

Betriebssystemtechnik

Adressräume: Trennung, Zugriff, Schutz

X. Mitbenutzung

Wolfgang Schröder-Preikschat

15. Juni 2018



Einleitung

Gemeinschaftssegmente

Allgemeines

Übertragungstechniken

Allgemeines

Prozessadressraumerzeugung

Nachrichtenversenden

Zusammenfassung



Informationsaustausch und gemeinsame Benutzung

Konsequenz der isolierten Adressräume sowie der physikalischen und logischen Verteilung von Funktionen

- die Prozesse kommunizieren über einen **Datenverbund** (*data sharing*)
 - alternativ oder ergänzend zum Nachrichtenversenden (*message passing*)
 - geeignet für den **Informationsaustausch** in homogenen Rechensystemen
 - als explizite Maßnahme in Systemen mit gemeinsamem Speicher *oder*
 - als implizite Optimierungsoption in Systemen mit verteiltem Speicher
- weitere Ergänzung ist ein **Textverbund** (*code sharing*) von Routinen
 - bspw. um eine **Gemeinschaftsbibliothek** (*shared library*) zu realisieren
 - aber auch als Optimierungsoption für Methoden der Prozesserzeugung
 - bei Prozessgabelung, ein für Elter- & Kindprozess gemeinsames Textsegment
 - ähnlich bei der Prozessüberlagerung mit einem Programm (insb. SASOS)

Thema sind grundlegende Konzepte zur kontrollierten Benutzung gemeinsamer Speicherbereiche durch Prozesse

- *shared memory segment, copy on write, copy on reference*



Einleitung

Gemeinschaftssegmente
Allgemeines

Übertragungstechniken
Allgemeines
Prozessadressraumerzeugung
Nachrichtenversenden

Zusammenfassung



Explizite Text- und Datenverbünde

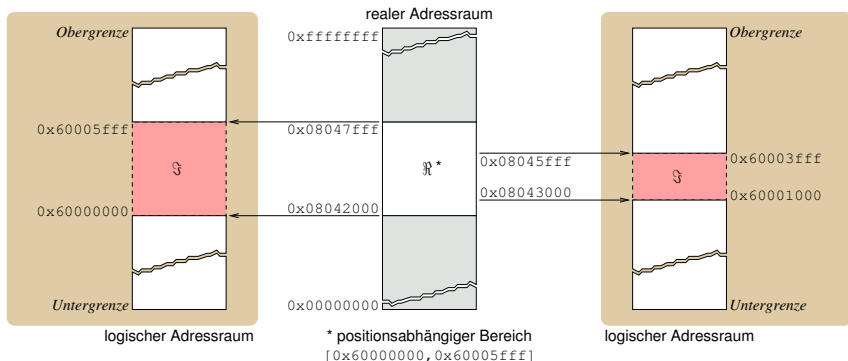
Mitbenutzung (*sharing*) von Text-/Datenbereichen durch Prozesse, die in voneinander getrennten Adressräumen residieren

- meint die $N : 1$ -Abbildung auf denselben realen Adressraumbereich
 - N Bereiche (\mathfrak{S}) in M logischen Adressräumen, $N \geq M$
 - $N = M \Rightarrow$ einfache
 - $N > M \Rightarrow$ mehrfache
- werden abgebildet auf 1 Bereich (\mathfrak{R}) des realen Adressraums
- wobei $\mathfrak{S} \subseteq \mathfrak{R}$, d.h., die Bereiche müssen nicht deckungsgleich sein
 - \mathfrak{S} kann kleiner, sollte aber nicht größer als \mathfrak{R} sein
 - \mathfrak{S} verschiedener logischer Adressräume:
 - i müssen denselben logischen Adressbereich abdecken *oder*
 - ii können verschiedenen Ausschnitten dieser Adressbereiche entsprechen
 - verschiedene \mathfrak{S} desselben logischen Adressraums können überlappen
- Ausrichtung (*alignment*) von \mathfrak{S} und \mathfrak{R} gemäß Granulatgröße
 - Segmentierung \mapsto Byte bzw. Block¹, Seitennummerierung \mapsto Seite

¹Vielfaches (Zweierpotenz) von Bytes, z.B. 16 Bytes für x86.

