www.uni-frankfurt.de

Modul: Programmierung B-PRG
Grundlagen der Programmierung 1

V24 Prozesse und Parallelarbeit

Prof. Dr. Detlef Krömker
Professur für Graphische Datenverarbeitung
Institut für Informatik
Fachbereich Informatik und Mathematik (12)

Rückblick: das Betriebssystem

- Realisiert eine abstrakte Schnittstelle zum Rechner
- Verwaltet Systemressourcen
- Es gibt verschiedene Arten von Betriebssystemen aufgrund verschiedener Anforderungen in unterschiedlichen Anwendungsgebieten
- Moderne Betriebssysteme sind Timesharing-Systeme mit Mehrprogrammbetrieb (plus zusätzliche Eigenschaften).
- Es gibt diverse Basis-Architekturen: Kern-Schale-Architektur, Hierarchische Schichten, Mikrokern (*micro kernel*), Virtuelle Maschinen.

Das Python Modul os versucht wiederum von verschiedenen Betriebssystemen zu abstrahieren.

Inhalt

- Was ist ein **Prozess**? Wie wird er erzeugt?
- Motivationen für Prozesse:
  - Es gibt naturgemäße Nebenläufigkeit!
  - **Verbesserung der CPU-Nutzung**
  - **Mehrkernprozessor** (***Multicore****-Prozessor)*
- Was sind **Threads**?

Was ist ein Prozess?

Zu einfache Antwort: **Prozess = Programm in Ausführung**

aber

Prozess, kann die Ausführung mehrerer Programme bedeuten
- ein Anwendungsprogramm ruft ein Betriebssystemprogramm auf
  - Systemaufruf (*system call*)
  - Programmunterbrechung (*trap, interrupt*)
- ein Prozess ist "Aktivitätsträger" von ggf. mehreren Programmen

Programm, kann von mehreren Prozessen ausgeführt werden
- nicht-sequentielles Programm, **im Falle von Uniprozessorsystemen**
  - präemptive (d.h. verdrängende) Programmverarbeitung
  - Aufgabe (engl. **task**), Faden (engl. **thread**)
- paralleles Programm **im Falle von Multiprozessorsystemen**

Prozess = Prozessinstanz ? … NEIN

Beachte: Analogie zu Typ oder Klasse einerseits und Instanz bzw. Objekt andererseits

- **Prozess** = ein abstraktes Gebilde
  - ein "Programm in Ausführung“, sequentieller Kontrollfluss
  - ein ”Ablauf“ , das heißt eine Verwaltungseinheit (?)
- **Prozessinstanz** (auch: Prozessinkarnation), ein konkretes Gebilde
  - die ”physische Instanz“ des abstrakten Gebildes ”Prozess“
    - an Betriebsmittel (Ressource; engl. resource) gebunden
    - die Identität (engl. identity) einer Programmausführung
  - die Verwaltungseinheit, die einen Prozess beschreibt und repräsentiert
    - ”dynamische Datenstruktur“ verschiedenartiger Strukturelemente

Synonyme Verwendung der Begriffe kann zu Missverständnissen führen.

Prozess = Programm? … NEIN

- **Programm ist statisch, Prozess ist dynamisch**
- Wissen über das gegenwärtig ausgeführte Programm sagt nicht viel aus über die zu dem Zeitpunkt im System stattfindende Aktivität.
  - Welches Zugriffsrecht besitzt das Programm zur Zeit?
  - auf ein Adressraumsegment, auf eine Datei, auf ein Gerät, . . .
  - allgemein: auf ein Betriebsmittel
  - Welcher Kontrollfluss ist im mehrädigen Programm zur Zeit aktiv?
  - Uni- vs. Multiprozessorsystem (SMP)
  - Wieviel Programmunterbrechungen sind zur Zeit gestapelt?
- Im Betriebssystemkontext ist das **Konzept ”Prozess“ daher nützlicher** als das Konzept ”Programm“, um Abläufe zu beschreiben und zu verwalten..

