Fachhochschule Reutlingen

Studiengang

Computer Based Engineering

Lehrveranstaltung

Betriebssystemarchitekuren

Nebenläufige Prozesse

Karlheinz Hug

12. April 2002

1 Der Prozessbegriff

Sequenzielles Programm :=

maschinell ausführbarer Algorithmus mit kausal bzw. zeitlich aufeinander folgenden Schritten.

Prozessor :=

aktives Organ, das ein Programm ausführen kann.

Umgebung :=

Menge der Dinge, die bei einer Programmausführung manipuliert werden können, d.h. Register des Prozessors, Daten im Hauptspeicher, Betriebsmittel wie Drucker, Dateien usw., das Programm selbst.

Prozess :=

Ausführung eines sequenziellen Programms auf einem Prozessor in einer Umgebung.

Prozesszustand :=

Schnappschuss seiner Umgebung zu einem Zeitpunkt.

1.1 Prozess vs. Thread

schwergewichtiger Prozess :=
Prozess mit eigenem Adressraum.

leichtgewichtiger Prozess, Thread, Strang :=
Prozess ohne eigenen Adressraum.

- Mehrere Threads laufen in gemeinsamen Adressraum (eines Prozesses).

- Threads:

  - kosten weniger als Prozesse;

  - vom BS oder von Programmiersprache bereitgestellt;

  - wichtig bei Multimediaanwendungen.

1.2 Sequenziell vs. nebenläufig

Voraussetzungen bei einzelnem sequenziellen Programm

- Anweisungen unteilbar ausgeführt;
- Umgebung nur durch Programmausführung selbst verändert;
- Ergebnisse der Programmausführung zeitunabhängig, beliebig reproduzierbar.

Folgerung

Sequenzielles Programm =
geschlossenes System ohne externen Einfluss.

Bei einem nebenläufigen System gelten diese Voraussetzungen nicht!

2 Nebenläufige Systeme

Nebenläufiges System :=

System mit mehreren gleichzeitigen, ineinander verschränkten sequenziellen Abläufen.

Prozesse = Einheiten der sequenziellen Abläufe.

System mit **nebenläufigen** (parallelen) **Prozessen** (*concurrent processes*, *tasks*, *threads*).

Mehrprozesssystem, **-betrieb**

(*multiprocessing*, *multitasking*).

2.1 Programm und Prozess

Programm = statische Strukturierungseinheit.

Prozess = dynamische Strukturierungseinheit.

- In einem Prozess können nacheinander mehrere Programme ablaufen.

- Mehrere Prozesse können unabhängig voneinander dasselbe Programm ausführen (z.B. Editor, Übersetzer).

  Code sharing: Code eines mehrfach ausgeführten Programms ist nur einmal im Speicher.

  Voraussetzung: Code ist **eintrittsinvariant** (*reentrant*), d.h. getrennt in

  - invarianten Codeteil aus sich nicht modifizierenden Befehlen & Konstanten;

  - varianten Datenteil.

2.2 Anfang und Ende von Prozessen

- Prozesse können
  - **statisch** im Programm festgelegt sein (Echtzeitsysteme, OSEK);
  - beim Systemstart oder
  - während des Betriebs zu beliebigen Zeitpunkten **dynamisch** erzeugt werden.

    BS-Operationen:
    - Unix: `fork`
    - NT: `CreateProcess`
- Prozesse können
  - freiwillig **terminieren** (Benutzerprozess);
  - von anderen Prozessen **abgebrochen** werden;
  - **ewig laufen** (Systemprozess).

2.3 Abhängigkeiten zwischen Prozessen

- Prozess ≠ geschlossenes System.
- Ein Programm kann in einem Ablauf andere Prozesse erzeugen & kontrollieren.
- Prozesse können sich synchronisieren & miteinander kommunizieren.
- Ein Prozess kann beliebig gestoppt & fortgesetzt werden;

  Prozesszustand muss erhalten bleiben.

2.4 Koordination von Prozessen

Synchronisation :=

Prozesse steuern ihren (kausalen, zeitlichen) Ablauf relativ zueinander durch Austausch von Signalen.

Kommunikation :=

Prozesse tauschen untereinander Botschaften (nicht nur Signale) aus.

Koordination :=

Synchronisation & Kommunikation.

