzen in virtuelle HS-Adressen (das eigentliche Swizzeln). Sofern die persistenten Adressen zu Seiten gehören, die bereits geladen wurden, ist außer der Übersetzung nichts weiter zu tun.
- Gehört die persistente Adresse zu einer noch nicht geladenen Seite, wird für diese Seite Platz im virtuellen Adreßraum bestimmt (fortlaufend wie im System Texas oder möglichst an der alten Adresse wie in ObjectStore).
- Die Seite wird geschützt (`mprotect`). Sie muß nicht geladen werden, ja sie hat ggf. keine Entsprechung im HS oder im Swapspace (`mprotect` verlangt nicht, daß ein zu schützender Bereich auch tatsächlich physisch existiert, das Schutzbit wird von der MMU in der VM-Seitentabelle angebracht).

Vorgehensweise (Forts.)

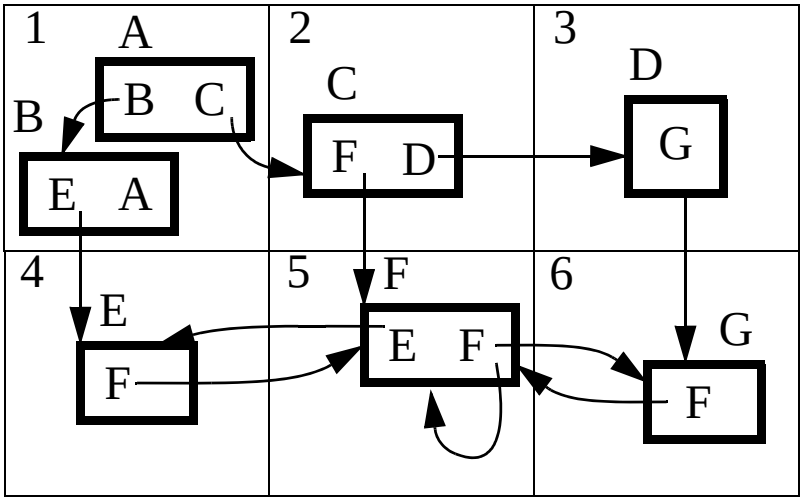
- Objektbaum kann jetzt mittels Pointer traversiert werden.
- wird auf eine geschützte Seite zugegriffen, wird `SIGSEGV` ausgelöst, die Seite wird jetzt aus der DB in die vorreservierte VM-Kachel geladen, der Schutz wird gelöscht und die Pointer darin werden geswizzelt.
- übersteigt der Bereich der angelegten und nicht geschützten Seiten den realen HS, setzt Paging in der Swap-Area ein, entweder selbstgesteuert über *BufferTab* oder durch UNIX.
- *BufferTab* ist daher dynamisch innerhalb einer Transaktion
- *DescriptorTab* ist statisch innerhalb einer Transaktion
- Pointer Swizzling bewegt sich einen Schritt vor Zugriff, d.h. Pointer zeigt schon auf richtige Adresse, aber Zugriff ist noch gesperrt, entweder weil sich dort noch ungeänderte Verweise befinden oder Seite noch nicht aus DB geladen ist.
- Kaum zu schlagende Performance beim sog. „hot traversal“, z.B. mehrfache Berechnung von Kosten und Gewicht einer A320 Tragfläche aus Stücklistenbaum einmal für Titanwaben und einmal mit Aluminiumwaben, ...

eigene Messung mit 5 Mill. Schleifendurchläufen:

Dereferenzierung	Laufzeit
SW swizzling - if ohne Auslösung Funktionsaufruf	1 500 ms
SW swizzling - if mit ausgelöstem Funktionsaufruf	1 960 ms
HW swizzling - kein Signal ausgelöst	1 130 ms
HW swizzling - signal ausgelöst	39 300 ms

