Systemprogrammierung

Grundlage von Betriebssystemen

Teil B – VI.1 Betriebssystemkonzepte: Prozesse

Wolfgang Schröder-Preikschat

7. Juni 2016

Agenda

Gliederung

Lehrstoff

- **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
- Prozessausprägungen und zugehörige Systemfunktionen beleuchten
  - typische (logische) Verarbeitungszustände von Prozessen einführen
  - Einplanung (*scheduling*) und Einlastung (*dispatching*) differenzieren
  - mehr- und einseitige Synchronisation beispielhaft zeigen: *bounded buffer*
  - Verortung von Prozessen auf Benutzer- und Systemebene skizzieren
  - Prozesskontrollblock, -zeiger und -identifikation begrifflich erfassen

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	▪ Rivalität[1] 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 !!!

[1] lat. *rivalis* „an der Nutzung eines Wasserlaufs mitberechtigter Nachbar“

Rezipiert als Informatikbegriff

Definition (Prozess ≡ 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

Ebene $_5$	↦ Programmanweisung	≥ 1 Assembliermnemoniks
Ebene $_4$	↦ Assembliermnemonik	≥ 1 Maschinenbefehle
Ebene $_3$	↦ Maschinenbefehl	≥ 1 Mikroprogramminstruktionen
Ebene $_2$	↦ 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 ≠ Prozessinkarnation)

*Eine Prozessinkarnation ist **Exemplar** eines Prozesses als **Bautyp**, wie ein Objekt Exemplar einer Klasse ist.*

Gliederung

Partielle Virtualisierung

Prozessorvirtualisierung

- 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*)
  - zum Ablaufstart last das Betriebssystem einen Prozess ein (*dispatching*)
    - Laufzeitkontext umschalten aktiviert dann einen anderen Programmablauf
  - hierzu plant das Betriebssystem Prozesse entsprechend ein (*scheduling*)
- geläufig ist das **Zeitteilverfahren** (*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. 13) ab

Simultanverarbeitung

multiprocessing

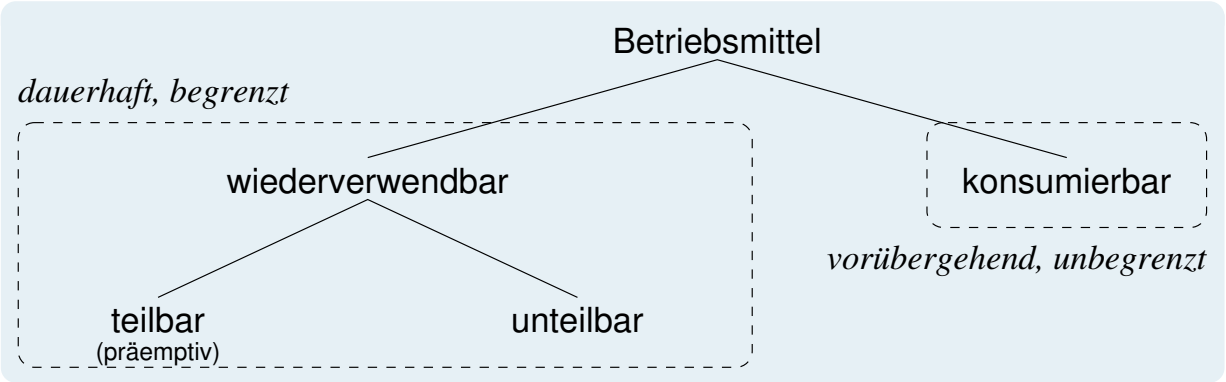
- Prozesse sind das Mittel zum Zweck, (pseudo/quasi) **gleichzeitige Programmabläufe** stattfinden zu lassen ⇝ **Parallelität**

  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 — angenommen korrekten — Programm daselbst
    - wodurch sich letzteres als **nichtsequentielles Programm** auszeichnet
- Synchronisation gestaltet sich recht unterschiedlich, je nach Art des Betriebsmittels und Zweck des Prozesszugriffs

Betriebsmittelklassifikation

nach [15, 16]



- dauerhafte[2] Betriebsmittel sind immer nur von begrenztem Vorrat ☹
  - 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 unbegrenzt verfügbar ☺
  - sie werden produziert, empfangen, benutzt und danach zerstört

[2] auch: persistente

Eigentümlichkeiten von Betriebsmitteln

- vom Betriebssystem zu verwaltende Hardwarebetriebsmittel:

wiederverwendbar		konsumierbar	
Prozessor	CPU, FPU, GPU; MMU	Signal	IRQ, NMI, *trap*
Speicher	RAM, *scratch pad*, *flash*		
Peripherie	Ein-/Ausgabe, *storage*		

- von jedem Programm verwaltete Softwarebetriebsmittel:

wiederverwendbar		konsumierbar	
Text	kritischer Abschnitt	Signal	Meldung
Daten	Variable, Platzhalter	Nachricht	Datenstrom

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

- gleichzeitige Zugriffe auf unteilbare und Zugriffe auf konsumierbare 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

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

Aktionen

Definition (Programm)

Die für eine Maschine konkretisierte Form eines Algorithmus.

