Skript zur Vorlesung:

Einführung in die Informatik: Systeme und Anwendungen

Sommersemester 2009

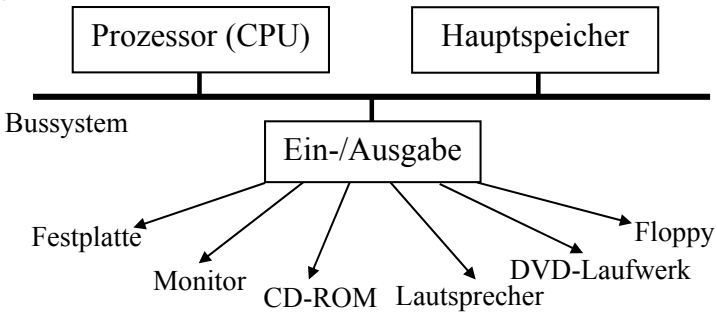
Kapitel 2: Betriebssysteme

Vorlesung: Dr. Peer Kröger
Übungen: Thomas Bernecker

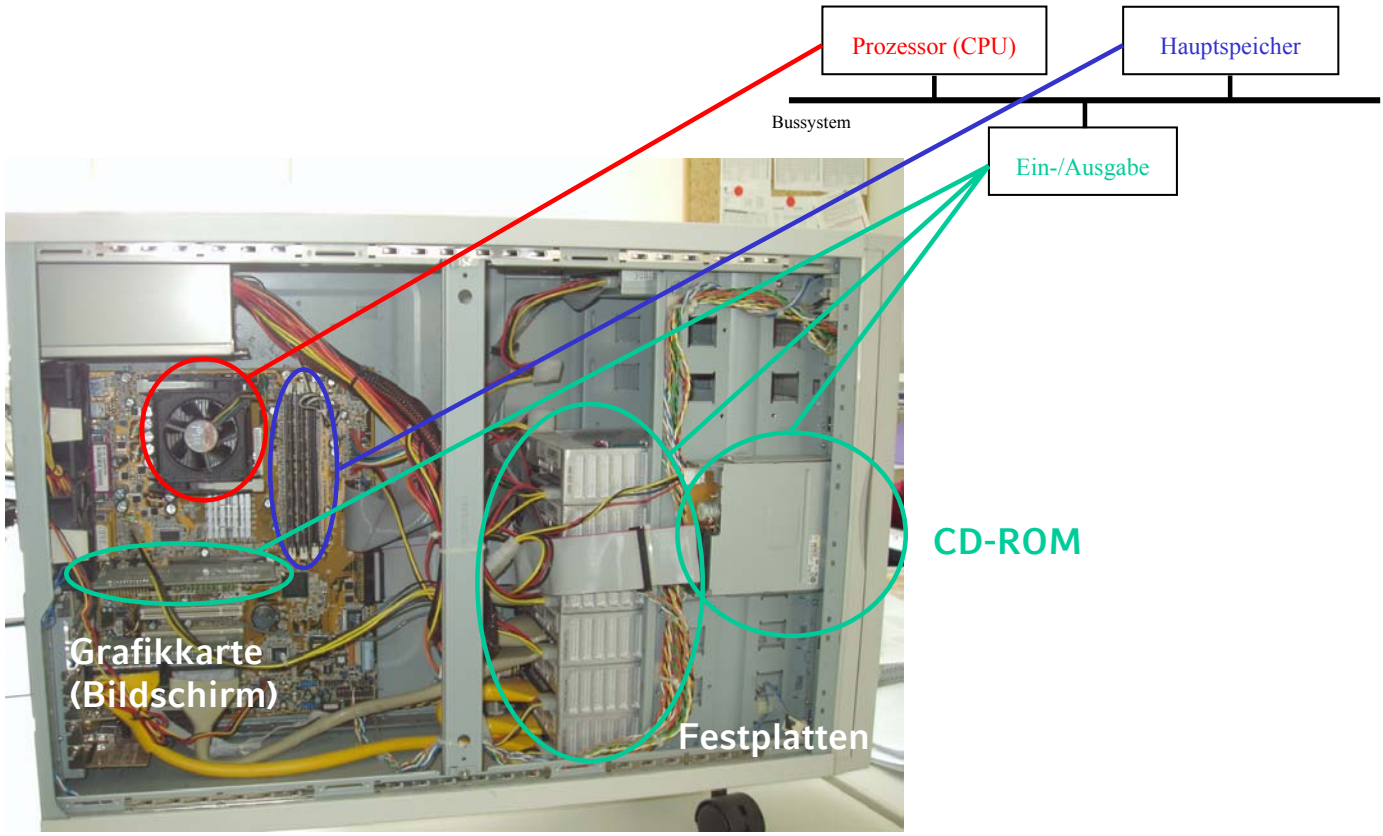
http://www.dbs.ifi.lmu.de/Lehre/InfoNF

2.1 Rechner und Programme

- Komponenten eines Rechners (von-Neumann-Architektur)/Hardware
  - CPU (Prozessor)
    - Ausführung von Befehle und Ablaufsteuerung
  - Speicher (Hauptspeicher)
    - Ablegen von Daten und Programmen, binär codiert
  - Ein-/Ausgabe-Einheiten
    - Ein- und Ausgabe von Daten und Programmen
    - Bildschirm, Tastatur, Drucker, Festplatte, ...
  - Busse
    - Informationsübertragung zwischen diesen Einheiten



2.1 Rechner und Programme



2.1 Rechner und Programme

- Beobachtung:
  - Hauptspeicher reicht nicht aus um alle Daten zu halten
    ⇒ Daten und Programme, die aktuelle nicht benötigt werden müssen auf den Hintergrundspeicher (Festplatte) ausgelagert werden
  - CPU arbeitet schneller als Hauptspeicher
  - Hauptspeicher arbeitet schneller als Hintergrundspeicher
- Flaschenhals:
  - CPU muss warten, bis benötigte Daten bzw. Befehle vom Hauptspeicher geholt worden sind
  - Teilweise sind die angeforderten Daten noch nicht im Hauptspeicher sondern müssen erst vom Hintergrundspeicher geladen werden

