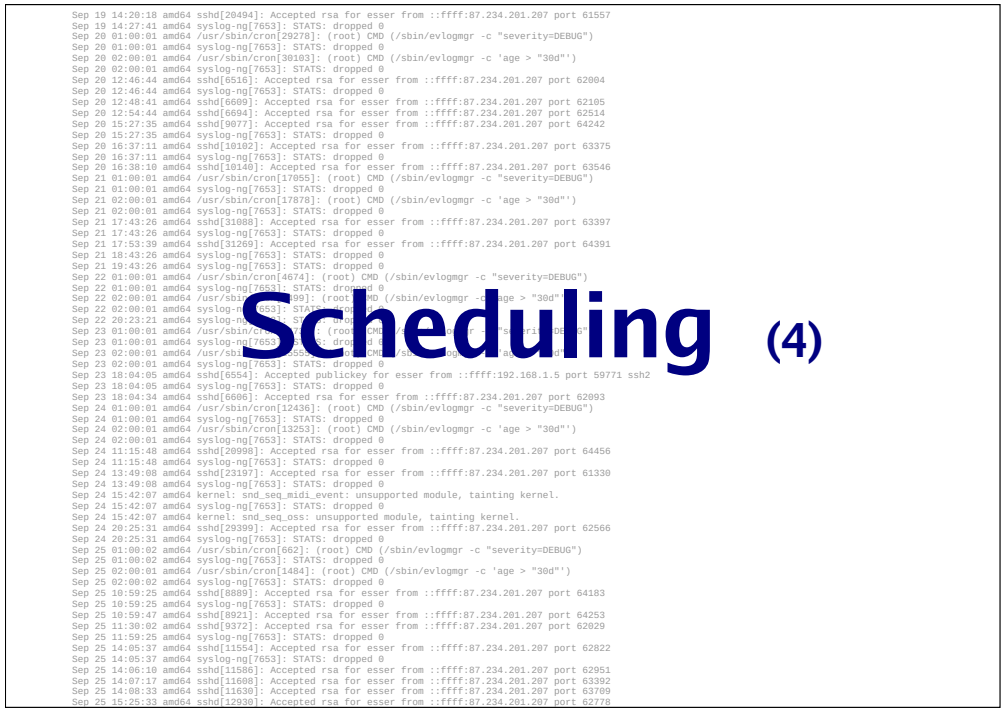


Scheduling (4)

Fair-Share-Implementierung (1)

```
#!/usr/bin/python
processes = [1,2,3]                     # Drei Prozesse
process_queue = [1,2,3]
base_priorities = {1:60, 2:60, 3:60}    # mit gleichen Prioritäten
groups = [ [1], [2,3] ]                 # zwei Gruppen
weight = [0.5, 0.5]                     # mit zugehörigen Gewichten
runtime = 30                            # wie lange laufen lassen?

# Initialisierung
proccount = len (processes)
groupcount = len (groups)
activity_log = []
seconds = 0
priorities = {}       # leere assoziative Listen
proc_cpu_util = {}
group_cpu_util = []   # geht bei Gruppen nicht...

for p in processes:
  priorities[p] = base_priorities[p]
  proc_cpu_util[p] = 0

for i in range(0,groupcount):
  group_cpu_util.append(0)
```

```
def print_status():
  print "%4d |" % seconds,
  for p in processes:
    prio = priorities[p]
    pcpu = proc_cpu_util[p]
    gcpu = group_cpu_util
      [group_of(p)]
    print "%3d %3d %3d  | "
      % (prio,pcpu,gcpu),
  return
# end print_status()
```


