Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

5 CPU Scheduling

5.1 Grundlagen

5.1.1 CPU Burst / I/O Burst

Beobachtung: Programme rechnen typischerweise etwas, dann tätigen sie Ein/Ausgabe:
- CPU-Burst: das Programm rechnet eine Weile intensiv
- I/O Burst: das Programm wartet auf I/O

-> Unterteilung der Programme nach CPU-intensiv und I/O intensiv

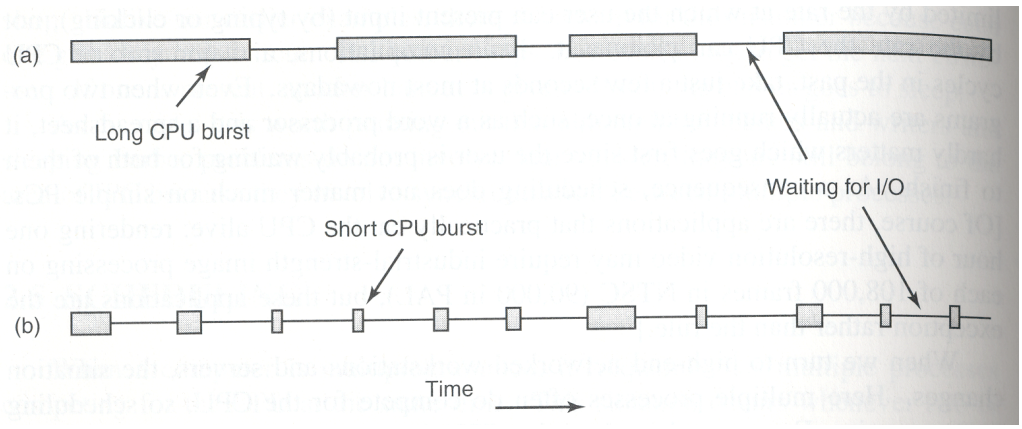

Abbildung 41: CPU intensiv (a) vs. I/O intensiv (b) (nach [2])

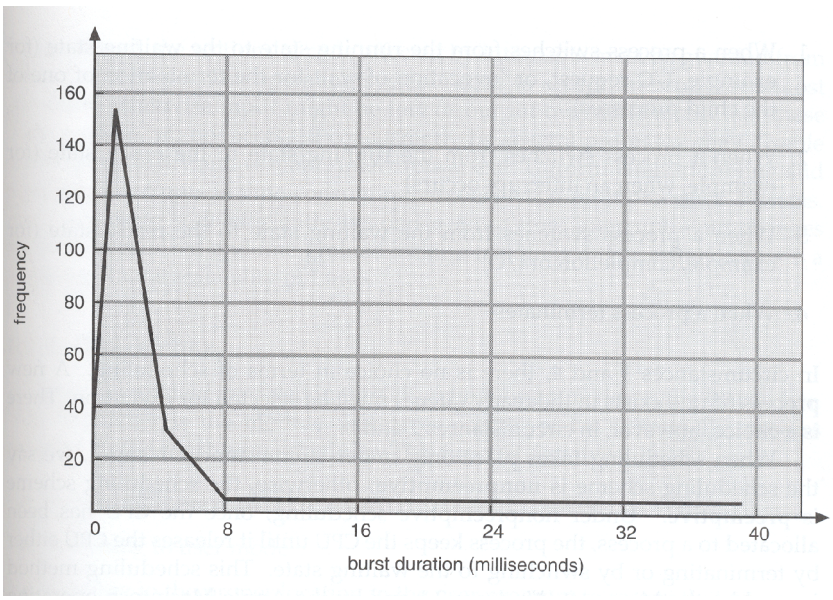

Abbildung 42: Burst Dauer (Histogramm) (nach [2])

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

5.1.2 Scheduler

Ein OS kann mehrere Scheduler haben, z.B.

- **Admission Scheduler:** startet Batch Jobs wenn ausreichende System-Resourcen vorhanden sind
- **CPU-Scheduler (Kurzzeit-Scheduler):** entscheidet, welcher Prozess (im Zustand READY) als nächstes aktiviert werden soll
- **Swapper-Demon:** entscheidet bei Systemüberlast, welcher Prozess ausgelagert werden soll; bei starker Systemüberlast können auch Prozesse gekillt werden; Linux: Prozess ***kswapd***

Im folgenden beschäftigen wir uns mit dem Kurzzeit-Scheduler.

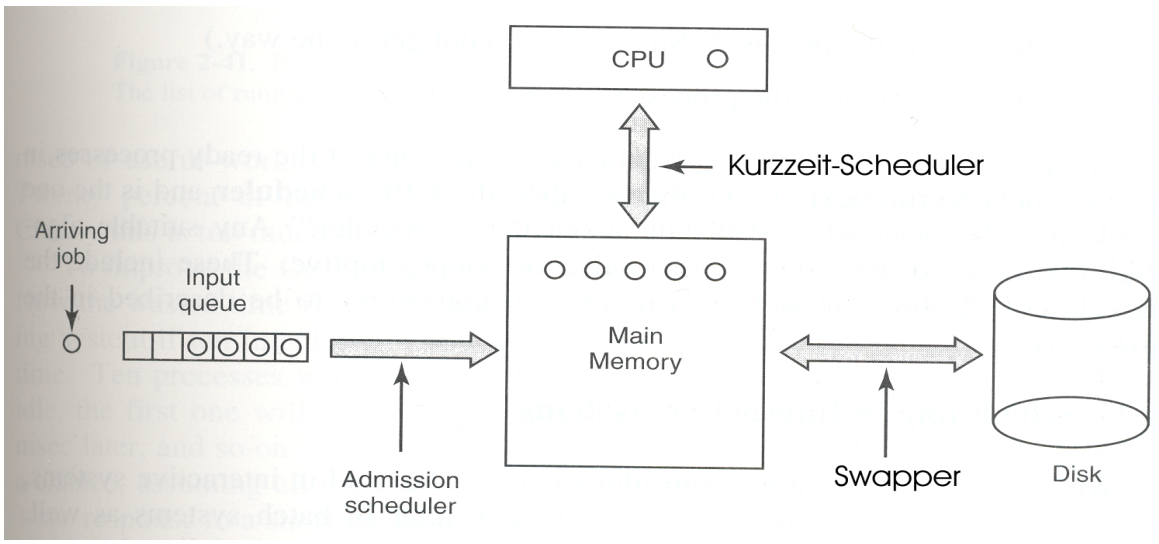

Abbildung 43: 3-Level Scheduler (nach [2])

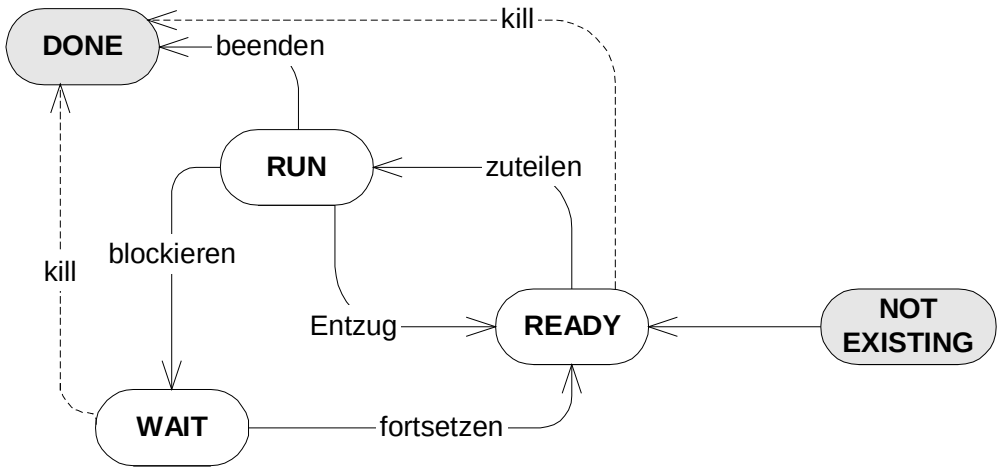

Abbildung 44: Prozess-Zustände

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

Wann werden nun Prozesse vom Scheduler zugeteilt?

1. Wenn ein Prozess vom Zustand RUNNING in den Zustand WAITING wechselt (z.B. I/O Request, oder wait()-Aufruf)
2. Wenn ein Prozess vom Zustand RUNNING in den Zustand READY wechselt (z.B. durch Interrupt)
3. Wenn ein Prozess vom Zustand WAITING in den Zustand READY wechselt (z.B. weil ein I/O Request fertig ist)
4. Wenn ein Prozess terminiert

Definition: Non-preemptive System
Ein aktiver Prozess läuft solange, bis er entweder blockiert (auf I/O wartet) oder von alleine die Kontrolle abgibt (z.B. sich selbst schlafen legt oder terminiert).
- D.h. Ein Prozesswechsel tritt nur bei 1) oder 4) auf.
- Alle diese Vorgänge sind unter Kontrolle des Prozesses, keine unerwarteten Prozesswechsel.

Definition: Preemptive System
Ein aktiver Prozess läuft maximal eine vorgegebene Zeit, danach wird er vom Scheduler unterbrochen und ein anderer Prozess wird aktiviert.
- Prozesswechsel auch bei 2 & 3: dies ist für den Prozess unverhersehbar und überraschend
- Zusätzlich benötigte Hardware: z.B. ein Timer, der regelmässig den Scheduler aktiviert

5.1.3 Dispatcher

Lädt den ausgewählten Prozess :
- Kontextwechsel
- Wechsel in den User-Mode (der Scheduler läuft im Kernel-Mode!)
- Sprung an die Stelle, an der der Prozess unterbrochen wurde, um das Programm weiterzuführen

Dispatch-Verzögerung: Zeit, die der Dispatcher braucht um einen anderen Prozess zum laufen zu bringen; diese Zeit geht verloren

5.2 Ziele des Scheduling

Was sind die Ziele des Scheduling?
Nach welchen Kriterien soll der nächste Prozess ausgewählt werden?

5.2.1 Allgemeine Kriterien

- **Gerechtigkeit:** alle Prozesse bekommen die CPU zugeteilt
- **Balance:** alle Teile des Rechners werden möglichst gleichmässig ausgelastet

5.2.2 Großrechner (Batch Systeme)

- **CPU-Auslastung:** die CPU-Auslastung sollte so hoch wie möglich sein.
- **Throughput (Durchsatz):** Möglichst viele Jobs in möglichst kurzer Zeit fertigstellen. Gesamtsicht auf das System: wieviele Jobs werden in 1 Stunde durchgerechnet?

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

- **Turnaround Time (Gesamtzeit):** Wie lange dauert es, bis ein Job fertig gerechnet wurde? Ist die Summe aus Rechenzeit, Wartezeit in der READY-Queue und I/O Wartezeit

5.2.3 Interaktive Systeme

- **Waiting Time (Wartezeit):** Wie lange muss ein Prozess warten? Scheduling kann nicht die Rechenzeit beeinflussen, sondern nur die Wartezeit.
- **Response Time (Antwortzeit):** Zeit, bis die erste Response vom System kommt. Turnaround-Time beschreibt die Beendigung des Jobs; in interaktiven Systemen ist oft die Zeit bis zur ersten Antwort wichtig (entscheidend für die Schnelligkeit aus User-Sicht).

5.2.4 Echtzeit-Systeme

- **Erfüllung von Zeitvorgaben:** Ein Echtzeitsystem muss die vorgegebenen Fristen (Deadlines) erfüllen.
- **Vorhersagbarkeit:** Vermeidung von Qualitätsverlusten, z.B. in Multimedia-Systemen.

5.3 Algorithmen

5.3.1 First Come First Served Scheduling (FCFS)

Algorithmus:
Die Jobs werden in der Reihenfolge ausgeführt, in der sie READY werden. Jeder Job läuft bis er blockt oder terminiert, dann startet der nächste.

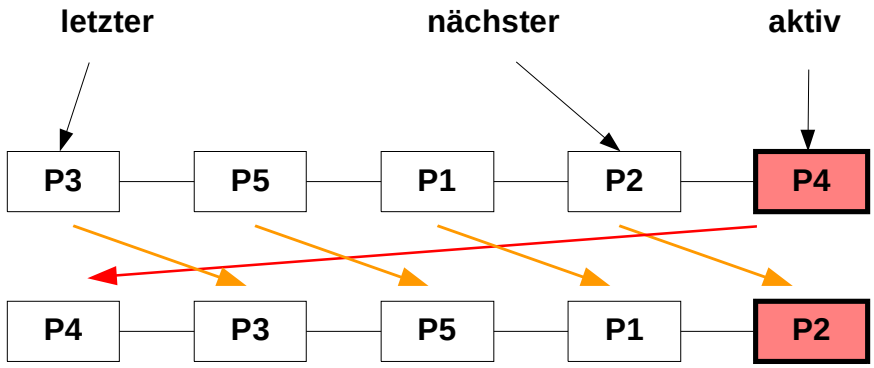

Abbildung 45: FCFS Warteschlange

- Der Algorithmus ist nicht-preemptive.
- Implementierung: als FIFO-Queue.
- Nachteil: durchschnittliche Wartezeit ist lang.

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

Beispiel:
Gegeben seien folgende Prozesse mit ihren Burst-Zeiten:

Prozess	*Burst Zeit*
P1	24 m
P2	3 ms
P3	3 ms

Annahme: Alle Prozesse werden gleichzeitig READY:

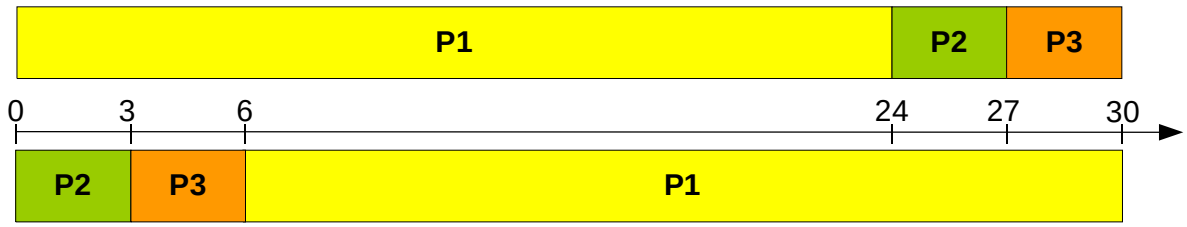

Abbildung 46: FCFS Beispiel

Je nach Reihenfolge ergeben sich folgende Auswertungen:

Auswertung	*Reihenfolge P1, P2, P3*	*P2, P3, P1*
Durchschnittliche Wartezeit	(0 + 24 + 27) / 3 = **17 ms**	(0 + 3 + 6) / 3 = **3 ms**

- Große Unterschiede in der Wartezeit, generell schlechte Performance
- Führt zu schlechter Ausnutzung des Rechners:

Konvoi-Effekt:
Annahme: ein CPU-intensiver Job und mehrere I/O intensive Jobs.
Der CPU-intensive Job blockiert die CPU für lange Zeit; alle anderen Prozesse sind idle. Blockt der CPU-intensive Job doch einmal aufgrund von I/O, dann kommen alle anderen Jobs der Reihe nach dran; da diese jedoch I/O intensiv sind, gehen sie schnell in den Wartezustand über. Dann kommt wieder der CPU-intensive Job dran, und alle anderen Warten.

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

Annnahme:
- P1 CPU intensiv,
- P2 und P3: I/O intensiv

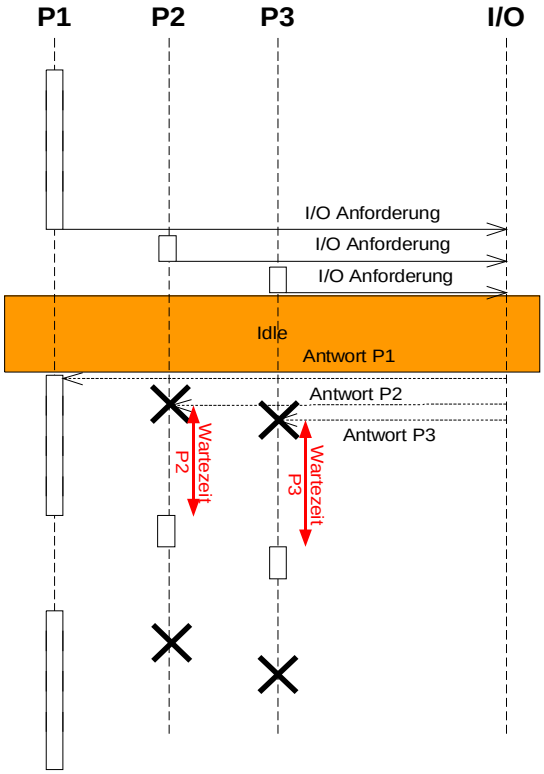

Abbildung 47: Konvoi Effekt bei FCFS

5.3.2 Shortest Job First Scheduling (SJF)

Algorithmus:
Der Job mit dem kürzesten CPU-Burst wird als nächstes drangenommen.

- Minimiert die Wartezeit: durchschnittliche Wartezeit ist kurz (sogar optimal unter der Bedingung dass alle Jobs zum gleichen Zeitpunkt ankommen)

Nachteile:
- Starvation (Verhungern) von CPU-intensiven Prozessen möglich: es werden immer Prozesse mit kurzen CPU-Bursts vorgezogen, ein Prozess mit sehr langem CPU-Burst kann verhungern
- Länge des nächsten CPU-bursts i.a. nicht bekannt, deshalb so nicht zu implementieren.

Schätzung der Länge aus bisherigen Daten,z.B. Exponentielles Mittel der bisherigen CPU Bursts, mit $0 \le \alpha \le 1$:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$$

Typischerweise $\alpha = ½$.

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

Auch mit Preemption möglich:
Definition Shortest Remaining Next Time (SRNT):
Wenn ein neuer Job READY wird, der eine kürzere verbleibende CPU Burst-Zeit hat als der aktuelle, so unterbricht er den aktuellen.

5.3.3 Round-Robin Scheduling (RR)

Algorithmus:
Jeder Prozess bekommt eine Zeitscheibe (Time-Quantum) zugeteilt, z.B. 10ms. Wenn der aktive Prozess sein Quantum aufgebraucht hat, wird er unterbrochen (preempted), und ein anderer Prozess wird aktiv. Blockiert ein Prozess vor Ende des Quantums, so wird wie FCFS geschedult.

