6. Scheduling

Michael Schöttner
Betriebssysteme und Systemprogrammierung

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

6.1 Vorschau

- Threads
- Prozess-/Thread-Wechsel
- Scheduling: FCFS, SJF, RoundRobin
- Multilevel-Scheduling
- Feedback-Scheduling
- Echtzeit-Scheduling
- Fallbeispiele: UNIX System V, BSD 4.3, Linux, Microsoft Windows

6.2 Begriffe

- **Programm**: statische Darstellung eines Algorithmus
  (ausführbare Datei auf dem Hintergrundspeicher)
- **Adressraum:** Der zu einem Zeitpunkt direkt zugreifbarer Speicher
- **Prozess**: Adressraum, Thread(s) und Verwaltungsinformationen
  (Programm, das sich in Ausführung befindet, und seine Daten)
- **Thread**: selbständige Aktivität
  - gehört immer zu einem Prozess
    → teilt sich Daten, Code & Betriebsmittel mit Threads des gleichen Prozesses
  - separater Registersatz & eigener Laufzeitkeller
  - (Manchmal auch als lightweight process bezeichnet)

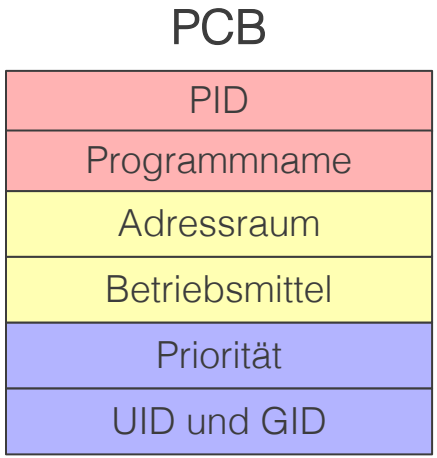
6.2 Begriffe

- **Nebenläufigkeit**, bezogen auf einen Core
  - **Zu einem Zeitpunkt ist immer ein Thread aktiv**
  - Ziel: Wartezeiten anderweitig zu nutzen

- **Parallelität**, bezogen auf mehrere Cores
  - **Zu einem Zeitpunkt sind mehrere Threads aktiv**
  - Ziel: wie oben und zusätzlich möglichst alle Cores simultan nutzen
  - Zuteilung eines Cores ist i.d.R. nichtdeterministisch
    (kann auch gesteuert werden → pinning)

6.3 Prozess

- Verwaltung erfolgt im Kernel-Space durch einen **Prozesskontrollblock** (engl. process control block = PCB)
  - Prozess-ID und Programmname
  - Zustandsinformation:
    - Beschreibung der belegten Adressraumbereiche
    - Betriebsmittel: z.B. geöffnete Dateien u. Geräte
  - Verwaltungsdaten für Scheduler:
    - Priorität
    - UID und GID → Berechtigungen



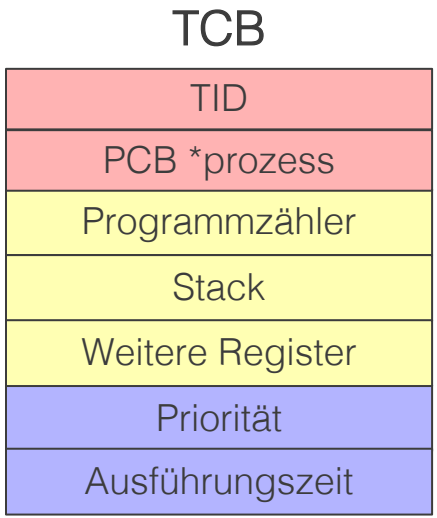
- Im Linux-Kernel ist der PCB in `struct task_struct`

6.4 Threads

- Ziel: effiziente nebenläufige/parallele Verarbeitung
  - Dies geht auch mit Prozessen (fork & pipes), aber das ist langsam und insbesondere das Umschalten zwischen Adressräumen ist teuer.
  - Beim Wechsel des Adressraums werden alle Caches gespült, sowie der TLB (siehe später, virtuelle Speicherverwaltung)
- Lösung: mehrere Threads pro Prozess respektive nebenläufige/parallele Verarbeitung innerhalb eines Adressraums
- Anwendungsbeispiele
  - parallele Berechnungen (simultan auf mehreren Cores)
  - blockierendes Warten auf ein Ereignis (I/O von Gerät)
  - gleichzeitige Bearbeitung von parallelen Einzelaufgaben (http-Requests, ...)

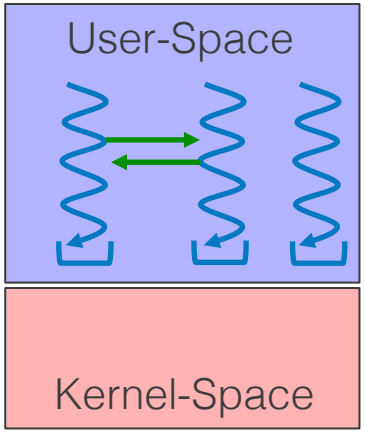
6.4 Threads

- Verwaltung im Kernel-Space durch Thread-Kontrollblock (=TCB):
  - Thread-Identifikation (ID)
  - Zuordnung zum Prozess
  - Threadzustand (Register)
    - Eigener Befehlszähler
    - Eigener Stack
    - ...
  - Verwaltungsinformationen für Scheduler
    - Priorität
    - Ausführungszeit (siehe Scheduler)



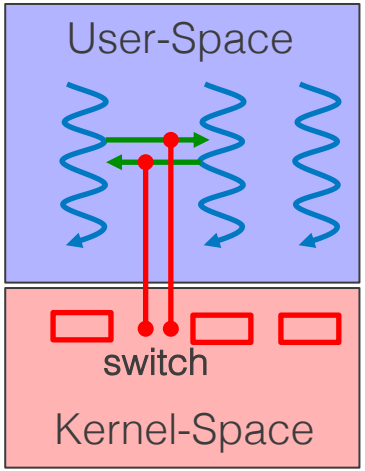
6.4.1 User-Level Threads

- Programm ruft API-Funktionen zum Umschalten zwischen Threads
  → User-Level-Threads sind dem BS-Kern nicht bekannt
- Vorteile:
  - Scheduling in der „Hand“ des Programmierers
  - Umschalten sehr schnell - ohne Umweg über BS
  - Funktioniert auch auf BS, welche keine Threads kennen
- Nachteile:
  - Bei einem blockierenden Systemaufruf werden alle Threads eines Prozesses blockiert
  - Fairnessproblem bei unterschiedlichen Thread-Anzahl in den einzelnen Prozessen
- Beispiele: Microsoft Windows Fibers, GNU Portable Threads, JVM virtual threads (Projekt Loom)



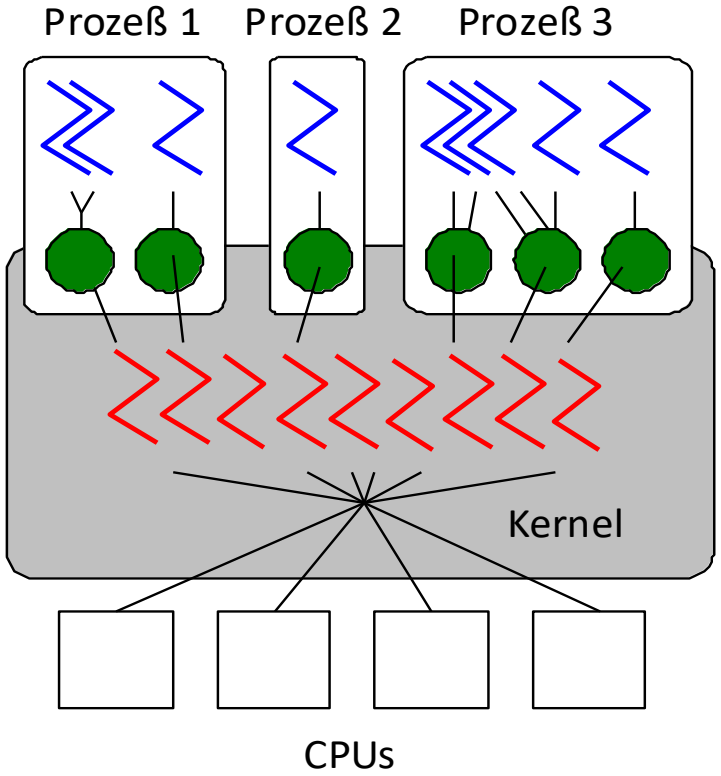
6.4.2 Kernel-Level Threads

- KL-Threads sind dem Kern bekannt und unterliegen dessen Kontrolle:
  - Umschaltung zwischen den Threads eines Prozesses geschieht durch den BS-Scheduler
  - Umschaltung durch BS auch prozessübergreifend
- Vorteile:
  - Threads eines Prozesses können sich nicht blockieren
  - transparent für den Programmierer
  - Scheduling für das Gesamtsystem
- Nachteile:
  - Umschaltung etwa aufwändiger als bei UL-Threads
- Beispiele: Microsoft Windows, Linux, MacOS



6.4.3 Hybride Threads

- Kombination von UL- und KL-Threads
- Lightweight-Prozesse (LWP):
  - Verbindet 1 oder mehrere UL-Threads mit einem KL-Thread
  - Laufen im User-Mode
- Beispiele: Solaris, UNIX System V



6.4.4 Threads in Linux

- KL-Threads gemäß POSIX-Standard.
- Implementiert in C-Bibliothek `pthread`
- Über 100 Funktionen, u.a. auch Synchronisierung (siehe später).

```c
#include <stdio.h>
#include <pthread.h>

/* second thread */
void *my_thread(void *param) {
   ...
}

int main () {
   pthread_t thread_ref;
   char ch ='*';

   /* create another thread */
   /* (reference, attributes (priority, stacksize, ...), function, params) */
   if ( pthread_create(&thread_ref, NULL, &my_thread, &ch) ) {
      ...
```

Pthread-Bibliothek (Auszug)

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Beispiel zu Attributen, siehe
`man pthread_attr_init`

- Achtung: wenn der Haupt-Thread terminiert, so terminiert der Prozess und damit alle anderen Threads! (Im Gegensatz zu Java)

Beispiel (Auszug)

- Haupt-Thread wartet bis anderer Thread terminiert

```
#include <stdio.h>
#include <pthread.h>

int main () {
   pthread_t thread_ref;
   char ch ='*';

   /* create another thread */
   if ( pthread_create(&thread_ref, NULL, &my_thread, &ch) ) {
      ...

   /* wait for the second thread to finish */
   if (pthread_join(thread_ref, NULL)) {
      fprintf(stderr, "Error joining thread\n");
      return 2;
   }
```

6.4.4 Threads in Linux

- Für die Thread-Erzeugung verwendet Linux den System-Aufruf **`clone`**:

```
int clone (int (*fn)(void *), void *stack, int flags, void *arg)
```

  - Flags (Auszug):
    - CLONE_VM Adressraum gemeinsam nutzen
    - CLONE_FILES Dateideskriptoren (offene Dateien) teilen
    - CLONE_SIGHAND Gemeinsame Signalbehandlungstabelle
    - ...

