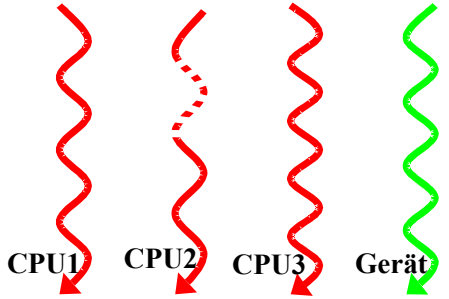
F. Nebenläufigkeit

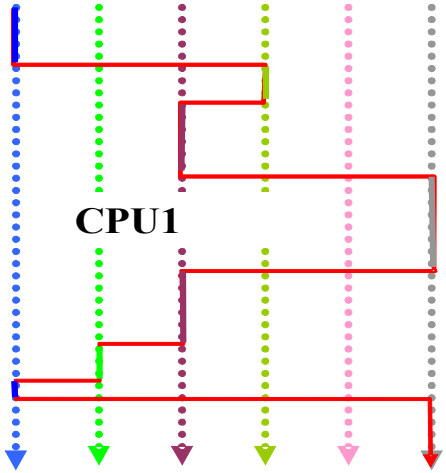
F.1 Terminologie

F.1.1 Nebenläufig vs. Parallel:

- **Parallelität**: simultaner Ablauf.
  - mehrere CPUs oder auch CPU+Geräte,
  - mehrere Orte der Verarbeitung,
  - bzw. Locations of Control ,
  - Ziel: höhere Rechenleistung,
  - gleichzeitiger Ablauf,
  - nichtdeterministisch.



- **Nebenläufig**: paralleler od. verzahnter Ablauf
  - Ziel: Wartezeiten anderweitig nutzen,
  - Umschaltung zwischen Threads =>
  - Resultat nichtdeterministisch.



F.1.2 Prozesse, Threads etc.

- **Adressraum:** Der in einem Zeitpunkt direkt adressierbare Speicher.
- **Thread**: Abfolge von sequentiell auszuführenden Instruktionen:
  - = Ausführungsspur, = Ablauf, = light weight process,
  - oft arbeiten mehrere Threads im selben Adressraum,
  - keine Umschaltung des Adressraumes erforderlich,
  - Threads teilen sich Daten, Code & Betriebsmittel,
  - separater Registersatz & eigener Laufzeitkeller,
  - erleichtert die Nutzung von Multiprozessoren.
- **Prozess: Adressraum + Thread(s).**
- **Task:**
  - instanziertes Programm,
  - erledigt eine best. Aufgabe mithilfe eines oder mehrerer Prozesse.
- **Programm**:
  - statische Darstellung eines Algorithmus,
  - fertig gebundenes Programm auf Festplatte z.B.,
  - Leider auch ausführendes Programm in einer Hauptspeicherpartition.

F.1.3 Multi-X

Multithreading:

- mehrere Threads pro Prozess vorhanden,
- **Zielsetzung**: Nutzung mehrerer Rechnerkerne mit gemeinsamem Speicher.

Multitasking:

- mehrere Tasks (Prozesse, Threads) werden (scheinbar/echt) gleichzeitig ausgeführt,
- **preemptive**: Betriebssystem kann einer Task jederzeit die CPU entziehen,
- **cooperative**: die Tasks müssen den Prozessor freiwillig abgeben,
- **Zielsetzung**: Entgegennehmen von asynchronen Ereignissen etc.

Multiprogramming:

- Mehrere Programme sind im Hauptspeicher vorhanden,
- **Zielsetzung:** Auslasten der CPU während E/A-Operationen & Wartezeiten

Multiprocessing:

- Typischerweise Multiprozessorsysteme mit gemeinsamem Hauptspeicher,
- **Zielsetzung:** Nutzung der mehrfachen Prozessoren.

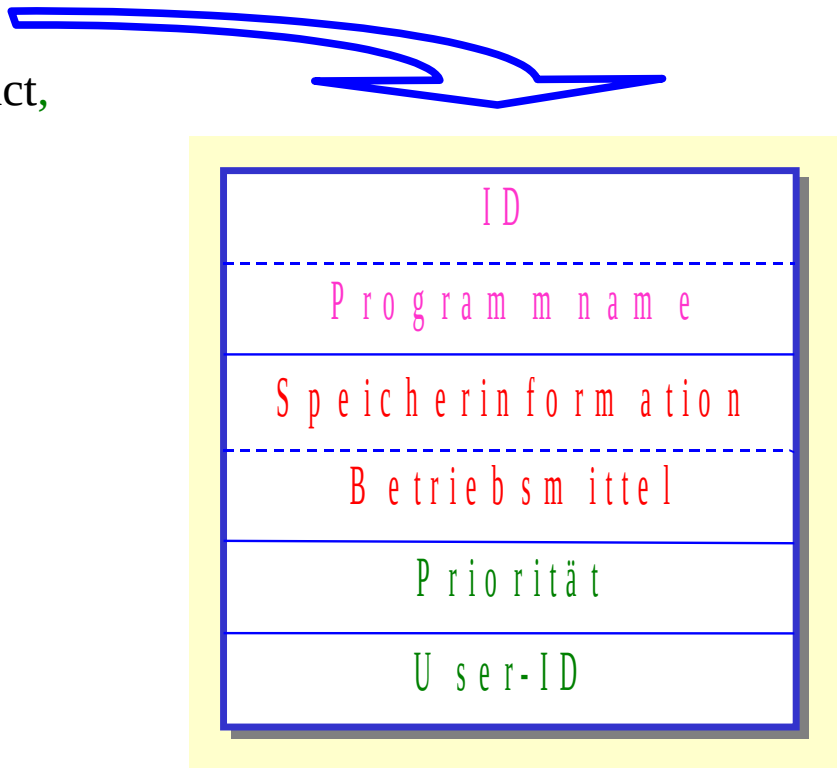
F.2 Prozesse

F.2.1 Der Prozessbegriff

- Die Terminologie ist in der Literatur nicht einheitlich.
- Klassische Unix-Prozesse ohne Threads:
  - Nebenläufigkeit in Programmen nur durch mehrere Prozesse,
  - eigentlich sind Adressraum & Thread zwei orthogonale Konzepte.
- Prozesse mit Threads (in Linux und Windows NT/XP):
  - eigener Adressraum pro Prozess, pro Prozess mindestens ein Thread,
  - Nebenläufigkeit in Programmen durch mehrere Threads.
  - Prozesse als Adressraumhülle für Threads.
  - Thread repräsentiert sequentielle Aktivität.
- Aktivitätsträger:
  - Scheduler bzw. Dispatcher schaltet zwischen Aktivitätsträgern um,
  - Je nachdem sind dies Prozesse oder Threads,
  - Umschalten von Threadzuständen ist leichter.
- Unix-Systeme bezeichnen Threads oft als light weight processes.

F.2.2 Prozesskontrollblock (= process control block (PCB))

- Beschreibt einen Prozess im Kontext des Betriebssystemkerns:
- In Linux als ***task_struct***:
  - pro Prozess oder Thread existiert ein PCB/task_struct,
  - 512 bis 3000 Prozesskontrollblöcke sind möglich,
  - Aufbau der Struktur ist nur dem Kern bekannt.
  - Anwendungen greifen per Handle indirekt zu.
- Prozess-ID und Programmname.
- Zustandsinformation/Resources:
  - Adressraum & Register,
  - geöffnete Dateien,
  - Geräte,
  - Locks ..
- Verwaltungsdaten für Scheduler:
  - Erwartetes Ereignis, Priorität, User-ID,
  - Prozesszustand (z.B. blockiert) &



F.2.3 Prozesserzeugung

- Zwei Arten der Prozesserzeugung:
  - Kopie des rufenden Prozesses anlegen.
  - Aufrufer durch ein neues Programm ersetzen.
- Neue Prozesse entstehen:
  - durch einen Befehl,
  - beim Start des Systems (Dienste),
  - durch Systemaufruf in einem laufenden Prozess.
- Erzeugen von Prozesskopien mit **fork**:
  - erzeugt eine Kopie des rufenden Prozesses,
  - Vater-Sohn Beziehung,
  - mit neuer PID,
  - für Unix.
- Neuer Prozess hat eigenen Adressraum:
  - ältere Unix BS kopieren sofort alle Seiten,
  - neuere Systeme (z.B. Linux) verwenden einen Copy-On-Write Modus und kopieren sukzessive bei Bedarf (bei Schreibzugiffen).

```
int pid=0;
pid = fork();
if (pid < 0) {
      /* fork failed */
}
else if (pid > 0) {
      /* Vater-Code */
} else {
      /* Kind-Code */
}
```

- Aufrufer durch ein neues Programm ersetzen:
  - int **execve**(char *path, char *const argv[]);
  - Aufruf kehrt nicht mehr zurück, rufender Prozess wird ersetzt.
  - neues Programm übernimmt PID.
- Problem: bei älteren *fork* Implementierungen
  - kopieren den ganzen Adressraum,
  - Start eines neuen Programms ist teuer,
  - falls *fork* und *execve* verwendet würden.
- Lösung: ***vfork***
  - Kind-Prozess teilt Adressraum mit Vater.
  - aber Aufrufer (Vater) wird blockiert, bis Kind *execve* oder *exit* ruft.
- Ablauf beim Erzeugen eines neuen Prozesses:
  - PCB initialisieren, Prozess-ID festlegen,
  - Adressraum anlegen bzw. kopieren,
  - Befehlszeiger auf Startadresse setzen,
  - Eintrag des PCBs in die *Bereit*-Warteschlange.
- Hintergrundprozesse werden in Unix als **Daemon** bezeichnet
  - z.B. für Spooler.

F.2.4 CreateProcess erzeugt einen neuen Prozess in Windows:

- Keine Kopie des Adressraums, keine Vater-Sohn-Beziehung.
- Beispiel:

```
main() {
   STARTUPINFO          si;   // windows stuff
   PROCESS_INFORMATION  pi;   // handle

   GetStartupInfo(&si);       // get some default values
   CreateProcess(
      „MyProg.exe“,  // program (incl. path)
      „“,            // command line params.
      NULL,NULL,     // process/thread security attrib.
      FALSE,         // handles vererbbar?
      CREATE_NEW_CONSOLE,
      NULL,          // environment block
      NULL,          // current dir. for new process
      &si,           // startup info.
      &pi            // process info.
   );
}
```

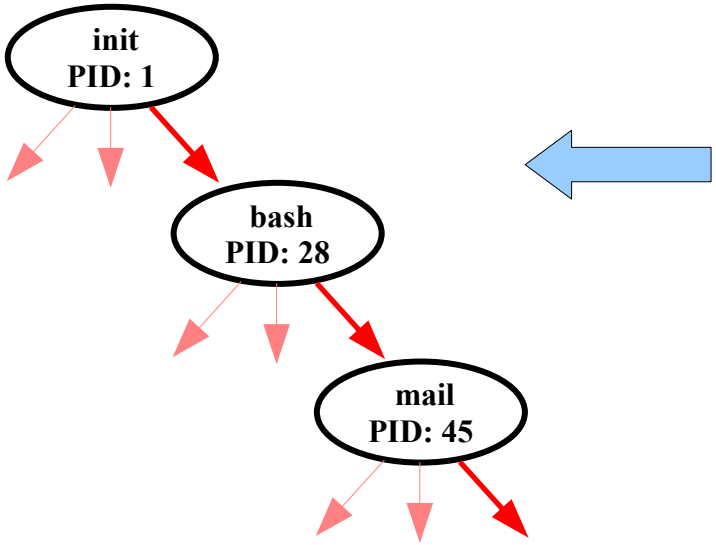
F.2.5 Hintergrundprozesse (... Vista)