Fair-Share-Implementierung (2)

```
def find_min_priority():
  # Prozess mit der niedrigsten
  # Prioritaet suchen; bei mehreren
  # mit niedrigster: ersten davon
  minval = 9999
  for p in process_queue:
    if priorities[p] < minval:
      minval = priorities[p]
      minproc = p
  # Prozess in Warteschlange nach
  # hinten schieben
  process_queue.remove(minproc)
  process_queue.append(minproc)
  return minproc
# end find_min_priority()

def group_of(p):
  for i in range (0,groupcount):
    if p in groups[i]: return i
# end group_of()
```

```
def recalculate():
  # Prioritaeten neu berechnen
  proc_cpu_util[active] += 60
  g = group_of (active)
  group_cpu_util[g] += 60

  for g in range(0,groupcount):
    group_cpu_util[g] /= 2

  for p in processes:
    g = group_of (p)
    proc_cpu_util[p] /= 2    # halbieren
    prio = base_priorities[p]          \
      + int( proc_cpu_util[p] / 2 )    \
      + int( group_cpu_util[g] /       \
                (4 * weight[g]) )
    priorities[p] = prio
  return
# end recalculate()
```


Fair-Share-Implementierung (3)

```
# begin main()
print "Zeit |",
for p in processes:
  print "Prozess %2d   | " % p,
print
print_status()

for i in range(0,runtime):
  active = find_min_priority()
  print "-> Scheduler aktiviert P.",
     active
  activity_log.append(active)
  # aktiven Prozess ausfuehren
  seconds += 1  # Zeit hochzaehlen
  recalculate()
  print_status()

print
```

```
# Statistik ausgeben

print

oldprocess = 0
for p in processes:
  st = ""  # leerer String
  if group_of(p) != group_of(oldprocess):
    print "NEUE GRUPPE"
  print "Prozess %1d:" % p,
  for i in range(0,runtime):
    if activity_log[i] == p:
      st += "x"
    else:
      st += "-"
  print st
  oldprocess=p

# end main()
```

Fair-Share-Implementierung (4)

```
> ./fair-share.py
Zeit | Prozess  1    |  Prozess  2    |  Prozess  3    |
   0 |  60   0   0   |   60   0   0   |   60   0   0   |  -> Scheduler aktiviert P. 1
   1 |  90  30  30   |  60   0   0    |   60   0   0   |  -> Scheduler aktiviert P. 2
   2 |  74  15  15   |   90  30  30   |   75   0  30   |  -> Scheduler aktiviert P. 1
   3 |  96  37  37   |   74  15  15   |  67   0  15    |  -> Scheduler aktiviert P. 3
   4 |  78  18  18   |   81   7  37   |   93  30  37   |  -> Scheduler aktiviert P. 1
   5 |  98  39  39   |   70   3  18   |   76  15  18   |  -> Scheduler aktiviert P. 2
   6 |  78  19  19   |   94  31  39   |   82   7  39   |  -> Scheduler aktiviert P. 1
   7 |  98  39  39   |   76  15  19   |   70   3  19   |  -> Scheduler aktiviert P. 3
   8 |  78  19  19   |   82   7  39   |   94  31  39   |  -> Scheduler aktiviert P. 1
   9 |  98  39  39   |   70   3  19   |   76  15  19   |  -> Scheduler aktiviert P. 2
  10 |  78  19  19   |   94  31  39   |   82   7  39   |  -> Scheduler aktiviert P. 1
  11 |  98  39  39   |   76  15  19   |   70   3  19   |  -> Scheduler aktiviert P. 3
  12 |  78  19  19   |   82   7  39   |   94  31  39   |  -> Scheduler aktiviert P. 1
  13 |  98  39  39   |   70   3  19   |   76  15  19   |  -> Scheduler aktiviert P. 2
  14 |  78  19  19   |   94  31  39   |   82   7  39   |  -> Scheduler aktiviert P. 1
  15 |  98  39  39   |   76  15  19   |   70   3  19   |  -> Scheduler aktiviert P. 3
  16 |  78  19  19   |   82   7  39   |   94  31  39   |  -> Scheduler aktiviert P. 1
  17 |  98  39  39   |   70   3  19   |   76  15  19   |  -> Scheduler aktiviert P. 2
  18 |  78  19  19   |   94  31  39   |   82   7  39   |  -> Scheduler aktiviert P. 1
  19 |  98  39  39   |   76  15  19   |   70   3  19   |  -> Scheduler aktiviert P. 3
  20 |  78  19  19   |   82   7  39   |   94  31  39   |  -> Scheduler aktiviert P. 1
  21 |  98  39  39   |   70   3  19   |   76  15  19   |  -> Scheduler aktiviert P. 2
  22 |  78  19  19   |   94  31  39   |   82   7  39   |  -> Scheduler aktiviert P. 1
  23 |  98  39  39   |   76  15  19   |   70   3  19   |  -> Scheduler aktiviert P. 3
  24 |  78  19  19   |   82   7  39   |   94  31  39   |  -> Scheduler aktiviert P. 1
  25 |  98  39  39   |   70   3  19   |   76  15  19   |

NEUE GRUPPE
Prozess 1: x-x-x-x-x-x-x-x-x-x-x-x-x
NEUE GRUPPE
Prozess 2: -x---x---x---x---x---x---
Prozess 3: ---x---x---x---x---x---x-
```


