

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil B – VI.1 Betriebssystemkonzepte: Prozesse

Wolfgang Schröder-Preikschat

9. Juni 2022



Agenda

Einführung
Begriff

Grundlagen
Virtualität
Betriebsmittel
Programme

Verwaltung
Planung
Synchronisation
Implementierungsaspekte

Zusammenfassung



Gliederung

Einführung
Begriff

Grundlagen
Virtualität
Betriebsmittel
Programme

Verwaltung
Planung
Synchronisation
Implementierungsaspekte

Zusammenfassung



- **Prozess** als das zentrale Konzept von Betriebssystemen kennenlernen
 - wobei von der **Verkörperung** (Inkarnation) dieses Konzepts getrennt wird
 - ob also ein Prozess z.B. als *Thread* oder *Task* implementiert ist bzw.
 - ob er allein oder mit anderen zusammen im selben Adressraum verweilt und
 - ob sein Adressraum eine physisch durchzusetzende Schutzdomäne darstellt
 - auf das Wesentliche konzentrieren: Prozess als „*program in execution*“ [7]
- auf (die Art der) **Betriebsmittel** eingehen, die ein Prozess benötigt
 - wiederverwendbare und konsumierbare Betriebsmittel unterscheiden
 - implizite und explizite Koordinierung von Prozessen verdeutlichen, d.h.,
 - geplante und programmierte **Synchronisation** von Prozessen erklären
 - mehr- und einseitige Synchronisation beispielhaft zeigen: *bounded buffer*
- **Prozessausprägungen** und zugehörige Systemfunktionen beleuchten
 - typische (logische) Verarbeitungszustände von Prozessen einführen
 - Einplanung (*scheduling*) und Einlastung (*dispatching*) differenzieren
 - Verortung von Prozessen auf Benutzer- und Systemebene skizzieren
 - Prozesskontrollblock, -zeiger und -identifikation begrifflich erfassen



Rezipiert als Informatikbegriff

Definition (Prozess \equiv Programmablauf)

Ein Programm in Ausführung durch einen Prozessor.

- das Programm spezifiziert eine Folge von Aktionen des Prozessors
 - die Art einer Aktion hängt von der betrachteten Abstraktionsebene ab
 - Ebene₅ \mapsto Programmanweisung ≥ 1 Assembliernemoniks
 - Ebene₄ \mapsto Assembliernemonik ≥ 1 Maschinenbefehle
 - Ebene₃ \mapsto Maschinenbefehl ≥ 1 Mikroprogramminstruktionen
 - Ebene₂ \mapsto Mikroprogramminstruktion
 - die Aktion eines Prozessors ist damit **nicht zwingend unteilbar** (atomar)
 - sowohl für den abstrakten (virtuellen) als auch den realen Prozessor
- das Programm ist statisch (passiv), ein Prozess ist dynamisch (aktiv)

Hinweis (Prozess \neq Prozessinkarnation, Prozessexemplar)

*Eine Prozessinkarnation ist **Exemplar** eines Programms als **Bautyp** für einen Prozess, wie ein Objekt Exemplar eines Datentyps ist.*



Gliederung

Einführung
Begriff

Grundlagen
Virtualität
Betriebsmittel
Programme

Verwaltung
Planung
Synchronisation
Implementierungsaspekte

Zusammenfassung

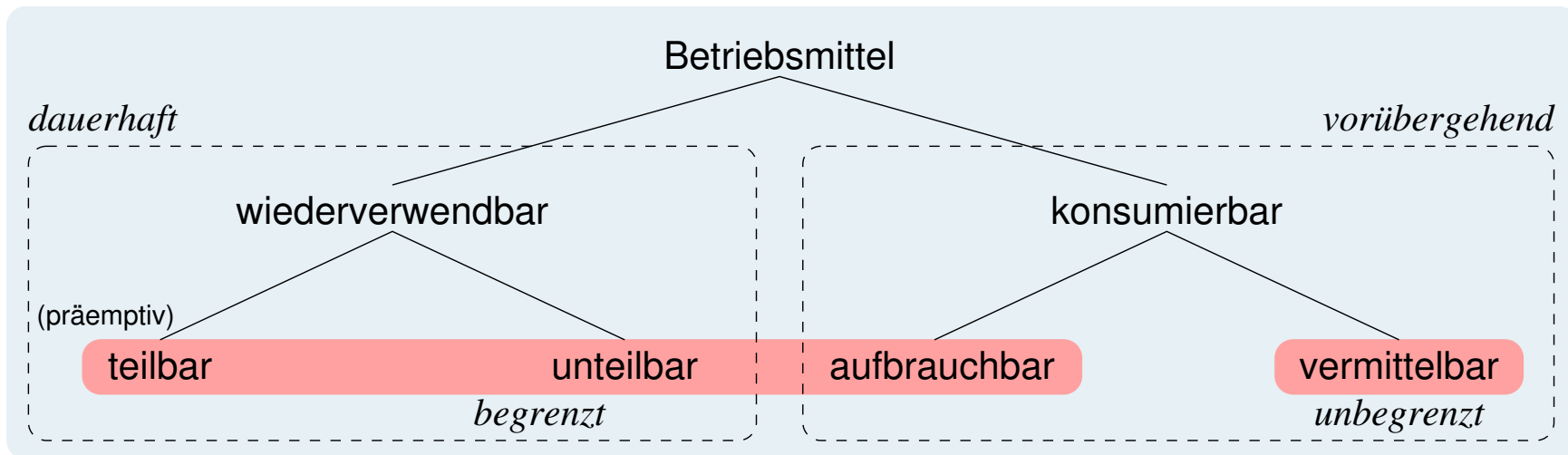


- Prozess bezeichnet sowohl den Ablauf eines Programms als auch die **Abstraktion** von einem solchen Programmablauf
 - der physisch durch seinen gegenwärtigen **Laufzeitkontext** definiert ist, insbesondere manifestiert im **Programmiermodell** des Prozessors
- diese Abstraktion ermöglicht es, simultan mehrere Programmabläufe im **Multiplexverfahren** auf einem Prozessor stattfinden zu lassen
 - dabei sind die Abläufe Teil eines einzelnen oder mehrerer Programme
 - Mehrfädigkeit (*multithreading*)/Mehrprogrammbetrieb (*multiprogramming*)
 - für den Ablauf lastet das Betriebssystem einen Prozess ein (*dispatching*)
 - Laufzeitkontext umschalten aktiviert dann einen anderen Programmablauf
 - hierzu plant das Betriebssystem Prozesse entsprechend ein (*scheduling*)
- geläufig ist das **Zeiteilverfahren** (*time-sharing*; CTSS [5]), von dem es verschiedene Ausführungen gibt
 - je nachdem, wie viel und wie oft den Prozessen Rechenzeit innerhalb einer bestimmten Zeitspanne zugeordnet werden kann, soll oder muss
 - pro **Zeitschlitz** laufen im Prozess meist mehrere Aktionen (S. 12) ab



