

Betriebssystemtechnik

Adressräume: Trennung, Zugriff, Schutz

XI. Bindelader

Wolfgang Schröder-Preikschat

30. Juni 2014



Rekapitulation

Mitbenutzung (*sharing*)

- Gemeinschaftssegment (*shared memory segment*)
 - explizite Text- und Datenverbände ursprünglich getrennter Adressräume
 - positionsabhängige/-unabhängige Mitbenutzung
 - Verbände erfordern einen passenden Zuschnitt der Text-/Datenbereiche
 - Ausrichtung gemäß Granulatgröße: byte-, block-, seitenausgerichtet
 - Bereichslänge ist Vielfaches der Länge einer Ausrichtungseinheit
 - Text-/Datenverbände haben statische/dynamische Systemeigenschaften
 - eine Frage der Bindezeit von Symbol und Adresse: vor/zur Laufzeit
- Übertragungstechniken: *copy on write*, *copy on reference*
 - Prozessadressraumerzeugung \leadsto COW
 - Interprozesskommunikation \leadsto COW und COR
 - Deduplizierung (VMM) \leadsto COW
- Thema heute: **Gemeinschaftsbibliothek** (*shared library*), d.h.:
 - Programmbibliothek im **Textverbund** (*code sharing*)
 - Gemeinsamkeiten mit und Unterschiede zu **dynamisches Binden**
 - von Bibliothek/sinhalt/en oder beliebigen symbolisch adressierten Entitäten



Gliederung

Einleitung

Gemeinschaftsbibliothek

Binden und Laden
Textverbände

Multics

Einführung
GE-645
Dynamisches Binden

Zusammenfassung



Programmbibliotheken

Bindungsarten

Bündel von Unterprogrammen oder Objekten, die von Programmen durch **symbolische Adressierung** angefordert werden

- statische Bindung läuft ab vor Lade- oder Laufzeit des Programms
 - ergibt **große Programme**, die alles Referenzierte eingebunden haben
 - Folge ist ein **großer Speicherplatzbedarf** im Vorder- und Hintergrund
 - für gebundene Programme sind **Bibliotheksänderungen wirkungslos**

↔ **Individualbibliothek**, ist in Teilen mehrfach repliziert gespeichert

↔ Symbolauflösung durch einen **Binder** (*linkage editor, linker*)
- dynamische Bindung läuft ab zur Lade- oder Laufzeit des Programms
 - kleinere Programme im Hintergrund, da kleinere Lademodule (Datei)
 - **bedingt kleinerer Speicherplatzbedarf** im Vordergrund: Art dyn. Bindung¹
 - Programme profitieren von Bibliotheksänderungen insb. Fehlerkorrekturen

↔ **Gemeinschaftsbibliothek**, ist im Arbeitsspeicher ggf. mehrfach repliziert

↔ Symbolauflösung durch einen **Bindelader** (*linking loader*)
- Gemeinschaftsbibliotheken haben Einfluss auf das Adressraummodell

¹Die komplette Bibliothek oder nur einzelne Einträge/Unterprogramme.



Einleitung

Gemeinschaftsbibliothek

- Binden und Laden
- Textverbünde

Multics

- Einführung
- GE-645
- Dynamisches Binden

Zusammenfassung



Arbeitsteilung von Binder und Lader, um die Anzahl redundanter Programme zu reduzieren und Speicherplatz zu sparen

- zur **Bindezeit** werden Symbole mit Adressen assoziiert
 - der Binder (*linker*) sucht in den Bibliotheken nach **Objektmodulen**, die undefinierte externe Symbole auflösen (d.h., exportieren)
 - jedoch kopiert er die gefundenen Module nicht in die Ausgabedatei
 - vielmehr vermerkt er, in welcher Bibliothek das Modul enthalten ist und
 - hinterlässt eine Liste der Bibliotheken in dem **Lademodul** (*executable*)
- die Inbetriebsetzung (*startup*) der Bibliothek(en) erfolgt zur **Ladezeit**
 - der Lader (*loader*) setzt für das Programm einen logischen Adressraum auf, gemäß den Vorgaben im Lademodul
 - anschließend durchläuft er eine **Anlaufprozedur** (*startup code*)
 - die die Bibliotheken findet und in den Programmadressraum einblendet
- die Anlaufprozedur vollzieht die **statische oder dynamische Bindung**
 - je nach den Einflussfaktoren für die Text-/Datenverbünde (s. vorige VL)
 - bevor das geladene Programm (durch `main`-Aufruf) eigentlich startet



Anlaufprozedur

- die Inbetriebsetzung der Bibliothek geschieht im Programmkontext:
 - sie ist Teil des Betriebssystems,
 - des ablauffähigen Programms selbst, das gerade geladen wird,
 - eines im Programmadressraum eingeblendeten dynamischen Binders oder
 - iv sie gestaltet sich als eine Kombination aus i, ii oder iii
- zur **Einlagerung** in den realen Adressraum bestehen die Optionen:
 - im Voraus**
 - vorgehend (*anticipatory*), vor Programmstart
 - bedingtes Laden² aller Objekte gebundener Symbole
 - auf Anforderung**
 - bei Bedarf (*on-demand*), nach Programmstart
 - speicherabgebildete Datei (*memory-mapped file*)
 - typisch in Verbindung mit virtuellem Speicher
- die **Einblendung** in den (log.) Programmadressraum erfolgt
 - im Voraus**
 - statische Gemeinschaftsbibliothek
 - auf Anforderung**
 - dynamische Gemeinschaftsbibliothek
 - mit Segmentattributen *read-only*, *copy on reference* oder *copy on write*
 - positionsabhängig (statisch) oder -unabhängig (dynamisch)

²Nur die Bibliotheksinhalte, die noch nicht geladen wurden.



