

# Betriebssysteme

## *Kap. 2: Prozesse und Threads*

### *2.3 Scheduling*

Stand: WS 11/12 (17.01.12)

Prof. Dr. Wolfgang Kuechlin

*Dipl.-Inform., Dr. sc. techn. (ETH)*

**Arbeitsbereich Symbolisches Rechnen  
Wilhelm-Schickard-Institut für Informatik  
Fakultät für Informations- und Kognitionswissenschaften**

**Universität Tübingen**

**Steinbeis Transferzentrum  
Objekt- und Internet-Technologien (OIT)**

[Wolfgang.Kuechlin@uni-tuebingen.de](mailto:Wolfgang.Kuechlin@uni-tuebingen.de)  
<http://www-sr.informatik.uni-tuebingen.de>



# Kap 2.3 Prozess-Ablaufplanung (Scheduling)

---

## ➤ Inhalt

- Ziele einer Prozess-Ablaufplanung
- Karussell Laufplanung (round robin scheduling)
- Laufplanung mit Prioritäten (priority scheduling)
- Mehrfach-Warteschlangen
- Kürzester Auftrag zuerst
- 2 Ebenen Ablaufplanung



# Ziele des Scheduling

---

- Fairness für die Prozesse
- Maximale HW-Auslastung
- Minimaler Scheduling-Aufwand
- Tolerierbare Antwortzeiten
- Bevorzugung von Prozessen mit erwünschtem Verhalten
- Sinnvolles Verhalten bei Überlast
- Behandeln von Prozessen gemäß ihrer Wichtigkeit
- Konsistentes und voraussagbares Schedulingverhalten



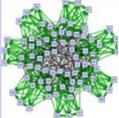
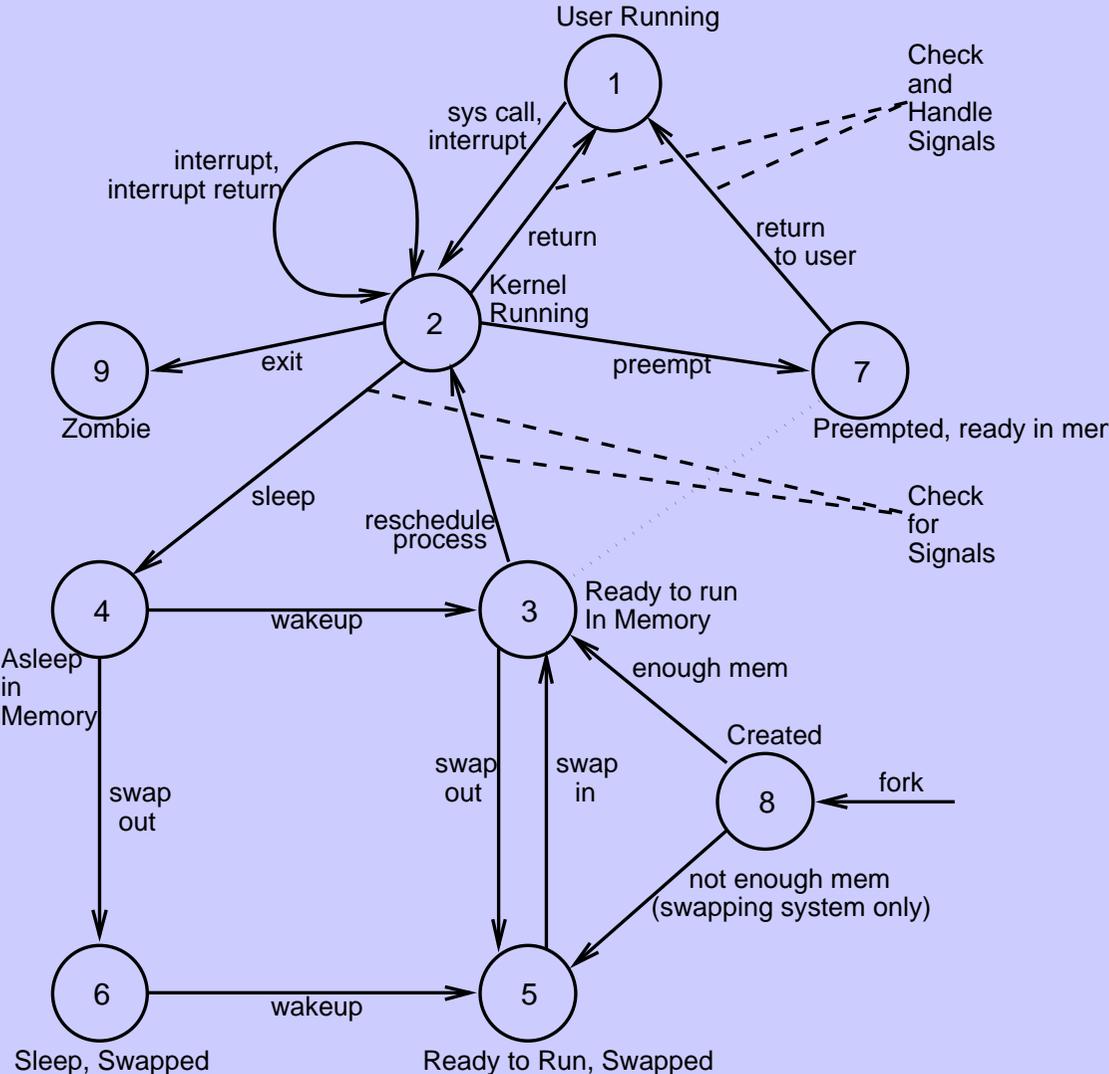
# Scheduling