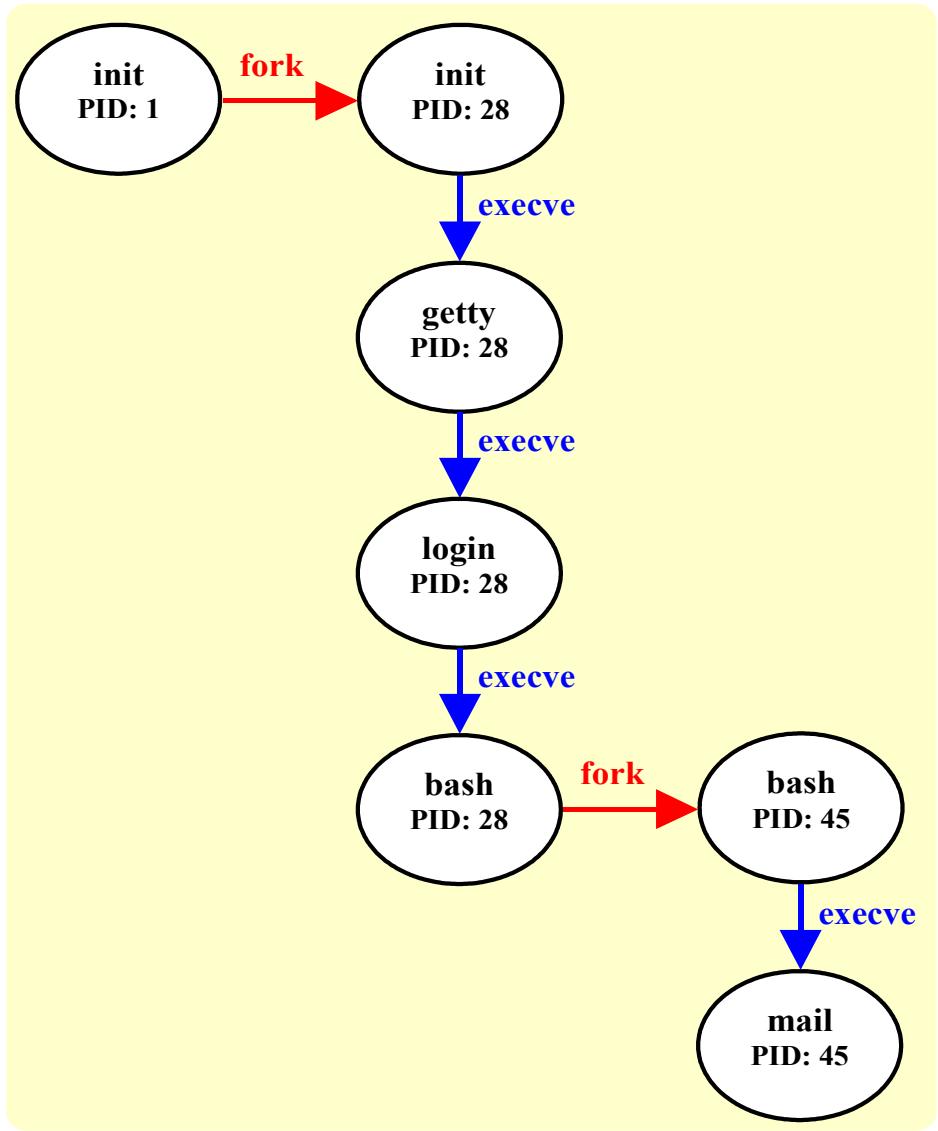
- In Windows existieren eine Anzahl Hintergrundprozesse:
  - als Dienste (Services) bezeichnet, werden überwacht durch den Service Control Manager,
  - implementieren z.B. Virusscan, Netzverwaltung, Fehlerlog, Wechselmedien, DRM, ...

Name	Beschreibung	Status	Starttyp
eLock Service	Acer eLock Management Service	Gestartet	Automatisch
eNet Service	Acer eNet Management Service	Gestartet	Automatisch
Enumeratordienst für tragbare Geräte	Erzwingt Gruppenrichtlinien für Wechsel-Massenspeichergeräte. Ermöglicht Anwendungen wie Windows Media Player und dem Bildimport-Assistenten, Inhalte unter Verwendung von Wechsel-Massenspeichergeräten zu übertragen und zu synchronisieren.	Gestartet	Automatisch
ePower Service	Acer ePower Management Service	Gestartet	Automatisch
eRecovery Service	Acer eRecovery Management	Gestartet	Automatisch
eSettings Service	Acer eSettings Management Service	Gestartet	Automatisch
Funktionssuche-Ressourcenveröffentlichung	Veröffentlicht diesen Computer und die daran angeschlossenen Ressourcen, damit sie über das Netzwerk gesucht werden können. Wenn dieser Dienst beendet wird, werden die Netzwerkressourcen nicht mehr veröffentlicht, damit sie von anderen Computer…	Gestartet	Automatisch
Gruppenrichtlinienclient	Von dem Dienst werden Einstellungen angewendet, die von Administratoren mithilfe der Gruppenrichtlinienkomponente für den Computer und Benutzer konfiguriert wurden. Wenn der Dienst beendet oder deaktiviert wurde, werden die Einstellungen nicht ang…	Gestartet	Automatisch
HP CUE DeviceDiscovery Service	Von diesem Dienst werden CUE-Geräte auf Ihrem System erkannt und überwacht.	Gestartet	Automatisch
IKE- und AuthIP IPsec-Schlüsselerstellungsmo…	Die IKEEXT-Diensthosts der Schlüsselerstellungsmodule für IKE (Internet Key Exchange) und Auth-IP (Authenticated Internet Protocol). Diese Schlüsselerstellungsmodule werden zur Authentifizierung und zum Schlüsselaustausch in Internet Protocol Security …	Gestartet	Automatisch
Intel(R) Matrix Storage Event Monitor		Gestartet	Automatisch
IP-Hilfsdienst	Bietet automatische IPv6-Konnectivität über ein IPv4-Netzwerk. Wenn dieser Dienst beendet wird, verfügt der Computer nur IPv6-Konnectivität, wenn er an ein IPv6-Netzwerk angeschlossen wird.	Gestartet	Automatisch
IPsec-Richtlinien-Agent	IPsec (Internet Protocol Security) unterstützt die Peerauthentifizierung auf Netzwerkebene, Datenursprungsauthentifizierung, Datenvertraulichkeit (Verschlüsselung) und Schutz vor Wiedergabeangriffen. Dieser Dienst erzwingt die IPsec-Richtlinien, die mit d…	Gestartet	Automatisch
Kryptografiedienste	Bietet vier Verwaltungsdienste: den Katalogdatenbankendienst, der die Signaturen von Windows-Dateien bestätigt und die Installation neuer Programme ermöglicht; den geschützten Stammdienst, der diesem Computer vertrauenswürdige Zertifikate von Sta…	Gestartet	Automatisch
LightScribeService Direct Disc Labeling Service	Used by the LightScribe software components to support 3rd party disc labeling applications using the LightScribe COM Application Programming Interface (LSCAPI). This service needs to run for LightScribe direct disc labeling to work.	Gestartet	Automatisch
MobilityService		Gestartet	Automatisch
Multimediaklassenplaner	Ermöglicht eine relative Prioritätseinstufung von Aufgaben basierend auf systemweiten Aufgabenprioritäten. Dies ist vor allem für Multimediaanwendungen bestimmt. Wenn dieser Dienst angehalten wird, werden die einzelnen Aufgaben auf die jeweilige Stan…	Gestartet	Automatisch
Net Driver HPZ12			Automatisch
Netzwerklistendienst	Identifiziert die Netzwerke, mit denen der Computer eine Verbindung hergestellt hatte, sammelt und speichert Eigenschaften für diese Netzwerke, und benachrichtigt Anwendungen, wenn sich diese Eigenschaften ändern.	Gestartet	Automatisch
Netzwerkspeicher-Schnittstellendienst	Dieser Dienst stellt Netzwerkbenachrichtigungen (z. B. beim Hinzufügen/Löschen von Schnittstellen) für Benutzermodusclients bereit. Wenn Sie diesen Dienst beenden, wird die Netzwerkkonnektivität getrennt. Wenn dieser Dienst deaktiviert wird, können and…	Gestartet	Automatisch
NLA (Network Location Awareness)	Sammelt und speichert Konfigurationsinformationen für das Netzwerk und benachrichtigt Programme, wenn diese Informationen geändert werden. Wenn dieser Dienst beendet wird, sind die Konfigurationsinformationen möglicherweise nicht verfügbar. Wenn …	Gestartet	Automatisch
PC Tools Auxiliary Service	Bietet zusätzliche Sicherheitsfunktionen von PC Tools. Wenn dieser Service deaktiviert wird, ist nur ein eingeschränkter Schutz gegen Spyware möglich.	Gestartet	Automatisch
PC Tools Security Service	Schützt das System gegen Spyware und Malware. Wenn dieser Service deaktiviert wird, ist das System nicht gegen Spyware geschützt.	Gestartet	Automatisch
Plug & Play	Ermöglicht dem Computer, Hardwareänderungen zu erkennen und sich ohne oder mit geringer Benutzerinteraktion darauf einzustellen. Beenden oder Deaktivieren dieses Dienstes wird die Systemstabilität beeinträchtigen.	Gestartet	Automatisch
Pml Driver HPZ12		Gestartet	Automatisch
Programmkompatibilitäts-Assistent-Dienst	Bietet Unterstützung für den Programmkompatibilitäts-Assistenten. Wenn dieser Dienst beendet wird, wird der Programmkompatibilitäts-Assistent nicht ordnungsgemäß ausgeführt. Wenn dieser Dienst deaktiviert wird, können alle von diesem Dienst abhängig…	Gestartet	Automatisch
ReadyBoost	Bietet Unterstützung zum Verbessern der Systemleistung mithilfe von ReadyBoost.	Gestartet	Automatisch
Remoteprozeduraufruf (RPC)	Dient der Endpunktzuordnung und als COM-Dienststeuerungsverwaltung. Wenn dieser Dienst beendet oder deaktiviert wird, werden Programme, die COM- oder RPC-Dienste verwenden, nicht mehr richtig funktionieren.	Gestartet	Automatisch
Sekundäre Anmeldung	Aktiviert das Starten von Prozessen mit verschiedenen Anmeldeinformationen. Wenn dieser Dienst beendet wird, wird dieser Typ von Anmeldezugriff nicht mehr verfügbar sein. Wenn dieser Dienst deaktiviert wird, können alle Dienste, die explizit von diesem …	Gestartet	Automatisch
Server	Unterstützt Datei-, Drucker- und Named-Piped-Freigabe für diesen Computer über das Netzwerk. Diese Funktionen sind nicht mehr verfügbar, falls dieser Dienst beendet wird. Falls dieser Dienst deaktiviert wird, können die Dienste, die von diesem Dienst aus…	Gestartet	Automatisch
Shellhardwareerkennung	Zeigt Meldungen für Hardwareereignisse für automatische Wiedergabe an.	Gestartet	Automatisch
Sicherheitskonto-Manager	Durch den Start dieses Dienstes wird anderen Diensten signalisiert, dass die Sicherheitskontenverwaltung (SAM) bereit ist, Anforderungen anzunehmen. Wenn Sie diesen Dienst deaktivieren, wird verhindert, dass andere Dienste im System benachrichtigt werd…	Gestartet	Automatisch
Sitzungs-Manager für Desktopfenster-Manager	Stellt Start- und Wartungsdienste für den Desktopfenster-Manager bereit	Gestartet	Automatisch
Softwarelizenzierung	Aktiviert das Herunterladen, die Installation und die Durchsetzung digitaler Lizenzen für Windows und Windows-Anwendungen. Wird dieser Dienst deaktiviert, werden das Betriebssystem und lizenzierte Anwendungen in einem reduzierten Funktionsmodus aus…	Gestartet	Automatisch
Superfetch	Verwaltet und verbessert die Systemleistung im Zeitablauf.	Gestartet	Automatisch
Tablet PC-Eingabedienst	Ermöglicht die Stift- und Freihandfunktionalität von Tablet PC.	Gestartet	Automatisch
TCP/IP-NetBIOS-Hilfsdienst	Bietet Unterstützung für den NetBIOS-über-TCP/IP-Dienst (NetBT) und die NetBIOS-Namensauflösung für Clients im Netzwerk, so dass Benutzer Daten gemeinsam nutzen, drucken und sich am Netzwerk anmelden können. Diese Funktionen sind nicht mehr ve…	Gestartet	Automatisch
Terminaldienste	Ermöglicht Benutzern das Herstellen einer interaktiven Verbindung mit einem Remotecomputer. Remotedesktop und Terminalserver funktionieren nur zusammen mit diesem Dienst. Wenn Sie die Remoteverwendung dieses Computers verhindern möchten, deak…	Gestartet	Automatisch
TVEnhance Background Capture Service (TBCS)	Provides background buffering, recording and burning functionality for TVEnhance Capturing	Gestartet	Automatisch
TVEnhance Task Scheduler (TTS))	Enables a user to configure and schedule a automated task for TVEnhance Scheduling	Gestartet	Automatisch
Überwachung verteilter Verknüpfungen (Client)	Hält Verknüpfungen für NTFS-Dateien auf einem Computer oder zwischen Computern in einem Netzwerk aufrecht.	Gestartet	Automatisch
UPnP-Gerätehost	Ermöglicht es, dass UPnP-Geräte auf diesem Computer gehostet werden können. Wenn dieser Dienst gestoppt wird, sind alle gehosteten UPnP-Geräte nicht mehr betriebsbereit, und es können keine weiteren gehosteten Geräte hinzugefügt werden. Wenn di…	Gestartet	Automatisch
WebClient	Ermöglicht Windows-basierten Programmen, Internet-basierte Dateien zu erstellen, darauf zuzugreifen und sie zu verändern. Wenn dieser Dienst beendet wird, werden diese Funktionen nicht mehr zur Verfügung stehen. Falls dieser Dienst deaktiviert wird, k…	Gestartet	Automatisch
Windows Driver Foundation - Benutzermodus-…	Verwaltet Benutzermodus-Treiberhostprozesse	Gestartet	Automatisch
Windows-Audio	Verwaltet Audioinhalte für Windows-basierte Programme. Wenn dieser Dienst beendet wird, funktionieren Audiogeräte und -effekte nicht ordnungsgemäß. Wenn dieser Dienst deaktiviert wird, können die Dienste, die von diesem Dienst explizit abhängig sind, …	Gestartet	Automatisch
Windows-Audio-Endpunkterstellung	Verwaltet Audiogeräte für den Windows-Audiodienst. Wenn dieser Dienst beendet wird, funktionieren Audiogeräte und -effekte nicht ordnungsgemäß. Wenn dieser Dienst deaktiviert wird, können die Dienste, die von diesem Dienst explizit abhängig sind, nich…	Gestartet	Automatisch
Windows-Bilderfassung	Stellt Bilderfassungsdienste für Scanner und Kameras zur Verfügung	Gestartet	Automatisch
Windows-Defender	Überprüft den Computer auf unerwünschte Software, plant Überprüfungen und lädt die neuesten Softwaredefinitionen herunter.	Gestartet	Automatisch
Windows-Ereignisprotokoll	Dieser Dienst verwaltet Ereignisse und Ereignisprotokolle. Er unterstützt die Protokollierung, Abfrage und Abonnierung von Ereignissen sowie die Archivierung von Ereignisprotokollen und die Verwaltung von Ereignismetadaten. Er kann Ereignisse im XML- und …	Gestartet	Automatisch
Windows-Fehlerberichterstattungsdienst	Ermöglicht das Berichterstatten über Fehler bei nicht mehr funktionierenden und reagierenden Programmen und das Angeben von Lösungen. Ermöglicht außerdem das Generieren von Protokollen für Diagnose- und Reparaturdienste. Wenn dieser Dienst been…	Gestartet	Automatisch
Windows-Firewall	Die Windows-Firewall trägt zum Schutz des Computers bei, indem der Zugriff durch nicht autorisierte Benutzer auf den Computer über das Internet bzw. ein Netzwerk verhindert wird.	Gestartet	Automatisch
Windows-Suche	Stellt Inhaltsindizierung und Eigenschaftenzwischenspeicherung für Dateien, E-Mail-Nachrichten und anderen Inhalt (über Erweiterbarkeits-APIs) bereit. Der Dienst reagiert auf Datei- und E-Mail-Benachrichtigungen aufgrund von geändertem Inhalt. Wenn de…	Gestartet	Automatisch
Windows-Verwaltungsinstrumentation	Bietet eine standardmäßige Schnittstelle und Objektmodell zum Zugreifen auf Verwaltungsinformationen über das Betriebssystem, Geräte, Anwendungen und Dienste. Die meiste Windows-basierte Software kann nicht ordnungsgemäß ausgeführt werden, fall…	Gestartet	Automatisch
Windows-Zeitgeber	Behält Datums- und Zeitsynchronisation auf allen Clients und Servern im Netzwerk bei. Wird dieser Dienst beendet, sind Datums- und Zeitsynchronisation nicht verfügbar. Wird dieser Dienst deaktiviert, können keine explizit von der Synchronisation abhängige…	Gestartet	Automatisch
xAudioService	User-mode gate for Modem Speakerphone	Gestartet	Automatisch
Zugriff auf Eingabegeräte	Ermöglicht einen Standardeingabezugriff auf Eingabegeräte (HID-Geräte), der die Verwendung von vordefinierten Abkürzungstasten auf Tastaturen, Fernbedienungen und anderen Multimediageräten aktiviert und unterstützt. Wenn dieser Dienst beendet wi…	Gestartet	Automatisch

