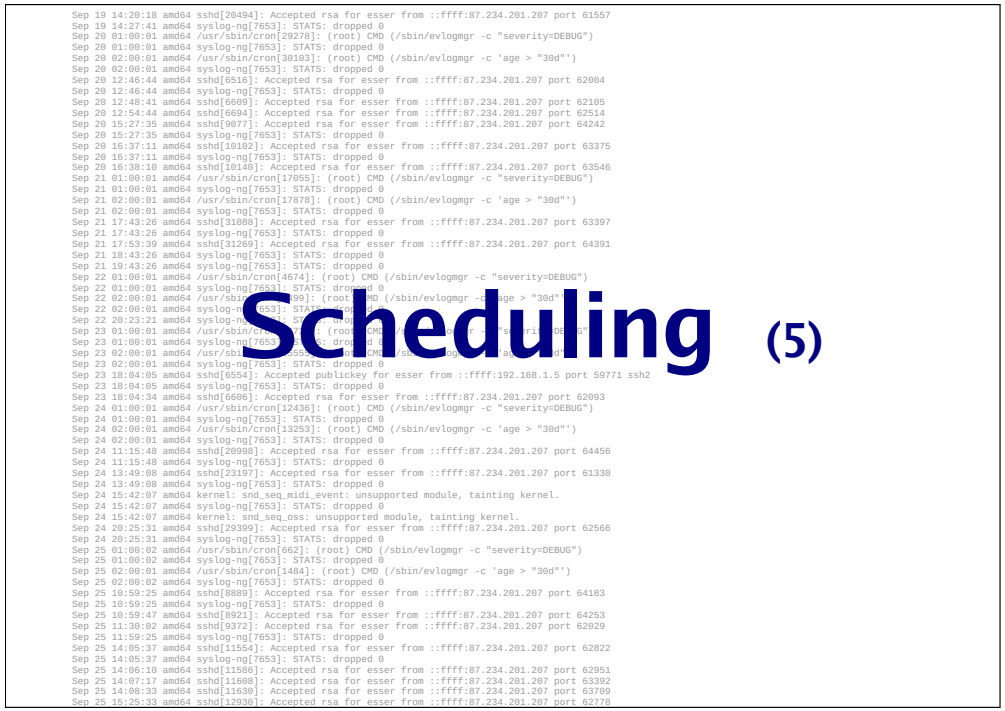


Lösungen Rechner-Praktikum

Lösung zu Aufgabe 2

hello-world-fork.py

```
#!/usr/bin/python

import os

print "Hello World"
chid=os.fork()
if chid == 0:          # Kind-Prozess
  os.execl ("/bin/ls","ls","/tmp")
else:                  # Vater-Prozess
  os.wait()
  print "Das war's"
```


Lösungen Rechner-Praktikum

2. Prozess mit fork() erzeugen und externes Programm starten

Vollziehen Sie ein Beispiel aus der Vorlesung nach: Öffnen Sie einen Editor, in dem Sie die neue Datei `hello-world-fork.py` erzeugen.

- Ihr Python-Programm soll zunächst eine „Hello-World"-Botschaft ausgeben und dann mit `fork()` einen Child-Prozess erzeugen;
- der Child-Prozess lädt mit `execl()` das Programm `/bin/ls` und führt es mit dem Parameter `/tmp` aus (um das Inhaltsverzeichnis von `/tmp` auszugeben).
- Währenddessen wartet der Parent-Prozess darauf, dass das Kind seine Arbeit beendet hat. Ist das der Fall, schreibt es noch „Das war's" auf die Konsole.

Das Programm sollte, wenn Sie es mit

```
$ python hello-world-fork.py
```

starten, also eine Ausgabe der Form

```
Hello World
datei1   datei2    datei3    datei4
datei5   datei6    datei7
Das war's
```

erzeugen.