---

- Multiprogramming
  - CPU Auslastung maximieren
- Time-Sharing
  - Wechsel der Prozesse so schnell, dass interaktives Arbeiten möglich ist.
- Realzeit-Systeme
  - Reaktionszeiten garantieren (Echtzeit → **R**echtzeitig)
- Queues
  - Job Queue
  - Ready Queue
  - Device Queue
- Prozesse migrieren zwischen den Queues.



# Prozess-Zustände in UNIX



# Scheduling-Warteschlange

## Lineare Liste von Prozessen (PCBs)

---

- Kurzfristiger Scheduler
  - sortiert Prozesse nach Prioritäten, ggf. Berechnung der Prio.
  - wählt Prozesse aus und teilt ihnen CPU zu (dispatching)
- Mittelfristiger Scheduler (Swapper)
  - entscheidet, welche Prozesse im Hauptspeicher bleiben und welche ausgelagert werden. Unterstützt Speicherverwaltung und gleicht Systemlast aus.
- Langfristiger Scheduler (Job-Scheduler in Batch-Systemen)
  - entscheidet, welche **Programme** als nächstes gestartet werden (Prozesse zum System zulassen und in Hauptspeicher laden)



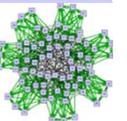
# Scheduling-Warteschlange

## Lineare Liste von Prozessen (PCBs)

---

### ➤ Dispatcher

- übernimmt Aufsetzen des Prozesses nach Auswahl durch Scheduler
  - Context Switching/Umschaltung in User Mode / Übergeben der Kontrolle an den Prozess



# Non-Preemptive Scheduling

- Non-preemptive: Kein Entzug des Prozessors
  - feste Priorität (keine Anpassung nach Ressourcenverbrauch)
  - harte Form: nur freiwillige Aufgabe: *yield* (cooperative sched.)
  - weiche Form: Entzug nur durch Prozess höherer Priorität
- Laufender Prozess bleibt auf CPU bis er entweder
  - blockiert oder terminiert
  - weiche Form: Prozess höherer Priorität lauffähig wird
- Max. Laufzeit eines Prozesses vorhersagbar (keine unfreiwillige Unterbrechung unbestimmter Dauer)
- Einsatz in Realzeitsystemen
  - Systeme mit deterministischem Verhalten: Vollständiges Ablaufdesign.