Fair-Share-Implementierung (5)

Mehr Prozesse in einer Gruppe

```
processes = [1,2,3,4,5]                 # Fünf Prozesse
base_priorities = {1:60, 2:60, 3:60, 4:60, 5:60}
groups = [ [1], [2,3,4,5] ]             # zwei Gruppen
weight = [0.5, 0.5]                     # mit zugehörigen Gewichten
```

```
NEUE GRUPPE
Prozess 1: x-x--x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x
NEUE GRUPPE
Prozess 2: -x------x-------x-------x-------x-------x-------
Prozess 3: ---x------x-------x-------x-------x-------x-----
Prozess 4: ----x-------x-------x-------x-------x-------x---
Prozess 5: ------x-------x-------x-------x-------x-------x-
```


Fair-Share-Implementierung (6)

Gleich große Gruppen

```
processes = [1,2,3,4,5,6]               # Sechs Prozesse
base_priorities = {1:60, 2:60, 3:60, 4:60, 5:60, 6:60}
groups = [ [1,2,3], [4,5,6] ]           # zwei Gruppen
weight = [0.5, 0.5]                     # mit zugehörigen Gewichten
```

```
NEUE GRUPPE
Prozess 1: x-----x-----x-----x-----x-----x-----x---
Prozess 2: --x-----x-----x-----x-----x-----x-----x-
Prozess 3: ----x-----x-----x-----x-----x-----x-----
NEUE GRUPPE
Prozess 4: -x-----x-----x-----x-----x-----x-----x--
Prozess 5: ---x-----x-----x-----x-----x-----x-----x
Prozess 6: -----x-----x-----x-----x-----x-----x----
```


Fair-Share-Implementierung (7)

Drei Gruppen mit verschieden vielen Prozessen

```
processes = [1,2,3,4,5,6,7]             # Sieben Prozesse
base_priorities = {1:60, 2:60, 3:60, 4:60, 5:60, 6:60, 7:60}
groups = [ [1], [2,3], [4,5,6,7] ]      # drei Gruppen
weight = [0.33, 0.33, 0.33]             # mit zugehörigen Gewichten
```

```
NEUE GRUPPE
Prozess 1: x--x--x--x--x--x--x--x--x--x--x--x--x--x--x--x- (34 Prozent)
NEUE GRUPPE
Prozess 2: -x-----x-----x-----x-----x-----x-----x-----x-----x (18 Prozent)
Prozess 3: ----x-----x-----x-----x-----x-----x-----x-----x--- (16 Prozent)
NEUE GRUPPE
Prozess 4: --x-----------x-----------x-----------x----------- ( 8 Prozent)
Prozess 5: -----x-----------x-----------x-----------x-------- ( 8 Prozent)
Prozess 6: --------x-----------x-----------x-----------x----- ( 8 Prozent)
Prozess 7: -----------x-----------x-----------x-----------x-- ( 8 Prozent)
```