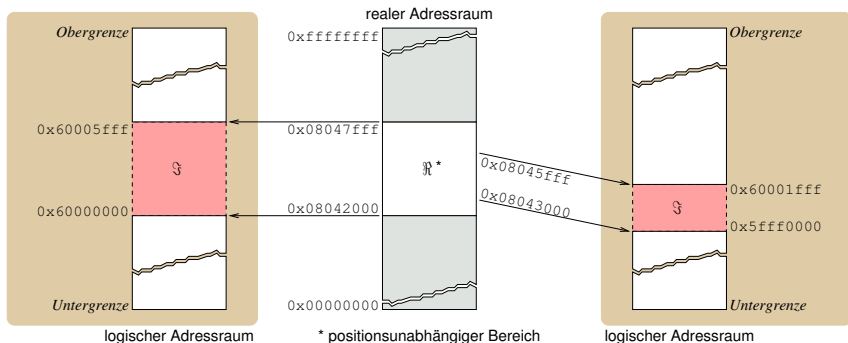


Positionsabhängige Mitbenutzung



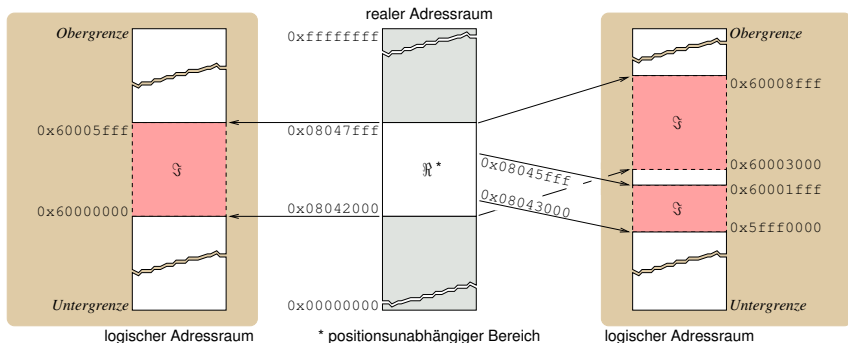
- totale Einblendung eines Bereichs \mathfrak{R} des realen Adressraums (li.)
 - Verwendung von absoluten Adressen bedingen Positionsabhängigkeit
 - Zeiger/Referenzen (Daten) und Sprungziele (Text)
 - Konsequenz entsprechend (vor-) gebundener Programmtexte
- partielle Einblendung desselben Bereichs \mathfrak{R} ist auch möglich (re.)





- totale/partielle Einblendung des Bereichs \mathfrak{R} des realen Adressraums
 - Verzicht auf absolute Adressen in \mathfrak{R} bedingt Positionsunabhängigkeit
 - zum Zeitpunkt der Einblendung von \mathfrak{R} in $\mathfrak{S} \rightsquigarrow$ **Migrationsin**transparenz
 - die Textbereiche in \mathfrak{R} enthalten ausschließlich **relative Adressen**
- **positionsunabhängiger Kode** (*position-independent code*, PIC; [2])





- mehrfache (total/partiell) Einblendung des Bereichs \mathfrak{R} (re.)
 - \mathfrak{R} liegt an verschiedenen Bereichen \mathfrak{S} im selben logischen Adressraum
 - z.B. verschiedene Datenstrukturen, die nacheinander referenziert werden
 - zum Datenzugriff werden auf \mathfrak{R} **verschieden breite Fenster** gelegt
- die Prozesse entscheiden selbst, wo der Bereich \mathfrak{R} eingeblendet wird



Mitbenutzung von Adressraumabschnitten bedingt einen **passenden Zuschnitt** von Text- und Datenbereichen

- Ausrichtung von \mathfrak{S} und \mathfrak{R} gemäß **Granulatgröße** (vgl. S. 5)
 - die Bereichsadresse ist byte-, block- oder seitenausgerichtet *und*
 - die Bereichslänge ist Vielfaches der Länge einer Ausrichtungseinheit