- Prozesse sind das Mittel zum Zweck, (pseudo/quasi) **gleichzeitige Programmabläufe** stattfinden zu lassen \leadsto **Parallelität**
 - multiprogramming* ■ mehrere Programme
 - multitasking* ■ mehrere Aufgaben mehrerer Programme
 - multithreading* ■ mehrere Fäden eines oder mehrerer Programme
 - pseudo/quasi gleichzeitig, wenn weniger reale Prozessoren zur Verfügung stehen als zu einem Zeitpunkt Programmabläufe möglich sind
 - ein Programmablauf ist möglich, wenn:
 - i er dem Betriebssystem explizit gemacht worden ist und
 - ii alle von ihm benötigten **Betriebsmittel** (real/virtuell) verfügbar sind
 - ist eine gemeinsame Benutzung (*sharing*) oder logische Abhängigkeit von Betriebsmitteln gegeben, wird **Synchronisation** erforderlich
 - die Fäden/Aufgaben/Programme teilen sich dieselben (realen) Daten
 - formuliert in dem Programm daselbst, das damit als **nichtsequentielles Programm** in Erscheinung tritt
- ↪ die Maßnahmen dazu gestalten sich recht unterschiedlich, je nach Art des Betriebsmittels und Zweck des Prozesszugriffs





- dauerhafte¹ Betriebsmittel sind von Prozessen **wiederverwendbar**
 - sie werden angefordert, belegt, benutzt und danach wieder freigegeben
 - in Benutzung befindliche Betriebsmittel sind ggf. **zeitlich teilbar**
 - je nachdem, ob der **Betriebsmitteltyp** eine gleichzeitige Benutzung zulässt
 - falls unteilbar, sind sie einem Prozess **zeitweise exklusiv** zugeordnet
- vorübergehende Betriebsmittel sind von Prozesse **konsumierbar**
 - sie werden produziert, zugeteilt/vermittelt, benutzt und aufgebraucht
- wiederverwendbare (gegenständliche) und aufbrauchbare (messbare) Betriebsmittel stehen nur begrenzt zur Verfügung

¹auch: persistente



Eigentümlichkeiten von Betriebsmitteln

- vom Betriebssystem zu verwaltende Betriebsmittel:

wiederverwendbar (Hardware)

- Prozessor ■ CPU, FPU, GPU; MMU
- Speicher ■ RAM, *scratch pad*, *flash*
- Peripherie ■ Ein-/Ausgabe, *storage*

konsumierbar

- Signal ■ IRQ, NMI, *trap*
- Größe ■ Zeit, Energie

- von jedem Programm verwaltete Softwarebetriebsmittel:

wiederverwendbar

- Text ■ kritischer Abschnitt
- Daten ■ Variable, Platzhalter

konsumierbar

- Signal ■ Meldung
- Nachricht ■ Datenstrom

- wiederverwendbare Betriebsmittel sind Behälter für vermittelbare
 - zur Verarbeitung müssen letztere in Variablen/Platzhaltern vorliegen
- Verfügbarkeit ersterer beschränkt Erzeugung/Verbrauch letzterer

- gleichzeitige Zugriffe auf unteilbare und Übernahme vermittelbarer Betriebsmittel erfordern die **Synchronisation** involvierter Prozesse



Gerichteter Ablauf eines Geschehens [25]

Betriebssysteme bringen Programme zur Ausführung, in dem dazu Prozesse erzeugt, bereitgestellt und begleitet werden

- im Informatikkontext ist ein Prozess ohne Programm nicht möglich
 - die als Programm kodierte Berechnungsvorschrift definiert den Prozess
 - das Programm legt damit den Prozess fest, gibt ihn vor
 - gegebenenfalls bewirkt, steuert, terminiert es gar andere Prozesse
 - wenn das Betriebssystem die dazu nötigen Befehle anbietet!
- ein Programm beschreibt (auch) die Art des Ablaufs eines Prozesses
 - sequentiell** ■ eine Folge von zeitlich nicht überlappenden Aktionen
 - verläuft deterministisch, das Ergebnis ist determiniert
 - parallel** ■ nicht sequentiell
- in beiden Arten besteht ein Programmablauf aus **Aktionen**

Beachte: Programmablauf und Abstraktionsebene (vgl. S. 5)

Ein und derselbe Programmablauf kann auf einer Abstraktionsebene sequentiell, auf einer anderen parallel sein. [21]



Definition (Programm)

Die für eine Maschine konkretisierte Form eines Algorithmus.

- virtuelle Maschine C
 - nach der Editierung und
 - vor der Kompilierung
- virtuelle Maschine ASM (x86)
 - nach der Kompilierung² und
 - vor der Assemblierung

```
1 #include <stdint.h>
2
3 void inc64(int64_t *i) {
4     (*i)++;
5 }
```

```
11 inc64:
12     movl 4(%esp), %eax
13     addl $1, (%eax)
14     adcl $0, 4(%eax)
15     ret
```

- eine Aktion (Zeile 4)
- drei Aktionen (Zeilen 12–14)

Definition (Aktion)

Die Ausführung einer Anweisung einer (virtuellen/realen) Maschine.

²Übersetzung des Unterprogramms (Z. 1–5) mit `-S`.



Nichtsequentielles Maschinenprogramm

Definition

Ein Programm P , das Aktionen spezifiziert, die parallele Abläufe in P selbst zulassen.

- ein Ausschnitt von P am Beispiel von *POSIX Threads* [17]:

```
1 pthread_t tid;
2
3 if (!pthread_create(&tid, NULL, thread, NULL)) {
4     /* ... */
5     pthread_join(tid, NULL);
6 }
```

- der in P selbst zugelassene parallele Ablauf:

```
7 void *thread(void *null) {
8     /* ... */
9     pthread_exit(NULL);
10 }
```



Nichtsequentielles Maschinenprogramm

- Aktionen für Parallelität, aber weiterhin **sequentielle Abläufe in P**

```
1 pid_t pid;
2
3 if (!(pid = fork())) {
4     /* ... */
5     exit(0);
6 }
7
8 wait(NULL);
```

- `fork` dupliziert den Adressraum A von P , erzeugt A' als Duplikat von A
- in A als Ursprungsadressraum entsteht damit jedoch kein paralleler Ablauf
- unabhängig vom Parallelitätsgrad in P , setzt `fork` diesen für A' immer auf 1

- Programm P spezifiziert zwar Aktionen, die Parallelität zulassen, diese kommt jedoch nur allein durch `fork` nicht in P selbst zur Wirkung