F.2.6 Prozesshierarchie in Unix

- Vater-Kind-Beziehung zwischen Prozessen führt zu Prozesshierarchie:
  - Beim Start von Unix wird *init* gestartet.
  - *init* erzeugt definierte Anzahl Terminals.
  - Terminals (*getty*-Proz.) führen *login* aus.
  - Nach erfolgreicher Anmeldung wird Terminal durch shell-Prozeß ersetzt.
- Resultierende Hierarchie:
  - Drei Prozess-IDs: #1, #28, #45,
  - weitere Abspaltungen möglich:



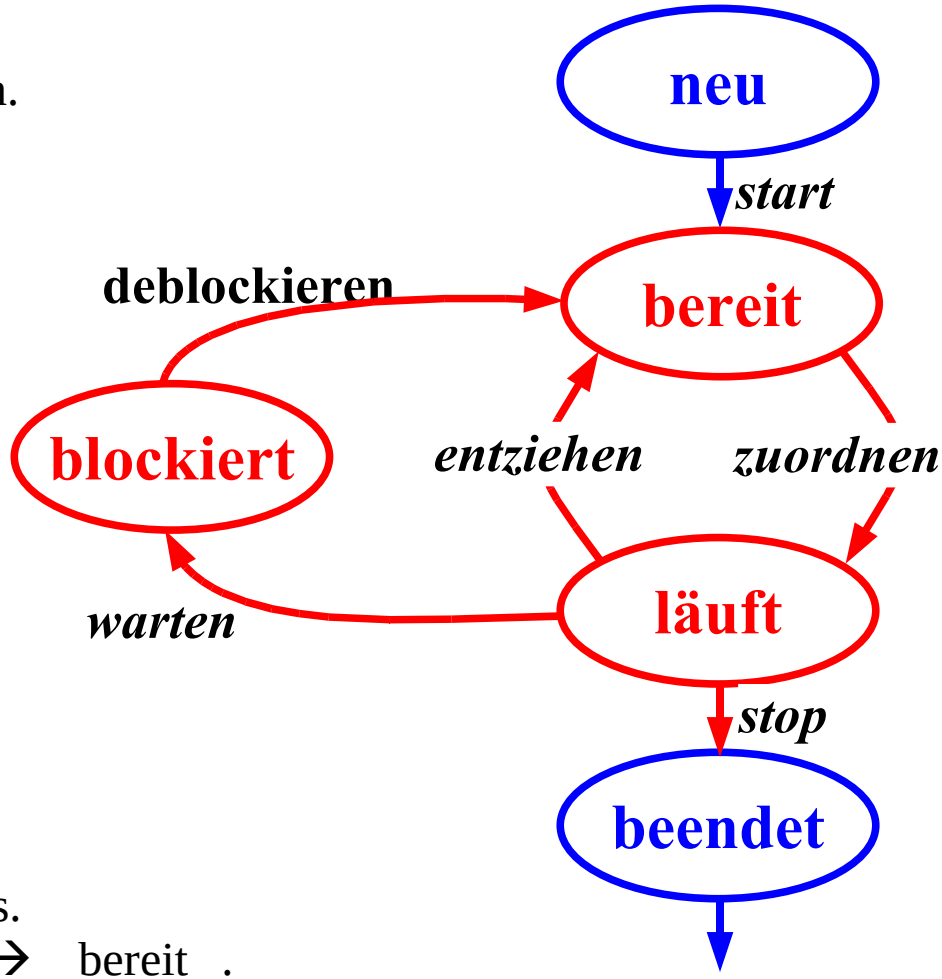


F.2.7 Beenden von Prozessen

- Prozess führt letzte Anweisung aus und ruft das BS mit exit(int status).
- Terminiert der Vaterprozess vor dem Kindprozess, so werden bei Linux die Kinder dem Init-Prozess zugeordnet.
- In Unix kann Elternprozess auf Beendigung eines Kindprozesses inklusive Statusrückgabe warten → *waitpid* oder *wait*.
- Evt. abrupt beenden mit *kill*, mögl. Gründe:
  - allozierte Ressourcen wurden verbraucht (Zeit, Speicher, Disk),
  - Kindprozess wird nicht länger benötigt,
  - Elternprozess terminiert.
- **Zombie-Prozess** in Unix:
  - Ein Prozess dessen *Exit-Code* nicht abgeholt wurde.
  - Infos über Kind (PCB) werden für Vater-Prozess noch gespeichert.
- Windows bietet *TerminateProcess(handle,exitcode)*:
  - im Prinzip nicht verwendet, da keine Vater-Sohn-Beziehung,
  - Anwendungen bestehen aus Threads, diese werden bei Prozessende automatisch terminiert.

F.2.8 Prozesszustände im klassischen Sinne

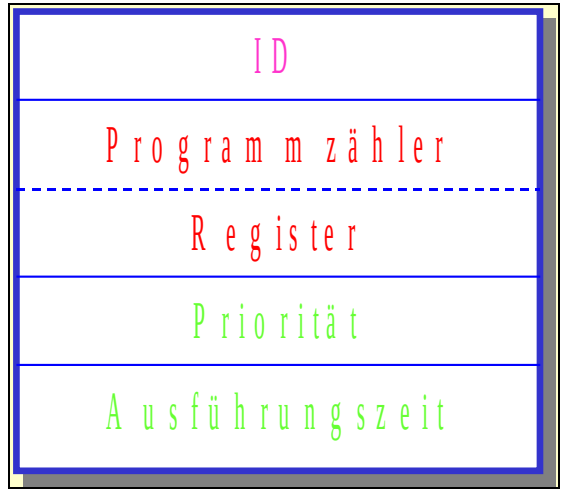
- Prozesszustände (vereinfacht):
  - **beendet**: Prozess hat Ausführung abgeschlossen.
  - **bereit**: Prozess wartet auf die CPU-Zuteilung,
  - **blockiert**: Prozess wartet auf ein Ereignis,
  - **läuft**: Prozess wird auf CPU ausgeführt,
  - **neu**: Prozess wurde erzeugt.
- Ereignisse:
  - Abschluss einer E/A-Operation,
  - verschiedene Signale,
  - ...



- Für alle Zustandsübergänge sind Kernfunktionen vorhanden:
  - ***stop, start***: Prozessende & -beginn.
  - ***entziehen***: Umschalten auf einen anderen Prozess. Der bisher rechnende Prozess bleibt fortsetzbar. → bereit .
  - ***Warten***: CPU-Freigabe, da Bedingung nicht erfüllt. → blockiert .
  - ***Zuordnen***: Auswahl des nächsten Prozesses aus der Bereit-Liste → läuft .
  - ***Deblockieren***: Ereignis, auf welches der Prozess gewartet hat tritt ein. → bereit .

F.3 Thread Varianten

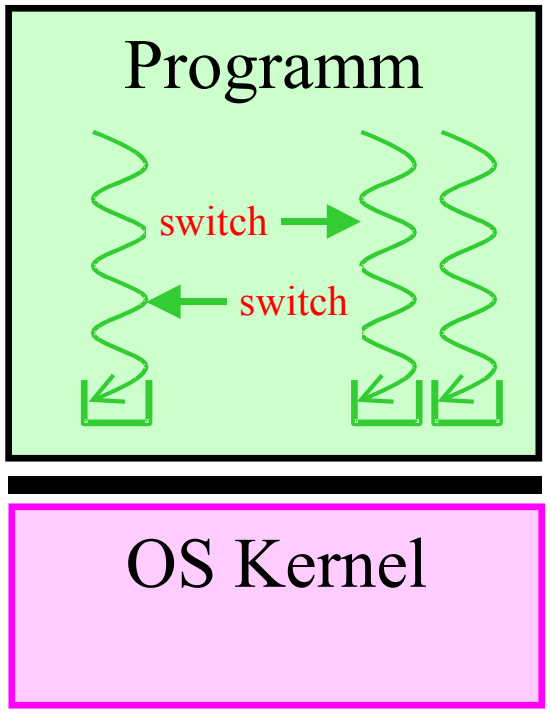
- Threads eines Proz. laufen im selben Adressraum → kein HW-Schutz.
- Nebenläufigkeit durch mehrere Threads ist aber effizienter:
  - Adressraumübergreifende Kommunikation ist langsam und umständlich.
  - Wechsel des Prozesskontextes ist teuer → TLB und Cache (bei IA32) spülen.
- Verwaltung durch Thread-Kontrollblock (=TCB):
  - enthält Zustand, wenn der Thread gerade nicht läuft.
  - Thread-Identifikation (ID),
  - Maschinenzustand (Register),
  - Verwaltungsdaten für Scheduler.



- Typische Einsatzgebiete:
  - parallele Berechnung für Multiprozessor,
  - blockierendes Warten auf Ereignisse (Eingabe ...),
  - separater Thread für die Benutzerschnittstelle vorsehen,
  - gleichzeitige Bearbeitung von parallelen Einzelaufgaben (Web-Requests, ...).

F.3.1 User-Level Threads

- Explizites Umschalten zwischen Threads ohne OS-Unterstützung.
  - User-Level-Threads sind dem OS-Kern nicht bekannt,
  - API-Aufrufe wie wait(), suspend(), switch() ...
  - Entspricht sogenannten Koroutinen.



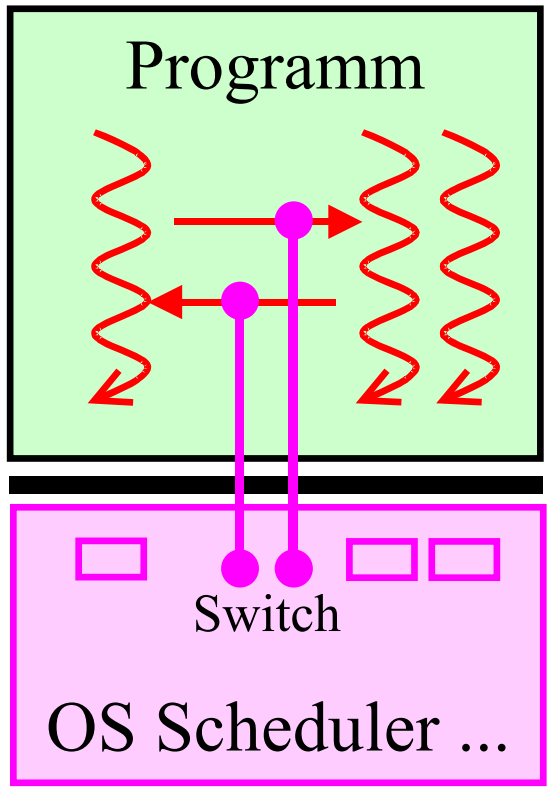
- Vorteile:
  - unabhängig vom unterliegenden BS.
  - Scheduling in der Hand des Programmierers.
  - Umschalten sehr schnell - ohne Umweg über OS.
  - Threads auch für BS, welches keine Threads kennt.

