Betriebssystemtechnik

Adressräume: Trennung, Zugriff, Schutz

VII. Sprachbasierte Systeme

SS 2026

Wolfgang Schröder-Preikschat / Volkmar Sieh

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Technische Fakultät

Gliederung

Isolation schafft Immunität

- Eigenschaften, um Sicherheit in einem Rechensystem zu fördern:

Immunität
- Angriffssicherheit (*security*)
- Schutz einer Entität vor seiner Umgebung
- verhindern, in einen Adressraum eindringen zu können

Isolation
- Betriebssicherheit (*safety*)
- Schutz der Umgebung vor einer Entität
- verhindern, aus einem Adressraum ausbrechen zu können

Isolation schafft Immunität

- Eigenschaften, um Sicherheit in einem Rechensystem zu fördern:

Immunität
- Angriffssicherheit (*security*)
- Schutz einer Entität vor seiner Umgebung
- verhindern, in einen Adressraum eindringen zu können

Isolation
- Betriebssicherheit (*safety*)
- Schutz der Umgebung vor einer Entität
- verhindern, aus einem Adressraum ausbrechen zu können

- beide Eigenschaften bedingen einander:

  Indem das System verhindert, dass Prozesse aus ihren Adressräumen ausbrechen können, wird eben auch verhindert, dass Prozesse in andere Adressräume einbrechen können.

Isolation schafft Immunität

- Eigenschaften, um Sicherheit in einem Rechensystem zu fördern:

  Immunität
  - Angriffssicherheit (*security*)
  - Schutz einer Entität vor seiner Umgebung
  - verhindern, in einen Adressraum eindringen zu können

  Isolation
  - Betriebssicherheit (*safety*)
  - Schutz der Umgebung vor einer Entität
  - verhindern, aus einem Adressraum ausbrechen zu können

- beide Eigenschaften bedingen einander:

  Indem das System verhindert, dass Prozesse aus ihren Adressräumen ausbrechen können, wird eben auch verhindert, dass Prozesse in andere Adressräume einbrechen können.

- damit kommt Betriebssicherheit jedoch nicht vor Angriffssicherheit
  - erstere erfordert Funktionen, die zweitere nicht benötigt – Termintreue
  - Schutz in räumlicher Hinsicht ist nur ein Aspekt – Zeit ein anderer

- aber umgekehrt wird eher ein Schuh draus…

Gliederung

Isolation erreicht durch Typsicherheit

Isolation erreicht durch Typsicherheit

- setzt auf eine typsichere Programmiersprache samt Kompilierer
  - falsche Verwendung von Datentypen kann Typverletzungen hervorrufen
  - die resultierenden Typfehler werden spätestens zur Laufzeit erkannt

Isolation erreicht durch Typsicherheit

- setzt auf eine typsichere Programmiersprache samt Kompilierer
  - falsche Verwendung von Datentypen kann Typverletzungen hervorrufen
  - die resultierenden Typfehler werden spätestens zur Laufzeit erkannt
- zentrale Maßnahme dabei ist die Typprüfung (*type checking*)
  - prüft die zum Typsystem konforme Verwendung der Datentypen

    zur Kompilierungszeit $\rightsquigarrow$ statisch typisierte Sprache (i.A. signierte Binaries)

    zur Laufzeit $\rightsquigarrow$ dynamisch typisierte Sprache (Überprüfung Teil des JIT-Compiler-Laufs)
  - ggf. auch in Kombination: was geht, statisch, ansonsten dynamisch

Isolation erreicht durch Typsicherheit

- offensichtliches Problem bereiten Zeiger (*pointer*) als Datentypen

Isolation erreicht durch Typsicherheit

- offensichtliches Problem bereiten Zeiger (*pointer*) als Datentypen:

  typisierte Zeiger

  - können dereferenziert und verändert werden
  - Typ des Zeigers ist der Typ, auf den er verweist
  - `char*` $\neq$ `unsigned*` $\neq$ `int*` $\neq$ `float*`

Isolation erreicht durch Typsicherheit

- offensichtliches Problem bereiten Zeiger (*pointer*) als Datentypen:

typisierte Zeiger
- können dereferenziert und verändert werden
- Typ des Zeigers ist der Typ, auf den er verweist
- `char*` ≠ `unsigned*` ≠ `int*` ≠ `float*`

untypisierte Zeiger
- auf ihnen sind keine Operationen definiert
- `POINTER` in Pascal/Modula, `void*` in C/C++

Isolation erreicht durch Typsicherheit

- offensichtliches Problem bereiten Zeiger (*pointer*) als Datentypen:

  typisierte Zeiger
  - können dereferenziert und verändert werden
  - Typ des Zeigers ist der Typ, auf den er verweist
  - `char*` ≠ `unsigned*` ≠ `int*` ≠ `float*`

  untypisierte Zeiger
  - auf ihnen sind keine Operationen definiert
  - `POINTER` in Pascal/Modula, `void*` in C/C++

- Adressraumausbrüche sind aber auch ohne (explizite) Zeiger möglich

Isolation gefährdende Schwachstellen

- Beispiele kritischer Sprachkonzepte als Mechanismen zum Ausbruch

Feld	Über-/Unterschreitung von Feldgrenzen
Zeiger	Wertezuweisungen an/Änderungen von Zeigervariablen
Rekursion	Laufzeitstapel unberechenbar ausdehnen
Argumente	Übergabe von beliebigen Parametern
Typisierung	Zahlenwert als Adresse auslegen (*typecast*)
Außenreferenz	beliebiges (externes) Unterprogramm aufrufen

- Beispiele kritischer Sprachkonzepte als Mechanismen zum Ausbruch:

Feld	Über-/Unterschreitung von Feldgrenzen
Zeiger	Wertezuweisungen an/Änderungen von Zeigervariablen
Rekursion	Laufzeitstapel unberechenbar ausdehnen
Argumente	Übergabe von beliebigen Parametern
Typisierung	Zahlenwert als Adresse auslegen (*typecast*)
Außenreferenz	beliebiges (externes) Unterprogramm aufrufen

- Beispiele kritischer Sprachkonzepte als Mechanismen zum Ausbruch:

Feld	Über-/Unterschreitung von Feldgrenzen
Zeiger	Wertezuweisungen an/Änderungen von Zeigervariablen
Rekursion	Laufzeitstapel unberechenbar ausdehnen
Argumente	Übergabe von beliebigen Parametern
Typisierung	Zahlenwert als Adresse auslegen (*typecast*)
Außenreferenz	beliebiges (externes) Unterprogramm aufrufen

- Beispiele kritischer Sprachkonzepte als Mechanismen zum Ausbruch:

Feld	Über-/Unterschreitung von Feldgrenzen
Zeiger	Wertezuweisungen an/Änderungen von Zeigervariablen
Rekursion	Laufzeitstapel unberechenbar ausdehnen
Argumente	Übergabe von beliebigen Parametern
Typisierung	Zahlenwert als Adresse auslegen (*typecast*)
Außenreferenz	beliebiges (externes) Unterprogramm aufrufen

Isolation gefährdende Schwachstellen

- Beispiele kritischer Sprachkonzepte als Mechanismen zum Ausbruch:

Feld	Über-/Unterschreitung von Feldgrenzen
Zeiger	Wertezuweisungen an/Änderungen von Zeigervariablen
Rekursion	Laufzeitstapel unberechenbar ausdehnen
Argumente	Übergabe von beliebigen Parametern
Typisierung	Zahlenwert als Adresse auslegen (*typecast*)
Außenreferenz	beliebiges (externes) Unterprogramm aufrufen

Isolation gefährdende Schwachstellen

- Beispiele kritischer Sprachkonzepte als Mechanismen zum Ausbruch:

Feld	Über-/Unterschreitung von Feldgrenzen
Zeiger	Wertezuweisungen an/Änderungen von Zeigervariablen
Rekursion	Laufzeitstapel unberechenbar ausdehnen
Argumente	Übergabe von beliebigen Parametern
Typisierung	Zahlenwert als Adresse auslegen (*typecast*)
Außenreferenz	beliebiges (externes) Unterprogramm aufrufen

- Beispiele kritischer Sprachkonzepte als Mechanismen zum Ausbruch:

Feld	▪ Über-/Unterschreitung von Feldgrenzen
Zeiger	▪ Wertezuweisungen an/Änderungen von Zeigervariablen
Rekursion	▪ Laufzeitstapel unberechenbar ausdehnen
Argumente	▪ Übergabe von beliebigen Parametern
Typisierung	▪ Zahlenwert als Adresse auslegen (*typecast*)
Außenreferenz	▪ beliebiges (externes) Unterprogramm aufrufen

Isolation gefährdende Schwachstellen

- Beispiele kritischer Sprachkonzepte als Mechanismen zum Ausbruch:

Feld	Über-/Unterschreitung von Feldgrenzen
Zeiger	Wertezuweisungen an/Änderungen von Zeigervariablen
Rekursion	Laufzeitstapel unberechenbar ausdehnen
Argumente	Übergabe von beliebigen Parametern
Typisierung	Zahlenwert als Adresse auslegen (*typecast*)
Außenreferenz	beliebiges (externes) Unterprogramm aufrufen

- zusätzlich notwendig: Garbage Collection bzw. Referenzzähler

Hinweise

- Was z.B. gehen sollte:

```
1  /* Directory Entry */
2  struct dentry {
3      short inode;
4      char name[14];
5  };
6
7  /* Disk Block */
8  char buf[512];
9
10 /* Read block from disk. */
11 readblk(..., buf);
12
13 /* Parse directory block. */
14 for (struct dentry *de = (struct dentry *) &buf[0];
15     de < (struct dentry *) &buf[sizeof(buf)];
16     de++) {
17         ...
18 }
```

Hinweise

- Ähnlich:
  - Blöcke *bekannter* Länge von Bytes auf Platte, im Speicher, ...
  - Pakete *unbekannter* Länge von Bytes über Ethernet, ...
- „darüber gelegte” Strukturen
  - Directories, Inodes, Page-/Segment-Deskriptoren, ...
  - IP/ICMP/UDP/TCP-Header und -Payload, ...
- *wichtig*:
  - „darüber gelegte” Strukturen dürfen über das Ende der darunter liegenden Blöcke/Pakete nicht hinausreichen
  - Strukturen dürfen keine Zeiger enthalten

Hinweise

- Ähnlich:
  - Blöcke *bekannter* Länge von Bytes auf Platte, im Speicher, ...
  - Pakete *unbekannter* Länge von Bytes über Ethernet, ...
- „darüber gelegte" Strukturen
  - Directories, Inodes, Page-/Segment-Deskriptoren, ...
  - IP/ICMP/UDP/TCP-Header und -Payload, ...
- *wichtig*:
  - „darüber gelegte" Strukturen dürfen über das Ende der darunter liegenden Blöcke/Pakete nicht hinausreichen
  - Strukturen dürfen keine Zeiger enthalten

- Forschungsbedarf...!

- die den Ausbruch ggf. bedingende Intention eines Subjektes:

unbeabsichtigt
- Soft- oder Hardwarefehler (vgl. auch [3])[1]

beabsichtigt
- Schadsoftware jeglicher Herkunft und Art

[1] Kein technisches System ist 100 % fehlerfrei.

- die den Ausbruch ggf. bedingende Intention eines Subjektes:
  - **unbeabsichtigt**
    - Soft- oder Hardwarefehler (vgl. auch [3])[1]
  - **beabsichtigt**
    - Schadsoftware jeglicher Herkunft und Art

- sichere Programmiersprachen sind frei von solchen Konzepten oder ihr Übersetzer bietet Wege für deren Absicherung
  - Simula, Mesa, Ada, Modula-3, Oberon, Java, Clay

[1] Kein technisches System ist 100 % fehlerfrei.

- die den Ausbruch ggf. bedingende Intention eines Subjektes:
  - **unbeabsichtigt**
    - Soft- oder Hardwarefehler (vgl. auch [3])[1]
  - **beabsichtigt**
    - Schadsoftware jeglicher Herkunft und Art

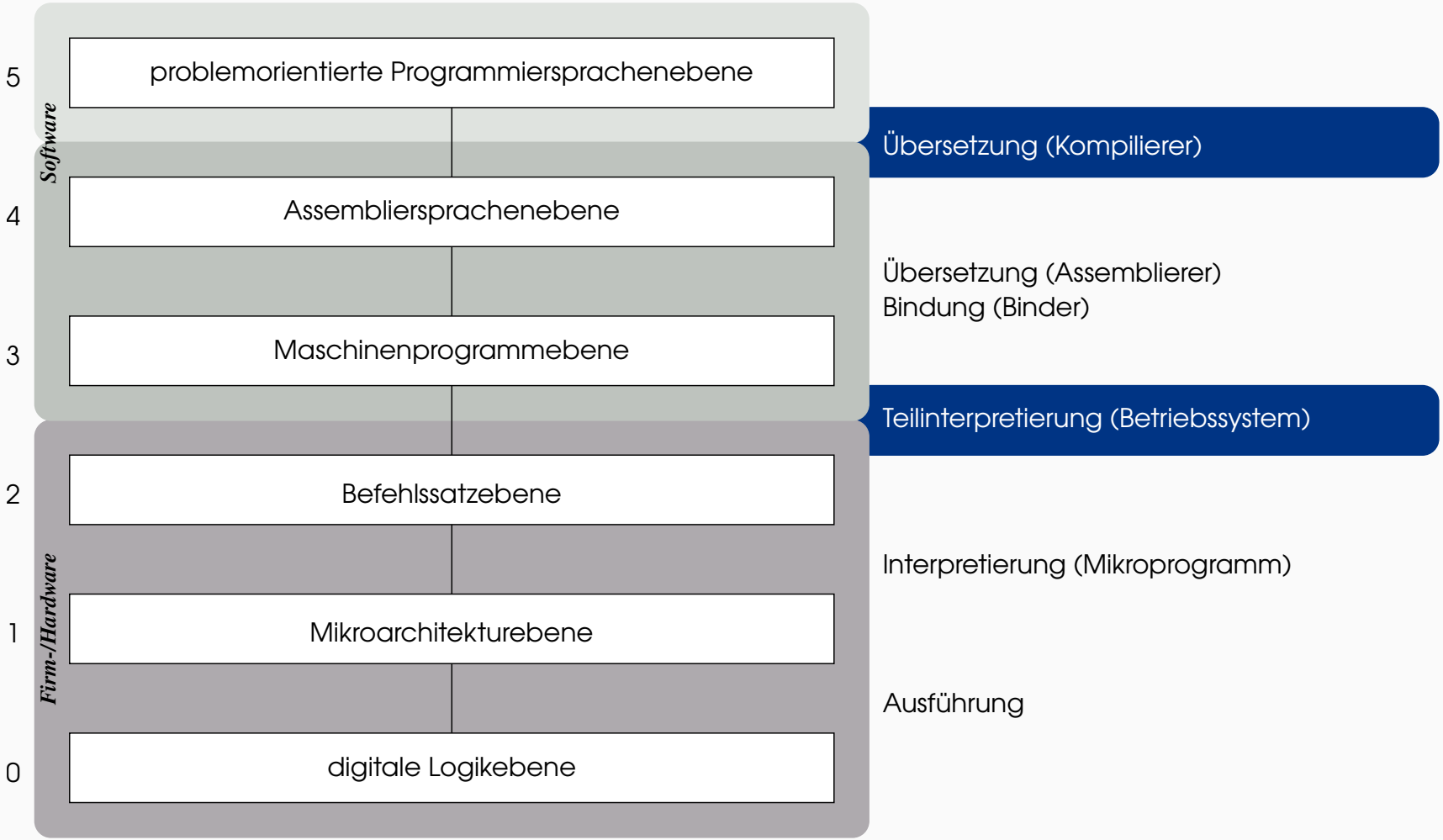
- sichere Programmiersprachen sind frei von solchen Konzepten oder ihr Übersetzer bietet Wege für deren Absicherung
  - Simula, Mesa, Ada, Modula-3, Oberon, Java, Clay

- die Eignung als Systemprogrammiersprache ist damit aber noch offen

[1] Kein technisches System ist 100 % fehlerfrei.

