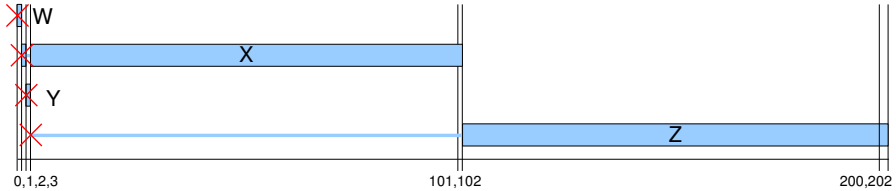
Scheduling (2)

SRT-Beispiel

Altes FCFS-Beispiel: SRT unterbricht jetzt X:
Denn Y kommt zwar später, ist aber kürzer

Prozess	Ankunftzeit	Service Time T_s (Rechenzeit)	Startzeit	Endzeit	Turnaround T_r (Endzeit-Ankunftzeit)	T_r/T_s
W	0	1	0	1	1	1,00
X (1)	1	100	1	2 (*)		
Y	2	1	2	3	1	1,00
X (2)			3	102	102-1=101	1,01
Z	3	100	102	202	199	1,99



Shortest Remaining Time (SRT)

- Ähnelt SJF, aber:
- präemptiv (mit Unterbrechungen)
- Regelmäßig Neuberechnung, wie viel Restzeit die Prozesse noch benötigen werden
- Scheduler prüft Reihenfolge immer, wenn ein neuer Job erzeugt wird
- Für kürzeren (auch neuen) Job wird der aktive unterbrochen
- Wie bei SJF gute Laufzeitprognose nötig

Scheduler für interaktive Systeme

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)

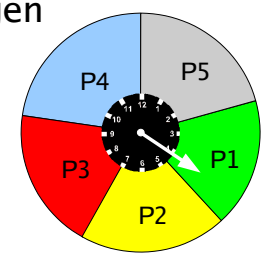
Interaktive Systeme

Scheduling-Verfahren für interaktive Systeme

- Round Robin
- Prioritäten-Scheduler
- Lotterie-Scheduler

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

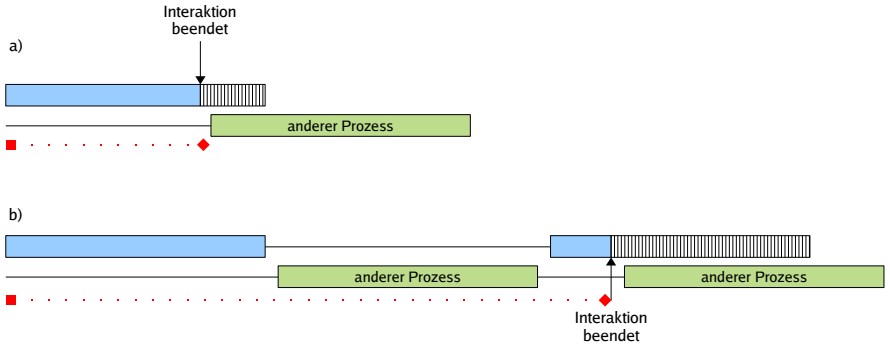


Round Robin (2)

- 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)

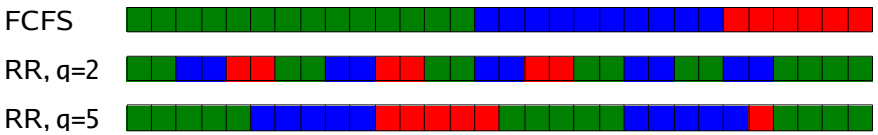
- Oft: Quantum *q* etwas größer als typische Zeit, die das Bearbeiten einer Interaktion benötigt



Round-Robin-Beispiel

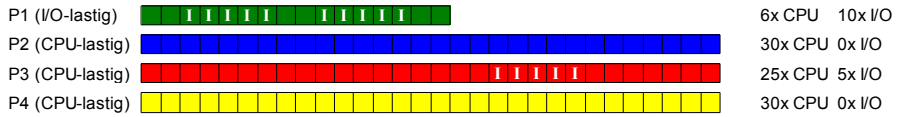
Szenario: Drei Prozesse

- FCFS (einfache Warteschlange, keine Unterbrechung)
- Round Robin mit Quantum 2
- Round Robin mit Quantum 5

