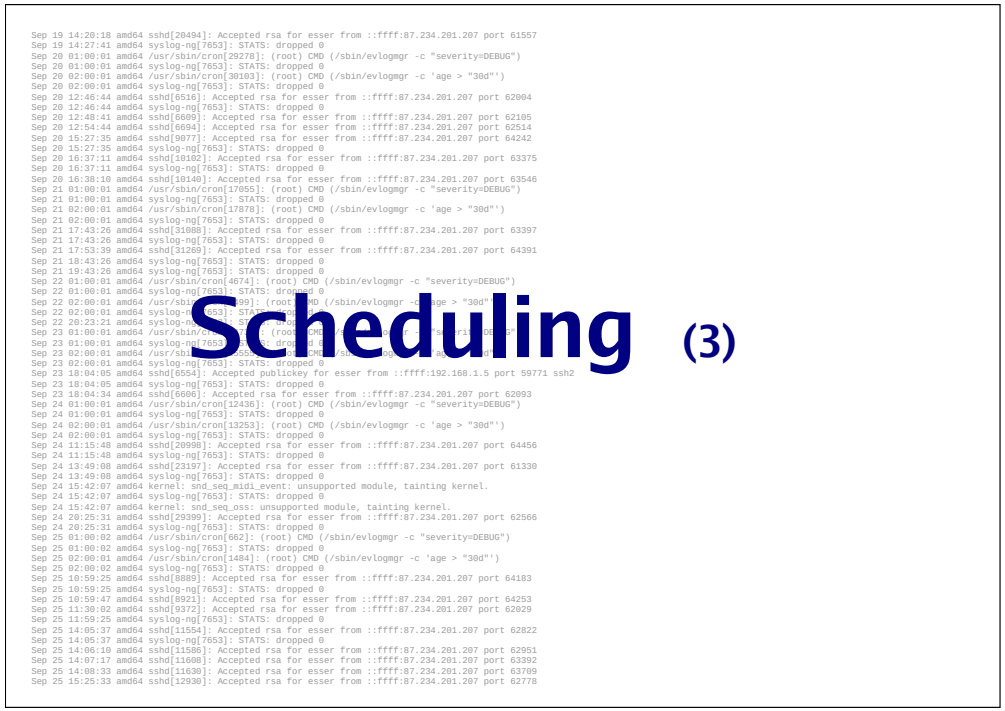
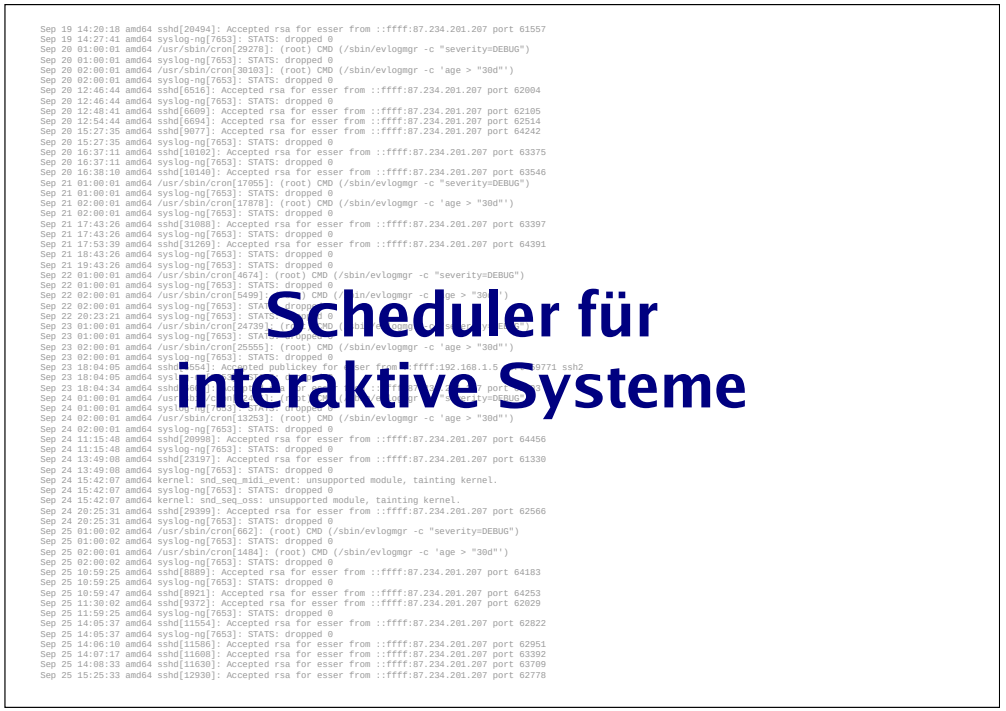


