— BS //—

Monoprozessor-Scheduling

Überblick

Stöße — *CPU bursts* und *I/O bursts*

- längere Rechen- und Blockadephasen wechseln sich einander ab
  - empirisch ermitteltes, typisches Verhalten vieler (Anwendungs-) Programme
- daraus resultiert die „stoßartige Belegung“ von (Hardware-) Betriebsmitteln:

  CPU burst die aktive Rechenphase eines Programms
  - der Prozessor führt Instruktionen des Programms aus

  I/O burst die durch E/A bedingte Blockadephase eines Programms
  - die Peripherie arbeitet auf Anweisung des Programms
  - das Programm muss die Beendigung der E/A-Operation abwarten
- Häufigkeit und Dauer der E/A-Stöße beeinflussen die Auslastung der CPU

E/A-Stöße vs. CPU-Auslastung

- im Einprogrammbetrieb grenzt Maximierung der CPU-Auslastung an „Hexerei“
  - asynchrone E/A, DMA und ggf. *Interrupts* machen 100 % (fast) möglich
  - ein anhaltender CPU-Stoß existiert nebenläufig zu ggf. mehreren E/A-Stößen
  - das Benutzerprogramm muss jedoch entsprechend formulierbar sein
- Mehrprogrammbetrieb maximiert die CPU-Auslastung „benutzerfreundlich“
  - E/A- und CPU-Stöße verschiedener Programme[1] werden überlappt
  - während der E/A-Stoß für P_x läuft, erfolgt der CPU-Stoß für P_y, P_z, . . .
  - die Maßnahmen sind transparent für die ausgeführten Benutzerprogramme

[1] Besser: Prozesse. Denn es ist leicht möglich, dass dasselbe Programm zugleich mehrfach zur Ausführung gebracht worden ist. Beispielsweise das Programm „gcc“, mehrfach aufgerufen von verschiedenen Benutzern.

Synchrone E/A $\Longleftrightarrow$ Asynchrone E/A

```
⋮
write(1, buffer, read(0, buffer, NBYTES));
⋮
```

Nicht sauber [warum?] — aber „benutzerfreundlich“. Im Vergleich zur asynchronen Variante ist der Programmausschnitt leicht verständlich. Asynchrone E/A ist nur dann brauchbar, wenn nebenläufig zu den E/A-Stößen im Programm sinnvolle Arbeit geleistet werden kann. Das durch das Programm zu lösende Problem muss dazu überhaupt den erforderlichen Spielraum bieten. Dieser Sachverhalt ist dem Problem der (maschinenenahen) Programmierung von Fließbandprozessoren ähnlich.

```
⋮
size = start(0, buffer, NBYTES, READ);
while (!size) {
    /* do some meaningful work */
    size = check(0);
}
done = start(1, buffer, size, WRITE);
while (!done) {
    /* do some meaningful work */
    done = check(1);
}
⋮
```


Programmfäden — Hilfsmittel zur Leistungsoptimierung

- strukturfördernde Maßnahme zur technischen Repräsentation von CPU-Stößen
  - der CPU-Stoß von Faden$_x$ erfolgt nebenläufig zum E/A-Stoß von Faden$_y$
  - ggf. werden die CPU-Stöße weiterer Fäden zum „Auffüllen“ genutzt
- die Auslastung wird verbessert durch die Überlappung der verschiedenen Stöße
  - in einem Monoprozessorsystem kann immer nur ein CPU-Stoß aktiv sein
  - parallel[2] dazu können jedoch viele E/A-Stöße (anderer Fäden) laufen
  - als Folge sind CPU und E/A-Geräte andauernd mit Arbeit beschäftigt
- bezogen auf eine CPU sind die ggf. vielen Programmfäden aber zu serialisieren

[2] Ein E/A-Stoß ist zu einem Zeitpunkt zwar genau einem Programmfaden zugeordnet, er wird jedoch von einem „eigenen“ Prozessor ausgeführt — dem E/A-Gerät. Dadurch ergibt sich echte Parallelität bezogen auf Stöße.

Zwangsserialisierung von Programmfäden

- die absolute Ausführungsdauer später „eintreffender“ Fäden verlängert sich:
  - Ausgangspunkt seien n Fäden mit gleichlanger Bearbeitungsdauer k
  - der erste Faden wird um die Zeitdauer 0 verzögert
  - der zweite Faden um die Zeitdauer k, der i-te Faden um $(i-1)\cdot k$
  - der letzte von n Fäden wird verzögert um $(n-1)\cdot k$

$$\frac{1}{n}\cdot\sum_{i=1}^{n}(i-1)\cdot k=\frac{n-1}{2}\cdot k$$

- die mittlere Verzögerung wächst (subjektiv) proportional mit der Fadenanzahl

Prozessorzuordnung ist Aufgabe eines jeden Fadens

In der Praxis wirkt der für viele Anwendungen typische Wechsel zwischen CPU- und I/O-Bursts subjektiv einer proportional mit der Thread-Anzahl wachsenden Verzögerung entgegen. [1]

- die Prozessorzuordnung ist Teil des CPU-Stoßes des blockierenden Fadens
  - sie erfolgt { nachdem der Faden { seinen E/A-Stoß ermöglicht hat / Betriebsmittelzuteilung anforderte / logisch blockiert worden ist } / jedoch bevor er die Kontrolle über die CPU wirklich abgibt }
- ein Faden leistet sinnvolle Arbeit auf Systemebene, nicht nur auf Benutzerebene

Dauer von CPU- und E/A-Stößen

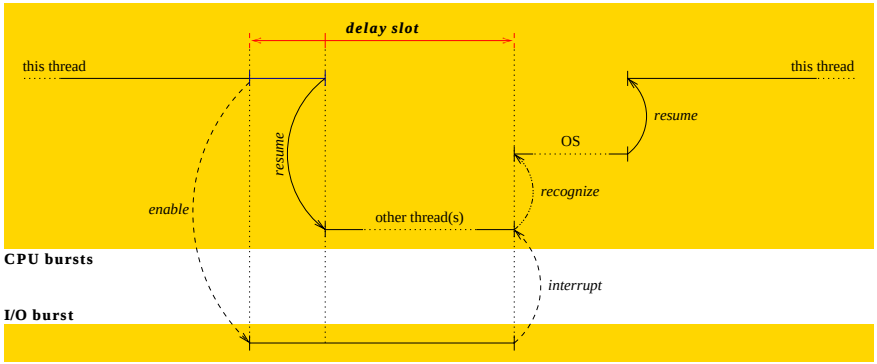
- CPU-Stöße sind verhältnismäßig kurz[3] — insbesondere im Dialogbetrieb
  - Messungen ihrer Dauer zeigen schnell eine Häufung bei 2 ms
  - über 90 % der CPU-Stöße sind kürzer als 8 ms [1]
- E/A-Stöße sind i.A. im Vergleich zu CPU-Stößen meist erheblich länger
  - z.B. dauern Plattenzugriffe (heute) im Mittel etwa 5 ms
  - zwischen zwei Tastatureingaben können Sekunden bis Minuten vergehen
  - die Dauer von E/A-Stößen ist abhängig vom „externen Prozess“
- die Stoßlängen sind abhängig von der Hardware, Software und „Humanware“

[3] Im Falle rechenintensiver Programme gilt dies nicht unbedingt. So lebt z.B. *high-performance computing* von (sehr) langen CPU-Stößen.

Subjektive Empfindung der Fadenverzögerung

- die mittlere Verzögerung eines Fadens ergibt sich zu: $\frac{n-1}{2} \cdot t_{Burst}$
  - mit t_{Burst} gleich der mittleren Dauer eines CPU-Stoßes
  - bei genügend vielen asynchron ablaufenden E/A-Operationen
- zwischen CPU- und E/A-Stößen besteht eine große Zeitdiskrepanz
  - die Verzögerung durch E/A-Operationen ist dominant
  - der proportionale Verzögerungsfaktor bleibt weitestgehend verborgen
  - er greift erst ab einer bestimmten Anzahl von Programmfäden
  - viele Anwendungen/Benutzer nehmen die Verzögerung daher nicht wahr
- die „Totzeit“ bei E/A-Stößen wird für CPU-Stöße laufbereiter Fäden genutzt

Parallelität auf Stoßebene



Monopolisierung der CPU durch Programmfäden

- mit erfolgter Prozessorzuteilung gewinnen Fäden die Kontrolle über die CPU
  - die CPU führt nur noch Anweisungen aus, die das Benutzerprogramm vorgibt
- das Betriebssystem kann die Kontrolle nur bedingt zurückgewinnen
  - die Fäden müssten $\left\{ \begin{array}{c} \text{synchrone} \\ \text{asynchrone} \end{array} \right\}$ Programmunterbrechungen erfahren
- synchrone Programmunterbrechungen sind ein eher schwaches Instrument
  - die Fäden müssten sich kooperativ dem Betriebssystem gegenüber erweisen
  - „böswillige“ Programme können schnell die Kooperative gefährden/auflösen

Vermeidung eines Ausführungsmonopols

- Betriebssysteme setzen auf asynchrone Programmunterbrechungen (*Interrupts*)

  sporadische Unterbrechung bei Beendigung eines E/A-Stoßes
  - der E/A-Stoß musste vorher von einem Faden erst ermöglicht werden
  - wann und ob überhaupt ein E/A-Stoß ausgelöst wird ist ungewiss
  - ebenso ungewiss ist die E/A-Stoßdauer und damit der Interrupt-Zeitpunkt

  periodische Unterbrechung durch Einsatz eines Zeitgebers
  - der Zeitgeber wird je nach Bedarf vom Betriebssystem programmiert
  - er sorgt in der Regel für zyklische Unterbrechungen (*timer interrupts*)
  - mit Ablauf der vorgegebenen Zeit wird das Betriebssystem reaktiviert

- Zugriffe auf Zeitgeber und Interrupt-Maske sind privilegierte Operationen **!**

Grundsätzliche Verfahren

- FCFS ..... *gerecht*
- {,V}RR ..... *zeitscheibenbasiert*
- SPN (SJF), SRTF, HRRN ..... *probabilistisch*
- FB (MLQ, MLFQ) ..... *mehrstufig*

FCFS — *first come, first serve*

- ein einfaches und gerechtes Verfahren: „wer zuerst kommt, mahlt zuerst“
  - Prozesse werden entsprechend der Reihenfolge ihrer Eintreffens abgearbeitet
  - der Scheduling-Algorithmus sortiert Prozesse nach ihren Ankunftszeiten
  - zwischen unterschiedlichen Dringlichkeiten wird nicht unterschieden
  - die Bedienzeiten der Prozesse bleiben unberücksichtigt
  - der laufende Prozesse wird nicht verdrängt
- Prozesse mit $\left\{ \begin{array}{l} \text{langen} \\ \text{kurzen} \end{array} \right\}$ CPU-Stößen werden $\left\{ \begin{array}{l} \text{begünstigt} \\ \text{benachteiligt} \end{array} \right\}$

FCFS — Durchlaufzeit

Prozess	Zeiten					
	Ankunft	Bedienung (T_s)	Start	Ende	Durchlauf (T_q)	T_q/T_s
A	0	1	0	1	1	1.00
B	1	100	1	101	100	1.00
C	2	1	101	102	100	100.00
D	3	100	102	202	199	1.99
Ø					100	26.00

- die normalisierte Duchlaufzeit (T_q/T_s) von C ist vergleichsweise sehr schlecht
- die Verzögerung von C steht im sehr schlechten Verhältnis zur Bedienzeit T_s
- mit dem Problem sind immer kurze Prozesse konfrontiert, die langen folgen

FCFS — Konvoi(d)effekt

- Ausgangspunkt: ein langer CPU-Stoß ist im Gange, mehrere kleine sind bereit

  E/A-Durchsatz ist vergleichsweise niedrig
  - parallel zum ablaufenden CPU-Stoß werden E/A-Stöße beendet
  - die E/A-Geräte bleiben untätitg, da der lange CPU-Stoß noch anhält

  Antwortzeit ist vergleichsweise hoch
  - der lange CPU-Stoß wird von einem (ggf. kurzen) E/A-Stoß abgelöst
  - die anstehenden bereiten kurzen CPU-Stöße kommen schnell zum Ende
    - weitere (ggf. lange) E/A-Stöße wurden ausgelöst
    - der vom langen CPU-Stoß ausgelöste E/A-Stoß ist beendet
  - der lange CPU-Stoß beginnt erneut und verzögert die kurzen CPU-Stöße