↪ Adressumsetzungseinheit (*memory management unit*, MMU)
- **Systemeigenschaft** der Text- bzw. Datenverbünde
 - statisch** ■ Symbole werden *vor* Laufzeit an Adressen gebunden
 - zur Programmlaufzeit ist die Bindung fest, unveränderlich
 - dynamisch** ■ Symbole werden *zur* Laufzeit an Adressen gebunden
 - *nach* der ersten Laufzeitreferenz ist die Bindung i. A. fest
- Positionierung der Objekt-/Lademodule als **Programmeigenschaft**
 - Positionsunabhängigkeit innerhalb des logischen Adressraums *oder*
 - feste Zuweisung von Adressbereichen, die das Betriebssystem vorgibt

↪ Betriebssystembelange, die auch in **Binder** und **Lader** verankert sind



Einleitung

Gemeinschaftssegmente
Allgemeines

Übertragungstechniken
Allgemeines
Prozessadressraumerzeugung
Nachrichtenversenden

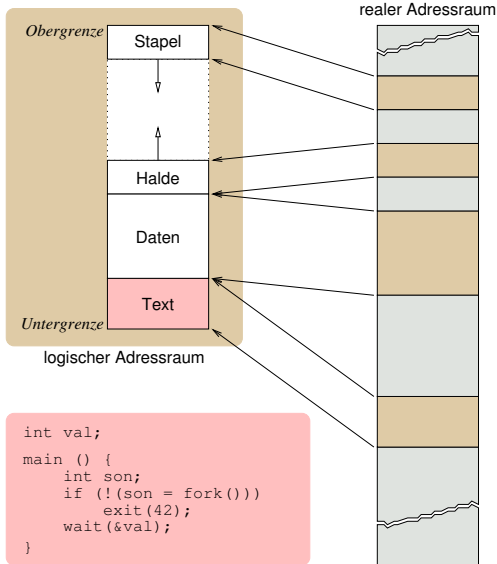
Zusammenfassung



Mitbenutzung von Text-/Datenbereichen durch Prozesse während der Durchführung von adressraumbezogenen Systemfunktionen

- automatisch, funktional transparent für die involvierten Prozesse
 - bspw. zum Duplizieren von Prozessinkarnationen (`fork`)
 - Initialisieren („nullen“) neu eingerichteter Adressräume oder
 - Zwischenpuffern oder Übertragen von Nachrichten/-bereichen
- wobei der gleichgestellte Prozess (*peer*) **implizites Wissen** besitzt
 - und zwar über die Existenz abgebildeter Objekte in seinem Adressraum
 - die ihm nur logisch, nicht aber real als „Eigentum“ überlassen wurden
- Übernahme: **kopieren beim Schreiben** (COW, *copy on write*, [1, 4, 3])
 - Schreibrechte wurden quellseitig entzogen und zielseitig nicht erteilt
 - **Momentaufnahme** (*snapshot*) des Zustands zum Schreibzeitpunkt
 - die Prozesse besitzen danach Schreibrechte auf Original und Kopie
- der schreibende Prozess legt eine **eigene Version** des Objektes an
 - gebunden an der originalen Adresse in seinem logischen Adressraum