Verschiedene Formen der Koordination

Kooperation :=

Mehrere Prozesse arbeiten gemeinsam an einer Aufgabe.

Konkurrenz :=

Mehrere (nicht notwendig kooperierende) Prozesse bewerben sich um gemeinsam benutzbare, rare Betriebsmittel.

2.5 Konzept zur Implementation

Prozessor-Multiplexing :=

Prozessor wird nach vorgegebener Strategie zwischen ablaufbereiten Prozessen umgeschaltet;

ermöglicht Mehrprozessbetrieb
auf Einprozessormaschine.

Virtueller Prozessor, Pseudoprozessor :=

sw-mäßig realisierte Kopie eines realen Prozessors;

Abstraktion des Begriffs „Prozessor“.

- Jedem Prozess ist sein virtueller Prozessor zugeordnet, auf dem er läuft.

- Prozess & virtueller Prozessor: 2 Sichtweisen.

  - Prozess =
    Ablauf eines Programms.

  - Virtueller Prozessor =
    Akteur,
    Verwaltungseinheit der BS-Komponente „Prozessverwaltung“.

3 Kooperationsmodelle

3.1 Peer-Modell

Peer-Prozesse holen Aufträge aus gemeinsamer Warteschlange & bearbeiten diese.

3.2 Boss-/Worker-Modell

Boss-Prozess nimmt Aufträge entgegen & verteilt Arbeit an **Worker-Prozesse**.

3.3 Pipeline-Modell

Aufträge sind in nacheinander zu bearbeitende Teilaufträge zerlegt.

Jeder Prozess bearbeitet einen Teilauftrag.

Ausgabe eines Prozesses ist Eingabe für nächsten Prozess in Teilauftragskette.

4 Kommunikationsmodelle

4.1 Kommunikation über gemeinsame Daten

Mit gemeinsamen Speicher.

Data sharing: Prozesse kommunizieren über gemeinsame Daten.

Prozessglobale Daten :=
Daten, auf die mehrere Prozesse zugreifen können.

Prozesslokale Daten :=
Daten, die nur zu einem Prozess gehören.

4.2 Kommunikation über Nachrichten

Ohne (oder mit) gemeinsamen Speicher.

Prozesse laufen auf

- demselben Rechner oder
- **verteilt** auf verschiedenen Rechnern.

5 Allgemeine Koordinationsprobleme

Aushungerung - Fairness

starvation, livelock

Verklemmungen - Verklemmungsfreiheit

deadlock

Prioritätsinversion - Vermeiden von P.

priority inversion

6 Synchronisation

6.1 Synchronisationsbedingungen

Einfacher & gegenseitiger Ausschluss

Reihenfolgebeziehungen

Warten auf das Eintreten von Ereignissen

6.2 Synchronisationsprobleme

Betriebsmittelverwaltungsprobleme

Erzeuger-/Verbraucher-Probleme

Leser-/Schreiber-Probleme

7 Beispiel: Ein Ausschluss- & Betriebsmittelverwaltungsproblem

Gegeben

Parkhaus, mehrere Zufahrten, Kapazität 100 Pkw.

Gesucht

Einfahrtregelung so, dass Kapazität nicht überschritten.

Lösung

Zähler für Pkw im Parkhaus;

bei jeder Zufahrt ein „Aufpasser“, der Zähler aktualisiert.

Bild

Siehe Tafel.

Lösungsansatz

```
-- process global declarations:
feature {watchdog}
  capacity : INTEGER is const 100
  counter  : INTEGER is init 0

-- process type with n instances:
process
  watchdog (entrance : 1..n) is
do
  loop
    if car enters car park then
      counter := counter + 1
    elseif car leaves car park then
      counter := counter - 1
    end -- if
  end -- loop
end -- watchdog
```

Probleme

(1) Der Überlastschutz fehlt noch.

(2) **Aktives Warten** (*busy waiting*) ist ineffizient!

(3) Was passiert beim Zugriff auf den Zähler?

8 Kritische Abschnitte

Kritischer Abschnitt (*critical section*, kA) :=

Codestück, in dem auf prozessglobale Daten oder Betriebsmittel zugegriffen wird.

- Zu einer Menge von Daten kann es mehrere kAe geben, die auf diese Daten zugreifen.