- Für Linux sind alle Threads und Prozesse intern „Tasks“
  → Scheduler macht hier keinen Unterschied

6.4.4 Threads in Linux

- Beispiel mit direktem System-Aufruf

```
#define _GNU_SOURCE /* needed for clone, must be before sched.h */
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>

int thread_proc(void *arg) {
   for(;;) putchar('*');
}

int main() {
   char *stack = (char*)malloc(4096);

   clone(&thread_proc, stack+4096, CLONE_VM, 0);

   for(;;) putchar('.');

   free(stack);
   return 0;
}
```

Thread-Zustände in Linux

- RUNNING: laufend oder wartet auf CPU

- SLEEPING: freiwilliges oder erzwungenes Schlafen, weil Ressourcen fehlen
  - INTERRUPTIBLE: blockiert wegen I/O oder aktiv schlafend, aber unterbrechbar
  - UNINTERRUPTIBLE: wie INTERRUPTIBLE, reagiert aber nicht auf Signale. Selten verwendet, z.B. wenn Treiber auf I/O wartet und nicht unterbrochen werden darf.
  - TRACED: angehalten durch Job-Control oder Debugger

- ZOMBIE: Terminiert, aber Exit-Code wurde noch nicht abgeholt.
- DEAD: Terminiert, Exit-Code abgeholt, wird gelöscht.

Thread-Zustände in Linux (2)

- Kürzel beim `ps` Befehl gemäß man-Page:
  - D uninterruptible sleep (usually IO)
  - R running or runnable (on run queue)
  - S interruptible sleep (waiting for an event to complete)
  - T stopped by job control signal
  - t stopped by debugger during the tracing
  - W paging (not valid since the 2.6.xx kernel)
  - X dead (should never be seen)
  - Z defunct ("zombie") process, terminated but not reaped by its parent

Thread-Zustände in Linux (3)

- Beispiel: CTRL-Z → SIGSTOP-Signal an Vordergrund-Prozesse

```
$ sleep 100
^Z      # Pressed CTRL+z
[1]+  Stopped
$ ps -o pid,state,command
  PID S COMMAND
13224 T sleep 100
...
```

- Sleep-Prozess ist im Zustand TRACED (T). Kann durch **`bg`** oder **`fg`** wieder „angestossen“ werden. → SIGCONT-Signal

```
$ bg
[1]+ sleep 100 &
$ ps -o pid,state,command
  PID S COMMAND
13224 S sleep 100
...
```

6.5 Thread-Umschaltung

- Situation: viele Threads warten auf die CPU/Cores
- Ziel: nebenläufige Ausführung aller rechenbereiter Threads
- Lösung: regelmäßig Umschalten zwischen den laufenden Threads

- Gründe für einen Thread-Wechsel:
  - Thread kann nicht weiterrechnen, weil er auf einen I/O wartet
  - Rechenzeit ist abgelaufen,
  - Thread terminiert.

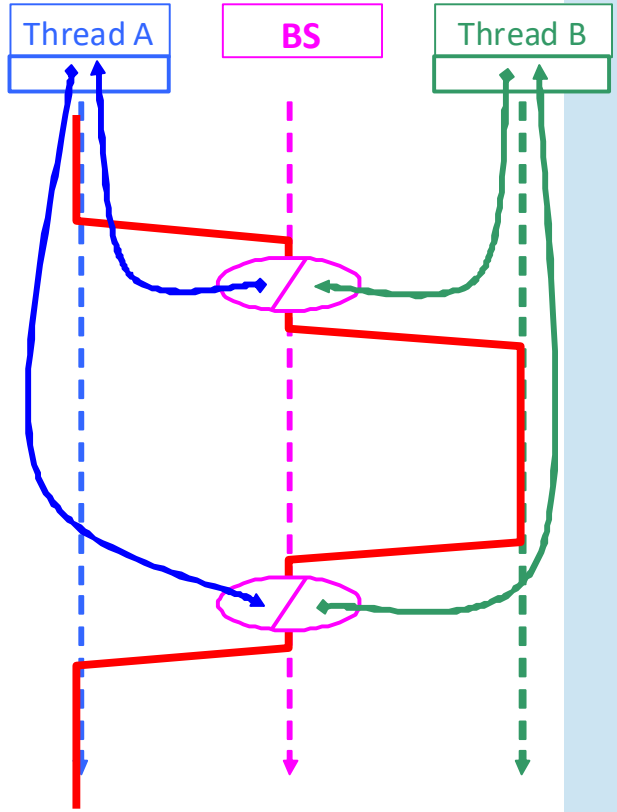
6.5.1 Kooperatives vs. preemptives Umschalten

- Kooperativ:
  - Alle Threads rufen zuverlässig in bestimmten Abständen eine Umschaltoperation der Thread-Implementierung auf → umständlich für den Programmierer
  - Nicht kooperierende Threads können das System lahm legen
  - Wichtig für User-Level Thread-Implementierungen

- Preemptiv:
  - Umschalten unter Kontroller des Betriebssystems
  - Die CPU kann jederzeit entzogen werden (nach jedem Assemblerbefehl)

6.5.2 Ablauf der Umschaltung

- Implementiert der Dispatcher im BS-Kern
- Schritte:
  - CPU entziehen
  - Zustand des unterbrochenen Threads sichern
  - Scheduler wählt nächsten Thread aus
  - Zustand des nächsten Threads wiederherstellen und diesen „anstoßen“. Falls der nächste Thread zu einem Prozess gehört, so muss der Adressraum umgeschaltet werden.
- Bemerkungen:
  - Zustände in Thread- und Process-Control-Block
  - TCB für Register; PCB für Adressraum



6.5.3 Entzug der CPU

- Problem: Wie kann das BS einem Thread die CPU entziehen?
- Lösung: mithilfe der Zeitgeber-Unterbrechung (engl. timer interrupt)
  - Zeitgeber löst periodisch einen Interrupt aus, z.B. jede jede Millisekunde
  - Das Zeitintervall ist programmierbar
  - Jeder Interrupt ist ein „tick“

- Einschub: Interrupts
  - Geräte melden sich bei der CPU mithilfe eines Interrupts
  - Die CPU stoppt dadurch sofort die aktuelle Befehlsausführung und springt eine Funktion an, die in einer Interrupt-Tabelle registriert ist.
  - Für jede Interrupt-Nummer gibt es eine entsprechende Funktion im BS-Kern
  - Mehr später im Kapitel „Ein-/Ausgabe“

6.5.4 Reentranter Code

- Wichtig: preemptiver CPU-Entzug kann jederzeit erfolgen
- Reentranter Code: kann von mehreren Threads gleichzeitig ausgeführt werden und arbeitet korrekt egal wo er unterbrochen wird
- Muss ggf. synchronisiert werden und sollte globale Variablen vermeiden
  → Zustände eines Threads als lokale Variablen auf dem Stack
  → jeder Thread hat seinen eigenen Stack

6.5.4 Reentranter Code

- Gegenbeispiel →
  - Stiftfarbe in globaler Variable
  - Zeichnen von Linien mit gesetzter Farbe

Thread 1

```
setPenColour(BLUE);
drawLine(0,   100,
         100, 100);
```

Thread 2

```
setPenColour(RED);
drawLine(0,   200,
         100, 200);
```

```
int pen_colour;

void setPenColour(int col) {
   pen_colour = col;
}

void drawLine(int x, int y) {
   ...
}
```

Erwünschtes Ergebnis

ebenso möglich

auch möglich

6.6 Scheduling

- Scheduler: entscheidet welcher Thread als nächstes die CPU erhält
  - i.d.R. warten viele Threads in einer Warteschlange auf die CPU
- Scheduler tritt in Aktion wenn ein Thread:
  - startet & terminiert
  - freiwillig die CPU freigibt
  - auf eine E/A-Operation wartet
  - seine Rechenzeit voll ausgenutzt hat
- Anzahl Threads und deren Verhalten ändert sich dynamisch:
  - rechenintensive Phasen mit ununterbrochener CPU Nutzung und seltenen E/As
  - E/A-lastige Zeitabschnitte mit nur kurzen CPU-Nutzungszeiten und häufigen E/As

6.6.1 Ziele

- **Effizienz**: optimale CPU-Ausnutzung (bei Bedarf bis 100%)
- **Wartezeit**: in der Bereit-Liste (engl. ready queue) minimieren
- **Fairness**: bezüglich der CPU-Verteilung
- **Durchsatz**: Viele Threads sollen in einem Zeitintervall die CPU bekommen
- **Ausführungszeit**: Jeder Thread sollte schnell seine Arbeit beenden können.
- **Antwortzeit**: interaktive Eingaben sollten schnell verarbeitet werden
- ...