# Preemptive Scheduling

---

- Preemption (=Verdrängung)
  - Allgemein in BS: Entzug von Betriebsmitteln
  - Beim Scheduling: Entzug des Prozessors
- Laufender Prozess wird vom Prozessor genommen bei
  - Termination
  - Blockierung nach system call (Wechsel *running* → *blocked*)
  - Preemption
    - für Scheduling relevantes Ereignis tritt ein
      - Interrupts, insbesondere Timer-Interrupts und Ende der Zeitscheibe (time-slice)
      - System call
    - Neuer Prozess höherer Priorität wird lauffähig
    - Priorität des laufenden Prozesses ändert sich (z.B. durch Ressourcenverbrauch)
    - Scheduler bestimmt Prioritäten neu
    - Scheduler findet anderen Prozess höchster Priorität
    - Scheduler verdrängt bisherigen Prozess (*running* → *ready*)
    - Scheduler veranlasst dispatching des neuen Prozesses höchster Priorität



# Einsatz des Preemptive Scheduling

---

- **Priorität von wichtigen Prozessen im Bedarfsfall**
  - Bei reaktiven Systemen (Realzeit) u.U. sofortige Reaktion auf externe Ereignisse
    - > laufender Prozeß wird verdrängt.
- **Akzeptable Antwortzeiten (Multitasking)**
  - Bei interaktiven Systemen (Timesharing) soll ein CPU-lastiger Prozeß (z.B. Simulationsberechnung) regelmäßig unterbrochen werden von interaktiven Prozessen
    - z.B. interaktive Dateneingabe durch Benutzer



# Beeinflussung des Scheduling

- Folgende Prozesseigenschaften können (abhängig vom eingesetzten Verfahren) das Scheduling beeinflussen
  - Laufzeitverhalten
    - I/O-lastiger (I/O-bound) Prozess blockiert nach kurzer Laufzeit
    - CPU-lastiger (CPU-bound) Prozess wird meistens preempted
  - Wichtigkeit des Prozesses
  - Ressourcenbedarf des Prozesses
    - Bisher verbrauchte Prozessorzeit
    - Bisherige Verweildauer im System
    - Speicherverhalten, Page Fault Rate
    - Bedarf an Ressourcen (neben Speicher/CPU)
    - Noch nötige Prozessorzeit bis zur Termination



# FIFO-Scheduling

---

- Einfache First-In First-Out Queue
- Neue Prozesse werden an die Queue angefügt
- Non-Preemptive
  - Prozess läuft bis zur Termination oder bis zum Blockieren
- Große durchschnittliche Antwortzeit
  - > ungeeignet für interaktive Systeme

Ungebräuchlich, enthalten in anderen  
Scheduling-Algorithmen



# Zeitscheiben

---

## ➤ Time-Slice

- Zeitquantum, das der Prozess verbrauchen darf
- Wird gemessen in  $n$  clock-ticks

## ➤ Hardware clock, clock-tick, system timer

- Prozessor hat System-Timer (hardware clock), der vom OS auf eine Zeitdauer programmiert wird. (Linux: 10ms, auf x86 1ms)
- Nach Ablauf der Zeit (clock-tick) wird ein timer-interrupt ausgelöst und von speziellem handler behandelt
- Der handler dekrementiert alle Software-Uhren und löst Aktionen aus, sobald die Uhr  $\leq$  Null wird.
- Der handler dekrementiert das time-slice des Prozesses



# Round Robin Scheduling

## ➤ FIFO-Scheduling mit Time-Slice

- Preemptive
  - Verdrängung durch Prozess höherer Prio oder nach Ablauf der time-slice
  - Verdrängung durch nächsten Prozess (FIFO) gleicher Priorität.
- Limitierte Wartezeiten, geeignet für interaktive Systeme

## ➤ Bestimmung der Time-Slice

- Grundsätzliches:
  - kurze Zeitscheibe ==> relativ großer Context Switch Overhead, aber relativ kurze Wartezeiten für Prozesse in der run-queue
  - lange Zeitscheibe ==> kleiner Overhead, lange Wartezeiten
  - feste time-slices oder dynamisch unterschiedliche time-slices
- Gebräuchliche Werte
  - Linux: Prozess höherer Prio bekommt längere time-slice
  - typisch: 10-100 ms. Linux: default=100ms, Min = 5ms, Max=800ms.