Scheduling (3)



Scheduler für interaktive Systeme

Beispiele für Scheduler

Konkrete Scheduling-Verfahren ...

1. für Batch-Systeme
2. für interaktive Systeme
3. für Echtzeitsysteme

Interaktive Systeme

- Typisch: Interaktive und Hintergrund-Prozesse
- Desktop- und Server-PCs
- Eventuell mehrere / zahlreiche Benutzer, die sich die Rechenkapazität teilen
- Scheduler für interaktive Systeme prinzipiell auch für Batch-Systeme brauchbar (aber nicht umgekehrt)

Interaktive Systeme

Scheduling-Verfahren für interaktive Systeme

- Round Robin
- Prioritäten-Scheduler
- Lotterie-Scheduler
- Fair-Share-Scheduler

Round Robin / Time Slicing (1)

Wie FCFS – aber mit Unterbrechungen

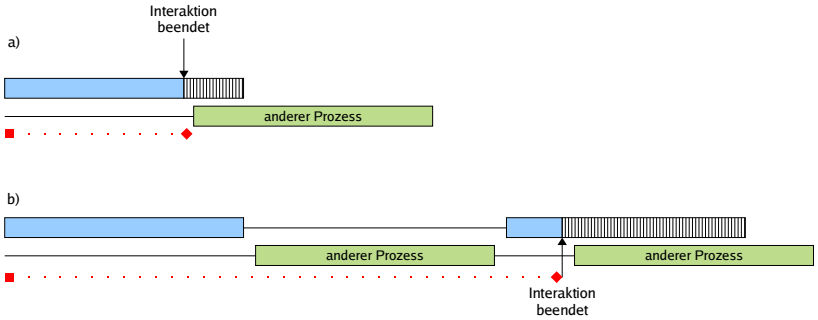
- Alle bereiten Prozesse in einer Warteschlange
- Jedem Thread eine Zeitscheibe (quantum, time slice) zuordnen
- Ist Prozess bei Ablauf der Zeitscheibe noch aktiv, dann:
  - Prozess verdrängen (preemption), also in den Zustand „bereit“ versetzen
  - Prozess ans Ende der Warteschlange hängen
  - Nächsten Prozess aus Warteschlange aktivieren

Round Robin (2)

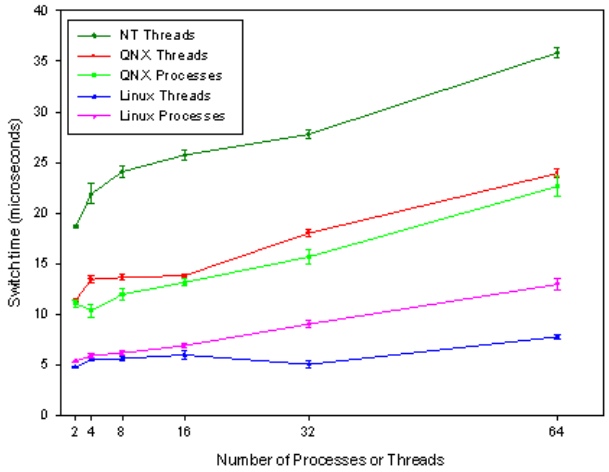
- Blockierten Prozess, der wieder bereit wird, hinten in Warteschlange einreihen
- Kriterien für Wahl des Quantums:
  - Größe muss in Verhältnis zur Dauer eines Context Switch stehen
  - Großes Quantum: evtl. lange Verzögerungen
  - Kleines Quantum: kurze Antwortzeiten, aber Overhead durch häufigen Context Switch