Aufbau eines Prozessadressraums

- `cat /proc/self/maps`³

```

address      perm offset dev  inode      pathname
08048000-08052000 r-xp 00000000 08:01 236651    /bin/cat
08052000-08053000 rw-p 0000a000 08:01 236651    /bin/cat
090aa000-090cb000 rw-p 00000000 00:00 0          [heap]
b7450000-b7650000 r--p 00000000 08:01 180420    /usr/lib/locale/locale-archive
b7650000-b7651000 rw-p 00000000 00:00 0
b7651000-b778f000 r-xp 00000000 08:01 57293     /lib/libc-2.11.3.so
b778f000-b7790000 ---p 0013e000 08:01 57293     /lib/libc-2.11.3.so
b7790000-b7792000 r--p 0013e000 08:01 57293     /lib/libc-2.11.3.so
b7792000-b7793000 rw-p 00140000 08:01 57293     /lib/libc-2.11.3.so
b7793000-b7797000 rw-p 00000000 00:00 0
b779c000-b779d000 rw-p 00000000 00:00 0
b779d000-b779e000 r-xp 00000000 00:00 0          [vdso]
b779e000-b77b9000 r-xp 00000000 08:01 57284     /lib/ld-2.11.3.so
b77b9000-b77ba000 r--p 0001b000 08:01 57284     /lib/ld-2.11.3.so
b77ba000-b77bb000 rw-p 0001c000 08:01 57284     /lib/ld-2.11.3.so
bfe58000-bfe79000 r-w-p 00000000 00:00 0          [stack]
            
```

 - .so**
 - *shared object*, Gemeinschaftsbibliothek (*libc*)
 - Anlaufprozedur als dynamischer Binder (*ld*)
 - *libc* und *ld* passend zu Bibliotheksversion 2.11.3
 - Erlaubnis (*perm*): *s* *shared*, *p* *private* (*copy on write*)
 - vdso**
 - *virtual dynamic shared object*
 - Einrichtung zur Systemaufrufbeschleunigung \rightsquigarrow `sysenter` [13]

³`proc(5)`: systemabhängig, der Aufbau ist nicht überall identisch.



Positionsabhängigkeit

Eigenschaft von Programmtext, der abhängig von seiner absoluten Lage in einem Adressraum ausführbar ist

- manifestiert in der **Adressierungsart** zum Zugriff auf Befehlsoperanden
 - **absolute** (auch: direkte) **Adressierung**

```
1 int fancy;                1 _sloth:
2                               2   cml $0, _fancy # data
3 void sloth () {           3   jne LBB1_2
4     if (fancy)            4   ret
5       labor();           5 LBB1_2:
6   }                       6   jmp _labor # text
```

- sprachlich artikuliert zur Programmier- oder Übersetzungszeit
 - unveränderlich an Adressen gebunden zur Binde- oder Ladezeit
- verankert im Programmiermodell der **Maschinenprogrammebene**
 - festgelegt durch das Adressraumkonzept und -modell des Betriebssystems
 - wenn es z.B. Gemeinschaftssegmenten einen festen Adressbereich zuweist
 - beeinflusst durch die Hardware-Vorgaben der Befehlssatzebene



Positionsunabhängigkeit I

Motiv: Verwendung von absoluter Adressierung (Text) oder absoluten Adressen (Daten) legt Programmbereiche fest

- **Relokation zur Laufzeit** ist zwar möglich, aber oft nicht praktikabel
 - es müssen alle zu ändernden Programmadressen bekannt sein **!?**
 - für Programmtext stehen diese in der **Symboltabelle** des Binders⁴
 - für Programmdateien sind **dynamische Datenstrukturen** zu verfolgen
 - ↳ Programme werden dann nicht „unverfälscht“ (*pure*) ablaufen können
 - aber nicht alle Adressen eines Programms liegen wirklich offen
 - z.B. dynamische Datenstrukturen des Laufzeitsystems (Halde, Stapel, ...)
- die Programme selbst müssten **frei verschiebbar** ausgelegt sein
 - ausschließliche Verwendung der relativen Adressierung im Programmtext
 - indizierte Adressierung mit **Basisregister**: Adressdistanz zu einem Basiswert
 - Befehlszähler für Programmtext, Adressregister für Programmdateien
 - Anweisungen von Hand (☺) oder durch den Kompilierer (☺) generieren

↳ **position-independent code** (PIC): durchgängig indizierte Adressierung

⁴Sofern nicht mittels `strip(1)` aus dem Objekt-/Lademodul entfernt.



Positionsunabhängiger Programmtext absolut vs. PC-relativ

```
1 int count () {
2   static int counter = 0;
3   return counter++;
4 }
```

```
1 _count: # gcc -O6 -static -m32 -S -fomit-frame-pointer
2   movl   _counter.1435, %eax
3   leal   1(%eax), %ecx
4   movl   %ecx, _counter.1435
5   ret
```

```
1 _count: # gcc -O6 -static -m32 -S -fomit-frame-pointer -fPIC
2   call   L1$pb # save program counter to stack5
3 L1$pb: # this is basis for PC-relative addressing5
4   popl   %ecx # load program counter from stack5
5   movl   _counter.1435-L1$pb(%ecx), %eax
6   leal   1(%eax), %edx
7   movl   %edx, _counter.1435-L1$pb(%ecx)
8   ret
```

⁵Funktionsprolog (Mac OS X) zur Bestimmung der eigenen Basisadresse.



Zusammenhang zur Gemeinschaftsbibliothek I

Profiteur, wenn überhaupt, von positionsunabhängigem Code ist die **dynamische Gemeinschaftsbibliothek** (*dynamic shared library*)

- deren Symbole erst **zur Laufzeit** an Adressen gebunden werden
 - durch einen bindenden Lader (*linking loader*) oder
 - dem dynamischen Binder (*dynamic linker*)
 - deren Verortung im logischen Adressraum **nicht** fest vorgegeben ist
- Gegenstück dazu ist die – durch positionsunabhängigen Code unnötig belastete – **statische Gemeinschaftsbibliothek** (*static shared library*)
- deren Symbole bereits **vor Laufzeit** an Adressen gebunden werden
 - normalerweise durch den (statischen) Binder (*linker*)
 - deren Verortung im logischen Adressraum damit fest vorgegeben ist

Gemeinsamkeit:

- der Programmtext wird zur Lade- oder Laufzeit eingeblendet (S. 7)



Eigenschaft von Programmtext, der unabhängig von seiner absoluten Lage in einem logischen Adressraum ausführbar ist