Stärke durch Zusammenarbeit

(vgl. auch [15])



- Trennung von Belangen (*separation of concerns*) $\Longleftrightarrow$ Körnigkeit

SPIN [2, S. 276]

- *Operating system protection structures are not the right level to impose modularity.*
- *In fact, protection structures do not impose modularity; they only enforce selected module boundaries.*

SPIN [2, S. 276]

- *Operating system protection structures are not the right level to impose modularity.*
- *In fact, protection structures do not impose modularity; they only enforce selected module boundaries.*

- ein Modul, nach [10, S. 1056], vereint Methoden, die:
  i. einen Hauptschritt in der Verarbeitung ausmachen *oder*
  ii. dem Geheimnisprinzip (*information hiding*, [9]) folgen

SPIN [2, S. 276]

- *Operating system protection structures are not the right level to impose modularity.*
- *In fact, protection structures do not impose modularity; they only enforce selected module boundaries.*

- ein Modul, nach [10, S. 1056], vereint Methoden, die:
  i. einen Hauptschritt in der Verarbeitung ausmachen *oder*
  ii. dem Geheimnisprinzip (*information hiding*, [9]) folgen
- allg. wird Geheimnisprinzip als das Merkmal von Moduln verstanden
  - oft auch mit Datenkapselung (*data encapsulation*) gleichgesetzt

SPIN [2, S. 276]

- *Operating system protection structures are not the right level to impose modularity.*
- *In fact, protection structures do not impose modularity; they only enforce selected module boundaries.*

- ein Modul, nach [10, S. 1056], vereint Methoden, die:
  i. einen Hauptschritt in der Verarbeitung ausmachen *oder*
  ii. dem Geheimnisprinzip (*information hiding*, [9]) folgen
- allg. wird Geheimnisprinzip als das Merkmal von Moduln verstanden
  - oft auch mit Datenkapselung (*data encapsulation*) gleichgesetzt

- all dies sind Aspekte der Softwaretechnik, die durch Schutzkonzepte eines Betriebssystems bestenfalls unterstützt werden können

Lipto [4, S. 512–513]

The fundamental reasons for providing support for modularity that is independent of protection are

- *that it allows modular decomposition without concerns for cross-domain communication costs, and*
- *the partitioning of functions into protection domains becomes a matter of configuration rather than design.*

Lipto [4, S. 512–513]

The fundamental reasons for providing support for modularity that is independent of protection are

- *that it allows modular decomposition without concerns for cross-domain communication costs, and*
- *the partitioning of functions into protection domains becomes a matter of configuration rather than design.*

- nach [10, S. 1053] bedeutet modulare Programmierung zweierlei:
  i. die Entwicklung eines Moduls ist ohne (viel) Wissen über das Innenleben anderer Module möglich *und*
  ii. Umstrukturierung und Austausch eines Moduls ist möglich, ohne das Gesamtsystem umstrukturieren zu müssen

Lipto [4, S. 512–513]

The fundamental reasons for providing support for modularity that is independent of protection are

- *that it allows modular decomposition without concerns for cross-domain communication costs, and*
- *the partitioning of functions into protection domains becomes a matter of configuration rather than design.*

- nach [10, S. 1053] bedeutet modulare Programmierung zweierlei:
  i. die Entwicklung eines Moduls ist ohne (viel) Wissen über das Innenleben anderer Module möglich *und*
  ii. Umstrukturierung und Austausch eines Moduls ist möglich, ohne das Gesamtsystem umstrukturieren zu müssen
- Schutz allein erzwingt keine Softwarestruktur dieser Eigenschaften

Schutz und Granularität

SPIN [2, S. 278]

- *Storage allocation, protection, and reclamation should be coarse grained at the operating system level.*
- *Fine-grained control is best provided at the language level by compilers and runtime systems.*

Schutz und Granularität

SPIN [2, S. 278]

- *Storage allocation, protection, and reclamation should be coarse grained at the operating system level.*
- *Fine-grained control is best provided at the language level by compilers and runtime systems.*

- ein Grundprinzip bei der Konstruierung von Rechensystemen
  - feinkörnig greifende Maßnahmen eher „nach oben“ positionieren
    - Halde (`malloc`, `free`): „typweise“ Speicherverwaltung
    - gepufferte Ein-/Ausgabe (`fread`, `fwrite`), Programmfäden, …
  - grobkörnig greifende eher “nach unten“ in der Hierarchie orientieren
    - `sbrk`: kachel-/segmentweise Speicherverwaltung
    - ungepufferte Ein-/Ausgabe (`read`, `write`), Prozesse, …

Schutz und Granularität

SPIN [2, S. 278]

- *Storage allocation, protection, and reclamation should be coarse grained at the operating system level.*
- *Fine-grained control is best provided at the language level by compilers and runtime systems.*

- ein Grundprinzip bei der Konstruierung von Rechensystemen
  - feinkörnig greifende Maßnahmen eher „nach oben“ positionieren
    - Halde (`malloc`, `free`): „typweise“ Speicherverwaltung
    - gepufferte Ein-/Ausgabe (`fread`, `fwrite`), Programmfäden, …
  - grobkörnig greifende eher “nach unten“ in der Hierarchie orientieren
    - `sbrk`: kachel-/segmentweise Speicherverwaltung
    - ungepufferte Ein-/Ausgabe (`read`, `write`), Prozesse, …
- es lässt Betriebssysteme als performante Konstruktion erscheinen

Schutz und Typsicherheit

SPIN [2, S. 278]

> *Even with safe languages, the operating system must still support hard protection boundaries in order to separate nontrusting parties and different safe or unsafe language environments.*

SPIN [2, S. 278]

> *Even with safe languages, the operating system must still support hard protection boundaries in order to separate nontrusting parties and different safe or unsafe language environments.*

- schwaches Argument, da es nicht in der Betriebsart differenziert
  - trifft zu nur unter folgenden Annahmen:
    i. das Rechensystem ist ein Mehrsprachensystem
    ii. verschiedene Arten von Dialogbetrieb werden gefahren
    iii. ein Universalrechner (*general-purpose computer*) ist zu betreiben
  - insb. für Spezialrechner (*special-purpose computer*) gilt dies nicht

Schutz und Typsicherheit

SPIN [2, S. 278]

Even with safe languages, the operating system must still support hard protection boundaries in order to separate nontrusting parties and different safe or unsafe language environments.

- schwaches Argument, da es nicht in der Betriebsart differenziert
  - trifft zu nur unter folgenden Annahmen:
    i. das Rechensystem ist ein Mehrsprachensystem
    ii. verschiedene Arten von Dialogbetrieb werden gefahren
    iii. ein Universalrechner (*general-purpose computer*) ist zu betreiben
  - insb. für Spezialrechner (*special-purpose computer*) gilt dies nicht
- starkes Argument, wenn man auf Aushärtung (*hardening*) setzt
  - um Robustheit gegenüber sporadischer Hardwarefehler zu erhöhen

SPIN [2, S. 278]

> *Even with safe languages, the operating system must still support hard protection boundaries in order to separate nontrusting parties and different safe or unsafe language environments.*

- schwaches Argument, da es nicht in der Betriebsart differenziert
  - trifft zu nur unter folgenden Annahmen:
    i. das Rechensystem ist ein Mehrsprachensystem
    ii. verschiedene Arten von Dialogbetrieb werden gefahren
    iii. ein Universalrechner (*general-purpose computer*) ist zu betreiben
  - insb. für Spezialrechner (*special-purpose computer*) gilt dies nicht
- starkes Argument, wenn man auf Aushärtung (*hardening*) setzt
  - um Robustheit gegenüber sporadischer Hardwarefehler zu erhöhen
- beide Ansätze ergänzen sich, sie schließen sich überhaupt nicht aus **!**

Beispiele sprachbasierter Betriebssysteme

- in chronologischer Reihenfolge (1961 – 2009):

MCP [8]

- ESPOL, später (1970) NEWP
- Einsprachen-/Multiprozessorsystem, Stapelbetrieb

Beispiele sprachbasierter Betriebssysteme

- in chronologischer Reihenfolge (1961 – 2009):

MCP [8]
- ESPOL, später (1970) NEWP
- Einsprachen-/Multiprozessorsystem, Stapelbetrieb

Pilot [12]
- Mesa
- Einsprachen-/Einbenutzer-/Mehrprozesssystem

Beispiele sprachbasierter Betriebssysteme

- in chronologischer Reihenfolge (1961 – 2009):

MCP [8]
- ESPOL, später (1970) NEWP
- Einsprachen-/Multiprozessorsystem, Stapelbetrieb