- die Aktionen bedingen parallele Abläufe innerhalb des Betriebssystems

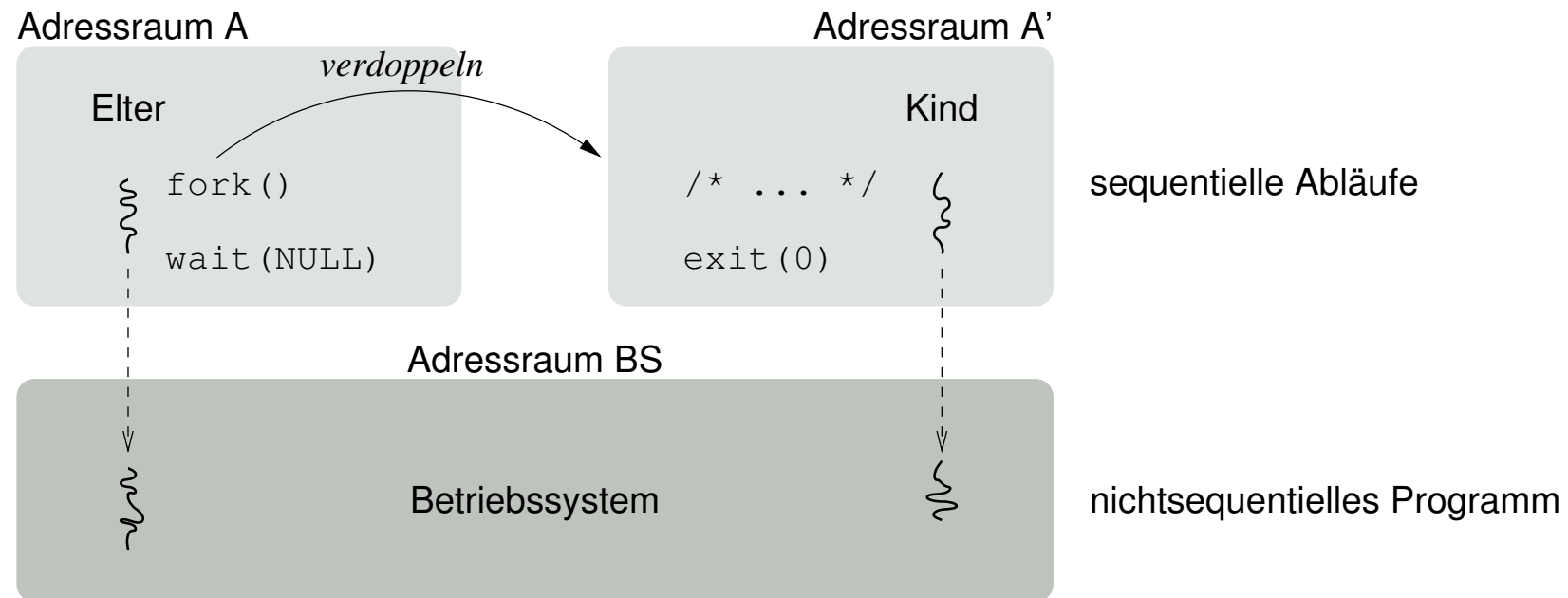
- Simultanbetrieb (*multiprocessing*) sequentieller Abläufe benötigt das Betriebssystem in Form eines nichtsequentiellen Programms
- hilfreiches Merkmal: Mehrfädigkeit (*multithreading*) im Betriebssystem

- ein Betriebssystem ist **Inbegriff** des nichtsequentiellen Programms³

³Ausnahmen (strikt kooperative Systeme) bestätigen die Regel.



Simultanverarbeitung sequentieller Abläufe



- dabei ist die **Parallelität** in dem System unterschiedlich ausgeprägt:
 - pseudo ■ durch *Multiplexen* eines realen/virtuellen Prozessors (vgl. S. 7)⁴
 - echte ■ durch *Vervielfachung* eines realen Prozessors
- Folge der Operationen sind **parallele Prozesse** im Betriebssystem
 - auch als **nichtsequentielle Prozesse** bezeichnet
 - nämlich Prozesse, deren Aktionen sich zeitlich überlappen können

⁴(gr.) *pseúdein* belügen, täuschen



Gliederung

Einführung
Begriff

Grundlagen
Virtualität
Betriebsmittel
Programme

Verwaltung
Planung
Synchronisation
Implementierungsaspekte

Zusammenfassung



Prozesse werden gestartet, unterbrochen, fortgesetzt und beendet

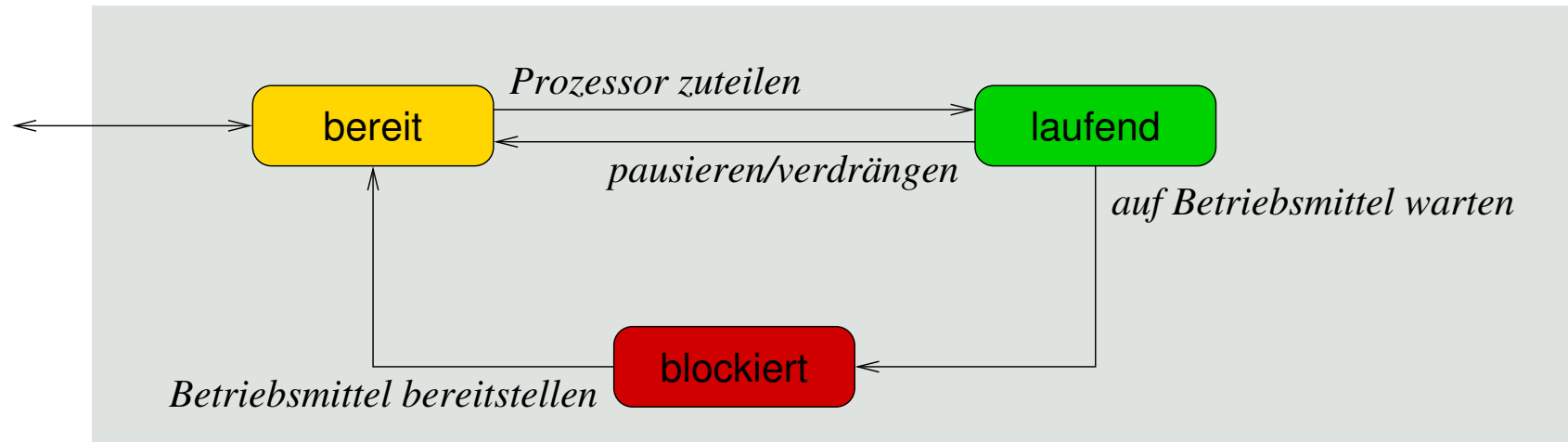
- zentrale Funktion dabei die **Prozesseinplanung** (*process scheduling*), die allgemein zwei grundsätzliche Fragestellungen zu lösen hat:
 - i Zu welchem (logischen/physikalischen) **Zeitpunkt** sollen Prozesse in den Kreislauf der Programmverarbeitung eingespeist werden?
 - ii In welcher **Reihenfolge** sollen die eingespeisten Prozesse stattfinden?
- Zweck aller hierzu erforderlichen Verfahren ist es, die **Zuteilung von Betriebsmitteln** an konkurrierende Prozesse zu kontrollieren

Einplanungsalgorithmus (*scheduling algorithm*)

Beschreibt und formuliert die **Strategie**, nach der ein von einem Rechensystem zu leistender Ablaufplan zur Erfüllung der jeweiligen **Anwendungsanforderungen** entsprechend der gewählten **Rechnerbetriebsart** aufzustellen, abzuarbeiten und fortzuschreiben ist.