- eine Art von Ortstransparenz, jedoch problemspezifisch ausgelegt:
 - Ladezeit** ■ Moment der Einblendung in den logischen Adressraum
 - zur Laufzeit können absolute Adressen gespeichert werden
 - ↔ Gemeinschaftsbibliothek, dynamisches Binden
 - Laufzeit** ■ Moment des Befehlsabrufs aus dem logischen Adressraum
 - es dürfen nur relative Adressen gespeichert werden
 - ↔ Kompaktifizierung, Speicherbereinigung, Migration (Umzug)
- Umzugsfähigkeit von (dynamischen) Daten ist nicht vordergründig
 - obgleich *copy on write/reference* damit uneingeschränkt nutzbar wird:
 - wenn Nachrichteninhalte direkt auf Programmdatenstrukturen verweisen, die unarrangiert („*unmarshalled*“) und direkt übertragen werden sollen
 - um dynamische Daten beliebig im Empfangsadressraum platzieren zu können
 - ganz unabhängig von dynamisch gebundenem Programmtext
- positionsunabhängige dynamische Daten sind üblw. **Quelltextmerkmal**



```

1 chain_p find (chain_p list, chain_p item) {
2     while (list && (list != item))
3         list = list->link;
4     return list;
5 }
```

Indizierte Basisadressierung (-fPIB)

<pre> 1 _find: 2 movl 4(%esp), %eax 3 testl %eax, %eax 4 je LBB1_4 5 movl 8(%esp), %ecx 6 LBB1_2: 7 cmpl %ecx, %eax 8 je LBB1_4 9 movl (%eax), %eax 10 testl %eax, %eax 11 jne LBB1_2 12 LBB1_4: 13 ret</pre>	<pre> 1 _find: 2 movl 4(%esp), %eax 3 testl %eax, %eax 4 je LBB1_4 5 movl 8(%esp), %ecx 6 LBB1_2: 7 cmpl %ecx, %eax 8 je LBB1_4 9 movl (%ebp,%eax), %eax 10 testl %eax, %eax 11 jne LBB1_2 12 LBB1_4: 13 ret</pre>
--	---



Zusammenhang zur Gemeinschaftsbibliothek II

Positionsunabhängigkeit von Programmen ist sehr praktisch – gibt es aber nicht für umsonst

There's no such thing as a free lunch!

- als Gegenleistung wird Geduld zur Ausführungszeit erwartet
- positionsunabhängige Programme werden langsamer laufen

- Gründe des Performanzrückgangs von Gemeinschaftsbibliotheken [16]:
 - Ladezeit** i Verschiebung (*relocation*) der Bibliotheken
 - ii Auflösung (*resolution*) von Programmsymbolen
 - Laufzeit** iii Entschleunigung durch Einsprungtabelle (S. 16)
 - iv Mehraufwand durch Funktionsprolog (vgl. S. 11) PIC
 - v Mehraufwand durch indirekte Datenreferenzen PIC
 - vi Verlangsamung durch reservierte Adressregister PIC
- Optionen zur Verbesserung [16]:
 - Zwischenspeicherung (i-ii), Positionsunabhängigkeit aufgeben (iv-vi)



Statischer Ansatz I

Symbole wurden zur **Bindezeit** zwar an Adressen gebunden, aber die Objekte dazu werden erst zur **Ladezeit** verknüpft

- zwischenzeitliche Bibliotheksänderungen sind nicht unproblematisch
 - Text- und Datenadressen sind im Programm aufgelöst und eingebunden
 - Inkonsistenzen bewirken undefinierte Programmzustände zur **Laufzeit**
- Beibehaltung direkter Adressierung von Bibliotheksartefakten bedingt
 - Mehrfachversion** ■ Vermerk der Versionsnummer zur Bindezeit *und*
 - Verknüpfung mit zgh. Bibliothek zur Ladezeit
- Alternativlösung mit indirekter Adressierung bedeutet zweierlei:
 - Sprungtabelle** ■ von Sprungbefehlen, je 1 pro exportierte Routine⁶
 - am Anfang der Bibliothek (ab der ersten Seite)
 - Gemeinschaftsblock** ■ von feldartigen Datenstrukturen bekannter Größe
 - ↔ *common block*: z.B. FILE*, errno, tzname [2]
 - der Sprungtabelle (seitenausgerichtet) folgend
- beide Ansätze definieren **Pufferzonen** hinter den Bibliotheksbereichen

⁶ISO-C99 hat 482 Funktionen, neben den 24 *header*-Dateien.



Hauptschwierigkeit jedoch besteht in der Festlegung des jeweiligen Adressbereichs einer jeden Bibliothek

- jede Gemeinschaftsbibliothek definiert einen festen Adressbereich
 - nicht überlappend, ggf. nicht identisch in allen Programmadressräumen
 - Linux ab 0x60000000, versionsabhängig⁷
 - BSD ab 0xa0000000 systemspezifische Bibliotheken und ab 0xa0800000 anbieterspezifische Bibliotheken
 - Windows jede DLL spezifiziert eine RVA (*relative virtual address*) als gewünschte Basisadresse (0x10000000 für Visual C++)
 - falls frei, platziert der Binder die DLL wie gewünscht
 - falls belegt, wird die Umplatzierung (*relocation*) versucht
 - bestimmt durch das Programmiermodell der Maschinenprogrammebene
- konfliktfreie Abbildung im Voraus (*pre-mapping*) ist nicht garantiert

Although it's possible to check mechanically that libraries don't overlap, assigning address space to libraries is a black art. [16]

⁷Linux 2.6 unterstützt nur noch dynamische Gemeinschaftsbibliotheken.