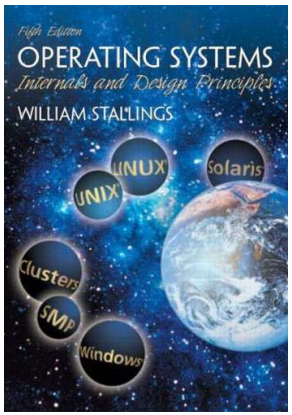
Shares als Ober-/Untergrenze

Zwei Möglichkeiten

- **Shares definieren Obergrenze:**
  System stellt sicher, dass kein Prozess mehr CPU-Zeit verbraucht als ihm zusteht (solange System sonst nicht idle wird)
- **Shares definieren Untergrenze,**
  also zugesicherte Eigenschaft:
  System stellt sicher, dass jeder Prozess exakt den versprochenen Anteil an Rechenzeit bekommt (oder mehr, falls möglich)

Scheduling auf Multi-Prozessor-Systemen

Folien basieren (teilweise) auf W. Stallings, Kap. 10



Klassifikation von Multiprozessor-Systemen

- Lose verbundener Multiprozessor
  - Jede CPU hat ihren eigenen Speicher und eigene I/O-Kanäle
- Funktionsspezialisierte Prozessoren
  - z. B. I/O-Prozessor
  - Von einem Master-Prozessor gesteuert
- Eng verbundenes Multiprocessing
  - Prozessoren haben gemeinsamen Hauptspeicher
  - Gesteuert vom Betriebssystem

Unabhängige Parallelität

- Separate Anwendungen
- Keine Synchronisierung
- Mehr als ein Prozessor verfügbar
  - Durchschnittliche Antwortzeit ist kleiner

(sehr) grobe Parallelität

- Synchronisation mehrerer Prozesse nur sehr selten nötig
- Gut für gleichzeitige Prozesse, die auf einem 1-CPU-System mit Multitasking laufen
  - Nur wenig Änderungen für Einsatz auf einer Multi-CPU-Maschine nötig

Mittel-grobe Parallelität

- Parallelverarbeitung / Multitasking innerhalb einer einzelnen Applikation
- Einzelne Applikation besteht aus einer Reihe Threads
- Threads synchronisieren sich typischerweise häufig

Feinkörnige Parallelität

- hoch-parallele Anwendungen
- spezialisiertes Fachgebiet

Granularität – Übersicht

Granularität	Beschreibung	Synchonisations-intervall
fein	Parallelität schon in einzelnen Befehlssequenzen	< 20
mittel	Parellelverarbeitung / Multitasking innerhalb einer Anwendung	20-200
grob	Multitasking von Prozessen in einer Multiprogramming-Umgebung	200-2.000
sehr grob	Verteilte Verarbeitung über mehrere Netzwerk-Knoten, die eine einheitliche Rechenumgebung bilden	2.000-1.000.000
unabhängig	Mehrere unabhängige Prozesse	–

Scheduling

- Zuordnung von Prozessen zu CPUs
- Einsatz von Multitasking auf einzelnen CPUs
- Dispatching eines Prozesses

Zuordnung von Prozessen zu CPUs (1)

- CPUs als einen Ressourcen-Pool behandeln und Prozesse CPUs bei Bedarfs zuordnen
- Permanente Zuordnung von Prozessen zu einer CPU
  - Short-term-Warteschlange für jede CPU
  - Weniger Overhead
  - Ein Prozessor könnte idle sein, während einer anderer eine lange Warteschlange abarbeitet

Zuordnung von Prozessen zu CPUs (2)

- Globale Warteschlange
  - Prozess einer beliebigen verfügbaren CPU zuordnen
- Master-/Slave-Architektur
  - Wichtige Kernel-Funktionen laufen immer auf einer bestimmten CPU
  - Master ist für das Scheduling verantwortlich
  - Slave schickt service request an den Master
  - Nachteile
    - Fehler im Master schießt ganzes System ab
    - Master kann zum Performance-Bottleneck werden