Pilot [12]
- Mesa
- Einsprachen-/Einbenutzer-/Mehrprozesssystem

Ethos [14]
- Oberon-2
- ereignisbasiertes (einfädiges) Einsprachen-/Einprozesssystem

Beispiele sprachbasierter Betriebssysteme

- in chronologischer Reihenfolge (1961 – 2009):

MCP [8]
- ESPOL, später (1970) NEWP
- Einsprachen-/Multiprozessorsystem, Stapelbetrieb

Pilot [12]
- Mesa
- Einsprachen-/Einbenutzer-/Mehrprozesssystem

Ethos [14]
- Oberon-2
- ereignisbasiertes (einfädiges) Einsprachen-/Einprozesssystem

SPIN [1]
- Modula-3
- basiert auf Mach 3.0 (Mikrokern) und OSF/1 Unix (Server)

Beispiele sprachbasierter Betriebssysteme

- in chronologischer Reihenfolge (1961 – 2009):

MCP [8]
- ESPOL, später (1970) NEWP
- Einsprachen-/Multiprozessorsystem, Stapelbetrieb

Pilot [12]
- Mesa
- Einsprachen-/Einbenutzer-/Mehrprozesssystem

Ethos [14]
- Oberon-2
- ereignisbasiertes (einfädiges) Einsprachen-/Einprozesssystem

SPIN [1]
- Modula-3
- basiert auf Mach 3.0 (Mikrokern) und OSF/1 Unix (Server)

JX [5]
- Java
- basiert auf eine eigene, mikrokernähnliche Exekutive[2]

[2] Die Mikrokernarchitektur von JX weicht ab vom sonst üblichen Modell [7].

Beispiele sprachbasierter Betriebssysteme

- in chronologischer Reihenfolge (1961 – 2009):

MCP [8]
- ESPOL, später (1970) NEWP
- Einsprachen-/Multiprozessorsystem, Stapelbetrieb

Pilot [12]
- Mesa
- Einsprachen-/Einbenutzer-/Mehrprozesssystem

Ethos [14]
- Oberon-2
- ereignisbasiertes (einfädiges) Einsprachen-/Einprozesssystem

SPIN [1]
- Modula-3
- basiert auf Mach 3.0 (Mikrokern) und OSF/1 Unix (Server)

JX [5]
- Java
- basiert auf eine eigene, mikrokernähnliche Exekutive[2]

KESO [16]
- Java
- basiert auf OSEK/VDX Echtzeitbetriebssysteme

[2] Die Mikrokernarchitektur von JX weicht ab vom sonst üblichen Modell [7].

Beispiele sprachbasierter Betriebssysteme

- in chronologischer Reihenfolge (1961 – 2009):
  - **MCP [8]**
    - ESPOL, später (1970) NEWP
    - Einsprachen-/Multiprozessorsystem, Stapelbetrieb
  - **Pilot [12]**
    - Mesa
    - Einsprachen-/Einbenutzer-/Mehrprozesssystem
  - **Ethos [14]**
    - Oberon-2
    - ereignisbasiertes (einfädiges) Einsprachen-/Einprozesssystem
  - **SPIN [1]**
    - Modula-3
    - basiert auf Mach 3.0 (Mikrokern) und OSF/1 Unix (Server)
  - **JX [5]**
    - Java
    - basiert auf eine eigene, mikrokernähnliche Exekutive[2]
  - **KESO [16]**
    - Java
    - basiert auf OSEK/VDX Echtzeitbetriebssysteme

- mit Ausnahme von MCP, war keines dieser Systeme ein Durchbruch in kommerzieller Hinsicht
  - auch in akademischer Hinsicht, haben sie sich nicht durchsetzen können

[2] Die Mikrokernarchitektur von JX weicht ab vom sonst üblichen Modell [7].

Gliederung

Kehrseite der Medaille typischerer Systeme

Gegenstand von Kritik an sprachbasierten Betriebssystemen ist das zu Grunde liegende Modell eines abstrakten Prozessors

- vornehmlich vorgegeben durch die typsichere Programmiersprache
  - bspw. JX [5], genauer dessen Mikrokern resp. DomainZero:
    - typische Mikrokernfunktionalität, ohne hardwarebasierten Speicherschutz
  - strukturelle Komplexität geführten Programmtextes (*managed code*)

Kehrseite der Medaille typischerer Systeme

Gegenstand von Kritik an sprachbasierten Betriebssystemen ist das zu Grunde liegende Modell eines abstrakten Prozessors

- vornehmlich vorgegeben durch die typsichere Programmiersprache
  - bspw. JX [5], genauer dessen Mikrokern resp. DomainZero:
    - typische Mikrokernfunktionalität, ohne hardwarebasierten Speicherschutz
  - strukturelle Komplexität geführten Programmtextes (*managed code*)
- implementiert in einer anderen, typunsicheren Programmiersprache
  - meistens C, gelegentlich C++, aber auch Assembliersprache
    - Urladen, Systeminitialisierung, Zustandswechsel der CPU, Einplanung
    - maschinenorientierte (*low-level*) Verwaltung von Schutzdomänen
    - Speicherbereinigung (*garbage collection*), Betriebsüberwachung (*monitoring*)
  - bei JX bedeutet dies 25 Tausend Zeilen C für den Mikrokern [5, S. 134]

Kehrseite der Medaille typischerer Systeme

Gegenstand von Kritik an sprachbasierten Betriebssystemen ist das zu Grunde liegende Modell eines abstrakten Prozessors

- vornehmlich vorgegeben durch die typsichere Programmiersprache
  - bspw. JX [5], genauer dessen Mikrokern resp. DomainZero:
    - typische Mikrokernfunktionalität, ohne hardwarebasierten Speicherschutz
  - strukturelle Komplexität geführten Programmtextes (*managed code*)
- implementiert in einer anderen, typunsicheren Programmiersprache
  - meistens C, gelegentlich C++, aber auch Assembliersprache
    - Urladen, Systeminitialisierung, Zustandswechsel der CPU, Einplanung
    - maschinenorientierte (*low-level*) Verwaltung von Schutzdomänen
    - Speicherbereinigung (*garbage collection*), Betriebsüberwachung (*monitoring*)
  - bei JX bedeutet dies 25 Tausend Zeilen C für den Mikrokern [5, S. 134]

- Grund: Funktionalitäten, die sprachlich nicht anders ausdrückbar sind

Kehrseite der Medaille typischerer Systeme

Gegenstand von Kritik an sprachbasierten Betriebssystemen ist das zu Grunde liegende Modell eines abstrakten Prozessors

- vornehmlich vorgegeben durch die typsichere Programmiersprache
  - bspw. JX [5], genauer dessen Mikrokern resp. DomainZero:
    - typische Mikrokernfunktionalität, ohne hardwarebasierten Speicherschutz
  - strukturelle Komplexität geführten Programmtextes (*managed code*)
- implementiert in einer anderen, typunsicheren Programmiersprache
  - meistens C, gelegentlich C++, aber auch Assembliersprache
    - Urladen, Systeminitialisierung, Zustandswechsel der CPU, Einplanung
    - maschinenorientierte (*low-level*) Verwaltung von Schutzdomänen
    - Speicherbereinigung (*garbage collection*), Betriebsüberwachung (*monitoring*)
  - bei JX bedeutet dies 25 Tausend Zeilen C für den Mikrokern [5, S. 134]

- Grund: Funktionalitäten, die sprachlich nicht anders ausdrückbar sind

Betriebssysteme als (typsicheres) Einsprachensystem zu realisieren, erfordert eine echte Systemprogrammiersprache.

