Windows 2000 – Scheduler

Konzepte von Betriebssystem – Komponenten

Friedrich – Alexander Universität
Erlangen – Nürnberg
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
Sommersemester 2005

Viktor Witt
Viktor.Witt@informatik.stud.uni-erlangen.de

Gliederung

1. Einleitung

Bei multiprogrammierbaren Rechnern konkurrieren oft mehrere rechenbereite Prozesse zur selben Zeit um eine einzige CPU. Deshalb muss entschieden werden, wann welcher Prozess die CPU bekommt und wie lange er laufen darf. Der Teil des Betriebssystems, der das bewerkstelligt ist der Scheduler und die Strategie nach der er verfährt nennt man Scheduling – Algorithmus.

2. Überblick über Funktionsweise und Begriffe

Windows 2000 implementiert einen **prioritätsgesteuerten** Scheduling-Algorithmus, der sich auf **Thread-Ebene** vollzieht. Es werden nur Threads eingeplant. Unter der Voraussetzung, dass alle Threads mit derselben Priorität ausgestattet sind, bekommt jeder Thread den gleichen Anteil an Rechenzeit, unabhängig davon zu welchem Prozess er letztendlich gehört. Prozesse selbst werden nicht ausgeführt sondern bilden nur Ressourcen und einen Kontext für ihre Threads.
Die Grundidee des prioritätsgesteuerten Scheduling besteht darin, dass jedem Thread eine Priorität zugewiesen wird und der Thread mit der **höchsten Priorität** für eine Zeitscheibe laufen darf. Danach wird erneut geprüft, ob es einen anderen laufbereiten Thread mit derselben oder einer höheren Priorität gibt. Da Windows 2000 – Scheduling **präemptiv** (verdrängend) ist, kann es passieren, dass sich eine gerade in Ausführung befindlicher Thread unterbrochen wird und seine Zeitscheibe nicht vollständig aufbrauchen kann. Dies ist beispielsweise der Fall, wenn ein Thread mit höherer Priorität ausführungsbereit wird. Wenn Windows 2000 einen anderen Thread zur Ausführung bestimmt, erfolgt ein **Kontextwechsel**. Dabei wird der Prozessorstatus des aktiven Threads gesichert und der des anderen Threads geladen.
Für den Code des Windows 2000-Schedulers existiert kein spezielles Modul oder Routine, sondern er ist vielmehr über den ganzen Kernel verteilt. Die Gesamtheit der Routinen, die die Aufgaben des Schedulers übernehmen, nennt man **Verteiler**.

3. Prioritätsstufen

3.1 Windows 2000 interne Prioritäten

Windows 2000 besitzt intern 32 Prioritätsstufen:
- Stufen für die Echtzeit (16 - 32)
- variable Stufen (1 - 15)
- Stufe für Systemzwecke (0)

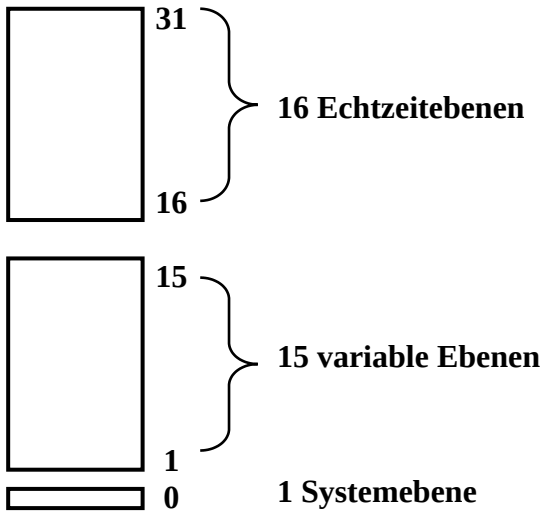


Abbildung 1 *Prioritätsstufen* [1]

3.2 Win32-API-Prioritäten

Mit der Win32-API-Funktion ***SetPriorityClass*** wird jedem Prozess eine **Prioritätsklasse** (Echtzeit, Hoch, Höher als Normal, Normal, Niedriger als Normal, Leerlauf) zugewiesen. Diese Prioritätsklasse stellt für die Threads des Prozesses gleichzeitig die **Basispriorität** dar.
Mit einem weiteren Aufruf ***SetThreadPriority*** wird die **relative Priorität** (Zeitkritisch, Maximum, Höher als Normal, Normal, Niedriger als Normal, Minimum, Leerlauf) den einzelnen Threads innerhalb eines Prozesses zugewiesen.
In der Win32 – API besitzt also jeder Thread eine Priorität, die eine Kombination aus prozessbezogener Prioritätsklasse und relativer Threadpriorität darstellt.