- Problem: teilweise konkurrierende Ziele und das Umschalten kostet Zeit

6.6.2 Strategie: First Come First Served (FCFS)

- Bearbeitung der Threads in Reihenfolge ihrer Ankunft in der Bereit-Liste.
- Prozessorbesitz bis zum Ende oder freiwilligen Abgabe.
  - Kooperatives Multitasking

- Vorteil: wenig Thread-Umschaltungen
- Nachteil: Konvoi-Effekt
  - Kurze Threads stauen sich hinter lang laufenden Threads
  - Dadurch werden I/O-lastige Threads benachteiligt und rechenintensive Threads bevorzugt

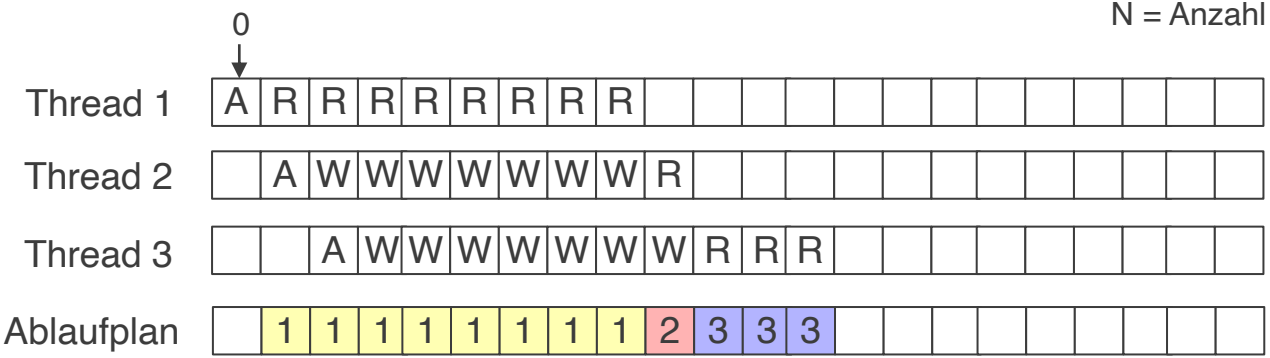
6.6.2 Strategie: First Come First Served (FCFS)

- Beispiel:

Thread	Ankunft (A)	Zeitbedarf (R=rechnen, B=blockiert)
T1	0	8R
T2	1	1R
T3	2	3R

$$W_D = \frac{\sum_{i=1}^{N} W_i}{N}$$

W_D : durchschnittl. Wartezeit
W_i = Wartezeit von Thread i
N = Anzahl Threads



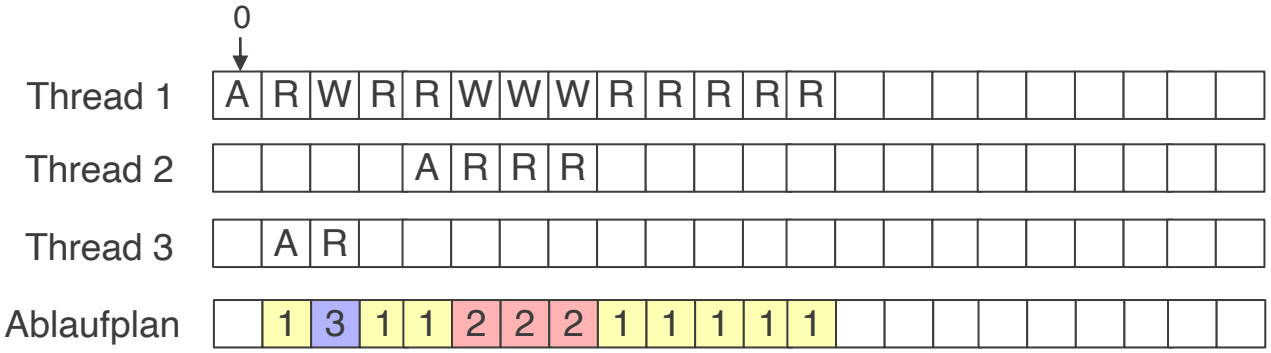
1 Kästchen = 1 Zeiteinheit, Zeit für das Umschalten ausgelassen

$W_D = 14/3 = 4{,}67$

6.6.3 Strategie: Shortest Job First (SJF)

- Thread mit kürzester Rechenzeit wird als nächster bearbeitet.
  - Betrachtet wird die Restlaufzeit.
  - Verwendet preemptives Multitasking
- Beispiel:

Thread	Ankunft (A)	Zeitbedarf (R=rechnen, B=blockiert)
T1	0	8R
T2	4	3R
T3	1	1R



1 Kästchen = 1 Zeiteinheit, Zeit für das Umschalten ausgelassen

$W_D = 4/3 = 1{,}33$

6.6.3 Strategie: Shortest Job First (SJF)

- Vorteil: vermeidet Konvoi-Effekt von FCFS

- Nachteile:
  - Lang laufende Threads verhungern, falls immer kürzere Threads ankommen
  - Laufzeiten müssen bekannt sein

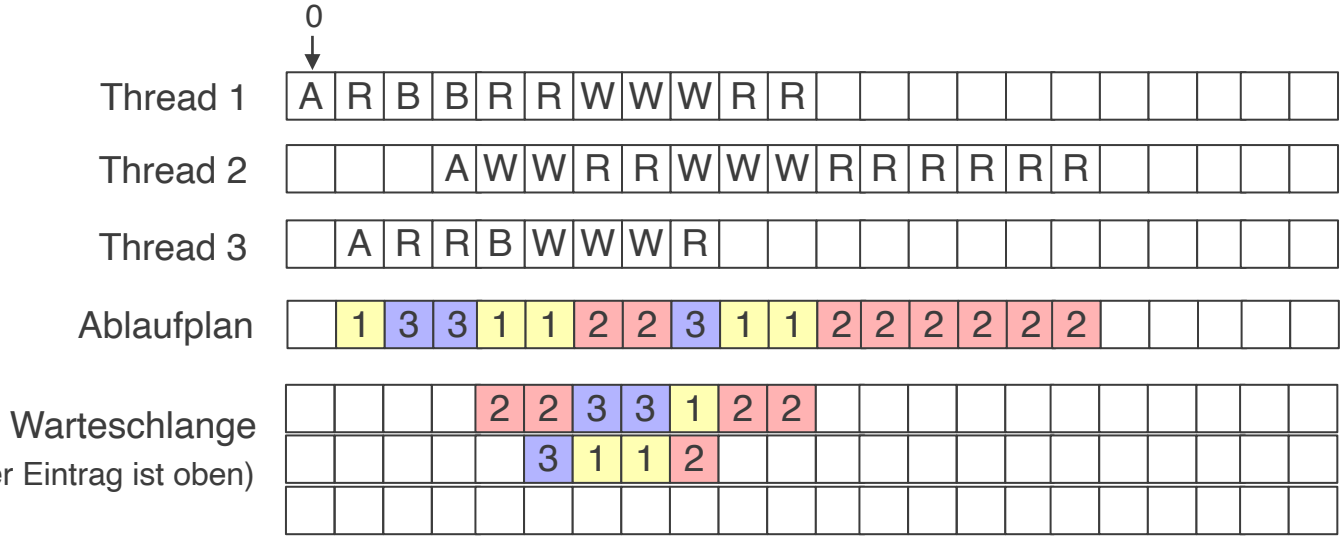
6.6.4 Strategie: Round-Robin (RR)

- Ziel: gleichmäßige Verteilung der CPU.
- Weit verbreitete Strategie (z.B. UNIX und NT):
  - Threads in Ankunftsreihenfolge verarbeiten
  - Nach Ablauf einer festgesetzten Frist, z.B. 10-100ms,
    wird die CPU entzogen (engl. preemption) → auch Zeitscheibe genannt
  - Nach CPU-Entzug wird der Threads am Ende der Bereit-Liste eingereiht,
    außer er ist blockiert.
- Wahl der Zeitscheibe:
  - Vernünftiges Verhältnis der Zeitscheibe und die notwendige Zeit für einen
    Kontextwechselzeit muss beachtet werden.
  - Große Zeitscheiben sparen Kontextwechsel,
    verursachen aber lange Antwortzeiten

6.6.4 Strategie: Round-Robin

Thread	Ankunft (A)	Zeitbedarf (R=rechnen, B=blockiert)
T1	0	1R, 2B, 4R
T2	3	8R
T3	1	2R, 1B, 1R

- Beispiel: Zeitscheibe = 2 ZE

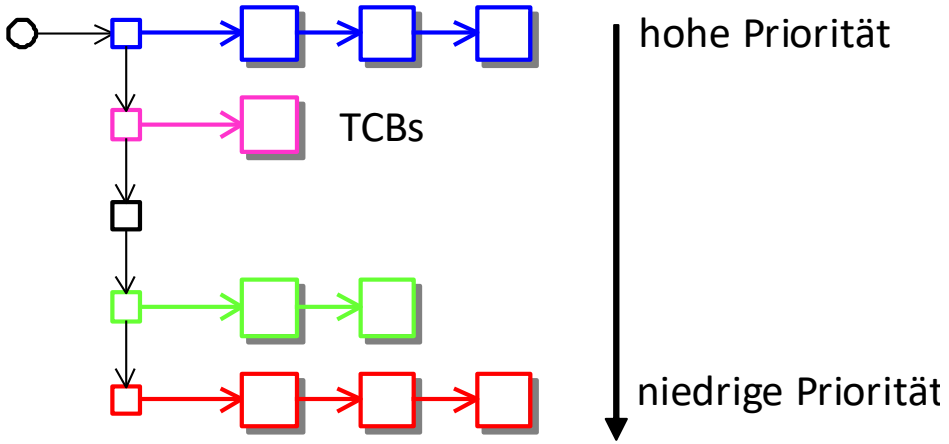


$W_D = 11/3 = 3{,}67$

1 Kästchen = 1 Zeiteinheit (ZE)

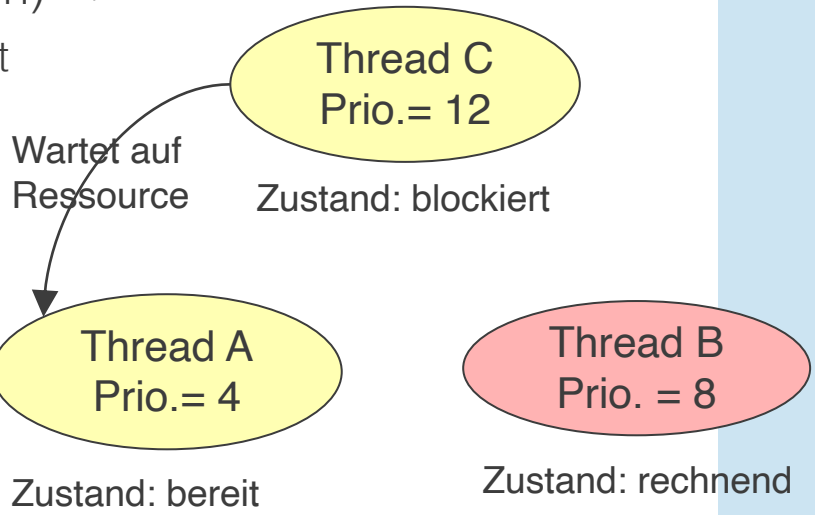
6.6.5 Multilevel Scheduling

- Es ist wünschenswert Threads nach Wichtigkeit zu unterscheiden.
  → Jeder Thread erhält eine **Prioritätsnummer**.
- Scheduler selektiert immer zuerst Thread mit der höchsten Priorität
- Implementierung:
  - Eine Bereit-Liste pro Priorität
  - So kann der richtige Thread schnell gefunden werden