Prozessmodelle -- Gewichtsklassen

- **schwergewichtiger Prozess** (*heavyweight process*)
  - Prozessinstanz und Benutzeradressraum bilden eine Einheit
  - Prozesswechsel ; **zwei** Adressraumwechsel: AR x ⇒ BS ⇒ AR y
  - ”klassischer“ UNIX Prozess
- **leichtgewichtiger Prozess** (*lightweight process*)
  - Prozessinstanz und Adressraum sind voneinander entkoppelt
  - Prozesswechsel; **einen** Adressraumwechsel: AR x ⇒ BS ⇒ AR x
  - Das ist ein "Kernfaden" (engl. kernel thread): Thread auf Kernebene
- **federgewichtiger Prozess** (*featherweight process*)
  - Prozessinstanzen und Adressraum bilden eine Einheit
  - Prozesswechsel ; **kein** Adressraumwechsel: AR x ⇒ AR x
  - Benutzerfaden (engl. user thread): Faden auf Benutzerebene

Warum Prozesse

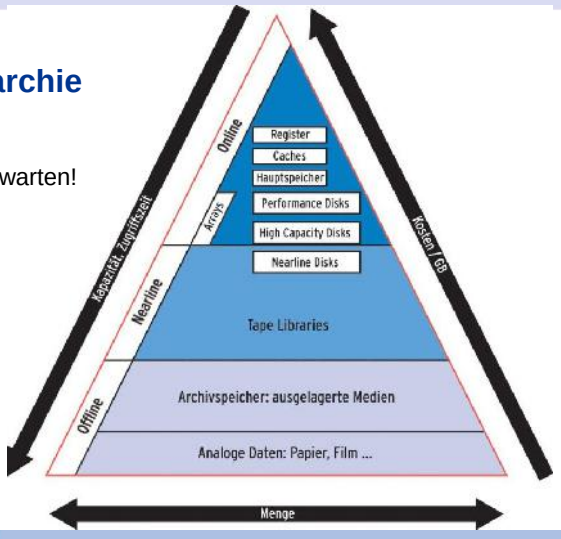
(1) Unterscheidung Programm vs. Prozess
(2) die **CPU-Ausnutzung** kann verbessert werden.

- Wenn ein Prozess einen Anteil p seiner Laufzeit auf die Beendigung von Ein-/Ausgaben wartet, so ist die Wahrscheinlichkeit, dass n solcher Prozesse auf die Ein-/Ausgabe **warten** = $\mathbf{p^n}$. Dies entspricht der Wahrscheinlichkeit, dass die CPU unbeschäftigt wäre.
- Die CPU-Ausnutzung kann dadurch als Funktion von n ausgedrückt werden, die Grad der Multiprogrammierung genannt wird:
  CPU-Ausnutzung = $1 - p^n$

Es ist durchaus üblich, dass ein **interaktiver Prozess 80 %** oder mehr im Ein-/Ausgabe-Wartezustand verbringt. Auch auf Servern, die viel Plattenein-/ausgabe durchführen, ist dieser Wert realistisch.

Die Speicherhierarchie

Es gibt viele Gründe zu warten!



Leider unfertig! Aber sichtbar: große Unterschiede

Speicher / Gerät	Typ. Größe bei Speichern	Zugriffszeit (Latenz)	Transferrate	
Prozessorregister	Byte - KiloByte	mit Prozessor-takt 1 ns	Prozessor-Rate 10.000 Mbytes/s	
Prozessorcache L1-L3	KiloByte-MegaByte	Je nach Level ca. 10-50 ns		
Arbeitsspeicher	GigaByte	ca. 100 ns		
SSD, Flash	TeraByte			
Festplatte (magnetisch)	TeraByte			
Netzwerk LAN WLAN	TeraByte	1 ms 30 ms	8-100 Mbyte/s 2-8 Mbyte/s	
DVD	100 GByte			
Drucker	6-100 Seiten a 4000 Zeichen/s	Sekunden	400 kByte/s	

Warum Prozesse?
2. Ziel: Mehrkernprozessoren …

… sind: **Mikroprozessoren** mit mehr als einem vollständigen Hauptprozessorkern in einem einzigen Chip.

- Sämtliche Ressourcen mit Ausnahme des Busses und eventuell einiger Caches sind mehrfach vorhanden.
- Es handelt sich also um **mehrere vollständige**, weitgehend voneinander unabhängige CPU-Kerne inklusive eigener Registersätze und arithmetisch-logischer Einheit (ALU).