  ⇒ CPU ist nicht ausgelastet!!!

2.1 Rechner und Programme

- Ausführung von Programmen auf Rechnern
  - Situation:
    - Zentrale Verarbeitungsschritte als „Maschinenanweisungen“ („Mikroprogramm“, ca. 50 – 300 Anweisungen je nach Hersteller)
      - Lesen/Schreiben einer Speicherzelle
      - Einfache Arithmetik
      - etc.
    - Daten sind maschinennah (meist binär) repräsentiert
  - Darstellung von Programmen:
    - Direkte Programmierung der Hardware mit „maschinennaher“ Programmiersprache (Beispiel Fahrtkosten-Algorithmus)

```
pushl    %ebp
movl     %esp, %ebp
movl     16(%ebp), %eax
subl     8(%ebp), %eax
leal     (%eax,%eax,2), %edx
...
```

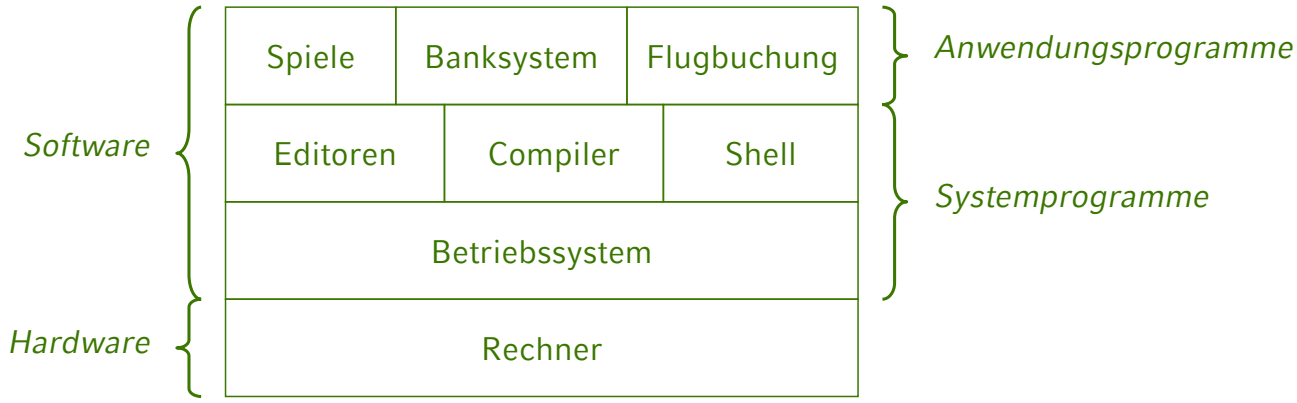

2.1 Rechner und Programme

    - Programmierung muss sich auch um die einzelnen Hardware-Komponenten kümmern, die benutzt werden sollen
    - Beispiel: Laden einer Datei von Festplatte
      1. [optional] starten des Laufwerkmotors
      2. Positionierung des Lesekopfs
      3. Sektorenweise Einlesen des Directory
      4. Suchen der Dateiinformationen (Dateianfang) im Directory
      5. Positionierung des Lesekopfs
      6. Teil einlesen, Verknüpfung zum nächsten Teil erkennen und weiter mit Schritt 5 solange Ende der Datei noch nicht erreicht