- Solche kAe dürfen nicht in beliebiger zeitlicher Überlappung, sondern nur unter gewissen Synchronisationsbedingungen ausgeführt werden.

Synchronisationsbedingung :=

Bedingung für die Ausführung einer Menge kAe als Ganzes.

- Eigentlich sind nicht die Codeabschnitte, sondern die Daten kritisch: **kritische Daten**.

- Problem ist, die **Konsistenz** der Daten unter nebenläufigen Zugriffen zu erhalten.

8.1 Unteilbarkeit

Eine Menge von kAen heißt **unteilbar** (atomar, *indivisible*, *atomic*), wenn es zu jeder nebenläufigen Ausführung von kAen dieser Menge eine sequenzielle Ausführung dieser kAe gibt, die dieselbe Wirkung wie die nebenläufige Ausführung hervorbringt.

- Unteilbare kAe stören sich nicht gegenseitig, auch wenn sie von mehreren Prozessen nebenläufig ausgeführt werden.

- Ein einzelner Speicherzugriff ist unteilbar.

- Ein normaler Maschinenbefehl ist ununterbrechbar, aber nicht notwendig unteilbar. (Er kann mehrere Speicherzugriffe hervorrufen.)

9 Gegenseitiger Ausschluss

Zustände eines kritischen Abschnitts

- Nicht in Ausführung, nicht aktiviert.
- Aufgerufen, aber noch nicht betreten, noch nicht aktiviert.
- In Ausführung, nach Aufruf betreten, aktiviert.
- Möglich: einfach oder mehrfach aktiviert.

Ausschluss

Seien *A*, *B* kAe.
A **schließt** (*excludes*) *B* **aus** :⇔
wenn sich *A* in Ausführung befindet, darf *B* nicht betreten werden.

Ausschlussgraphen

Siehe Tafel.

Gegenseitiger Ausschluss

Eine Menge kAe steht unter der Synchronisationsbedingung **gegenseitiger Ausschluss** (*mutual exclusion*), wenn sich zu jedem Zeitpunkt höchstens einer der kAe in einfacher Ausführung befinden darf.

- KAe unter gegenseitigem Ausschluss dürfen/können/werden nebenläufig aufgerufen, aber nicht nebenläufig ausgeführt.

- KAe unter gegenseitigem Ausschluss sind unteilbar.

Fragen

- Wie erreicht man gegenseitigen Ausschluss von kAen?
  - Mit geeigneten Operationen.
  - Diese Operationen sind selbst wieder kAe!

- Wie erreicht man Unteilbarkeit von Operationen zur Implementation kAe?

9.1 Implementationen

Klassische Implementationen sind prozedural, folgende objektorientiert.

- Es gibt eine Klasse mit einem Paar Operationen, genannt **Synchronisationsprimitive**.

- Zu jeder Menge kAe, die sich gegenseitig ausschließen sollen, wird ein Objekt dieser Klasse vereinbart.

- Jeder kA wird mit Aufrufen der Synchronisationsprimitive geklammert.

```
class LOCK
feature
  enter
     -- Eintrittsprotokoll,
     -- stellt den Ausschluss her.
  exit
     -- Austrittsprotokoll,
     -- hebt den Ausschluss auf.
end -- LOCK
```

Ungeschützte Situation

critical data D

process P:
    critical section A accessing D

process Q:
    critical section B accessing D

Geschützte Situation

```
mutex : LOCK is init ...
```
critical data D

```
mutex.enter
```
critical section A accessing D
```
mutex.exit
```

```
mutex.enter
```
critical section B accessing D
```
mutex.exit
```

Implementationsmittel

- **Unterbrechungssperren**

  Unterbrechungen während der Ausführung eines kAs verbieten.

  - Erfordert Aufruf einer BS-Operation (Trap-Befehl!).
  - Nur bei Einprozessorsystemen geeignet.
  - Reaktionsfähigkeit auf externe Unterbrechungen herabgesetzt, insbes. bei langen kAen.
  - Nur für kurze kAe geeignet.
  - Ununterbrechbarkeit garantiert i.a. nicht Unteilbarkeit.
  - Ein unteilbarer kA muss nicht notwendig ununterbrechbar sein.

- **Teste-und-Setze-Befehl**

  Spezieller unteilbarer Maschinenbefehl.

