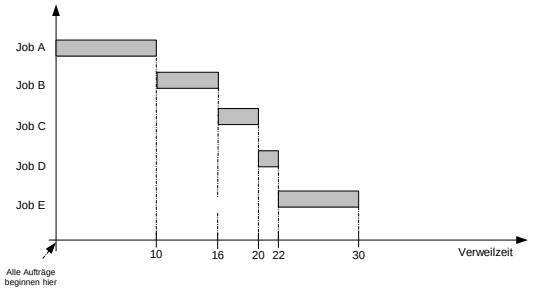
MAS: Betriebssysteme

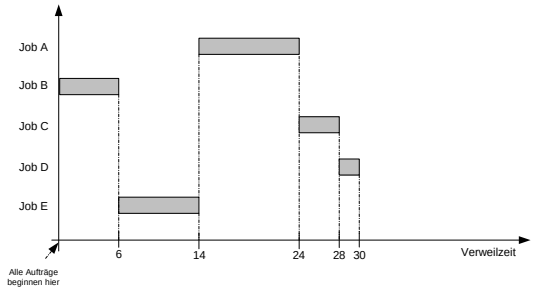
CPU-Scheduling - Fallbeispiele

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: Linux
2. Fallbeispiel: Windows
3. Fallbeispiel: Scheduling in Java

Überblick

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

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

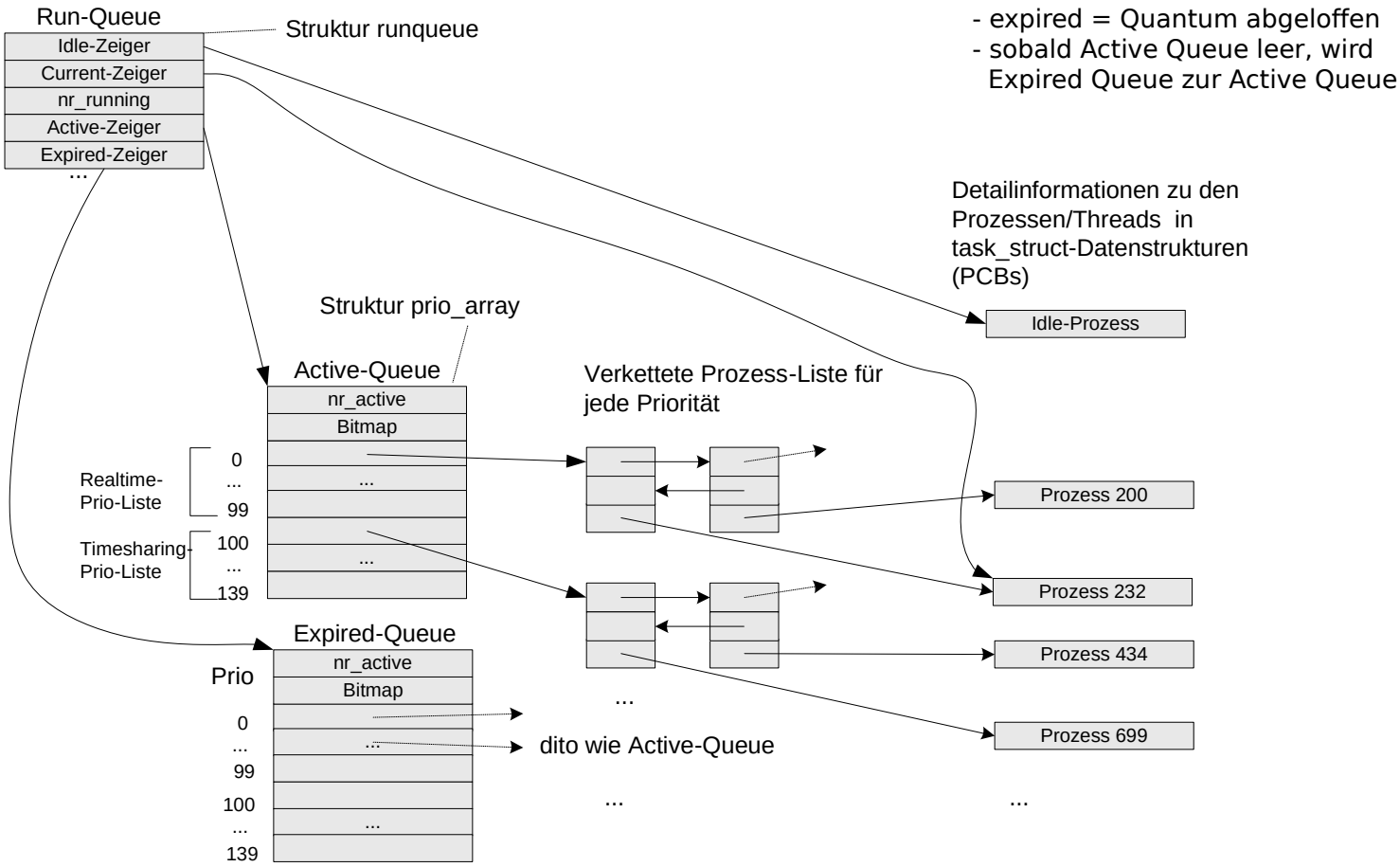
Scheduling-Strategien unter Linux: O(1)-Scheduler (bis Kernel-Version 2.6.22)