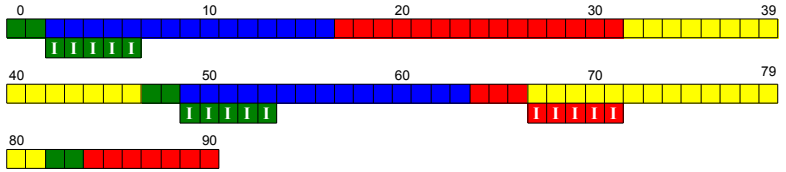


Round Robin: I/O- vs. CPU-lastig

Idealer Verlauf (wenn jeder Prozess exklusiv läuft)



Ausführreihenfolge mit Round Robin, Zeitquantum 15:



Prozess	CPU-Zeit	I/O-Zeit	Summe	Laufzeit	Wartezeit *)
P1	6	10	16	84	68
P2	30	0	30	64	34
P3	25	5	30	91	61
P4	30	0	30	82	52

*) im Zustand *bereit*, nicht *blockiert*!

Virtual Round Robin (1)

Beobachtung:

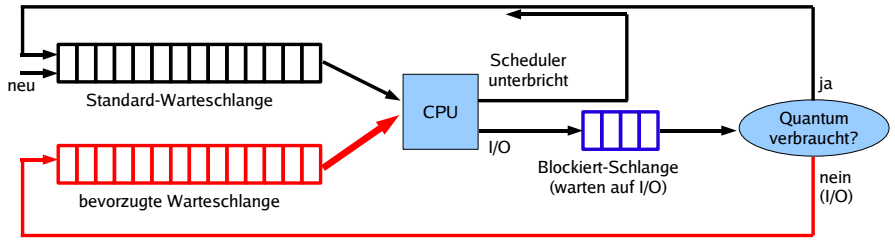
- Round Robin unfair gegenüber I/O-lastigen Prozessen:
  - CPU-lastige nutzen ganzes Quantum,
  - I/O-lastige nur einen Bruchteil

Lösungsvorschlag:

- Idee: Nicht verbrauchten Quantum-Teil als „Guthaben“ des Prozesses merken
- Sobald blockierter Prozess wieder bereit ist (I/O-Ergebnis da): Restguthaben sofort aufbrauchen

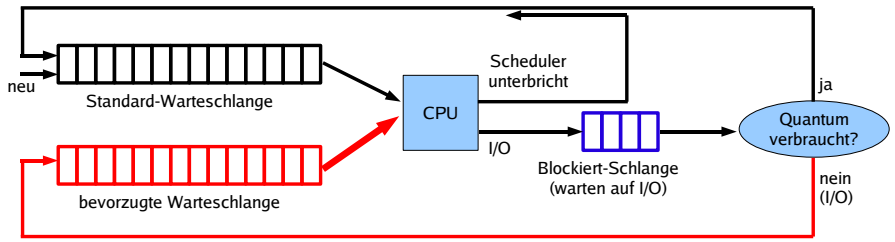
Virtual Round Robin (2)

- 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, bei Blockieren in Zusatzwarteschlange stecken



Virtual Round Robin (3)

- 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)

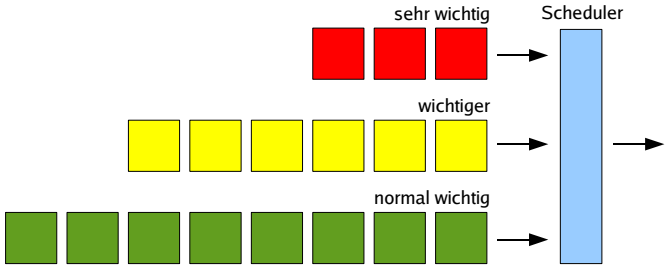


Prioritäten-Scheduler (1)

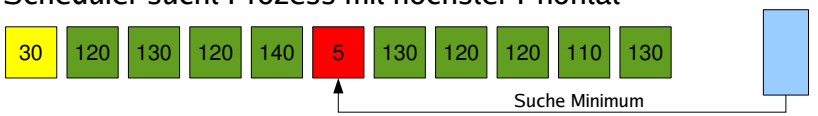
- Idee:
  - a) Prozesse in Prioritätsklassen einteilen oder
  - b) jedem Prozess einen Prioritätswert zuordnen
- Scheduler bevorzugt Prozesse mit hoher Prior.
- Priorität
  - bei Prozesserzeugung fest vergeben
  - oder vom Scheduler regelmäßig neu berechnen lassen
- Scheduling kooperativ oder präemptiv

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 (4)

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 (5)

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 (6)

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 (7)