Sinn und Zweck der Mehrkernprozessoren

- Bis zum Jahre 2005 dominierten die **Einzelkernprozessoren** den PC-Bereich. Im Vordergrund stand die Erhöhung der **Taktfrequenz**.
- **Problem:** ab Taktfrequenzen von etwa 4 GHz war die entstehende Abwärme nicht mehr sinnvoll handhabbar.
- **Weitere Leistungssteigerungen** (Moores Law: Verdoppelung alle 2 Jahre) durch die Einführung von Mehrkernprozessoren möglich.
- Heutzutage werden nur noch in wenigen Fällen Einzelkerne verbaut, da die entsprechenden Mehrkerne nur unwesentlich teurer sind.
- Es ist **kostengünstiger**, mehrere Kerne in einen Chip zu implementieren, als mehrere Prozessorsockel auf der Hauptplatine zu haben.

Wirkung von Mehrkernprozessoren

- Theoretisch kann eine vervielfachte Rechenleistung erzielt werden (das *n*-fache bei n Kernen).
- In der Realität kann diese Steigerung aber kaum erreicht werden.
- Die tatsächliche Leistungssteigerung hängt vor allem davon ab, wie gut die Software parallelisiert ist.
- Der Zugriff mehrerer aktiver Kerne auf den gemeinsamen Arbeitsspeicher kann zu Engpässen und Leistungsgrenzen führen, dagegen setzt man hochentwickelte Cache-Strategien ein (L1-, L2-, L3-Cache) .

Aktuelle Chips

Typisch aktuell **2-8 Kerne** (Dual-Core bis Octa-Core)

Aber auch sogenannte Manycore Architekturen:

- **AMD** Ryzen Threadripper 1950X: mit 16 Kernen und 32 Threads, seit August 2017 erhältlich
- **Intel** Core™ X Prozessoren: 18 Kerne und 36 Threads
- **Achtung:** Die o.g. Threads sind sogenannte **Hyper-Threads**.
  Idee: Um die Rechenwerke eines Prozessors besser auszulasten, füllt man die Lücken in der Pipeline mit Befehlen eines anderen Threads. Wir erreichen aber keine Verdoppelung der Leistung bei 2 Threads (gegenseitige Behinderung) sondern laut Intel + 30%.
- Wir kommen darauf zurück, aber: das ist dann doch zu viel Hardware.

Unterstützung durch Betriebssysteme

- Unix, der Linux-Kernel und Microsoft Windows ab XP **unterstützen Mehrkernprozessoren.**
- Windows NT erkennt einen Mehrkernprozessor **als mehrere Einzelkernprozessoren** (dadurch sind zwar alle Kerne nutzbar, spezielle Mehrkernprozessoroptimierungen können aber nicht greifen).
- Das Betriebssystem verteilt Prozesse und Anwendungen auf die einzelnen Prozessoren, die diese dann unabhängig parallel ausführen.
- Wird hingegen nur eine Anwendung ausgeführt, **so muss diese für die mehreren Prozessoren parallelisiert werden.**
- **Prozesse** sind die Entitäten, die vom Betriebssystem auf die Kerne verteilt werden können..

Programme und Prozesse

- **Warum Mehrprozessbetrieb**?
- **Antwort: Effiziente Nutzung** des Systems
- ❶ **Mehrprogrammbetrieb:** mehrere Teilnehmer am Rechner bzw. Server-Betrieb im Netz
- ❷ **Parallelbetrieb:** unterschiedliche CPU-Nutzung parallel ausführbarer Prozesse eines Programms

Parallele Programmierung …

ist ein Programmierparadigma. Dazu gehören:

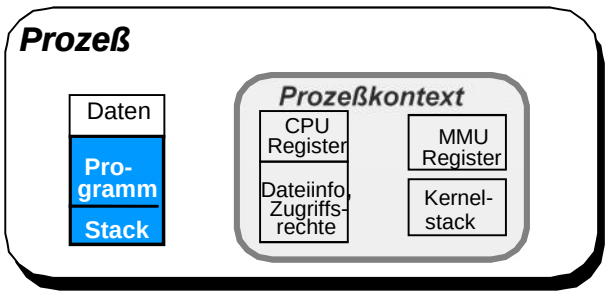
- Methoden, ein Programm in einzelne Teilstücke aufzuteilen, die **nebenläufig** ausgeführt werden können,
- Methoden, nebenläufige Programmabschnitte zu synchronisieren.

Dies steht im Gegensatz zur klassischen, **sequentiellen** (oder seriellen) Programmierung. Man erreicht hiermit:

- schnellerer Programmausführung (bspw. bei Nutzung mehrerer Prozessorkerne)
- die Möglichkeit, **von Natur her parallel bearbeitbare Aufgaben** direkt in Programmen abzubilden. Führt (hoffentlich) zu einfacherem, verständlicherem Quelltext.