Jahr 2007

- **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)

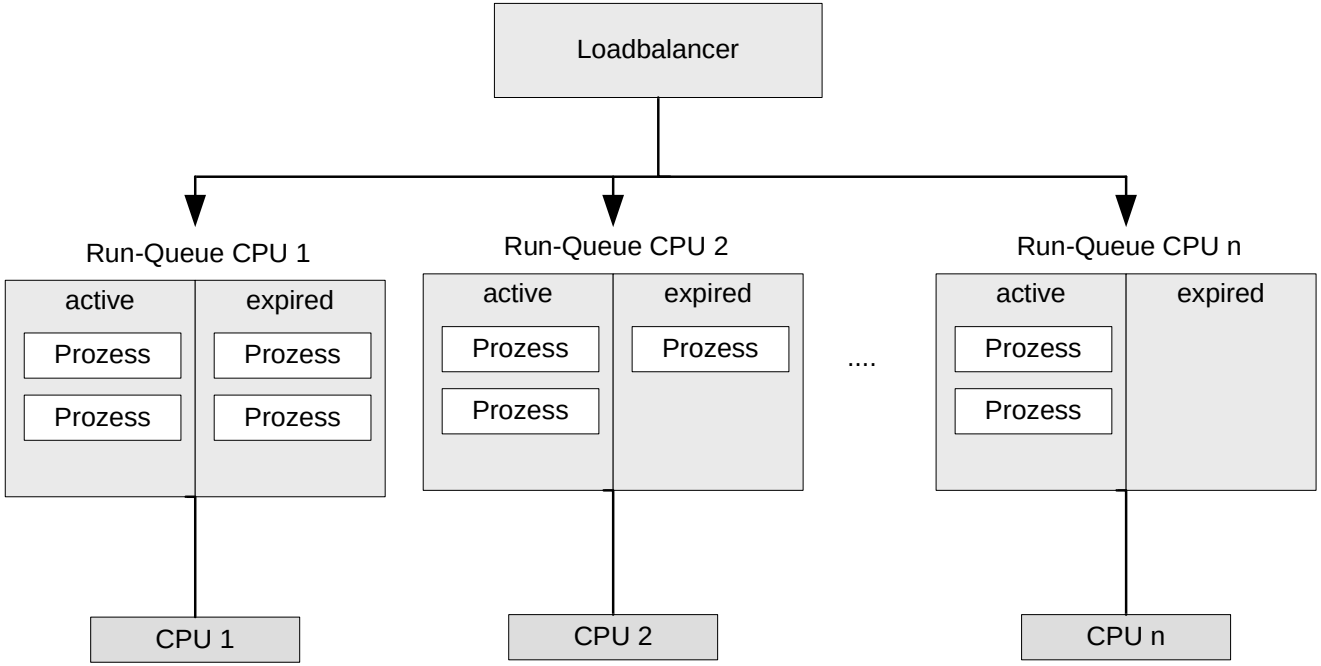
O(1)-Scheduler: Run-Queue unter Linux (2)

- Active Queue und Expired Queue



O(1)-Scheduler: Loadbalancer unter Linux

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



Skizze einer Timer-Interrupt-Routine

```
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
    }
    ...
}
```