  - Aktives Warten.
  - Nur für kurze kAe geeignet.
  - Verklemmungsgefahr bei Einprozessorsystemen mit Prioritäten.
  - Nicht fair, nicht aushungerungsfrei.

- **Unterbrechungssperren & Teste-und-Setze-Befehl**

10 Höhere Synchronisationsmechanismen

Semaphore

Dijkstra, Ende 60er Jahre

Bedingungen

Bedingte kritische Abschnitte

Monitore

Hoare, 1974
Concurrent Pascal, Mesa, Modula-2, Java

Pfadausdrücke

11 Semaphore

Ein **Semaphor** (*semaphore*) ist ein Synchronisationsmechanismus mit zwei Operationen,

- `P` (von holländisch „passerem“) und
- `V` (von holländisch „vrygeven“).

Intern gehören zu einem Semaphor

- ein Zähler und
- ein Wartezustand für Prozesse.

Wir nennen `P wait` und `V signal` & formulieren ein Semaphor als Klasse in objektorientierter Notation.

- `wait` und `signal` sind unteilbare Operationen! Darstellung durch **`atomic`**.

- Bedeutung des Zählers `count:`

  `count > 0` : Anzahl verfügbarer Betriebsmittel.

  `count < 0` : Anzahl wartender Prozesse.

  `count = 0` : kein Betriebsmittel verfügbar, kein Prozess wartet.

- Der Initialisierungswert für `count` ist ≥ 0.

```
class SEMAPHORE

feature {NONE}
  count : INTEGER

feature
  wait is atomic
  do
    count := count - 1
    if count < 0 then
      keep waiting until signalled
    end
  end -- wait

  signal is atomic
  do
    count := count + 1
    if count <= 0 then
      let some waiting process continue
    end
  end -- signal

end -- SEMAPHORE
```

11.1 Gegenseitiger Ausschluss

Die Betriebsmittel sind die kAe, die unter gegenseitigem Ausschluss stehen. Zu jedem Zeitpunkt ist höchstens ein kA verfügbar.

Wir brauchen ein Semaphor, dessen Zähler mit 1 initialisiert wird.

```
mutex : SEMAPHORE
        is init (count is 1)
```

Das Eintrittsprotokoll für kAe ist ein `wait`-Aufruf des Semaphors,

das Austrittsprotokoll ein `signal`-Aufruf.

```
mutex.wait
critical section
mutex.signal
```

12 Erzeuger-/Verbraucher-Probleme

Aufgabe

Es gibt

- zwei Arten von Prozessen: **Erzeuger**- & **Verbraucherprozesse**;
- eine gemeinsam benutzte Datenstruktur: ein **Puffer** für Datenelemente.

Es sollen

- Erzeuger Datenelemente erzeugen & im Puffer ablegen,
- Verbraucher Datenelemente dem Puffer entnehmen & verbrauchen.

Bedingungen

- Jedes erzeugte Datenelement wird genau einmal verbraucht.

- Erzeuger & Verbraucher dürfen nur unter gegenseitigem Ausschluss auf ein einzelnes Pufferelement zugreifen: **Ausschlussproblem**.

- Nur Elemente, die erzeugt wurden, können verbraucht werden: **Reihenfolgeproblem**. Wenn der Puffer leer ist, müssen Verbraucher warten.

- Wenn der Puffer voll ist, können keine erzeugten Elemente mehr deponiert werden. Erzeuger müssen warten, bis wieder Elemente verbraucht wurden: **Reihenfolgeproblem**.

Bemerkungen

- Strategien, die obige Bedingungen erfüllen:
  - Last-In-First-Out (LIFO).
  - First-In-First-Out (FIFO).
- Je nach Problem gibt es ein oder mehrere Exemplare jedes Prozesstyps.