Probleme

- 1. Verhungern (engl. starvation)
  - Thread mit niedriger Priorität bekommt die CPU nicht zugeteilt, falls immer Threads mit höherer Priorität rechenbereit sind.
- 2. Prioritätsinvertierung (engl. priority inversion) →
  - Thread C „verhungert“, trotz höchster Priorität
  - Thread B dominiert die CPU
  - Dadurch kommt Thread A nicht zum Zuge und kann daher die Ressource nicht freigeben.
  - Thread C auf unbestimmte Zeit blockiert



Altern (engl. aging)

- Vermeidung von Verhungern und Prioritätsinversion
- Scheduler passt die Priorität dynamisch an → diese steigt mit der Wartezeit
- Für Beispiel zuvor
  - Priorität von Thread A steigt schrittweise an.
  - Sobald sie 8 ist, bekommt A die CPU und kann die Ressource freigeben
  - Dadurch wird C deblockiert und bekommt dann als nächstes die CPU

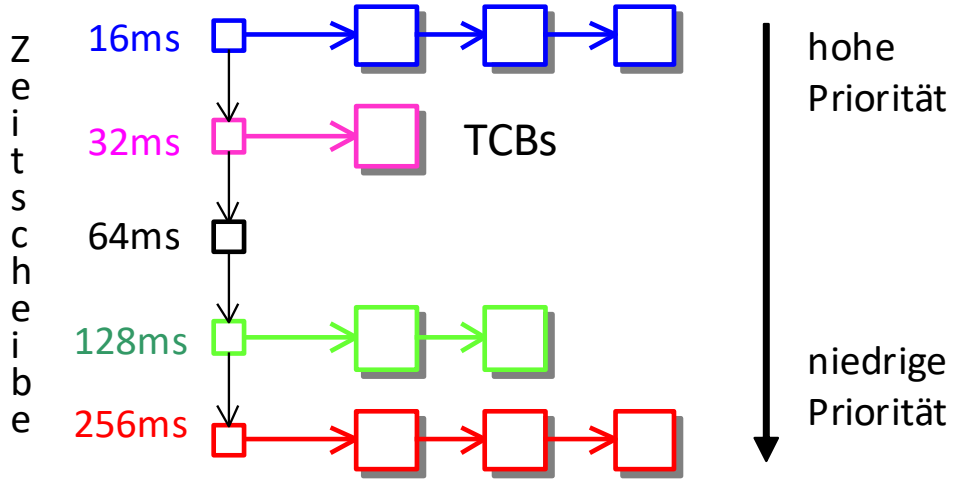
- Bem.: Mars Pathfinder hatte Prioritätsinversion
  - siehe D. Wilner, „Vx-Files: What really happened on Mars?”, Keynote at the 18th IEEE Real-Time Systems Symposium, Dec. 1997

6.6.6 Multilevel-Feedback-Scheduling

- **Feedback** → dynamische Anpassung der Scheduling-Kriterien abhängig vom Verhalten eines Threads (Rechenintensität bzw. Wartezeit)
- Angepasst werden kann die Priorität, Einsortierung in Queue (am Anfang oder am Ende), sowie die Zeitscheibenlänge
- Ziele:
  - Verhungern verhindern, Prioritätsinversion auflösen
  - Balance zw. E/A- u. rechenintensiven Threads
- Zwei grundlegende Strategien:
  - wartende Threads hochstufen
  - rechenintensive Threads herabstufen

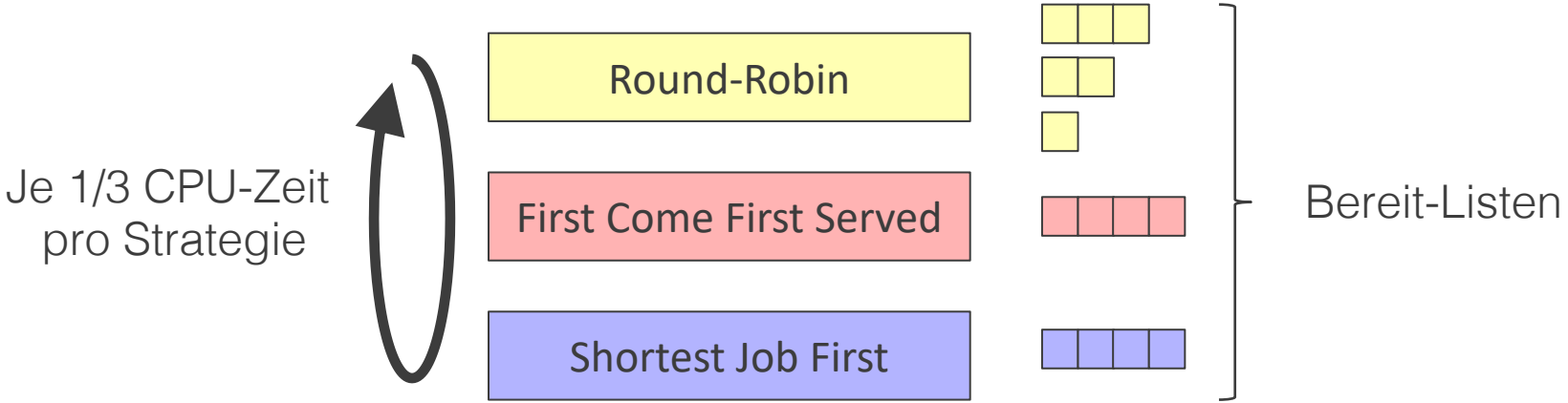
Multilevel-Feedback Scheduling Round-Robin

- Multilevel Scheduling ergänzt um Feedback-Adaptierung
- Threads, welche blockierende I/O-Funktionen aufrufen oder die CPU freiwillig abgeben bleiben in ihrer Bereit-Liste
  → I/O-lastige Threads bevorzugen
- Threads, welche verdrängt werden, in Bereit-Liste mit niedrigerer Priorität einordnen
  → rechenintensive Threads bestrafen



Kombination mehrerer Scheduling-Verfahren

- Gleichzeitig mehrere Scheduling-Strategien
  - Für jede Strategie eigene Bereit-Liste
  - Threads in passende Bereit-Liste einsortieren
  - Scheduling zwischen den Strategien: statische Priorität oder Zeitscheiben



Hörsaal-Aufgabe

- Schreiben Sie ein Programm, welches zwei Threads verwendet.
- Das Programm soll eine Zahl als Argument vom Terminal übergeben bekommen. (Zahl muss nicht unbedingt geprüft werden).
- Der zweite Thread soll die Quersumme dieser Zahl berechnen.
- Der Haupt-Thread soll das Ergebnis der Berechnung ausgeben (dazu muss er informiert werden, wann der 2. Thread fertig ist)
- Das Programm soll keine Pipes verwenden. Das Ergebnis von Thread 2 soll stattdessen als Rückgabewert zurückgegeben werden. Der Haupt-Thread kann den Rückgabewert über `pthread_join` entgegennehmen.

6.7 Echtzeit-Scheduling

- Echtzeitsysteme müssen definierte Zeitschranken einhalten, auch in Fehlersituationen.
- Threads erhalten Sollzeitpunkte (engl. deadlines).
  - Voraussetzung: Laufzeit vorab bekannt.
  - Unterscheidung bei Sollzeitpunkten zwischen:
    - harter Echtzeit: Verletzung bedeutet Ausfall des Systems (nicht tolerierbar) → für kritische Anwendungen (z.B. Kernkraftwerk).
    - weicher Echtzeit: Qualitätseinbußen bei Verletzung, aber in Grenzen tolerierbar; Einhaltung von SLA (Service Level Agreement)

6.7.1 Statisches Scheduling

- Ziel: einhalten harter Echtzeitanforderung
- Scheduling vor der eigentlichen Ausführung (bei der Compilation)
- Vorberechnung eines vollständigen Ausführungsplans in Tabellenform.
- Damit ist vorab bekannt wann und wo umgeschaltet wird.

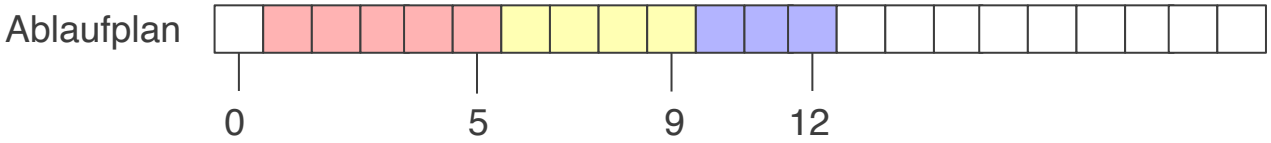
- Bemerkungen:
  - Umschalten ist sehr effizient
    → einfacher Tabellenzugriff während Ausführung genügt.
  - Keine Desktop-Betriebssysteme, sondern eingebettete Systeme (z.B. OSEK).