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)

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

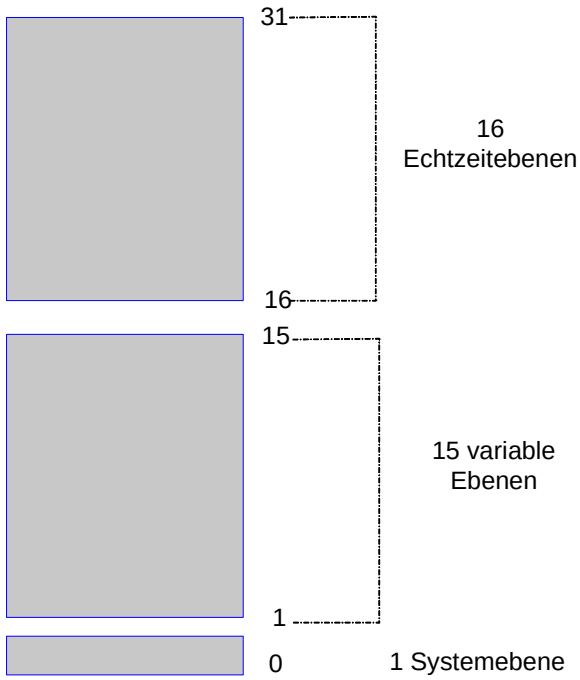
1. Fallbeispiel: Linux
2. **Fallbeispiel: Windows**
3. Fallbeispiel: Scheduling in Java

Scheduling-Strategien unter Windows

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

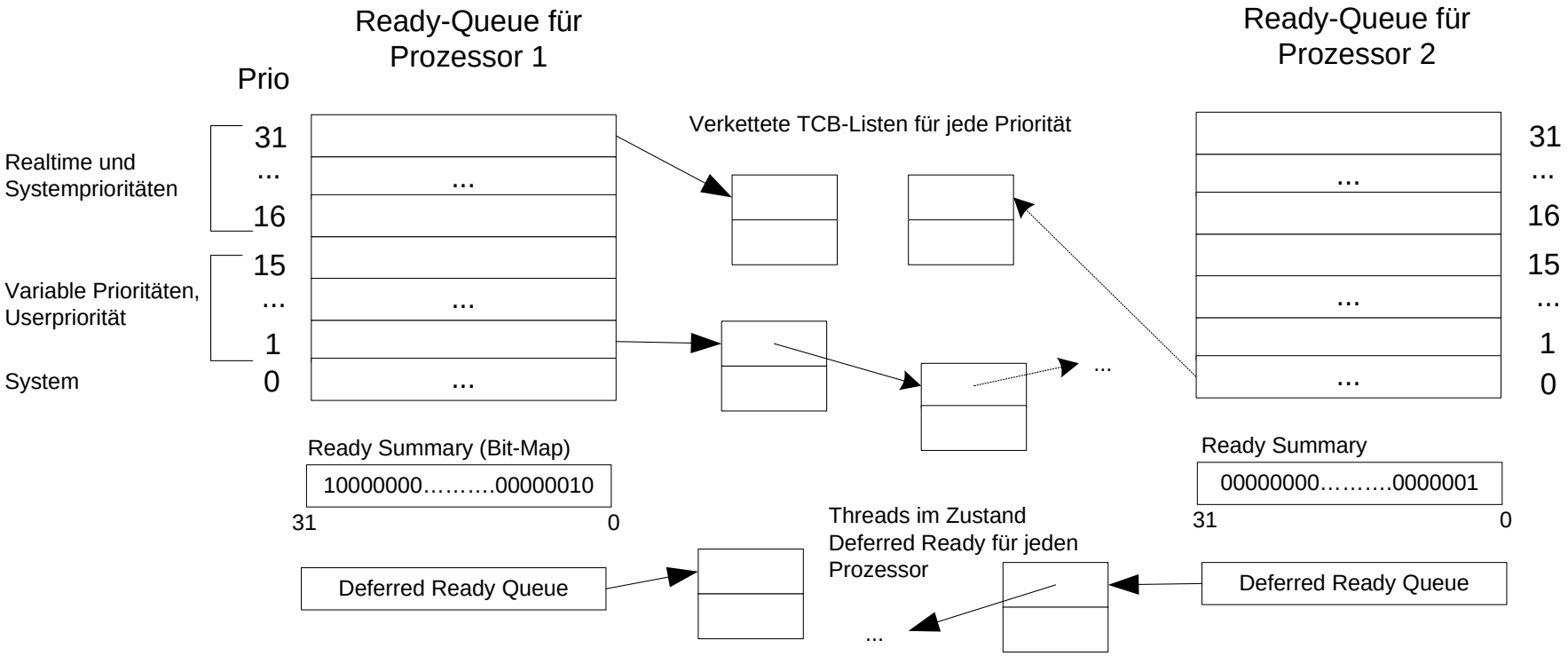
Prioritäten unter Windows: Kernelprioritäten

- 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

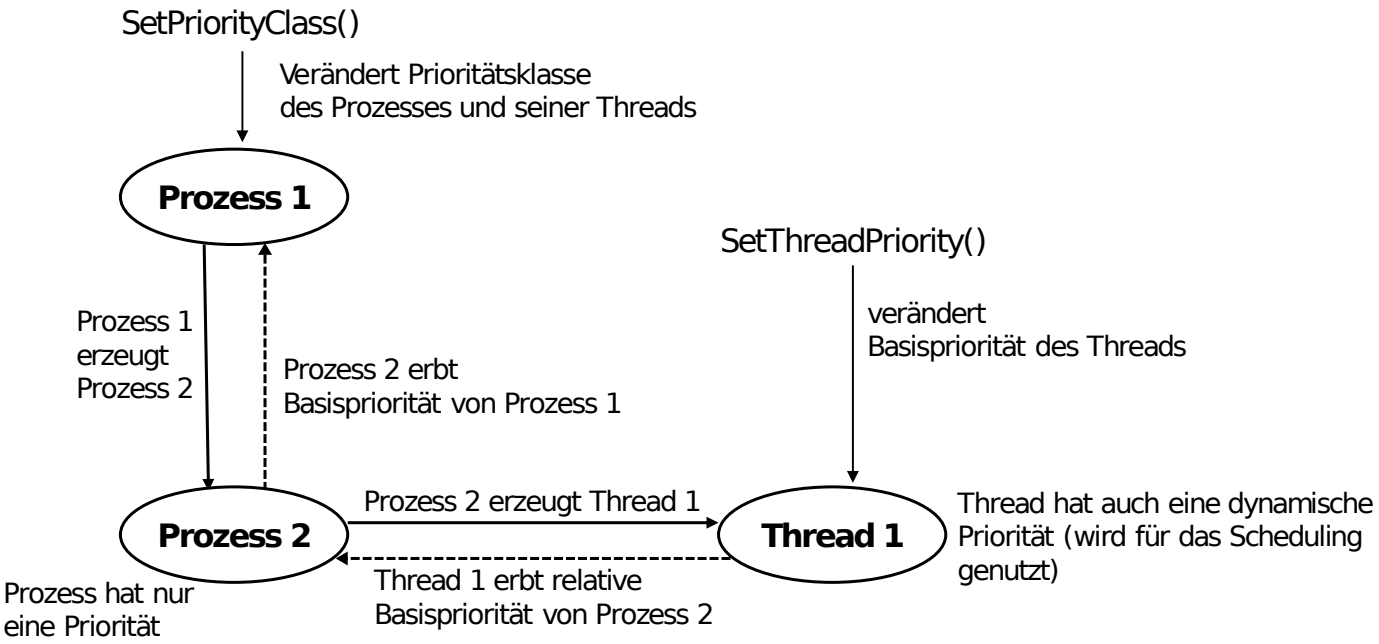
	Prioritätsklasse des Prozesses			
Windows-Thread-Priorität	real-time	high	above normal	normal
time critical	31	15	15	15
highest	26	15	12	10
above normal	25	14	11	9
normal	24	13	10	8
below normal	23	12	9	7
lowest	22	11	8	6
idle	16	1	1	1

...