Allgemeine Vereinbarungen

- Der Elementtyp ist beliebig.

```
class ITEM
  ...
end -- ITEM
```

- Wir realisieren den synchronisierten Puffer sinnvollerweise als Modul (oder Klasse).

```
module buffer
feature
  capacity : INTEGER is const ...

feature {producer}
  deposit (item : ITEM)
    -- Puts item into the buffer.

feature {consumer}
  remove (item : ITEM)
    -- Delivers an element stored in the buffer
    -- as item and deletes it from the buffer.
end -- buffer
```

- Erzeuger & Verbraucher sind nach folgendem Muster gestrickt, oft als zyklische Prozesse.

```
process producer (...) is
local
  item : ITEM
do
  ...
  produce item
  buffer.deposit (item)
  ...
end -- producer

process consumer (...) is
local
  item : ITEM
do
  ...
  buffer.remove (item)
  consume item
  ...
end -- consumer
```

Lösung mit Mehrfachpuffer

```
module buffer
feature {NONE}
  capacity : INTEGER is const ...
  store    : ARRAY [ITEM] is
             sized 0..capacity-1
  nextfree : INTEGER is init 0
  nextfull : INTEGER is init 0
  mutex : SEMAPHORE is
          init (count is 1)
  full  : SEMAPHORE is
          init (count is 0)
  free  : SEMAPHORE is
           init (count is capacity)
```

```
feature {producer}
  deposit (item : ITEM) is
  do
    free.wait
    mutex.wait
    store.put (item, nextfree)
    nextfree :=
      (nextfree + 1) mod capacity
    mutex.signal
    full.signal
  end -- deposit
```

```
feature {consumer}
  remove (item : ITEM) is
  do
    full.wait
    mutex.wait
    item.copy
      (store.item (nextfull))
    nextfull :=
      (nextfull + 1) mod capacity
    mutex.signal
    free.signal
  end -- remove
end -- buffer
```

- Garantiert die Lösung gegenseitigen Ausschluss?

- Ist die Lösung verklemmungsfrei?

13 Kommunikationsmechanismen

Synchroner Nachrichtenaustausch

Asynchroner Nachrichtenaustausch

Kommunikationskanäle, Pipes

Message Queues

Briefkästen

Ports

Verbindungen

Rendezvous

Ferne Prozeduraufrufe

14 Prozessverwaltung

process management

14.1 Aufgaben

- Virtualisierung des Betriebsmittels Prozessor.
- Realisierung nebenläufiger Prozesse.
- Bereitstellung von Mechanismen zur Synchronisation und Kommunikation von Prozessen.
- Ablaufsteuerung.
- Behandlung von Unterbrechungen.