Round Robin (3)

- Oft: Quantum *q* etwas größer als typische Zeit, die das Bearbeiten einer Interaktion benötigt



Context-Switch-Latenz



Pentium Pro (200 MHz)

QNX 4.2
Red Hat Linux 5.1
Windows NT 4.0

Werte aus 1999

Round-Robin-Beispiel

Ankunftzeiten:
A: 0, B: 2, C: 4,
D: 6, E: 8

Zum Vergleich: First-Come-First Served (FCFS)

Round-Robin (RR), $q = 1$

Round-Robin (RR), $q = 4$

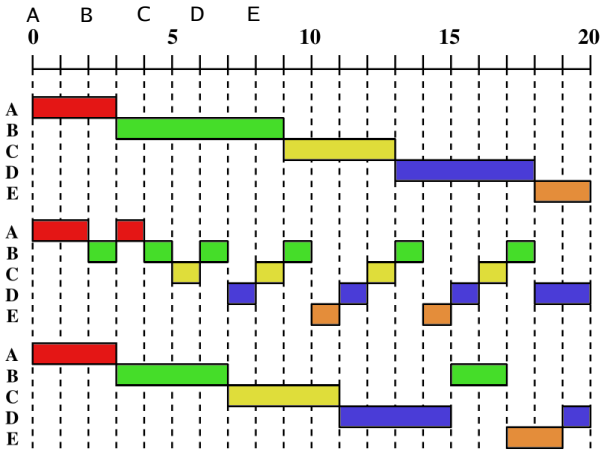


Bild: Stallings, S. 405

Virtual Round Robin (1)

- Round Robin unfair gegenüber I/O-lastigen Prozessen:
  - CPU-lastige nutzen ganzes Quantum,
  - I/O-lastige nur einen Bruchteil
- Idee: Nicht verbrauchten Quantum-Teil als „Guthaben“ des Prozesses merken
- Sobald I/O-Prozess wieder bereit ist (I/O-Ergebnis da): Restguthaben sofort aufbrauchen

Virtual Round Robin (2)

- Prozesse, die Zeitquantum verbrauchen, wie bei normalem Round Robin behandeln: zurück in Warteschlange
- Prozesse, die wegen I/O blockieren und nur Zeit $u < q$ ihres Quantums verbraucht haben, nach Blockade in Zusatzwarteschlange stecken
- Scheduler bevorzugt Prozesse in Zusatzschlange
- Quantum für diesen Prozess: $q–u$ (kriegt nur das, was ihm „zusteht“, was er beim letzten Mal nicht verbraucht hat)

Virtual Round Robin

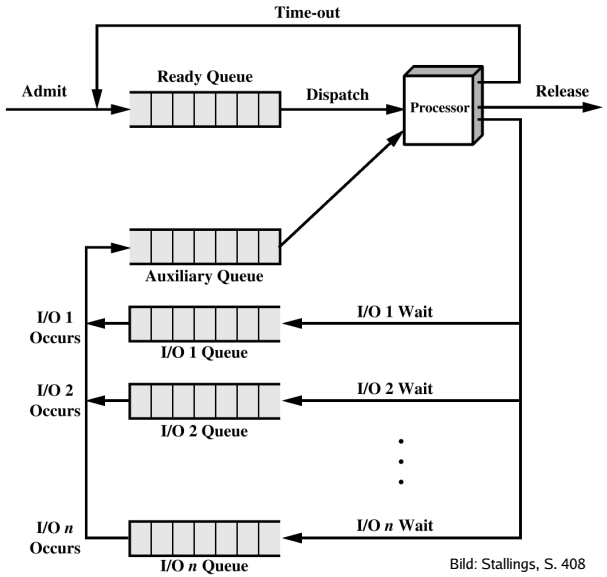


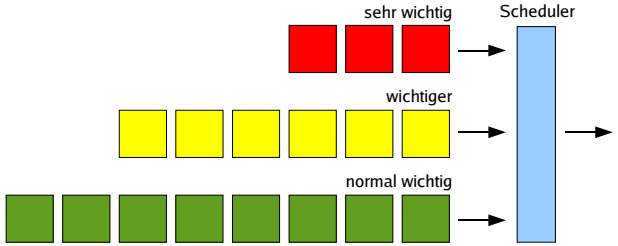
Bild: Stallings, S. 408

Prioritäten-Scheduler (1)