- virtuelle Maschine C
  - nach der Editierung und
  - vor der Kompilierung

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

- eine Aktion (Zeile 4)

- virtuelle Maschine ASM (x86)
  - nach der Kompilierung[3] und
  - vor der Assemblierung

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

- drei Aktionen (Zeilen 12–14)

Definition (Aktion)

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

[3] `gcc -O3 -m32 -static -fomit-frame-pointer -S`

Nichtsequentielles Programm I

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 Programm II

- trotz Aktionen für Parallelität, 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

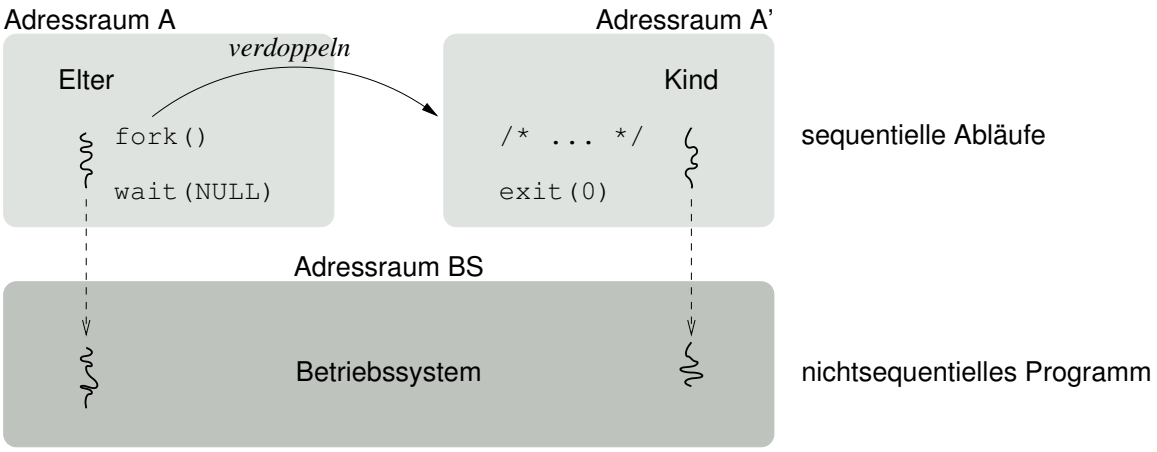
  - Programm P spezifiziert zwar Aktionen, die Parallelität zulassen, diese kommt jedoch nur allein durch `fork` nicht in P selbst zur Wirkung

- 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 Programms[4]

[4] Ausnahmen (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. 8)[5]
  - echte: durch *Vervielfachung* eines realen Prozessors
- beiden Fällen gemeinsam sind **parallele Prozesse** im Betriebssystem
  - auch als **nichtsequentielle Prozesse** bezeichnet
  - nämlich Prozesse, deren Aktionen sich zeitlich überlappen können

[5] (gr.) *pseúdein* belügen, täuschen

Gliederung

Planung der Einlastung

„scheduling“ von *„dispatching“*

Prozesse werden gestartet, unterbrochen, fortgesetzt und beendet

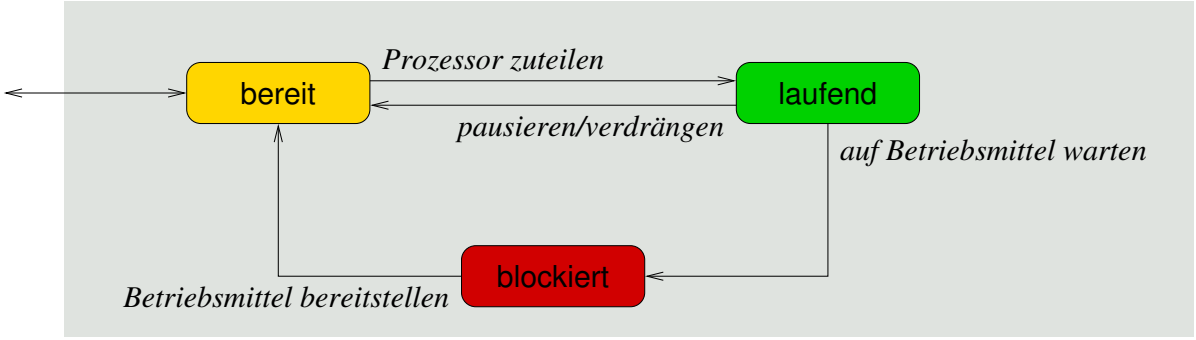
- zentrale Funktion dabei die **Prozesseinplanung** (*process scheduling*), die allgemein zwei grundsätzliche Fragestellungen zu lösen hat:
  - i Zu welchem (logischen/physikalischen) Zeitpunkt sollen Prozesse in den Kreislauf der Programmverarbeitung eingespeist werden?
  - ii In welcher Reihenfolge 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.