# Priority Scheduling

---

- Unterteilung der Prozesse in Prioritätsklassen
- Separate Round Robin Queue für jede Priorität
- Prozess höherer Priorität hat Bearbeitungs-Vorrang vor Prozessen mit tieferer Priorität.
  - Problem
    - > Prozesse mit tieferer Priorität können verhungern
  - Lösungen
    - (gewisse) Zeit für tiefere Prioritäten reservieren
    - Periodische Erhöhung der Prioritäten von nicht bearbeiteten Prozessen (=> Multi-Level Feedback Queue Scheduling)

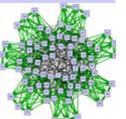


# Multi-Level Feedback Queue Scheduling

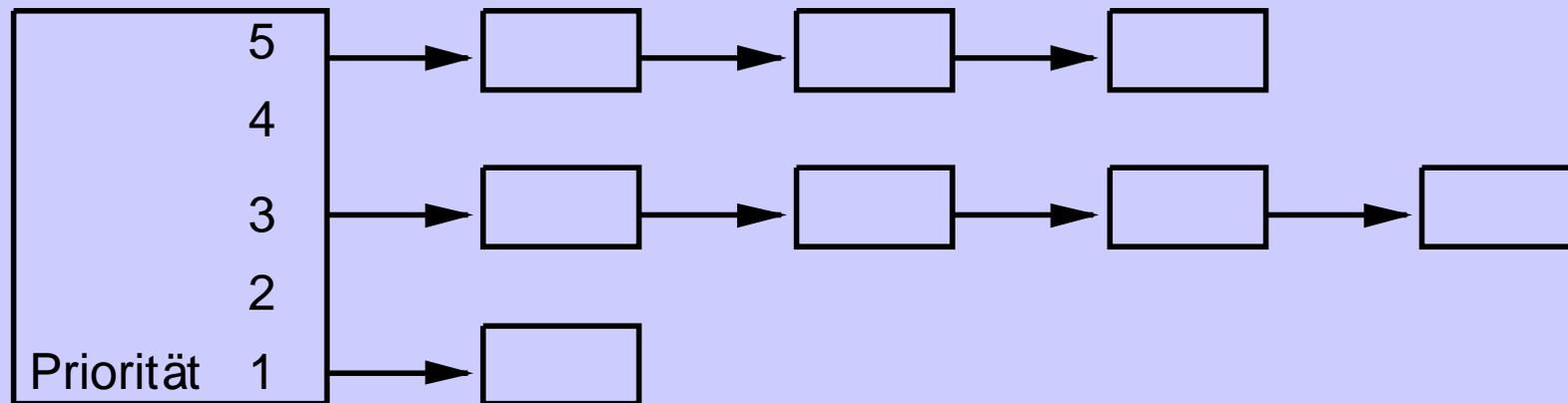
---

## ➤ Zusätzlich

- Veränderung der Prioritäten aufgrund des Prozessverhaltens
  - blockiert: Priorität wird erhöht
  - preempted: Priorität wird gesenkt
  - hoher Rechenzeitverbrauch: Priorität wird gesenkt
  - niedriger Rechenzeitverbrauch: Priorität wird erhöht



# Mehrfach-Warteschlangen



# Multi-Processor Scheduling

## ➤ Probleme

- Gleichmäßige Auslastung aller Prozessoren
- Berücksichtigung von Prozessor-Affinitäten
- Vermeidung von Lock-Contention der run-queue(s)

## ➤ Run-Queue(s)

- separate Run-Queues pro Prozessor, ansonsten lock-contention

## ➤ Load Balancing

- Last-Ausgleich zwischen Prozessoren
  - Pull-Verfahren: Prozessor holt sich Last von langen run-queues
  - Push-Verfahren: Prozessor gibt Last an kurze run-queues ab