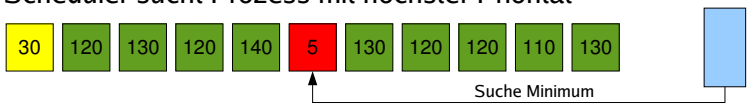
- Idee: Prozesse in mehrere Prioritätsklassen einteilen oder
  jedem Prozess einen Prioritätswert zuordnen
- Scheduler bevorzugt Prozesse mit hoher Priorität
- Priorität bei Prozesserzeugung fest vergeben oder vom Scheduler regelmäßig neu berechnen lassen
- Scheduling kooperativ oder präemptiv

Prioritäten-Scheduler (2)

a) Mehrere Warteschlangen für Prioritätsklassen



b) Scheduler sucht Prozess mit höchster Priorität



Prioritäten-Scheduler (3)

Mehrere Warteschlangen

- Prozesse verschiedenen Prioritätsklassen zuordnen und in jeweilige Warteschlangen einreihen
- Scheduler aktiviert nur Prozesse aus der höchsten nicht-leeren Warteschlange
- Präemptiv: Prozesse nach Zeitquantum unterbrechen
- Innerhalb der Warteschlangen: Round Robin

Prioritäten-Scheduler (3)

Keine Hierarchien, sondern individuelle Prozess-Prioritäten

- Alle Prozesse stehen in einer Prozessliste
- Scheduler wählt stets Prozess mit der höchsten Priorität
- Falls mehrere Prozesse gleiche (höchste) Priorität haben, diese nach Round Robin verarbeiten

Prioritäten-Scheduler (4)

Prozesse können verhungern ➞ Aging

Prioritätsinversion:

- Prozess hoher Priorität ist blockiert (benötigt ein Betriebsmittel)
- Prozess niedriger Priorität bestitzt dieses Betriebsmittel, wird aber vom Scheduler nicht aufgerufen (weil es höher-prioritäre Pr. gibt)
- Beide Prozesse kommen nie dran, weil immer Prozesse mittlerer Priorität laufen
- Ausweg: Aging

Prioritäten-Scheduler (5)

Aging:

- Priorität eines Prozesses, der bereit ist und auf die CPU wartet, wird regelmäßig erhöht
- Priorität des aktiven Prozesses und aller nicht-bereiten (blockierten) Prozesse bleibt gleich
- Ergebnis: Lange wartender Prozess erreicht irgendwann ausreichend hohe Priorität, um aktiv zu werden

Prioritäten-Scheduler (6)

Verschiedene Quantenlängen

- Mehrere Prioritätsklassen
- 1. Priorität = 1 Quantum, 2. Priorität = 2 Quanten, 3. Priorität = 4 Quanten, 4. Priorität = 8 Quanten
- Prozesse mit hoher Priorität erhalten kleines Quantum.
- Geben Sie die CPU vor Ablauf des Quantums zurück, behalten sie hohe Priorität
- Verbrauchen sie Quantum, verdoppelt Scheduler die Quantenlänge und stuft die Priorität runter – solange, bis Prozess sein Quantum nicht mehr aufbraucht