3.3 Abbildung der Win32-API-Prioritäten auf interne System-Prioritäten

		Win32-Prozessklassen-Prioritäten				
Win32-Thread-Prioritäten	**Echtzeit**	**Hoch**	**Über normal**	**Normal**	**Unter normal**	**Leerlauf**
Zeitkritisch	31	15	15	15	15	15
Höchste	26	15	12	10	8	6
Über normal	25	14	11	9	7	5
Normal	24	13	10	8	6	4
Unter normal	23	12	9	7	5	3
Niedrigste	22	11	8	6	4	2
Leerlauf	16	1	1	1	1	1

Tabelle 1 *Abbildung der Win32-Prioritäten auf Windows 2000-Prioritäten* [2]

Die Win32-API-Prioriät eines Threads wird auf den Wert der internen Windows 2000 Priorität abgebildet (Siehe Tabelle 1).
Anders als Benutzer-Threads, die ihre System-Prioritäten mit Hilfe von Win32-API-Funktionen erlangen, verwenden System-Threads für diesen Zweck interne Windows 2000-Funktionen. Diese vergeben den Sytem-Threads meistens höhere System-Prioritäten als der verwendete Standardwert bei Win32-API für die Benutzer-Threads. Das hat zur Folge, dass die System-Threads beim Einplanen oft höher eingestuft werden.

3.4 aktuelle Prioritäten

Neben der Basispriorität besitzt jeder Thread noch zusätzlich eine **aktuelle Priorität**.
Diese kann im dynamischen Bereich (zwischen 1 und 15) höher als die Basispriorität sein, aber niemals kleiner. (Die dynamische Prioritätsanpassung wird später ausführlicher im Kapitel 6 behandelt) Im Echtzeitbereich findet keine dynamische Prioritätsanpassung statt, sodass die aktuelle Priorität immer gleich der Basispriorität ist. Der Scheduler wählt immer den Thread mit der im Augenblick höchsten aktuellen Priorität für die Ausführung aus.

4. Threadzustände

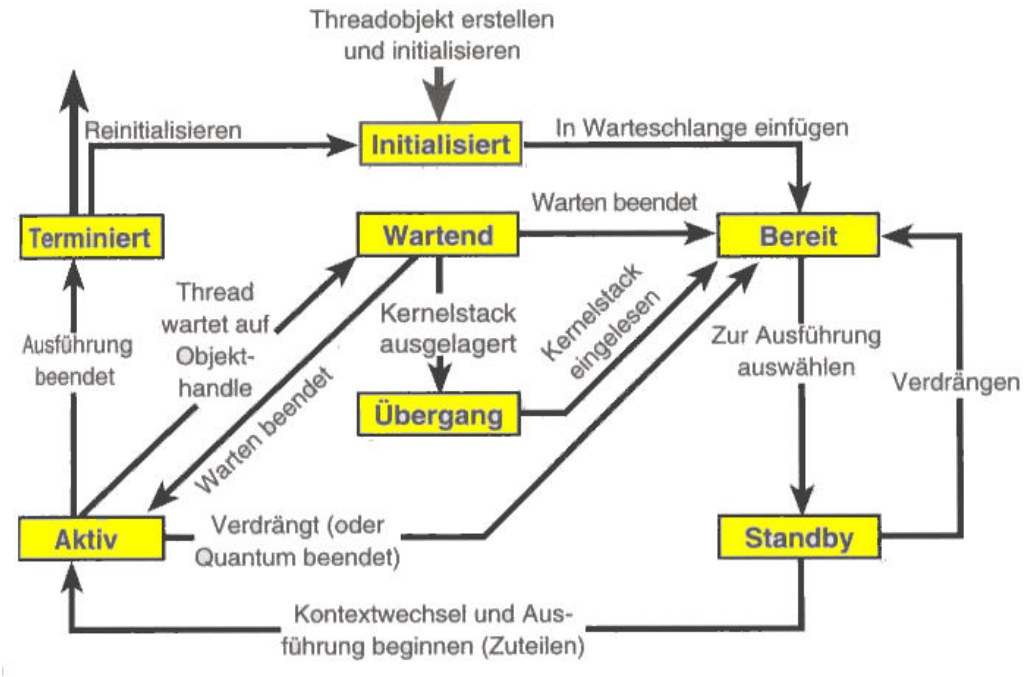


Abbildung 2 *Threadzustände* [1]

- **Bereit**
  Bei der Suche nach einem auszuführenden Thread berücksichtigt der Verteiler nur die Threads im Bereit-Zustand. Diese Threads warten nur darauf, ausgeführt zu werden.

- **Standby**
  Ein Thread in diesem Zustand ist bereits ausgewählt worden, als Nächstes vom Prozessor ausgeführt zu werden.

- **Aktiv**
  Sobald der Verteiler einen Kontextwechsel zu einem Thread ausführt, wechselt der Thread in den Zustand Aktiv und setzt seine Ausführung fort. Die Threadausführung wird so lange fortgesetzt, bis der Kernel sie unterbricht, um einen Thread höherer Priorität auszuführen, oder bis sein Zeitquantum erschöpft ist oder bis er terminiert bzw. freiwillig in den Wartezustand wechselt.

- **Wartend**
  Thread kann in den Wartezustand übergehen wegen:
  - freiwilligem Warten auf ein Objekt zu Synchronisationszwecken (z. Bsp. Mutex oder Semaphore)
  - Ausführung von E/A-Operationen

  Nach dem Wartezustand kann der Thread abhängig von seiner aktuellen Priorität wieder zur Ausführung kommen oder in den Bereit-Zustand wechseln.

- **Übergang**
  Ein Thread wechselt in den Übergangszustand, wenn er ausführungsbereit ist, aber sein Kernelstack aus dem Speicher ausgelagert ist. Nach dem Einlagern des Stacks wechselt er in den Bereit-Zustand.

- **Beendet**
  Ein Thread beendet seine Ausführung und terminiert.

- **Initialisiert**
  Wird intern während der Erstellung eines Threads verwendet.

5. Praxisbeispiele

5.1 Freiwilliger Kontextwechsel

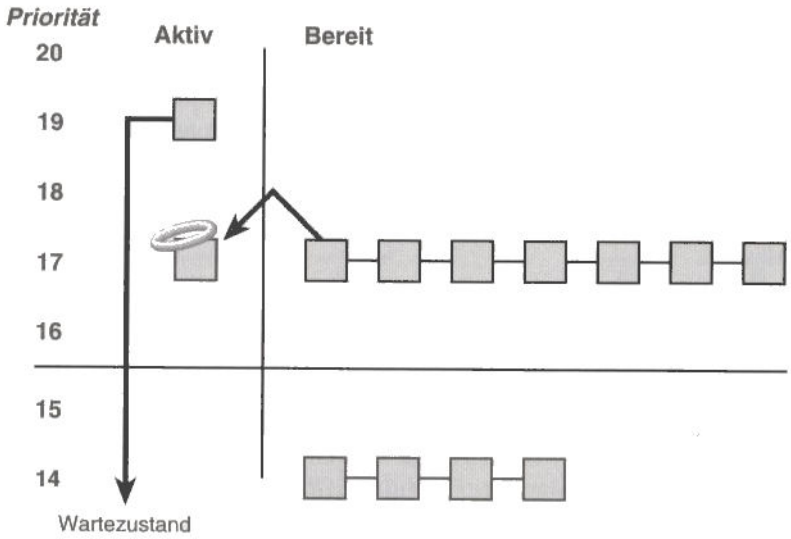
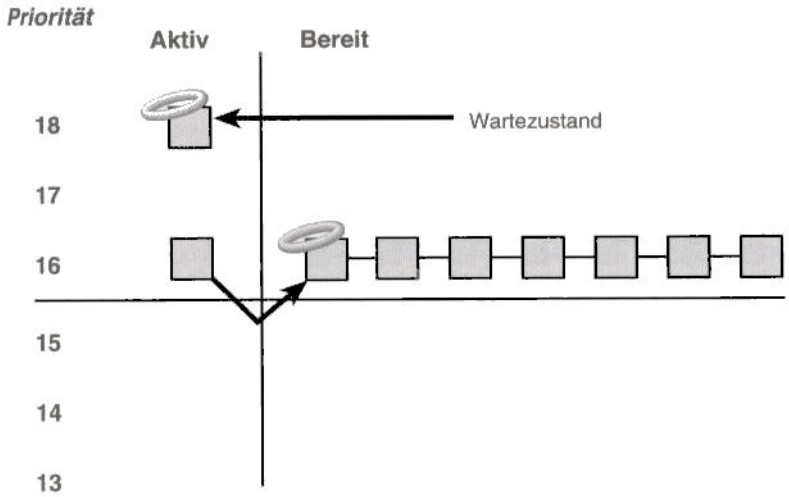