- Nachteile:
  - Bei einem blockierenden Systemaufruf werden alle Threads eines Prozesses blockiert.

- Beispiele:
  - Windows Fibers, GNU Portable Threads, JVM Greenthreads ...

F.3.2 Kernel-Level Threads

- KL-Threads sind dem Kern bekannt und unterliegen dessen Kontrolle:
  - Umschaltung zwischen den Threads eines Prozesses geschieht durch den OS-Scheduler,
  - müssen nicht, können jedoch mit einem Benutzerprozess verbunden sein,
  - Umschaltung durch BS auch prozessübergreifend,
  - Kern verwaltet Thread- und Prozesstabelle.
- Vorteile:
  - Threads eines Prozesses können sich nicht blockieren,
  - transparent für den Programmierer,
  - Scheduling für Gesamtsystem.
- Nachteile:
  - längere Umschaltzeiten.
- Beispiele:
  - Linux (neuere Versionen),
  - Windows NT/XP/Vista,
  - Solaris 2,
  - BeOS, ...



Programmbeispiel - Threads in Windows:

```
#include "stdio.h"
#include "windows.h"

HANDLE hEvent, hChild;

DWORD WINAPI Child( LPVOID lpParm ) {
    int * iParam = (int *)lpParm; // Parameter umkopieren, sonst später
    int loop = *iParam;           // evt. ungültige Referenz

    while (loop>0) { loop--; };

    printf("'Child' signalisiert:\n");
    SetEvent(hEvent);
    return 0;
}
int main(int argc, char* argv[]) {
    DWORD  loop = 500*1000*1000;

    // defaultSecurity, autoReset, non-signalled, no-name
    hEvent  =CreateEvent(NULL, 0, 0, NULL);

    // defaultSecurity, stack, procAddr, tParam, noname
    hChild =CreateThread(NULL, 8, &Child, &loop, 0, 0);

    printf( "'main' wartet auf Signal.\n" );
    result =WaitForSingleObject( hEvent, INFINITE );
}
```

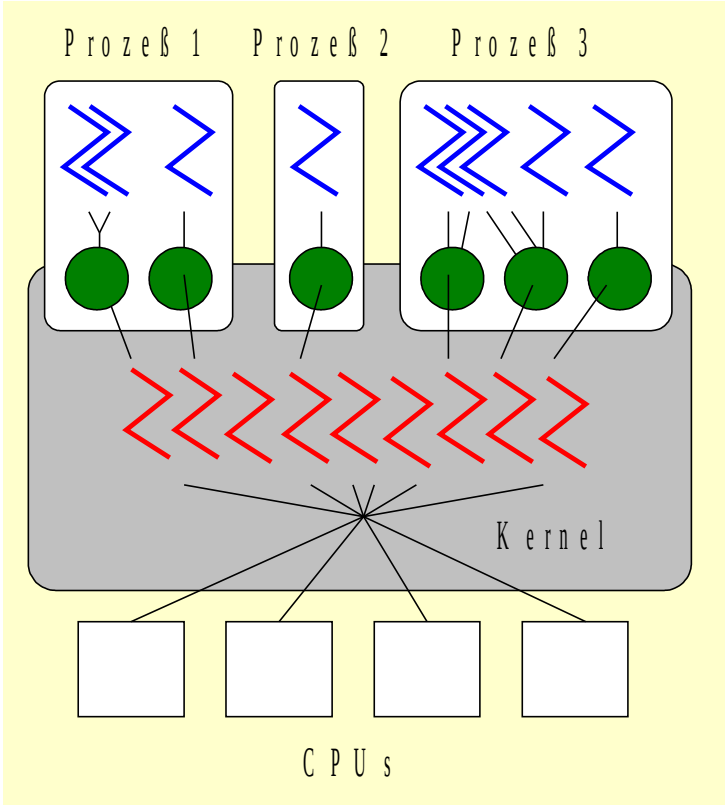
F.3.3 Hybride Threads

- Lightweight-Prozesse (LWP):
  - zur Kommunikation der UL-Threads mit Kern,
  - Jeder LWP hat einen Kernel-Level Thread,
  - langsam falls Adressraumwechsel nötig,
  - Jeder Prozess hat >= 1 LWP.

- User-Level Threads:
  - Multiplexen von UL-Threads auf LWPs.
  - schneller Wechsel → ohne Kern-Interaktion.
  - Mit LWPs verbundene UL-Threads können arbeiten, andere sind blockiert.

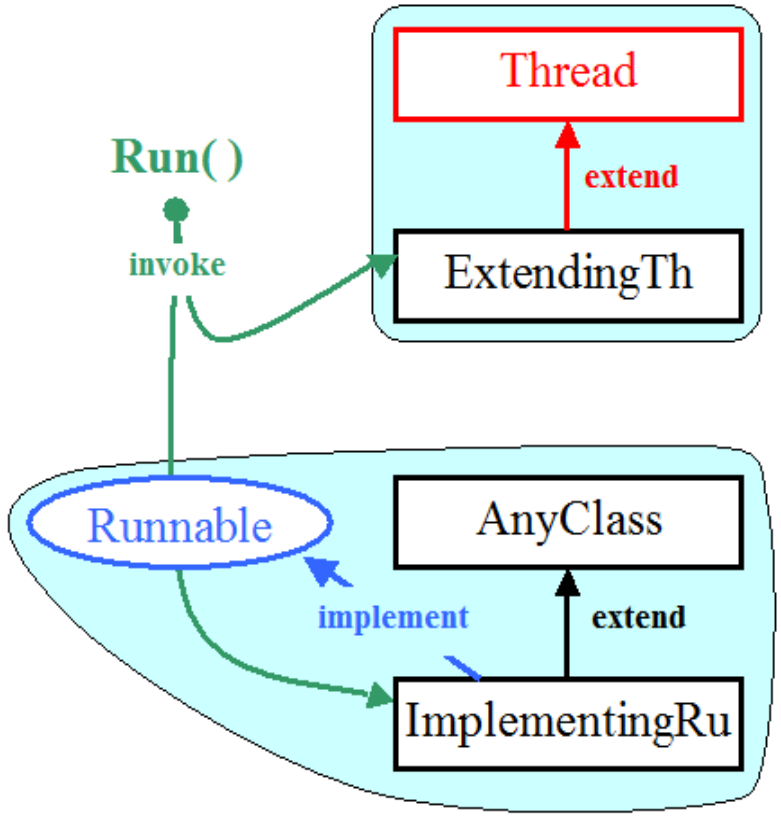
- Kernel-Level Threads:
  - Dienen der Ausführung der Kern-Operationen.
  - relativ schnelle Wechsel (nur ein Adressraum),
  - optional feste Prozessor-Zuordnung.

- Beispiel: Solaris 2, Linux.



F.3.4 Threads in Java

- Erweitern der Klasse *java.lang.***Thread**
  - Methode ***run()*** wird **überschrieben**.
- Alternativ das Interfaces *java.lang.***Runnable** implementieren.
  - Methode ***run()*** wird **implementiert**.
- Methoden der Klasse Thread:
  - Priorität (1-10): *setPriority(int p);*
  - Warten auf Beendigung: *join();*
  - Verzögern: *sleep(int milliSec);*
  - CPU Abgeben: *yield();*
  - Starten: *start(); ...*
- Veraltete (deprecated) Methoden:
  - Abbrechen: *stop();*
  - Aufwecken: *resume();*
  - Unterbrechen: *suspend();*
  - Erhöhen die Gefahr von Verklemmungen.



F.3.5 Beispiel: Konkurrierende Java Threads.

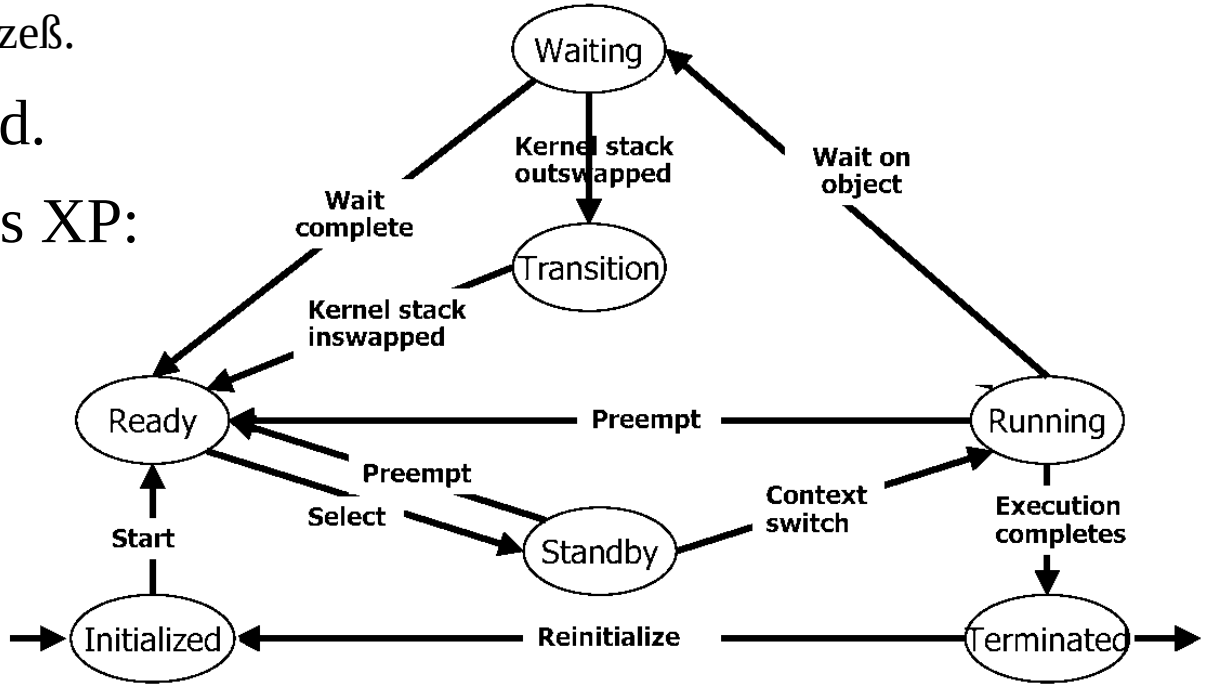
- Ausgabe entsprechend Bildschirmauszug.
  - Keine deterministische Ausgabe, da keine Synchronisierung vorgesehen,
  - Synchronisierungstechniken später.

```
public class threadExample extends Thread{
        static long count=40; static char last;
        static Object lock=new Object();
        char thName;

        threadExample (char name) { thName=name; }

        public void run(){
                while (count >0) {
                        if ( last != thName){
                                count--;
                                synchronized( lock ){          // atomic block
                                        System.out.print(thName); // print on change
                                        last= thName;          // Thread    Name    is executing
        }       }       }       }
        public static void main( String[] args ) {
                new threadExample ('A').start();               // new threads
                new threadExample ('B').start();
                new threadExample ('C').start();
}       }
```

```
c:\  Command Prompt

C:\j2sdk>java threadExample
ABCBCBCACBCACBCACBCACBCACBCACBCACBCACBCA
C:\j2sdk>java threadExample
ABCBACABACABABCBABCBABCBABCBABCBABCBABCA
C:\j2sdk>java threadExample
CBABCBABCBABCBABCBABCBABCBABCBABCBABCBAC
C:\j2sdk>
```

F.3.6 Threads in Windows XP

- Der OS-Kern in Windows XP ist ein Prozess mit mehreren Threads.
- **Job**: als Gruppe von Prozessen (ab Windows 2000).
- **Prozess** als Container für Ressourcen:
  - keine Verwandtschaft zwischen Prozessen.
- **Thread** als Aktivitätsträger:
  - Eins-zu-Eins Abbildung von UL- und KL-Threads.
  - ein oder mehrere Thread(s) pro Prozeß.
- **Fiber** als User-Level Thread.
- Threadzustände in Windows XP:



F.3.7 Threads in Linux

- Ziel: Gemeinsame Nutzung der umgebenden Prozess-Ressourcen.
- User-Level Threads bereitgestellt durch verschiedene Bibliotheken:
  - z.B. GNU Pth (Portable Threads) → pthread_create , &
  - Abbildung auf LWPs abhängig von Implementierung.
- Lightweight Process als Schedulable Entity .
  - UL Threads müssen zur Ausführung an einen LWP gebunden werden,
  - Erst dann können sie vom Kern gescheduled werden,
  - evtl. nicht jederzeit genug LWPs im Kern vorrätig.
- Kernel-Level Threads (KL):
  - Für Kernel-Dienste oder LWPs (für User-Levl Threads),
  - Ohne eigenen Adressraum.
- ***clone***: erzeugt KL-Thread mit eigener TID:
  - Parameter: **CLONE_VM: share virtual address space**.