## ➤ Prozessor Affinitäten

- entweder statisch festgelegt (siehe threads)
- oder dynamisch gefüllte caches beachten: cache-hot Prozess nicht bewegen



# VMS Scheduling

---

- Ziel
  - Gleichzeitige Bearbeitung von Realtime- und Timesharing-Applikationen
- Neue Scheduling Klassen-Einteilung
  - Realtime
  - System
  - User
- Preemptive
  
- Prioritätsgesteuerte Multilevel (32) Queue
  - Immer der lauffähige Prozess mit der höchsten Proirität wird ausgewählt



# VMS Scheduling

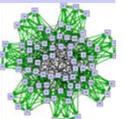
---

## ➤ 0-15: Timesharing

- Round Robin mit Timeslice innerhalb Prioritätsklasse
  - Prozessverhalten verändert Priorität (Feedback, Priorität verbleibt innerhalb 0-15)
  - Unterbrechung bei Ende der Timeslice oder falls Prozess mit höherer Priorität lauffähig wird

## ➤ 16-31: Realtime

- FIFO innerhalb Prioritätsklasse
- Feste Prioritäten
- Unterbrechung nur falls Prozess mit höherer Priorität lauffähig wird.



# Scheduling in UNIX (BSD 4.4)

## ➤ Ziel

- Timesharing, Bevorzugung von interaktiven Prozessen, nicht-interaktive Prozesse sollen aber nicht verhungern.

## ➤ Zuteilung von CPU-Zeit aufgrund der Prioritäten

## ➤ Multilevel Feedback

- Mehrere Prioritätsebenen
- Prioritätsanpassung nach feed-back (Ressourcennutzung)

## ➤ Preemption

- nach Ablauf der Zeitscheibe (z.B. 100ms)
- wenn ein Prozess höherer Priorität lauffähig wird
  - Prozess im user mode wird sofort verdrängt
  - Prozess im Kernel Mode wird erst bei return-to-user-mode verdrängt

(klassisches UNIX ist nicht realzeit-fähig)



# Scheduling in UNIX (BSD 4.4)

## ➤ Feedback

- Prozessornutzung geht in Priorität ein: wenn der Prozess läuft, wird sie gesenkt, wenn er nicht läuft, wird sie erhöht.
- Neuberechnung alle 4 clock ticks (4\*10ms)
- numerisch größere Werte → niedrigere Priorität
- $p\_userpri = PUSER + [p\_estcpu / 4] + 2 * p\_nice$ 
  - PUSER = 50 (baseline prio for user processes)
  - $-20 \leq p\_nice < 20$  (user settable)
  - p\_estcpu zählt die clock-ticks, in denen der Prozess läuft
- Einmal pro Sekunde wird p\_estcpu gefiltert, um alte Werte zu vergessen
  - $p\_estcpu = [(2 * load) / (2 * load + 1)] * p\_estcpu + p\_nice$
  - Die Systemlast *load* wird aus der durchschnittl. Länge der Run-Queues in der letzten Minute bestimmt



# Scheduling in UNIX (BSD 4.4)

## ➤ Beispiel des p\_estcpu Filters

- $p\_estcpu = [(2 \cdot load) / (2 \cdot load + 1)] \cdot p\_estcpu + p\_nice$
  - Annahme: 1 Prozess im System
    - 1 Filterschritt:  $p\_estcpu = [2/3] \cdot p\_estcpu + p\_nice$
  - Annahme: 1 Prozess,  $p\_nice=0$ ,  $p\_estcpu = T_i$  in Intervall  $i$ 
    - 1. Schritt:  $p\_estcpu = [2/3] \cdot T_0$
    - 2. Schritt:  $p\_estcpu = [2/3] \cdot (T_1 + [2/3] \cdot T_0) = 0.66 \cdot T_1 + 0.44 \cdot T_0$
    - 3. Schritt:  $p\_estcpu = 0.66 \cdot T_2 + 0.44 \cdot T_1 + 0.30 \cdot T_0$
    - 4. Schritt:  $p\_estcpu = 0.66 \cdot T_3 + 0.44 \cdot T_2 + 0.30 \cdot T_1 + 0.20 \cdot T_0$
    - 5. Schritt:  $p\_estcpu = 0.66 \cdot T_4 + 0.44 \cdot T_3 + 0.30 \cdot T_2 + 0.20 \cdot T_1 + 0.13 \cdot T_0$
  - Nach 5 Schritten gehen nur noch 13% des Verbrauchs  $T_0$  ein
  - Nach 5 Sekunden sind grob 90% des Verbrauchs vergessen.
- Nach sleep von  $n$  sec:  $p\_estcpu = p\_estcpu \cdot [2 \cdot load / (2 \cdot load + 1)]^n$