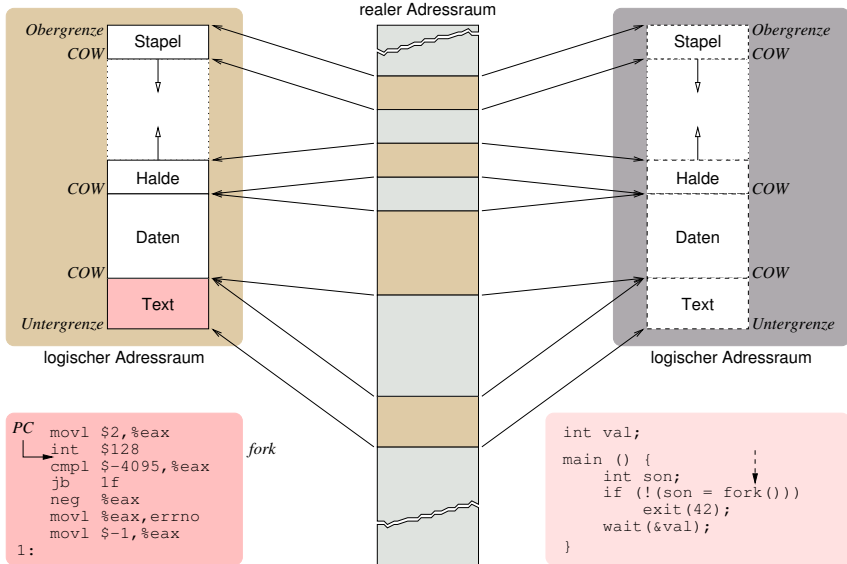


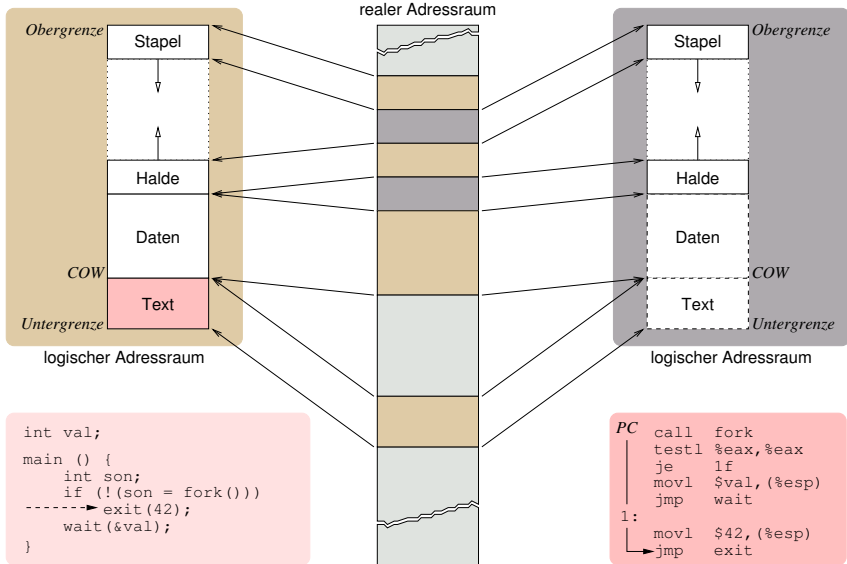


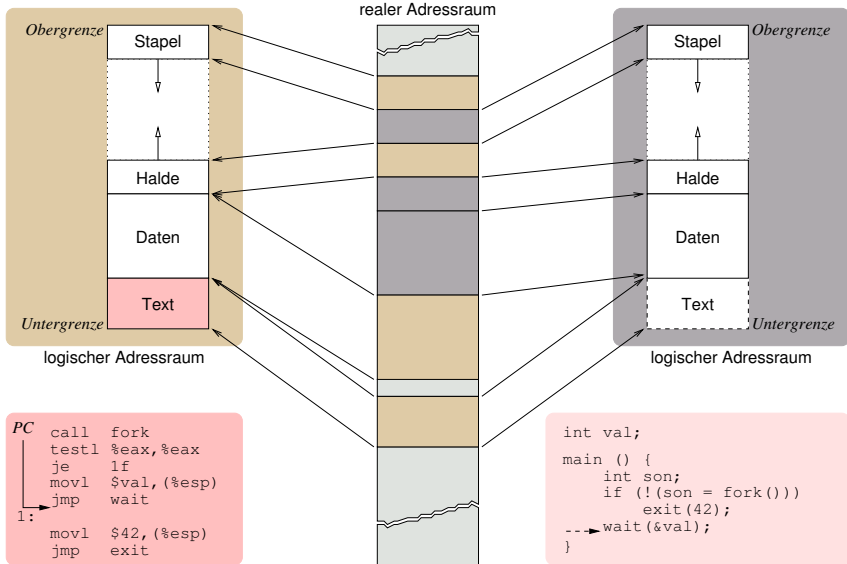
■ Klonen (fork)

- hier die Erzeugung eines baulich identischen Individuums von einem Prozess
- aus einem **Elterprozess** der Parentalgeneration ein **Kindprozess** der Filialgeneration erzeugen
- minimalinvasiv
- hier mit kleinstmöglichem Kopieraufwand in der Erzeugung eingreifend
- aber nur bedingt aufwandsarm...