Zuordnung von Prozessen zu CPUs (3)

- Peer-Architektur
  - Betriebssystem kann auf jeder CPU laufen
  - Auf jeder CPU: eigener Scheduler (als Teil der dort laufenden BS-Instanz)
  - Verkompliziert das Betriebssystem
    - Sicher stellen, dass nicht zwei CPUs den gleichen Prozess auswählen

Prozess-Scheduling

- Eine Warteschlange für alle Prozesse
- Mehrere Warteschlangen für Prioritäten
- Alle Warteschlangen nutzen den gemeinsamen CPU-Pool
- Spezielle, optimierte Scheduling-Strategien sind weniger wichtig als im 1-CPU-Fall

Threads

- Bei Threads: Ausführung und restliche Prozess-Eigenschaften werden separat behandelt
- Anwendung kann aus einer Menge von Threads bestehen, die kooperieren und gleichzeitig im selben Adressraum laufen
- Threads, die auf verschiedenen CPUs laufen, können die Performance (des Prozesses) dramatisch erhöhen

Multiprocessor Thread Scheduling (1)

- Load Sharing
  - Prozesse werden **k**einer bestimmten CPU zugeordnet
- Gang Scheduling
  - Eine Reihe zusammengehöriger Threads erhält vom Scheduler gleichzeitig eine Menge von CPUs zugeteilt

Multiprocessor Thread Scheduling (2)

- Dedizierte CPU-Zuordnung
  - Threads werden einer bestimmten CPU zugeordnet
- Dynamic scheduling
  - Anzahl der Threads kann sich im Verlauf der Programmausführung ändern

Load Sharing

- Rechenlast wird gleichmäßig auf alle CPUs verteilt
- Kein zentralisierter Scheduler nötig
- nutzt globale Warteschlange(n)

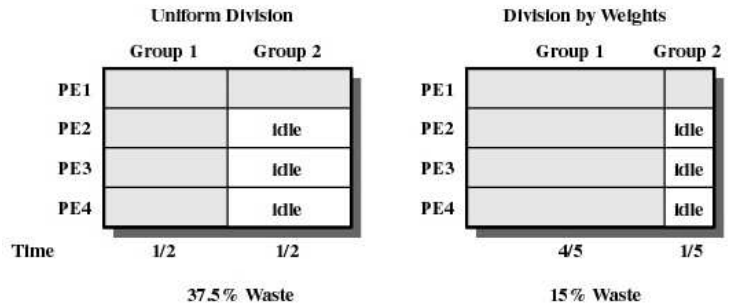
Nachteile des Load Sharing

- Zentrale Warteschlange erfordert gegenseitigen Ausschluss (*mutual exclusion*)
  - Kann ein Bottleneck sein, wenn mehr als eine CPU gleichzeitig „nach Arbeit sucht“
- Präemptive Threads werden vermutlich oft die CPU wechseln
  - Cache-Nutzung ist weniger effizient
- Wenn alle Threads in der globalen Warteschlange stehen, werden nicht alle Threads eines Prozesses gleichzeitig eine CPU erhalten

