Scheduling-Verfahren für Mehrbenutzer-Systeme

Klaus Kusche, Juni 2012

Inhalt

- Einleitung & Begriffe
- Ziele & Voraussetzungen
- Das Round-Robin-Verfahren ...
- ... und seine Probleme
- Die Scheduler in Windows und Linux
- (Tücken aktueller Hardware)

Der „Scheduler“

(übersetzt in etwa **„Arbeitsplaner“**)

… ist derjenige *Teil des Betriebssystems*
der für die *Verwaltung der CPU’s* zuständig ist:

Der Scheduler bestimmt die

zeitliche Zuteilung der laufwilligen „Programme“

zu einzelnen CPU-Cores.

(kritisch, wenn *weniger CPU-Cores* vorhanden sind
als *Programme gleichzeitig laufen* wollen!)

Anmerkung:
Es gibt auch **Disk- und Netzwerk-Scheduler**.
Sie werden hier nicht behandelt!

Die Anfänge

Bevor es interaktive PC's, Mehrbenutzersysteme usw. gab:

Batchverarbeitung

„First come - first served“:

Ein Programm nach dem anderen
wurde allein und von Anfang bis Ende abgearbeitet.

==> Der Computer konnte
nicht mehrere Dinge gleichzeitig tun.

==> Wenn ein Programm *auf I/O wartete*,
stand die (teure!) *CPU leer*.

Was läuft alles „gleichzeitig“?

- Mehrere *gleichzeitig angemeldete Benutzer*
  (z.B. via Remote Desktop).
- Mehrere *gleichzeitig aktive Programme* eines Benutzers
  (Mail-Check, Media Player, ...).
- *Mehrere Threads* innerhalb eines *einzelnen Programms*
  (mehrere Flash-Plugins, Photoshop Filter, ...).
- *Hintergrund-Programme*
  (Virenscanner, Indexer für die Suche, Druckspooler,
  Windows Update, File-Server, Web-Server, Backup, ...).
- *Betriebssystem-Threads*
  (Page Swapout, Disk Cache Flush, Disk Read-Ahead, ...)

Begriffsverwirrung „Programm“...

- **Prozess** ... „gestartetes *Programm*“:

  Ein Prozess belegt *Speicher*, hat *Rechte*, hat *offene Files*, ...

- **Thread** ... „*Ablauf* innerhalb eines Prozesses“:

  Ein Thread hat nur eine *aktuelle Codestelle* (+ Stack),
  ==> nutzt Speicher, Rechte, Files, ... von seinem Prozess.

  Ein Thread *bekommt CPU zugeteilt*.

  Jeder Prozess enthält *zumindest einen Thread, oft mehrere*
  (es gibt keine Threads ohne Prozess).
  Daneben gibt es *Threads des Betriebssystem-Kerns*.

- **Task** ... ??? (viele verschiedene Definitionen...)

Und der Scheduler?

- … hat *nichts mit „Programmen“* zu tun
  („Programm“ = Datei auf der Platte),
- … sondern teilt einem CPU-Core
  einen **Thread** zur Ausführung zu
- … und bewirkt dadurch ev. *indirekt*
  eine *Umschaltung zwischen Prozessen*
  (wenn der neue Thread zu einem anderen Prozess gehört).

Ein solcher Prozesswechsel ist *aufwändig*
(u.a. Umschaltung der virtuellen Speicherverwaltung)

==> *Überlegt und nicht zu oft machen!*

Status von Threads

- **Running:**

  Hat eine CPU, *rechnet gerade*.

- **Ready:**

  Möchte rechnen, aber muss *auf eine freie CPU warten*.

- **Waiting:**

  Kann gerade nicht weiterrechnen, *möchte gar keine CPU*:

  Wartet auf I/O, einen Timer, einen anderen Thread, …

- **Suspended:**

  Wurde ohne sein Wissen und Zutun „eingefroren“.

Was läuft *ohne* den Scheduler?