Großteil des Bindevorgangs wird aufgeschoben bis zur **Startzeit** eines Programms und ggf. auch darüberhinaus

- Verortung im logischen Adressraum kann selbstbestimmt sein, also vorgegeben durch **Bibliothekseigenschaften**
 - vorgebunden
 - bedingt positionsabhängiger Kode \leadsto DLL (S. 17)
 - Umplatzierung, falls der vorgegebene Bereich belegt ist
 - \hookrightarrow Mitbenutzung nur für dieselben Programminkarnationen
 - \hookrightarrow nicht aber für Inkarnationen verschiedener Programme **!!!**
 - sonst
 - positionsunabhängiger Kode \leadsto SunOS, ELF [25, 24]
 - Einblendung scheitert, falls kein (anderer) passender Bereich verfügbar ist
 - das Betriebssystem definiert ggf. weitere Vorgaben zur Verortung, in dem es einen **reservierten Bibliotheksbereich** vorsieht
 - typischerweise zwischen Stapel und Halde liegend (Linux, vgl. S. 8)
 - innerhalb dieses Bereiches sind Bibliotheken bedingt (s.o.) frei platzierbar
- \hookrightarrow echte Segmentierung hat diese Einschränkungen nicht \leadsto Multics [20]



Dynamischer Ansatz II

Durchführung des Bindevorgangs zur **Laufzeit** des Programms, und zwar im Moment der wirklichen Benötigung

- der gewünschten Bibliothek
 - explizit, durch **programmiertes Nachladen**
 - `dlopen()` in Linux, `LoadLibrary()` in Windows
 - intransparent für den Prozess, in jeder Hinsicht
 - in der Benutzung ähnlich zur Technik der Überlagerung (*overlay*, [22])
 - vorwegnehmend ausgelöst durch Prozedur- oder Systemaufruf
- einer Routine des Programms selbst – oder eines anderen Programms
 - implizit, durch **partielle Interpretation**
 - „Bindungsfalle“ (*link trap*) in Multics [20]
 - transparent für den Prozess, in funktionaler Hinsicht
 - auch bei zwischenzeitlicher Verdrängung eines nachgeladenen Objektes
 - ausnahmebedingt ausgelöst durch synchrone Programmunterbrechung
- die Adressen bleiben **bis zum Aufruf** (einer Routine) **ungebunden**



Gliederung

Einleitung

Gemeinschaftsbibliothek
Binden und Laden
Textverbünde

Multics
Einführung
GE-645
Dynamisches Binden

Zusammenfassung



- 1963 ■ *Project MAC (Mathematics and Computation, MIT)*, DARPA
 - ein Ziel war die Entwicklung des Nachfolgers von CTSS [4]
- 1964 ■ General Electric (GE) übernimmt Bull, frz. Rechnerhersteller
- Multics
 - 1965 ■ FJCC, Entwurfsideen und Grundprinzipien [6, 9, 26, 7, 21, 8]
 - eigene Sitzung: *A new remote accessed man-machine system*
 - 1967 ■ Januar, Erstinstallation (GE-645) am MIT
 - 1969 ■ April, Rückzug der Bell Laboratorien aus dem Projekt
 - Herbst, Mehrbenutzerbetrieb (*timesharing*) am MIT
 - 1970 ■ Honeywell übernimmt die Rechnersparte von General Electric
 - GE-600 Serie wird zur Honeywell 6000-Serie
 - 1973 ■ Januar, Honeywell 6180: zweite Generation Multics-Maschine
 - 1985 ■ Juli, Honeywell stellt weitere Entwicklungen (im 6. Versuch) ein
 - August, Sicherheitszertifizierung der Kategorie B2 [15]
 - 2000 ■ 30. Oktober, Betriebseinstellung der letzten Installation
 - DND-H, Canadian Department of National Defence, Halifax
 - 2006 ■ Offenlegung des Quelltextes durch Bull SAS [18]

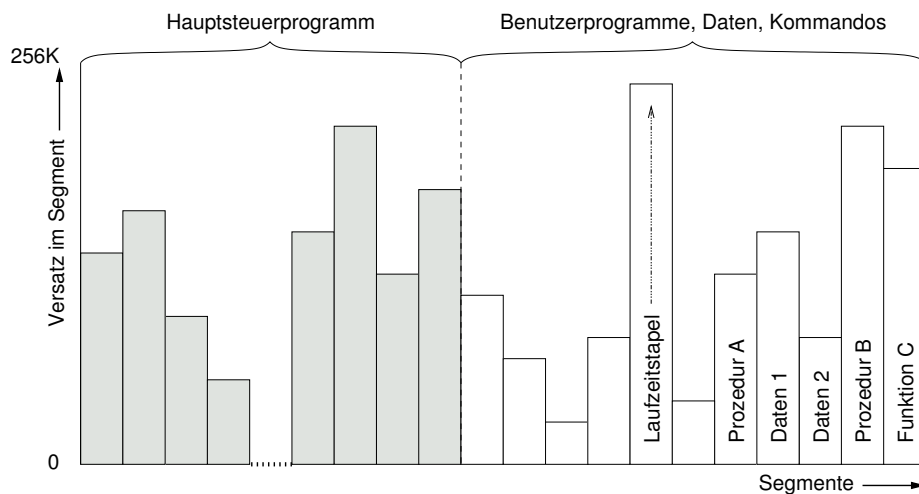


- neue Ideen
 - Verschmelzung von Dateien und Arbeitsspeicher zum **einstufigen Speicher**
 - bedarfsorientierte Programmausführung durch **dynamisches Binden**
 - benutzerorientiertes **hierarchisches** (baumstrukturiertes) **Dateisystem**
 - von Anbeginn als **sicheres System** entworfen (ringorientierter Schutz)
 - äußerst energische, **mitlaufende** (*on-line*) **Hardware-Rekonfigurierung**
- andere Merkmale: bis 1973, 150 PJ Entwicklung und 50 PJ Pflge [5]
 - Mehrsprachensystem, implementiert in:
 - PL/1 [3, 12], aus Gründen der Produktivität nicht Portabilität [10] und
 - ALM [11], für zwingend hardwarebezogene Operationen
 - kompaktes residentes Hauptsteuerprogramm (*supervisor*)
 - ~ 30 KW Prozedur- und (für 55 Benutzer) 36 KW Daten-/Pufferbereiche [5]
 - gegenüber SELinux [19] schlanke vertrauenswürdige Rechenbasis⁸ [14]
 - ~ 628 KB Sicherheitskern (Ring 0) vs. 1767 KB nur für das Sicherheitsmodul
 - ~ zweieinhalbfach kleinere TCB, den Linux-Kern selbst nicht eingeschlossen
 - inhärent, stark querschneidend abhängig vom GE-645/Honeywell 6180