Beispiel

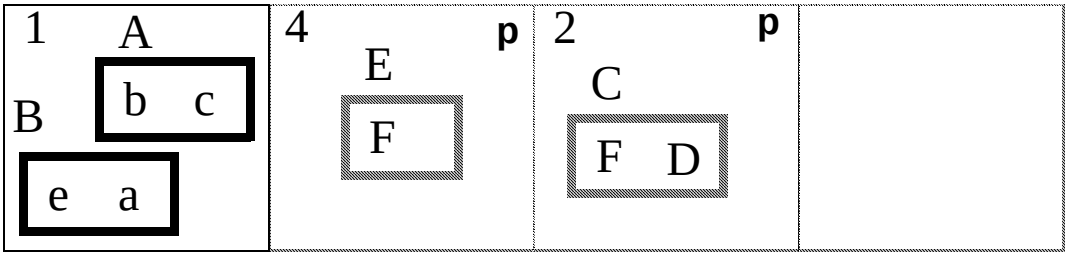
N = 6 Seiten DB, 4 Seiten virtueller Adreßraum, *s* = 1 Seite Swapspace, *k* = 2 Seiten realer HS mit i und ii Kachelnummer, A, B, ... Plattenadressen, a, b, ... Pointer



Entwicklung:

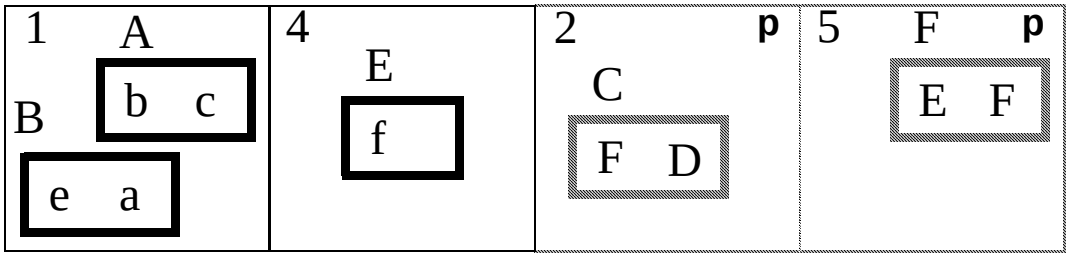
1. Seite geladen und Pointer geswizzled, *m* = 3

e dereferen-ziert, 2. Seite geladen und F geswizzled, *m* =



1	4	2	-
i	-	-	-

DescriptorTab
BufferTab



1	4	2	5
i	ii	-	-

DescriptorTab
BufferTab

Beispiel (Forts.)

Wichtig: f kann dereferenziert werden, führt zum Seitenaustausch, d.h. Seite 5 verdrängt z.B. Seite 1 in Kachel i. Plattenadressen E und F werden in Seite 5 geswizzled. Dies führt jedoch zu keiner neuen Seitenzuweisung im VM, da E und F aus Seite 5 bereits berücksichtigt wurden. Der Swapspace kann 1 Seite aufnehmen, ist jetzt dann aber voll, d.h. weitere Seiten können nicht geladen werden.

Wäre statt f eine Dereferenzierung von c erfolgt, so hätte Seite 2 geladen werden müssen. Diese hätte statt Seite 5 zwar auch noch in den Swapspace gepaßt, Seite 2 enthält aber eine Referenz nach Seite 3. Für Seite 3 müßte eine Adresse im VM festgelegt werden. Der virtuelle Adreßraum ist aber mit den 4 Seiten 1, 4, 2, 5 erschöpft.

Sofern das System nicht Verdrängung von Seiten aus dem VM unterstützt (schwierig wegen *dangling pointer*) bricht das System die Traversierung im Fall der Dereferencierung von c mit einer Fehlermeldung ab!