Prioritäten-Scheduler-Implementierung

```
#!/usr/bin/python
# proc: [start, runtime, prio, used time]
processes = {
  1:[0,20,100,0],  2:[5,10,50,0],
  3:[6,30,100,0],  4:[7, 2,10,0] }
runtime = 40                # wie lange laufen lassen?

# Initialisierung
proccount = len (processes)
activity_log = []
seconds = 0

def find_min_priority():
  # Prozess mit der niedrigsten Prioritaet suchen
  minval = 9999
  for p in processes:
    # Test auf drei Bedingungen: Wert kleiner als Minimum,
    # Prozess schon erzeugt und noch nicht beendet
    [start,maxtime,prio,used] = processes[p]
    if prio < minval and start <= seconds and used < maxtime:
      minval = prio
      minproc = p
  return minproc
# end find_min_priority()
```


Prioritäten-Scheduler-Implementierung

```
def recalculate():
  # Prioritaeten neu berechnen
  processes[active][3] += 1   # Used-Time-Wert erhoehen
  return
# end recalculate()

def print_status():
  print "%4d |" % seconds,
  for p in processes:
    if p==active: st="*"
    else: st=" "
    [start,maxtime,prio,used] = processes[p]
    if (start <= seconds) and (used < maxtime):
      print "%2d/%2d %3d %s|" % (used, maxtime, prio, st),
    else: print "           |",
  return
# end print_status()
```


Prioritäten-Scheduler-Implementierung

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

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

print
```

```
# Statistik ausgeben

print

for p in processes:
  [start,maxtime,prio,used] = \
    processes[p]
  st = ""  # leerer String
  print "Prozess %1d:" % p,
  for i in range(0,runtime):
    if   start>i:             st += " "
    elif activity_log[i]==p: st += "x"
    else:                     st += "-"
  print st

# end main()
```


Prioritäten-Scheduler-Implementierung

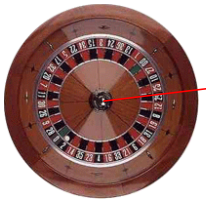
```
> ./prio-sched.py
Zeit | Prozess  1 | Prozess  2 | Prozess  3 | Prozess  4 |
   0 |  0/20 100 *|            |            |            | -> Scheduler aktiviert P. 1
   1 |  1/20 100 *|            |            |            | -> Scheduler aktiviert P. 1
   2 |  2/20 100 *|            |            |            | -> Scheduler aktiviert P. 1
   3 |  3/20 100 *|            |            |            | -> Scheduler aktiviert P. 1
   4 |  4/20 100 *|            |            |            | -> Scheduler aktiviert P. 1
   5 |  5/20 100  |  0/10  50 *|            |            | -> Scheduler aktiviert P. 2
   6 |  5/20 100  |  1/10  50 *|  0/30 100  |            | -> Scheduler aktiviert P. 2
   7 |  5/20 100  |  2/10  50  |  0/30 100  |  0/ 2  10 *| -> Scheduler aktiviert P. 4
   8 |  5/20 100  |  2/10  50  |  0/30 100  |  1/ 2  10 *| -> Scheduler aktiviert P. 4
   9 |  5/20 100  |  2/10  50 *|  0/30 100  |            | -> Scheduler aktiviert P. 2
  10 |  5/20 100  |  3/10  50 *|  0/30 100  |            | -> Scheduler aktiviert P. 2
...
  15 |  5/20 100  |  8/10  50 *|  0/30 100  |            | -> Scheduler aktiviert P. 2
  16 |  5/20 100  |  9/10  50 *|  0/30 100  |            | -> Scheduler aktiviert P. 2
  17 |  5/20 100 *|            |  0/30 100  |            | -> Scheduler aktiviert P. 1
  18 |  6/20 100 *|            |  0/30 100  |            | -> Scheduler aktiviert P. 1
...
  30 | 18/20 100 *|            |  0/30 100  |            | -> Scheduler aktiviert P. 1
  31 | 19/20 100 *|            |  0/30 100  |            | -> Scheduler aktiviert P. 1
  32 |            |            |  0/30 100 *|            | -> Scheduler aktiviert P. 3
  33 |            |            |  1/30 100 *|            | -> Scheduler aktiviert P. 3
...