Verarbeitungszustände von Prozessen

- 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

- die **Bereitliste** (*ready list*) definiert einen **Ablaufplan** (*schedule*) zur Prozessorzuteilung, der zur Laufzeit fortgeschrieben wird
  - Elemente dieser Liste sind **Prozesskontrollblöcke** (siehe S. 30), geordnet nach Ankunft, Zeit, Termin, Dringlichkeit, Gewicht, ...
    - gemäß **Einplanungsstrategie**, d.h., zur Unterstützung einer bestimmten **Rechnerbetriebsart** (Stapel-, Mehrzugangs-, Echtzeitbetrieb)
- allgemein bedeutet Planung innerhalb eines Betriebssystems:

Prozessen das Betriebsmittel { **Prozessor**, **Speicher**, **Gerät**, ⋮, **Datei**, **Nachricht**, **Puffer** } koordiniert zuteilen

  - Betriebsmittel, die in **Hardware** oder **Software** ausgeprägt vorliegen

Implizite vs. Explizite Koordinierung

- Prozesseinplanung profitiert von **Vorwissen** zu Kontrollfluss- und Datenabhängigkeiten, die vorhersehbar sind
  - dann ist ein Ablaufplan möglich, der die Prozesse impliziert koordiniert
- ohne solch Vorwissen sind die 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 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)

Synchronisationsarten

mehr-/einseitige Synchronisation

- 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 Synchronistaion **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)...

Koordinationsmittel: Semaphor

- 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*
  - verringert[6] 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*
  - erhöht[6] 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 $n \leq m$)

[6] Nicht zwingend durch Subtraktion oder Addition im arithmetischen Sinn.

Fallbeispiel

. . . für kritische Abschnitte

- 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` ⇝ **kritischer Wettlauf**

  `PUT` ■ läuft Gefahr, Listeneinträge zu überschreiben[7]

  `GET` ■ läuft Gefahr, denselben Listeneintrag mehrfach zu liefern[7]

  `++` ■ läuft Gefahr, falsch zu zählen (vgl. [18, S. 28])
- **Simultanverarbeitung** lässt die beliebige zeitliche Überlappung von Prozessen zu, so dass **explizite Koordinierung** erforderlich wird