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 keiner 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)

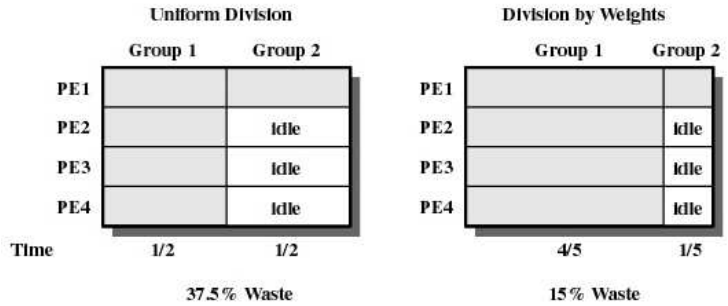
Nachteile des Load Sharing

- 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

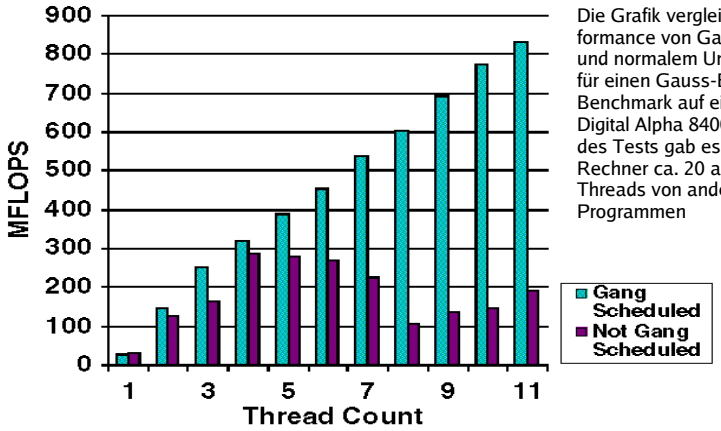
- Simultanes Scheduling von Threads, die alle zu einem Prozess gehören
- Nützlich für Anwendungen, in denen die Performance stark sinkt, sobald ein beliebiger Programmteil nicht läuft
- Threads müssen sich oft synchronisieren

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)

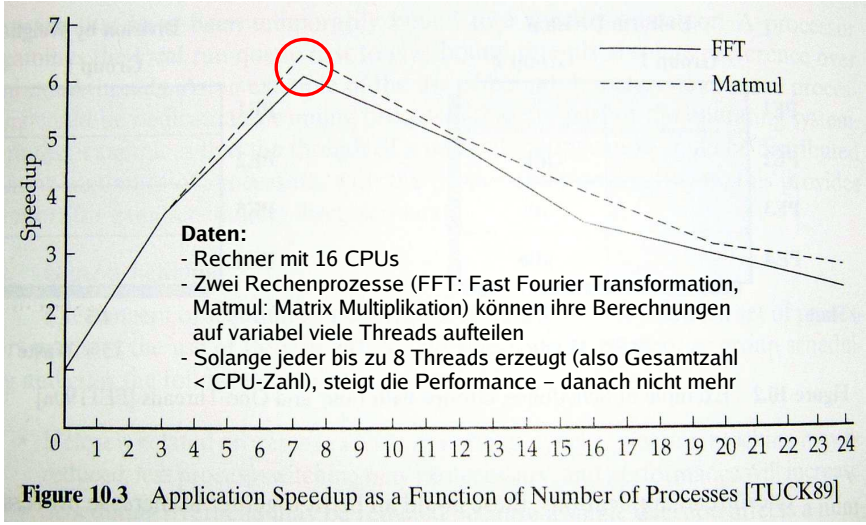


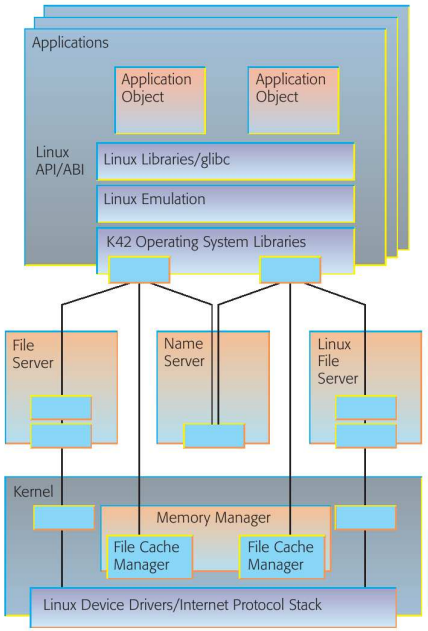
Figure 10.3 Application Speedup as a Function of Number of Processes [TUCK89]