      ⇒ *Für jeden Befehl an den Plattenkontroller werden die Adresse der Spur, die Adresse der Hauptspeicherzelle (Ziel), die Menge der zu übertragenden Daten, der Befehlscode (Lesen/Schreiben), etc. benötigt*
  - Problem:
    - Programm ist sehr unübersichtlich und für Menschen schwer zu verstehen
    - Hardware ist ebenfalls sehr komplex und besteht aus vielfältigen Komponenten, deren Realisierungsdetails ebenfalls schwer für den Menschen zu verstehen sind

2.1 Rechner und Programme

- Lösung: Prinzip der „Software-Schichtung“
  - BS bildet Schnittstelle für Anwendungsprogramme und spezielle Systemprogramme zur Hardware
    - ⇒ BS bewahrt den Nutzer vor der Komplexität der HW
    - ⇒ ermöglicht indirekt Programmiersprachen, die für den Menschen leichter zu verstehen/benutzen sind
  - BS bildet SW-Schicht die alle Teile des Systems verwaltet und auf dem Anwendungen einfacher zu programmieren sind



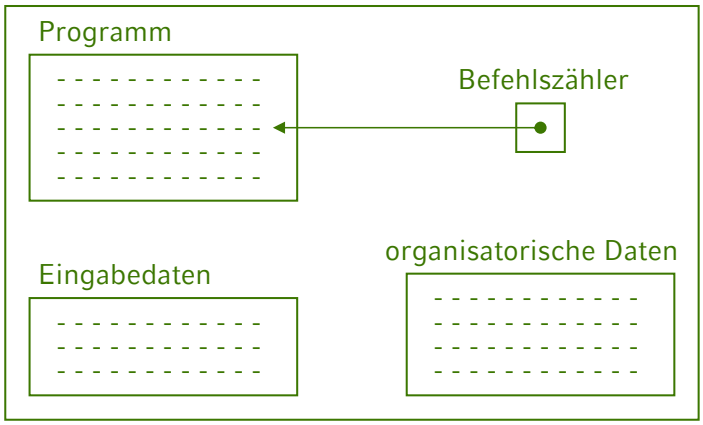
2.1 Rechner und Programme

- Aufgaben des Betriebssystems
  - Schnittstelle für Anwendungsprogramme zur HW
  - Steuerung und Verwaltung von Computersystemen
    - ***Prozessverwaltung***: Steuerung der Ausführung eines oder mehrerern Prozesse, insbesondere im Mehrprogrammbetrieb
    - ***Speicherverwaltung*** für den Hauptspeicher
    - ***Dateiverwaltung*** (im Hintergrundspeicher)
    - ***Verwaltung*** der E/A-Geräte
- Einige bekannte Betriebssysteme
  - Windows NT, Windows 2000, Windows XP, Windows Vista
  - Apple OS X
  - LINUX
  - Unix-Varianten wie Solaris, HP/UX
  - MVS, VM/SP, CMS, BS 2000 (alle für Großrechner)

2.2 Prozesse

- Auf einem Rechner laufen „gleichzeitig“ verschiedene (Anwender- und System-) Programme
- ***Prozess*:**

  Ein Prozess ist ein in Ausführung befindliches Programm. Dies umfasst:
  - Befehlszähler (Programmzähler)
    - bestimmt den als nächstes auszuführenden Befehl
  - Programmtext
  - Eingabedaten
  - Organisatorische Daten



2.2 Prozesse

- Multiprogramming:
  - BS kann mehrere Prozesse gleichzeitig ausführen, d.h. auch ein Programm kann mehrmals gleichzeitig ausgeführt werden
  - Warum ist das sinnvoll?
    - Programme benötigen außer CPU meist auch E/A-Geräte
    - E/A-Geräte sind deutlich langsamer als CPU (Flaschenhals)

      $\Rightarrow$ Prozessor muss warten und ist nicht ausgelastet

      Beispiel:

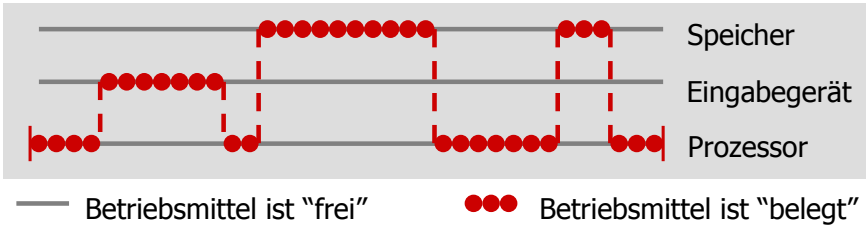
Lesen eines Datensatzes	0,0015 sec.
Ausführen von 100 Befehlen	0,0001 sec.
Schreiben des Datensatzes	0,0015 sec.
	0,0031 sec.

CPU-Auslastung: 0,0001/0,0031 ≈ 3,2 %

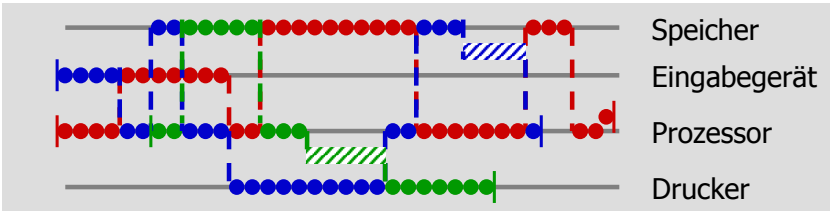
3,2 % der Zeit arbeitet CPU; 96,8 % der Zeit wartet CPU

2.2 Prozesse

- Lösung: Anwendungsprogramme sollen also dem Prozessor abwechselnd zugeteilt werden
  - Einbenutzerbetrieb (***Uniprogramming***) [1 Prozess]



  - Mehrbenutzerbetrieb (***Multiprogramming***) [hier: 3 Prozessen rot/grün/blau]



DATABASE SYSTEMS GROUP
Kapitel 2: Betriebssysteme
LMU

2.2 Prozesse

- Motivation (cont.)
  - Einzelne Prozesse durchlaufen damit ihre Anweisungsfolge (zumeist) nicht in einem Schritt sondern werden häufig unterbrochen
  - Es entsteht der Eindruck von ***Parallelität*** aller momentan existierenden Prozesse (dies ist allerdings nur eine ***Quasi-Parallelität*** !)
  - Folge:
    - zeitliche Dauer eines Prozesses kann bei verschiedenen Programmausführungen unterschiedlich sein
    - Insbesondere keine *a priori* Aussagen über den zeitlichen Ablauf eines Prozesses möglich
- Rolle des Betriebssystems
  - Aus Sicht des Anwendungsprogramms steht jedem Prozess ein eigener (virtueller) Rechner (CPU aber auch Hauptspeicher, etc.) exklusiv zur Verfügung
  - BS muss Abbildung auf den realen Rechner leisten