6.7.2 Earliest Deadline First (EDF)

- Verbreitete dynamisch Echtzeit-Strategie.
  - Threads mit Ausführungsfristen
  - Thread mit engster Frist (engl. deadline) wird bevorzugt
- Beispiel:
  - non-preemptive

Thread	Ankunft	Laufzeit	Frist
T1	0	4	10
T2	0	5	7
T3	0	3	17

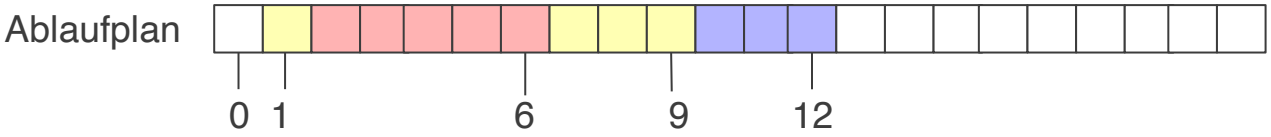
Frist ab Zeitpunkt 0



6.7.2 Earliest Deadline First (EDF)

- Weiteres Beispiel:
  - Preemptive
  - Verschiedene Ankunftszeiten

Thread	Ankunft	Laufzeit	Frist
T1	0	4	10
T2	1	5	7
T3	2	3	17



- Dynamisches Einplanen, wenn ein neuer Thread hinzukommt.
- (CPU Auslastungsgrenze bis zu 100% möglich).

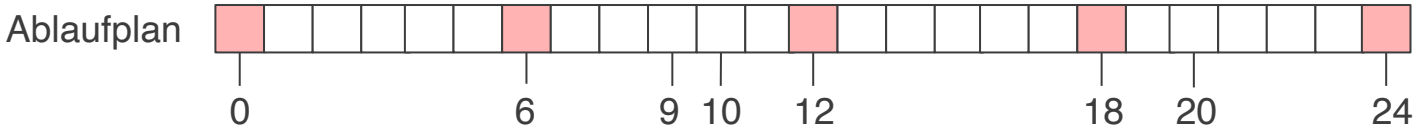
6.7.3 Rate Monotic Scheduling (RMS)

- Gut geeignet für periodischer Aufgaben
- Priorität = Frequenz
  - Hohe Priorität für Aktivitäten mit hoher Frequenz,
    niedrige Priorität für niederfrequente Aufgabe
  - Ende der Periode entspricht der Frist (engl. deadline)
  - z.B. für Multimedia-Ströme mit unterschiedlichen Frame-Raten
- Voraussetzung: Threads sind unabhängig voneinander

6.7.3 Rate Monotic Scheduling (RMS)

- Beispiel mit Preemption
  - Je höher die Frequenz / kleiner die Periode desto höher ist die Priorität des Threads
  - Vorgehensweise:
    - Trage in den Ablaufplan zuerst den Thread mit der höchsten Priorität ein.
    - Danach alle anderen in absteigender Reihenfolge ihrer Prioritäten

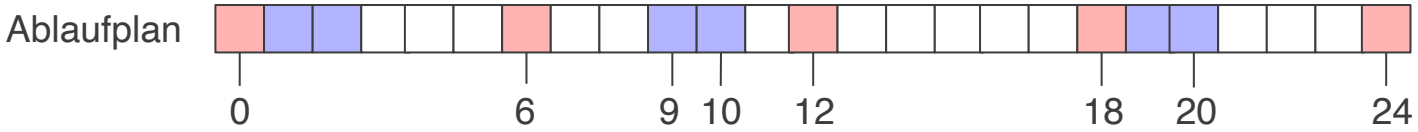
Thread	Ankunft	Laufzeit	Periode
T1	-1	3	10
T2	-1	1	6
T3	-1	2	9
T4	-1	2	12



6.7.3 Rate Monotic Scheduling (RMS)

- Beispiel mit Preemption
  - Je höher die Frequenz / kleiner die Periode desto höher ist die Priorität des Threads
  - Vorgehensweise:
    - Trage in den Ablaufplan zuerst den Thread mit der höchsten Priorität ein.
    - Danach alle anderen in absteigender Reihenfolge ihrer Prioritäten

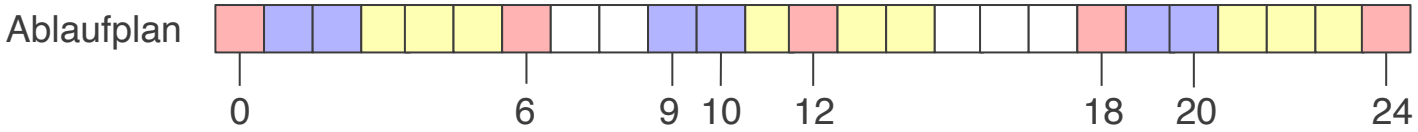
Thread	Ankunft	Laufzeit	Periode
T1	-1	3	10
T2	-1	1	6
T3	-1	2	9
T4	-1	2	12



6.7.3 Rate Monotic Scheduling (RMS)

- Beispiel mit Preemption
  - Je höher die Frequenz / kleiner die Periode desto höher ist die Priorität des Threads
  - Vorgehensweise:
    - Trage in den Ablaufplan zuerst den Thread mit der höchsten Priorität ein.
    - Danach alle anderen in absteigender Reihenfolge ihrer Prioritäten

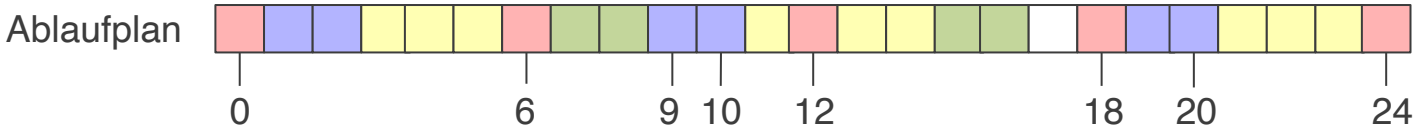
Thread	Ankunft	Laufzeit	Periode
T1	-1	3	10
T2	-1	1	6
T3	-1	2	9
T4	-1	2	12



6.7.3 Rate Monotic Scheduling (RMS)

- Beispiel mit Preemption
  - Je höher die Frequenz / kleiner die Periode desto höher ist die Priorität des Threads
  - Vorgehensweise:
    - Trage in den Ablaufplan zuerst den Thread mit der höchsten Priorität ein.
    - Danach alle anderen in absteigender Reihenfolge ihrer Prioritäten

Thread	Ankunft	Laufzeit	Periode
T1	-1	3	10
T2	-1	1	6
T3	-1	2	9
T4	-1	2	12



6.7.3 Rate Monotic Scheduling (RMS)

- Bemerkungen:
  - Minimale Verzögerung für häufig wiederholte Tasks
  - Aber Zerstückelung niederfrequenter Tasks
  - Situationen, die durch Verzögerung eines höher priorisierten Threads, zu einem gültigen Ablauf führen, sind nicht lösbar
  - (CPU Auslastungsgrenze nur bis ca. 69 - 85% möglich)

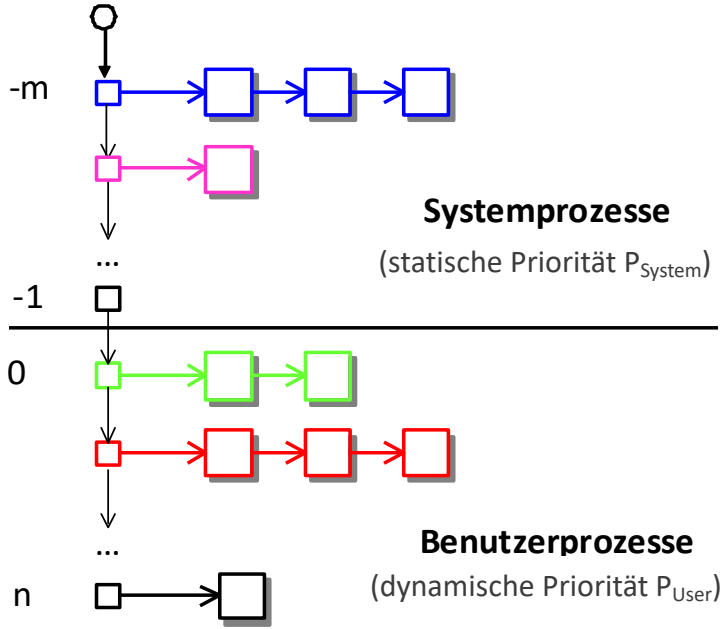
6.8 Leerlauf (engl. idle)

- Falls alle Threads warten ist CPU frei u. es läuft ein Leerlauf-Thread:
  - darf nicht anhalten, hat geringste Priorität, muss jederzeit verdrängbar sein.
  - Busy-Looping mit speziellen Instruktionen vermeidbar
    → Beispiel: „HLT“ (x86): stoppt CPU; (Timer-)Interrupt weckt CPU wieder auf.