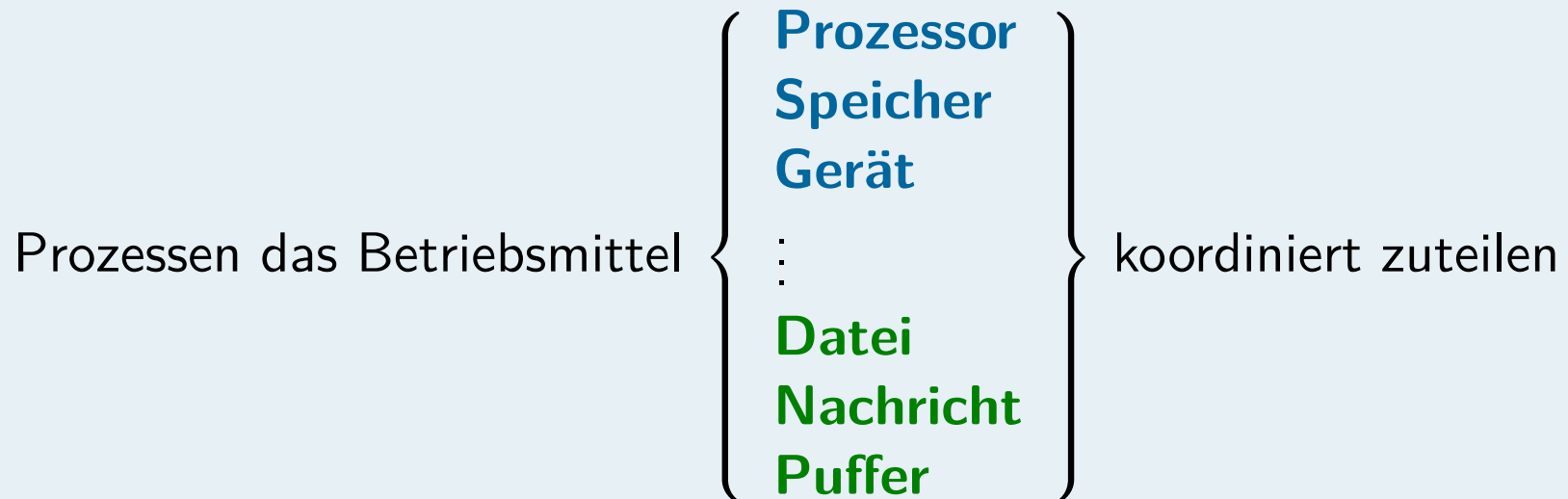
- ein Prozess kann angeordnet werden und stattfinden, wenn alle dazu benötigten Betriebsmittel verfügbar sind



- die Zustandsübergänge bewirkt der **Planer** (*scheduler*), sie definieren verschiedene Phasen der Prozessverarbeitung
 - scheduling* ■ beim Übergang in die Zustände „bereit“ oder „blockiert“
 - dispatching* ■ beim Übergang in den Zustand „laufend“
- je **Rechenkern** kann es zu einem Zeitpunkt stets nur einen laufenden, jedoch mehr als einen blockierten oder bereiten Prozess geben



Reihenfolgebestimmung



- Betriebsmittel, die in **Hardware** oder **Software** ausgeprägt vorliegen
- fehlen Prozessen nur noch ein Prozessor als Betriebsmittel, definiert die **Bereitliste** (*ready list*) den **Ablaufplan** zur Prozessorzuteilung
 - Listenelemente sind **Prozesskontrollblöcke** (siehe S. 29), geordnet⁵ nach Ankunft, Zeit, Termin, Dringlichkeit, Gewicht, ...
 - die Liste repräsentiert sich als statische oder dynamische Datenstruktur

⁵Gemäß Einplanungsstrategie für eine bestimmte Rechnerbetriebsart (Stapel-, Mehrzugangs-, Echtzeitbetrieb).



- Prozesseinplanung profitiert von **Vorwissen** zu Kontrollfluss- und Datenabhängigkeiten, die vorhersehbar sind
 - dann ist ein Ablaufplan möglich, der die Prozesse impliziert koordiniert
- ohne Abhängigkeitsvorwissen sind Prozesse explizit zu koordinieren, per **Programmanweisung** \rightsquigarrow nichtsequentielles Programm
 - der Ablaufplan reiht zwar Prozesse, koordiniert diese jedoch nicht

Definition (Synchronisation [14])

Koordination der Kooperation und Konkurrenz zwischen Prozessen.

- verläuft unterschiedlich, je nach Betriebsmittel- und Prozesszugriffsart

Beachte: Auch vorhergesagte Prozesse finden unvorhersehbar statt, wenn nämlich der Ablaufplan sich als nicht durchsetzbar erweist.

- weil das Vorwissen unvollständig, durch **Ungewissheit** geprägt ist
- weil die **Berechnungskomplexität** den engen Zeitrahmen sprengt
- weil plötzlichem **Hintergrundrauschen** nicht vorgebeugt werden kann
 - Unterbrechungen, Zugriffsfehler auf Zwischen- oder Arbeitsspeicher
 - Befehlsverknüpfung (*pipelining*), Arbitrationslogik (Bus)



- bei unteilbaren Betriebsmitteln greift Synchronisation **multilateral**
 - vorausgesetzt die folgenden beiden Bedingungen treffen zu:
 - i Betriebsmittelzugriffe durch Prozesse geschehen (quasi) **gleichzeitig** und
 - ii bewirken **widerstreitende Zustandsänderungen** des Betriebsmittels
 - Zugriffe auf gemeinsam benutzte Betriebsmittel sind zu koordinieren
 - was sich blockierend oder nichtblockierend auf die Prozesse auswirken kann
 - im blockierenden Fall wird das Betriebsmittel von einem Prozess exklusiv belegt, im nichtblockierenden Fall kann die Zustandsänderung scheitern
- bei konsumierbaren Betriebsmittel wirkt Synchronisation **unilateral**
 - allgemein auch als logische oder bedingte Synchronisation bezeichnet:
 - logisch** – wie durch das Rollenspiel der involvierten Prozesse vorgegeben
 - bedingt** – wie durch eine Fallunterscheidung für eine Berechnung bestimmt
 - Benutzung eines vorübergehenden Betriebsmittels folgt einer Kausalität
 - nichtblockierend für Produzenten und blockierend für Konsumenten
- Prozesse, die gleichzeitig auftreten, überlappen einander zeitweilig
 - sie interagieren zwingend, wenn sie sich dann auch räumlich überlappen
 - dies bedeutet **Interferenz** (*interference*: Störung, Behinderung)...



- fundamentale Primitiven [9] für Erwerb/Abgabe von Betriebsmitteln, wobei die Operationen folgende **intrinsische Eigenschaften** haben:
 - P Abk. für (Hol.) **prolaag**; alias *down*, *wait* oder *acquire*
 - verringert⁶ den Wert des Semaphors s um 1:
 - i genau dann, wenn der resultierende Wert nichtnegativ wäre [10, p. 29]
 - ii logisch uneingeschränkt [11, p. 345]
 - ist oder war der Wert vor dieser Aktion 0, blockiert der Prozess
 - er kommt auf eine mit dem Semaphor assoziierte Warteliste
 - V Abk. für (Hol.) **verhoog**; alias *up*, *signal* oder *release*
 - erhöht⁶ den Wert des Semaphors s um 1
 - ein ggf. am Semaphor blockierter Prozess wird wieder bereitgestellt
 - welcher Prozess von der Warteliste genommen wird, ist nicht spezifiziert
- beide Primitiven sind logisch oder physisch **unteilbare Operationen**, je nachdem, wie dies technisch sichergestellt ist [24]
- ursprünglich definiert als **binärer Semaphor** ($s = [0, 1]$), generalisiert als **allgemeiner Semaphor** ($s = [n, m]$, $m > 0$ und $n \leq m$)

⁶Nicht zwingend durch Subtraktion oder Addition im arithmetischen Sinn.