Implizite Datenverbünde

Mitbenutzung von Datenbereichen durch Prozesse nach der logischen Bereitstellung im Rahmen von Weitergabeoperationen

- automatisch, funktional transparent für die involvierten Prozesse
 - bspw. zum Migrieren von Prozessinkarnationen auf ferne Rechner oder
 - Transferieren von Nachrichten bei der Interprozesskommunikation (IPC)
- Empfang: **kopieren beim Referenzieren** (COR, *copy on reference*, [5])
 - erstellen einer Objektabbildung in den Adressraum des Zugreifers
 - zweiseitig mit Lese-, Schreib- oder Ausführungsrechten versehen
 - quelseitig die Schreibrechte entzogen, d.h., auf COW umgestellt
- Kopplung mit IPC zur empfangsseitigen Einblendung der Nachricht
 - reliable-blocking send* ■ Annahme im BS, Sender gibt Verortung vor
 - synchronization send* ■ receive, Empfänger gibt Verortung ggf. vor
 - remote-invocation send* ■ receive, Empfänger gibt Verortung vor
 - aber auch explizit, zw. receive und reply
- bedarfsorientierter (*on-demand*) Datentransfer, auch netzwerkweit



```
1 site message::send (site afar) {
2     act *peer = stage::being(afar);           // identify receiver
3     if (peer) {                               // is valid
4         act *self = stage::being();          // identify sender
5         peer->serve(self->stock(*this));     // deliver message
6         self->block(&self->depot());         // await receive
7         return peer->label();                // send done
8     }
9     return -1;                               // send failed
10 }
```

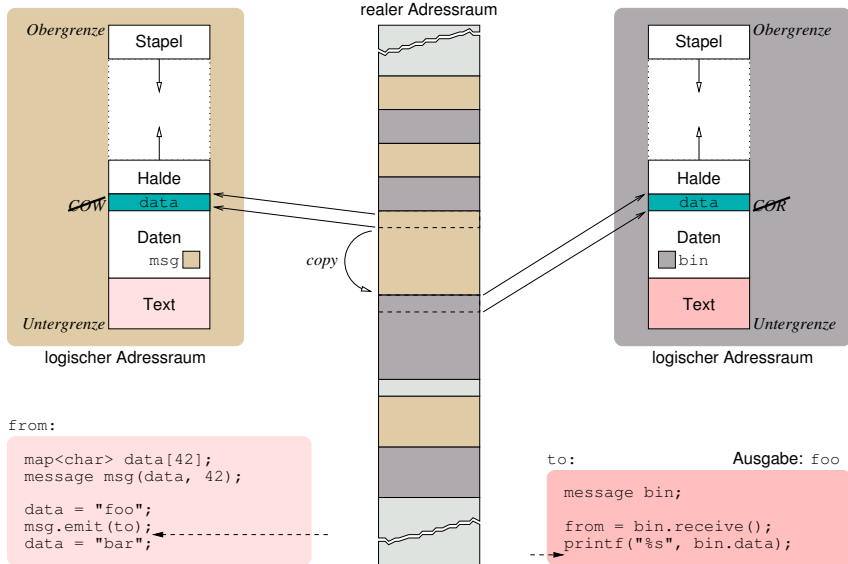
- direkte Kommunikation des Nachrichtendeskriptors durch den Sender
 - einlagern (stock) und zustellen (serve) des Deskriptors
 - Entleerung des Lagers durch den Empfänger abwarten (block)
- indirekte Kommunikation der Nachricht durch den Empfänger
 - erst bei Bedarf, durch COW (Sender) oder COR (Empfänger) übertragen
- Sender wird beim Empfang des Nachrichtendeskriptors deblockiert



```
1 int message::emit (site afar) {
2     act *peer = stage::being(afar);           // identify receiver
3     if (peer) {                               // is valid
4         act *self = stage::being();          // identify sender
5         area* area = peer->allot(*this, zone::COR, self);
6         if (area)                            // link COR/COW map
7             self->alter(*this, zone::COW, peer);
8         self->exert(self->depot());           // acquire buffer
9         peer->serve(self->stock(*this));     // deliver message
10        return self->depot().range();        // send done
11    }
12    return -1;                                // send failed
13 }
```