F.3.8 Beispiel: zusätzlicher KL-Thread in Linux

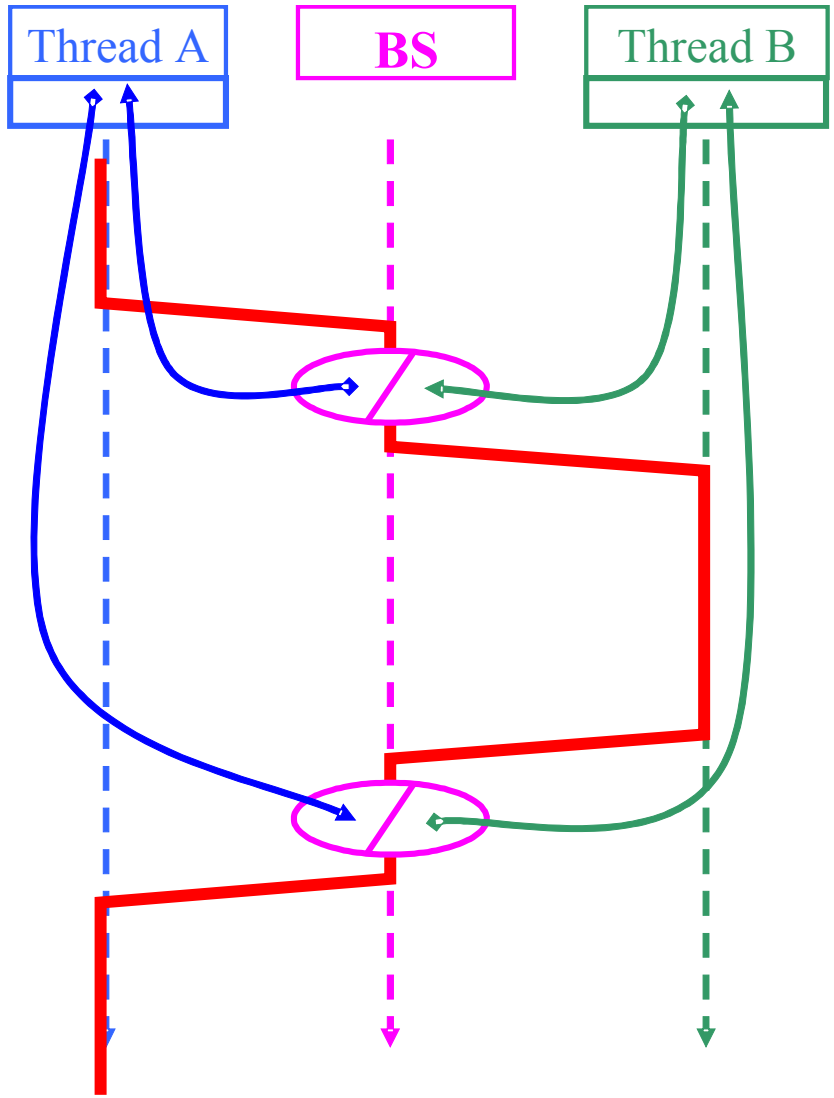
```
#include <sched.h>

int  thread_stack[4000];                    /* Stack für den zweiten Thread */

int  thread_proc(void *arg) {
      /* Endlosschleife für zweiten Thread */
      for(;;) printf("*    );
}

 main() {
   /* int clone(int (*fn)(void*), void *stack, int flgs, void *arg)        */;
   /* stack wächst nach unten → Start am Ende des Arrays    +4000        */
   clone(thread_proc, thread_stack+4000, CLONE_VM, 0);

  /* Endlosschleife für Hauptthread */
  for(;;) printf(".    );
}
```

F.4 Prozess- bzw. Threadumschaltung

- **Dispatcher**: Modul, welches das Umschalten implementiert.
- **Scheduler**: Modul, welches die CPU-Zuteilungsstrategie realisiert.
- Gründe für Threadwechsel:
  - freiwillige Abgabe der CPU,
  - Warten auf E/A-Operation,
  - Interrupt von HW & SW,
  - Rechenzeit abgelaufen.
- Was wird umgeschaltet?
  - Prozessorregister austauschen, evtl. auch Gleitkomma- & MMX-Register,
  - beim Austauschen des Stackpointerregisters wird auch der Keller umgeschaltet,
  - beim Prozesswechsel auch Segment- und Seitentabelle austauschen (Adressraum),
  - Kontrollblöcke, Prioritäts- & Privilegierungsebenen umschalten ...
- Teuer ist das Umschalten des Adressraumes:
  - Zustandsumschaltung des Prozessors ist aufwendig → schneller mit HW-Unterstützung (bei IA32 → Task State Segment, beinhaltet auch Wechsel in den Kern)
  - IA32: beim Setzen eines neuen PageDirectory wird Cache gespült → Optimierung: Tabelle beim Umschalten umbauen ,
  - TLB muss geleert werden.

Ablauf der Prozess- bzw. Threadumschaltung:

- Bearbeitung des aktuellen Threads / Prozesses wird unterbrochen,
- Zustand/Kontext (Register, Stack, Adressraum) sichern und vom nächsten TCB laden,
- PCB/TCB sind wichtige BS-Datenstrukturen,
- neuen Prozess/Thread anstossen .



Ablauf:

- meist Umweg über BS,
- bzw. über Scheduler.

Reentrancy :

- = Nebenläufige Nutzung von Code,
- erlaubt Umschalten zu beliebigen Zeiten,
- ab Linux 2.6 Multithreaded Kernel,
- BS Nutzung durch parallele Threads,
- Wichtig: keine thread-Zustandsinformation in globalen Variablen halten.

F.4.1 Umschalttechniken

- Automatisches Umschalten für interaktive BS (Linux & Windows):
  - Umschalten der UL-Threads zu beliebigen Zeiten,
  - aber innerhalb einem definiertem Zeitintervall,
  - Voraussetzung: Interrupts im User-Modus nicht maskierbar.
- **Entzug der CPU mit Hilfe des Hardware-Zeitgebers**:
  - Zeitgeberbaustein (Timer) verwendet IRQ-0 (höchste Interruptpriorität),
  - Erzeugt Interrupts nach konfigurierbarer Periode (typ. 10ms).
  - Interrupt im User-Mode nicht maskierbar.
  - Standard für preemptives Multitasking.
- Manuelles Umschalten für BS mit harten Echtzeitanforderungen
  - einfachste Umschaltform durch direkten Sprung.
  - Umschaltzeitpunkt und -ziel im Quelltext.
  - sehr effizient, aber inflexibel.
- Bedingtes Umschalten:
  - Umschalten abhängig von einer Bedingung, z. B. wenn E/A-Operation notwendig ist.
  - Thread ist dann blockiert und benötigt CPU temporär nicht.

F.4.2 Kurzeitig Umschalten verhindern

- Einfache Lösung:
  - Interrupts kurzfristig sperren.
- Interruptsperre genügt für Multiprozessor-Systemen nicht:
  - Hier erfolgt Interrupt-Signalisierung z.B. über den **lokalen** APIC an einzelne CPUs,
  - Somit werden Interrupt nur auf einer CPU gesperrt,
  - zusätzl. Synchronisierung notwendig.
- Geräte-Treiber:
  - HW erwartet manche Befehlssequenzen innerhalb eines bestimmten Zeitintervalls.
  - Werden diese Zeitvorgaben nicht eingehalten, so kann ein Gerät evtl. nur nach Ein-/Ausschalten wieder verwendet werden.
- Kritische Funktionen im Kern:
  - Umschalten zwischen Prozessen/Threads,
  - Bearbeiten von Scheduling-Queues, ...

F.5 Scheduling (Ablaufplanung)

F.5.1 Überblick

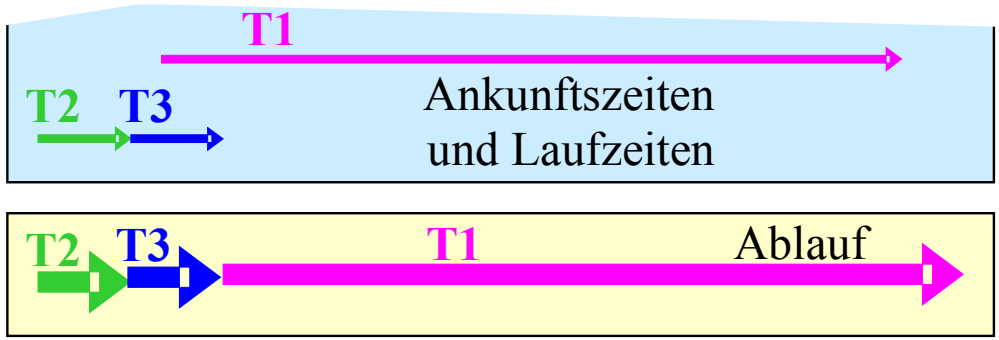
- **Scheduler**:
  - regelt die Zuteilung der CPU:
  - wählt nächsten Thread aus der Bereit-Queue.
- Unterscheidung zw. Ablaufplanung mit und ohne **Verdrängung**:
  - Verdrängung (=Preemption) = Entzug der CPU.
  - → realisiert mit Hilfe von Zeitgeber-Interrupt.
- Scheduler tritt in Aktion wenn ein Thread:
  - startet oder terminiert,
  - freiwillig die CPU freigibt,
  - auf eine E/A-Operation wartet,
  - seine Zeitscheibe 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.

F.5.2 Scheduling-Ziele

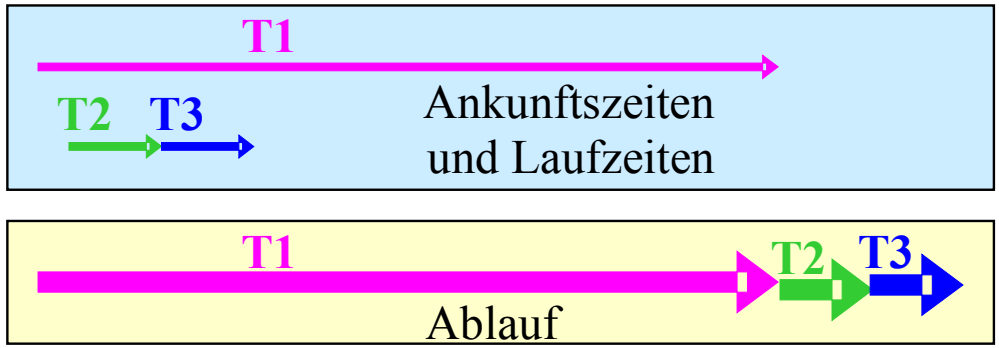
- Effizienz:
  - Hohe CPU-Ausnutzung, wirtschaftliche Auslastung des Gesamtsystems.
- Wartezeit:
  - Wartezeit in der *Bereit*-Liste minimieren.
- Fairness:
  - Jeder Benutzer bzw. Prozess sollte im Mittel den gleichen CPU-Zeitanteil erhalten.
- Durchsatz:
  - #Threads pro Zeiteinheit sollte maximal sein (für Stapelsysteme wichtig).
- Ausführungszeit:
  - Die Zeitspanne vom Jobbeginn bis zum Jobende sollte minimal sein.
  - Sie enthält alle Zeiten in Warteschlangen, der Ausführung (Bedienzeit) und der E/A.
- Antwortzeit:
  - Die Zeit zwischen einer Eingabe und der Übergabe der Antwortdaten an die Ausgabegeräte sollte minimal sein (→ interaktive Systeme),
  - keine grossen Streuungen bei den Antwortzeiten.
- Problem: Overhead beim Umschalten und widersprechende Ziele.

F.5.3 First Come First Served (FCFS)

- Bearbeitung der Threads in Reihenfolge ihrer Ankunft in der Bereitliste.
- Prozessorbesitz bis zum Ende oder bis zur freiwilligen CPU-Aufgabe.
- Günstiger Ablauf:
  - wenig Wartezeit.