- als Tabelle implementierte **Universalzeigerliste** begrenzter Länge:

```
1 typedef struct table {
2     size_t get, put;
3     void *bay[TABLE_SIZE];
4 } table_t;
5
6 #define PUT(list,item) list.bay[list.put++ % TABLE_SIZE] = item
7 #define GET(list,item) item = list.bay[list.get++ % TABLE_SIZE]
```

- angenommen, mehrere Prozesse agieren mit GET oder PUT gleichzeitig auf derselben Datenstruktur `list` \leadsto **kritischer Wettlauf**

++ ■ läuft Gefahr, falsch zu zählen (vgl. [18, S. 28])

PUT ■ läuft Gefahr, Listeneinträge zu überschreiben⁷

GET ■ läuft Gefahr, denselben Listeneintrag mehrfach zu liefern⁷

- **Simultanverarbeitung** lässt die beliebige zeitliche Überlappung von Prozessen zu, so dass **explizite Koordinierung** erforderlich wird

⁷Mehrere sich zeitlich überlappende Prozesse könnten denselben Wert aus der Indexvariablen (`put` bzw. `get`) lesen, bevor diese verändert wird.



Multilaterale Synchronisation

- **wechselseitiger Ausschluss** (*mutual exclusion*) sich sonst womöglich überlappender Ausführungen von PUT und GET: **binärer Semaphor**

```
1 typedef struct buffer {
2     semaphore_t lock;
3     table_t data;
4 } buffer_t;
5
6 inline void store(buffer_t *pool, void *item) {
7     P(&pool->lock);           /* enter critical section */
8     PUT(pool->data, item);    /* only one process at a time */
9     V(&pool->lock);           /* leave critical section */
10 }
11
12 inline void *fetch(buffer_t *pool) {
13     void *item;
14     P(&pool->lock);           /* enter critical section */
15     GET(pool->data, item);    /* only one process at a time */
16     V(&pool->lock);           /* leave critical section */
17     return item;
18 }
```

Vorbelegung des Semaphors

```
/* critical section is free */
buffer_t buffer = {{1}};
```

- ein **Unter-/Überlauf** der Universalzeigerliste bzw. des Puffers kann nicht ausgeschlossen werden \rightsquigarrow **Programmierfehler**



Unilaterale Synchronisation

- **Reihenfolgenbildung** von Prozessen, die als Produzent (stuff) oder Konsument (drain) agieren: **allgemeiner Semaphor**

```
1 typedef struct stream {
2     semaphore_t free, full;
3     buffer_t data;
4 } stream_t;
```

Vorbelegung der Semaphore

```
/* all table items available, no consumable
 * critical section is free */
stream_t stream = {{TABLE_SIZE}, {0}, {{1}}};
```

```
5
6 void stuff(stream_t *pipe, void *item) {
7     P(&pipe->free);           /* prevent overflow */
8     store(&pipe->data, item);
9     V(&pipe->full);          /* signal consumable */
10 }
```

```
11
12 void *drain(stream_t *pipe) {
13     void *item;
14     P(&pipe->full);          /* prevent underflow */
15     item = fetch(&pipe->data);
16     V(&pipe->free);         /* signal space */
17     return item;
18 }
```

- typisches Muster der Implementierung eines Klassikers — nicht nur in der Systemprogrammierung: **begrenzter Puffer** (*bounded buffer*)



- Gemeinsamkeit besteht darin, einen gewissen **Vorgang** auszudrücken, Unterschiede ergeben sich in der technischen Auslegung

Prozess

- Ausführung eines Programms (*locus of control* [8])
- seit Multics eng verknüpft mit „eigenem Adressraum“ [6]
- seit Thoth, im **Team** vereint, denselben Adressraum teilend [2]

Faden

- konkreter Strang, roter Faden (*thread*) in einem Programm
- **sequentieller Prozess** [1, S. 78] — ein „Thoth-Prozess“

andere

- Aufgabe (*task*), Arbeit (*job*) ~ Handlung (wie Prozess)
- Faser (*fiber*), Fäserchen (*fibril*) ~ Gewichtsklasse (wie Faden)

separation of concerns [12, S. 1]

Steht das „Was“ (ein ohne Zweifel bestehender Programmablauf) oder das „Wie“ (Art und Grad der Isolation) im Vordergrund der Diskussion?

- Informatikfolklore vermischt Programmablauf und Adressraumschutz
 - ein Prozess bewegt sich in dem Adressraum, den ein Programm definiert
 - das tut er aber unabhängig davon, ob dieser Adressraum geschützt ist
 - wenn überhaupt, dann ist daher sein Programmspeicher zu schützen...



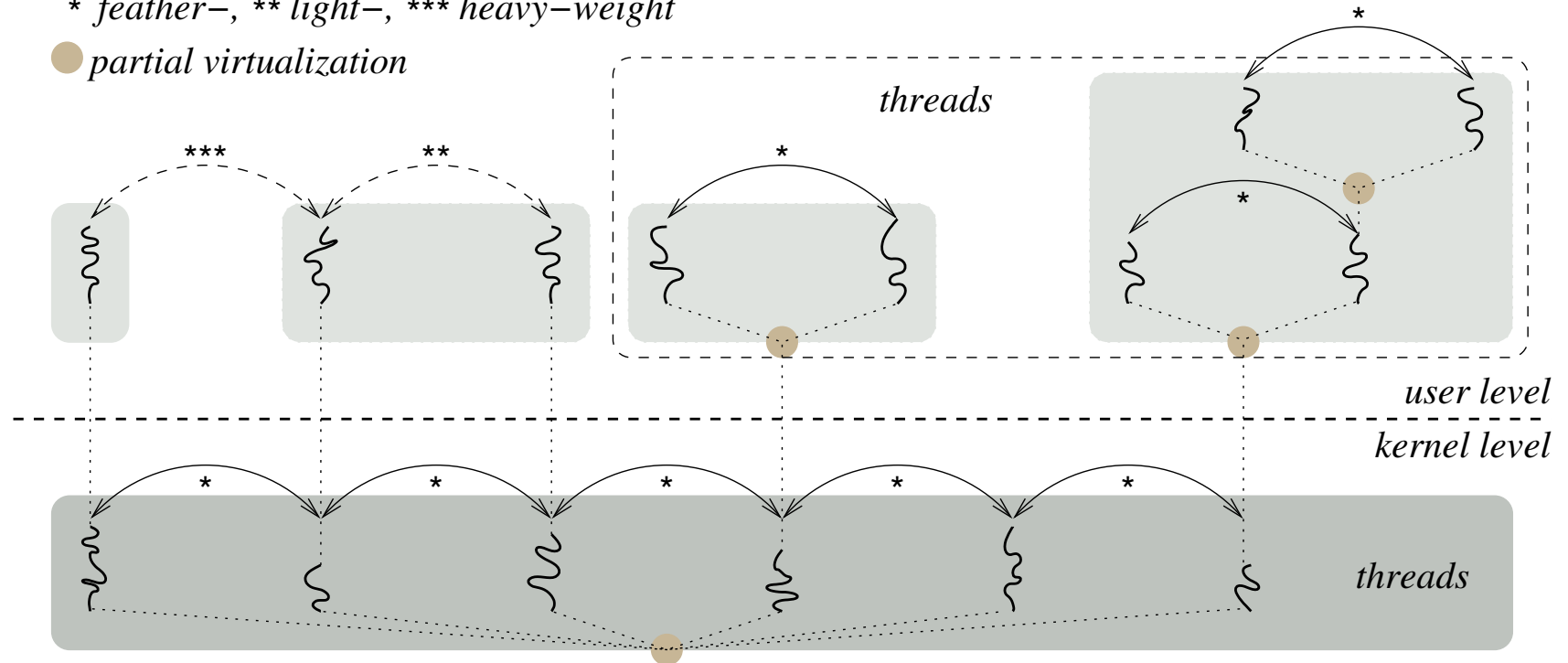
Prozesse sind in einem Rechner-System verschiedenartig verankert