- Preemptiver Algorithmus
- Sehr einfach: Verkettete Liste, wie FCFS (siehe Graphik bei FCFS)
- Einziger Parameter: Länge des Time-Quantums
- Verhindert Konvoi-Effekt, da lange Prozess-Bursts unterbrochen werden

Beispiel:
Gegeben seien wieder folgende Prozesse mit ihren Burst-Zeiten:

Prozess	*Burst Zeit*
P1	24
P2	3
P3	3

Zeitscheibe (Quantum): 4ms

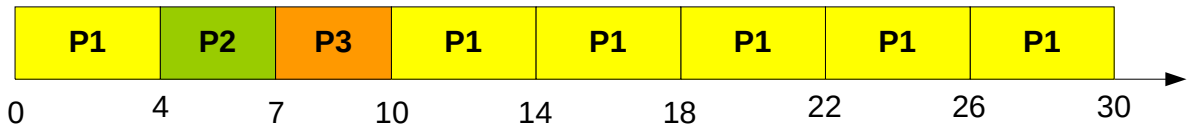

Abbildung 48: Round-Robin Beispiel

- Wartezeit: 6 ms (P1) + 4 ms (P2) + 7 ms (P3) = 17 ms, durchschnittliche Wartezeit: 17/3 = 5.66 ms

Nachteil:
Zusätzliche Scheduler-Aufrufe und Kontextwechsel kosten Zeit:

Beispiel:
CPU Burst des Prozesses: 100 ms
Beispiel A: Zeit Scheduler-Aufruf 1 ms
Beispiel B: Zeit Scheduler-Aufruf 5 ms

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

Graphik	*Quantum*	*Kontext-wechsel*	*Verschwen-dung Bsp. A*	*Verschwen-dung Bsp. B*
	120 ms	0	0	0
	60 ms	1	1 ms = 1%	5 ms = 5 %
	10 ms	9	9 ms = 9 %	45 ms = 45 %

Time-Quantum:
- zu kurz: Verschwendung durch zuviele Kontextwechsel
- zu lang: kein Job wird unterbrochen, Round-Robin degeneriert zu FCFS

5.3.4 Prioritäten-Scheduling

Algorithmus:
Jeder Prozess erhält eine weitere Eigenschaft, die Priorität. Es wird derjenige Prozess mit der höchsten Priorität ausgewählt.

Beispiel: Batch-Betrieb an der FH
Jobs von Studenten: Priorität 30
Jobs von Assistenten: Priorität 40
Jobs von der Verwaltung: Priorität 50
Jobs von Professoren: Priorität 60

- Achtung: manchmal bedeuten höhere Prioritäten höhere Zahlen, manchmal niedrigere.
- Bei mehreren Prozessen mit derselben Priorität: FCFS möglich, oder RR.
- Prioritäten können statisch vergeben werden (bei Prozessstart), oder dynamisch angepasst werden.

Prozess	*Burst Zeit*	*Priorität*
P1	10 ms	3
P2	1 ms	1
P3	2 ms	4
P4	1 ms	5
P5	5 ms	2

(niedrige Zahlen = hohe Priorität)

Reihenfolge (und Startpunkte): P2 (0 ms), P5 (1 ms), P1(6 ms), P3(16 ms), P4(18 ms)
Durchschnittliche Wartezeit: 41 ms/5 = 8.2 ms

Problem: Starvation (Aushungern)
Prozesse mit niedriger Priorität können "verhungern", d.h. niemals drankommen. (Gerücht: IBM 7094 bei MIT im Jahre 1973: ein niedrig-priorer Job lief seit 1967...)

Abhilfe:

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

"Altern", d.h. dynamische Prioritätenvergabe; je länger ein Prozess wartet, umso mehr wird seine Priorität heraufgesetzt, oder alternativ: je mehr ein Prozess rechnet, desto mehr wird seine Priorität herabgesetzt. Irgendwann ist seine Priorität so hoch, dass er drankommt.

5.3.5 Multilevel Queue Scheduling

Problem: Unterscheidung von CPU-intensiven und I/O-intensiven Prozessen
1. Möglichkeit: Priorität invers zur Länge des CPU-bursts:
    - I/O-intensive Prozesse haben kurze CPU-bursts -> erhalten eine hohe Priorität
    - CPU-intensive Prozesse -> niedrige Priorität
    - Stellt sicher, dass I/O-intensive Prozesse oft drankommen

Möglichkeit: Multilevel Queues

Algorithmus: Multilevel-Queues
Die Prozesse werden in verschiedene Queues eingeteilt, die unterschiedliche Prioritäten haben.
- System-Prozesse (höchste Priorität)
- Interaktive Prozesse
- Interaktive Editier-Prozesse
- Batch-Prozesse
- Studenten-Prozesse (niedrigste Priorität)

2-stufiges Scheduling:
Zunächst Auswahl der höchsten nicht-leeren Prioritätsklasse; innerhalb dieser Klasse FCFS oder RR.

Multilevel Queue Scheduling mit Feedback
Die Einteilung der Prozesse in verschiedene Queues wird mit der Zeit angepasst.