Dynamisches Scheduling

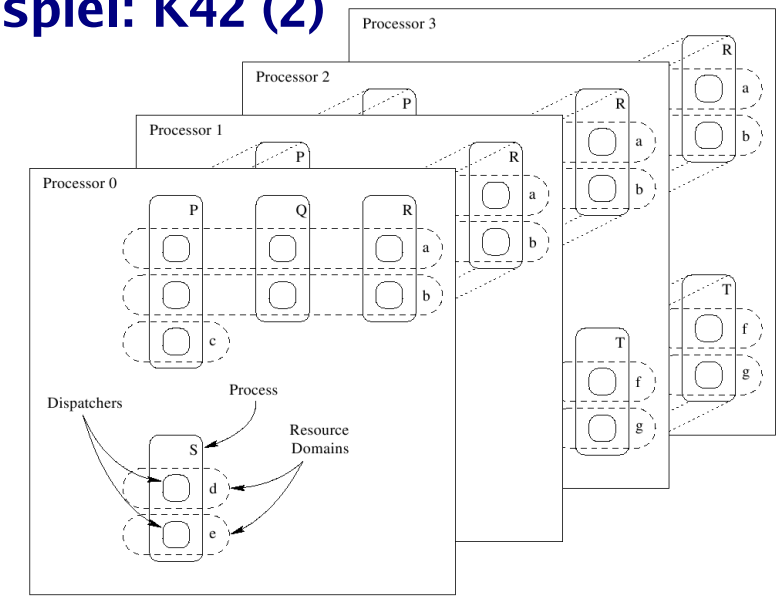
- Anzahl der Threads eines Prozesses dynamisch verändern: Prozess erzeugt neue Threads und beendet sie – und zwar *auf Basis der Ressourcen*, die das Betriebssystem bereitstellt
- Betriebssystem passt die Last an, um die CPU-Nutzung zu optimieren
  - Untätigen CPUs Threads zuordnen
  - Neue Prozesse 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

