

Rechnernetze und Betriebssysteme

4. Scheduling

Prof. Dr. Martin Becke
SS2015 – Version 1.0

1. Motivation

- Moderne OS müssen häufig mehr als einen Prozess oder Thread organisieren
- Ressourcen sind limitiert, daher müssen Entscheidungen getroffen werden
 - Wann soll ein Prozess/Thread ausgewählt werden (dispatched)
 - Welcher Prozess/ Thread soll ausgewählt werden

➔ Aufgabe übernimmt Scheduler (Arbiter) als Steuereinheit

- Verschiedene OS benötigen andersartige Scheduler
 - Stapelverarbeitungssysteme (Auslastung, Durchsatz, Durchlaufzeit)
 - Interaktive Systeme (Userverhalten) ➔ Typische Desktop PCs
 - Echtzeitsysteme (Zeit als zusätzliche Anforderung)
- Scheduler kann unterbrechend (präemptiv) oder nicht unterbrechend (kooperativ) arbeiten
 - Präemptiv: Prozesse/Threads werden in regelmässigen Intervallen unterbrochen für neue/andere Aufgaben
 - Kooperativ: Prozess/Thread gibt Ressourcen nach vollendeter Aufgabe wieder frei

2. Präemptives Scheduling

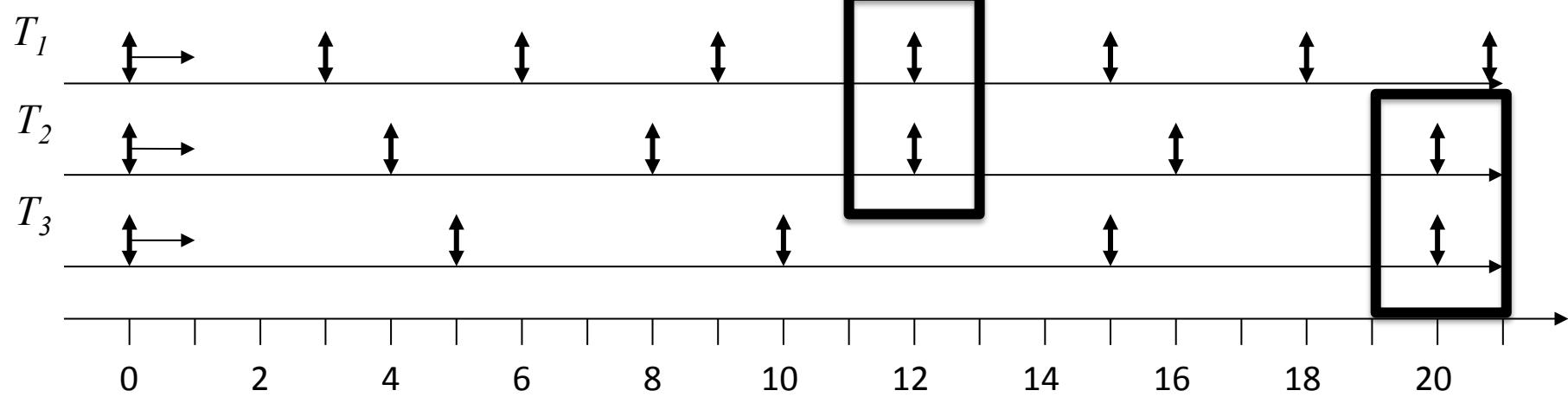


- Es gibt verschiedene Events, die ein Scheduler nutzen kann um den Kontext (Prozess/Thread) zu wechseln
 - Timer läuft ab (Zeitschlitzverfahren)
 - E/A (I/O) Interrupt
 - Thread wird erstellt oder beendet
 - Thread ist blockiert
 - Systemcalls
 - Priorisierungsänderungen
 - Veränderte Echtzeitanforderungen
 - ...

3. Schedulingentscheidungen

- Scheduling (Zeitplanung) kann optimal genutzt werden wenn alle Anforderungen bekannt sind:
- Beispiel: Persönlicher Terminplan (Freie Nachmittle)

 - RNB: mindestens jeden 3. Nachmittag für Wiederholung reservieren (Kurzzeit- ↗ Langzeitgedächtnis)! (T_1)
 - Sport: mindestens jeder 4. Nachmittag für Sport! (T_2)
 - Freund(in): mindestens jeder 5. Nachmittag! (T_3)



4. Schedulingentscheidungen – Regeln (Policies)



- Jede Aufgabe erhält eine Priorität
- **Es wird immer die Aufgabe mit der höchsten Priorität ausgeführt**
- Wird eine Aufgabe höherer Priorität ausführbereit, während eine Aufgabe niedrigerer Priorität ausgeführt wird, so wird die Aufgabe niedrigerer Priorität unterbrochen (*preemption*) und später fortgeführt.
- Bei gleichen Prioritäten wird die bisherige Aufgabe weiter ausgeführt
- Prioritäten können von vornherein festgelegt sein (d.h. statisch sein) oder immer wieder neu ausgerechnet werden müssen (d.h. dynamisch sein)

4. Schedulingentscheidungen - Regeln



- Übung (Folie wird nachgereicht)

5. Scheduler – Periodische Tasks



- **Definition:** Ein Schedulingverfahren für periodische *Tasks* heißt **optimal**, wenn es immer einen Ablaufplan findet, sofern einer existiert.
- **Definition:** Die **Hyperperiode** (engl. *hyperperiod*) ist das kleinste gemeinsame Vielfache der Perioden der *Tasks*.
- Die einzelnen Hyperperioden unterscheiden sich in ihrem Verhalten nicht.
- Zum Nachweis von Eigenschaften eines Systems ist es also ausreichend, **eine** Hyperperiode zu betrachten
- EDF findet Anwendung in Echtzeitbetriebssystemen mit periodischen Tasks (Industrielle Steuersysteme) – z.B. Siemens Simatec
- Alternative zu EDF ist z.B. Rate Monotonic Scheduling RMS
 - Die Prioritäten werden statisch anhand Periodendauer festgelegt

6. Scheduler – Nicht Periodische Tasks



- Häufig kommen nicht periodische Prozesse/
Threads zum Einsatz
 - Multikriterielle Optimierungsaufgabe , i.d.R. NP
Vollständig
 - Notwendigkeit von heuristischen Algorithmen
- Eine Vielzahl von (optimierten) Algorithmen
abhängig von den Zielen bekannt
 - Ressourcenauslastung
 - Fairness
 - Wichtigkeit

Es können nur Beispiel
besprochen werden!
Entwicklung eines guten
Schedulers immer noch
im Fokus aktueller
Forschung

7. Scheduler – Ziel: Ressourcenauslastung



- First Come First Served (FCFS) oder First In First Out (FIFO)
 - Reihenfolge der Bearbeitung wird durch Zeitpunkt der Bereitstellung festgelegt
 - Besteht aus einer Warteschlange
 - Algorithmuskosten sind gering - O(1)
 - Schneller Zugriff auf den nächsten Task - O(1)
 - Nicht geeignet bei E/A intensiven Prozessen
 - Hohe Varianz der Bearbeitungszeiten für Threads/Prozessen mit unterschiedlichen Lastmerkmalen
- FCFS/FIFO ist typisch für Stapelverarbeitung

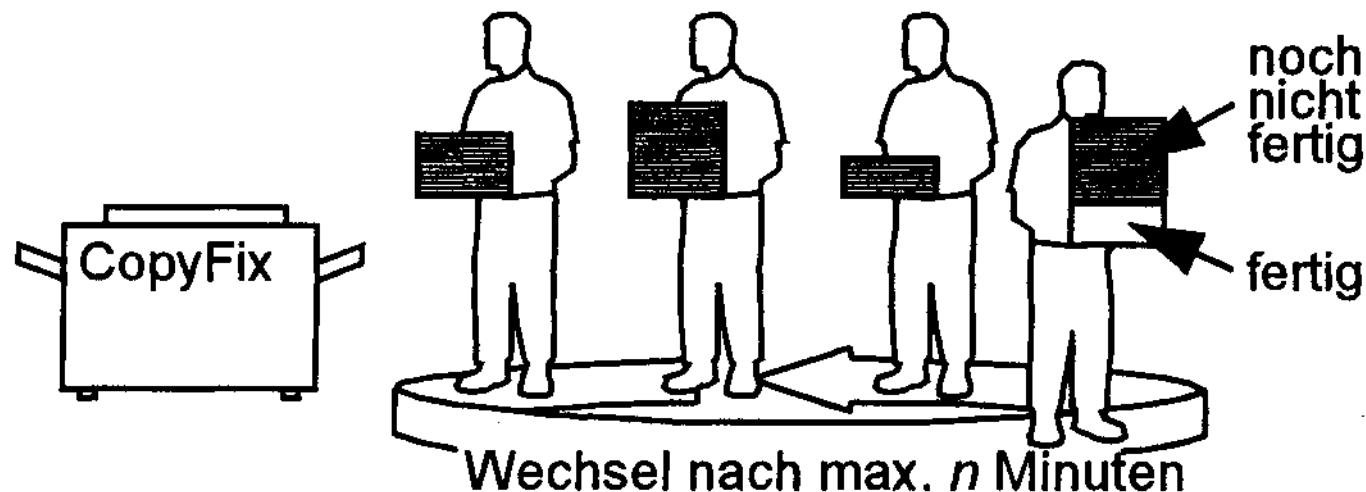
8. Scheduler – Ziel: Ressourcenauslastung