Gang Scheduling

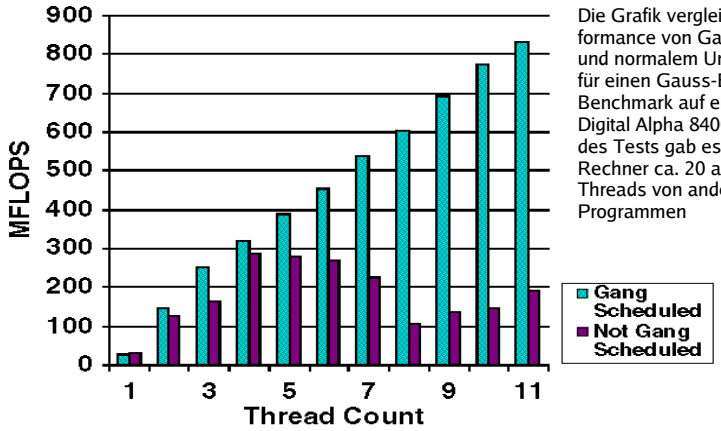
- Simultanes Scheduling von Threads, die alle zu einem Prozess gehören
- Nützlich für Anwendungen, in denen die Performance stark sinkt, sobald ein beliebiger Programmteil nicht läuft
- Threads müssen sich oft synchronisieren

Scheduling-Gruppen



Scheduling-Gruppen mit 4 / 1 Threads

Gang Scheduling: Performance



Die Grafik vergleicht die Performance von Gang Scheduling und normalem Unix-Scheduling für einen Gauss-Eliminations-Benchmark auf einer 12-CPU-Digital Alpha 8400. Während des Tests gab es auf dem Rechner ca. 20 ausführbare Threads von anderen Programmen

Bild: http://www.llnl.gov/asci/pse_trilab/sc98.summary.html

Dedizierte CPU-Zuordnung (1)

- Wenn Scheduler einen Prozess auswählt, teilt er dessen Threads je eine CPU zu
- Einige CPUs sind vielleicht idle
- Kein Multitasking auf den CPUs

Dedizierte CPU-Zuordnung (2)

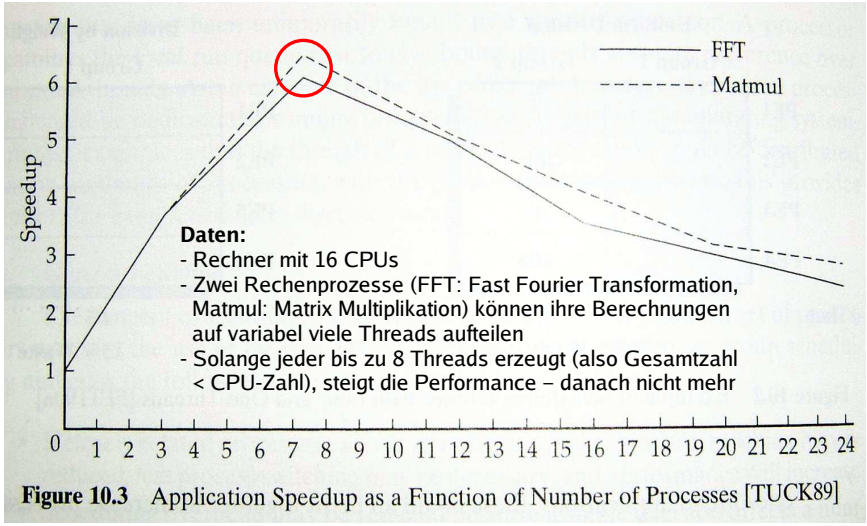


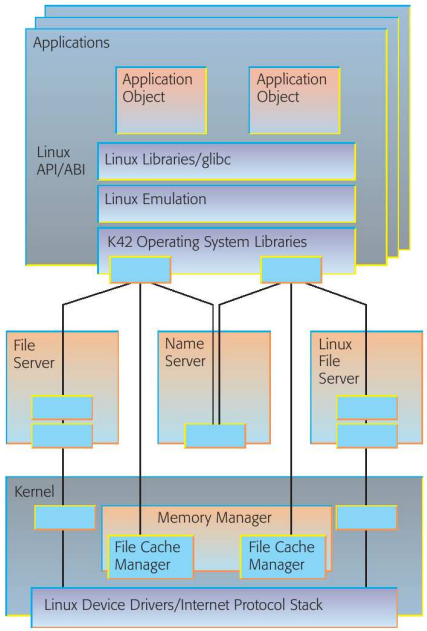
Figure 10.3 Application Speedup as a Function of Number of Processes [TUCK89]