⁸trusted computing base, TCB

Zweidimensionaler Prozessadressraum



- max. 2^{18} Segmente, jew. max. $2^{18} * 4 = 2^{20}$ Bytes, pro Adressraum
- letztere teilen sich die Hauptsteuerprogrammsegmente (*supervisor*)



GE-600 Serie

1960er Jahre

Großrechnerfamilie mit 36-Bit Systemzentraleinheit (*mainframe*)

- Basis bildete die bis auf vier Prozessoren ausbaufähige GE-635
 - symmetrisches Multiprozessorsystem (*symmetric multiprocessing, SMP*)
 - Rechnersystem der zweiten Generation: Transistor-Transistor-Logik (TTL)
 - 36-Bit wortorientierte Maschine, 72-Bit Speicherpfad, 18-Bit Adressen
- GE-645 [9]: speziell für Multics modifizierte GE-635
 - seitennummerierte Segmentierung (*paged segmentation*)
 - zweiteilige Adresse: 1. Segment- und 2. Wortnummer, jew. $[0, 2^{18} - 1]$
 - implizite und explizite Gestaltung von effektiven (zweiteiligen) Adressen
 - ausgewählt über Steuerschalter (Bit 29) im Befehlswort
 - Prozedurbasisregister oder eines von acht Adressbasisregistern (24-Bit)
 - bei Bedarf Paarbildung letzterer zu vier (2×24 -Bit) „Zeigerregister“⁹
 - segmentierte Adresse als Speicherindirektwort (ITS, ITB) darstellbar
 - zusätzlich zu (GE-635) *master* und *slave* auch Ausführungsmodus *absolute*
 - 512 KW \approx 2 MiB Hauptspeicher – riesig für damalige Verhältnisse
- Betriebssysteme für GE-635 waren GECOS [1] und Mark II bzw. III



⁹Effektiv nur 2×18 -Bit, jeweils für Segmentnummer und Adresse.

```

000 000000011111111 112222222 222 333333
012 345678901234567 890123456 789 012345
+-----+-----+-----+-----+
| BR | ADDRESS | OPCODE | UIB | TAG |
+-----+-----+-----+-----+

```

Field Name	Size	Purpose
BR	3 bits	base register; 0-7
ADDRESS	15 bits	word address; 0-32767 (32KW)
OPCODE	9 bits	instruction opcode
U	1 bit	unused
I	1 bit	interrupt inhibit flag
B	1 bit	0=no base register (GE-635 form) 1=base register (GE-645 form)
TAG	6 bits	index/indirect type

- um dynamisches Binden durch Multics auszulösen:
 - i ist der Schalter im B-Feld auf 1 gesetzt und
 - ii Basisregister BR adressiert ein Speicherindirektwort ~ S. 27
 - effektiv vier Basisregisterpaare, die jew. eine zweiteilige Adresse speichern



```

01 2345
+---+---+
|Tm| Td |
+---+---+

```

Tm	2 bits	Tag modifier
00	R-type	register, no indirect
01	RI-type	register then indirect
10	IT-type	indirect then tally
11	IR-type	indirect then register

Td	4 bits	Tag descriptor
0ccc	ccc is a 3-bit code	
1rrr	rrr is a 3-bit index register	

- Typ RI: multilevel indexed indirect addressing¹⁰ [20, S. 27]
- das Etikett des Leitwortes (Wort 0) gibt den weiteren Verlauf vor:
 - ITS ■ indirect to segment ~ Zugriff, gesteuert durch TAG von Wort 1
 - FT2 ■ fault tag 2 ~ Auslösung eines Bindefehlers (trap)
- beide Ausprägungen realisiert durch freie Nummern im Etikettfeld

¹⁰rrr im Etikettdeskriptor identifiziert das Indexregister.



```

00000000011111111 112222222 222 333333
012345678901234567 890123456 789 012345
+-----+-----+-----+-----+
| SEGMENT # | UNUSED | UUU | ITS | WORD 0
+-----+-----+-----+-----+

```

```

00000000011111111 112222222 222 333333
012345678901234567 890123456 789 012345
+-----+-----+-----+-----+
| ADDRESS | UNUSED | UUU | TAG | WORD 1
+-----+-----+-----+-----+

```

Field Name	Size	Purpose
SEGMENT #	18 bits	segment number; 0-262143
UNUSED	9 bits	unused
UUU	3 bits	unused
ITS	6 bits	'43'b3

Field Name	Size	Purpose
ADDRESS	18 bits	word address; 0-262143
UNUSED	9 bits	unused
UUU	3 bits	unused
TAG	6 bits	index/indirect type



Adressbasisregister (ABR), die durch Befehle vom Typ 1 aktiviert und zur Programmierung sichtbar wurden¹¹

- das 3-Bit-Feld wird auch als Segmentetikett (segment tag) bezeichnet
 - effektiv vier Registerpaare, in Multics wie folgt benannt bzw. verwendet:

0-1	Argumentenliste (argument pointer/base)	AP & AB
2-3	allgemeine Basis (base pointer/base)	BP & BB
4-5	Bindungssegment (linkage pointer/base)	LP & LB
6-7	Stapelsegment (stack pointer/base)	SP & SB

- die Paarbildung steuert ein Kontrollfeld (Bits 18-23) im ABR:

```

012345678901234567 890 1 23
+---+---+---+---+---+
| ADDRESS/NAME | PAL | X | UU |
+---+---+---+---+---+

```

pal (dt. Kumpel)
Partnerregistername bzw. -nummer

- X bestimmt die Verwendung von Bits 0-17 und Bits 18-20:
 - 0 → segmentlokale Wortadresse, pal ist ABR mit Segmentnamen
 - 1 → globaler Segmentname (externe Basis), pal bleibt ungenutzt

¹¹Befehlsbit 29 (B) ist 1, vgl. S. 25.