- 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



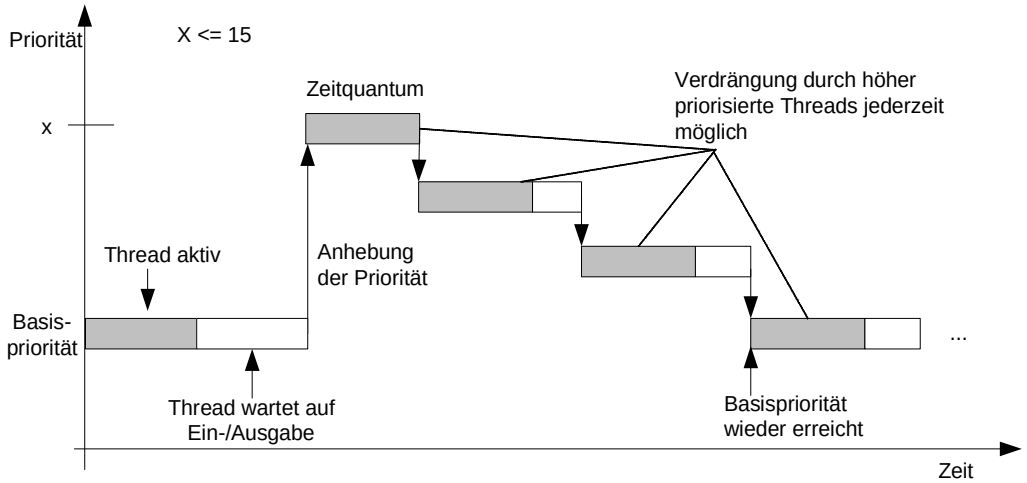
Clock-Intervall und Quantum

- 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

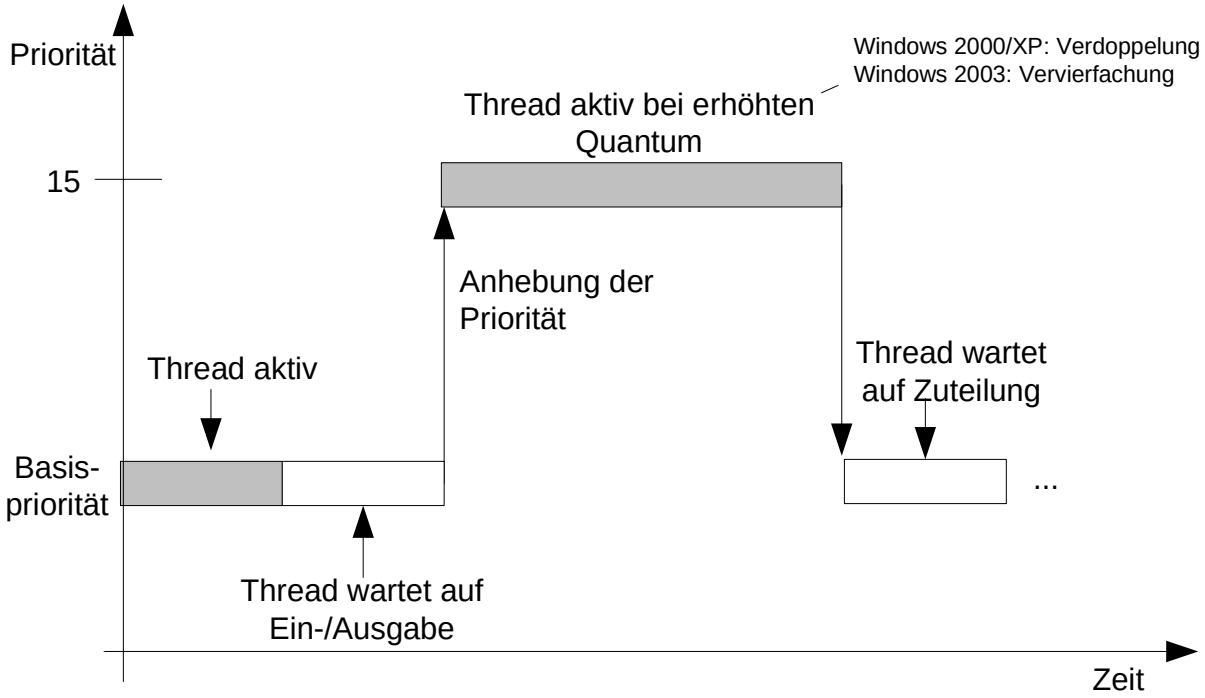


Arbeitsweise des Schedulers: Szenario 2

- 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: 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: 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