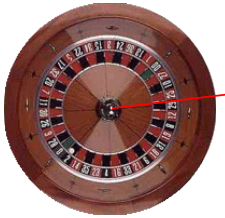
Verschiedene Quantenlängen

- 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

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

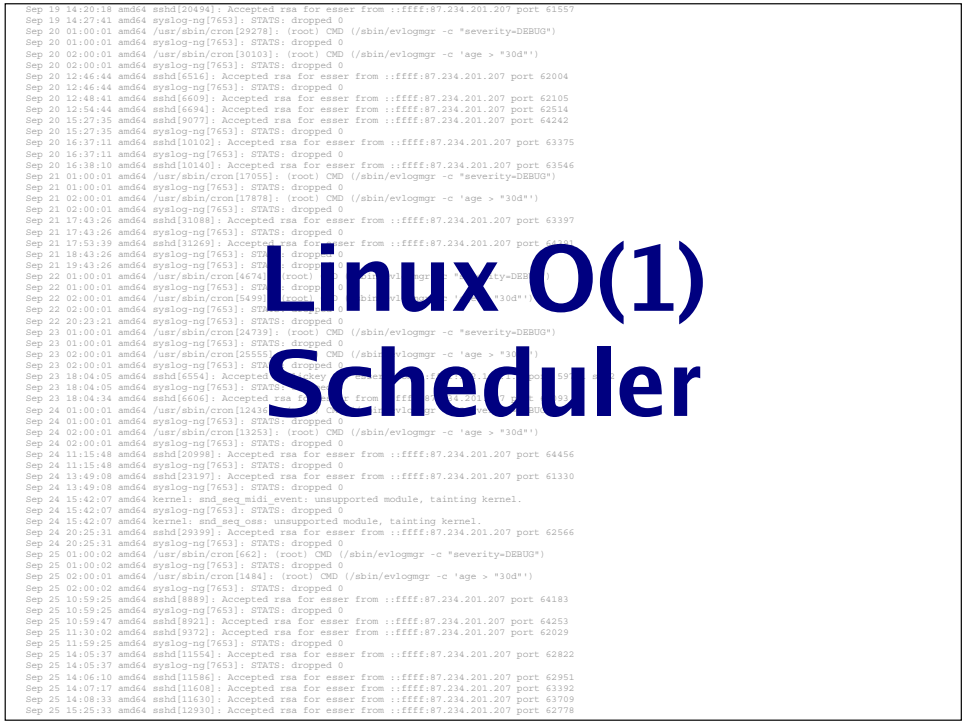
Prozess 4
Los 10

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

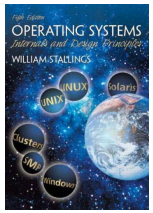
Lotterie-Scheduler (4)

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



Scheduling auf Multi-CPU-Systemen

- Multitasking auf einzelnen CPUs (oder nicht?)
- CPUs gleich-behandeln oder Master/Slaves?
- Zuordnung Prozess ↔ CPU: fest/variabel?
- BS-Instanz auf jeder CPU (was passiert, wenn zwei Scheduler denselben Prozess auswählen?)
- Gang Scheduling
- Dynamisches Scheduling



Literatur: William Stallings, „Operating Systems – Internals and Design Principles“, Kapitel 10

Linux O(1) Scheduler (1)

- Mit Linux-Kernel 2.6 (bis 2.6.22): neuer Scheduler, der Probleme des alten 2.4er Schedulers behebt:
  - Schedule-Zeit direkt abhängig von Anzahl der Prozesse, O(n)
    -> schlechte Performance bei sehr vielen Prozessen
  - schlechte Performance auf SMP-Maschinen

Linux O(1) Scheduler (2)

Ursachen (Kernel 2.4)

- Eine gemeinsame Warteschlange für alle Prozesse auf allen CPUs; darin keine Sortierung
- Scheduler muss ganze Schlange durchsuchen, um richtigen Prozess zu finden
- Eine einzige Sperre für die Runqueue
  ⇒ Zugriff einer CPU auf diese Warteschlange blockiert alle übrigen CPUs
- Ergebnis: Schedule-Aktionen sehr aufwendig

Linux O(1) Scheduler (3)

Kernel 2.4

- Prozesse nicht an CPU gebunden, Zuordnung eher zufällig
  -> häufige CPU-Wechsel eines Prozesses
  -> CPU-Caches werden schlecht genutzt

Linux O(1) Scheduler (4)

Kernel 2.6: neuer O(1) Scheduler mit folgenden Features:

- O(1) Scheduler: Zeit, die der Scheduler für die Auswahl des nächsten Prozesses (für eine CPU) braucht, ist konstant – unabhängig von der Anzahl der Prozesse
- CPUs blockieren sich nicht gegenseitig bei gleichzeitigen Schedule-Entscheidungen
- Load-Balancer verteilt Rechenlast gleichmäßig auf mehrere CPUs