Nichts außer den

Interrupt Handlern:

Hardwarebezogene Codestücke (meist Treiber),
die „*sofort*“ ausgeführt werden müssen.

I/O-Baustein, Timer, … löst Interrupt aus (eigene Leitung)
==› Prozessor unterbricht hardwaremäßig die laufende Ausführung,
wechselt in den Kernel-Modus,
startet den zum Interrupt gehörenden Handler,
und macht danach (meist) mit dem unterbrochenen Code weiter.

Ohne Zutun von Software / Betriebssystem / Scheduler!

Aber: Interrupt Handler laufen *extrem kurz* (wenige µs!)
==› Ihr Anteil am CPU-Verhalten ist meist *vernachlässigbar*!

Ziele des Schedulers

- **Fairness:**

  Jedes Programm bekommt *gleichviel CPU*.

  Besser: Einen *„vernünftigen“ Anteil* der CPU,
  abhängig von seiner **„Wichtigkeit“ = Priorität**.

- **Gute Reaktionszeit:**
  - Das System *ruckelt nicht*,
    Anwendungen *reagieren „sofort“ auf Eingaben*.
  - Wichtige Geräte (z.B. CD-Brenner, WebCam, ...)
    werden *rechtzeitig vom jeweiligen Programm bedient*.
- **Effizienz:**
  Der *CPU-Verbrauch* des Schedulers selbst soll *minimal* sein, im Idealfall *unabhängig* von der *Anzahl* der Threads (ev. tausende!) und der Cores.

Ziele nach Programm-Art

- **Echtzeit-Programme** (z.B. für Steuerungen):
  - Absoluter *Vorrang*:
    Garantierte CPU-Zuteilung mit kürzester Reaktionszeit!

  Echtzeit-Programme auf „normalen“ PC’s:
  - *Audio- / Video-Anwendungen*
  - CD-Brennprogramme
- **Interaktive Anwendungen, Netzwerk-Server**:
  - Gute *Reaktionszeit*
  - Bestmögliche *Fairness*
- **Hintergrund-Programme**, ...:
  - Maximale *Effizienz*
  - Minimale *„Störwirkung“* auf andere Anwendungen

Was ist dazu notwendig?

- Die Möglichkeit,
  einem Thread die CPU **jederzeit wegzunehmen**:
  - Wenn er *unfair viel* CPU braucht.
  - Wenn *sofort ein wichtigerer Thread* laufen soll.

- Irgendein **hardware-basierter Zeitgeber**
  zur Messung und Durchsetzung der Fairness:
  - Wer hat *wie viel CPU-Zeit* bekommen?
  - Rechnet das aktuelle Programm
    jetzt schon *länger als ihm zusteht*?

Das „Wegnehmen“ der CPU

- *Früher* (bis Win 95, erste Mac-OS):

  Unmöglich!

  ==> „**Non-preemptive**“ Scheduling

  Man musste warten,
  bis der Thread die CPU *freiwillig hergab*!

- *Heute*, auf „großen“ Systemen schon seit ~1965:

  Geht jederzeit!

  ==> „**Preemptive**“ Scheduling

  Das Betriebssystem kann einen Thread
  jederzeit *unterbrechen* und „schlafen legen“.

Wirklich jederzeit? (1)

Nein, nur im *„User Mode“*
(während „normaler“ Programm-Ausführung)!

Innerhalb von Betriebssystem-Aufrufen

- … gibt es immer kurze
  nicht unterbrechbare Codestücke.
- … ist bei einigen Systemen
  gar kein Threadwechsel möglich.

Meist egal, aber unter extremen Bedingungen
einige 100 ms Reaktionszeit-Verzögerung!