Abbildung 3 *Freiwilliger Kontextwechsel* [1]

Ein Thread kann mittels zahlreicher Win32-API-Wait-Funktionen freiwillig in den Wartezustand wechseln. Die Gründe dafür können verschieden sein: Warten auf ein Ereignis, ein Mutex- oder Semaphorobjekt usw.
Dabei wechselt der Thread in den Zustand Wartend und ein anderer Thread wird aus der Bereit-Warteschlange zur Ausführung ausgewählt. Sobald das Warten des Threads auf ein Objekt beendet ist, wird er an das Ende der Bereit-Warteschlange seiner Priorität gestellt. Da ein Thread aber nach den meisten Warteoperationen eine Prioritätsanhebung erfährt, kann er oft ohne Verzögerung mit seiner Ausführung wieder fortfahren.
Ein ähnliches Szenario ist in der Abbildung 3 vorzufinden, mit einer Ausnahmen, dass das Ganze sich im Bereich der Echtzeitstufen abspielt und es daher keine Prioritätserhöhungen gibt. Hier geht der Thread auf Prioritätsstufe 19 freiwillig in den Wartezustand über. Ein anderer Thread auf Prioritätsstufe 17 findet sich am Anfang der Bereit-Warteschlange, mit der derzeit höchsten Priorität und wird deshalb als nächstes zur Ausführung bestimmt.

4.2 Kontextwechsel durch Verdrängung



Ein Thread mit einer niedrigen Priorität wird immer dann unterbrochen, wenn ein Thread höherer Priorität ausführungsbereit wird. Die Entscheidung darüber, ob der aktuell aktive Thread weiterlaufen darf, oder ein anderer Thread dran kommt muss getroffen werden, wenn die Warteoperation eines Threads höherer Priorität abgeschlossen ist, oder die Priorität eines Threads angehoben oder abgesenkt wird. In Abbildung 4 handelt es sich auf der Prioritätsstufe 16 um einen Tread, der gerade eben in Ausführung war. Seine Ausführung wurde unterbrochen, weil sich ein anderer Thread mit der höheren Priorität 18 aus dem Wartezustand befreit hat.
Wird der aktuelle Thread, in der Abbildung 4 der Thread auf der Prioritätsstufe 16, verdrängt, dann wird er an den Anfang der Bereit-Warteschlange gestellt. Seine Zeitscheibe wird zurückgesetzt, sofern es sich um einen Thread aus dem Echtzeitbereich handelt. Andernfalls darf er beim nächsten Mal sein vorhandenes Quantum aufbrauchen.