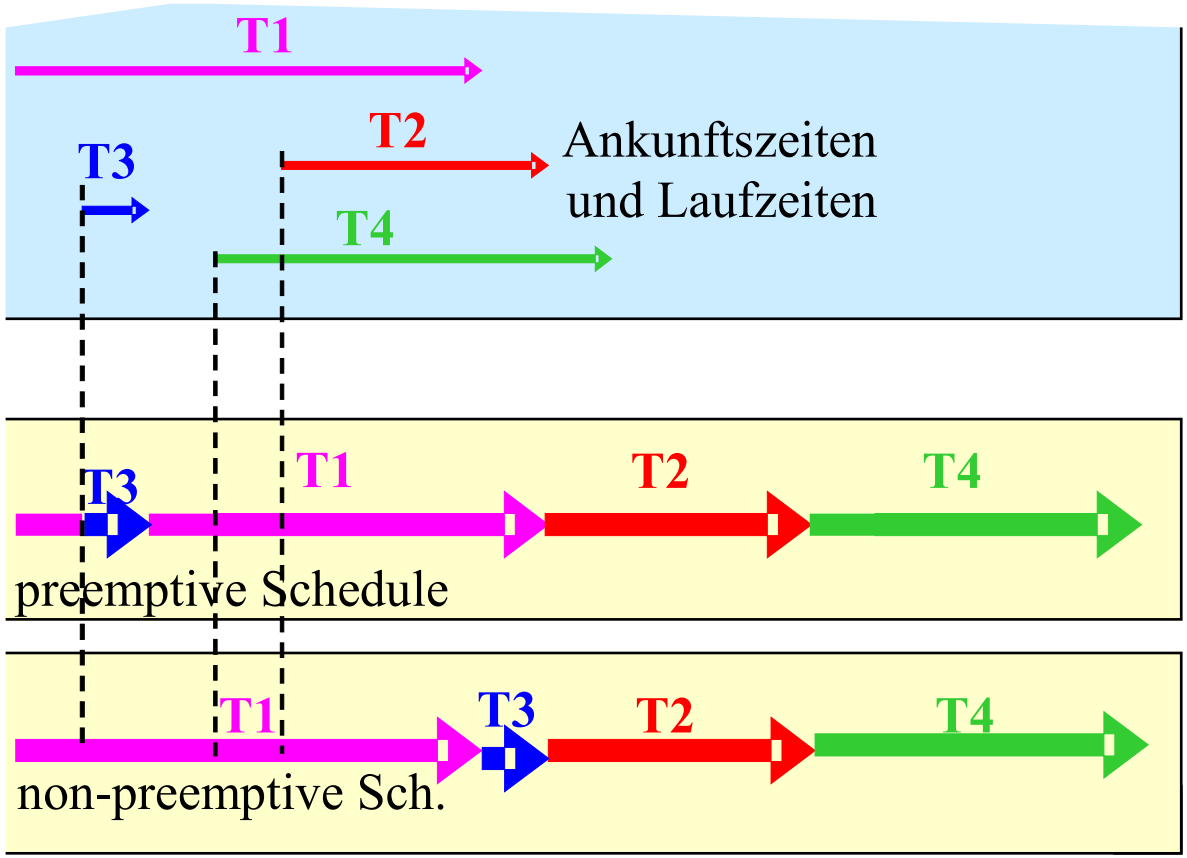


- Ungünstiger Ablauf:
  - schlechte Antwortzeiten,
  - viel Wartezeit.
- Konvoi-Effekt:
  - hinter einem langsamen Thread stauen sich kurze Threads.
  - CPU-lastige Threads halten z.B. I/O-lastige Threads auf.



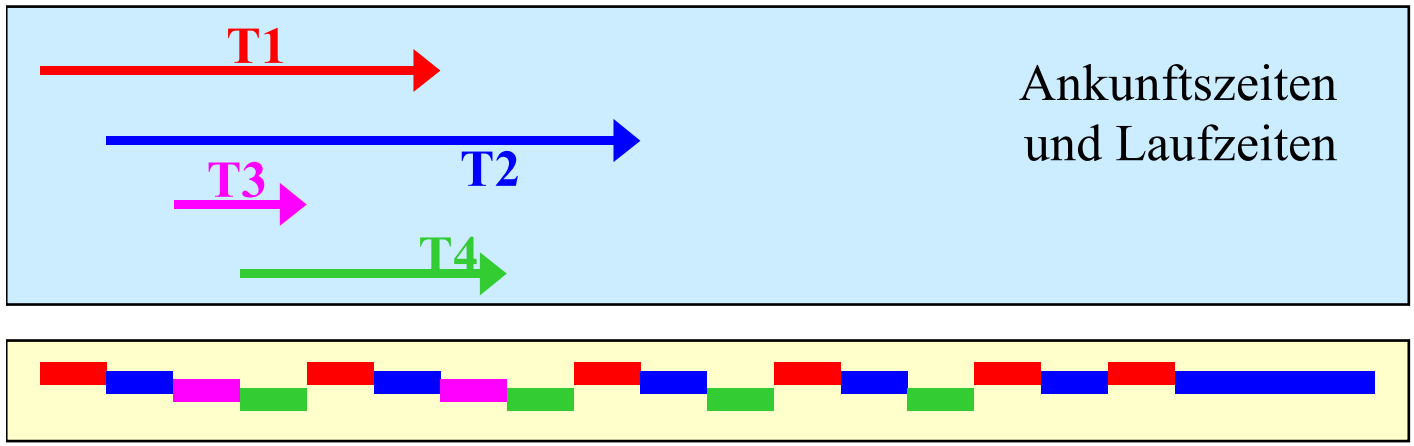
F.5.4 Shortest Job First (SJF)

- Thread mit kürzester Rechenzeit wird als nächster bearbeitet:
  - bevorzugt kurze Threads,
  - u.U. präemptives Umschalten,
  - evt. verhungern lang laufende Threads,
  - Zeitanforderung muss vorab bekannt sein.
- Vorteil:
  - Resourcen frühzeitig freigegeben.
- Beispiel:
  - Restlaufzeiten bewerten,
  - nonpreemptive,
  - preemptive.



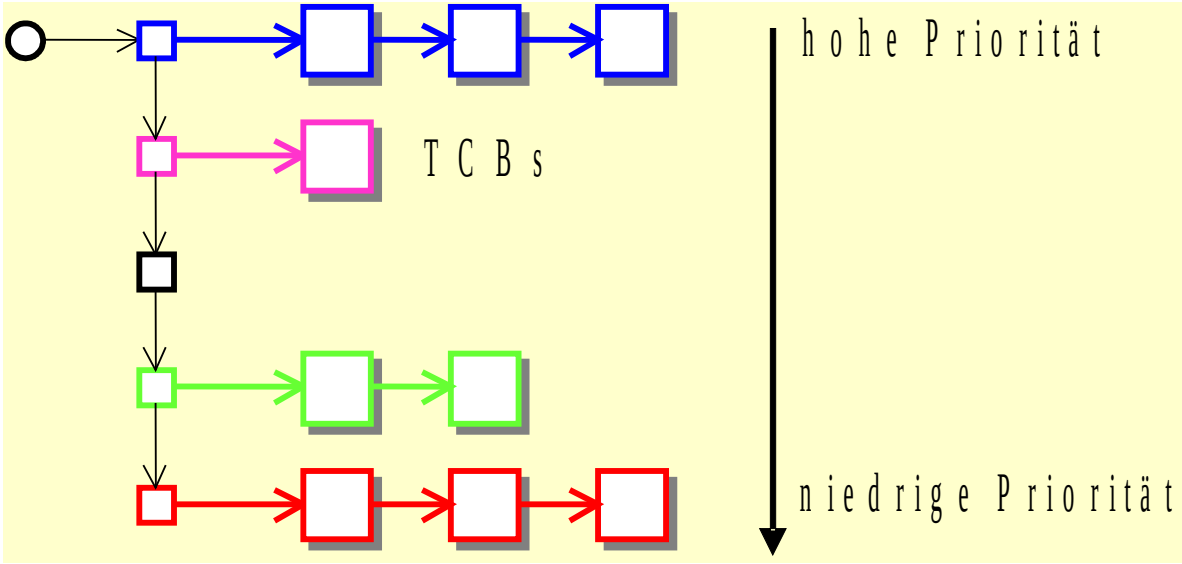
F.5.5 Round Robin (Zeitscheibenverfahren)

- **Ziel**:
  - gleichmäßige Verteilung der CPU keiner soll verhungern.
- Weit verbreitete Strategie (z.B. Linux und NT/XP):
  - Threads in Ankunftsreihenfolge verarbeiten.
  - Nach Ablauf einer vorher festgesetzten Frist (**z.B. 10-100ms**) findet Verdrängung statt.
  - Einreihung des Threads nach Ablauf der **Zeitscheibe (Zeitquantums)** am Ende der Bereit-Queue, sofern er nicht blockiert ist.
- Problem: richtige Wahl der Zeitscheibe:
  - Vernünftiges Verhältnis zwischen Zeitquantum und Kontextwechselzeit notwendig.
  - Grosse Zeitscheiben sparen Kontextwechsel, verursachen aber lange Antwortzeiten.
- Beispiel:

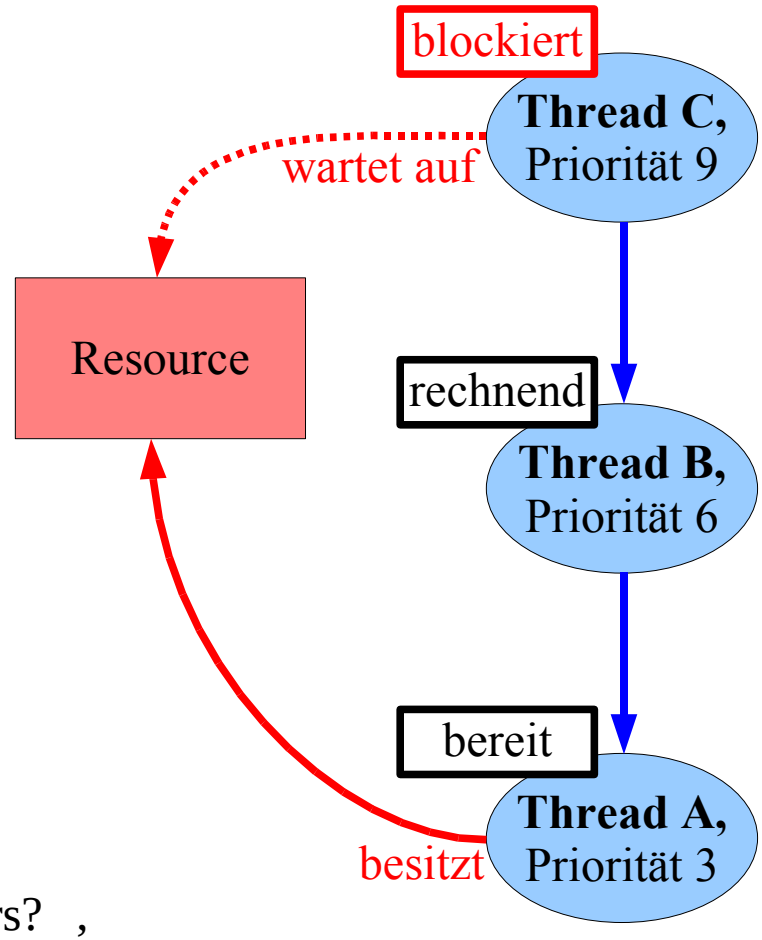


F.5.6 Scheduling mit Prioritäten

- Jeder Thread/TCB erhält eine Prioritätsnummer.
- Threads mit der höchsten Priorität wird aus der *Bereit*-Queue selektiert.
- Implementierung: mehrere *Bereit-Queues*:
  - Vordergrund- und Hintergrundprozesse,
  - oder für jede Priorität eine Warteschlange.



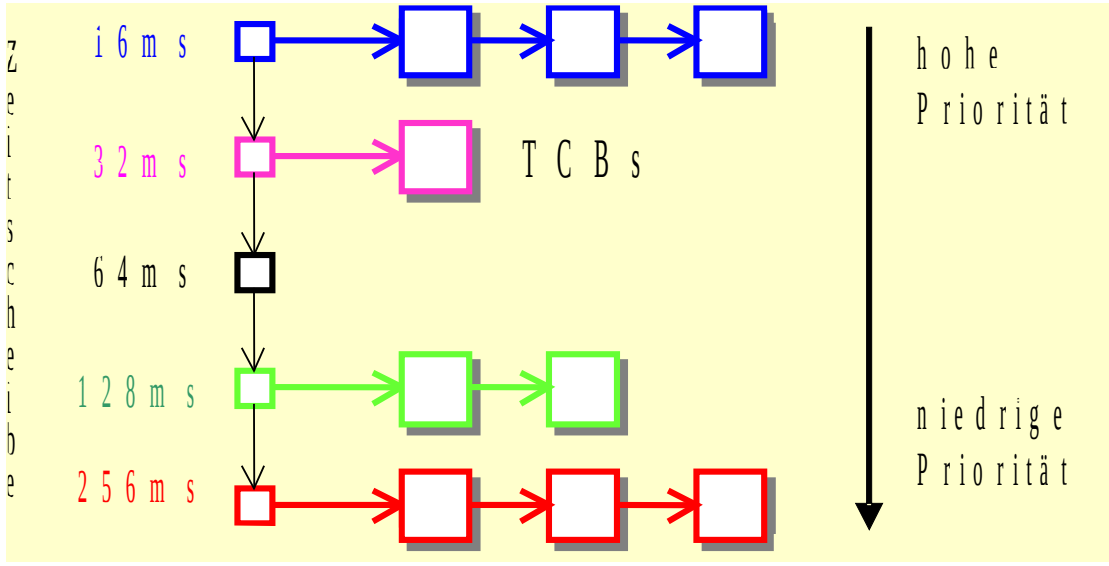
- Problem-1: **Aushungern (Starvation)**
  - Thread mit niedriger Priorität bekommt die CPU nicht zugeteilt, da immer Threads mit höherer Priorität rechenbereit sind.
- Problem-2: **Prioritätsinvertierung**
  - Thread C verhungert , trotz höchster Priorität.
  - Thread B dominiert die CPU,
  - Thread A kommt nicht zum Zuge,
  - Thread A kann Ressource nicht freigeben,
  - Thread C bleibt auf unbestimmte Zeit blockiert.
- Lösung: **Altern (Aging)**
  - Priorität steigt mit Wartezeit (Feedback Scheduling).
  - für Beispiel: Priorität von Thread A steigt langsam an. Sobald sie >6 ist, löst sich das Problem.



- Bem.: Mars Pathfinder hatte Problem-2
  - siehe D. Wilner, Vx-Files: What really happened on Mars? , Keynote at the 18th IEEE Real-Time Systems Symposium, Dec. 1997.