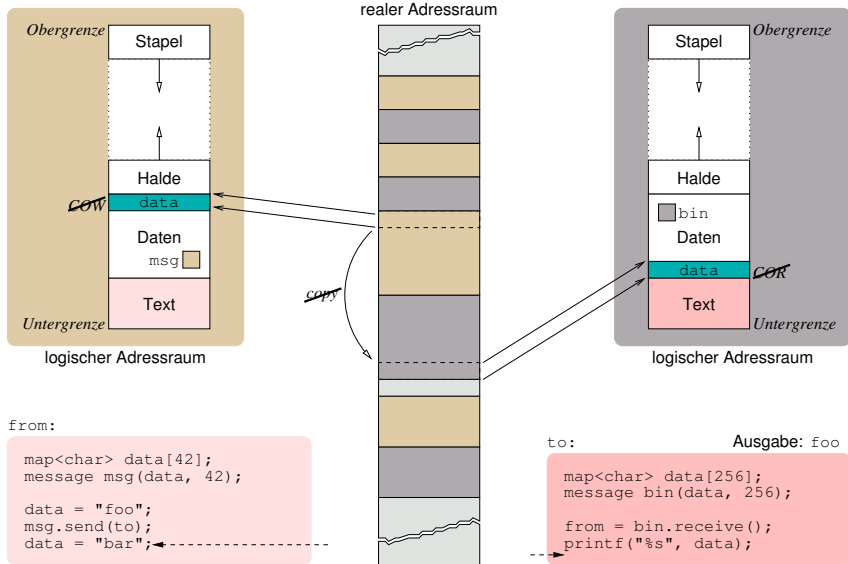
- wie send, jedoch **speicherabbildend** und **bedingt synchronisiert**
 - Sender reserviert „Platzhalterbereich“ im Adressraum des Empfängers
 - Zieladressbereich zuweisen (allot), Quelladressbereich abändern (alter)
 - Sendepuffer gebrauchen (exert): blockieren, falls dieser noch belegt ist





```
1 site message::receive () {
2     act *self = stage::being();           // identify receiver
3     lump* item = self->glean();           // accept message
4     if (item) {                           // message received
5         act *peer = stage::being(item->label());
6         if (peer) {                       // valid sender
7             if (size == 0) *(section*)this = *item;
8             area* area = self->allot(*this, zone::COR, peer);
9             if (area) {                   // message mapped
10                peer->alter(*item, zone::COW, self);
11                peer->quote(peer->depot()); // release buffer
12                return item->label();      // receive done
13            }
14            self->retry(*item);           // keep message
15        }
16    }
17    return 0;                             // receive failed
18 }
```





Muster der Empfangs-, Verarbeitungs- und Antwortschleife für eine **Diensteinheit** (*server*) insb. mikrokernbasierter Betriebssysteme

```
1 void serve () {
2     request msg;
3     for (;;) {
4         site peer = msg.receive(); // blocking acceptance
5         if (peer) {
6             zone area(peer, msg.sort.part[0]);
7             msg.sort.part[0].size = area.apply();
8             msg.reply(peer);          // non-blocking response
9         }
10    }
11 }
```

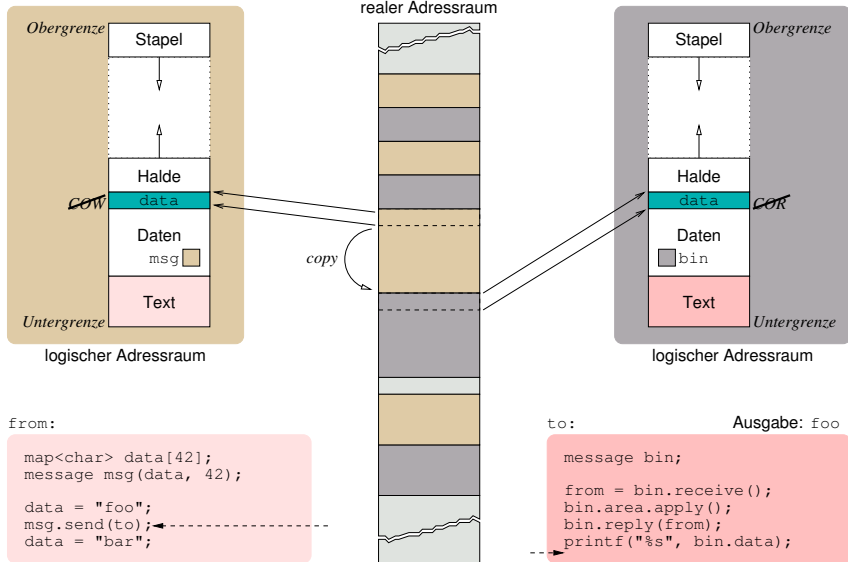
- Empfang (*receive*) des Auftrags als Nachrichtendeskriptor
- Nachrichtenverarbeitung durch Anwendung (*apply*) des Deskriptors
- Rückantwort (*reply*) und Übermittlung einer Ergebnismessage



```
1 int zone::apply () {
2     act *peer = stage::being(label()); // identify callee
3     if (peer) {                          // is valid
4         act *self = stage::being();      // identify caller
5         area* area = self->allot(*this, zone::COR, peer);
6         if (area) {                      // link COR/COW map
7             peer->alter(*this, zone::COW, self);
8             return range();              // mapping done
9         }
10    }
11    return -1;                            // mapping failed
12 }
```

- analog zu emit oder receive, jedoch explizit im Anwendungsprozess
 - Reservierung eines „Platzhalterbereichs“ im Adressraum des Aufrufers
 - Zieladressbereich zuweisen (allot), Quelladressbereich abändern (alter)
 - zuvor das Wesen (being, Zeile 2) des Quelladressbereichs validieren
- der Bereich (zone) im Quelladressraum wurde geeignet kommuniziert





Einleitung

Gemeinschaftssegmente

Allgemeines

Übertragungstechniken

Allgemeines

Prozessadressraumerzeugung

Nachrichtenversenden

Zusammenfassung



- Informationsaustausch und gemeinsame Benutzung
 - Konsequenz der isolierten Adressräume, logische Funktionsverteilung
 - Interprozesskommunikation über einen Datenverbund (*data sharing*)
 - Gemeinschaftsbibliothek (*shared library*) als Textverbund (*code sharing*)
 - Gemeinschaftssegment (*shared memory segment*)
 - explizite Text- und Datenverbünde ursprünglich getrennter Adressräume
 - positionsabhängige/-unabhängige Mitbenutzung
 - Mitbenutzung heißt auch passender Zuschnitt der Text-/Datenbereiche
 - Ausrichtung gemäß Granulatgröße: byte-, block-, seitenausgerichtet
 - Bereichslänge ist Vielfaches der Länge einer Ausrichtungseinheit
 - Text-/Datenverbünde haben statische/dynamische Systemeigenschaften
 - ein Frage der Bindezeit von Symbol und Adresse: vor/zur Laufzeit
 - Übertragungstechniken: *copy on write*, *copy on reference*
 - Prozessadressraumerzeugung (*fork*) als ein Beispiel für COW
 - Interprozesskommunikation (IPC) als ein Fall für COW und COR
- ↔ insb. auch netzwerkweit, d.h., Rechengrenzen überschreitend



- [1] BALL, J. E. ; FELDMAN, J. ; LOW, J. R. ; RASHID, R. ; ROVNER, P. :
RIG, Rochester's Intelligent Gateway: System Overview.
In: *IEEE Transactions on Software Engineering SE-2* (1976), Nov., Nr. 4, S. 321–328
- [2] IBM CORPORATION:
IBM Time-Sharing System/360: Concepts and Facilities.
White Plains, NY, USA, 1967 (Z20-1788-0). –
Sales and Systems Guide
- [3] RASHID, R. F.:
From RIG to Accent to Mach: The Evolution of a Network Operating System.
In: WINKLER, S. (Hrsg.) ; STONE, H. S. (Hrsg.): *Proceedings of the 1986 ACM Fall Joint Computer Conference*, IEEE Computer Society Press, 1986. –
ISBN 0-8186-4743-4, S. 1128–1137
- [4] RASHID, R. F. ; ROBERTSON, G. G.:
Accent: A Communication Oriented Network Operating System Kernel.
In: HOWARD, J. (Hrsg.) ; REED, D. P. (Hrsg.): *Proceedings of the Eighth ACM Symposium on Operating System Principles (SOSP '81)*, ACM Press, 1981. –
ISBN 0-89791-062-1, S. 64–75



- [5] ZAYAS, E. R.:
Attaching the Process Migration Bottleneck.
In: BÉLÁDY, L. (Hrsg.): *Proceedings of the Eleventh ACM Symposium on Operating System Principles (SOSP '87)*, ACM Press, 1987. –
ISBN 0-89791-242-X, S. 13-24



```
1 class stage {
2     static act* life;           // currently running thread
3     static unsigned mask;      // limit of following array
4     static act list[];         // thread descriptor table
5 public:
6     static act* being()        { return life; }
7     static act* being(site slot) {
8         act* item = &list[slot & mask];
9         return item->label() == slot ? item : 0;
10    }
11 };
```

- einelementige Menge (*singleton*) zentraler Datenstrukturen

- life* ■ wird vom Umschalter (*dispatcher*) aktualisiert

- list* ■ Prozesstabelle dynamischer/konfigurierbarer Größe

- wird zur Systeminitialisierungs oder -laufzeit dimensioniert

- mask* ■ zur Maskierung von Prozessnamen bei der Tabellenindizierung

- Abstraktion für **prozessor(kern)lokale Systemdaten und -operationen**



```
1 class act : public lockbox, public scope {
2     int trim;           // mood, scheduling state
3     void* wait;        // blocked-on event
4 public:
5     enum mood {READY, RUNNING, BLOCKED};
6
7     void block ();     // release processor
8     void ready ();    // compete for processor
9
10    void apply (void*); // set blocked-on event
11
12    void block (void*); // fall asleep until given event
13    void rouse (void*); // awake, if blocked on event
14
15    lump* glean ();    // await inbox message
16    int serve (lump&); // awake inbox-blocked act
17    int exert (lump&); // await outbox message
18    int quote (lump&); // awake outbox-blocked act
19 };
```



Prozesssteuerung für IPC

```
1  inline void act::apply (void* link) { wait = link; }
2
3  inline void act::block (void* link) {
4      apply(link);           // set blocked-on event and
5      block();              // release processor
6  }
7
8  inline void act::rouse (void* link) {
9      if (wait == link) {   // the right wakeup event?
10         wait = 0;         // yes, cancel and
11         ready();         // compete for processor
12     }
13 }
14
15 inline lump* act::glean () {
16     lump* item = clear(); // remove next message descriptor from inbox
17     if (!item) {         // none available, prepare to block
18         block(&booth()); // await specified event
19         if (!wait)       // no exceptional wakeup
20             item = clear(); // there must be a message descriptor
21     }
22     return item;
23 }
24
25 inline int act::exert (lump& item) {
26     if (item.range() != 0) // outbox available, i.e., not in use?
27         block(&item);     // no, await release event (quote)
28 }
29
30 inline int act::quote (lump& item) { range(0); rouse(&item); }
31 inline int act::serve (lump& item) { aback(item); rouse(&booth()); }
```



- Postfach (*lockbox*) mit Ein- und Ausgang

```
1 class lockbox : public lump {
2     queue inbox;
3 public:
4     lump& depot () { return *this; }
5     queue& booth () { return inbox; }
6     lump* clear () { return (lump*)inbox.clear(); }
7     void aback (lump& item) { inbox.aback(item); }
8     void retry (lump& item) { aback(item); }
9 };
```

- im Postfach lagerungsfähige Posten (*lump*)

```
1 class lump : public chain, public zone {
2 public:
3     lump& stock (const section&); // keep section
4     lump& operator= (const section&); // stock it
5 };
```

- Warteschlange (*queue*) und Verkettung (*chain*)

↪ VL 9



- standortbezogener Bereich (*zone*) im Adressraum

```
1 class zone : public area {
2     site name;           // thread identification
3 public:
4     enum mode {COW, COR};
5
6     site label ()        { return name; }
7     site label (site name) { this->name = name; }
8     int  apply ();      // map zone to address space
9
10    zone (site name, const section& slot) {
11        label(name);
12        *(section*)this = slot;
13    }
14 };
```

- systemweit eindeutige Bezeichnung eines Adressbereiches



■ Geltungsbereich (*scope*) des Adressraums

```
1 class scope {
2   public:
3     area* allot (area&, int, act*); // map address range
4     int  alter (area&, int, act*); // modify mapping
5 };
```

■ Bereich (*area*) im Adressraum

```
1 class area : public section {
2   public:
3     int  range ()           { return size; }
4     void range (int size) { this->size = size; }
5     void clear ()          { range(0); }
6 };
```

■ Standort (*site*)

↪ VL 9