4.3 Aufgebrauchtes Quantum

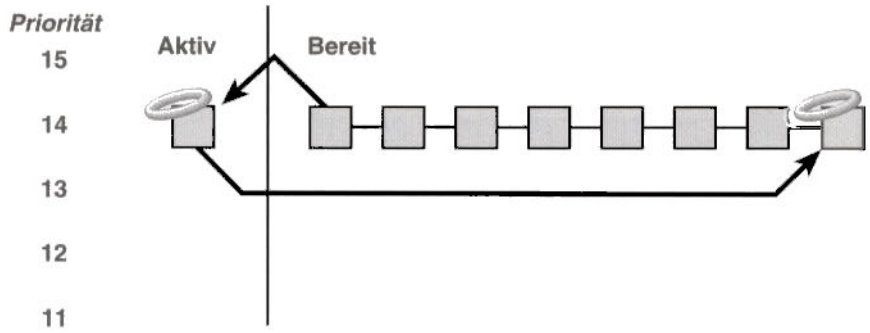


Abbildung 5 *Aufgebrauchtes Quantum* [1]

Wenn ein Thread sein Quantum aufgebraucht hat, muss der Scheduler entscheiden, ob seine Priorität abgesenkt wird und welcher Thread als Nächstes ausgeführt wird. Geeignet ist hierfür ein ausführungsbereiter Thread mit der höchsten Priorität. Der gerade aktive Thread bekommt einen neuen Quantumswert zugeteilt und wird an das Ende der Bereit-

Warteschlange seiner neuen Priorität gestellt. Es ist nicht auszuschließen, dass der gleiche Thread nochmal für eine Zeitscheibe ablaufen darf.

6. Prioritätsanhebungen

6.1 Erhöhung nach dem Abschluss der E/A-Operationen

Windows 2000 kann die aktuelle Priorität eines Threads erhöhen. Die Erhöhung bezieht sich auf die Basispriorität und betrifft nur solche Threads, deren Prioritätsstufen im dynamischen Bereich (0 bis 15) liegen. Die maximale Erhöhung erfolgt bis zur Prioritätsstufe 15 und niemals darüber hinaus.

Um den **E/A-lastigen** Threads mehr Rechenzeit zu gewahren, erfahren sie nach dem Abschluss bestimmter E/A-Operationen eine **temporäre Prioritätserhöhung**. Der tatsächliche Wert der Erhöhung hängt vom **Gerätetreiber** ab.

Gerät	Prioritätsanhebung
Festplatte, CD-ROM, parallele Schnittstelle, Grafik	1
Netzwerk, Mailslot, Named Pipe, serielle Schnittstelle	2
Tastatur, Maus	6
Sound	8

Tabelle 2 *Werte für Prioritätsanhebung* [1]

Nach jedem Ablauf eines Quantums fällt der Thread eine Prioritätsstufe niedriger, bis er das Niveau seiner ursprünglichen Basispriorität erreicht.

6.2 Prioritätsanhebung nach dem Warten auf Ereignisse und Semaphore

Die Priorität solcher Threads, die auf ein Ereignis gewartet haben, wird um 1 erhöht und dann nach dem zu 6.1 identischen Verfahren herabgesetzt.

6.3 Prioritätsanhebung als Mittel gegen das „Verhungern“

Threads niedriger Priorität, die eine bestimmte Zeit im Bereit-Zustand verharren, erfahren eine Prioritätserhöhung auf die Stufe 15. Dort dürfen sie für zwei Zeitscheiben laufen und kehren dann **unmittelbar** zu ihrer Basispriorität zurück.

6.4 Prioritätsanhebung von GUI-Threads und Vordergrundthreads

Threads von Vordergrundprozessen, die sich aus Wartezuständen befreien, sowie Threads, die als Besitzer von Fenstern agieren, erfahren ebenfalls eine Prioritätserhöhung. Dies soll insbesondere **interaktive Anwendungen** begünstigen.

7. Zusammenfassung

Windows 2000 implementiert einen prioritätsgesteuerten Scheduling-Algorithmus.
Die Grundlage hierfür bieten die Threads. Während des Scheduling durchlaufen diese

verschiedene Zustände. Außerdem besitzen sie eine Basis- und eine aktuelle Priorität. Letztere kann unter bestimmten Umständen angehoben und abgesenkt werden. Auf diese Weise werden insbesondere die interaktiven Anwendungen bevorzugt.

8. Quellen

[1] David A. Solomon, Mark Russinovich: Inside Microsoft Windows 2000
3. Auflage, Microsoft *Press*

[2] Andrew S. Tannenbaum: „Moderne Betriebssysteme“
2. Auflage, Person Studium, 2002