- das Verfahren ist suboptimal bei einem Mix von kurzen und langen CPU-Stößen

RR — *round robin*

- verringert die bei FCFS auftretende Benachteiligung kurzer CPU-Stöße
  - Grundlage dafür ist Verdrängung (*preemption*) auf Basis von **Zeitscheiben**
  - ein Zeitgeber bewirkt periodische Unterbrechungen (*time slicing*)
  - die Periodenlänge entspricht typischerweise einer Zeitscheibe

- mit Ablauf der Zeitscheibe erfolgt (logisch) ein Prozesswechsel
  - der unterbrochene Prozess wird ans Ende der Bereitliste verdrängt
    - ihm wird die Kontrolle über die CPU entzogen
  - der nächste Prozess wird gemäß FCFS der Bereitliste entnommen
    - ihm wird die Kontrolle über die CPU gewährt

- die Kernfrage des Entwurfs ist die Länge der jeweiligen Zeitscheibe

RR — Zeitscheibenlänge

- Faustregel: sie sollte etwas größer sein als die Dauer einer typischen Interaktion
  - was die „typische Interaktion“ ist, hängt ab von der Anwendungsdomäne
  - ihre Dauer wird meist durch spezielle Lastprofile experimentell ermittelt

- ist die Zeitscheibe sehr kurz. . .
  - laufen Prozesse mit kurzen CPU-Stößen relativ schnell durch +
  - wird jedoch ein hoher *Overhead*[4] zu erwarten sein −

- ist die Zeitscheibe sehr lang. . .
  - kann RR sehr schnell zu FCFS degenerieren −

[4] Aufgrund der Unterbrechungsbehandlung sowie der Scheduling- und Dispatching-Funktion.

RR — CPU- vs. E/A-intensive Prozesse

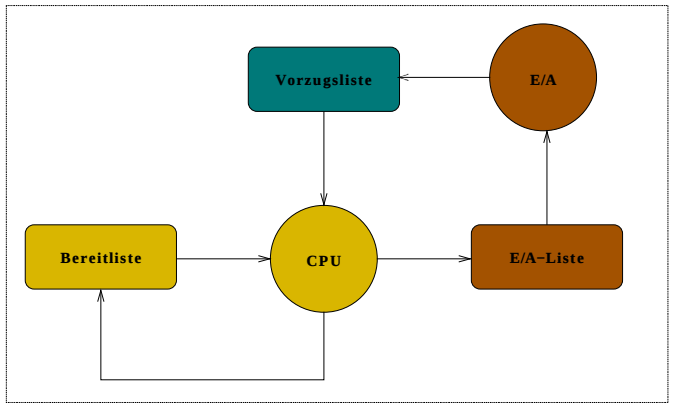
- ein Mix aus CPU- und E/A-intensiven Prozessen wirft Leistungsprobleme auf
  - E/A-intensive Prozesse beenden ihren CPU-Stoß innerhalb ihrer Zeitscheibe
    - sie blockieren und kommen mit Ende ihres E/A-Stoßes in die Bereitliste
  - CPU-intensive Prozesse schöpfen dagegen ihre Zeitscheibe voll aus
    - sie werden verdrängt und kommen sofort wieder in die Bereitliste

- die CPU-Zeit ist zu Gunsten CPU-intensiver Prozesse ungleich verteilt
  - E/A-intensive Prozesse werden schlecht bedient, Geräte schlecht ausgelastet
  - die Varianz der Antwortzeit E/A-intensiver Prozesse erhöht sich

VRR — *virtual round robin*

- vermeidet die bei RR mögliche ungleiche Verteilung der CPU-Zeiten
  - Prozesse kommen mit Ende ihrer E/A-Stöße in eine Vorzugsliste
  - diese Liste wird vor der Bereitliste abgearbeitet
- das Verfahren arbeitet mit Zeitscheiben unterschiedlicher Längen
  - Prozesse der Vorzugsliste bekommen keine volle Zeitscheibe zugeteilt
  - ihnen wird die Restlaufzeit ihrer vorher nicht voll genutzten Zeit gewährt
  - sollte ihr CPU-Stoß länger dauern, werden sie in die Bereitliste verdrängt
- der Dispatching-Vorgang ist dadurch im Vergleich zu RR etwas aufwendiger