Prozess 1: xxxxx-----------xxxxxxxxxxxxxxx--------
Prozess 2:      xx--xxxxxxxx-----------------------
Prozess 3:       -------------------------xxxxxxxx
Prozess 4:         xx-------------------------------
```

Lotterie-Scheduler (1)

- Idee: Prozesse erhalten „Lotterie-Lose“ für die Verlosung von Ressourcen
- Scheduler zieht ein Los und lässt den Prozess rechnen, der das Los besitzt
- Priorisierung: Einige Prozesse erhalten mehr Lose als andere

Lotterie-Scheduler (2)



Scheduler zieht Los Nr. 5

Prozess 1
Lose 1,2,3,4

Prozess 2
Lose 5,6

Prozess 3
Lose 7,8,9

Prozess 4
Los 10

Lotterie-Scheduler (3)

- Gruppenbildung und Los-Austausch:
  - Zusammenarbeit Client / Server
  - Client stellt Anfrage an Server, gibt ihm seine Lose und blockiert
  - Nach Bearbeitung gibt Server die Lose an den Client zurück und weckt ihn auf
  - Keine Clients vorhanden?
    ➞ Server erhält keine Lose, rechnet nie

Lotterie-Scheduler (4)

- Aufteilung der Rechenzeit nur statistisch korrekt
- In konkreten Situationen verschieden lange Wartezeiten möglich
- Je länger mehrere Prozesse laufen, desto besser ist erwartete CPU-Aufteilung

Fair-Share Scheduling (1)

- Idee: CPU-Zeiten nicht auf Prozessbasis, sondern auf Applikations- oder User-Basis aufteilen
- Normale Scheduler betrachten jeden Prozess oder Thread separat
- Prozesse / Threads gruppieren
- Jeder Computer-Benutzer erhält seinen „fairen Anteil“ (*fair share*) an Rechenzeit
- wird in einigen Unix-Systemen benutzt

Fair-Share Scheduling (2)

Algorithmus von G. Henry (1984):

Prozess j in Gruppe k

$$\left.\begin{aligned} CPU_j(i) &= \frac{CPU_j(i-1)}{2} \\ GCPU_k(i) &= \frac{GCPU_k(i-1)}{2} \end{aligned}\right\} \quad P_j(i) = Base_j + \frac{CPU_j(i)}{2} + \frac{GCPU_k(i)}{4\,W_k}$$

- $CPU_j(i)$ Maß für CPU-Nutzung durch Prozess j in Intervall i
- $GCPU_k(i)$ Maß für CPU-Nutzung durch Gruppe k in Intervall i
- $P_j(i)$ Priorität von Prozess j am Anfang von Intervall i (kleiner Wert = hohe Priorität)
- $Base_j$ Basispriorität von Prozess j
- W_k Gewicht für Gruppe k; $0 < W_k \le 1$; $\sum_k W_k = 1$

Fair-Share Scheduling (3)

Zeit	GRUPPE 1 – A: Priorität	Prozess-CPU-Nutzung	Gruppen-CPU-Nutzung	GRUPPE 2 – B: Priorität	Prozess-CPU-Nutzung	Gruppen-CPU-Nutzung	GRUPPE 2 – C: Priorität	Prozess-CPU-Nutzung	Gruppen-CPU-Nutzung
0	60	0 1 2 ... 60	0 1 2 ... 60	60	0	0	60	0	0
1	90	30	30	60	0 1 2 ... 60	0 1 2 ... 60	60	0	0
2	74	15 16 17 ... 75	15 16 17 ... 75	90	30	30	75	0	30

Fair-Share Scheduling (4)

Zeit	GRUPPE 1 – A: Priorität	Prozess-CPU-Nutzung	Gruppen-CPU-Nutzung	GRUPPE 2 – B: Priorität	Prozess-CPU-Nutzung	Gruppen-CPU-Nutzung	GRUPPE 2 – C: Priorität	Prozess-CPU-Nutzung	Gruppen-CPU-Nutzung
3	96	37	37	74	15	15 16 17 ... 75	67	0 1 2 ... 60	15 16 17 ... 75
4	78	18 19 20 ... 78	18 19 20 ... 78	81	7	37	93	30	37
5	98	39	39	70	3	18	76	15	18

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-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)