Probleme

- Swap Space erschöpft: Applikation berührt in Transaktion zu viele Seiten! Schlechtes Lokalitätsverhalten (Clustering).
- Virtueller Adreßraum erschöpft, z.B. 4 GB für 32-bit VM-Pointer: Seiten haben hohen Verzweigungsgrad, VM-Vorreservierung ist nötig obwohl Seiten gar nicht berührt werden (typisch für B-Bäume, dort ggf. im Verhältnis 1:500!!)
- Compiler muß sagen, wo Pointer in den Seiten stehen
- weil Signalhandler Instruktion wieder aufnimmt, muß Zeiger gültig sein (kein nachträgliches Richtigsetzen möglich, sofern nicht weiterer Indirektionsschritt vorgesehen)
- läßt sich abschwächen, wenn SW-Gültigkeitsprüfung eingesetzt wird (if Pointer A = nil then HandleException(A) else verwende *A)
- Signal wird erst beim Dereferenzieren ausgelöst
- woher weiß SignalHandler, welche Adresse SIGSEGV ausgelöst hat? Globale Variable (compilererzeugt, Macro), Interpretation der Kontextstruktur, ...?
- Ist Unswizzling notwendig beim Zurückschreiben?
- Dürfen geswizzelte Pointer in einem Cache, z.B. in lokalen Variablen, zwischengespeichert werden?
- Können Objekte verschoben werden? Große Objekte über mehrere Seiten hinweg?
- Können Seiten aus dem VM ausgelagert werden? Wie bekommen es die Seiten mit, in denen Zeiger auf die verschwundene Seite existieren?

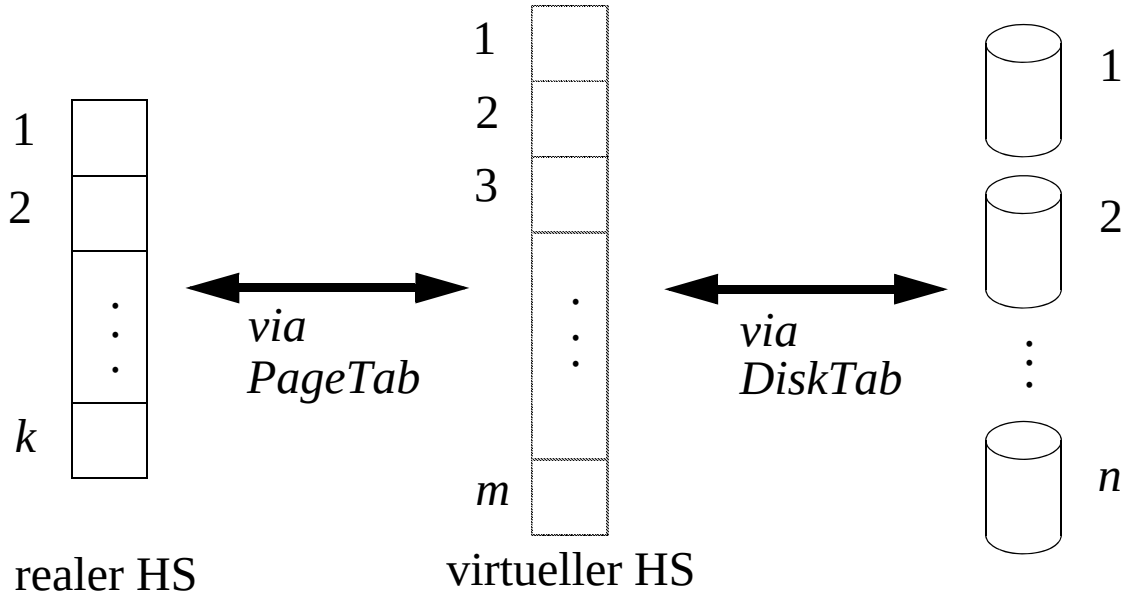
Varianten

Variation	Kurzbeschreibung
swizzling vs. no-swizzling	bei no-swizzling Umsetzung OID in Pointer über Tabelle, z.B. O_2, relativ teuer
harware vs. software	HW sehr schnell aber begrenzt Objektbaum auf VM Größe, wird von ObjectStore, Texas, Cricket, Dali, QuickStore gemacht
in-place vs. copy	extra *object cache* oder nicht, Kopieren kann Aufwand für swizzling und unswizzling reduzieren, Technik unabh. von swizzling
release vs. non-release	auch uncaching vs. no uncaching genannt, Freigabe von Seiten aus VM erlaubt oder nicht, hängende Zeiger bei Freigabe
eager vs. lazy swizzling	*eager* bedeutet im Extremfall swizzling aller Pointer in der transitiven Hülle des Einstiegpunktes, *lazy* swizzling arbeitet inkrementell und zwar entweder - je Seite (alle Pointer in einer Seite, heißt dann oft auch *eager*) - je Objekt (alle Pointer in einem Objekt) - je einzelnem Pointer
swizzling on dereference vs ... on discovery	swizzle erst wenn Zugriff auf Objekt versucht wird oder swizzle alle Pointer einer Seite wenn erkannt, auch bei load/store/copy, häufig gemischt: swizzle sofort, laden Seite bei Dereferenzierung
direct vs. indirect	bei indirekt zeigt geswizzelter Pointer nicht direkt auf das Objekt sondern auf einen *fault block*; günstig bei release, aber teuer
partial vs. full swizzling	nicht jeden Pointer wandeln und separaten Stack dafür halten, erleichtert Entdeckung hängender Referenzen
optimistic vs. pessimistic	optimistic = non-release, geht davon aus, daß VM-Bereich ausreichend groß ist, keine Seitenfreigabe vorgesehen