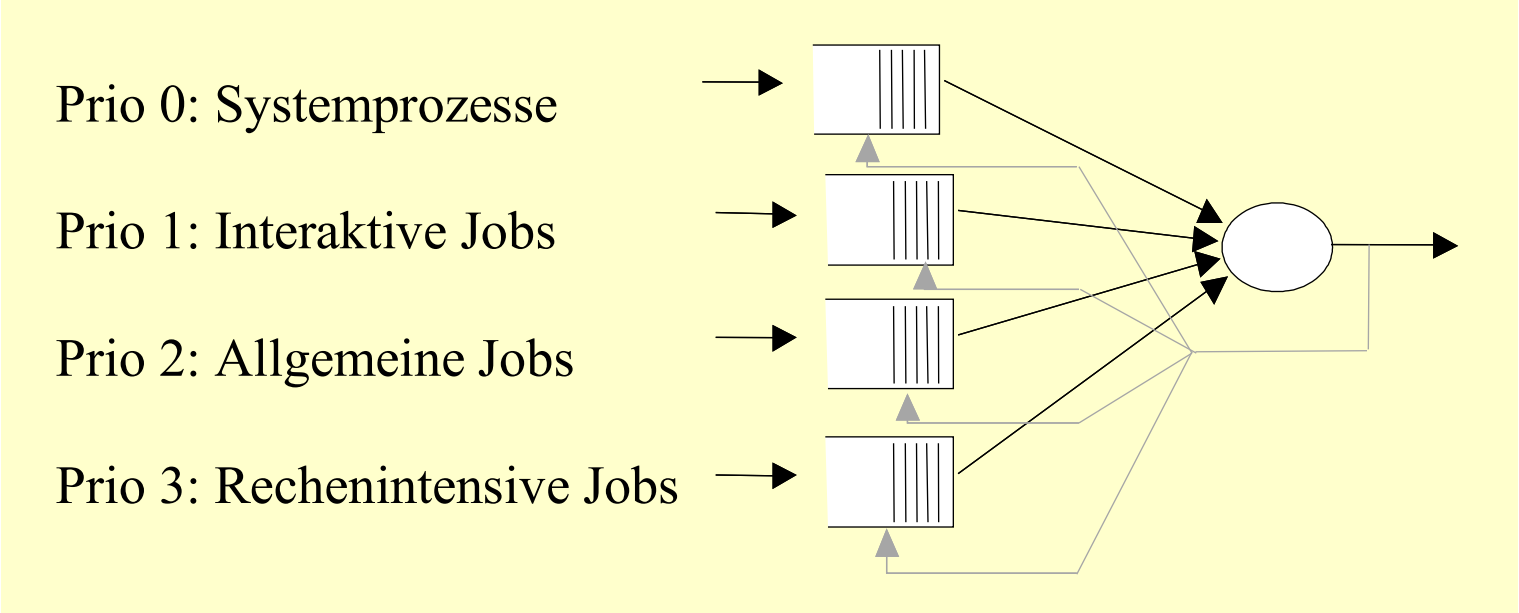
F.5.7 Feedback-Scheduling

- **Idee: dynamische Anpassung der Scheduling-Kriterien**:
  - Verhalten (Rechenintensität bzw. Wartezeit) eines **rechenbereiten** Threads berücksichtigen,
  - Vorgehen: Priorität, Zeitscheibenlänge oder Einsortierung anpassen,
  - Ziele: Verhungern verhindern, Prioritätsinversion auflösen,
    Balance zwischen E/A- und rechenintensiven Threads.
- Beispiel: Multilevel-Round-Robin
  - Threads, welche blockierende E/A-Fkt. aufrufen oder die CPU freiwillig abgeben,
    bleiben in ihrer Warteschlange → E/A-lastige Threads bevorzugen.
  - Verdrängte Threads in Bereit-Liste mit längerer Zeitscheibe einordnen
    (mit niedrigerer Priorität; benötigen mehr Zeit).
- Zwei grundlegende Strategien:
  - wartende Threads hochstufen,
  - rechenintensive Threads herabstufen.



F.5.8 Multilevel Scheduling

- Kombination mehrerer Scheduling-Verfahren.
  - *Bereit-Liste* wird in Sublisten unterteilt.
  - Pro Subliste eigene Scheduling-Strategie.
  - Aufträge jeweils in eine passende Liste einsortieren.
  - Meist sind die Sublisten weiter unterteilt (nach Prioritäten).
- Scheduling zwischen den Sublisten:
  - statische Priorität oder Zeitscheiben.
- Beispiel:



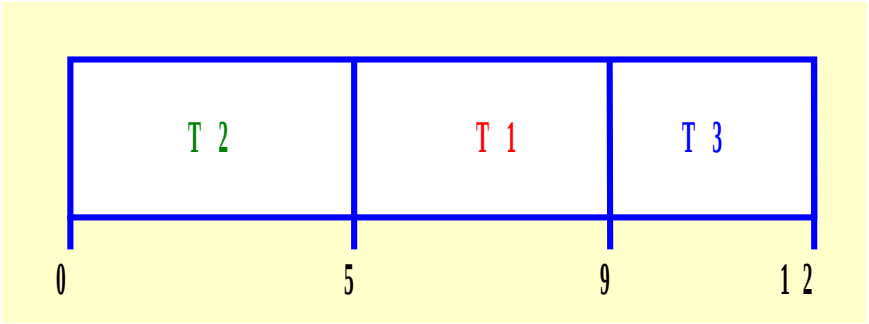
F.5.9 Strategien für Echtzeitsysteme

- Echtzeit-Scheduling:
  - typischerweise für eingebettete Systeme (z.B. AutoSAR),
  - unüblich für Desktop-Betriebssysteme.
- Definierte Zeitschranken einhalten, auch in Fehlersituationen:
  - Für kritische Anwendungen (Kernkraftwerk, Flugzeug ...),
  - Unkritische, aber zeitbezogene Anwendungen (Multimedaströme ...).
- Threads erhalten Sollzeitpunkte (Deadlines).
  - Voraussetzung: Laufzeit vorab bekannt.
  - Unterscheidung bei Sollzeitpunkten zwischen:
    - harte Echtzeit: Verletzung bedeutet Ausfall des Systems (nicht tolerierbar).
    - weiche Echtzeit: Qualitätseinbußen bei Verletzung, aber in Grenzen tolerierbar (MM).
- Offline-Scheduling:
  - Scheduling vor der eigentlichen Ausführung (bei Compilation) zur Vermeidung von Scheduling-Overhead.
  - Vorberechnung eines vollständigen Ausführungsplans in Tabellenform.
  - Einfacher Tabellenzugriff während Ausführung genügt.
  - Damit ist vorab bekannt wann und wo umgeschaltet wird.
  - Sinnvoll für harte Echtzeitanforderung.

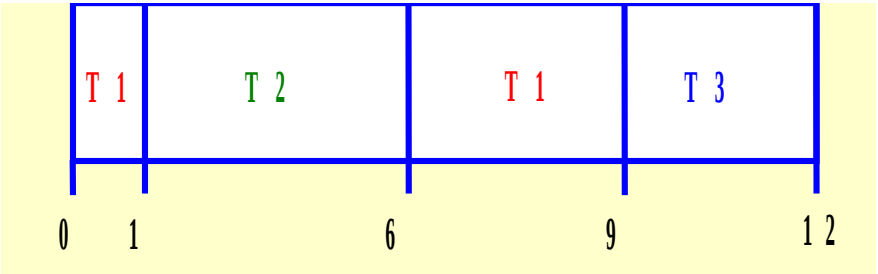
F.5.10 Earliest Deadline First (EDF)

- Verbreitete Echtzeit-Strategie.
  - Threads mit Ausführungsfristen.
  - Thread mit engster Frist wird bevorzugt.

- EDF Beispiel:
  - Laufzeit: t(T1)=4, t(T2)=5, t(T3)=3
  - Fristen: f(T1)=10, f(T2)=7, f(T3)=17
  - non-preemptive; gleiche Ankunftszeit.



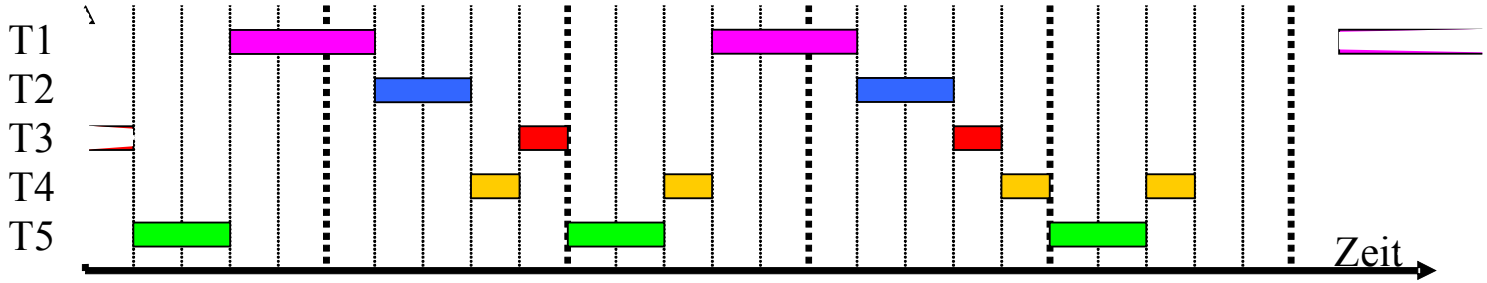
  - preemptive: Ankunftszeiten:
    at(T1)=0, at(T2)=1, at(T3)=2.



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

F.5.11 Rate Monotonic Scheduling (RMS)

- Zur Beschreibung **periodischer** Systeme:
  - hohe Priorität für Aktivitäten mit hoher Frequenz, niedrige Priorität für seltene Tasks.
  - Ende der Periode entspricht der Deadline.
  - z.B. für Multimedia-Ströme.
- Bewertung:
  - Voraussetzung: Threads sind unabhängig voneinander.
  - Minimale Verzögerung für häufig wiederholte Tasks, Zerstückelung niederfreq. Tasks.
  - Situationen, die durch Verzögerung eines höher priorisierten Threads, zu einem gültigen Ablauf führen, sind nicht lösbar (CPU Auslastungsgrenze nur bis ca. 70% möglich).
- Beispiel:
  - Laufzeit: t(P1)=3, t(P2)=2, t(P3)=1, t(P4)=2, t(P5)=2.
  - Periode: p(P1)=13, p(P2)=15, p(P3)=9, p(P4)=16, p(P5)=10.



F.5.12 Mehrprozessor-Aspekte

- Lastausgleich zwischen CPUs durch den Scheduler
- Prozessor-Affinität:
  - Hat Thread *t* zuletzt Prozessor *p* genutzt, so besteht die Chance, dass noch Teile seines Adressraums im Cache/TLB von *p* vorhanden sind,
  - Wahllose CPU-Zuteilung schlecht, TCB wird erweitert um zuletzt benutzte CPU.
- Nachteil: Aufweichung von Prioritäten:
  - evt. zuletzt genutzte CPU belegt und andere frei
  - falls auf alte CPU gewartet wird, so läuft evt. ein Thread mit niedriger Priorität zuerst.

F.5.13 Leerlaufproblem (Idle Task)

- Falls alle Threads warten ist CPU frei u. es läuft ein Leerlauf-Thread:
  - darf nicht anhalten, hat geringste Priorität, muss jederzeit verdrängbar sein.
  - nutzbar für Konsistenzprüfungen, Garbage Collection, Heap Kompaktierung ...
- Busy-Looping mit speziellen Instruktionen vermeidbar:
  - z.B.: HLT (IA32): stoppt CPU;
  - (Timer-)Interrupt weckt CPU wieder auf.

F.5.14 Multi-level Scheduling in Linux

- Alle lauffähigen Prozesse in drei *Bereit*-Queues gespeichert.
- Kernel Thread (V. 2.4) ist nicht verdrängbar (preemptable).
- **SCHED_FIFO**:
  - für Echtzeitprozesse.
  - Queue nur für *root* zugänglich.
  - statische Prioritäten von 1 bis 99.
  - verwendet FCFS innerhalb einer Prioritätsstufe.
  - Verdrängung durch Proz. mit höherer **statischer** Priorität (sonst unlimitierte CPU-Zeit).
- **SCHED_RR**:
  - wie SCHED_FIFO, aber Round Robin für jede Prioritätsstufe.
- **SCHED_OTHER**:
  - für normale Prozesse.
  - statische Priorität: -20 bis + 20 (zu Beginn 0).
  - Gewichtung mit *nice* & *setpriority* (-19 bis +20).
  - zusätzlich **dynamische** Anpassung v. Prioritäten.
- Bem.: keine richtigen Echtzeitfähigkeiten, sondern nur Prioritäten.