14.2 Das Prozesszustandsmodell

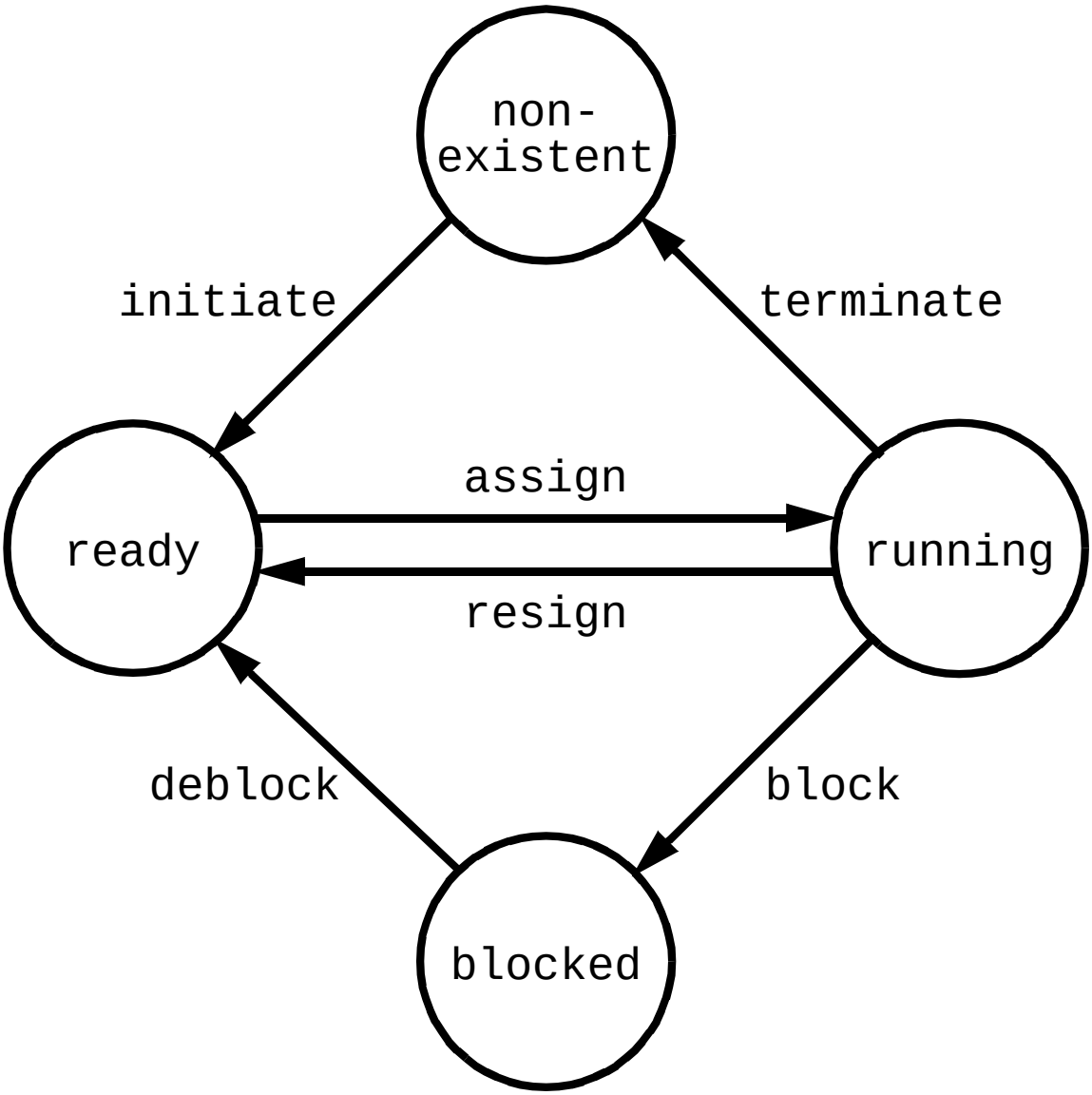
Zustände

- running
- ready
- blocked
- nonexistent

Übergangsoperationen

- initiate (context)
- assign
- resign
- block (queue)
- deblock (queue)
- terminate

Prozesszustandsübergangsdiagramm



14.3 Prozessleitblock

process control block

- Identifikation
- Zustand
- Registerinhalte (Kontext)
- Kernkeller

- Scheduling:
  - Art
  - Prioritäten

- Betriebsmittel:
  - Bedarf, Verbrauch, Abrechnung (*accounting*)
  - Speicher, Dateien

- Synchronisation: Signale
- Prozessgruppen, Threads

- Verwaltungsdaten

15 Ablaufsteuerung

scheduling

- Zwei Begriffe: Prozess - Auftrag.
- **Auftrag**: von Benutzer initiiert.
- **Prozess**: Ablaufeinheit des BSs.
- Ein Auftrag muss nicht unmittelbar zu einem Prozess führen.
- Zu einem Auftrag kann es mehrere Prozesse geben.
- **Benutzerprozess** resultiert aus Benutzerauftrag.
- **Systemprozess** führt Aufgabe des BSs aus.

15.1 Ebenen der Ablaufsteuerung

- **Prozesssteuerung** (*process scheduling*)
  Kurzzeitablaufsteuerung.

- **Auftragssteuerung** (*job scheduling*)
  Langzeitablaufsteuerung.

- **Laststeuerung**.

15.2 Zeitscheibenverfahren

- Hardware-Unterstützung:
  Je Prozessor ein Zeitgeber `timer`:
  - wird mit `timer.load (time)` geladen;
  - fordert nach `time` Zeiteinheiten eine Zeitgeberunterbrechung an.

- Unterbrechungsannahme =
  erzwungener Prozeduraufruf;
  Aufruf der Unterbrechungsbehandlungsroutine.

15.3 Prioritäten

- Externe Auftragspriorität.

- Interne Prozesspriorität.

Einflüsse auf Prozesspriorität

- Statische Eigenschaften:

  - Externe Priorität.
  - Geschätzte Prozessorzeit.
  - Geforderte E/A-Benutzung.
  - Geforderter Speicher.

- Dynamische Eigenschaften:

  - Momentane Benutzung von Betriebsmitteln.
  - Vorherige Wartezeit.
  - Verbrauchte Prozessorzeit.
  - Vorherige E/A-Aktivität.
  - Gesamtzeit im System.