- unter oder auf der Maschinenprogrammzebene
 - unter**
 - ursprünglich, im Betriebssystem bzw. Kern (*kernel*)
 - Prozessinkarnation als Wurzel
 - partielle Virtualisierung des realen Prozessor(kern)s
 - ↪ „*kernel-level thread*“ in der Informatikfolklore
 - auf**
 - optional, im Laufzeit- oder gar Anwendungssystem
 - Prozessinkarnation als Blatt oder innerer Knoten
 - partielle Virtualisierung eines abstrakten Prozessor(kern)s
 - ↪ „*user-level thread*“ in der Informatikfolklore
- der jew. Prozessor weiß nicht, dass er ggf. (partiell) virtualisiert wird
 - ein „*user-level thread*“ ist ein in Zeit gemultiplexer „*kernel-level thread*“
 - einem „*kernel-level thread*“ sind seine „*user-level threads*“ unbewusst
- Betriebssysteme kennen nur ihre eigenen Prozessinkarnationen
 - ein „*kernel-level thread*“ entsteht durch Raum-/Zeitmultiplexen der CPU
 - hält ein „*kernel-level thread*“ inne, setzen seine „*user-level threads*“ aus



* feather-, ** light-, *** heavy-weight

● partial virtualization



- Arten von **Prozesswechsel** zur partiellen Prozessorvirtualisierung:
 - * im selben (Anwendungs-/Kern-) Adressraum, ebenda fortsetzend
 - ** im Kernadressraum, denselben Anwendungsadressraum teilend
 - *** im Kernadressraum, im anderen Anwendungsadressraum landend



- der **Prozesskontrollblock**⁸ (*process control block*, PCB) bündelt alle zur partiellen Virtualisierung relevanten Attribute eines Prozesses
 - in dem (pro Prozess) typischerweise folgende Daten verbucht sind:
 - Adressraum, Speicherbelegung, Laufzeitkontext, . . . , Ressourcen allgemein
 - Verarbeitungszustand, Blockierungsgrund, Dringlichkeit, Termin
 - Name, Domäne, Zugehörigkeit, Befähigung, Zugriffsrechte, Identifikationen
 - als die zentrale **Informations- und Kontrollstruktur** im Betriebssystem
- pro Prozessor verwaltet das Betriebssystem einen **Prozesszeiger**, der die jeweils laufende Prozessinkarnation identifiziert
 - so, wie der Befehlszähler der CPU den laufenden Befehl adressiert, zeigt der Prozesszeiger des Betriebssystems auf den gegenwärtigen Prozess
 - beim Prozesswechsel (*dispatch*) wird der Prozesszeiger weitergeschaltet
- nach außen wird eine so beschriebene Prozessinkarnation systemweit eindeutig durch eine **Prozessidentifikation** (PID) repräsentiert
 - wobei „systemweit“ recht dehnbar ist und sich je nach Auslegung auf ein Betriebssystem, ein vernetztes System oder verteiltes System bezieht

⁸auch: **Prozessdeskriptor** (*process descriptor*, PD).



Gliederung

Einführung
Begriff

Grundlagen
Virtualität
Betriebsmittel
Programme

Verwaltung
Planung
Synchronisation
Implementierungsaspekte

Zusammenfassung



- in der Einführung zunächst prinzipielle **Begrifflichkeiten** erklärt
 - einen **Prozess** als „Programm in Ausführung“ definiert und damit die originale (klassische) Definition [7] übernommen
 - den Unterschied zur **Prozessinkarnation**/-verkörperung hervorgehoben
- darauf aufbauend wichtige **Grundlagen** zum Thema behandelt
 - partielle Virtualisierung und **Simultanverarbeitung**
 - **Betriebsmittel**, deren Klassifikation und Eigentümlichkeiten
 - Programm als Verarbeitungsvorschrift für eine Folge von **Aktionen**
 - **nichtsequentielles Programm**, das Aktionen für Parallelität spezifiziert
- verschiedene Aspekte der **Ausprägung** von Prozessen beleuchtet:
 - Planung** ■ implizite Koordinierung, **Einplanung** von Prozessen
 - logische Verarbeitungszustände, **Einlastung**
 - Synchronisation** ■ explizite **Koordinierung** durch Programmanweisung
 - binärer/allgemeiner **Semaphor**, Abgrenzung *Mutex*
 - Repräsentation** ■ **Verortung** der Prozesse im Rechensystem
 - **Fäden** inner-/oberhalb der Maschinenprogrammebene
 - ↪ Ressource: Prozesskontrollblock, -zeiger, -identifikation



Literaturverzeichnis I

- [1] BAUER, F. L. ; GOOS, G. :
Betriebssysteme.
In: *Informatik: Eine einführende Übersicht* Bd. 90.
Springer-Verlag, 1971, Kapitel 6.3, S. 76–92
- [2] CHERITON, D. R. ; MALCOLM, M. A. ; MELEN, L. S.:
Thoth, a Portable Real-Time Operating System.
In: *Communications of the ACM* 22 (1979), Febr., Nr. 2, S. 105–115
- [3] COFFMAN, E. G. ; DENNING, P. J.:
Operating System Theory.
Prentice Hall, Inc., 1973
- [4] CONWAY, R. W. ; MAXWELL, L. W. ; MILLNER, L. W.:
Theory of Scheduling.
Addison-Wesley, 1967
- [5] CORBATÓ, F. J. ; MERWIN-DAGGETT, M. ; DALEX, R. C.:
An Experimental Time-Sharing System.
In: *Proceedings of the AIEE-IRE '62 Spring Joint Computer Conference*, ACM,
1962, S. 335–344