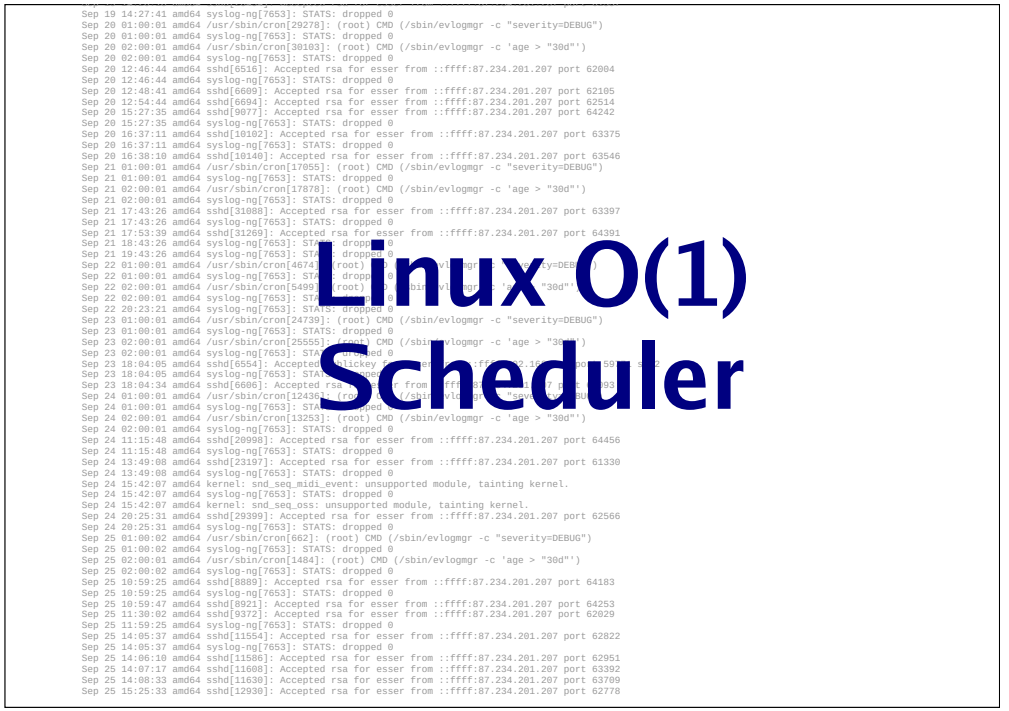


Beispiel: K42 (2)



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

Linux O(1) Scheduler (1)

- Mit Linux-Kernel 2.6:
  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

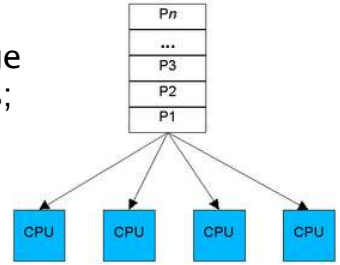


Bild: Linux Journal, http://www.linuxjournal.com/node/7178/

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)

Linux O(1) Scheduler (7)

Für jede CPU und jede Priorität eine Warteschlange (also 140 Listen pro CPU)!

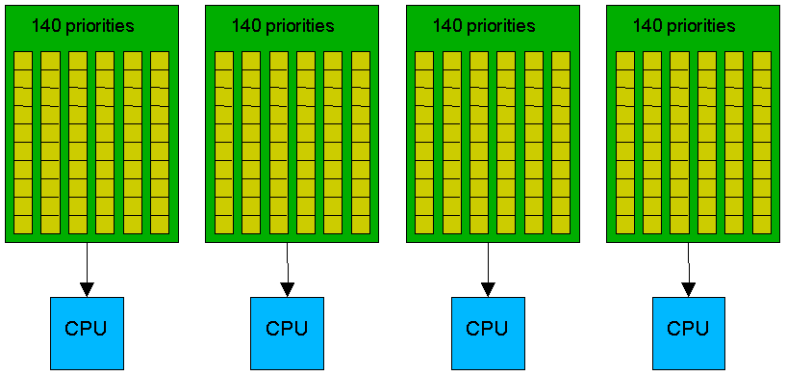


Bild: Linux Journal, http://www.linuxjournal.com/node/7178/

Linux O(1) Scheduler (8)

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 (9)

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 Prioritätsstufe ein).
- Ist die Runqueue komplett leer, werden Runqueue und Expired Runqueue vertauscht

Linux O(1) Scheduler (10)

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 (11)

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

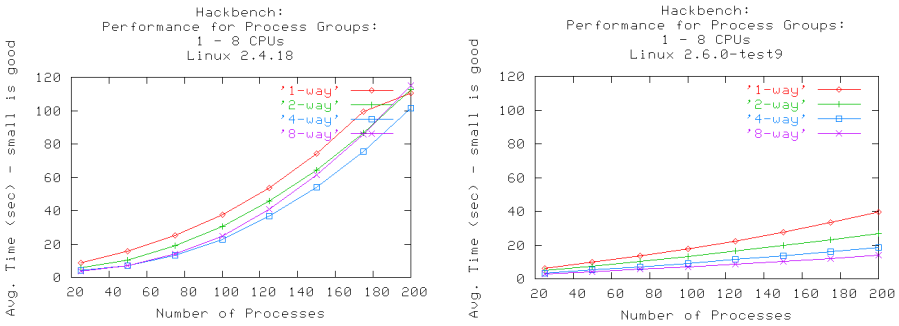


Bild: http://developer.osdl.org/craiger/hackbench/