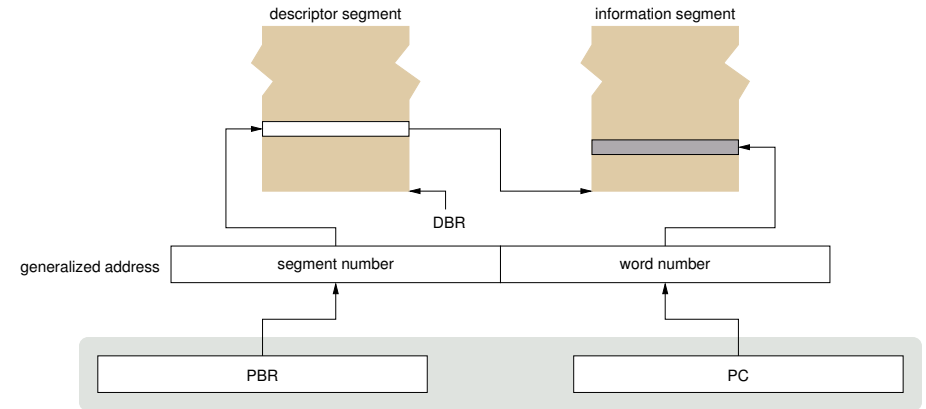
Adressen sind zweiteilige Gebilde, bestehend aus (1) **Segmentnummer** und (2) **Wortnummer** innerhalb des Segments¹²

- Einrichtungen zur Bildung einer „generalisierten Adresse“ daraus:
 - i Basisadresse der Segmentdeskriptortabelle (*descriptor base register, DBR*)
 - ii Segmentnummer der aktuellen Prozedur (*procedure base register, PBR*)
 - zusammen mit dem PC, einem Adressbasisregisterpaar nicht unähnlich
 - iii vier Adressbasisregisterpaare (AP, BP, LP, SP)
- **beachte:** Segment- und Wortnummern sind ortsunabhängige Daten
 - nur DBR hält eine Speicheradresse: Tabelle indiziert mit Segmentnummer
 - ein Segment ist sodann eine Worttabelle indiziert mit Wortnummer

Prozessadressräume werden jeweils durch ein **Deskriptorsegment**, das einer Segmenttabelle entspricht, technisch repräsentiert

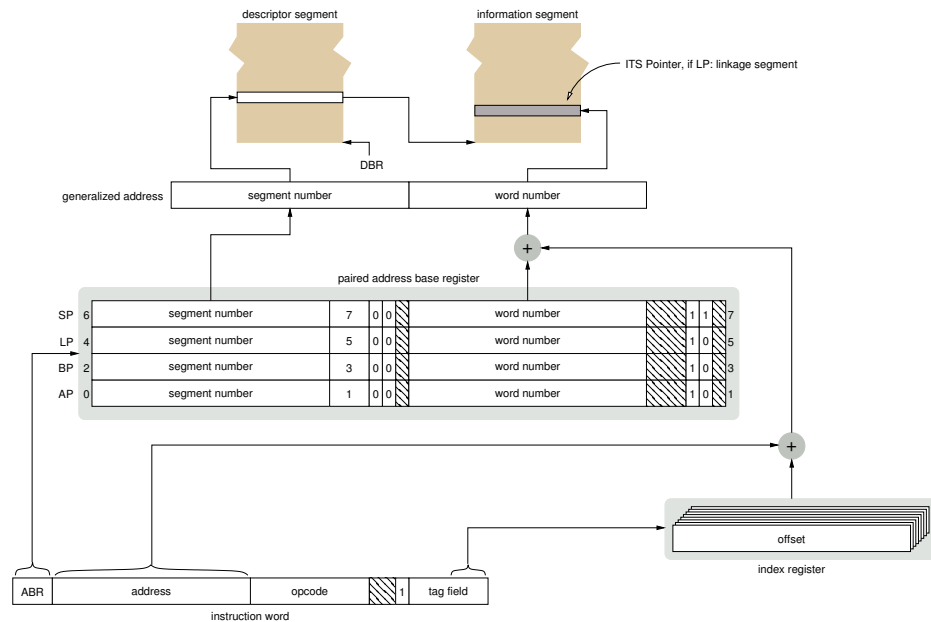
- DBR enthält die Anfangsadresse dieses Segments im Hauptspeicher
 - ein Softwareprototyp davon ist Bestandteil des Prozesskontrollblocks
 - beim Prozesswechsel wird der Hardwareprototyp entsprechend gesetzt

¹²Segmentnummer $\hat{=}$ Segmentname, Wortnummer $\hat{=}$ Wortadresse.



Bildung einer Adresse *ga* für den Befehlsabruf (*instruction fetch*) ist vergleichsweise unkompliziert \sim **Konkatenation**

- $ga = (\text{Segmentnummer} \langle DBR \rangle, \text{Wortnummer} \langle PC \rangle)$
- erst im Befehl ist der **Operandenabruf** \sim **dynamisches Binden** kodiert



Exemplar eines Hardwaredatentypen zur **Zwischensegmentbindung** (*inter-segment linking*) \sim **dynamisches Binden**

- Doppelwort (72-Bits) im Arbeitsspeicher/Adressraum eines Prozesses
 - referenziert durch die gebildete generalisierte Adresse, *falls*
 - das Befehletikett **indiziert-indirekte Adressierung** (RI, S. 26) spezifiziert
- das Etikett im **Leitwort** (gerade Adresse) definiert den Zeigerzustand
 - snapped**
 - der Zeiger ist an eine generalisierbare Adresse gebunden **ITS**
 - Segment- und (im Folgewort) Wortnummer sind gültig
 - **Normalfall**, der Prozessor führt den Zugriff direkt aus
 - das Etikett im Folgewort spezifiziert den weiteren Zugriff¹³
 - unsnaped**
 - der Zeiger ist ungebunden **FT2**
 - **Ausnahmefall**, der Prozessor löst einen Zugriffsfehler aus
 - das Betriebssystem behandelt den **Bindefehler** (*link trap*)
 - bei Wiederaufnahme wiederholt der Prozessor den Zugriff
- FT2 stellt Segment- und Wortnummernfeld zur freien Verfügung
 - das Betriebssystem kann darin eine **symbolische Adresse** kodieren

¹³Hier kann abermals indiziert-indirekte Adressierung kodiert sein.