Epochen und Prioritäten:

- Linux verwendet keine fortlaufende Zeit, sondern **Epochen**.
  - In einer Epoche hat jeder Prozess ein best. Zeitquantum für die CPU-Nutzung zur Verfügung.
  - Zeitquantum entspricht einigermassen einer zeitweiligen Priorität.
  - Das Zeitquantum wird zu Beginn der Epoche berechnet.
  - Proz. kann CPU pro Epoche mehrfach erhalten, sofern sein Zeitquantum nicht verbraucht ist.
  - Epoche endet, wenn alle Prozesse in der Bereit-Queue ihr Zeitquantum verbraucht haben.
  - Blockierte Prozesse werden hierbei nicht beachtet!
- Berechnung des **Zeitquantums (ZQ)**:
  - Wenn Prozess in *Bereit*-Queue eingefügt wird und zu Beginn einer neuen Epoche.
  - Basis sind 200ms (20 clocks ticks).
  - falls Zeitquantum (ZQ) verbraucht: Quantum = Basis ,
  - falls nicht verbraucht: Quantum = Basis + $\mathbf{ZQ_{alt}/2}$
    → I/O-lastige Proz. verbrauchen ihre Zeitscheibe nicht und erhalten dafür einen **Bonus**.
- Prioritäten im Prozesskontrollblock:
  - *rt_priority*: statische Prio. von Echtzeitprozessen.
  - *priority*: Basis-Zeitquantum (inkl. *nice*-Gewichtung).
  - *counter*: Anzahl verbleibender CPU-Ticks im Zeitquantum in der akt. Epoche.

F.5.15 Prozessbewertung:

- Scheduler selektiert Proz. aus Bereit-Queue mit Hilfe der *goodness*-Fkt.
- Rückgabewert der *goodness*-Funktion:

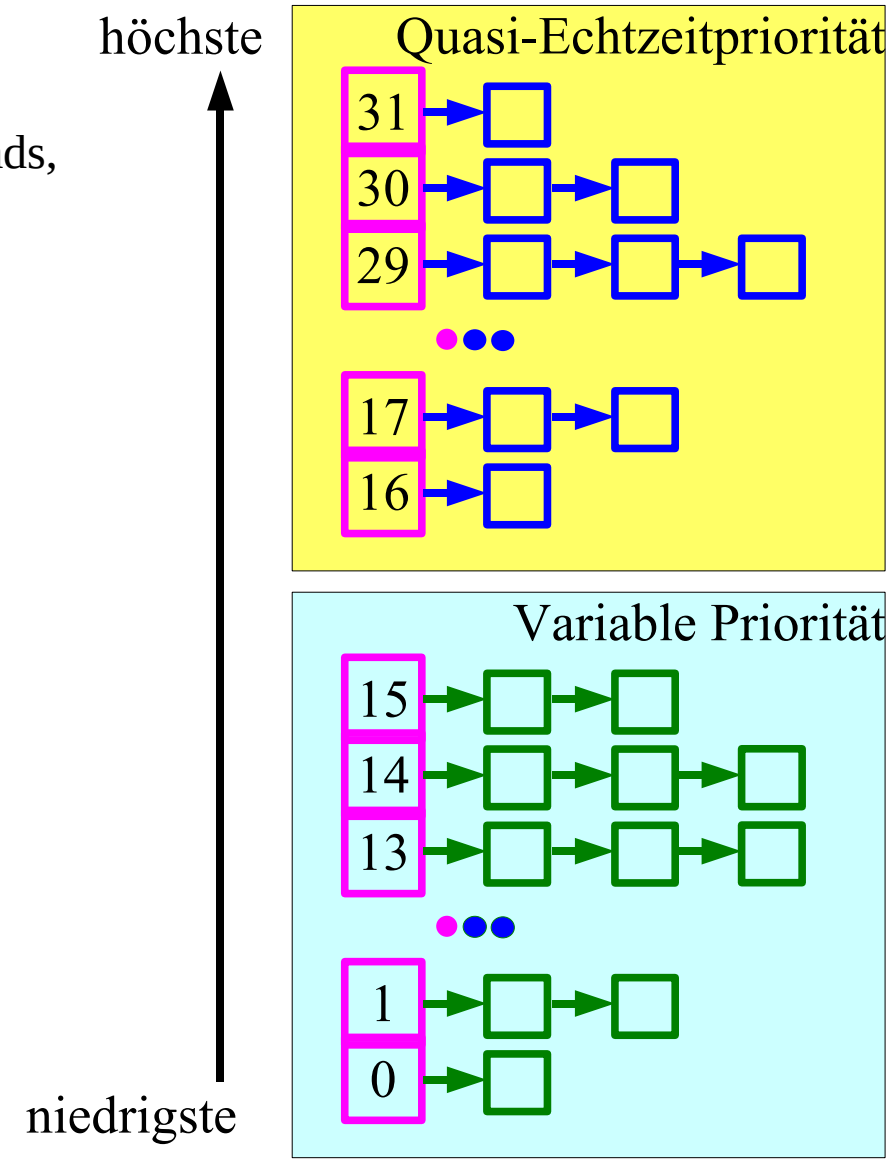
-1	Proz. hat CPU freiwillig abgegeben.
0	Proz. hat Quantum voll ausgenutzt.
1-999	normaler Proz. mit noch (teilweise) unverbrauchtem Quantum.
> 999	Echtzeitprozess.

- Bem.: Je größer der Rückgabewert, desto besser ist die Bewertung.
- Berechnung:

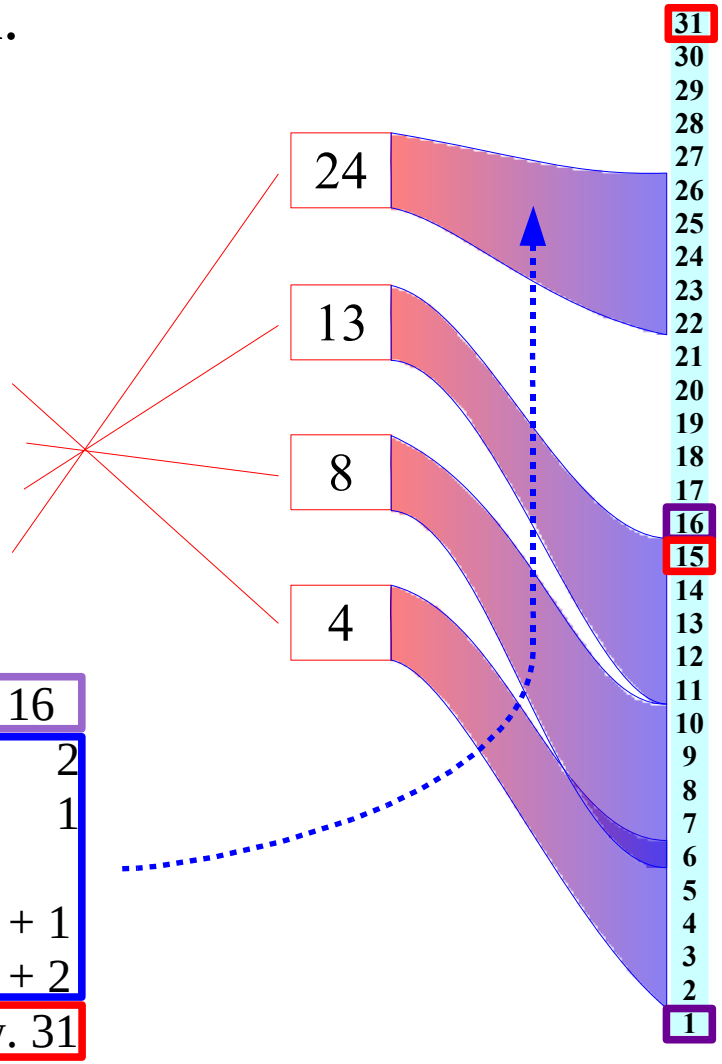
```
if (p->policy != SCHED_OTHER)          // Echtzeitprozess (SCHED_FIFO und SCHED_RR)
    return 1000 + p->rt_priority;      // Echtzeitpriorität zurückgeben

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

if (p->mm == prev->mm)                 // gleicher Adr.raum?
    return p->counter + p->priority + 1;  // dyn. Prio. + Bonus (+ 1)
return p->counter + p->priority;       // ansonsten: dyn. Prio.
```

F.5.16 Scheduling in Windows

- Schedulerstrategie:
  - Scheduling betrachtet ausschließlich Threads,
  - angelegt sind 32 Prioritätsklassen,
  - verdrängend/preemptive,
  - Zeitscheiben.
- 32 Prioritätsklassen:
  - Pro Klasse eine FiFo-Warteschlange,
  - Echtzeitpriorität: 16-31 (statisch)
  - variable Priorität: 1-15 (dynamisch)
  - Leerlaufthread: 0
- Zeitscheiben:
  - round-robin innerhalb einer Queue,
  - Workstation-Zeitsch.: 20-30 ms.
  - Server-Zeitscheibe: 150-180 ms.

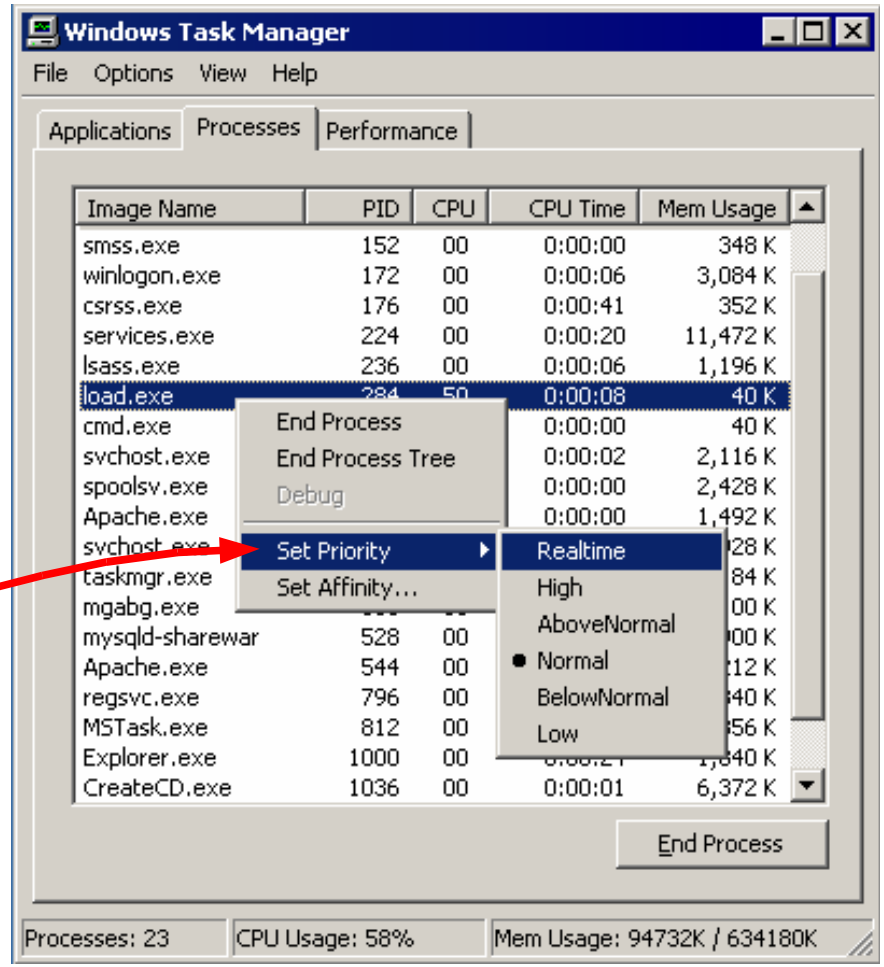


- Richtige Echtzeitfähigkeiten (Deadlines) nicht vorhanden, sondern nur höhere Prioritäten.
- Prioritätsberechnung:
  - aus relativem **Thread-Prioritätslevel,**
  - und aus **Prozessklasse.**
- **Prozessklassen (PKL)**:
  - IDLE: Priorität 4
  - NORMAL: Priorität 8
  - HIGH: Priorität 13
  - REALTIME: 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



- Präemptiver Kern; User-Threads können auch KL-Tread verdrängen.

- Prioritätserhöhung für GUI-Threads:
  - Die mit einem Fenster arbeiten und auf Benutzereingaben oder Fensternachrichten warten.
  - Auf Priorität 14 mit Verdopplung der Zeitscheibe.
- Prioritätserhöhung nach Blockierung:
  - Priorität beenden (1-8), wenn E/A-Auftrag beendet ist → *Priority Boost*.
  - Nach Priority-Boost, wird nach jedem Ablauf einer Zeitscheibe, die Priorität wieder um 1 erniedrigt (bis 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.
- Die Prozessklasse ist über den Taskmanager anpassbar



F.6 Zusammenfassung

- Prozess:
  - Adressraum + Thread(s)
  - Unix: Prozesshierarchie und alte Betriebssysteme ohne Threads.
  - Windows XP: Prozesse mit mindestens einem Thread, keine Hierarchie.
- Threads:
  - Threads teilen sich den Adressraum eines Prozesses,
  - User-, Kernel- u. Hybride Implementierungen,
  - Vorteil: weniger Kontextwechsel.
- Dispatcher:
  - Implementiert Kontextwechsel/Umschaltvorgang.
- Scheduler:
  - regelt CPU Zuteilung,
  - CPU- und I/O-lastige Threads,
  - Round Robin: präemptiv, 10-100ms Zeitscheibe, weit verbreitet.
  - Prioritäten können zu Verhungern & Prioritätsinversion führen → Aging.
  - Echtzeit-Scheduling berücksichtigt Sollzeitpunkte/Deadlines.
  - Unix Scheduler: hohe CPU Nutzung führt zu niedriger Priorität
  - Windows XP: bei Abschluss eines E/As oder bei langem Warten gibt es einen Priority Boost.