Was sind Prozesse ?

Prozess = **Programmdaten** + **Prozeßkontext**



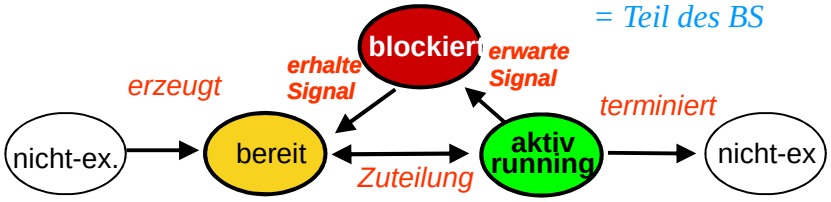
Prozesstabelle

- Das Betriebssystem führt eine Tabelle mit aktuellen Prozessen in einer kerneleigenen Datenstruktur, der sog. Prozesstabelle: Prozesskontext
- Bei der Erzeugung eines neuen Prozesses wird darin ein **Prozesskontrollblock (PCB)** als neuer Eintrag angelegt.
- In der Regel wird über den PCB auf den Prozess zugegriffen.
- Da PCBs Informationen über die belegten Ressourcen enthalten, müssen sie im Speicher direkt zugreifbar sein. Wesentliches Kriterium, in was für einer Datenstruktur die Tabelle im Betriebssystem gespeichert wird, ist daher ein möglichst effizienter Zugriff.

Prozeßzustände

Dispatcheraktionen

= Teil des BS

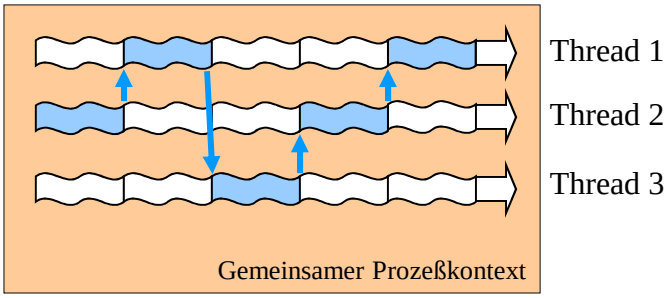


Prozesse warten in einer (Prioritäts-)Warteschlange ...

- auf den Prozessor (*bereit*)
- auf Daten des I/O-Geräts (*blockiert*)
- auf eine Nachricht (*blockiert*)
- auf ein Zeitsignal (*blockiert*)

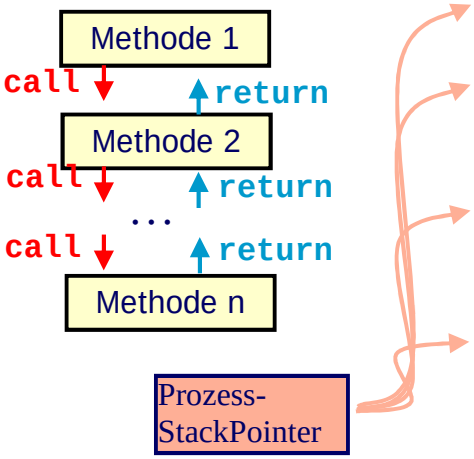
Threads ("Fäden", Coroutinen)

- asynchroner, paralleler, unterschiedlicher Programmverlauf (eigenen *Stack*)
- **gemeinsamer** Prozeßkontext
  (Speicher-Addressbereich, Dateien (*file handles*), Ressourcen)



Nebenläufigkeit bei Methoden ?