Dynamisches Scheduling

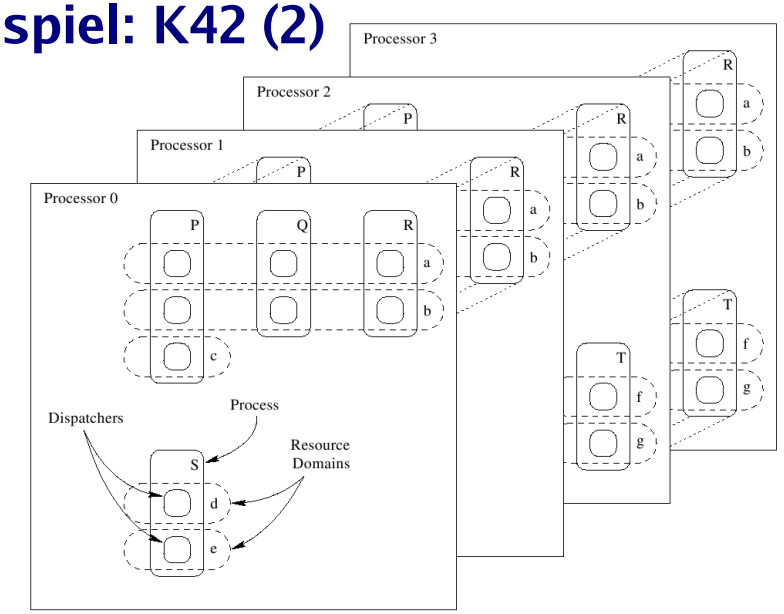
- Anzahl der Threads eines Prozesses dynamisch verändern: Prozess erzeugt neue Threads und beendet sie – und zwar *auf Basis der Ressourcen*, die das Betriebssystem bereitstellt
- Betriebssystem passt die Last an, um die CPU-Nutzung zu optimieren
  - Untätigen CPUs Threads zuordnen
  - Neue Prozesse können eine CPU erhalten, die bisher von einem Prozess benutzt wurde, der gerade mehr als eine CPU verwendet
  - Anfrage nicht bearbeiten, bis CPU verfügbar wird
  - Neue Prozesse erhalten eine CPU, bevor laufende Anwendungen weitere erhalten

Beispiel: K42 (1)

- Linux-kompatibles BS für Maschinen mit hunderten Prozessoren
- objekt-orientiert; BS-Aufrufe: IPC-Calls
- Zwei Scheduler-Ebenen
- Thread-Scheduling: komplett im User-Mode
- Kernel-Scheduler
  - verwaltet sog. Dispatcher, die dann die Threads verwalten (oft: ein Dispatcher pro Prozess);
  - läuft unabhängig auf jeder CPU



Beispiel: K42 (2)



Beispiel: K42 (3)

- Thread Migration:
  K42-Thread-Scheduler macht Load Balancing, indem er Threads von beschäftigten zu untätigen Dispatchern verschiebt
- Kernel verschiebt im Ausnahmefall auch Dispatcher zu einer anderen CPU
- Kernel-Scheduler aktiviert Resource-Domains (innerhalb einer Domain: Dispatcher in einem Ring)
- macht auch Gang-Scheduling (siehe Prozesse P,R)
- Dispatcher kann einzelne Threads blockieren lassen (z.B. wegen Page Fault oder I/O), ohne selbst zu blockieren

Beispiel: K42 (4)

- Prozess wählt aus:
  - echtes Multitasking -> mehrere Dispatcher nutzen
  - nur Programmierkomfort von Threads -> ein Dispatcher reicht
- Verschiedene Thread-Bibliotheken für Programmierer, darunter auch POSIX Threads
- K42 Scheduler: http://www.research.ibm.com/K42/white-papers/Scheduling.pdf (2002)
- Einführung in K42: http://www.research.ibm.com/journal/sj/442/appavoo.pdf