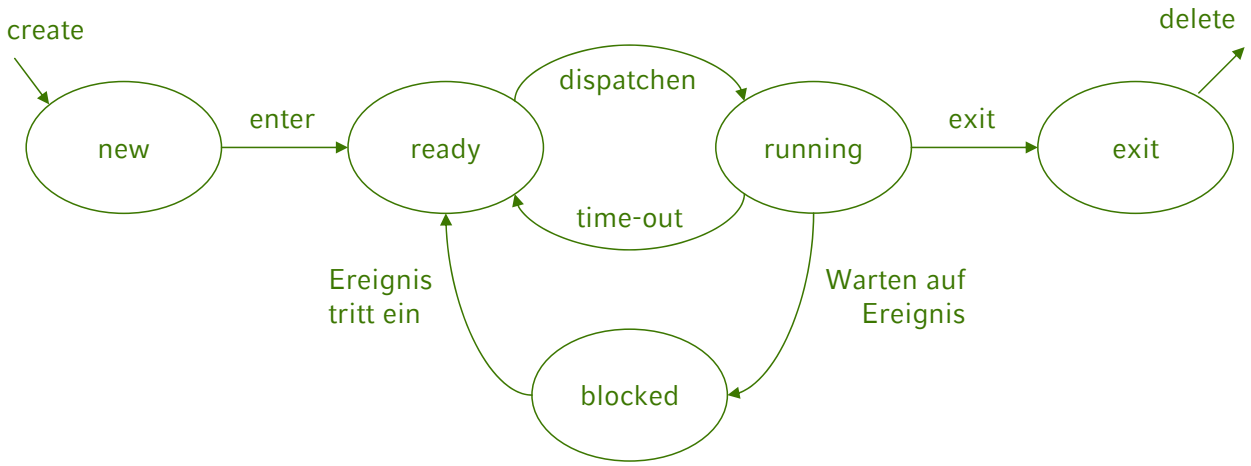
2.2 Prozesse

- Prozesszustände
  - 5-Zustands-Modell

    Prozess ist entweder
    - **new**: Prozess ist erzeugt, aber noch nicht zu der Menge der ausführbaren Prozesse hinzugefügt
    - **ready**: Prozess ist zur Ausführung bereit aber ein anderer Prozess ist dem Prozessor zugeteilt
    - **running**: Prozess ist dem Prozessor zugeteilt und wird gerade ausgeführt
    - **blocked**: Prozess ist blockiert, d.h. er wartet auf das Eintreten eines externen Ereignisses (Beendigung einer E/A-Operation, benötigtes Betriebsmittel ist belegt, ...)
    - **exit**: Prozess wurde beendet, d.h. Ausführung ist abgeschlossen

2.2 Prozesse

- Graphische Darstellung



2.2 Prozesse

– 7-Zustands-Modell
  - bisher: alle Prozesse werden im Hauptspeicher gehalten
  - wenn alle Prozesse blockiert sind (E/A-intensive Prozesse) und der Hauptspeicher voll ist, können keine weiteren Prozesse gestartet werden

    ⇒ Prozessor wäre wieder unbenutzt!!!
  - Lösung 1: Hauptspeicher erweitern

    ⇒ schlecht: kostet Geld, größere Programme?
  - Lösung 2: Prozesse (oder Teile davon) auf den Hintergrundspeicher (Festplatte, „HGS“) auslagern (***Swapping***)

    ⇒ gut: neue Prozesse haben im Hauptspeicher Platz

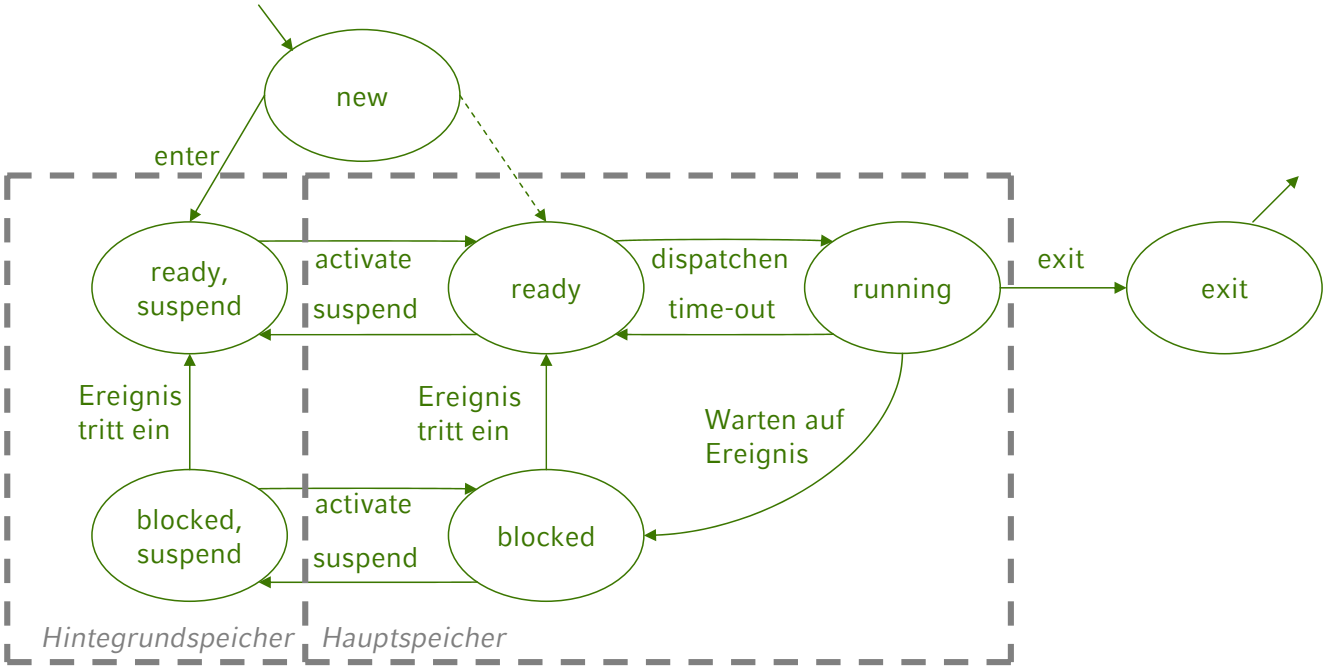
    ⇒ schlecht: eine zusätzliche E/A-Operation
  - Um übermäßig große Warteschlangen auf Festplatte zu vermeiden, sollte immer ein ausgelagerter Prozess wieder eingelagert werden; dieser sollte aber nicht mehr blockiert sein