Literaturverzeichnis II

- [6] DALEY, R. C. ; DENNIS, J. B.:
Virtual Memory, Processes, and Sharing in MULTICS.
In: *Communications of the ACM* 11 (1968), Mai, Nr. 5, S. 306–312
- [7] DENNING, P. J.:
Third Generation Computer Systems.
In: *Computing Surveys* 3 (1971), Dez., Nr. 4, S. 175–216
- [8] DENNIS, J. B. ; HORN, E. C. V.:
Programming Semantics for Multiprogrammed Computations.
In: *Communications of the ACM* 9 (1966), März, Nr. 3, S. 143–155
- [9] DIJKSTRA, E. W.:
Over seinpalen / Technische Universiteit Eindhoven.
Eindhoven, The Netherlands, 1964 ca. (EWD-74). –
Manuskript. –
(dt.) Über Signalmasten



Literaturverzeichnis III

- [10] DIJKSTRA, E. W.:
Cooperating Sequential Processes / Technische Universiteit Eindhoven.
Eindhoven, The Netherlands, 1965 (EWD-123). –
Forschungsbericht. –
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New
York, NY, 1996)
- [11] DIJKSTRA, E. W.:
The Structure of the “THE”-Multiprogramming System.
In: *Communications of the ACM* 11 (1968), Mai, Nr. 5, S. 341–346
- [12] DIJKSTRA, E. W.:
On the Role of Scientific Thought.
<http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF>, Aug. 1974
- [13] DRESCHER, G. ; SCHRÖDER-PREIKSCHAT, W. :
An Experiment in Wait-Free Synchronisation of Priority-Controlled Simultaneous
Processes: Guarded Sections / Friedrich-Alexander-Universität Erlangen-Nürnberg,
Department of Computer Science.
Erlangen, Germany, Jan. 2015 (CS-2015-01). –
Technical Reports



Literaturverzeichnis IV

- [14] HERRTWICH, R. G. ; HOMMEL, G. :
*Kooperation und Konkurrenz — Nebenläufige, verteilte und echtzeitabhängige
Programmsysteme.*
Springer-Verlag, 1989. –
ISBN 3-540-51701-4
- [15] HOLT, R. C.:
On Deadlock in Computer Systems.
Ithaca, NY, USA, Cornell University, Diss., 1971
- [16] HOLT, R. C.:
Some Deadlock Properties of Computer Systems.
In: *ACM Computing Surveys* 4 (1972), Sept., Nr. 3, S. 179–196
- [17] IEEE:
POSIX.1c Threads Extensions / Institute of Electrical and Electronics Engineers.
New York, NY, USA, 1995 (IEEE Std 1003.1c-1995). –
Standarddokument
- [18] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :
Betriebssystemmaschine.
In: LEHRSTUHL INFORMATIK 4 (Hrsg.): *Systemprogrammierung.*
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien), Kapitel 5.3



Literaturverzeichnis V

- [19] KLEINROCK, L. :
Queuing Systems. Bd. I: Theory.
John Wiley & Sons, 1975
- [20] LISTER, A. M. ; EAGER, R. D.:
Fundamentals of Operating Systems.
The Macmillan Press Ltd., 1993. –
ISBN 0–333–59848–2
- [21] LÖHR, K.-P. :
Nichtsequentielle Programmierung.
In: INSTITUT FÜR INFORMATIK (Hrsg.): *Algorithmen und Programmierung IV*.
Freie Universität Berlin, 2006 (Vorlesungsfolien)
- [22] SCHRÖDER-PREIKSCHAT, W. ; LEHRSTUHL INFORMATIK 4 (Hrsg.):
Concurrent Systems — Nebenläufige Systeme.
FAU Erlangen-Nürnberg, 2014 (Vorlesungsfolien)
- [23] SCHRÖDER-PREIKSCHAT, W. :
Guarded Sections.
In: [22], Kapitel 10



Literaturverzeichnis VI

[24] SCHRÖDER-PREIKSCHAT, W. :
Semaphore.
In: [22], Kapitel 7

[25] WIKIPEDIA:
Prozess.
<http://de.wikipedia.org/wiki/Prozess>, Nov. 2013



Ursprünglich als Rechtsbegriff

Prozess bedeutet „streitiges Verfahren vor Gericht, mit dem Ziel, den Streit durch eine verbindliche Entscheidung zu klären“ [25, Recht]

- Analogie in der Informatik bzw. zu Betriebssystemkonzepten:

Streit ■ Rivalität⁹ bei Inanspruchnahme von Betriebsmitteln

■ Konkurrenz (lat. *concurrere* zusammenlaufen)

Verfahren ■ Vorgehensweise zur planmäßigen Problemlösung

■ Strategie (*policy*) oder Methode der Problemlösung

Gericht ■ Funktion zur Einplanung (*scheduling*), Koordinierung

■ Synchronisationspunkt in einem Programm

Verbindlichkeit ■ Konsequenz, mit der die Einplanungszusagen gelten

■ Einhaltung zugesagter Eigenschaften, Verlässlichkeit

- in der Regel folgen die Verfahren einer hierarchischen Gerichtsbarkeit

■ Betriebssysteme verfügen oft über eine mehrstufige Prozessverarbeitung

■ was aber kein Verfahrensabschnitt, keine Instanz (*instance*) impliziert

– Übernahme von „Instanz“ in die Informatik war eher ungeschickt

!!!

⁹lat. *rivalis* „an der Nutzung eines Wasserlaufs mitberechtigter Nachbar“



- Adressbereich und virtuelle Maschine SMC¹⁰
 - Textsegment
 - Linux
- reale Maschine
 - nach dem Binden und
 - vor dem Laden
- gleiche Anzahl von Aktionen (Zeilen 1–3, jew.), aber verschiedene Darstellungsformen

1	0x080482f0:	mov 0x4(%esp),%eax	8b 44 24 04
2	0x080482f4:	add \$0x1, (%eax)	83 00 01
3	0x080482f7:	adc \$0x0, 0x4(%eax)	83 50 04 00
4	0x080482fb:	ret	c3

Hinweis (ret bzw. c3)

Die Aktion zum Unterprogrammrücksprung korrespondiert zur Aktion des Unterprogrammaufrufs (gdb, disas /rm main):