Zwischensegmentbindung wird über ein eigenes Segment gesteuert, das die *ITS Pointer* eines Prozesses zusammenfasst

- angenommen, Prozedur *P* setzt folgenden Maschinenbefehl ab:¹⁴

OPR <D>| [x] ■ Darstellung in ALM [11]

- eine beliebige Operation OPR auf Wort *x* in Datensegment *D*
- daraus wird eine **Verknüpfung** (*link*) generiert, z.B. *ITS Pointer #42*
- jede dieser Verknüpfungen ist zunächst ein „*unsnapped link*“ \leadsto FT2
- der Operand ist als „*ACC String*“ kodiert, allg.: <seg>| [ext]+exp,m
 - definiert eine symbolische Verknüpfung mit einem symbolischen Eintrag
 - im Beispiel von OPR zeigt der „*unsnapped link*“ dann auf <D>| [x]
 - Verwendung der Segment- und Wortnummernfelder als Zeichenkettenzeiger

- Assemblierung des Befehls kodiert indiziert-indirekte Adressierung:

OPR 1p|42,* ■ „*link*“ mit Wortnummer 42 im Bindungssegment

- FT2-Behandlung lässt die Befehlskodierung und damit *P* invariant (S. 35)

¹⁴In PL/1 für Multics würde die Referenz als x\$D dargestellt sein.

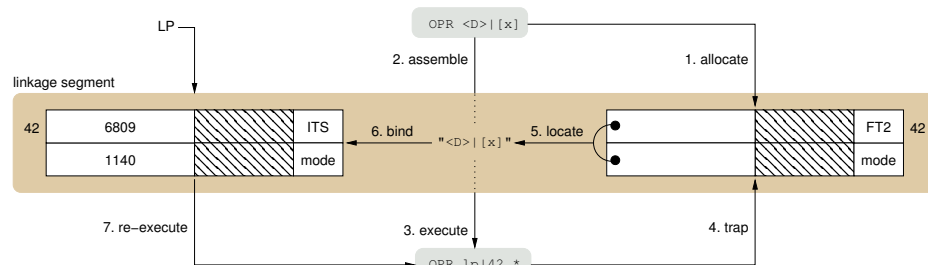


Aktionen zur Änderung der Verknüpfungsdaten im Bindungssegment, um eine Verknüpfung (*ITS Pointer*) zu etablieren

- Ausgangspunkt ist die **offene Verknüpfung** (*unsnapped link*)
 - das FT2-Etikett des Leitworts verursacht einen Bindefehler (*link trap*)
 - die Fehlerbehandlung übernimmt der **Bindelader** (*linking loader*)
 - dieser lokalisiert das über den „*ACC String*“ symbolisch adressierte Segment
 - er versucht, die symbolische in eine generalisierte Adresse umzuwandeln und
 - blendet das angeforderte Objekt in den (virtuellen) Prozessadressraum ein \hookrightarrow speicherabgebildete Datei (*memory-mapped file*), mitbenutztes Segment
 - bei positiver Symbolauflösung wurde <D>| [x] umgewandelt in $d\#\alpha | x$:
 - $d\#\alpha$ ist Segmentnummer *d* für Prozess α
 - x* ist Wortnummer *x* im Segment namens $d\#\alpha$
 - bei negativem Ausgang scheitert das Binden als Segmentierungsfehler
- Operationsergebnis ist die **ingerastete Verknüpfung** (*snapped link*)
 - $d\#\alpha$ und *x* wurden als Segment- und Wortnummer übernommen
 - im Leitwort wurde das Etikettfeld von FT2 auf ITS abgeändert



Binden II



- der Ansatz lässt das **Prozedursegment unverfälscht** (*pure procedure*)
 - die Änderungen betreffen nur das Bindungssegment des Prozesses
- Segmente von Prozeduren sind invariant gegenüber Außeneinflüsse
 - insb. durch Änderungen bei Neuübersetzung anderer Segmente
 - wenn dadurch Werte von Bezeichnern variieren, die Adressen in diesen Segmenten kennzeichnen *und*
 - diese Bezeichner in anderen Segmenten verwendet bzw. gespeichert werden
 - Immunität**, die uneingeschränkte Mitbenutzung (*sharing*) sicherstellt



Sonstiges

Segmentierung als Konzept zur Unterteilung von Programmen bzw. Prozessadressräumen hat seinen Ursprung im B 5000 [2, 17]

- aber Multics hat dieses Konzept konsequent zur Geltung gebracht
 - Segmente waren direkt und kontrolliert adressierbare Informationspakete
 - sämtliche (*online*) Information war direkt über Segmente referenzierbar
 - wohingegen in anderen Systemen dies nur über Dateien möglich war
 - nicht nur zur damaligen Zeit, auch wieder heute (2014) wäre das ein Novum
- Segmentzusammenstellung fokussierte auf **nichtfunktionale Aspekte**
 - Referenzlokalitäten** (*locality of reference*) standen im Vordergrund
 - segmentlokale Referenzen bedingen keine indirekt adressierten Wortpaare
 - funktionale Aspekte und Gemeinsamkeiten traten eher in den Hintergrund
- auch die interaktive Benutzung von Multics war „segmentorientiert“
 - Kommandos/Programme waren als Segmente repräsentierte Prozeduren
 - Benutzereingaben trieben den Kommandointerpretierer in die „Bindefälle“
- unter Multics lagen alle Prozeduren in einer **Gemeinschaftsbibliothek**
 - die allen Benutzern kontrolliert (ggf. auch ändernd) zugänglich war



Einleitung

Gemeinschaftsbibliothek

Binden und Laden
Textverbände

Multics

Einführung
GE-645
Dynamisches Binden

Zusammenfassung



- Gemeinschaftsbibliotheken haben Einfluss auf das Adressraummodell
 - Arbeitsteilung von Binder und Lader beugt redundanten Programmen vor
 - zur Bindezeit externe Programmsymbole mit Bibliothekseinträgen assoziieren
 - zur Ladezeit die benötigten Bibliothek in Betrieb setzen: Anlaufprozedur
 - die Inbetriebsetzung der Bibliothek geschieht im Programmkontext:
 - i sie ist Teil des Betriebssystems,
 - ii des ablauffähigen Programms selbst, das gerade geladen wird,
 - iii eines im Programmadressraum eingeblendeten dynamischen Binders oder
 - iv sie gestaltet sich als eine Kombination aus i, ii oder iii
 - der Einlagerung in den realen folgt Einblendung im logischen Adressraum
- Gemeinschaftsbibliotheken sind unterschiedlich ausgeprägt:
 - statisch ■ Symbole zur **Bindezeit** mit Adressen assoziieren, aber
 - die zugehörigen Objekte erst zur **Ladezeit** verknüpfen
 - dynamisch ■ Symbolauflösung zur Bindezeit, Bindung zur **Startzeit** oder
 - **Laufzeit** \leadsto programmiertes Nachladen, Teilinterpretation
- Multics leistete Pionierarbeit zum dynamischen Binden: *lazy binding*