System{implementierungs,programmier}sprache

- Sprachkonzepte speziell zur Implementierung von Betriebssystemen:
  - Ummantelung der Unterbrechungsbehandlung
  - Zugriff auf gestapelten Prozessorstatus (*trap*)
  - Grundlage für Programmfäden/Prozessinkarnationen
  - Ausprägung für ereignis- und prozessbasierte Systeme
  - kontextabhängiger Maschinenzustand einer Koroutine
  - echte Elementaroperationen: TAS, FAA, CAS, …
  - Unterbrechungssteuerung, LL/SC
  - Laden/Spülen des Übersetzungspuffers (TLB)
  - Auslösen eines asynchronen Systemsprungs (AST)
  - Ruhezustand, Bereitschaftsbetrieb
  - Speichersynchronisation
  - Prozessor(kern)signalisierung
  - Repräsentation des Speicherworts des Prozessors
  - Repräsentation des realen Adressraums (Tabelle)

System{implementierungs,programmier}sprache

- Sprachkonzepte speziell zur Implementierung von Betriebssystemen:

Flansch	▪ Ummantelung der Unterbrechungsbehandlung ▪ Zugriff auf gestapelten Prozessorstatus (*trap*)
Koroutine	▪ Grundlage für Programmfäden/Prozessinkarnationen ▪ Ausprägung für ereignis- und prozessbasierte Systeme
Prozessorstatus	▪ kontextabhängiger Maschinenzustand einer Koroutine
Transaktion	▪ echte Elementaroperationen: TAS, FAA, CAS, …
Spezialbefehle	▪ Unterbrechungssteuerung, LL/SC ▪ Laden/Spülen des Übersetzungspuffers (TLB) ▪ Auslösen eines asynchronen Systemsprungs (AST) ▪ Ruhezustand, Bereitschaftsbetrieb ▪ Speichersynchronisation ▪ Prozessor(kern)signalisierung
Maschinenwort	▪ Repräsentation des Speicherworts des Prozessors
Speicherfeld	▪ Repräsentation des realen Adressraums (Tabelle)

System{implementierungs,programmier}sprache

- Sprachkonzepte speziell zur Implementierung von Betriebssystemen:

Flansch	▪ Ummantelung der Unterbrechungsbehandlung ▪ Zugriff auf gestapelten Prozessorstatus (*trap*)
Koroutine	▪ Grundlage für Programmfäden/Prozessinkarnationen ▪ Ausprägung für ereignis- und prozessbasierte Systeme
Prozessorstatus	▪ kontextabhängiger Maschinenzustand einer Koroutine
Transaktion	▪ echte Elementaroperationen: TAS, FAA, CAS, …
Spezialbefehle	▪ Unterbrechungssteuerung, LL/SC ▪ Laden/Spülen des Übersetzungspuffers (TLB) ▪ Auslösen eines asynchronen Systemsprungs (AST) ▪ Ruhezustand, Bereitschaftsbetrieb ▪ Speichersynchronisation ▪ Prozessor(kern)signalisierung
Maschinenwort	▪ Repräsentation des Speicherworts des Prozessors
Speicherfeld	▪ Repräsentation des realen Adressraums (Tabelle)

- alle Prozessormerkmale mit Betriebssystembezug sind abzudecken

System{implementierungs,programmier}sprache

- Sprachkonzepte speziell zur Implementierung von Betriebssystemen:

Flansch	▪ Ummantelung der Unterbrechungsbehandlung ▪ Zugriff auf gestapelten Prozessorstatus (*trap*)
Koroutine	▪ Grundlage für Programmfäden/Prozessinkarnationen ▪ Ausprägung für ereignis- und prozessbasierte Systeme
Prozessorstatus	▪ kontextabhängiger Maschinenzustand einer Koroutine
Transaktion	▪ echte Elementaroperationen: TAS, FAA, CAS, …
Spezialbefehle	▪ Unterbrechungssteuerung, LL/SC ▪ Laden/Spülen des Übersetzungspuffers (TLB) ▪ Auslösen eines asynchronen Systemsprungs (AST) ▪ Ruhezustand, Bereitschaftsbetrieb ▪ Speichersynchronisation ▪ Prozessor(kern)signalisierung
Maschinenwort	▪ Repräsentation des Speicherworts des Prozessors
Speicherfeld	▪ Repräsentation des realen Adressraums (Tabelle)

- alle Prozessormerkmale mit Betriebssystembezug sind abzudecken
- zum Vergleich: hardware-abhängiger Teil von JITTY-OS

Flansch: Pro-/Epilog einer Behandlungsroutine

- Prozessorstatus invariant halten und Systemmodus verlassen

```
1  .macro flange vec
2    pushl %edx ; save volatile register
3    pushl %ecx ; dito
4    pushl %eax ; dito
5    flxh  \vec ; run first-level exception handler #vec
6    popl  %eax ; restore volatile register
7    popl  %ecx ; dito
8    popl  %edx ; dito
9    iret       ; return from trap/interrupt
10 .endm
```

  - 2–4: Prolog $\mapsto$ Sicherung der flüchtigen Prozessorregister *und*
  - 6–9: Epilog $\mapsto$ deren Wiederherstellung und Rücksprung

Flansch: Pro-/Epilog einer Behandlungsroutine

- Prozessorstatus invariant halten und Systemmodus verlassen

```
1  .macro flange vec
2    pushl %edx ; save volatile register
3    pushl %ecx ; dito
4    pushl %eax ; dito
5    flxh  \vec ; run first-level exception handler #vec
6    popl  %eax ; restore volatile register
7    popl  %ecx ; dito
8    popl  %edx ; dito
9    iret       ; return from trap/interrupt
10 .endm
```

  - 2–4: Prolog $\mapsto$ Sicherung der flüchtigen Prozessorregister *und*
  - 6–9: Epilog $\mapsto$ deren Wiederherstellung und Rücksprung

- Mustervorlage für den Text der eigentlichen Behandlungsroutine

```
1  .macro flxh xhn
2    ...        ; code for handler "xhn" comes here
3  .endm
```

Flansch: Definition von Behandlungsroutinen

- synchrone Programmunterbrechung

```
1  trap[128] = handler(trapframe state) {
2      ...
3  }
```

  - Definition für Vektornummer 128 (Systemaufruf)
  - Werteübergabe des gesicherten Prozessorstatus' an `state`

Flansch: Definition von Behandlungsroutinen

- synchrone Programmunterbrechung

```
1  trap[128] = handler(trapframe state) {
2      ...
3  }
```

  - Definition für Vektornummer 128 (Systemaufruf)
  - Werteübergabe des gesicherten Prozessorstatus' an `state`

- asynchrone Programmunterbrechung

```
1  interrupt[42]<level triggered> = handler() {
2      ...
3  }
```

  - Definition für Vektornummer 42
  - Angabe der Signalisierungsart: Unterbrechungen müssen gesperrt bleiben

Flansch: Definition von Behandlungsroutinen

- synchrone Programmunterbrechung

```
1  trap[128] = handler(trapframe state) {
2      ...
3  }
```

  - Definition für Vektornummer 128 (Systemaufruf)
  - Werteübergabe des gesicherten Prozessorstatus' an `state`

- asynchrone Programmunterbrechung

```
1  interrupt[42]<level triggered> = handler() {
2      ...
3  }
```

  - Definition für Vektornummer 42
  - Angabe der Signalisierungsart: Unterbrechungen müssen gesperrt bleiben

- der Kompilierer definiert die Makros `flxh` bei der Kodegenerierung

Flansch: Definition von Behandlungsroutinen

- synchrone Programmunterbrechung

```
1  trap[128] = handler(trapframe state) {
2      ...
3  }
```

  - Definition für Vektornummer 128 (Systemaufruf)
  - Werteübergabe des gesicherten Prozessorstatus' an `state`

- asynchrone Programmunterbrechung

```
1  interrupt[42]<level triggered> = handler() {
2      ...
3  }
```

  - Definition für Vektornummer 42
  - Angabe der Signalisierungsart: Unterbrechungen müssen gesperrt bleiben

- der Kompilierer definiert die Makros `flxh` bei der Kodegenerierung

- Beispiel: `ISR(INT0_vect) { ... }` definiert AVR-Interrupt-Handler

Koroutine: Kontrollflusswechsel

- Koroutinenwechsel für ereignisbasierte Systeme: gemeinsamer Stapel

```
1  .macro resume this, save
2    movl  \this, \save ; keep target instruction pointer
3    movl  $.r\@, \this ; return address of this coroutine
4    jmp   *(\save)     ; switch to next coroutine
5    .p2align 3         ; ensure branch target alignment
6  .r\@:                ; come here when being switched on
7  .endm
```

Koroutine: Kontrollflusswechsel

- Koroutinenwechsel für ereignisbasierte Systeme: gemeinsamer Stapel

```
1 .macro resume this, save
2   movl  \this, \save ; keep target instruction pointer
3   movl  $.r\@, \this ; return address of this coroutine
4   jmp   *(\save)     ; switch to next coroutine
5   .p2align 3         ; ensure branch target alignment
6 .r\@:                ; come here when being switched on
7 .endm
```