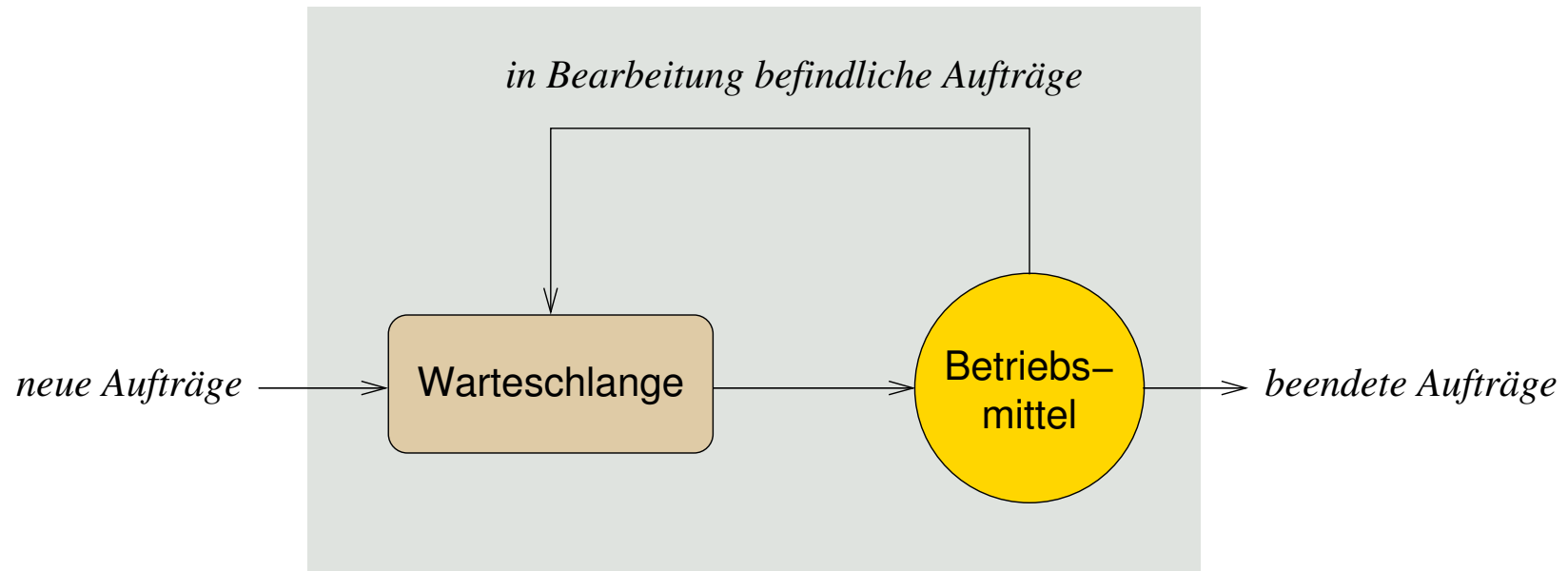
1	0x080481c9:	c7 04 24 b0 37 0d 08	movl \$0x80d37b0, (%esp)
2	0x080481d0:	e8 1b 01 00 00	call 0x80482f0 <inc64>

¹⁰symbolischer Maschinenkode (*symbolic machine code*): x86 + Linux.



Einplanungsalgorithmen

- Verwaltung von (betriebsmittelgebundenen) **Warteschlangen**



Ein einzelner Einplanungsalgorithmus ist charakterisiert durch die Reihenfolge von Prozessen in der Warteschlange und die Bedingungen, unter denen die Prozesse in die Warteschlange eingereiht werden. [20]



Warteschlangentheorie und -praxis

- die Charakterisierung von **Einplanungsalgorithmen** macht glauben, Betriebssysteme fokussiert „mathematisch“ studieren zu müssen:
 - R. W. Conway, L. W. Maxwell, L. W. Millner. *Theory of Scheduling*.
 - E. G. Coffman, P. J. Denning. *Operating System Theory*.
 - L. Kleinrock. *Queuing Systems, Volume I: Theory*.
- praktische Umsetzung offenbart jedoch einen **Querschnittsbelang** (*cross-cutting concern*), der sich kaum modularisieren lässt
 - spezifische Betriebsmittelmerkmale stehen ggf. Bedürfnissen der Prozesse, die Aufträge zur Betriebsmittelnutzung abgesetzt haben, gegenüber
 - dabei ist die Prozessreihenfolge in Warteschlangen (bereit, blockiert) ein Aspekt, die Auftragsreihenfolge dagegen ein anderer Aspekt
 - **Interferenz** bei der Durchsetzung der Strategien kann die Folge sein
- Einplanungsverfahren stehen und fallen mit den Vorgaben, die für die jeweilige **Zieldomäne** zu treffen sind
 - die „Eier-legende Wollmilchsau“ kann es nicht geben
 - Kompromisslösungen sind geläufig — aber nicht in allen Fällen tragfähig



- ein **Synchronisationsverfahren**, das die Formulierung unteilbarer Aktionsfolgen eines nichtsequentiellen Programms unterstützt
 - wobei eine solche Aktionsfolge einem kritischen Abschnitt entspricht

Definition (Kritischer Abschnitt)

Ein Programmabschnitt, der bei nichtsequentieller Ausführung durch **gleichzeitige Prozesse** einen **kritischen Wettlauf** impliziert:

*critical in the sense, that the processes have to be constructed in such a way, that at any moment at most one of [them] is engaged in its **critical section**. [10, S. 11]*

- S. 24 zeigt solche Abschnitte — jedoch ist wechselseitiger Ausschluss zur Vorbeugung eines kritischen Wettlaufs (dort) nicht zwingend

Axiom

(s. [23, 13])

Jede „aufgefährliche Aktionsfolge“ lässt sich ohne wechselseitigen Ausschluss absichern \rightsquigarrow **nichtblockierende Synchronisation**.



- die Semaphorprimitiven P und V sind so definiert (S. 22), paarweise verwendet zu werden — nicht aber zwingend vom selben Prozess
 - sonst wäre einseitige (unilaterale, logische, bedingte) Synchronisation von Prozessen unmöglich \leadsto **allgemeiner Semaphor**
 - sonst wäre mehrseitige (multilaterale) Synchronisation für einen kritischen Abschnitt, der den Prozess wechselt, falsch¹¹ \leadsto **binärer Semaphor**
- in der Informatikfolklore wird dies jedoch verschiedentlich als Makel angesehen und damit ein alternatives „Konzept“ motiviert

Definition (Mutex)

Ein **spezialisierter binärer Semaphor** s , der Aktion $V(s)$ nur dem Prozess, der zuvor die Aktion $P(s)$ verantwortet hat, erlaubt.

- unautorisierte Verwendung von $V(s)$ gilt als **schwerwiegender Fehler**
 - der fälschlicherweise $V(s)$ durchführende Prozess ist abubrechen!
- \hookrightarrow allerdings fehlt dieses Merkmal, POSIX: *an error shall be returned* ☹

¹¹Prozessumschaltung innerhalb von Betriebssystemen ist typischer Kandidat dafür: Ein anderer Prozess muss den kritischen Abschnitt verlassen!



- die klassischen Semaphorprimitiven von Dijkstra sind direkt abbildbar auf **semantisch äquivalente Operationen** von POSIX:

```
1 #include <semaphore.h>
2
3 typedef struct semaphore {
4     sem_t sema;
5 } semaphore_t;
6
7 inline void P(semaphore_t *sema) {
8     sem_wait(&sema->sema);
9 }
10
11 inline void V(semaphore_t *sema) {
12     sem_post(&sema->sema);
13 }
```

42

Wert für *pshared*, der hier nur ungleich 0 sein sollte, damit der betreffende Semaphor auch von „Fäden“ anderer „Prozesse“ mitbenutzbar (*shared*) ist.

- nur die Initialisierung des POSIX-Semaphors gestaltet sich anders:

```
lock ■ sem_init(&pipe->data.lock.sema, 42, 1)
free ■ sem_init(&pipe->free.sema, 42, TABLE_SIZE)
full ■ sem_init(&pipe->full.sema, 42, 0)
```

- **Laufzeitinitialisierung** ist die Regel, nicht Übersetzungs- oder Bindezeit