[7] Mehrere 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 ⇝ **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;
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 }
```

Vorbelegung der Semaphore

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

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

Prozess ↔ Faden

process ↔ thread

- 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*), Teilaufgabe (*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...

Verortung

Betriebssystem- vs. Anwendungsdomäne

Prozesse sind in einem Rechensystem verschiedenartig verankert

- und zwar inner- oder oberhalb der Maschinenprogrammebene

  innerhalb
  - ursprünglich, im Betriebssystem bzw. Kern (*kernel*)
  - Prozessinkarnation als Wurzel
  - partielle Virtualisierung des Prozessor(kern)s

  ↪ „*kernel thread*“ in der Informatikfolklore

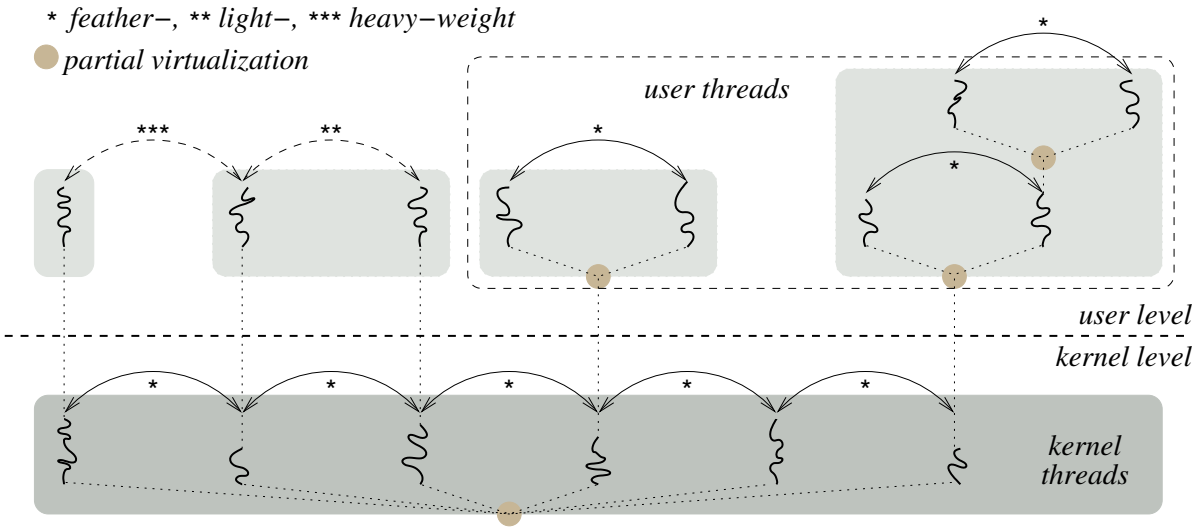
  oberhalb
  - optional, im Laufzeit- oder gar Anwendungssystem
  - Prozessinkarnation als Blatt oder innerer Knoten
  - partielle Virtualisierung der Wurzelprozessinkarnation

  ↪ „*user thread*“ in der Informatikfolklore

- die jew. Entität weiß nicht, dass sie ggf. (partiell) virtualisiert wird
  - einem „*kernel thread*“ sind seine „*user threads*“ gänzlich unbewusst
  - ein „*user thread*“ entsteht durch Zeitmultiplexen eines „*kernel threads*“
- Betriebssysteme kennen nur ihre eigenen Prozessinkarnationen
  - schaltet ein „*kernel thread*“ weg, setzen alle seine „*user threads*“ aus
  - ein „*kernel thread*“ entsteht durch Raum-/Zeitmultiplexen der CPU

Gewichtsklasse

Unterbrechungs-/Wiederaufnahmeaufwand



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

Verkörperung

Inkarnation

- 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

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

Gliederung

Resümee

. . . Trennung von Belangen trägt zum Verständnis bei

- 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:
  - Arbitration
    - implizite Koordinierung durch **Einplanung**
    - 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 Though.
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

Aktionen II

- Adressbereich und virtuelle Maschine SMC[9]
  - Textsegment
  - Linux
- nach dem Binden und
  - vor dem Laden
- reale Maschine
  - nach dem Laden
  - ablauffähig

```
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
```

- gleiche Anzahl von Aktionen (Zeilen 1–3, jew.), aber verschiedene Darstellungsformen

Hinweis (`ret` bzw. c3)

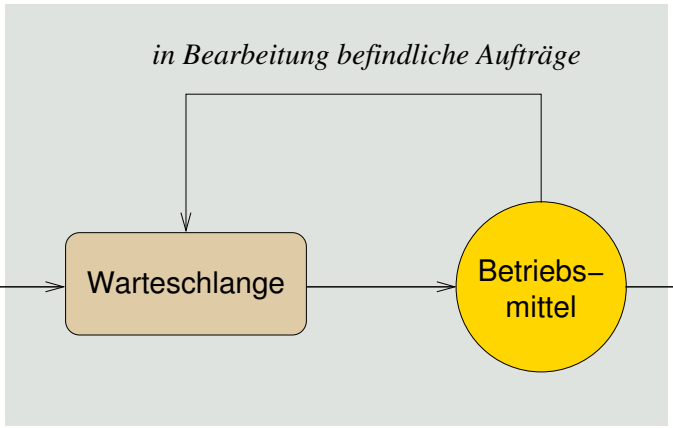
Die Aktion zum Unterprogrammrücksprung korrespondiert zur Aktion des Unterprogrammaufrufs (`gdb, disas /rm main`):

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

[9] symbolischer 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

- 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

Zwischenbemerkung I: Wechselseitiger Ausschluss

- 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. 25 zeigt solche Abschnitte — jedoch ist wechselseitiger Ausschluss zur Vorbeugung eines kritisches Wettlaufs (dort) nicht zwingend

Axiom (s. [23, 13])

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

Zwischenbemerkung II: *Mutex*

mutual exclusion

- die Semaphorprimitiven P und V sind so definiert (S. 23), paarweise verwendet zu werden — nicht aber zwingend vom selben Prozess
  - sonst wäre einseitige (unilaterale, logische, bedingte) Synchronisation von Prozessen unmöglich $\rightsquigarrow$ **allgemeiner Semaphor**
  - sonst wäre mehrseitige (multilaterale) Synchronisation für einen kritischen Abschnitt, der den Prozess wechselt, falsch[10] $\rightsquigarrow$ **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 abzubrechen!
  - umso unbegreiflicher ist es, dass dies kein implementierter *Mutex* leistet... ☹

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

Koordinationsmittel: Semaphor $\mapsto$ POSIX

- 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