VRR — Scheduling-Modell



SPN — *shortest process next*

- verringert die bei FCFS auftretende Benachteiligung kurzer CPU-Stöße
  - Grundlage dafür ist die Kenntnis über die Prozesslaufzeiten
- das Hauptproblem besteht darin, die Laufzeiten vorhersagen zu können
  - beim Stapelbetrieb geben Programmierer das erforderliche *time limit*[5] vor
  - im Produktionsbetrieb läuft der Job mehrfach nur zu statistischen Zwecken
  - im Dialogbetrieb wird ein Mittelwert der Stoßlängen eines Prozesses gebildet
- Antwortzeiten werden wesentlich verkürzt und die Gesamtleistung steigt

[5] Die Zeitdauer, innerhalb der der Job (wahrscheinlich/hoffentlich) beendet wird, bevor er abgebrochen wird.

SPN — Abschätzung der Dauer eines CPU-Stoßes

- Basis ist die Mittelwertbildung über alle CPU-Stoßlängen eines Prozesses:

$$S_{n+1} = \frac{1}{n} \cdot \sum_{i=1}^{n} T_i = \frac{1}{n} \cdot T_n + \frac{n-1}{n} \cdot S_n$$

- das Problem bei dem Vorgehen ist die gleiche Wichtung aller CPU-Stöße
  - jüngere CPU-Stöße sind jedoch von größerer Bedeutung als ältere
  - sie sollten daher auch mit größerer Wichtung berücksichtigt werden
- das Lokalitätsprinzip erfordert eine stärkere Einbeziehung jüngerer CPU-Stöße

SPN — Wichtung der CPU-Stöße

- die am weitesten zurückliegenden CPU-Stöße sollen weniger Gewicht erhalten:

$$S_{n+1} = \alpha \cdot T_n + (1 - \alpha) \cdot S_n$$

  - für den konstanten Wichtungsfaktor α gilt dabei: $0 < \alpha < 1$
  - er drückt die relative Wichtung einzelner CPU-Stöße der Zeitreihe aus

- teilweise Expansion der Gleichung führt zu:

$$S_{n+1} = \alpha T_n + (1 - \alpha)\alpha T_{n-1} + \ldots + (1 - \alpha)^i \alpha T_{n-1} + \ldots + (1 - \alpha)^n S_1$$

- für $\alpha = 0.8$: $S_{n+1} = 0.8T_n + 0.16T_{n-1} + 0.032T_{n-2} + 0.0064T_{n-3} + \ldots$

SRTF — *shortest remaining time first*

- lässt den SPN-Ansatz geeignet erscheinen für den Dialogbetrieb
  - Grundlage dafür ist Verdrängung (*preemption*)

- die Verdrängungsentscheidung wird getroffen, wenn ein Prozess laufbereit wird
  - sei t_{et} die erwartete CPU-Stoßlänge des eintreffenden Prozesses
  - sei t_{rt} die verbleibende CPU-Stoßlänge des laufenden Prozesses
  - der laufende Prozess wird verdrängt, wenn gilt: $t_{et} < t_{rt}$

- wie SPN kann auch SRTF Prozesse zum „Verhungern“ (*starvation*) bringen
  - dafür führt das verdrängende Verhalten zu besseren Duchlaufzeiten
  - dem RR-Overhead steht Overhead zur Stoßlängenabschätzung gegenüber

HRRN — *highest response ratio next*

- vermeidet das bei SRTF mögliche Verhungern von Prozessen langer CPU-Stöße
  - Grundlage dafür ist das Altern (*aging*) von Prozessen
  - d.h., die Wartezeit eines Prozesses findet Berücksichtigung

- ein kleines „Antwortsverhältnis“ wirkt sich positiv auf die Durchlaufzeit aus

$$R = \frac{w + s}{s}$$

  - mit $w =$ „Wartezeit des Prozesses“ und $s =$ „erwartete Bedienzeit“

- die Auswahl wird getroffen, wenn ein Prozess blockiert oder terminiert
  - ausgewählt wird der Prozess mit dem größten Wert für R