- Aber Leerlauf auch nutzbar für:
  - Prüfungen (Speichermodule)
  - Heap Kompaktifizierung
  - Garbage Collection
  - Software-Updates …

6.9 Fallbeispiel: Scheduling in UNIX System V R3

- Scheduler kennt nur Prozesse, keine Threads.
- Preemtpives Feedback-Scheduling:
  - pro Priorität eine Queue mit Round Robin
  - Scheduler passt Prioritäten von Benutzerprozessen dynamisch an (beginnend ab n/2)
  - Benutzer beeinflusst Priorität mit `setpriority` bzw. `nice`, aber es gilt immer $P_{User} > P_{System}$
    - `nice` beim Prozess-Start
    - `renice` für laufende Prozesse
    - (negative Werte kann nur root setzen)



6.9 Fallbeispiel: Scheduling in UNIX System V R3

- Zeitscheibe z.B. 1000ms; Uhrtick = 10ms (alle 10ms ein Interrupt vom Zeitgeber)
- Dynamische Anpassung von Prioritäten:
  - Alle 40ms wird `dyn_prio` des laufenden Prozesses neu berechnet, indem seine `cpu_usage` erhöht wird
  - Jede Sekunde wird für alle Prozesse in der Bereit-Liste die Priorität berechnet
    - `dyn_prio := base_prio + cpu_usage / 2 + nice` (CPU-Nutzung in „Uhr-Ticks")
    - Prozessornutzung wird schrittweise „vergessen": `cpu_usage := cpu_usage / 2`
- Wirkung: starke CPU-Nutzung führt zu schlechter Priorität
  - rechenintensive Proz. werden benachteiligt und I/O-intensive Proz. werden bevorzugt → I/O-lastige Proz. belegen die CPU nur kurz, um einen E/A-Auftrag abzusetzen
  - Man erreicht dadurch eine Balance zw. CPU- und I/O-lastigen Aufgaben

6.9 Fallbeispiel: Scheduling in UNIX System V R3

- Blockiert Prozess, weil er auf ein Ereignis wartet:
  - so erhält er abhängig vom Ereignis eine höhere Kernel-Priorität
  - Beim Übergang vom Kernel- in den User-Mode erhält er wieder seine alte User-Priorität

- Bemerkungen:
  - Prozess der im Kern-Modus läuft kann erst beim Rücksprung in den User-Modus verdrängt werden (Kern ist nicht reentrant)

6.10 Fallbeispiel: BSD UNIX 4.3

- Dynamische Prioritäten mit 32 Queues (Einordnung Priorität/4):
  - System-Prozesse: 0-49, Benutzer-Prozesse: 50-127
  - je kleiner Prioritätswert, desto höher ist die Priorität
  - jeweils mit einer Zeitscheibe von 100ms
- Update der dynamischen Priorität p_usr: alle 4 Ticks (40 ms):
  - **`p_usr = P`$_{USER}$ `+ [p_cpu / 4] + 2*p_nice`**

  - `p_cpu` = geschätzte CPU-Nutzung eines Prozesses
    (wird für den laufenden Prozess alle 10ms inkrementiert)
  - `p_nice` = Gewichtungsfaktor, vom Benutzer definiert (-20 bis +20)
  - P_{USER} = statische Priorität des Prozesses

6.10 Fallbeispiel: BSD UNIX 4.3

- Glättung des Wertes der Prozessornutzung `p_cpu` jede Sekunde:
  - **`p_cpu = (2*load) / (2*load+1) * p_cpu + p_nice`**
  - `load` = Abschätzung der CPU-Auslastung
    → mittlere Anzahl lauffähiger Prozesse in der Bereit-Queue in der letzten Minute
  - Bem.: bei hoher Last wird `p_cpu` langsamer verringert

- Nachteil: Prioritätsanpassung erfolgt nur für rechenbereite Prozesse.

6.10 Fallbeispiel: BSD UNIX 4.3

- Deshalb zusätzlich **dynamische Prioritäten für wartende Prozesse:**
  - Neuberechnung der Priorität wartender Prozesse beim Deblockieren
  - Basis: `p_slptime` = Wartezeit des Prozesses in Sekunden
  - Beim Aufwecken des Prozesses wird `p_cpu` berechnet:

    $$p\_cpu = \left(\frac{(2*load)}{(2*load+1)} \right)^{p\_slptime} * p\_cpu$$

  - Auch hier wird aktuelle Last berücksichtigt:
    - Bei hoher Last erhält ein wartender Prozess weniger Bonus für seine Wartezeit
    - `p_cpu` wird dann weniger stark geglättet

6.11 Fallbeispiel: Linux

- `SCHED_FIFO`:
  - für wichtige „Echtzeit“-Threads, Queue nur für **`root`** zugänglich
  - statische Prioritäten von 1 bis 99 (FCFS innerhalb einer Prioritätsstufe)
  - Verdrängung nur durch Threads mit höherer statischer Priorität
- `SCHED_RR`:
  - wie **`SCHED_FIFO`**, aber Round Robin für jede Prioritätsstufe
- `SCHED_OTHER`:
  - für normale Threads
  - statische Priorität: -20 bis + 20 (zu Beginn 0)
  - Gewichtung mit **`nice`** und **`setpriority`** (-19 bis +20)
    - je größer der Wert des geringer die Priorität)
  - zusätzlich dynamische Anpassung von Prioritäten

SCHED_OTHER: O(n)-Scheduler, Kernel 2.4

- Prozessorzeit ist in Epochen unterteilt
  - Epochen-Beginn: alle lauffähigen Threads haben ihr Zeitquantum erhalten
  - Epochen-Ende: alle lauffähigen Prozesse haben ihr Zeitquantum verbraucht
- Zeitquanten variieren mit den Threads und Epochen
  - In einer Epoche hat jeder Thread ein bestimmtes Zeitquantum für die CPU-Nutzung zur Verfügung (wird zu Beginn der Epoche berechnet)
  - Ein Thread kann CPU pro Epoche mehrfach erhalten, sofern er sein Zeitquantum noch nicht verbraucht hat.
- Jeder Thread besitzt eine Zeitquantum-Basis (beeinflussbar mit `nice`)
  - ZQ_{Basis} = 20 – `nice`-Wert (20 Ticks ~200ms)
  - Zeitquantum entspricht der Priorität (siehe später)
  - Das Zeitquantum eines Threads nimmt periodisch ab.

SCHED_OTHER: O(n)-Scheduler, Kernel 2.4

- Berechnung des Zeitquantums (ZQ):
  - Wenn Thread in Bereit-Liste eingefügt wird und eine neue Epoche begonnen hat
  - Falls Zeitquantum (ZQ) verbraucht ist: ZQ_{Basis} zuteilen, ansonsten: $ZQ_{Basis} + ZQ_{alt}/2$
    - I/O-lastige Threads verbrauchen ihr Zeit-Quantum nicht ganz und erhalten dafür den Bonus $ZQ_{alt}/2$

- Variablen im Threadkontrollblock (`struct task_struct`):
  - `rt_priority`: statische Priorität; für Echtzeit-Threads („rt_“ = realtime)
  - `priority`: Basispriorität für normale Threads, entspricht ZQ_{Basis}
  - `counter`: Anzahl verbleibender CPU-Ticks im Zeitquantum in der aktuellen Epoche
    - Entspricht ZQ (siehe oben)
    - Wird abhängig von gerechneten Ticks reduziert, bis auf 0

Prozessbewertung mithilfe der goodness-Funktion

- Thread mit der besten Bewertung (größter Rückgabewert) darf rechnen
- Rückgabewert der goodness-Funktion:
  - 0: Thread hat Zeit-Quantum voll ausgenutzt
  - 1-999: normaler Thread mit noch (teilweise) unverbrauchtem Zeit-Quantum
  - >=1000: Echtzeitprozess

```
if (t->policy != SCHED_OTHER)              // Echtzeit-Thread? (SCHED_FIFO/SCHED_RR)
    return 1000 + t->rt_priority;          // Echtzeitpriorität zurückgeben

if (t->counter == 0)                       // Quantum verbraucht?
    return 0;

if (t->mm == prev->mm)                     // gleicher Adressraum?
    return t->counter + t->priority + 1;   // dynamische Priorität + Bonus (+ 1)

int weight = t->counter + t->priority;     // dynamische Priorität
weight += 20 - p->nice;                    // nice Faktor berücksichtigen
return weight;
```

SCHED_OTHER: O(1)-Scheduler, Kernel 2.6

- Problem: Kern 2.4 verbringt zu viel Zeit in der `goodness`-Funktion
  - Er muss ständig für alle Threads in der Bereit-Liste deren „goodness“ berechnen
  - Zudem nur eine Bereit-Liste für alle Cores → Wettlaufsituation & Synchronisierung
- Lösung: Entwicklung eines O(1)-Schedulers für SCHED_OTHER
- Ziel: bessere Skalierbarkeit vor allem mit Hinblick auf Multicore CPUs

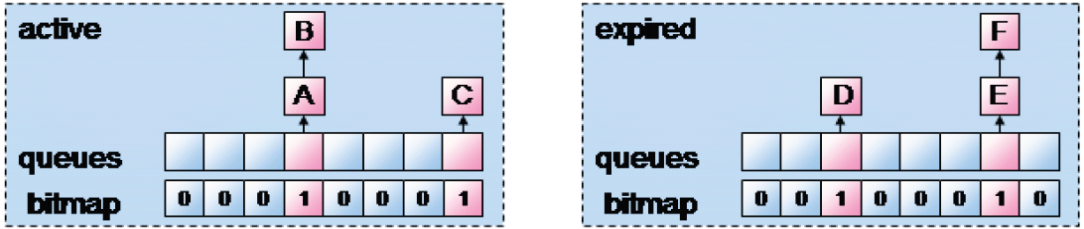
SCHED_OTHER: O(1)-Scheduler, Kernel 2.6

- 140 Prioritätslevel (kleiner Wert = hohe Priorität)
  - 0…99 für „Echtzeit“
  - 100…139 für „normale“ Threads (Default = 120)
    - Werden vom Scheduler unterteilt in „interactive“ und „batch“ Threads

- Statische Priorität wird beim Start zugeordnet (`nice`), wird dann dynamisch angepasst
  - CPU-intensive Prozesse werden bestraft
  - I/O-intensive Prozesse werden belohnt
  - Details siehe später

Prioritätsarray

- Pro Core zwei Prioritäts-Arrays:
  - `active`: Array für alle Threads mit positivem Quantum
  - `expired`: Array für alle Threads mit verbrauchtem Quantum
  - Wenn kein Thread mit positivem Quantum mehr vorhanden ist, tauschen die beiden Arrays ihre Rollen

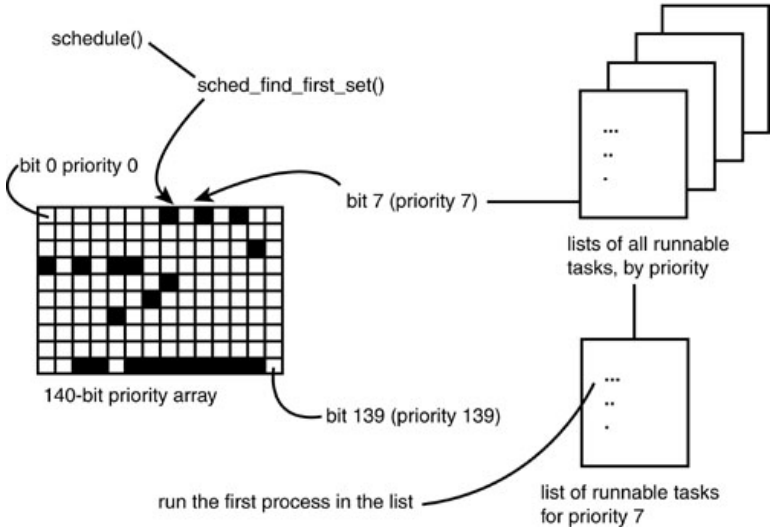


Auswahl des nächsten Threads

- Finden des nächsten Threads reduziert sich auf das Finden des ersten gesetzten Bits (höchstpriore Queue) im Array **`active`**
  - Dies ist unabhängig von der Anzahl der Threads

- **`sched_find_first_set()`** kann effizient implementiert werden
  - Assemblerbefehl `bsf` bei x86
    - BSF = Bit Scan Forward

- Innerhalb einer Priorität wird Round Robin genutzt



ptgmedia.pearsoncmg.com/images/chap3_0672325128/elementLinks/03fig02.jpg

Berechnung des Zeitquantums

- Erfolgt zunächst auf Basis der statischen Priorität in [ms]
- ZQ = (140 - static_prio) * 20, falls static_prio < 120
- ZQ = (140 - static_prio) * 5, falls static_prio >= 120
- Beispiel: ZQ = (140 -100) * 20 = 800 [ms] (static_prio=100)

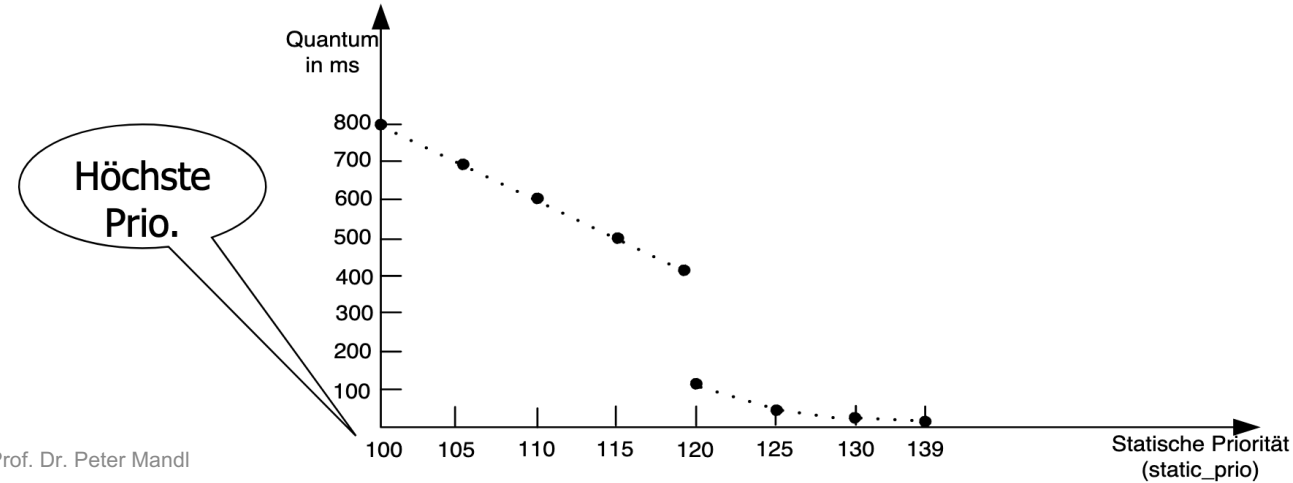
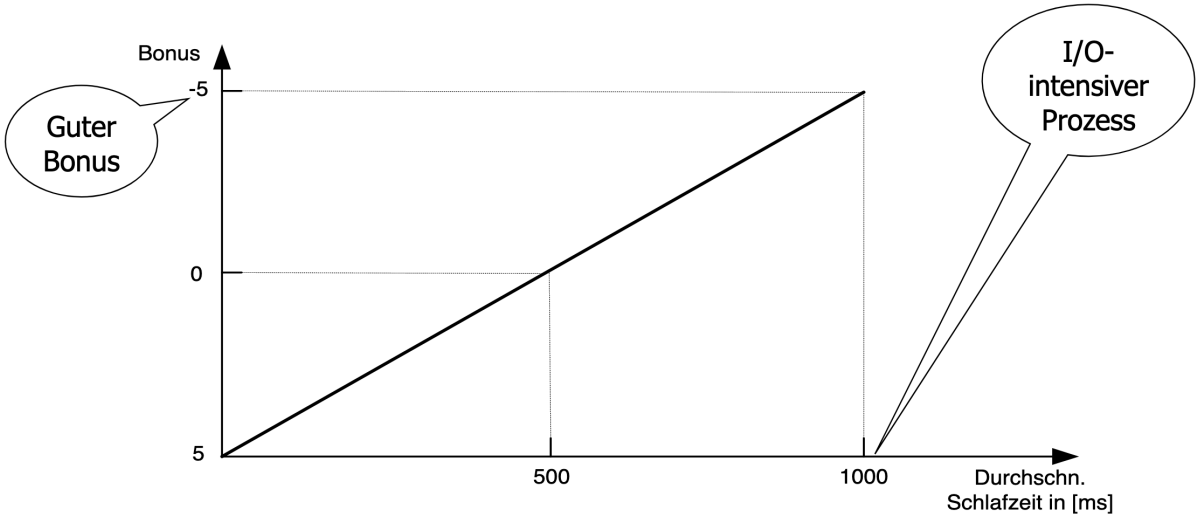


Bild von Prof. Dr. Peter Mandl

Dynamische Priorität

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



O(1)-Scheduler: Zusammenfassung

- Statische Prioritäten bestimmen die CPU-Zuteilung und die Quantumsgröße
- Die dynamische 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 dynamische Priorität
  - Also beeinflusst die statische Priorität das Quantum und die Rechenintensität die dynamische 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

SCHED_OTHER: Completely Fair Scheduler (ab 2.6.24)

- Default-Scheduler bis heute, ab Kernel 2.6.24
- Grundidee: jeder Thread sollte fairen Anteil an CPU bekommen
- Besonderheiten:
  - Keine Statistiken
  - Keine Bereit-Listen
  - Kein Verschieben von „active“ nach „expired“

- Verwaltung rechenbereiter Threads in einem Red-Black-Tree

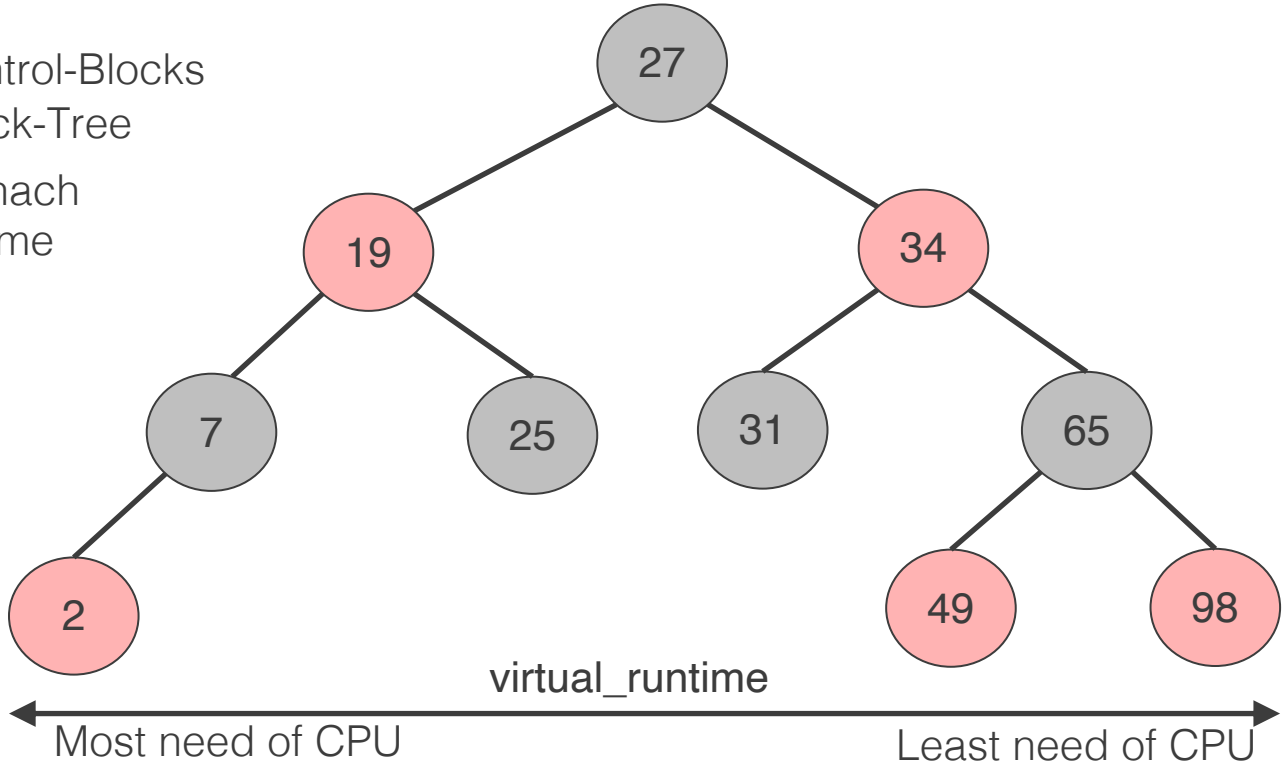
SCHED_OTHER: Completely Fair Scheduler (ab 2.6.24)