Einstufiger Speicher: die Lösung aller Probleme?

Im Prinzip ja, speziell durch 64-bit Rechnerarchitekturen (Alpha Chip, PowerPC, ...)

Grundidee: alles wird in einen großen virtuellen, persistenten Adreßraum gepackt, der einerseits auf *n* Platten der Größe *x* abgebildet wird und andererseits in den HS seitenweise geladen wird (paged virtual memory). Virtueller Adreßraum hat Freispeicherverwaltung analog zum Heap, z.B. mit Buddy-Verfahren.



Ansätze hat es dazu gegeben, bzw. gibt es:

- Burroughs B5000 und Multics
- IBM /38 (1984/85), Nachfolger AS/400, HP's MPE Operating System für 9000er Serie
- indirekt auch IMS FastPath

Wieso funktioniert dieses Konzept nicht so recht?

Vermutung: fehlende Unterstützung von sog. „transaktionsgestütztem virtuellen Speicher“

Zweitens: Größenordnung der Ansprüche.

Dazu kleines Horrorkabinett der Zweierpotenzen:

Größe	Bezeich.	≈ Zehnerpot	Sprechw.	Beispiel
2^{10}	KB Kilo	10^3	Tausend	1/2 Schreibmaschinenseite ASCII
2^{20}	MB Mega	10^6	Million	1 Buch, 1 Floppy
2^{30}	GB Giga	10^9	Milliarde	Film in TV-Qualität, 10 m Bücher
2^{40}	TB Tera	10^{12}	Billion (dt.)	alle Röntgenaufn. eines Krankenhauses, kleinere Bibliothek
2^{50}	PB Peta	10^{15}	Billiarde	3 Jahre EOS Landsat Daten was ein/e 60-jährige/r bisher gesehen hat bei 1 MB/s
2^{60}	EB Exa	10^{18}	Trillion	5 EB = alle jemals gesprochenen Worte
2^{70}	ZB Zetta	10^{21}	Trilliarde	
2^{80}	YB Yotta	10^{24}	Quattrillion	

Tabelle der Einzelhandelskette WalMart: 1,8 TB, 4 Milliarden Tupel, Library of Congress in ASCII ungefähr 25 TB

aber: Seitentabelle für 2^{48} Adreßraum bei 4KB Seite hat 64 Milliarden Einträge zu ca. 6 Bytes $\rightarrow \approx$ 360 GB !!

Zusammenfassung

- Swizzling in den Datenseiten als Voraussetzung für schnelles Traversieren von Objektbäumen (Visualisierung, CAx-Anwendungen)
- HW-unterstütztes Swizzling eng verknüpft mit `mmap` und `mprotect`, sehr schnell, „no black magic“
- SW-unterstütztes Swizzling flexibler und portabler, langsamer, ggf. Caching übersetzter Pointer möglich
- Pointer Swizzling als generischen Begriff verstehen, ca. ein Dutzend verschiedene Realsierungsmöglichkeiten
- Schwächen im Bereich Sicherheit, Recovery, Redo/Undo
- Swizzling damit ernstzunehmende Technologie im Übergang zum einstufigen persistenten Speicher mit 48/64 bit Technologie, der als transaktionsfähiger virtueller Speicher realisiert werden muß.