# Scheduling in Linux 2.6

- UNIX ähnlicher scheduler,  $O(1)$  Laufzeit, variable time-slices, ergänzt durch Realzeit-Prioritäten
- Prioritäten (insgesamt 140)
  - Realzeit-Prioritäten von 0 bis 99. Realzeit Prozesse sind in höherer Prio-Klasse als User-Prozesse
  - Nice Werte von -20 (höchste Prio) bis +19 (niedrigste Prio)
- Time-Slices
  - höchste Prio=max timeslice (800ms), kleinste=min (5ms)
  - Rest von timeslice bleibt erhalten, falls Prozess vorzeitig den Prozessor aufgibt.
  - Prozesse mit positivem Rest laufen vor Prozessen ohne Rest
  - Sind alle Reste = 0 so werden alle time-slices neu berechnet.



# Scheduling in Linux 2.6

## ➤ Run-Queues

- jeder SMP-Prozessor hat eine eigene (multi-level) run-queue
- Jede Run-Q mit eigenem lock
  - Synchronisation zwischen Prozessoren durch ordered lock acquisition
- Pro Run-Q 2 priority-arrays:
  - active array: runnable Tasks mit Rest timeslice
  - expired array: runnable Tasks mit erschöpftem timeslice
- Priority-array hat 140 sub-queues auf den Prioritäts-Ebenen
- Bitmap aus  $5 \cdot 32 = 160$  bits zur Auswahl der nächsten Task
- $\text{Bitmap}[k]=1$  falls eine Task der Prio  $k$  runnable wird
  - `sched_find_first_bit()` findet höchstes Bit
  - oftmals mit hardware-support
- Schneller Zugriff auf Task höchster Prio (wenige Instruktionen)



# Scheduling in Linux 2.6

---

- Es läuft die nächste Task mit der höchsten Prio, die noch einen Rest vom timeslice hat
- Sobald die Tasks im active array erschöpft sind, werden die arrays vertauscht (pointer swap)
  - das neue expired array ist leer
  - das neue active array besteht aus tasks mit frischen timeslices
- Neuberechnung der Timeslices
  - Neuberechnung nach Array-Austausch geht nicht in  $O(1)$ !
  - Deshalb: wenn eine Task ihr timeslice erschöpft hat, wird sie in das expired array eingetragen und es wird sofort ihr neues zukünftiges timeslice auf Vorrat berechnet.
    - Umschalten zu neuem active array in  $O(1)$  möglich (pointer swap).



# Scheduling in Linux 2.6

- Multi-Processor Scheduling (SMP-Scheduling)
  - Per-processor locked run-queues with load balancing
- Load balancing (Last-Ausgleich)
  - Load balancing wird alle 200ms aktiviert oder wenn eigene Run-Queue leer ist
  - *Pull-Mechanismus*: Prozessor mit wenig Tasks holt Nachschub vom Prozessor mit den meisten Tasks
    - Ordered locking der run-queues zur Vermeidung von deadlocks
    - Ungleichgewicht muss mindestens 5:4 sein
    - Vorzugsweise werden tasks vom expired array genommen
    - Vom ausgewählten array werden Tasks der höchsten Prio genommen
    - Task darf nicht laufen, nicht Prozessor-affin sein, nicht cache-hot sein
    - Wird wiederholt bis das Ungleichgewicht weg ist.