- Koroutinenwechsel für prozessbasierte Systeme: individueller Stapel

```
1 .macro resume this, save
2   movl  \this, \save ; keep target stack pointer
3   pushl $.r\@        ; create address of this coroutine
4   movl  %esp, \this  ; and return its stack pointer
5   movl  \save, %esp  ; switch to stack of next coroutine
6   ret                ; and resume its execution
7   .p2align 3         ; ensure branch target alignment
8 .r\@:                ; come here when being switched on
9 .endm
```

Koroutine: Kontrollflusserzeugung I

- Koroutinengabelung für ereignisbasierte Systeme: gemeinsamer Stapel

```
1 .macro fork this
2   movl $.f\@, \this   ; child start address
3   orl  $1, \this      ; indicate parent continuation
4   .p2align 3          ; ensure 16-bit aligned label
5 .f\@:                 ; initial resume of child
6   btrl $1, \this      ; restore address and check it
7 .endm
```

  - `this` ist Start- bzw. Fortsetzungsadresse der neuen Koroutine

Koroutine: Kontrollflusserzeugung I

- Koroutinengabelung für ereignisbasierte Systeme: gemeinsamer Stapel

```
1 .macro fork this
2   movl $.f\@, \this   ; child start address
3   orl  $1, \this      ; indicate parent continuation
4   .p2align 3          ; ensure 16-bit aligned label
5 .f\@:                 ; initial resume of child
6   btrl $1, \this      ; restore address and check it
7 .endm
```

  - `this` ist Start- bzw. Fortsetzungsadresse der neuen Koroutine

- Koroutinengabelung für prozessbasierte Systeme: individueller Stapel

```
1 .macro fork this      ; this must be 16-bit aligned
2   movl $.f\@, (\this) ; setup child start address
3   orl  $1, \this      ; indicate parent continuation
4 .f\@:                 ; initial resume of child
5   btrl $1, \this      ; restore address and check it
6 .endm
```

  - `this` ist Platzhalteradresse für die Start- bzw. Fortsetzungsadresse

Koroutine: Kontrollflusserzeugung II

- Koroutinenerzeugung durch Aufspaltung eines Kontrollflusses

```
1  coroutine<event based> next;
2
3  next = fork {                    /* resp. fork(hook) */
4      next = resume(next); quit;
5  }
6
7  next = resume(next);
```

Koroutine: Kontrollflusserzeugung II

- Koroutinenerzeugung durch Aufspaltung eines Kontrollflusses

```
1  coroutine<event based> next;
2
3  next = fork {                    /* resp. fork(hook) */
4      next = resume(next); quit;
5  }
6
7  next = resume(next);
```

- Beispiel des vom Kompilierer generierten Programmfragments

```
1    fork   %eax        ; event-based coroutine spin-off
2    jc     1f          ; parent is first, child is second
3    resume %eax, %edx  ; child switches back to parent
4    quit               ; and stops upon repeated resume
5  1:                   ; parent comes here after fork
6    resume %eax, %edx  ; and switches to child coroutine
```

Koroutine: Kontrollflusserzeugung II

- Koroutinenerzeugung durch Aufspaltung eines Kontrollflusses

```
1  coroutine<event based> next;
2
3  next = fork {                    /* resp. fork(hook) */
4      next = resume(next); quit;
5  }
6
7  next = resume(next);
```

- Beispiel des vom Kompilierer generierten Programmfragments

```
1    fork   %eax        ; event-based coroutine spin-off
2    jc     1f          ; parent is first, child is second
3    resume %eax, %edx  ; child switches back to parent
4    quit               ; and stops upon repeated resume
5  1:                   ; parent comes here after fork
6    resume %eax, %edx  ; and switches to child coroutine
```

↪ **beachte**: auf dieser Ebene können Koroutinen nicht terminieren

Koroutine: Kontrollflussabbruch

Koroutinen werden nicht als Unterprogramm aufgerufen und besitzen daher auch keinen Aktivierungsblock (*activation record*)

Koroutine: Kontrollflussabbruch

Koroutinen werden nicht als Unterprogramm aufgerufen und besitzen daher auch keinen Aktivierungsblock (*activation record*)

- wohin sie bei Beendigung zurückkehren können sollten, ist unbekannt
  - unabhängig von der Art ihrer Deklaration oder Definition
    i. als Basisblock (wie zuvor auf S. 69 gezeigt) oder
    ii. als wirkliches Unterprogramm (d.h., Funktion oder Prozedur)

Koroutine: Kontrollflussabbruch

Koroutinen werden nicht als Unterprogramm aufgerufen und besitzen daher auch keinen Aktivierungsblock (*activation record*)

- wohin sie bei Beendigung zurückkehren können sollten, ist unbekannt
  - unabhängig von der Art ihrer Deklaration oder Definition
    i. als Basisblock (wie zuvor auf S. 69 gezeigt) oder
    ii. als wirkliches Unterprogramm (d.h., Funktion oder Prozedur)
  - von selbst können sie bestenfalls ihre Laufbereitschaft „aufkündigen“
    - einen Programmabbruch oder -stopp erzwingen und
    - Hilfestellung bei ihrer (weiteren) Zerstörung von außen erwarten

Koroutine: Kontrollflussabbruch

Koroutinen werden nicht als Unterprogramm aufgerufen und besitzen daher auch keinen Aktivierungsblock (*activation record*)

- wohin sie bei Beendigung zurückkehren können sollten, ist unbekannt
  - unabhängig von der Art ihrer Deklaration oder Definition
    - i als Basisblock (wie zuvor auf S. 69 gezeigt) oder
    - ii als wirkliches Unterprogramm (d.h., Funktion oder Prozedur)
  - von selbst können sie bestenfalls ihre Laufbereitschaft „aufkündigen“
    - einen Programmabbruch oder -stopp erzwingen und
    - Hilfestellung bei ihrer (weiteren) Zerstörung von außen erwarten
  - denn unter ihnen ist nur noch die CPU: **Koroutine $\neq$ Faden** **!!!**

Koroutine: Kontrollflussabbruch

Koroutinen werden nicht als Unterprogramm aufgerufen und besitzen daher auch keinen Aktivierungsblock (*activation record*)

- wohin sie bei Beendigung zurückkehren können sollten, ist unbekannt
  - unabhängig von der Art ihrer Deklaration oder Definition
    i. als Basisblock (wie zuvor auf S. 69 gezeigt) oder
    ii. als wirkliches Unterprogramm (d.h., Funktion oder Prozedur)
  - von selbst können sie bestenfalls ihre Laufbereitschaft „aufkündigen“
    - einen Programmabbruch oder -stopp erzwingen und
    - Hilfestellung bei ihrer (weiteren) Zerstörung von außen erwarten
  - denn unter ihnen ist nur noch die CPU: **Koroutine $\neq$ Faden** **!!!**

- Aufkündigung der Laufbereitschaft einer Koroutine:

```
1  .macro quit
2  .q\@:
3    hlt      ; don't know how to proceed...
4    jmp .q\@ ; idle and wait to be assisted
5  .endm
```

Koroutine: Plausibilisierung

as -alm, IA-32

```
                        expanded:
                          fork   %eax
0000 B8000000           > movl $.f0,%eax
     00
0005 83C801             > orl $1,%eax
                        > .p2align 3
                        > .f0:
0008 0FBAF001           > btrl $1,%eax
000c 720D                 jc     1f
                          resume %eax, %edx
000e 89C2               > movl %eax,%edx
0010 B8000000           > movl $.r1,%eax
     00
0015 67FF22             > jmp *(%edx)
                        > .p2align 3
                        > .r1:
                          quit
                        > .q2:
0018 F4                 > hlt
0019 EBFD               > jmp .q2
                        1:
                          resume %eax, %edx
001b 89C2               > movl %eax,%edx
001d B8000000           > movl $.r3,%eax
     00
0022 67FF22             > jmp *(%edx)
0025 000000             > .p2align 3
                        > .r3:
0028 C3                   ret
```

Prozessorstatus: Sicherung/Wiederherstellung

Koroutinenwechsel sind kontextabhängig hinsichtlich des aktuellen Maschinenzustands des Prozessors

Prozessorstatus: Sicherung/Wiederherstellung

Koroutinenwechsel sind kontextabhängig hinsichtlich des aktuellen Maschinenzustands des Prozessors