15.3.1 Prozessprioritätsklassen in Unix

- **Time-Sharing**.
  Dynamisches Mehrstufenverfahren
  (*multilevel feedback*).

- **Real-Time**.
  Feste Prioritäten.

- **System**.
  Feste Prioritäten,
  nicht konfigurierbar.

15.4 Dispatcher-Strategien

- Welcher bereite Prozess soll als nächster laufen?
- Effizienz!
- Prozesswechsel: Häufigkeit im ms-Bereich, ca. 5000 Befehle!

Klassifizierungsmerkmale

- Prioritätssteuerung.
- Zeitscheibenverfahren (*time slicing*).
- Verdrängung (*preemption*).

Strategien

- First-In-First-Out (FIFO),
  First-Come-First-Served (FCFS).
- Round Robin (RR).
- Highest-Priority-First (HPF).
- Shortest-Processing-Time-First (SPTF).
- Shortest-Remaining-Time-First (SRTF).
- Kombination der Strategien,
- z.B. dynamisches Mehrstufenverfahren.

16 Ein-/Ausgabe-Steuerung

- Geschwindigkeitsunterschied zwischen Prozessoren und E/A-Geräten: Faktor 10^3 - 10^5.
- HW: Asynchrone E/A mit Unterbrechungen.
- BS: Synchrone oder asynchrone E/A.

16.1 Synchrone Ein-/Ausgabe

Schnittstellenoperation für Prozess

`io_transfer` (*Parameter für E/A-Auftrag*)

E/A starten
blockieren

Unterbrechungsbehandlungsroutine für E/A-Gerät

`io_interrupt_handler`

wartenden Prozess deblockieren

16.2 Asynchrone Ein-/Ausgabe

Schnittstellenoperationen für Prozess

`io_start` (*Parameter für E/A-Auftrag*)

E/A starten (ohne blockieren)

`io_wait`

if *Unterbrechung noch nicht angekommen*
then
blockieren
end

Unterbrechungsbehandl.routine für E/A-Gerät

`io_interrupt_handler`

if *Prozess wartet schon* **then**
wartenden Prozess deblockieren
else
Unterbrechung notieren
end

17 Prozesse in Unix

- Bezüglich Beziehung „wer erzeugt wen“ ist die Menge der Prozesse baumförmig strukturiert: **Prozessbaum**.

- Knoten = Prozess.

- Kante = Erzeugungsrelation.

- Oberer Knoten = **Vaterprozess**, hat andere Prozesse (= untere Knoten) erzeugt.

- Unterer Knoten = **Kindprozess**, wurde von Vaterprozess (= oberem Knoten) erzeugt.

- Wurzel = **Urprozess**, wird beim Systemstart erzeugt, führt Programm `init` aus & existiert bis zum Herunterfahren des Systems.

- Blatt = Kindprozess ohne weitere Kindprozesse.

- Nach ihrer Erzeugung existieren Prozesse unabhängig von anderen Prozessen, auch von ihrem Vater.

17.1 Shell - Kommandos - Prozesse

- Eine Shell (Programm) wird als Prozess ausgeführt.
- Es kann viele Prozesse geben, die eine Shell ausführen.

- Erkennt die Shell im Eingabestrom ein Kommando, so erzeugt sie einen Kindprozess, der dieses Kommando ausführt.

- Ein Aufruf einer Shell ist ein normales Kommando.

- **Synchrone Kommandoabarbeitung**: Der Shellprozess wartet, bis der Kommandoprozess terminiert.

```
Kommando
cp Quelle Ziel
```

- **Asynchrone Kommandoabarbeitung**: Der Shellprozess wartet nicht, bis der Kommandoprozess terminiert, sondern beide laufen parallel weiter.

```
Kommando &
cp Quelle Ziel &
```

17.2 Interprozesskommunikation

Dateien

- Mehrere Prozesse benutzen eine Datei zum Datenaustausch, als Kommunikationsmedium.

Pipes

- *Pipe*: röhrenartiger Datenkanal
- mit einer Schreib- und einer Leseseite;
- aus Prozesssicht wie Datei;
- ein Prozess kann entweder schreiben oder lesen;
- realisiert als Puffer im Hauptspeicher.