Wirklich jederzeit? (2)

Bei Linux konfigurierbar:

- **Non-preemptible** (für *Server*):
  Kein Threadwechsel während System Calls.
  Maximaler Durchsatz, aber schlechte Reaktionszeit.
- **Voluntary Preemption** (für *Desktop*):
  Freiwilliger Wechsel an bestimmten Stellen im System Call.
  Mittelweg (typischerweise wenige ms Reaktionszeit).
- **Fully Preemptible** (für *Echtzeit & Multimedia*):
  Auch innerhalb von System Calls *jederzeit Threadwechsel*.
  Hoher Overhead, aber typ. Reaktionszeiten im µs-Bereich.
- … + **Threaded Interrupts** (für *schnelle Echtzeit*):
  Auch *Interrupt Handler* sind *unterbrech- und verdrängbar*.

Die „Zeitscheibe“

= *maximale Zeit*,
die *ein Thread*
die CPU *durchgehend*
(allein) benutzen darf.

Technisch:

Regelmäßiger **Timer Interrupt („System Tick“)**

==> Der *Scheduler wird aufgerufen*
und aktiviert einen *anderen Thread*.

(der Scheduler ist sozusagen der „Interrupt Handler“ des System-Ticks)

Der Timer Interrupt

System-Tick-Takt **hoch**:

==> *„Flüssigeres“ Verhalten* des Systems.

==> *Mehr Overhead* durch den Scheduler
und durch häufigere Taskwechsel.

System-Tick-Takt **niedrig**:

==> *Effizienter, aber „ruckelig“*.

Windows: 60 Hz (16.6 ms) (u.a. wegen Video-Wiedergabe)

Linux: Wählbar:

100 Hz (10 ms)	...	Server
250 oder 300 Hz	...	Desktop
1000 Hz (1 ms)	...	Echtzeit-Systeme, AV-Systeme, ...

Aktivierung des Schedulers

Der Scheduler wird aktiv und wählt neuen Thread wenn:

- **Ein Interrupt auftritt** (I/O fertig, Timer abgelaufen, ...), der einen *wartenden hochprioren Thread „ready“* macht.
- Ein **Interrupt des periodischen Timers** auftritt und die *Zeitscheibe des laufenden Threads zu Ende* ist.
- Der **laufende Thread das Betriebssystem aufruft** und:
  - Ein *I/O* macht, das auf Hardware warten muss.
  - Auf einen *Timer* wartet.
  - Auf eine *Kommunikation* mit anderen Threads wartet.
  - *Freiwillig* die CPU hergibt.
  - Einen *höherprioren Thread* startet oder aufweckt.

Dazwischen kann der Scheduler *nicht „von sich aus“* aktiv werden!

Die letzte zentrale Frage

... und das wesentliche Unterscheidungs-Kriterium verschiedener *Scheduler-Strategien*:

Welcher Thread

wird vom Scheduler

als nächstes aktiviert?

Round Robin Scheduling

(deutsch: „*Zeitscheibenverfahren*“)

Jeder Thread bekommt
reihum nacheinander die CPU …

- … bis seine *Zeitscheibe abgelaufen* ist
- … oder bis er *vorher I/O* macht usw..

In beiden Fällen: Wieder „**hinten anstellen**“!

Alle Zeitscheiben sind *gleich*
==> Das Verfahren ist *fair*!

Technisch: Speicherung aller laufbereiten Threads
in einer *zyklischen Liste* oder einer *Queue* (*FIFO*).

Problem 1: Prioritäten (1)

Klassisches Round Robin kennt **keine Prioritäten**!

Zwei unterschiedliche Bedeutungen von „Priorität“:

- Bei *Echtzeit, Batch- & Hintergrund-Programmen:*

  Absolute Priorität = Vorrang

  Ein hochpriorer Thread hat *absoluten Vorrang*,
  ein niederpriorer Thread bekommt *gar keine CPU mehr*!

- Bei *interaktiven Anwendungen:*

  Relative Priorität = „Anteil am Kuchen“

  Hochpriore bekommen *mehr CPU* (meist exponentiell),
  aber *keiner darf ganz „verhungern“!*

Problem 1: Prioritäten (2)

Implementierung:

- Für **absolute Prioritäten**:

  Meist ein *separater Round-Robin-Ring pro Priorität*.