- nur aktive Prozessorregister brauchen jedoch beachtet zu werden
  - Register, die im gesamten Ablaufpfad bisher ungesichert geblieben sind
    - von der „Wurzel“ (z.B. `flxh`, S. 59) ausgehend bis zum `resume` (S. 65)
  - schlimmstenfalls alle im Programmiermodell der CPU definierten Register
- der nötige Sicherungspuffer ist von variabler aber maximaler Größe

Prozessorstatus: Sicherung/Wiederherstellung

Koroutinenwechsel sind kontextabhängig hinsichtlich des aktuellen Maschinenzustands des Prozessors

- nur aktive Prozessorregister brauchen jedoch beachtet zu werden
  - Register, die im gesamten Ablaufpfad bisher ungesichert geblieben sind
    - von der „Wurzel“ (z.B. `flxh`, S. 59) ausgehend bis zum `resume` (S. 65)
  - schlimmstenfalls alle im Programmiermodell der CPU definierten Register
- der nötige Sicherungspuffer ist von variabler aber maximaler Größe

Optionen zur Verwaltung des Maschinenzustands:

- den Stapelspeicher implizit und dynamisch nutzende Operationen:
  - `push` — auf den Stapel drauflegen & Stapelzeigerwert zurückliefern
  - `pull` — vom Stapel runternehmen
- Operationen, die auf ein Behältnis (*bin*) statischer Größe arbeiten:
  - `dump(bin)` — in den Sicherungspuffer abladen
  - `pick(bin)` — aus dem Sicherungspuffer aufsammeln

Fäserchenwechsel

```
1  class fibril {
2      coroutine<event based> label;
3      bin state;
4  public:
5      static fibril& being(); /* return current fibril */
6      void           apply(); /* define current fibril */
7
8      void board() {          /* switch to this fibril */
9          fibril& self = being();
10         feature("nonpreemptive") {
11             apply();        /* unseal this fibril */
12             assembly {      /* switch processor state */
13                 dump(self.state); /* releasing one */
14                 self.label = resume(label);
15                 pick(state);      /* continued one */
16             }
17         }
18     }
19 };
```

- `feature`: logisch kritischer Abschnitt
- `assembly`: physischen Befehlsverbund bilden

Faserwechsel

```
1  class fibre {
2      coroutine<process based> batch;
3  public:
4      static fibre& being(); /* return current fibre */
5      void          apply(); /* define current fibre */
6
7      void board() {         /* switch to this fibre */
8          register fibre& self = being();
9          feature("nonpreemptive") {
10             apply();       /* unseal this fibre */
11             assembly {     /* switch processor state */
12                 push;      /* releasing one */
13                 self.batch = resume(batch);
14                 pull;      /* continued one */
15             }
16         }
17     }
18 };
```

`register`

- Zuweisung in Zeile 13 ist kritisch
- `self` muss eine Registervariable sein

Diskussion

- es lohnt ein Blick auf Betriebssystemtechnik vom SS 2012 [13]
  - dort wurden die Sprachkonzepte als funktionale Abstraktionen realisiert
    - insb. die hier definierten Koroutinen und darauf aufbauend Programmfäden
    - diese „minimale Teilmenge von Systemfunktionen“[3] ist operationsfähig und
    - wiederverwendbar für problemspezifische, „minimale Systemerweiterungen“[3]
  - mangels Spracheigenschaften entstand ein Zweisprachensystem: C/ASM

[3] Grundprinzipien einer Programmfamilie [11].

Diskussion

- es lohnt ein Blick auf Betriebssystemtechnik vom SS 2012 [13]
  - dort wurden die Sprachkonzepte als funktionale Abstraktionen realisiert
    - insb. die hier definierten Koroutinen und darauf aufbauend Programmfäden
    - diese „minimale Teilmenge von Systemfunktionen“[3] ist operationsfähig und
    - wiederverwendbar für problemspezifische, „minimale Systemerweiterungen“[3]
  - mangels Spracheigenschaften entstand ein Zweisprachensystem: C/ASM
- alle Sprachkonstrukte bilden ab auf elementare Machinenbefehle
  - vergleichbar mit Konzepten einer „Anwendungsprogrammiersprache“
    - Index- oder Typüberprüfungen, dynamische Typisierung
    - parametrischer Polymorphismus (z.B. polymorphe Methoden)
  - damit lassen sich abstrakte Prozessoren typsicherer Sprachen realisieren

[3] Grundprinzipien einer Programmfamilie [11].

Diskussion

- es lohnt ein Blick auf Betriebssystemtechnik vom SS 2012 [13]
  - dort wurden die Sprachkonzepte als funktionale Abstraktionen realisiert
    - insb. die hier definierten Koroutinen und darauf aufbauend Programmfäden
    - diese „minimale Teilmenge von Systemfunktionen“[3] ist operationsfähig und
    - wiederverwendbar für problemspezifische, „minimale Systemerweiterungen“[3]
  - mangels Spracheigenschaften entstand ein Zweisprachensystem: C/ASM
- alle Sprachkonstrukte bilden ab auf elementare Machinenbefehle
  - vergleichbar mit Konzepten einer „Anwendungsprogrammiersprache“
    - Index- oder Typüberprüfungen, dynamische Typisierung
    - parametrischer Polymorphismus (z.B. polymorphe Methoden)
  - damit lassen sich abstrakte Prozessoren typsicherer Sprachen realisieren
- ausschließlich auf Einsprachensysteme zu setzen, ist aber unrealistisch
  - für Universalrechner sind Mehrsprachensysteme Normalität
    - Problem- und Lösungsdomänen bilden auch verschiedene Sprachdomänen
  - Einsprachensysteme sind domänenspezifisch – Betriebssysteme ebenso
    - beides zu vereinen, ist naheliegend – es braucht aber die passende Sprache

[3] Grundprinzipien einer Programmfamilie [11].

Gliederung

Resümee

- Sicherheit in Rechensystemen braucht Immunität und Isolation
  - dabei wird Immunität insbesondere auch durch Isolation erreicht
- Typsicherheit als Option, um Adressraumausbrüchen vorzubeugen
  - Grundlage ist eine statisch oder dynamisch typisierte Programmiersprache
  - sprachbasierten Betriebssystemen mangelt es an Sprachunterstützung
- Systemprogrammiersprachen verbergen keine Prozessormerkmale
  - Betriebssysteme können damit als Einsprachensysteme realisiert werden
  - Typsicherheit steht nicht im Widerspruch dazu, ist sinnvolle Ergänzung
- Einsprachensysteme verlagern mehr Verantwortung in Kompilierer
  - nehmen aber Betriebssystemen damit wenig Verantwortung ab
  - sie setzen nicht zwingend auf geführten Programmtext (*managed code*)
  - ihre Bestimmung ergibt sich durch Betriebssysteme – und nichts anderes
  - der „Schuster bleibt bei seinem Leisten“: Betriebssystem & Kompilierer
- sprachbasierte Betriebssysteme heute (2026) sind zu „maschinenfern“

Resümee

- Sicherheit in Rechensystemen braucht Immunität und Isolation
  - dabei wird Immunität insbesondere auch durch Isolation erreicht
- Typsicherheit als Option, um Adressraumausbrüchen vorzubeugen
  - Grundlage ist eine statisch oder dynamisch typisierte Programmiersprache
  - sprachbasierten Betriebssystemen mangelt es an Sprachunterstützung
- Systemprogrammiersprachen verbergen keine Prozessormerkmale
  - Betriebssysteme können damit als Einsprachensysteme realisiert werden
  - Typsicherheit steht nicht im Widerspruch dazu, ist sinnvolle Ergänzung
- Einsprachensysteme verlagern mehr Verantwortung in Kompilierer
  - nehmen aber Betriebssystemen damit wenig Verantwortung ab
  - sie setzen nicht zwingend auf geführten Programmtext (*managed code*)
  - ihre Bestimmung ergibt sich durch Betriebssysteme – und nichts anderes
  - der „Schuster bleibt bei seinem Leisten“: Betriebssystem & Kompilierer
- sprachbasierte Betriebssysteme heute (2026) sind zu „maschinenfern“