Signale

- Mit einem Signal kann einem Prozess eine Botschaft gesendet werden;
- Ursachen: intern oder extern;
- Wirkung: Unterbrechung des signalisierten Prozesses;
- Prozess kann Signal abfangen durch
  - Ignorieren;
  - Aufrufen einer Signal-Behandlungsfunktion.

17.2.1 Pipes und Filter

Die Standardausgabe eines Prozesses zur Standard-eingabe eines anderen Prozesses leiten.

- **Pipe** auf Shell-Ebene: `|`

```
Kommando1 | Kommando2
```

```
ls | pr -2 | lpr
who | wc -l
ls -R /usr/src|grep '.c'|wc -l
```

- Worin besteht der Unterschied zu „Pipes“ in MS-DOS?

- **Filter** := Programm, das aus einer Eingabedatei liest, die Daten transformiert & in eine Ausgabe-datei schreibt.

- Viele Shell-Kommandos sind Filter & lassen sich daher beliebig durch Pipes kombinieren.

+ Keine temporären Dateien erforderlich.

17.3 Prozessbezogene Systemaufrufe

17.3.1 Überlagerung

`exec (`*`Datei, Arg1,..., Argn`*`)`

- Im aufrufenden Prozess wird das aktuelle Programm durch das Programm in *`Datei`* überlagert & dann ausgeführt.
- Das Programm erhält die Argumente *`Argi`*.
- Die Umgebung des Prozesses bleibt erhalten.

17.3.2 Erzeugung

`Prozessnr` `= fork ()`

- Erzeugt eine Kopie des aufrufenden Prozesses;
- der aufrufende Prozess wird zum **Vaterprozess**;
- die erzeugte Kopie wird zum **Kindprozess**;
- unmittelbar nach Ausführung von `fork` haben beide Prozesse dieselbe Umgebung, außer:
- der Vater erhält die Prozessnummer des Kindes;
- das Kind erhält `0` zurück.

17.3.3 Synchronisation

`Prozessnr = wait (Status)`

- Der aufrufende Prozess wartet auf Beendigung eines Kindprozesses;
- er erhält Prozessnummer & Statusinformation des beendeten Prozesses zurück.

17.3.4 Terminierung

`exit (Status)`

- Der aufrufende Prozess beendet sich;
- seine Prozessnummer & Statusinformation werden an den Vaterprozess zurückgegeben.
- Falls der Vater schon terminiert hat, geht die Information an den Großvater, usw.

17.4 Prozessverwaltung

17.4.1 Aufteilung von Adressräumen

- Ein virtueller Adressraum ist in zwei Bereiche geteilt:
  - Der **Kernadressraumbereich** (*system address space*) ist dem Kern zugeordnet.
  - Der **Benutzeradressraumbereich** (*user address space*) ist einem Prozess zugeordnet.
- Der Kernadressraumbereich gehört zu jedem virtuellen Adressraum, der Benutzeradressraumbereich ist austauschbar.
- Jeder Prozess läuft in seinem eigenen Benutzeradressraumbereich.
  - Ein Prozesswechsel erfordert einen Wechsel des Benutzeradressraumbereichs.
  - Die Adressraumbereiche verschiedener Prozesse sind gegeneinander geschützt.

17.4.2 Ausführungsmodi

- **Benutzermodus** (*user mode*):
  - nur nichtprivilegierte Befehle ausführbar,
  - nur Benutzeradressraum zugreifbar.

- **Kernmodus** (*kernel mode*):
  - auch privilegierte Befehle ausführbar,
  - auch Kernadressraum zugreifbar.

17.4.3 Kontext eines Prozesses

- Der **Kontext** eines Prozesses besteht aus
  - dem Status auf Benutzerebene,
  - dem Status auf Kernebene.
- Der **Status auf Benutzerebene** besteht aus
  - dem Benutzeradressraumbereich,
    enthält als logische Segmente
    - ein Textsegment
    - zwei Datensegmente,
    - ein Kellersegment.
  - dem Inhalt der Prozessorregister.
- Der **Status auf Kernebene** besteht aus
  - einer Prozessstruktur, d.h. einem Eintrag in der Prozesstabelle,
  - dem Benutzerbereich (*user area*), der die Benutzerstruktur (*user structure*) enthält,
  - dem Status der Speicherverwaltung.