Linux O(1) Scheduler (5)

- Für jede CPU eine separate Warteschlange
- 140 Prioritätslevel,
  kleiner Wert = hohe Priorität:
  - 1-100: Realtime-Prozesse (MAX_RT_PRIO=100)
  - 101-140: Normale Prozesse (MAX_PRIO=140)
- Normale Tasks
  - haben Nice-Wert n ($-19 \leq n \leq 20$),
  - Prio = MAX_RT_PRIO + n + 20,
  - erhalten Zeitquantum

Linux O(1) Scheduler (6)

- Echzeit-Tasks
  - statische Priorität
  - zwei Klassen:
    - FIFO (ohne Unterbrechungen) und
    - Round Robin (mit Zeitquanten)
- Interaktivitätsschätzer:
  prüft, ob ein Prozess interaktiv ist – wenn ja, erhält er eine höhere Priorität
  (nur für normale Prozesse, nicht Echtzeit)
- Für jede CPU und jede Priorität eine Warteschlange (also 140 Listen pro CPU)!

Linux O(1) Scheduler (7)

Nächsten Prozess finden ist sehr einfach:

- Jede CPU muss nur in ihrer privaten Prozessliste suchen
- Bitmap speichert, welche (der 140) Queues leer sind – Suche der Form „1. Bitmap-Feld mit Wert 1“ geht schnell
- Innerhalb der so gefundenen Liste einfach den ersten Prozess wählen
- Suchoperation hängt zwar „von 140“ ab, aber nicht von der Anzahl der Prozesse -> O(1)

Linux O(1) Scheduler (8)

Zusätzlich zu Runqueue gibt es eine „Expired Runqueue“

- aktiver Prozess, dessen Quantum ausläuft, wird unterbrochen und in die Expired Queue verschoben
- beim Verschieben berechnet der Scheduler Quantum und Priorität für diesen Prozess neu (sortiert ihn also evtl. auf eine andere Priorititätsstufe ein).
- Ist die Runqueue komplett leer, werden Runqueue und Expired Runqueue vertauscht

Linux O(1) Scheduler (9)

Interaktivitätsschätzer

- Scheduler versucht zu erkennen, ob Prozesse I/O- oder CPU-lastig sind
- Metrik: Verhältnis Rechenzeit zu (I/O-) Wartezeit
- Scheduler
  - belohnt I/O-lastige Prozesse
  - bestraft CPU-lastige Prozesse

  bis zu +/- 5 Punkte bei Prior.-Berechnung

Linux O(1) Scheduler (10)

Load Balancer

- Eigentlich: CPU-Wechsel vermeiden, da CPU-Cache unbrauchbar wird
- Andererseits: CPUs, die längere Zeit idle sind, sind noch schlimmer
- Alle 200 ms prüft eine CPU, ob die Lastverteilung ungleichmäßig ist; wenn ja, werden die Prozesse neu verteilt
- Problem: Behandlung von HyperThreading-CPUs mit virtuellen CPUs

Performance Linux 2.4 / 2.6

Hackbench: bis zu 200 Client/Server-Prozesse

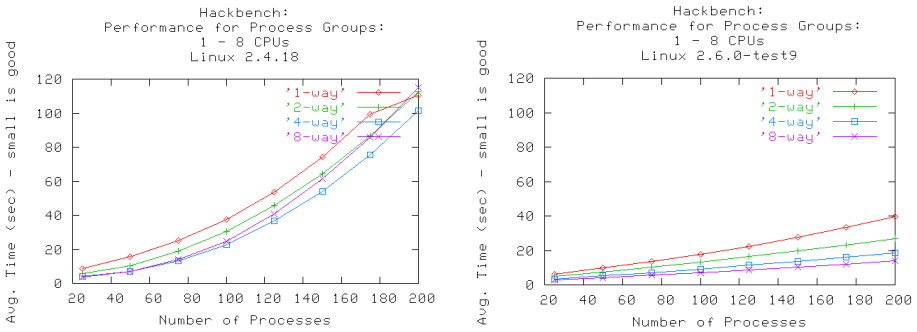


Bild: http://developer.osdl.org/craiger/hackbench/

Completely Fair Scheduler

Seit Linux 2.6.23:

- schon wieder ein neuer Scheduler (CFS)
- speichert zu jedem Prozess die bereits vergangene Wartezeit auf die CPU (in Nanosekunden)
- wer am längsten wartet, kommt dran (hierfür Suche in Binärbaum nötig – nicht mehr O(1); trotzdem schnell genug)
- *http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt*