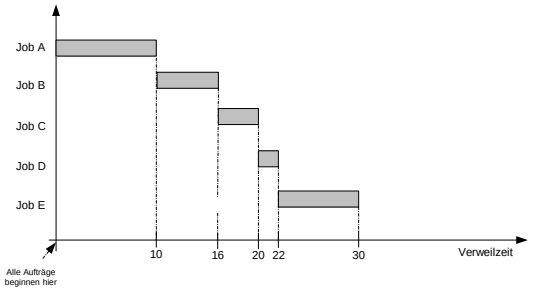
MAS: Betriebssysteme

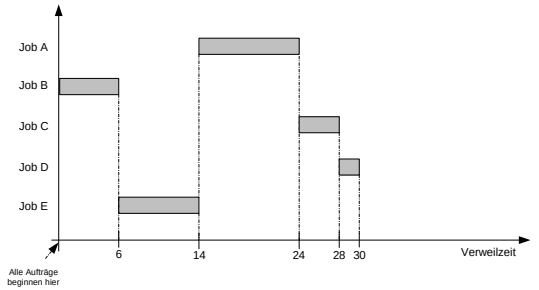
CPU-Scheduling - Fallbeispiele

T. Pospíšek

Gesamtüberblick

1. Einführung in Computersysteme
2. Entwicklung von Betriebssystemen
3. Architekturansätze
4. Interruptverarbeitung in Betriebssystemen
5. Prozesse und Threads
6. **CPU-Scheduling**
7. Synchronisation und Kommunikation
8. Speicherverwaltung
9. Geräte- und Dateiverwaltung
10. Betriebssystemvirtualisierung





Überblick

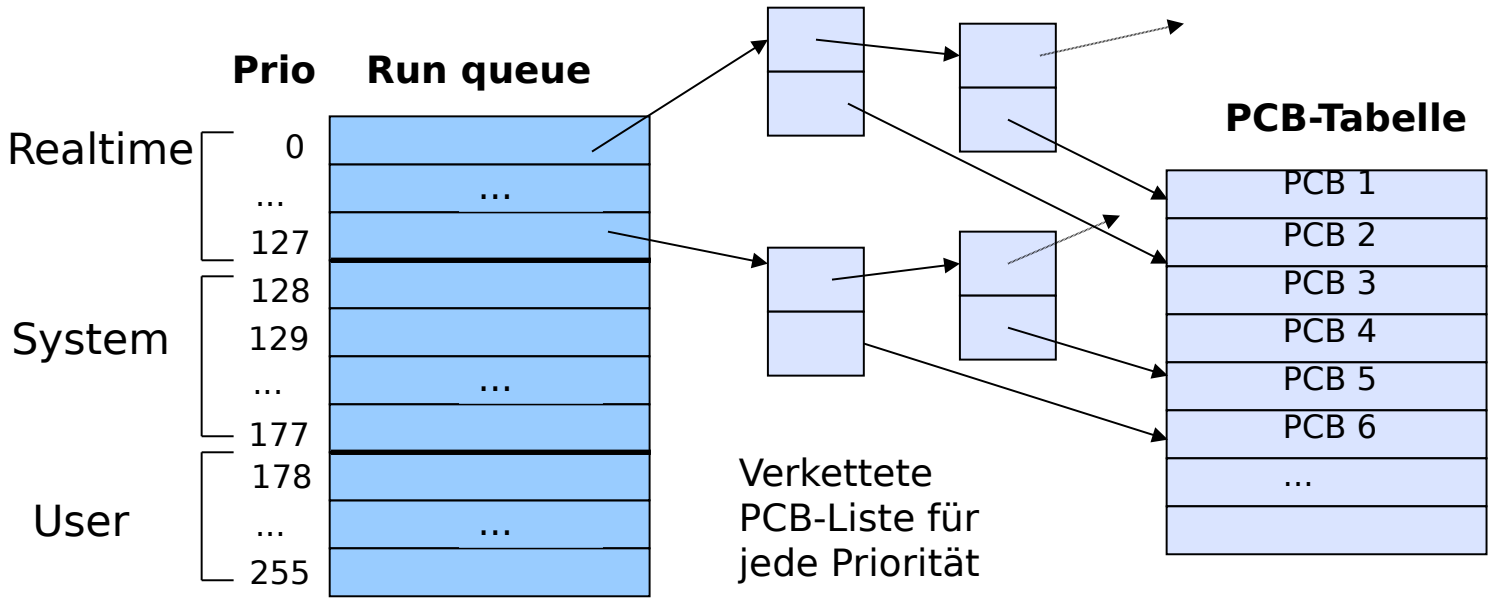
1. **Fallbeispiel: Unix**
2. Fallbeispiel: Linux
3. Fallbeispiel: Windows
4. Fallbeispiel: Scheduling in Java

Scheduling-Strategien unter Unix

- Jedes Unix-Derivat hat Feinheiten in der Scheduling-Strategie
- Es gibt aber prinzipielle Gemeinsamkeiten
- Unterstützte Scheduling-Strategie:
  - **Preemptives** Verfahren (Linux auch non-preemptive)
  - **Prozess als Scheduling-Einheit** in traditionellem Unix
  - **RR mit Prioritäten** und **Multi-Level-Feedback-Scheduling**
    - Eine Queue je Priorität
    - Round Robin innerhalb der gleichen Priorität
    - Vorderster Prozess aus aktuell höchster Priority-Queue wird als nächstes ausgewählt
    - Nach Ablauf der Zeitscheibe wird der Prozess ans Ende seiner aktuellen Queue oder ans Ende einer anderen Queue gehängt

Run-Queue unter Unix

- Multi-Level-Feedback-Warteschlangen organisiert in der sog. Run-Queue



PCB = Process Control Block

Scheduling unter Unix: Prioritätsberechnung

- Jeder Prozess erhält eine **initiale Priorität** beim Start, die sich mit der Zeit verbessert
- Priorität in früheren Unix-Systemen: -127 (höchste) bis +127
- Priorität in heutigen Unix-Systemen: z.B. von 0 (höchste) bis 255
- Die Priorität wird zyklisch, z.B. jede Sekunde, neu berechnet (System V)
- Als Parameter geht die **CPU-Nutzung** der Vergangenheit in die Berechnung ein
  - War diese in der letzten Zeit hoch, wird die Priorität niedriger
  - I/O-intensive Prozesse (Dialogprozesse) werden bevorzugt

Einschub: nice-Befehl unter Unix

- **nice-Befehl** beeinflusst die statische Priorität beim Starten eines Programms:
  - **Nett** zu anderen Prozessen sein, da die eigene Priorität meist herabgesetzt wird
  - Nice-Prioritätenskala: von 20 bis -19 (höchste)→ je größer der nice-Wert, desto niedriger die Priorität)
- Nutzung: *nice -<nicewert> <command>*
  - Programm <command> wird mit einer um <nicewert> niedrigeren Priorität als der Standardwert gestartet
  - Vorsicht: Syntax je nach Shell etwas anders!
  - Test: nice -10 bash
  - Nur der User mit root-Berechtogung darf die Priorität erhöhen
- Andere Befehle/Systemcalls
  - **renice**: Dient zum Verändern der Priorität eines laufenden Prozesses
  - **setpriority**: Neuer Befehl anstelle von nice

Überblick

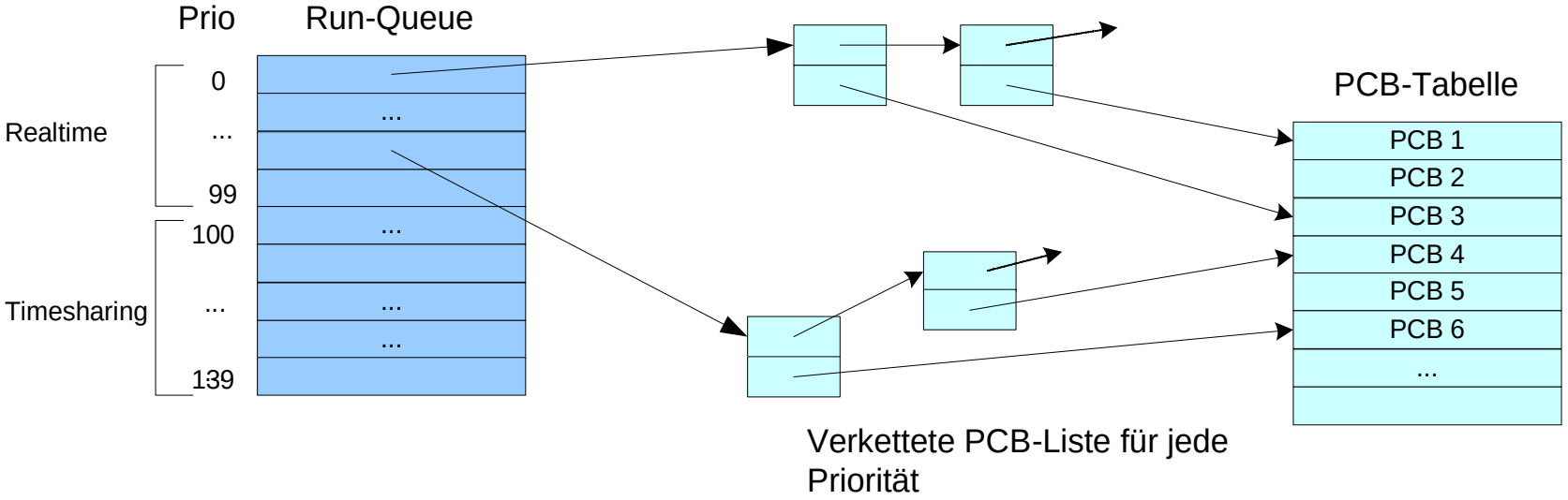
1. Fallbeispiel: Unix
2. **Fallbeispiel: Linux**
3. Fallbeispiel: Windows
4. Fallbeispiel: Scheduling in Java

Scheduling-Strategien unter Linux: O(1)-Scheduler (bis Kernel-Version 2.6.22)

- **RR mit Prioritäten** und **Multi-Level-Feedback-**Mechanismus
- Scheduling-Einheit ist der Thread (Implementierung auf Kernelebene) → Thread und Prozess sind identisch!
- Unterstützte Scheduling-Strategien: „ps -c“ → zeigt Strategien der Threads/Prozesse an!
  - **Realtime FIFO** (POSIX) → SCHED_FIFO
    - Höchste Priorität und **non-preemptive (!)**
  - **Realtime Round Robin** (POSIX) → SCHED_RR
    - Wie FIFO aber **preemptive**, Nutzung einer Zeitscheibe
  - **Timesharing**
    - Standard-Threads
    - SCHED_BATCH, SCHED_OTHER und SCHED_IDLE
- Anm: **Keine echten Realtime-Threads**, sondern P.1003.4-Konformität (Unix real-time extension)

O(1)-Scheduler: Run-Queue unter Linux (1)

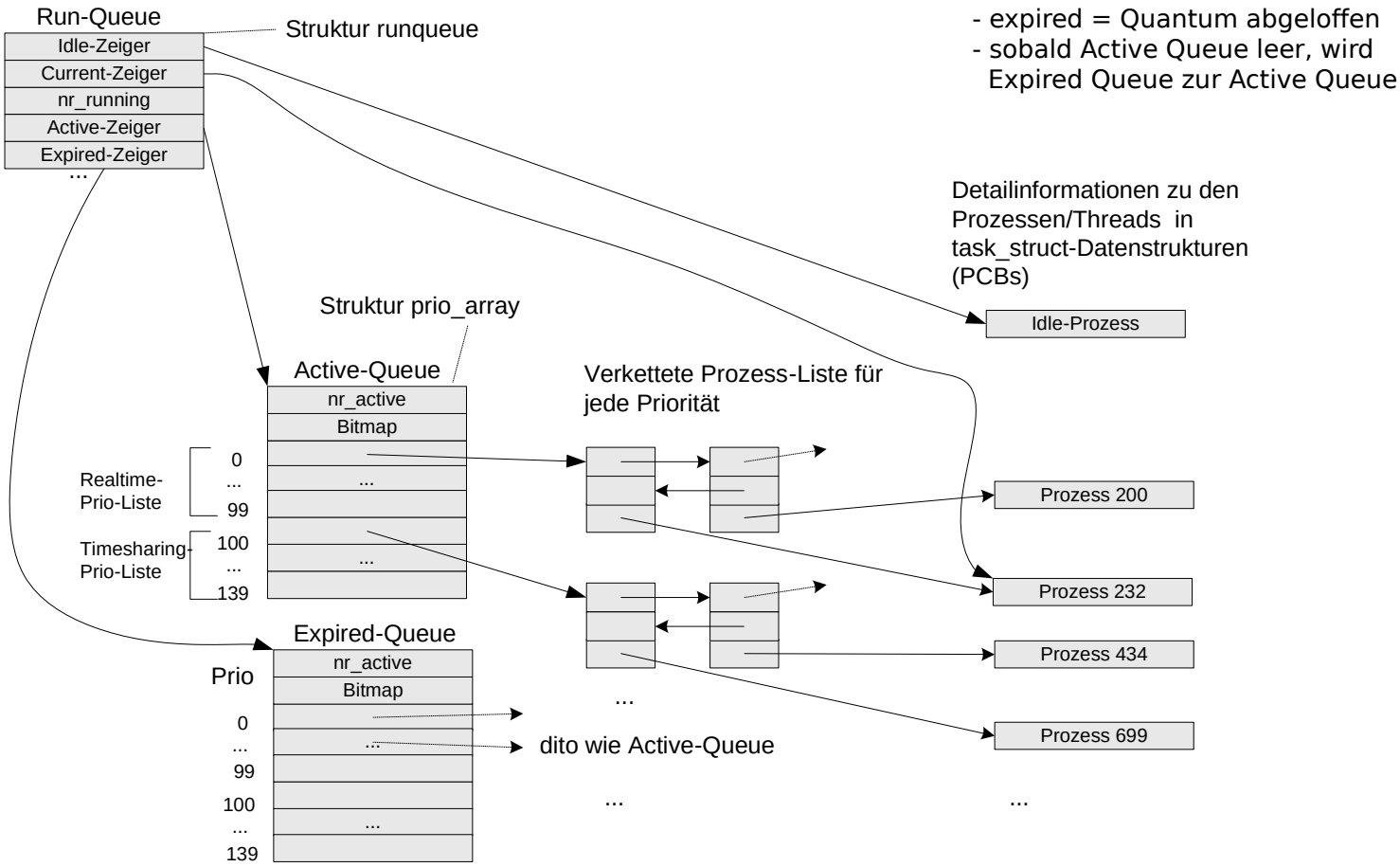
- Multi-Level-Feedback-Warteschlangen werden auch unter Linux über eine „Run-Queue“ verwaltet



Hinweis zum O(1)-Scheduler: Bezeichnung deshalb, weil die Laufzeit für die Auswahl des nächsten Prozesses unabhängig von aktueller Prozessanzahl ist

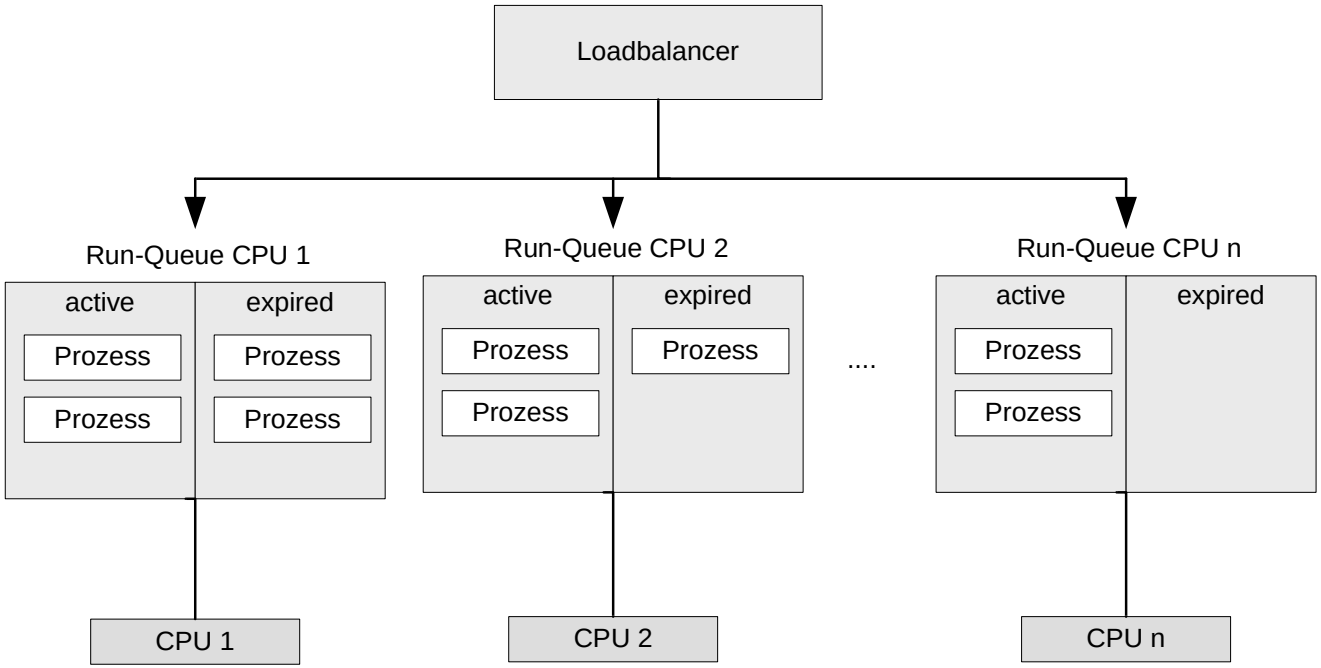
O(1)-Scheduler: Run-Queue unter Linux (2)

- Active Queue und Expired Queue



O(1)-Scheduler: Loadbalancer unter Linux

- Je Prozessor gibt es eine **Run-Queue**
- **Loadbalancer** verteilt die Arbeit



O(1)-Scheduler: Arbeitsweise des Linux-Schedulers

- Die CPU wird einem Thread **entzogen**, wenn
  - die Zeitscheibe (Quantum) abgelaufen ist („expired“)
  - der Thread blockiert (Warten auf Ressource)
  - ein Thread mit höherer **Priorität** „ready“ wird
    - d.h. zwischendurch muss dies geprüft werden
    - Prüfung geschieht nach jedem Tick bei der Systemclock-Interrupt-Bearbeitung
- Die Quanten der Prozesse der höchsten Priorität werden soweit möglich abgearbeitet (außer wenn Prozess blockiert ist), danach wird die nächst niedrige Prioritäts-Queue bearbeitet
  - → Echtzeitprozesse können die anderen behindern

Einschub: Skizze einer Timer-Interrupt-Routine

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering

```
timer_interrupt()        // Interrupt Service Routine für Systemuhr
{
    Systemuhr anpassen;
    Statistiken in Kerneldatenstrukturen aktualisieren;
    Quantumszähler aktiver Prozesse reduzieren;


    if (Prozess mit  höherer Priorität  im Zustand „ready“) {
        schedule();        // Dispatcher  aufrufen
    }
    else if  (Quantum des laufenden Threads == 0) {
        schedule();        // Dispatcher aufrufen
    }
    ...
}
```

Linux-Prioritäten

- Jedem Thread wird eine **statische** und eine **dynamische** (effektive) **Priorität** zugeordnet
  - Prioritätenskala intern: 0 bis 139
  - Timesharing-Prioritäten liegen im Bereich von 100 bis 139 (100 ist höchste)
    - Standardwert = 120
    - nice-/setpriority-Kommando ermöglicht eine Veränderung der statischen Priorität zwischen -20 und +19 (wie unter Unix)
  - Priorität 0 bis 99 sind Echtzeitprioritäten
- Höhere Priorität → mehr CPU-Zeit (höheres Quantum) für den Prozess
- Veränderung der effektiven Priorität führt zum Einhängen des Prozesses in andere Queue

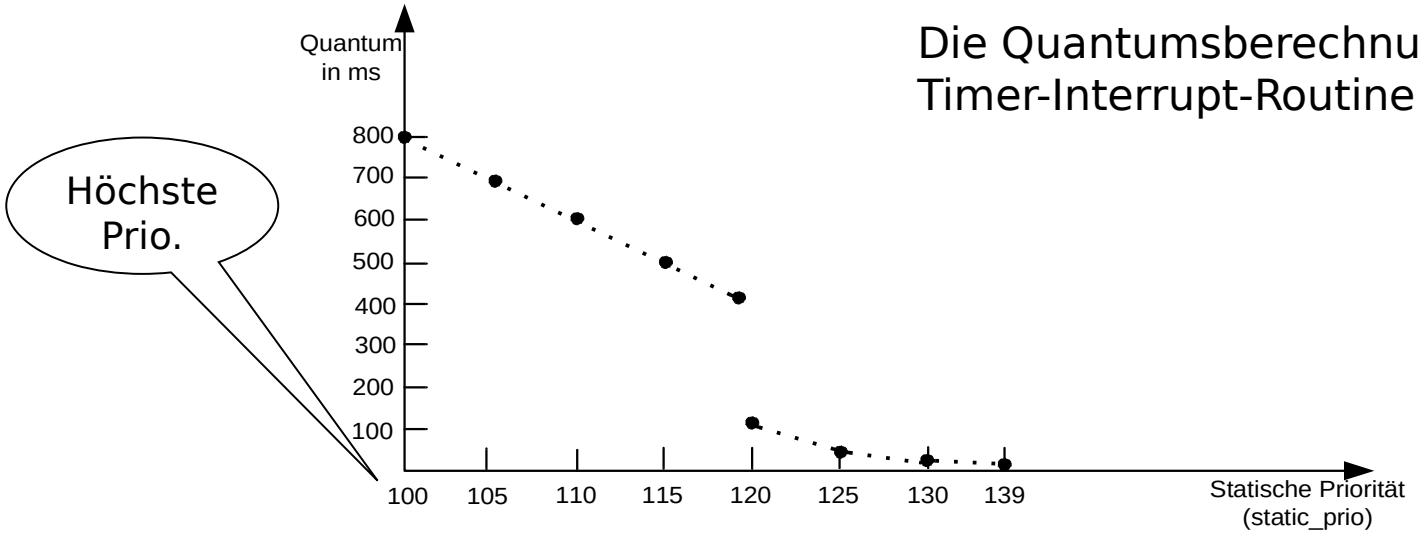
O(1)-Scheduler: Quantumsberechnung unter Linux

- **Timesharing:** Quantumsberechnung auf Basis der statischen Priorität in [ms]

  quantum = f(static_prio) = (140 – static_prio) * 20, falls static_prio < 120

  quantum = f(static_prio) = (140 – static_prio) * 5, falls static_prio >= 120

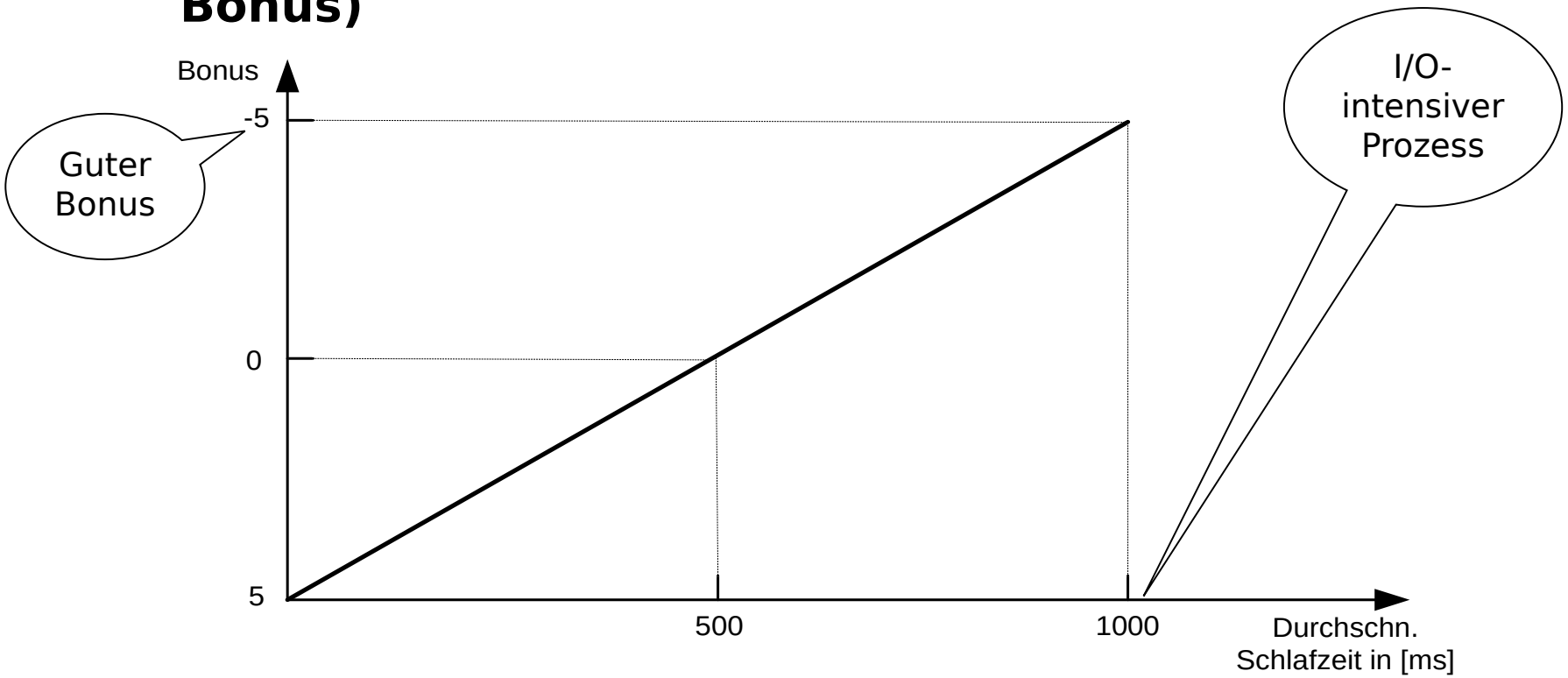
- Beispiel: f(100) = (140 - 100) * 20 = 800 [ms]



Die Quantumsberechnung z.B. in Timer-Interrupt-Routine

O(1)-Scheduler: Bonuszuteilung unter Linux

- Bonus für Priorität in Abhängigkeit der durchschnittlichen Schlafzeit
  - Negativer Bonus → Verbesserung der Priorität
  - Prio-Veränderung → Einordnen in andere Queue
- **Effektive Priorität ~ (Statische Priorität + Bonus)**



O(1)-Scheduler: Zusammenfassung

- Statische Prioritäten bestimmen die CPU-Zuteilung und die Quantumszuteilung
- Die effektive Priorität bestimmt die Einordnung in der Run-Queue
- Rechenintensive Threads verbrauchen ihr Quantum schnell und erhalten dann ein Quantum abhängig von ihrer statischen Priorität und einen positiven Bonus (schlecht!) auf die effektive Priorität
  - → also beeinflusst die statische Priorität das Quantum und die Rechenintensität die Priorität
- Durchschnittliche Schlafzeit beeinflusst die Entscheidung, ob ein Prozess interaktiv (I/O-intensiv) oder rechenintensiv ist
- I/O-intensive Threads erhalten einen negativen (gut!) Bonus auf die effektive Priorität und werden damit früher zugeteilt

  → Bonuszuteilung für *müde* Prozesse → höhere Priorität

Completely Fair Scheduler CFS: Überblick

- Löst O(1)-Scheduler ab **Linux-Version 2.6.23** ab
  - https://en.wikipedia.org/wiki/Completely_Fair_Scheduler
- Scheduler für Timesharing-Prozesse
- Modul fair.c, ca. 4.400 SLOC, implementiert CFS
- Modul rt.c: nur noch ca. 1700 SLOC, implementiert nur noch Realtime-Scheduling mit nur noch 100 Prioritäten
- Besonderheiten:
  - **Keine** Statistiken, **keine** Run Queue und kein Switching von active nach expired Queue
  - **keine** Quanten (Zeitscheiben)
- Hinweis: CFS ist umstritten, da evtl. Nachteile für Workstations, aber Vorteile für Server

Completely Fair Scheduler CFS: Strategie

- **Einfache** Strategie:
  - Für jeden Prozess/Thread wird ein **vruntime**-Wert (Nanosekunden-Basis) verwaltet
  - **vruntime** enthält die Entfernung von der idealen CPU-Nutzung
  - Beispiel: 5 Prozesse → 20 % CPU je Prozess
  - Prozess mit niedrigster **vruntime** wird als nächstes gewählt und bleibt so lange aktiv, bis wieder ein fairer Zustand erreicht wurde
  - **Ziel**: Wert von **vruntime** für alle Prozesse **gleich** halten
- Hinweis:
  - Linux-Kommando chrt -p <pid> liefert Scheduling-Policy und Priorität eines Prozesses und man kann mit chrt auch die Scheduling-Policy eines Prozesses setzen

Überblick

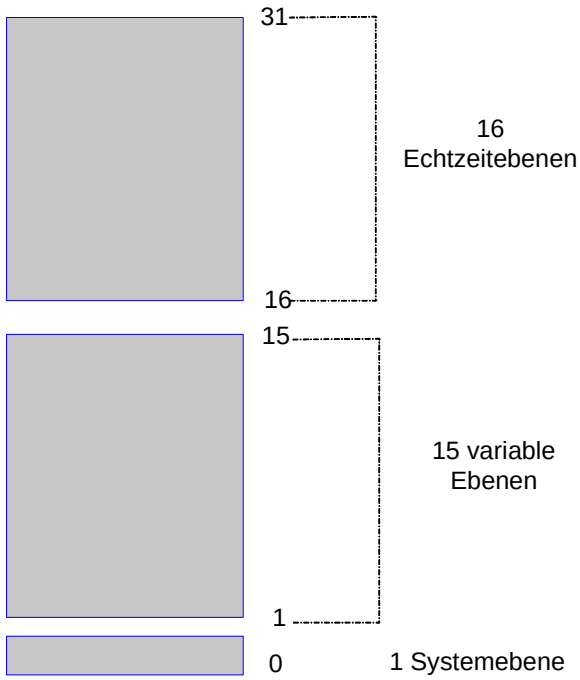
1. Fallbeispiel: Unix
2. Fallbeispiel: Linux
3. **Fallbeispiel: Windows**
4. Fallbeispiel: Scheduling in Java

Scheduling-Strategien unter Windows

- Windows verwendet ein
  - prioritätsgesteuertes,
  - preemptives Scheduling
  - mit Multi-Level-Feedback
- Threads dienen als Scheduling-Einheit

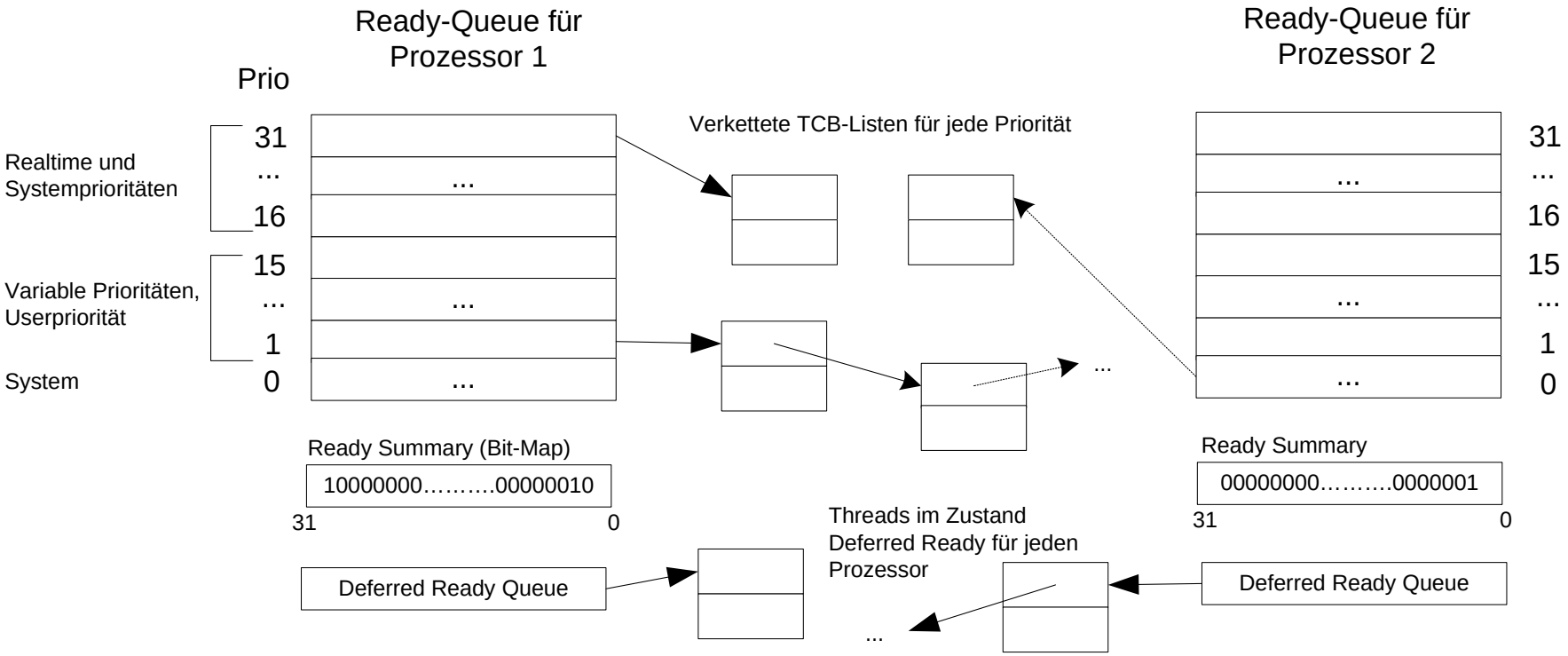
Prioritäten unter Windows: Kernelprioritäten

- Threads werden nach Prioritäten in Multi-Level-Feedback-Warteschlangen organisiert
- Interne Kernelprioritäten 0 (niedrigste) bis 31
  - Nullseiten- und Idle-Thread haben die Priorität 0 (siehe Speicherverwaltung)
  - Prioritäten 16 – 31 (Echtzeit) **verändern sich nicht**
  - Prioritäten 1 – 15 sind **dynamisch** (Benutzerprozesse)



Run-Queue unter Windows

- Multi-Level-Feedback-Warteschlangen



Deferred Ready: Threads, die schon einem Prozessor zugeordnet sind, aber noch nicht aktiv sind

Prioritäten unter Windows: WinAPI-Prioritäten

- Es gibt 6 **WinAPI-Prioritätsklassen der Prozesse** (Basisprioritäten)
  - Werte sind: idle, below normal, normal, above normal, high und real-time
- ... und (relative) **Threadprioritäten**
  - Werte sind: time critical, highest, above normal, normal, below normal, lowest, idle
- Aufruf von ***SetPriorityClass()***
  - bewirkt die Veränderung der Prioritätsklasse für alle Threads eines Prozesses
- Aufruf von ***SetThreadPriority()***
  - Threadpriorität kann aktiv verändert werden
  - Nur innerhalb eines Prozesses

Prioritäten unter Windows: Mapping

- WinAPI-Prioritäten werden auf die internen Kernelprioritäten abgebildet

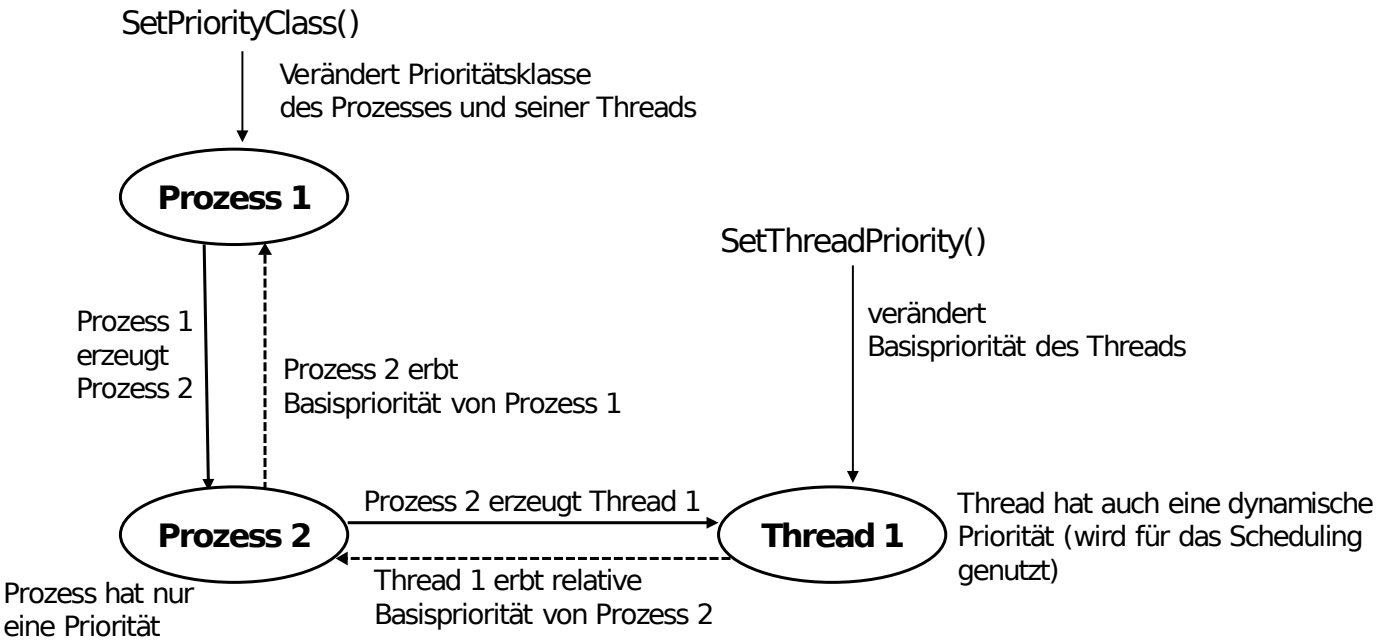
	Prioritätsklasse			
Windows-Thread-Priorität	real-time	high	above normal	normal
time critical	31	15	15	15
highest	26	15	12	10
above normal	25	14	11	9
normal	24	13	10	8
below normal	23	12	9	7
lowest	22	11	8	6
idle	16	1	1	1

...

- Hinweis: Prioritätsklasse eines Prozesses kann mit dem Taskmanager verändert werden
- Testen: Testen Sie das Kommando „start /high notepad“, geht auch über Task Manager

Prioritäten unter Windows: Vererbung

- Ändert man die Priorität eines Prozesses, werden die Prioritäten der Threads ebenfalls geändert
- System-Prozesse nutzen speziellen Systemcall *NTSetInformationProcess*
- Taskmanager zeigt nur Basispriorität



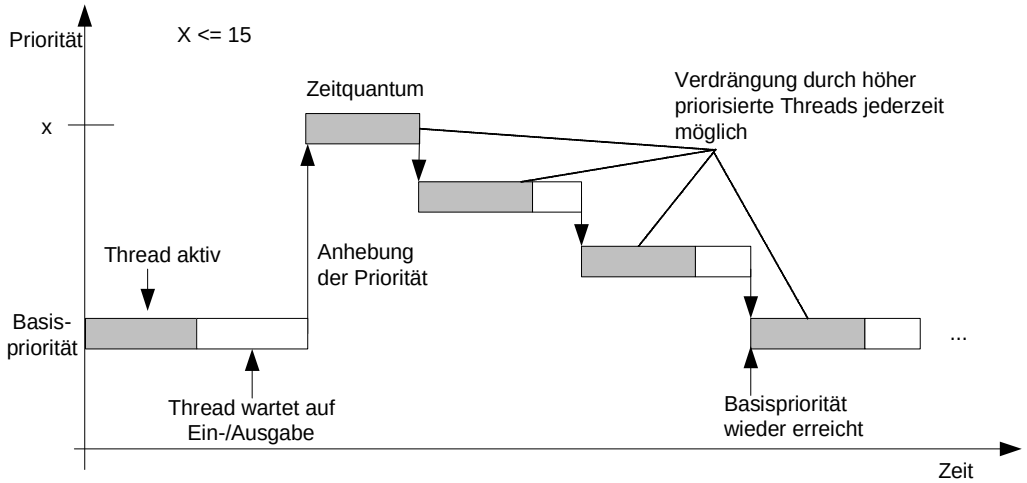
Clock-Intervall und Quantum

- Clock-Intervall
  - ca. 10 ms bei x86 Singleprozessoren
  - ca. 15 ms bei x86 Multiprozessoren
  - siehe *clockres-Tool* zum Feststellen des Clock-Intervalls
- Quantum
  - Quantumszähler je Thread
    - Auf 6 bei Workstations (Windows 2000, XP, ...) eingestellt
    - Auf 36 bei Windows-Servern eingestellt
    - Quantumszähler wird je Clock-Interrupt um 3 reduziert

  → Quantumsdauer = 2 (Workstation) bzw. 12 (Server) Clock-Intervalle
    - 2 * 15 ms = 30 ms bei x86-Workstations
    - 12 * 15 ms = 180 ms bei Servern

Arbeitsweise des Schedulers: Szenario 1

- Szenario: **Priority Boost** (Prioritätsanhebung) für Threads nach einer Ein-/Ausgabe-Operation
  - Anhebung max. auf Priorität 15
  - Treiber entscheidet
    - Maus-/Tastatureingabe: +6
    - Ende einer Festplatten-I/O: +1

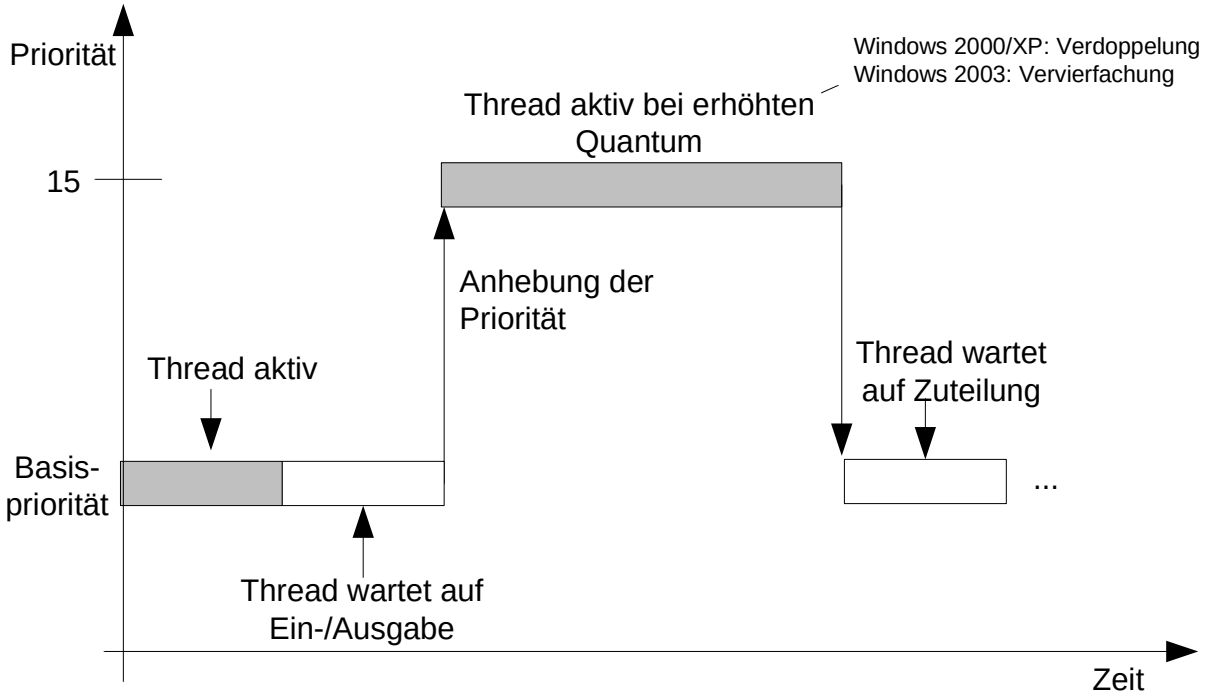


Arbeitsweise des Schedulers: Szenario 2

- Szenario: **Rettung verhungernder Threads**
- Threads mit niedrigerer Priorität **könnten** benachteiligt werden
  - Rechenintensive Threads höherer Priorität kommen immer vorher dran
- Daher: Jede Sekunde wird geprüft, ob ein Thread schon länger nicht mehr dran war, obwohl er bereit ist
  - Ca. 4 Sekunden nicht mehr dran?
  - Wenn ja, wird er auf Prio. 15 gehoben und sein Quantum wird vervierfacht (ab Windows 2003) → Prüfen
- Danach wird er wieder auf den alten Zustand gesetzt
- Also: **Ein Verhungern wird vermieden!**

Arbeitsweise des Schedulers: Szenario 3

- Szenario: **Rettung verhungernder Threads**



Einschub: Vergleich Linux - Windows

Kriterium	Windows	Linux O(1)-Scheduler
Unterstützte Prozesstypen	Timesharing- und Echtzeitprozesse; Unterscheidung über Priorität	Timesharing, Realtime mit FIFO, Realtime mit RR; Unterscheidung über Priorität
Prioritätsstufen	Interne Kernelprioritätsstufen von 0 – 31: 0 – 15: Timesharing-Threads 16 – 31: Realtime- und System-Threads Eigene WinAPI-Prioritäten mit Basisprioritäten für Prozesse und Relativprioritäten für Threads	Statische und effektive Prioritäten mit Stufen von 0 – 139 (139 niedrigste): 0 – 99: Realtime-Prozesse 100 – 139: Timesharing-Prozesse
Scheduling-Strategie für Timeharing-Prozesse	Thread-basiert, Begünstigung von interaktiven vor rechenintensiven Threads Quanten, prioritätsgesteuert, Round Robin, Multi-Level-Feedback-Queue, 32 Queues	Prozess-basiert, Begünstigung von interaktiven vor rechenintensiven Prozessen (Thread = Prozess) Quanten, prioritätsgesteuert, Round Robin, Multi-Level-Feedback-Queue, 140 Queues
Prioritätenberechnung für Timesharing-Prozesse	Aktuelle Priorität wird zur Laufzeit berechnet, Priority-Boost bei wartenden Threads nach Wartezeit oder bei GUI-Threads	Effektive Priorität wird zur Laufzeit berechnet, abhängig von statischer Priorität und einem Bonus (+/- 5) für lang schlafende Prozesse; Effektive Prio = statische Prio + Bonus (vereinfacht)
Quantumsberechnung für Timesharing-Prozesse	Quantumszähler = 6 bei Workstations und 36 bei Windows Servern, bei jedem Clock-Intervall wird Zähler um 3 dekrementiert; Clock-Intervall ist ca. 10 ms bei x86-Singleprozessoren (100 Hz) und ca. 15 ms (67 Hz) bei Multiprozessoren Quantumserhöhung bei GUI-Threads oder zum Vorbeugen vor Thread-Verhungern	Abhängig von statischer Priorität Quantum = (140 - statische Prio) * 20 [ms] (oder * 5) → Maximum 800 ms Takt der internen Systemuhr von Linux bis Linux 2.5 auf 100 Hz einstellbar, ab Linux 2.6 auf 100, 250, 300, 1000 Hz; bei jedem Tick wird das Quantum der aktiven Prozesse um das Clock-Intervall reduziert

Einschub: Echtzeit- versus Universalbetriebssysteme

Echtzeitbetriebssystem: Embedded Linux, …	**Universal-Betriebssystem: Windows, Linux, Unix, Mainframes, ...**
Unterstützt harte Echtzeitanforderungen (Garantien, Deadlines)	Unterstützt nur “weiche” Echtzeitanforderungen (keine Garantien, keine Deadlines)
Ist auf den Worst-Case hin optimiert	Optimierung auf die Unterstützung verschiedenster Anwendungsfälle hin, Reaktionszeit nicht im Vordergrund
Vorhersagbarkeit hat hohe Priorität bei der Auswahl des nächsten Tasks	Effizientes Scheduling mit fairem Bedienen von allen Prozessen, insbesondere von Dialogprozessen steht meist im Vordergrund
Meist werden wenige dedizierte Funktionen unterstützt	Viele Anwendungen können auf einem System laufen
Zeitoptimierung wichtig	Durchsatz und Antwortzeitverhalten wichtig

- Universalbetriebssysteme implementieren Mechanismen, die deterministische Umschaltungen erschweren:
  - Schedulingstrategien, Paging, Kernelmodus, Interruptbearbeitung, vorgegebene Zeitgranularitäten,...

Überblick

1. Fallbeispiel: Unix
2. Fallbeispiel: Linux
3. Fallbeispiel: Windows
4. **Fallbeispiel: Scheduling in Java**

Scheduling in Java: Prioritäten

- Scheduling auf Basis von **Priorität und Zeitscheibe**
- Jeder Thread hat eine Priorität
- Mögliche Prioritäten und deren Manipulation:
  - siehe *Attribut Thread.Max.Priority*
  - MIN, NORM, MAX
  - *setPriority()*
  - *getPriority()*
- Thread mit hoher Priorität wird bevorzugt

- Aber: Tatsächliche Priorisierung hängt von der JVM-Implementierung ab

Scheduling in Java: Strategien

- Scheduling ist **preemptive**
  - Thread wird nach einer bestimmten Zeit unterbrochen (Zeitscheibe)
- Scheduling ist „**weak fair**“:
  - Irgendwann einmal kommt jeder Thread dran
- **Eine Queue je Priorität**
  - Thread der ganz vorne in Queue mit höchster Priorität ist, kommt als nächstes dran
  - Prioritäten werden für lange wartende Threads erhöht

Überblick

- ✓ Fallbeispiel: Unix
- ✓ Fallbeispiel: Linux
- ✓ Fallbeispiel: Windows
- ✓ Fallbeispiel: Scheduling in Java

Gesamtüberblick

- ✓ Einführung in Computersysteme
- ✓ Entwicklung von Betriebssystemen
- ✓ Architekturansätze
- ✓ Interruptverarbeitung in Betriebssystemen
- ✓ Prozesse und Threads
- ✓ CPU-Scheduling

7. Synchronisation und Kommunikation
8. Speicherverwaltung
9. Geräte- und Dateiverwaltung
10. Betriebssystemvirtualisierung