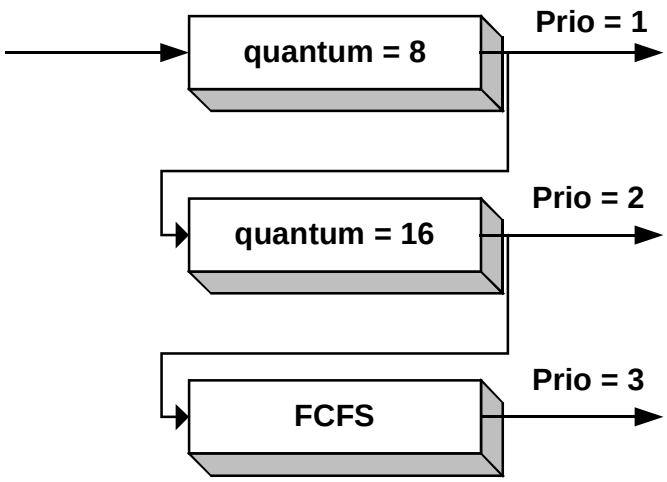

Abbildung 49: Multilevel Feedback Queue (nach [2])

Beispiel:
- Ein Prozess startet in der höchst-prioren Quantum Queue mit Zeitquantum 8. I/O intensive Prozesse brauchen ihr Zeitquantum nicht auf; sie verbleiben in dieser Queue. CPU-intensive

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

Prozesse brauchen ihr Zeitquantum auf und werden in die nächste Queue weiter geschoben.
- Wird der Prozess in der mittleren Queue ebenfalls unterbrochen, d.h. ist er sehr CPU-intensiv, so wird er in die niedrigt-priore Queue verschoben.
- Wechselt ein Prozess sein Verhalten, z.B. von CPU-intensiv zu I/O-intensiv, so kann er wieder in eine höher-priore Queue eingestuft werden.

5.3.6 Realtime Scheduling: Rate-Monotonic Scheduling (RMS)

5.3.6.1 Hard-Realtime

Definition:
Alle Prozesse müssen innerhalb der vorgegebenen Zeit beendet werden.
- Beispiel: ABS Bremssystem, Airbag-Auslöser (eine gelegentliche Überschreitung der Fristen ist inaktzeptabel)
- Ist schwierig zu garantieren

5.3.6.2 Soft-Realtime

Weniger strenge Anforderungen: Zeitliche Vorgaben ***sollten*** eingehalten werden. Ein Überschreiten der Fristen stellt zwar ein Qualitätsverlust dar, ist jedoch noch kein Totalausfall.
- Beispiel: Video-Player, der ab und zu ein Bild "auslässt"
- typischerweise durch hohe Prioritäten für Realtime-Prozesse

5.3.6.3 Rate-Monotonic Scheduling

Für periodische Aktivierungen von Prozessen

Definition:
Prioritäten-basiertes Scheduling, wobei die Priorität des Prozesses umso höher ist je kürzer die Periode ist

Ausführbarkeit (Schedulability), System mit Preemption:
Gegeben seien n Prozesse mit den (worst-cast) Laufzeiten c1, ... cn und den jeweiligen Perioden p1, ..., pn. Eine Bedingung für die Ausführbarkeit ist (bei Preemption) ist:

$$U = \sum_{i=1}^{n} \frac{c_i}{p_i} \leqslant 1$$

Beispiel:
Gegeben seien 3 periodische Prozesse P1, P2 und P3 mit:

Prozess	***Periode (ms)***	***Worst-case Dauer (ms)***	***Prio***[1]
P1	100	50	1
P2	200	30	2
P3	500	100	5

[1]Niedrige Zahlen = hohe Priorität

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

Dann ergibt sich

$$U = \frac{50}{100} + \frac{30}{200} + \frac{100}{500} = 0.5 + 0.15 + 0.2 = 0.85 < 1$$

Annahme: Aktivierung von
- P1 bei 50, 150, 250, ...
- P2 bei 66, 266, 466,
- P3 bei 0, 500, 1000, ...

Dann ergibt sich folgendes Aktivierungsschema:

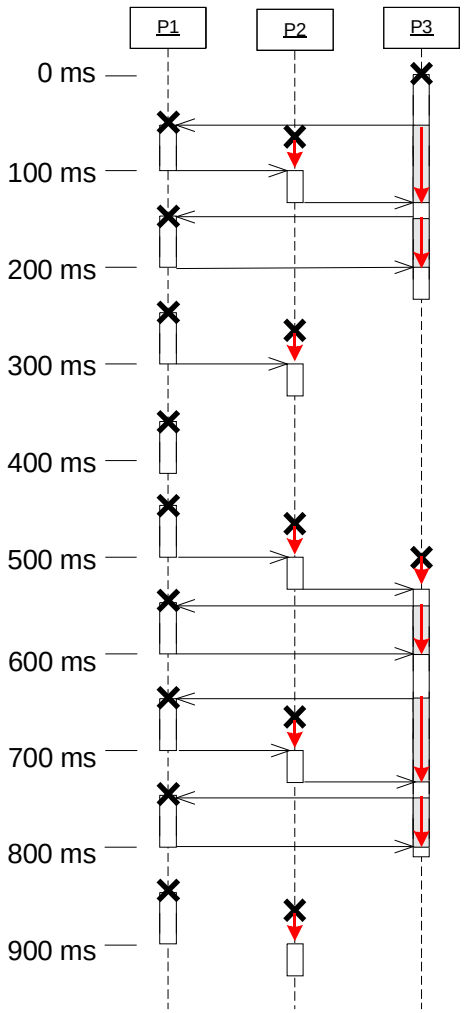

Abbildung 50: RMS Aktivierungsschema (Beispiel)

Dies reicht jedoch nicht aus, um die Ausführbarkeit aller Deadlines zu garantieren.
Eine untere Grenze ergibt sich aus:

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

Gegeben seien n Prozesse mit den (worst-cast) Laufzeiten c1, ... cn und den jeweiligen Perioden p1, ..., pn. Eine Bedingung für die Ausführbarkeit ist (bei Preemption) ist:

$$U = \sum_{i=1}^{n} \frac{c_i}{p_i} \leqslant n(2^{1/n} - 1)$$

5.3.6.4 Problem: Prioritäteninversion

1. Ein niedrig-priorer Prozess P1 ist aktiv und allokiert eine Resource A
2. Ein mittel-priorer Prozess P2 wird aktiv und unterbricht P1
3. Ein hoch-priorer Prozess P3 wird aktiv, P2 wird unterbrochen
4. P3 benötigt die Resource A und geht in den Wartezustand über
5. Als nächstes rechnet P2 zum Ende
6. Dann erst kommt P1 dran: P1 rechnet weiter und gibt A wieder frei
7. Nun erst kommt P3 (höchst-prior) wieder dran und unterbricht P1
Folge: P2 hat vor P3 fertiggerechnet, obwohl die Priorität von P2 niedriger ist als die von P3!

Mögliche Alternativen:
- *Priority Inheritance: warten ein oder mehrere Prozesse auf eine Resource, die ein anderer Prozess gelockt hat, so erhält dieser Prozess kurzzeitig die Priorität des höchsten wartenden Prozesses*
- *Priority Ceiling: Für jede Resource wird im Vorfeld eine Ceiling-Priorität festgelegt, die höher ist als die jeden Prozesses der diese Resource zu irgendeinem Zeitpunkt nutzen will. Ein Prozess, der die Resource aquiriert, erhält automatisch die Ceiling-Priorität.*

5.3.7 Evaluierung der Algorithmen

Um den geeignetsten Algorithmus auswählen zu können, werden diese evaluiert. Ein wichtiges Mittel dazu sind Simulationen, die auf aktuellen Szenarien beruhen.

5.4 Thread Scheduling

Mit Threads haben wir zwei ausführbare Einheiten im System.
Unterschiede bei User-level Threads (many-to-one) und Kernel-Threads (one-to-one):
- Bei User-level Threads (many-to-one) kennt der Scheduler keine Threads. Er teilt die CPU jeweils einem Prozess zu; der User-level-Scheduler führt dann verschiedene Threads aus.
  Beispiel: Time-Quantum 50ms (für Prozesse), 10ms für User-level threads:
  Kernel schedult Prozess A: A1, A2, A3, A1, A2, dann Prozess B: B1, B2, B3, B1, B2
- Bei Kernel-level Threads (one-to-one) kennt der Kernel-Scheduler die Threads und schedult diese einzeln:
  Beispiel: Thread-Quantum 20ms: A1, B1, A2, B2, A3, B3, A1, ...

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

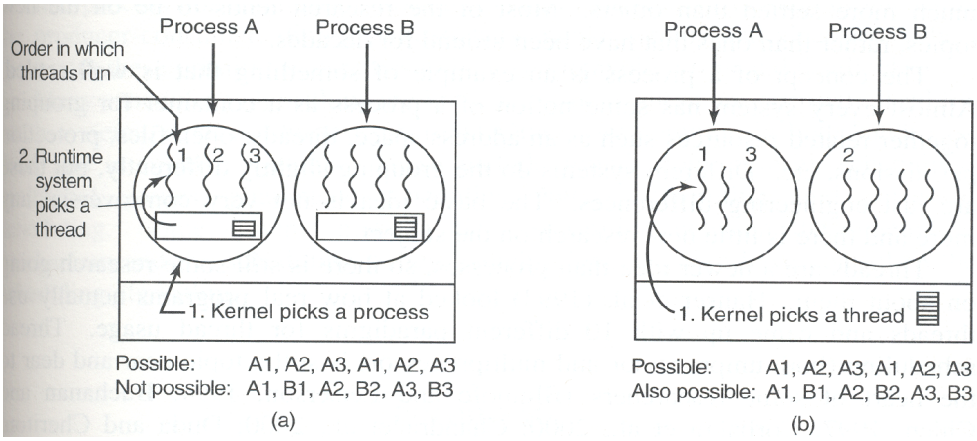

Abbildung 51: Overview Thread Scheduling (nach [1])