- Methoden

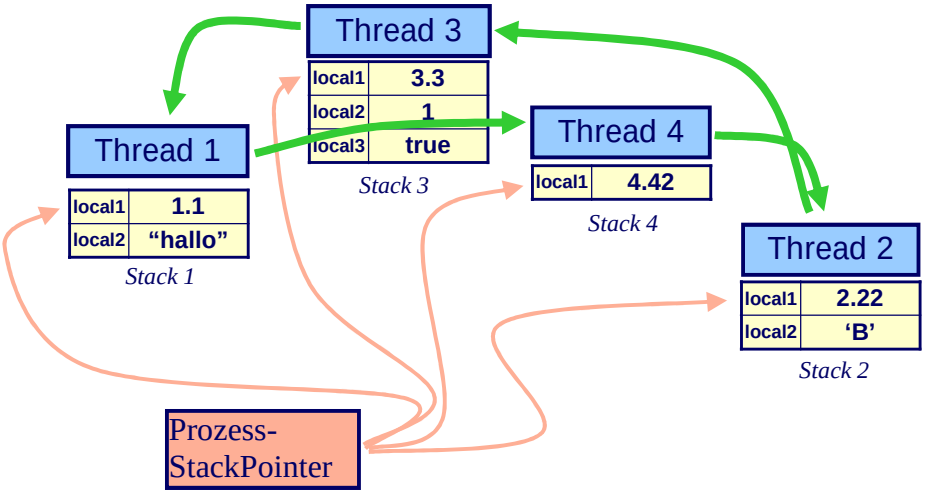


gemeinsamer Stack:
temp., lokale Variable

local1	1.1
local2	'B'

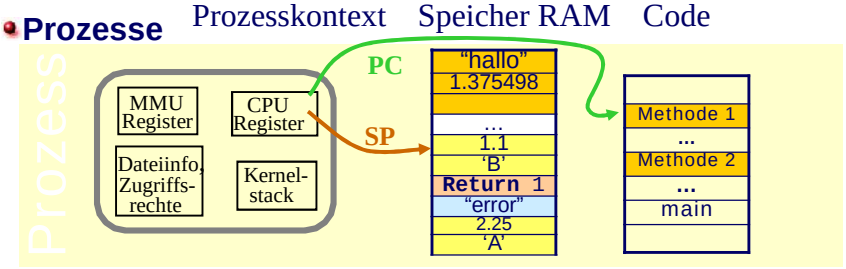
Nebenläufigkeit

- Coroutinen und Threads

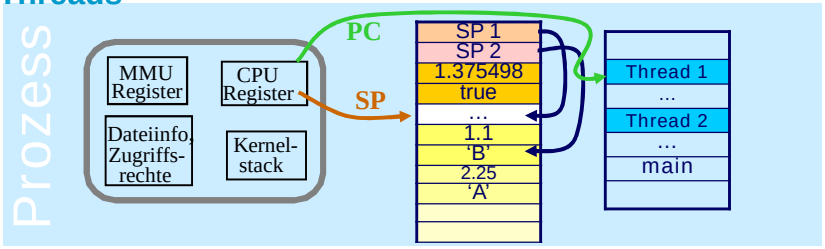


Nebenläufigkeit

- **Prozesse**



- **Threads**



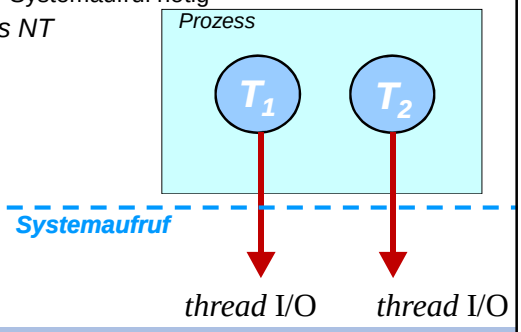
Thread- Typen: *heavyweight threads = lightweight process*

kontrolliert vom Betriebssystem *(z.B. Windows NT)*

Vorteil: Unabhängiger I/O aller threads T_1, T_2

Nachteil: langsamer BS-Systemaufruf nötig

⇒ *„fibers“ in Windows NT*

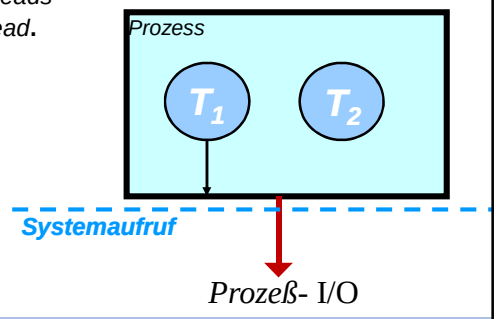


Thread- Typen: *lightweight threads = featherweight process)*

kontrolliert vom Benutzerprogramm *(z.B. Unix-Bibliothek)*

Vorteil: sehr schneller *thread*-Wechsel T_1, T_2

Nachteil: Blockieren aller *threads* bei I/O-Warten von einem *thread*.



Fragen ?

und (hoffentlich) Antworten

Ausblick

- Wann nimmt man Threads, wann Prozesse?
- *Race conditions* und **Semaphore**

Multiprogramming und Threading in Python

Interrupts und Exceptions

und, herzlichen Dank für Ihre Aufmerksamkeit!