- Für **relative Prioritäten**: Schwierig!

  „Halbe“ Zeitscheiben sind technisch nicht möglich!

  ==> Niederpriore Threads bekommen *seltener* eine Zeitscheibe.

  ==> Abkehr vom strikten „Reihum“-Prinzip!

Problem 1: Prioritäten (3)

Problem:

- *Echtzeit-Code* wird meist *explizit als solcher gestartet*.
- Bei *nicht-Echtzeit-Code* wird *nicht explizit unterschieden*, ob ein Programm interaktiv oder im Hintergrund läuft.

==> *Wie erkennt und separiert man*

- Reine **Hintergrund-Programme**?
  (absolute Priorität, *bei Hochlast völlig deaktivieren*!)
- **Interaktive Programme**?
  (relative Priorität, *nie ganz verhungern lassen*!)

==> *Heuristiken zur Erkennung*
und unterschiedlichen Behandlung notwendig!

Problem 2: I/O (1)

Viele Programme (vor allem interaktive!)
brauchen immer *nur ganz kurz CPU*,
und machen *dann I/O* (Disk, Maus, Netz, ...).

Round Robin: Nach *jedem I/O* wieder „*ganz hinten anstellen*“!

==› Nur ein I/O & ganz wenig CPU pro Umlauf!

==› Sehr lange Reaktionszeit (zuerst alle anderen)!

==› *Starke Benachteiligung*
von *interaktiven* und von *I/O-intensiven* Programmen
gegenüber reinen CPU-Fressern!

Selber Effekt bei jedem einzelnen Paging / Swapping
==› Starke Benachteiligung Memory-intensiver Programme
gegenüber Programmen mit geringem Speicher-Bedarf!

Problem 2: I/O (2)

==> **Diverse Heuristiken:**

- Bevorzugung *I/O-intensiver Programme*
- Bevorzugung von Programmen
  mit aktueller *Benutzer-Interaktion*

Beispiel:

Jeder Thread bekommt

- *gleich nach einem I/O*
  mehr CPU oder *höhere Priorität* (*sofort wieder CPU*)
  als ihm fairerweise zusteht,
- dann *bei viel reinem CPU-Verbrauch*
  anteilsmäßig *immer weniger*!

Windows (Mac, BSD, ...) Scheduling: Multilevel Feedback Queues

- Eine *First-Come-First-Served-Queue pro Priorität*.
- Die letzte (schlechteste) Queue arbeitet *Round Robin*.
- Ein Thread hat *eine ganze Zeitscheibe durchgerechnet*:

  Abstufung in die *nächstschlechtere* Queue.
- Ein Thread hat *wegen I/O früher die CPU abgegeben*:

  Aufstieg in die *nächstbessere* Queue.
- Ein Thread wartet wegen *Thread-Sync, Sleep* usw.:

  Nachher wieder in *dieselbe* Queue.

==> Bevorzugt I/O, degradiert CPU-Fresser.

Linux Scheduling:
CFS = „Completely Fair Scheduler“

Prinzip „**fair queueing**“ statt Round Robin
(eigentlich aus dem Netzwerk-Scheduling):

- Jeder Thread hat
  - **Sollzeit** (wieviel CPU *hätte ihm fairerweise gehört*)
  - **Istzeit** (wieviel CPU *hat er wirklich verbraucht*)
- Wer bekommt als nächstes die CPU?
  Der Thread mit dem

größten Defizit „Sollzeit - Istzeit“!

Thread hat viel I/O gemacht
==> Thread hat weniger CPU als erlaubt verbraucht
==> Thread hat hohes Defizit, wird bevorzugt!

Linux Group Scheduling

Erweiterung des CFS:

- Einteilung aller Threads in *Gruppen*:

  Typischerweise eine Gruppe *pro Benutzer*.

- *Fairer Anteil* wird

  pro Gruppe statt pro Thread berechnet!