- Verwaltung rechenbereiter Threads in einem Red-Black-Tree:
  - Knoten geordnet nach verbrauchter CPU-Zeit (virtual_runtime)
  - Balancierter Binärbaum (der längste Pfad von der Wurzel zu einem Blatt ist maximal zwei Mal so lang wie der kürzeste Pfad von der Wurzel zu einem Blatt)
    - Baumoperation sind damit immer in O(log N) möglich.
  - Keine aufwändige Baumrotationen wie bei einem AVL oder B* Baum

- Ablauf:
  - Scheduler nimmt immer das am weitesten links liegende Blatt (Threads, die bisher am wenigsten Zeit verbraucht haben)
  - Danach wird Thread wieder in Baum eingefügt
  - Dadurch wandern die Knoten und Blätter von links nach rechts

SCHED_OTHER: Completely Fair Scheduler

- Beispiel:
  - Thread-Control-Blocks im Red-Black-Tree
  - Sortierung nach virtual_runtime



SCHED_OTHER: Completely Fair Scheduler

- Prioritäten dienen als Dämpfungsfaktor für die Berechnung der virtual_runtime
  - hohe Priorität → größere Dämpfung → virtual runtime wächst langsamer
  - geringe Priorität → kleine Dämpfung → virtual runtime wächst schneller
  - Damit sind separate Bereit-Listen für verschiedene Prioritäten überflüssig

- Zeitquantum:
  - Ändert sich dynamisch, ZQ = 1 / N; (N = Anzahl rechenbereiter Threads)
  - Interne Parameter:
    - `target_latency`: Zeitspanne bis wann ein Thread an die Reihe kommen muss
    - `minimum_granularity`: minimales Zeitquantum
      (kann nicht beliebig klein werden, da das Umschalten teuer ist)
  - `nice`-Werte fließen in die Berechnung auch mit ein

SCHED_OTHER: Completely Fair Scheduler

- Beispiel zu Parametern
  - `target_latency` = 20ms, N=4 Threads → jeder Thread bekommt 5ms ZQ
  - `target_latency` = 20ms, N=200 T. → jeder Thread bekommt 0.1ms ZQ?
    → `minimum_granularity` legt Minimum für ZQ fest, z.B. 1 oder 4ms

- Weitere Infos: M. Tim Jones, „Inside the Linux 2.6 Completely Fair Scheduler", IBM developer networks, Dec. 2009
  - http://public.dhe.ibm.com/software/dw/linux/l-completely-fair-scheduler/l-completely-fair-scheduler-pdf.pdf

Bemerkungen

- Damit `SCHED_FIFO` und `SCHED_RR` nicht komplett alle anderen Threads blockieren gibt es das Real-Time-Throttling
  → Default: max. 95% CPU-Nutzung für `SCHED_FIFO` und `SCHED_RR`

- Es gibt noch weitere Scheduling-Strategien in Linux:
  - `SCHED_DEADLINE`: basiert auf EDF
  - `SCHED_BATCH`: nicht-interaktive Tasks
  - `SCHED_IDLE`: sehr niedrige Priorität

Beispiel: SCHED_FIFO

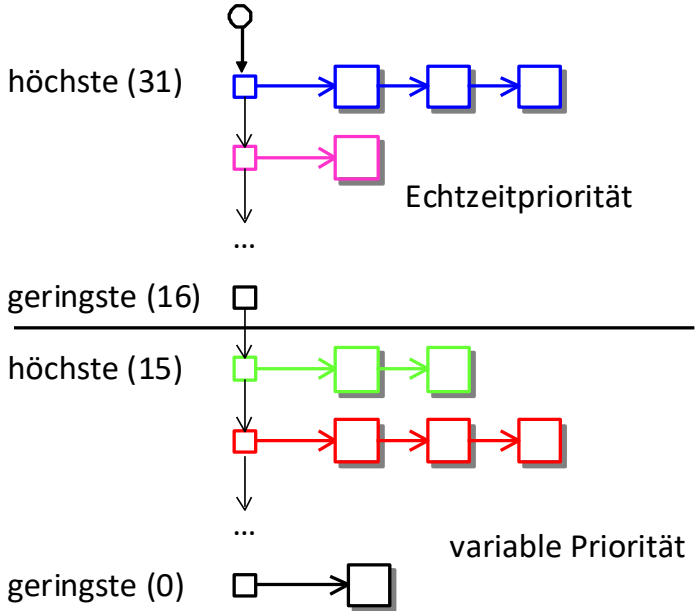
```
#include <pthread.h>

...

void set_prio( pthread_t id, int policy, int prio ) {
   struct sched_param param;

   param.sched_priority = prio;

   if ((pthread_setschedparam( id, policy, &param)) != 0  ) {
      printf("error: unable to change scheduling strategy.\n");
      pthread_exit( (void *) id );
   }
}

...

set_prio( pthread_self(), SCHED_FIFO, 99 ); /* prio. 1 .. 99 */
```

6.12 Fallbeispiel: Microsoft Windows

- Der BS-Kern in Windows 10 ist ein Prozess mit mehreren Threads.
- Job: als Gruppe von Prozessen (ab Windows 2000).
  - Job-Objekt bündelt Prozesse und deren Threads, z. B. für Ressourcenbeschränkung
  - Wird aber beim Scheduling nicht berücksichtigt
- Prozess als Container für Ressourcen:
  - keine Verwandtschaft zwischen Prozessen
- Thread = KL-Thread (Aktivitätsträger):
  - Eins-zu-Eins Abbildung von UL- und KL-Threads
  - ein oder mehrere Thread(s) pro Prozess
- Fiber = UL-Thread.
  - Kein automatisches Scheduling

Scheduling

- Scheduler betrachtet ausschließlich Threads.
- Verdrängendes, auf Zeitscheiben basierendes Scheduling mit Prioritäten:
  - Workstation-Zeitsch.: 20-30 ms
  - Server-Zeitscheibe: 150-180 ms
- 32 Prioritätsklassen:
  - 32 FiFo-Queues
  - Leerlaufthread: 0
  - variable Priorität: 1-15
  - Echtzeitpriorität 16-31 (statisch)

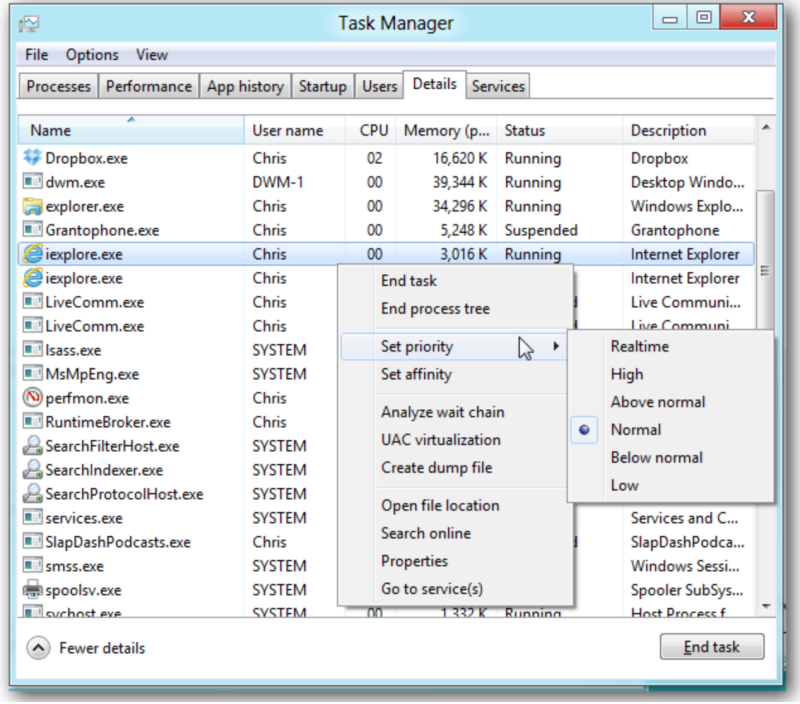


Prozessklasse und relativer Thread-Prioritätslevel

- Prozessklassen (PKL):
  - IDLE: Priorität 4
  - NORMAL: Priorität 8
  - HIGH: Priorität 13
  - REALTIME: Priorität 24
- Thread-Prioritätslevel:
  - IDLE: Priorität 1 bzw. 16
  - LOWEST: Priorität = PKL – 2
  - BELOW_NORMAL: Priorität = PKL – 1
  - NORMAL: Priorität = PKL
  - ABOVE_NORMAL: Priorität = PKL + 1
  - HIGHEST: Priorität = PKL + 2
  - TIME_CRITICAL: Priorität 15 bzw. 31

6.12 Fallbeispiel: Microsoft Windows

- Richtige Echtzeitfähigkeiten (Deadlines) nicht vorhanden, sondern nur höhere Prioritäten.
- Präemptiver Kern; User-Threads können auch KL-Tread verdrängen.
- Prozessklasse über Taskmanager anpassbar oder System-Aufruf →



Dynamische Anpassung der Prioritäten

- Prioritätserhöhung für Threads die Ereignisse für die GUI verarbeiten:
  - Verarbeiten Benutzereingaben und allgemein Nachrichten an Fenster
  - Auf Priorität 14 & Verdopplung der Zeitscheibe
- Prioritätserhöhung nach Blockierung (Priority Boost):
  - Priorität erhöhen (1-8 Stufen), z.B. wenn E/A-Auftrag beendet ist
  - Bonus durch Priority-Boost wird nach jedem Ablauf einer Zeitscheibe jeweils um 1 erniedrigt (bis zum Ausgangswert)
- Verhungern / Prioritätsinversion:
  - Rechenbereite Threads, die seit mind. 3 Sek. auf den Prozessor warten → Priorität auf 15 und Quantum verdoppeln
  - Sobald CPU abgegeben bzw. entzogen wird, Werte wieder zurücksetzen