Unterschiede:

- User-level Thread Scheduling:
  - Vorteil: ist schnell, da kein Prozesswechsel stattfindet
  - Nachteil: Blockiert ein Thread, wird der gesamte Prozess -> WAITING und alle anderen Threads blockieren auch
  - Alle Threads eines Prozesses laufen in derselben Prioritätsklasse
- Kernel-level Thread Scheduling:
  - Kontextwechsel erforderlich -> schlechtere Performance
  - Threads eines Prozesses können unterschiedlich geschedult werden -> unterschiedliche Prioritäten möglich
  - Kernel kann Threads und Prozesse berücksichtigen

5.5 Beispiele

Generell gilt: ein gutes Scheduling für allgemeine Rechner erfordert viel Alchemie: man liest oft so Sachen wie "kurzzeitig um 3 Prioritätsstufen erhöhen" (Win 2000), oder bei Linux: "im einen Fall um zwei Einheiten erniedrigen, andernfalls verdoppeln" - wo kommen diese Zahlen her? Erfahrungswerte, gewonnen aus vielen verschiedenen Simulationen. Gutes Scheduling = Kompromiss zwischen vielen verschiedenen, sich widersprechenden Anforderungen.

5.5.1 Solaris 2

Prioritäts-basiertes Scheduling mit 4 Klassen:
1. Real-time: garantierte Antwortzeit (nur sehr wenige Prozesse), FCFS
2. System: Reserviert für Kernel-Use FCFS
3. Time-sharing
4. Interaktive

Realtime hat die höchste Priorität, gefolgt von System. Time-Sharing und Interaktive haben dieselbe Priorität.
Interaktive und Time-Sharing benutzen Multilevel-Feedback-Queue mit Round-Robin, wobei Interaktive für Fenster-Applikationen (Response) optimiert ist, während Time-Sharing für CPU-intensive Prozesse optimiert ist.

Scheduler: ein Prozess läuft bis

- er blockiert

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

- sein Zeit-Quantum abgelaufen ist (nicht Realtime- und System-Threads)
- er durch einen höher-prioren Thread unterbrochen wird.

Klasse	*Priorität*	*Scheduler*	*Bemerkungen*
Realtime	Höchste	FCFS	Realtime-Kernel Threads
System	Hoch	FCFS	Kernel-Threads
Interaktive	Normal	Multilevel-Feedback-Queue	Optimiert für interaktive Anwendungen
Time-Sharing			Optimiert für CPU-intensive Prozesse

5.5.2 Windows 2000

- Scheduler nutzt ein 32-Stufen Prioritätsschema:
  - 16-31: Realtime Prioritäten
  - 1-15: Variable Klasse Prioritäten
  - 0: Idle-Prozess und Swapper
- Für jede der 32 Prioritäten gibt es eine eigene READY-Queue; der Scheduler aktiviert jeweils den nächsten Job aus der höchsten, nicht-leeren Queue.
- Wenn es keinen READY-Job gibt, wird der Idle-Thread ausgeführt.
- Beendet ein Thread den WAITING-Zustand, so erhält er einen Boost: bessere Performance für interaktive Prozesse, die oft auf Tastatur und Maus warten.
- Wenn ein Thread aus der variablen Klasse sein Time-Quantum aufgebraucht hat, so wird seine Priorität in gewissen Grenzen erniedrigt. Dadurch werden CPU-intensive Prozesse etwas gebremst.
- Win2K unterscheidet zwischen *Foreground* und *Background*-Prozessen. Erstere erhalten einen Performance-Bonus.

5.5.3 Linux (Kernel < 2.6)

- Zwei unterschiedliche Prozess-Scheduling Algorithmen:
  - ein fairer preemptiver Algorithmus für normale Prozesse
  - und ein prioritäsbasierter für zeitkritische Prozesse
- Kredit-basierter Algorithmus:
  - jeder Prozess erhält einen bestimmten Kredit
  - der Prozess mit dem höchsten Kredit wird ausgewählt
  - Hat kein Prozess mehr einen Kredit, so erhalten *alle* (auch die im WAITING-Zustand) Prozesse zusätzliche Kredite nach der Formel

    credits = credits/2 + priority

    (gibt I/O-intensiven und interaktiven Prozessen mehr Kredits)
- 2.6-Kernel: anderer Scheduler