Resümee

- Sicherheit in Rechensystemen braucht Immunität und Isolation
  - dabei wird Immunität insbesondere auch durch Isolation erreicht
- Typsicherheit als Option, um Adressraumausbrüchen vorzubeugen
  - Grundlage ist eine statisch oder dynamisch typisierte Programmiersprache
  - sprachbasierten Betriebssystemen mangelt es an Sprachunterstützung
- Systemprogrammiersprachen verbergen keine Prozessormerkmale
  - Betriebssysteme können damit als Einsprachensysteme realisiert werden
  - Typsicherheit steht nicht im Widerspruch dazu, ist sinnvolle Ergänzung
- Einsprachensysteme verlagern mehr Verantwortung in Kompilierer
  - nehmen aber Betriebssystemen damit wenig Verantwortung ab
  - sie setzen nicht zwingend auf geführten Programmtext (*managed code*)
  - ihre Bestimmung ergibt sich durch Betriebssysteme – und nichts anderes
  - der „Schuster bleibt bei seinem Leisten“: Betriebssystem & Kompilierer
- sprachbasierte Betriebssysteme heute (2026) sind zu „maschinenfern“

Resümee

- Sicherheit in Rechensystemen braucht Immunität und Isolation
  - dabei wird Immunität insbesondere auch durch Isolation erreicht
- Typsicherheit als Option, um Adressraumausbrüchen vorzubeugen
  - Grundlage ist eine statisch oder dynamisch typisierte Programmiersprache
  - sprachbasierten Betriebssystemen mangelt es an Sprachunterstützung
- Systemprogrammiersprachen verbergen keine Prozessormerkmale
  - Betriebssysteme können damit als Einsprachensysteme realisiert werden
  - Typsicherheit steht nicht im Widerspruch dazu, ist sinnvolle Ergänzung
- Einsprachensysteme verlagern mehr Verantwortung in Kompilierer
  - nehmen aber Betriebssystemen damit wenig Verantwortung ab
  - sie setzen nicht zwingend auf geführten Programmtext (*managed code*)
  - ihre Bestimmung ergibt sich durch Betriebssysteme – und nichts anderes
  - der „Schuster bleibt bei seinem Leisten“: Betriebssystem & Kompilierer
- sprachbasierte Betriebssysteme heute (2026) sind zu „maschinenfern“

Resümee

- Sicherheit in Rechensystemen braucht Immunität und Isolation
  - dabei wird Immunität insbesondere auch durch Isolation erreicht
- Typsicherheit als Option, um Adressraumausbrüchen vorzubeugen
  - Grundlage ist eine statisch oder dynamisch typisierte Programmiersprache
  - sprachbasierten Betriebssystemen mangelt es an Sprachunterstützung
- Systemprogrammiersprachen verbergen keine Prozessormerkmale
  - Betriebssysteme können damit als Einsprachensysteme realisiert werden
  - Typsicherheit steht nicht im Widerspruch dazu, ist sinnvolle Ergänzung
- Einsprachensysteme verlagern mehr Verantwortung in Kompilierer
  - nehmen aber Betriebssystemen damit wenig Verantwortung ab
  - sie setzen nicht zwingend auf geführten Programmtext (*managed code*)
  - ihre Bestimmung ergibt sich durch Betriebssysteme – und nichts anderes
  - der „Schuster bleibt bei seinem Leisten“: Betriebssystem & Kompilierer
- sprachbasierte Betriebssysteme heute (2026) sind zu „maschinenfern“

Resümee

- Sicherheit in Rechensystemen braucht Immunität und Isolation
  - dabei wird Immunität insbesondere auch durch Isolation erreicht
- Typsicherheit als Option, um Adressraumausbrüchen vorzubeugen
  - Grundlage ist eine statisch oder dynamisch typisierte Programmiersprache
  - sprachbasierten Betriebssystemen mangelt es an Sprachunterstützung
- Systemprogrammiersprachen verbergen keine Prozessormerkmale
  - Betriebssysteme können damit als Einsprachensysteme realisiert werden
  - Typsicherheit steht nicht im Widerspruch dazu, ist sinnvolle Ergänzung
- Einsprachensysteme verlagern mehr Verantwortung in Kompilierer
  - nehmen aber Betriebssystemen damit wenig Verantwortung ab
  - sie setzen nicht zwingend auf geführten Programmtext (*managed code*)
  - ihre Bestimmung ergibt sich durch Betriebssysteme – und nichts anderes
  - der „Schuster bleibt bei seinem Leisten“: Betriebssystem & Kompilierer
- sprachbasierte Betriebssysteme heute (2026) sind zu „maschinenfern“

Literaturverzeichnis (1)

[1] Bershad, B. N. ; Savage, S. ; Pardyak, P. ; Sirer, E. G. ; Fiuczynski, M. E. ; Becker, D. ; Chambers, C. ; Eggers, S. :
Extensibility, Safety and Performance in the SPIN Operating System.
In: [6], S. 267–284

[2] Chase, J. S. ; Levy, H. M. ; Freeley, M. J. ; Lazowska, E. D.:
Sharing and Protection in a Single-Address-Space Operating System.
In: *ACM Transactions on Computer Systems* 12 (1994), Nov., Nr. 4, S. 271–307

[3] Chou, A. ; Yang, J. ; Chelf, B. ; Hallem, S. ; Engler, D. :
An Empirical Study of Operating System Errors.
In: Marzullo, K. (Hrsg.) ; Satyanarayanan, M. (Hrsg.): *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP 2001)*, ACM, 2001. –
ISBN 1–58113–389–8, S. 73–88

Literaturverzeichnis (2)

[4] Druschel, P. ; Peterson, L. L. ; Hutchinson, N. C.:
Beyond Micro-Kernel Design: Decoupling Modularity and Protection in Lipto.
In: *Proceedings of the 12th International Conference on Distributed Computing Systems (ICDCS 1992)*, IEEE Computer Society, 1992. –
ISBN 0–8186–2865–0, S. 512–520

[5] Golm, M. :
***The Structure of a Type-Safe Operating System*,**
Friedrich-Alexander-Universität Erlangen-Nürnberg, Diss., Dez. 2002

[6] Jones, M. B. (Hrsg.):
Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95).
ACM Press, 1995 . –
ISBN 0–89791–715–4

[7] Liedtke, J. :
On μ-Kernel Construction.
In: [6], S. 237–250

Literaturverzeichnis (3)

[8] Lonergan, W. ; King, P. :
Design of the B 5000 System.
In: *DATAMATION Magazine* 7 (1961), Mai, Nr. 5, S. 28–32

[9] Parnas, D. L.:
Information Distribution Aspects of Design Methodology.
In: Freiman, C. V. (Hrsg.) ; Griffith, J. E. (Hrsg.) ; Rosenfeld, J. L. (Hrsg.): *Information Processing 71, Proceedings of the IFIP Congress 71* Bd. 1 (Foundations and Systems), North-Holland Publishing Company, 1971. – ISBN 0–7204–2063–6, S. 339–344

[10] Parnas, D. L.:
On the Criteria to be used in Decomposing Systems into Modules.
In: *Communications of the ACM* 15 (1972), Dez., Nr. 12, S. 1053–1058

[11] Parnas, D. L.:
On the Design and Development of Program Families.
In: *IEEE Transactions on Software Engineering* SE-2 (1976), März, Nr. 1, S. 1–9

Literaturverzeichnis (4)

[12] Redell, D. A. ; Dalal, Y. K. ; Horsley, T. R. ; Lauer, H. C. ; Lynch, W. C. ; McJones, P. R. ; Murray, H. G. ; Purcell, S. C.:
Pilot: An Operating System for a Personal Computer.
In: *Communications of the ACM* 23 (1980), Febr., Nr. 2, S. 81–92

[13] Schröder-Preikschat, W. :
Betriebssystemtechnik.
`http://www4.informatik.uni-erlangen.de/Lehre/SS12/V_BST`, 2012

[14] Szyperski, C. A.:
***Insight ETHOS: On Object-Orientation in Operating Systems*,**
Eidgenössische Technische Hochschule Zürich, Diss., 1992

[15] Tanenbaum, A. S.:
Multilevel Machines.
In: *Structured Computer Organization.*
Prentice-Hall, Inc., 1979. –
ISBN 0–130–95990–1, Kapitel 7, S. 344–386

Literaturverzeichnis (5)

[16] Wawersich, C. W. A.:
***KESO: Konstruktiver Speicherschutz für Eingebettete Systeme*,**
Friedrich-Alexander-Universität Erlangen-Nürnberg, Diss., März 2009