- Threads *innerhalb einer Gruppe*
  müssen sich den *Gruppen-Anteil aufteilen*!

==> *Jeder Benutzer bekommt bei Vollast denselben Anteil*,
egal, ob er nur ein oder viele Programme laufen hat!

Alternatives Linux Scheduling: BFS

Prinzip „**Virtual Deadline**“ statt Round Robin:

- Jeder Thread hat eine „Virtual Deadline“:

  Wann sollte der Thread
  spätestens das nächste Mal drankommen?

  (trickreich berechnet aus CPU-Verbrauch, relativer Priorität, …)

- Wer bekommt als nächstes die CPU?
  Der Thread mit der

  zeitlich knappsten Deadline!

Weiters:

- Eine unabhängige *Queue pro absoluter Priorität*.

Tücken aktueller Hardware

Leider verhält sich eine reale CPU
in der Praxis viel komplizierter…

- **Cache-Lokalität**
- **Hyperthreading** etc.
- **Power Saving**

Cache-Lokalität

L1/L2-Caches sind pro Core,
L3-Caches pro Prozessor:

==> Wenn möglich:
Jeden Thread *immer auf denselben Core*
(zumindest denselben Prozessor) schedulen!

==> Schneller, weil *Code & Daten schon im Cache* sind!
(Worst case: Faktor 3-10 !!!)

Bei fast allen Betriebssystemen
(aus genau diesem Grund!):

Möglichkeit, Threads **explizit**
fix an bestimmte Cores zu **binden**!

Hyperthreading etc.

Hyperthreading = 2 „logische“ Cores pro Hardware-Core

- Logisch: 2 idente, eigenständige und *unabhängige* Cores.
- Technisch: *Teilen* sich denselben Core samt Caches, *rechnen abwechselnd!*

==> *In Summe nur max. 20 % schneller* als einer davon!

==> Threads *verdrängen sich gegenseitig* aus dem Cache!

==> Zuerst alle Hardware-Cores *einfach* belegen, erst wenn notwendig die „Hyperthreading-Brüder“!

==> Auch *Scheduling auf Bruder* nutzt *alte Cache-Inhalte*!

Analog bei aktuellen AMD-Prozessoren:
Je 2 Cores teilen sich FPU, Instruction Decoder, Cache, ...

Power Saving (1)

Bei *Notebooks* usw.:

„**Aufwecken**“ eines Cores / des Prozessors
aus dem Schlafzustand **kostet viel Energie**!

==> Möglichst lange im „Deep Sleep“ lassen!

==> Wenn **wenig / nichts zu tun** ist:

60-300 mal Aufwecken pro Sekunde
nur für den Scheduler (der nichts tut)
ist *sinnlose Verschwendung*!

==> **System Tick nur bei Bedarf** aktivieren, sonst abdrehen!
(Linux: Konfigurierbar, „Tickless System“)

Power Saving (2)

Power Management moderner CPU's:

Mehrere Stufen mit wachsender Strom-Einsparung, aber wachsender Umschalt-Zeit.

Z.B. Intel Core i7 (~100 Seiten kluges Buch):

0 ... active
1 ... core clock off, core voltage low
3 ... L1/L2 cache flushed to L3,
cache clock off, cache voltage low
6 ... core state flushed to L3,
core & cache voltage off
7 ... common units (L3 cache, bus) powered down,
clock PLL stopped

Power Saving (3)

Verschiedene Optimierungsziele:

- Jede **Power-State-Umschaltung**
  kostet *Zeit* (max. einige ms Verzögerung!)
  und *Strom*

  ==› Möglichst selten umschalten!
  ==› *Jeden Core möglichst lang „ein“ oder „aus“ lassen!*

- Aber wenn **alle Cores aus** sind:
  Viel höhere Einsparung
  durch *Abschalten gemeinsamer Teile*: L3 Cache, ...

  ==› Ein einzelner rechnender Core ist ungünstig!
  ==› *Lieber kurz viele ein, dann alle aus!*