- Shortest Remaining Time Next (SRTN)
 - Thread/Prozess mit voraussichtlich kürzester Rechenzeit erhält Rechenzeit
 - SRTN ist preäemptiv, Threads können durch andere Threads verdrängt werden
 - Kalkulation über Laufzeit kann aufwendig sein
 - SRTN benötigt 1 Warteschlange
 - SRTN benötigt für den Zugriff eine Sortierung, üblich $O(n)$
 - Zugriff selbst ist direkt $O(1)$
 - Reaktionszeiten für kurze Prozesse wird verkürzt
 - Langläufige Threads können „verhungern“
 - Mehr Scheduling Overhead durch mögliche Kontextwechsel
- SRTN findet auch besonders Anwendung in Stapelverarbeitung

9. Scheduler – Ziel: Fairness

- Round Robin
 - Die gesamte CPU-Zeit wird gleichmäßig auf alle Prozesse verteilt
 - Eine feste Zeitscheibe („Quantum“) von z.B. x ms wird für alle Prozesse festgelegt
 - Jeder Prozess bekommt für x ms die CPU und wird nach Ablauf dieser Zeitscheibe unterbrochen
 - Jeder Prozess erhält die CPU erst wieder, wenn zwischenzeitlich alle anderen Prozesse ihre Zeitscheibe verbrauchen durften



10. Scheduler – Ziel: Fairness



- Round Robin Implementierung
 - Regelmäßige Unterbrechung durch einen Timer-Interrupt (Timer – Interrupt wird periodisch generiert)
 - Wenn ein Interrupt erfolgt, wird der aktuell laufende Prozess zurück in die „Ready“-Queue gestellt und der nächste Prozess wird ausgesucht (→ Prozesswechsel!)
 - Blockiert ein Prozess in „seiner“ Zeitscheibe, bekommt der nächste Prozess (Thread) die CPU
 - Optimierung der Zeitscheibengröße (Quantum)?
 - zu kurz: viele Prozesswechsel, viel Overhead
 - zu lang: evtl. schlechte Antwortzeiten
 - Erfahrungswert: ~20 – 50 ms

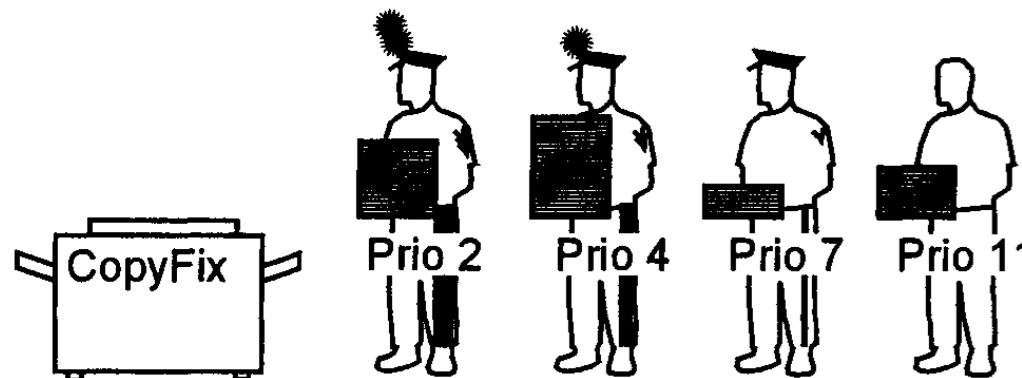
10. Scheduler – Ziel: Fairness



- Round Robin Nachteile
 - Gewöhnliches „Timesharing“ Verfahren für interaktive Systeme sehr effektiv, aber I/O-lastige Tasks werden unterschiedlich behandelt
 - I/O Event blockiert Task
 - Task kann erst weitergeführt werden, wenn Blockade aufgelöst wurde, kommt aber wieder ans Ende der Warteschlange

11. Scheduler – Ziel: Fairness mit Priorität

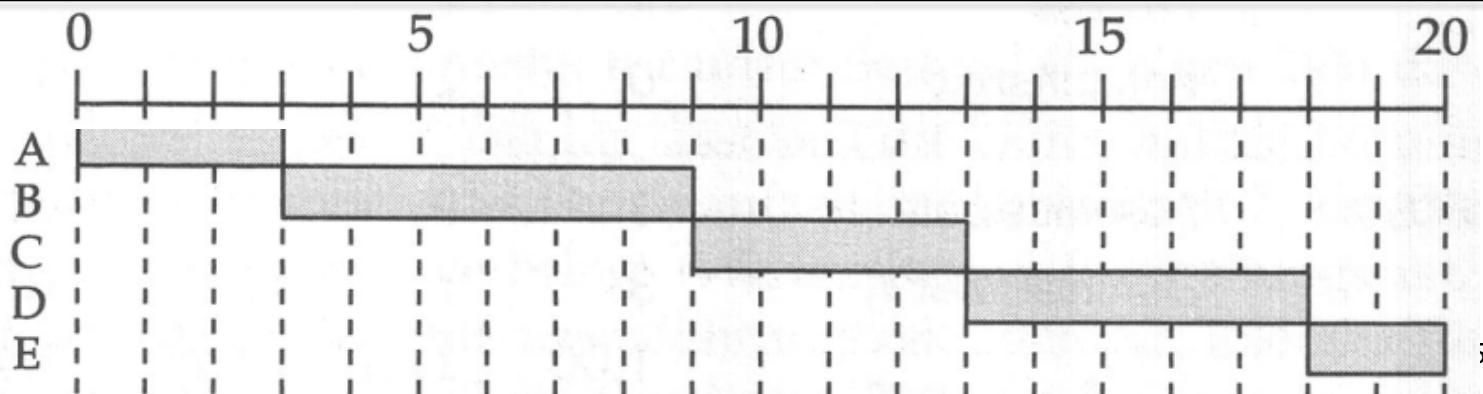
- Dynamic Priority Round Robin (DPRR) (nach Prioritäten gewichtet)
 - Zuweisen von Prioritäten an Prozesse
 - Bevorzugung des Prozesses mit der höchsten Priorität
 - Zuweisung von Prioritäten kann dynamisch erfolgen



12. Scheduler – Ziel: Fairness mit Priorität

Prozess	A	B	C	D	E
Ankunftszeit	0	2	4	6	8
Bearbeitungszeit T_s	3	6	4	5	2

FCFS	A	B	C	D	E	Mittelwert
Abschlusszeit	3	9	13	18	20	
Durchlaufzeit T_r	3	7	9	12	12	8,60
T_r / T_s	1,00	1,17	2,25	2,40	6,00	2,56



13. Scheduler – Ziel: Fairness mit Priorität

Prozess	A	B	C	D	E
Ankunftszeit	0	2	4	6	8
Bearbeitungszeit T_s	3	6	4	5	2

		A	B	C	D	E	Mittelwert
RR $q = 1$	Abschlusszeit	4	18	17	20	15	
	Durchlaufzeit T_r	4	16	13	14	7	10,80
	T_r / T_s	1,33	2,67	3,25	2,80	3,5	2,71
RR $q = 4$	Abschlusszeit	3	17	11	20	19	
	Durchlaufzeit T_r	3	15	7	14	11	10,00
	T_r / T_s	1,00	2,5	1,75	2,80	5,50	2,71

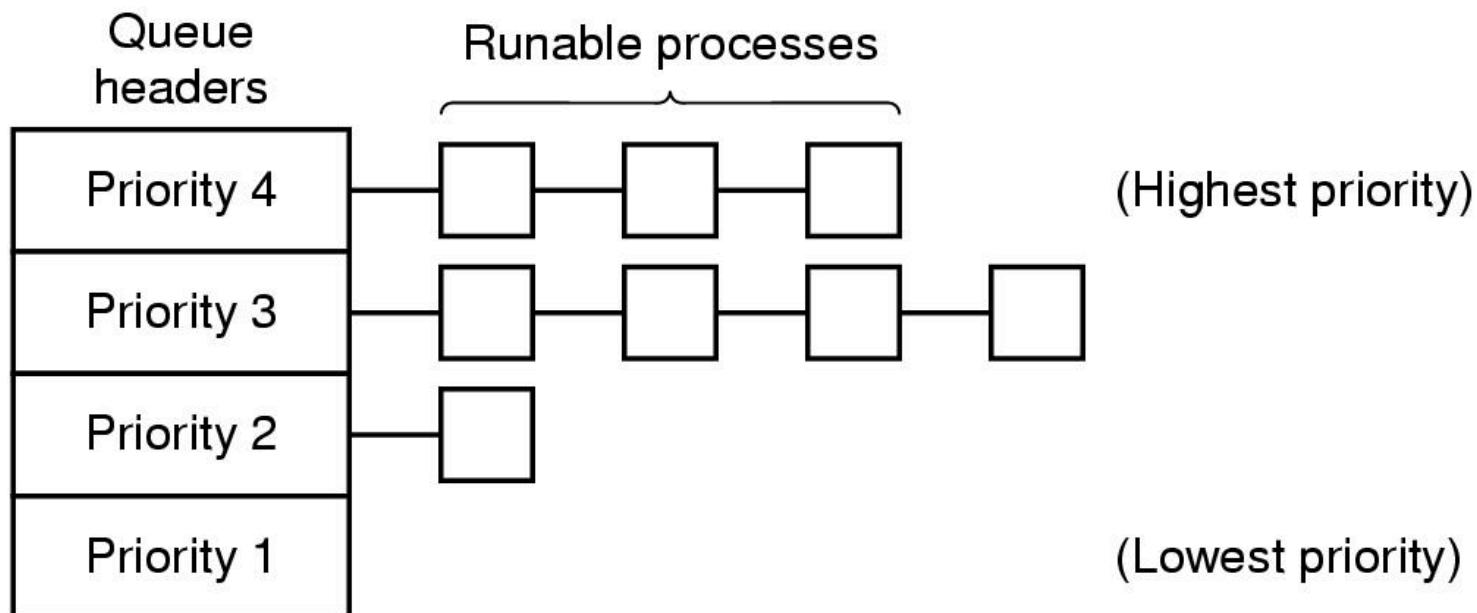
13. Scheduler – Ziel: Fairness mit Priorität



- Übung (Folie wird nachgereicht)

14. Scheduler – Prioritätsklassen

- Realisierung i.d.R. durch eine Warteschlange pro Prioritätsklasse.
- Die ankommenden Prozesse werden je nach Prioritätsklasse in die entsprechende Warteschlange eingereiht.
- Prozesse einer Klasse sind erst „dran“, wenn alle höher priorisierten Klassen-Warteschlangen leer sind!
- Scheduling innerhalb einer Klasse: meist Round-Robin-Verfahren (ggf. mit unterschiedlichen Zeitscheiben)!

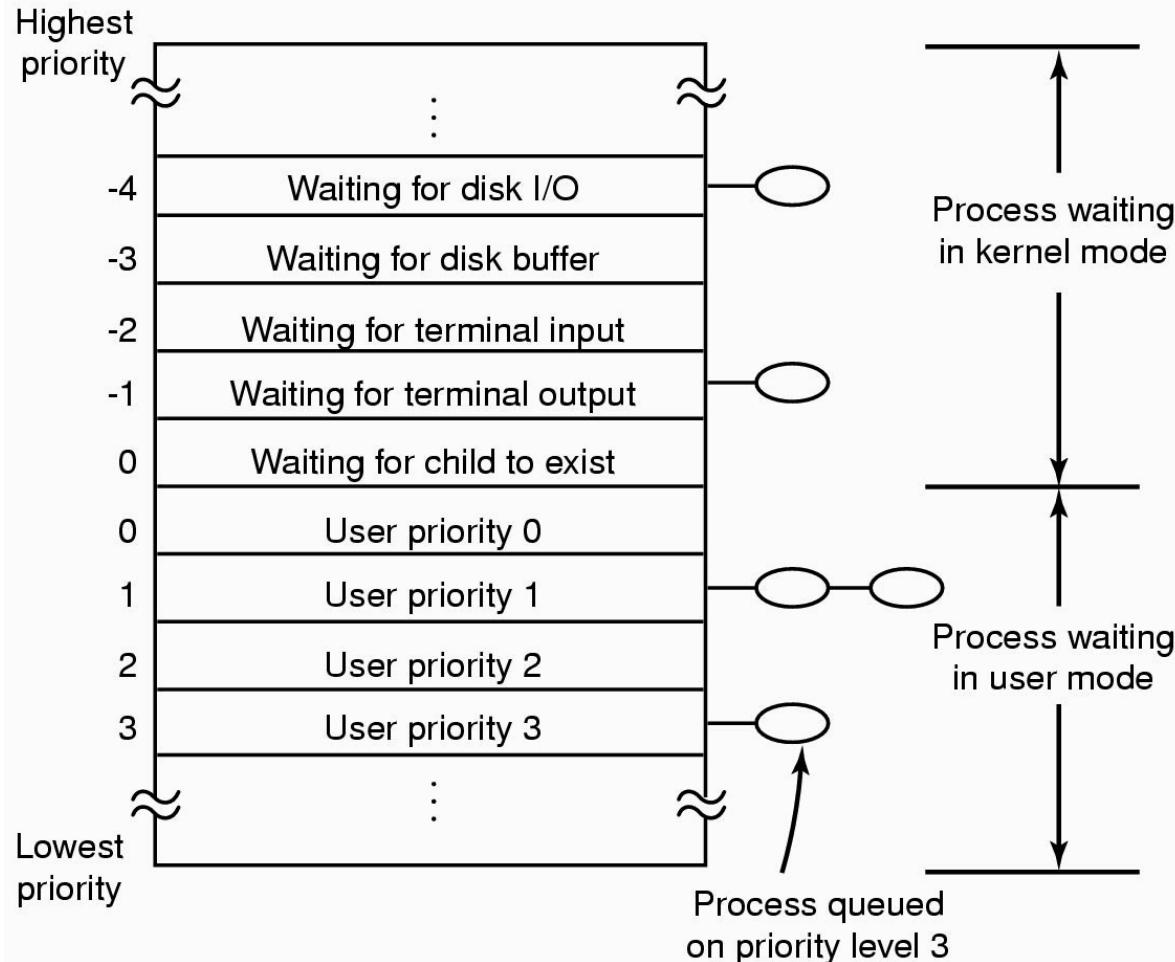


15. Traditionelles UNIX-Scheduling



- Multilevel-Feedback mit Round-Robin in den einzelnen Prioritätsklassen
- Prioritäten werden 1-mal pro Sekunde neu berechnet
- Basis-Priorität bestimmt für jeden Prozess die Ausgangs-Prioritätsklasse
- Es wird unterschieden zwischen Systemprozessen und User-Prozessen
 - Systemprozesse haben immer eine höhere Priorität
- Korrektur der Priorität bevorzugt Interaktive Prozesse (E/A-intensiv)
- Mit einem „Nice“-Wert kann ein Benutzer die Priorität eines seiner Prozesse heruntersetzen

16. Traditionelles UNIX-Scheduling



Neuberechnung der Priorität nach der Formel:
Prioritätsklasse = CPU-Verbrauch + Nice + Basispriorität

17. Windows Scheduling

- Multilevel-Feedback mit Round-Robin in den einzelnen Prioritätsklassen
- Windows schadelt nur Threads
- 32 Prioritätsskalen (0-31) in 2 Kategorien angeordnet:
 - „Echtzeit“ (Klasse 31-16) [hohe Priorität]
 - Für System-Threads
 - Feste Prioritäten, Round-Robin bei gleicher Priorität
 - Variabel (Klasse 0-15) [niedrige Priorität]
 - Für Benutzer-Threads
 - Variable Prioritäten, je nach Thread-Verhalten
 - Priorität steigt bei Warten auf Ereignis
 - Priorität sinkt bei Verbrauch einer ganzen Zeitscheibe

18. JAVA Thread - Scheduling

- Verwendung von Prioritätsklassen
- Realisierung der Java-Thread-Priorität über Betriebssystem-API
- Methoden der Klasse Thread für das Scheduling:

public void setPriority(int newPrio)

- Weist dem Thread eine Priorität zu
 - zwischen Thread.MIN_PRIORITY (üblich: 1) und Thread.MAX_PRIORITY (üblich: 10)
 - Default: Thread.NORM_PRIORITY (üblich: 5)

public int getPriority()

- Liefert die aktuelle Priorität

→ Aber: *Das Betriebssystem trifft die Scheduling-Entscheidungen nach seinen Regeln!*

29. Zusammenfassung

Prozess-Scheduling



- Prozesse und auch Threads müssen organisiert werden
- Scheduler übernehmen diese Aufgabe
- Ziele von Scheduling-Algorithmen können unterschiedlich sein
- Interaktive Systeme
 - Round-Robin („Zeitscheibenverfahren“)
 - Prioritätenbasiertes Scheduling
 - statisch / dynamisch
 - mit mehreren Prioritätsklassen / Warteschlangen
- Besprochen wurden UNIX / Windows / JAVA - Scheduling