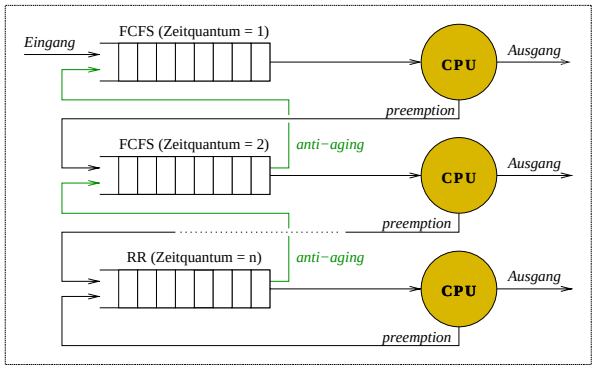
FB — *feedback*

- begünstigt kurze Prozesse, ohne die relativen Längen der Prozesse zu kennen
  - Grundlage ist die „Bestrafung“ (*penalization*) lange gelaufener Prozesse
  - Prozesse unterliegen dem Verdrängungsprinzip
  - je nach Laufzeitverhalten werden dynamische Prioritäten vergeben

- mehrere Bereitlisten kommen zum Einsatz, je nach Anzahl von Prioritätsebenen
  - wenn ein Prozess erstmalig eintrifft, läuft er auf höchster Prioritätsebene
  - mit Ablauf seiner Zeitscheibe, wird er in die nächst niedrige Ebene verdrängt
  - die unterste Ebene arbeitet nach RR, alle anderen (höheren) nach FCFS

- kurze Prozesse laufen relativ schnell durch, lange Prozesse können verhungern
  - die Wartezeit kann berücksichtigt werden, um höhere Ebenen zu erreichen

FB — Scheduling-Modell



Prioritäten

- ein Prozess-„Vorrang“, der Zuteilungsentscheidungen maßgeblich beeinflusst
  - häufig wird dem zahlenmäßig kleinsten Wert die höchste Priorität zugeordnet
- unterschieden wird zwischen statischen und dynamischen Verfahren:

  statische Prioritäten werden zum Zeitpunkt der Prozesserzeugung festgelegt
  - der Wert kann im weiteren Verlauf nicht mehr verändert werden
  - das Verfahren erzwingt eine deterministische Ordnung zwischen Prozessen

  dynamische Prioritäten werden während der Prozesslaufzeit aktualisiert
  - die Aktualisierung erfolgt im Betriebssystem, aber auch vom Benutzer aus
  - SPN, SRTF, HRRN und FB sind z.B. Spezialfälle dieses Verfahrens
- statische prioritätsbasierte Verfahren sind geeignet zur Echtzeitverarbeitung

Kombinierte Verfahren — *multilevel scheduling*

- mehrere Betriebsformen lassen sich nebeneinander („gleichzeitig“) betreiben
  - z.B. gleichzeitige Unterstützung von { Dialog- und Hintergrundbetrieb / Echtzeit- und sonstigem Betrieb }
  - jeweils werden dialogorientierte bzw. zeitkritische Prozesse bevorzugt bedient
- die technische Umsetzung erfolgt typischerweise über mehrere Bereitlisten
  - jeder Bereitliste ist eine bestimmte Zuteilungsstrategie zugeordnet
  - die Listen werden typischerweise nach Priorität, FCFS oder RR ausgewählt
- FB kann als Spezialfall dieses Verfahrens aufgefasst werden

Zusammenfassung

- Betriebssysteme müssen drei Arten von Zuteilungsentscheidungen treffen:
  1. *long-term scheduling* von Prozessen, die zum System zugelassen werden
  2. *medium-term scheduling* von aus- oder einzulagernden Prozessen
  3. *short-term scheduling* von Prozessen, die die CPU zugeteilt bekommen
- alle hier betrachteten Verfahren werden dem *short-term scheduling* zugerechnet
  - benutzer- und systemorientierte Kriterien sind schwer zu vereinheitlichen
  - die Auswahl des geeigneten Verfahrens kommt einer Gratwanderung gleich
- kombinierte Verfahren bieten Flexibiltät — gegen Implementierungskomplexität

Referenzen

[1] J. Nehmer and P. Sturm. *Systemsoftware: Grundlagen moderner Betriebssysteme*. dpunkt.Verlag GmbH, zweite edition, 2001. ISBN 3-89864-115-5.

[2] W. Stallings. *Operating Systems: Internals and Design Principles*. Prentice-Hall, fourth edition, 2001. ISBN 0-13-031999-6.