    ⇒ unterscheide, ob ausgelagerte Prozesse blockiert oder bereit sind

    ⇒ 2 neue Zustände

2.2 Prozesse

- **ready, suspend**: auf dem HGS ausgelagert, bereit
- **blocked, suspend**: auf dem HGS ausgelagert, wartet auf Ereignis (blockiert)

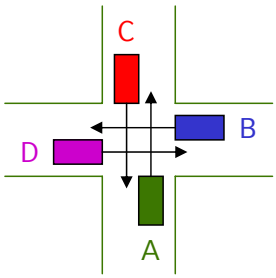


2.2 Prozesse

- Prozess-Scheduling
  - Scheduler:
    - („Faire“) Zuteilung eines Prozesses an den Prozessor
    - (Entscheidung über Swapping)
  - Scheduling-Verfahren
    - ***Round Robin*** (einfach und häufig verwendet)
      - Wartende Prozesse in einer (FIFO-) Warteschlange organisieren
      - Jeder Prozess hat für eine bestimmte Zeitspanne die CPU
      - Danach wird Prozess gegebenenfalls wieder in Warteschlange eingefügt
      - Zyklisches Abarbeiten der Warteschlange
    - ***Prioritäts-Scheduling***
      - Jedem Prozess ist eine Priorität zugeordnet
      - Prozess mit höchster Priorität bekommt den Prozessor zugeordnet
    - ***Shortest-Job-First***
      - Prozess mit der kürzesten Laufzeit

2.3 Prozessverwaltung

- Probleme der Parallelität beim Multiprogramming
  - Parallel ablaufende Prozesse können voneinander abhängig sein
  - Um fehlerhaftes Verhalten zu verhindern müssen die Prozesse geeignet koordiniert werden
    - Beispiel für fehlerhaftes Verhalten: Verklemmung (***Deadlock***)
      - Ein Deadlock ist die dauerhafte Blockierung einer Menge M von Prozessen, die eine Menge S gemeinsamer Systemressourcen nutzen oder miteinander kommunizieren (|M|>1, |S|>1)
      - Anschauliches Beispiel: 4 Fahrzeuge gleichzeitig an einer Rechts-vor-Links Kreuzung



A wartet auf B
B wartet auf C
C wartet auf D
D wartet auf A

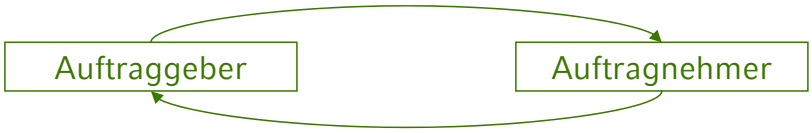
=> daher notwendig: ***Prozessverwaltung*** (***Prozess-Synchronisation***)

2.3 Prozessverwaltung

- Grundmuster der Prozess-Synchronisation
  - Prozesskooperation (hier nur kurz besprochen)
  - Wechselseitiger Ausschluss (hier etwas genauer besprochen)

- Prozesskooperation
  - Prozesse können zusammenarbeiten
  - Operationen eines Prozesses können voraussetzen, dass gewisse Operationen in anderen Prozessen erledigt sind
  - Grundmuster
    - Erzeuger-Verbraucher-Schema

    - Auftraggeber-Auftragnehmer-Schema



DATABASE SYSTEMS GROUP
Kapitel 2: Betriebssysteme

2.3 Prozessverwaltung

LMU

- Wechselseitiger Ausschluss
  - Parallel abgearbeitete Prozesse können sich gegenseitig beeinflussen
  - Beispiel: Flugbuchung
    - Prozedur (Algorithmus) zur Reservierung von n Plätzen

```
algorithmus Reservierung
    variables n, anzpl : Nat
    begin
            …
            while Buchung nicht abgeschlossen do {
                    n = Anzahl der zu buchenden Plätze;
                    anzpl = aktuelle Anzahl der freien Plätze;
                    if anzpl ≥ n
                    then anzpl um n verringern;
                    else STOP mit Auskunft „Ausgebucht“;
                    Reservierung bestätigen;
                    Frage nach weiterer Buchung;        // Nein: Verlassen der Schleife
            }
            …
end
```

2.3 Prozessverwaltung

- Variable anzpl gibt zu jedem Zeitpunkt die Anzahl noch verfügbarer Plätze an
- Zwei parallel ablaufende Prozesse A und B (Buchungen für jeweils 2 Plätze) führen den Algorithmus aus zum Zeitpunkt anzpl = 3
- Möglicher Ablauf:

Prozess A		Prozess B	Wert der Variablen anzpl
anzpl ≥ n (3 ≥ 2) ?			3
	Prozesswechsel →		
		anzpl ≥ n (3 ≥ 2) ?	3
		anzpl um n=2 verringern	1
		Reservierung bestätigen	1
	Prozesswechsel ←		
anzpl um n=2 verringern			-1
Reservierung bestätigen			-1

2.3 Prozessverwaltung

- Effekt: es wurden 4 Plätze vergeben, obwohl nur noch 3 Plätze frei waren
- Wie ist das zu verhindern?
  - Im Algorithmus Reservierung gibt es einen Bereich, der „kritisch“ ist

    if anzpl ≥ n
    then anzpl um n verringern
    else STOP mit Auskunft „Ausgebucht“
    Reservierung bestätigen

  - Gleichzeitige Ausführung dieses ***kritischen Bereiches*** eines Algorithmus durch mehrere verschiedene Prozesse muss verhindert werden
  - ***Wechselseitiger Ausschluss***:

    „Keine zwei Prozesse befinden sich gleichzeitig in ein und dem selben kritischen Bereich“

2.3 Prozessverwaltung

- Kritischer Bereich
  - Ein ***kritischer Bereich*** ist ein Programm(stück), das auf gemeinsam benutzte Ressourcen (globale Daten, Dateien, bestimmte E/A-Geräte, ...) zugreift oder den Zugriff darauf erfordert.
  - Die Ressource wird entsprechend ***kritische Ressource*** genannt

  ⇒ Aufteilung von Programmen in kritische und unkritische Bereiche

- Wechselseitiger Ausschluss
  - Solange ein Prozess sich in einem kritischen Bereich befindet, darf sich kein anderer Prozess in diesem kritischen Bereich befinden!

2.3 Prozessverwaltung

- Anforderungen an den wechselseitigen Ausschluss

  1. mutual exclusion

  zu jedem Zeitpunkt darf sich höchstens ein Prozess im kritischen Bereich befinden

  2. progress – no deadlock

  wechselseitiges Aufeinanderwarten muss verhindert werden

  Beispiel: Prozesse A und B; kritische Ressourcen a und b;

  A belegt a,

  B belegt b,

  B möchte a belegen => muss auf A warten

  A möchte b belegen => muss auf B warten

  => Deadlock !!!

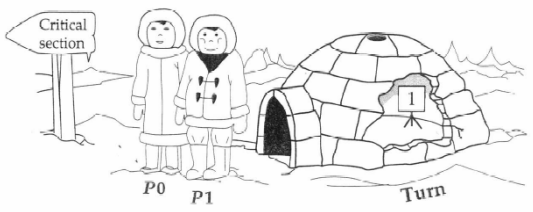
  3. bounded waiting – no starvation

  bei 3 Prozessen A, B, C könnten sich z.B. A und B in der Nutzung einer kritischen Ressource immer abwechseln

  => C müsste beliebig lange warten („***Verhungern***", ***starvation***) !!!

2.3 Prozessverwaltung

- Lösung nach Dekker (ca. 1965)
  - Prinzip:
    - globale, geschützte Variable `turn` zeigt an, welcher Prozess in kritischen Bereich eintreten darf
    - nur wenn Variable den Wert des entsprechenden Prozesses enthält, darf dieser Prozess in den kritischen Bereich
    - Nach Verlassen des kritischen Bereichs wird die Variable auf den Wert des anderen Prozesses gesetzt



    - Schema:

```
prozess P0                          prozess P1

…                                   …
while turn <> 0                     while turn <> 1
  do { nothing };                     do { nothing };
< kritischer Bereich >;             < kritischer Bereich >;
turn = 1;                           turn = 0;
…                                   …
```


2.3 Prozessverwaltung

  - Probleme:
    - Prozesse können nur abwechselnd in den kritischen Bereich eintreten. Auch bei Erweiterung auf mehrere Prozesse (dann würde `turn` Werte von 0 bis *n*-1 annehmen) wäre die Reihenfolge des Eintritts in den kritischen Bereich festgeschrieben.
    - Terminiert ein Prozess (im unkritischen Bereich), so kann der andere nur noch einmal in den kritischen Bereich eintreten. Bei allen weiteren Versuchen, in den kritischen Bereich einzutreten würde er dann blockiert werden

  => *progress*-Eigenschaft wird verletzt!

2.3 Prozessverwaltung

– Beispiel: Flugbuchung mit nur zwei Prozessen (d.h. es dürfen nur zwei Reisebüros weltweit gleichzeitig Flüge buchen)

Prozess 0	*Prozess 1*
…	…
while Buchung nicht abgeschlossen **do** {	**while** Buchung nicht abgeschlossen **do** {
n = Anzahl der zu buchenden Plätze;	n = Anzahl der zu buchenden Plätze;
anzpl = aktuelle Anzahl der freien Plätze;	anzpl = aktuelle Anzahl der freien Plätze;
while turn <> 0 **do** { nothing; }	**while** turn <> 1 **do** { nothing; }
if anzpl ≥ n	**if** anzpl ≥ n
then anzpl um n verringern;	**then** anzpl um n verringern;
else STOP mit Auskunft „Ausgebucht“;	**else** STOP mit Auskunft „Ausgebucht“;
Reservierung bestätigen;	Reservierung bestätigen;
turn = 1;	turn = 0;
Frage nach weiterer Buchung;	Frage nach weiterer Buchung;
}	}
…	…

- Problem: wenn z.B. Prozess 0 endet (keine weiteren Buchungen), kann Prozess 1 nur noch eine weitere Buchung machen

DATABASE SYSTEMS GROUP
Kapitel 2: Betriebssysteme

2.3 Prozessverwaltung

LMU

- HW-Lösung: Unterbrechungsvermeidung
  - bei Einprozessorsystemen können Prozesse nicht echt parallel ausgeführt werden
  - Prozesswechsel während des Aufenthalts im kritischen Bereich verursacht die Probleme
  - Idee: Unterbrechungen (Prozesswechsel) im kritischen Bereich ausschließen
  - Muster:

```
…
< unkritischer Bereich >;
ab hier: verbiete Unterbrechungen;
< kritischer Bereich >;
ab hier: erlaube Unterbrechungen;
< unkritischer Bereich >;
…
```

Funktioniert nicht bei Multiprozessorsystemen !!! (Warum?)

2.3 Prozessverwaltung

- Beispiel: Flugbuchung

```
algorithmus Reservierung
  variables n, anzpl : Nat
  begin
    ...
    while Buchung nicht abgeschlossen do {
      n = Anzahl der zu buchenden Plätze;
      anzpl = aktuelle Anzahl der freien Plätze;

      ab hier: verbiete Unterbrechungen;

      if anzpl ≥ n
      then anzpl um n verringern;
      else STOP mit Auskunft „Ausgebucht“;
      Reservierung bestätigen;

      ab hier: erlaube Unterbrechungen;

      Frage nach weiterer Buchung;
    }
    ...
end
```


2.3 Prozessverwaltung

- Semaphore
  - Semaphore sind spezielle Variablen, die Signale zwischen Prozessen übertragen
  - Ein Semaphor kann
    - Werte 0 oder 1 annehmen ⇒ binäres Semaphor
    - Beliebige **Int**-Werte annehmen ⇒ Zählsemaphor
  - Operationen auf Semaphor S:
    - `init`(S, Anfangswert) setzt S auf den Anfangswert
    - `wait`(S) versucht S zu dekrementieren; ein negativer Wert blockiert `wait`
    - `signal`(S) inkrementiert S; ggfs. wird dadurch die Blockierung von `wait` aufgehoben

Ausreichende Intuition: Operationen auf Semaphoren sind so etwas wie die „elementaren Verarbeitungsschritte“ (z.B. Multiplikation etc. auf natürlichen Zahlen) in Kapitel 1