Literaturverzeichnis I

- [1] BELEEC, J. :
From GECOS to GCOS8: An History of Large Systems in GE, Honeywell, NEC and Bull.
http://www.feb-patrimoine.com/english/gecos_to_gcoss8_part_1.htm, Febr. 2003
- [2] BURROUGHS CORPORATION (Hrsg.):
The Descriptor — A Definition of the B 5000 Information Processing System.
Detroit 32, Michigan, USA: Burroughs Corporation, Febr. 1961.
(Bulletin 5000-20002-P)
- [3] CORBATÓ, F. J.:
PL/I as a Tool for System Programming.
In: *Datamation* 15 (1969), Mai, Nr. 5, S. 68–76
- [4] CORBATÓ, F. J. ; DAGGETT, M. M. ; DALEY, R. C.:
An Experimental Time-Sharing System.
In: *Proceedings of the 1962 Spring Joint Computer Conference (AFIPS '62)*
American Federation of Information Processing Societies, AFIPS Press, 1962, S. 335–344



Literaturverzeichnis II

- [5] CORBATÓ, F. J. ; SALTZER, J. H. ; CLINGEN, C. T.:
Multics: The First Seven Years.
In: *Proceedings of the Spring Joint Computer Conference (AFIPS '72)*.
New York, NY, USA : ACM, 1972, S. 571–583
- [6] CORBATÓ, F. J. ; VYSSOTSKY, V. A.:
Introduction and Overview of the Multics System.
In: [23], S. 185–196
- [7] DALEY, R. C. ; NEUMANN, P. G.:
A General-Purpose File System for Secondary Storage.
In: [23], S. 213–229
- [8] DAVID, E. E. Jr. ; FANO, R. M.:
Some Thoughts About the Social Implications of Accessible Computing.
In: [23], S. 243–247
- [9] GLASER, E. L. ; COULEUR, J. F. ; OLIVER, G. A.:
System Design of a Computer for Time Sharing Applications.
In: [23], S. 197–202



Literaturverzeichnis III

- [10] GREEN, P. :
Multics Virtual Memory — Tutorial and Reflections.
<ftp://ftp.stratus.com/pub/vos/multics/pg/mvm.html>, Mai 1993
- [11] HONEYWELL INFORMATION SYSTEMS (Hrsg.):
ALM Assembler.
Waltham, MA, USA: Honeywell Information Systems, Febr. 1975.
(AN63)
- [12] HONEYWELL INFORMATION SYSTEMS (Hrsg.):
Multics PL/I Language Specification.
Waltham, MA, USA: Honeywell Information Systems, März 1981.
(AG94-02)
- [13] INTEL CORPORATION (Hrsg.):
Intel Architecture Software Developer's Manual.
Order Number: 243191.
Santa Clara, California, USA: Intel Corporation, 1999



Literaturverzeichnis IV

- [14] KARGER, P. A. ; SCHELL, R. R.:
Thirty Years Later: Lessons from the Multics Security Evaluation.
In: *Proceedings of the 18th Annual Computer Security Applications Conference (CSAC 2002)*, IEEE Computer Society, 2002. –
ISBN 0–7695–1828–1, S. 119–126
- [15] LATHAM, D. C.:
Department of Defense Trusted Computer System Evaluation Criteria / Department of Defense.
1985 (DoD 5200.28-STD). –
Department of Defense Standard. –
Orange Book
- [16] LEVINE, J. R.:
Linkers & Loaders.
Morgan Kaufmann Publishers Inc., 1999 (Software Engineering and Programming)
- [17] MAYER, A. J. W.:
The Architecture of the Burroughs B5000: 20 Years Later and Still Ahead of the Times?
In: *ACM SIGARCH Computer Architecture News* 10 (1982), Jun., Nr. 4, S. 3–10



Literaturverzeichnis V

- [18] NIVELET, B. :
Multics Internet Server.
http://web.mit.edu/multics-history/source/Multics_Internet_Server/Multics_sources.html, 2006. –
Final Multics Release, MR 12.5, November 1992
- [19] NSA Public and Media Affairs:
National Security Agency Shares Security Enhancements to LINUX.
http://www.nsa.gov/public_info/press_room/2001/se-linux.shtml, Jan. 2001
- [20] ORGANICK, E. I.:
The Multics System: An Examination of its Structure.
MIT Press, 1972. –
ISBN 0–262–15012–3
- [21] OSSANNA, J. F. ; MIKUS, L. E. ; DUNTEN, S. D.:
Communications and Input/Output Switching in a Multiplex Computing System.
In: [23], S. 231–241
- [22] PANKHURST, R. J.:
Operating Systems: Program Overlay Techniques.
In: *Communications of the ACM* 11 (1968), Febr., Nr. 2, S. 119–125



Literaturverzeichnis VI

- [23] RECTOR, R. W. (Hrsg.):
Proceedings of the 1965 Fall Joint Computer Conference (AFIPS '65).
Bd. *Part I.*
New York, NY, USA : ACM, 1965
- [24] SANTA CRUZ OPERATION, INC. (Hrsg.):
System V Application Binary Interface.
Edition 4.1.
Santa Cruz, CA, USA: Santa Cruz Operation, Inc., März 1997
- [25] SUN MICROSYSTEMS, INC.:
Solaris SunOS.
1991. –
SunOS 5.0 Release Report
- [26] VYSSOTSKY, V. A. ; CORBATÓ, F. J. ; GRAHAM, R. M.:
Structure of the Multics Supervisor.
In: [23], S. 203–212

