Kapitel 3
Prozeßverwaltung

- Aspekte von Prozessen
- Prozesszustände
- Prozesswechsel
- Prozesssysteme
- Ablaufplanung (Scheduling)
- Fallstudie 1: Unix System V
- Fallstudie 2: Windows NT
- Fallstudie 3: BS 2000

Prozesse - Grundlagen

- **Grundlegendes Konzept**
  - *ein Prozess ist ein in Ausführung befindliches Programm*

- **Analogie des kulinarisch interessierten Informatikers, der einen Kuchen backen möchte**
  - Programm : Backrezept
  - CPU : Informatiker/in
  - Ressourcen/Betriebsmittel : Ofen, Rührgerät, etc.
  - Eingabedaten : Zutaten des Kuchens (Mehl, Eier, Zucker, etc.)

  - Prozess : Aktivität, dass der backende Informatiker das Rezept liest, die Zutaten holt und unter Verwendung der Küchengeräte den Kuchen bäckt.

Prozesse - Grundlagen

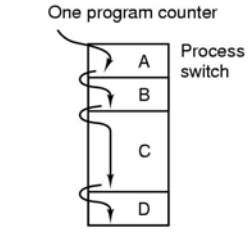
- **Analogie des kulinarisch interessierten Informatikers, der einen Kuchen backen möchte (Forts.)**
  - Bekannter des/r Informatikers/in schneidet sich in die Hand und braucht Unterstützung beim Verbinden
  - Informatiker/in notiert sich, an welcher Stelle des Rezepts er/sie sich befindet, ➔ Zustand des aktuellen Prozesses wird gespeichert
  - Holt das Erste-Hilfe-Buch und beginnt den Anweisungen darin zu folgen ➔ Prozesswechsel
  - CPU (Informatiker) wechselt von einem Prozess (Backen) zu einen Prozess höherer Priorität (Erste-Hilfe)
  - Jedem Prozess liegt ein unterschiedliches Programm zugrunde (Backrezept bzw. Erste-Hilfe-Buch)
  - Nach der Erste-Hilfe-Behandlung kann der Backvorgang an derselben Stelle fortgesetzt werden ➔ Prozesswechsel
  - ...

Prozesse - Grundlagen

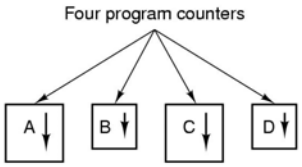
- **Vorgehen bei der Ausführung von Programmen**
  - Betriebssystem erzeugt je einen virtuellen Prozessor für jede Ausführung eines einzelnen Programms
  - Virtuelle Prozessoren werden über Prozesstabelle verwaltet
    - CPU Registerwerte
    - Speicherbelegung
    - Offene Dateien
    - Berechtigungen
    - Etc.
- **Prozess : ein in Ausführung befindliches Programm**
  - Betriebssystem und Hardware verwalten Unabhängigkeit der Prozesse
    - Keine unbeabsichtigte gegenseitige Beeinflussung
    - Vollständig getrennte und unabhängige Adressräume
  - Nebenläufigkeits-Transparenz

Prozesse-Wechsel

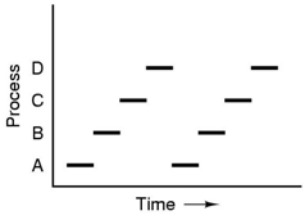
- **Prozess-Wechsel : abwechselnde Zuteilung der CPU an verschiedene Prozesse**
  - Sicherung des CPU Kontexts :
    Registerwerte, Programmzähler, Stack-Pointer, ...
  - Sicherung der Register der Memory Management Unit (MMU)
  - Auslagerung und Einlagerung von Prozessen zwischen Hauptspeicher und Festplatte



Multiprogrammierung



Individuelle sequentielle Prozesse



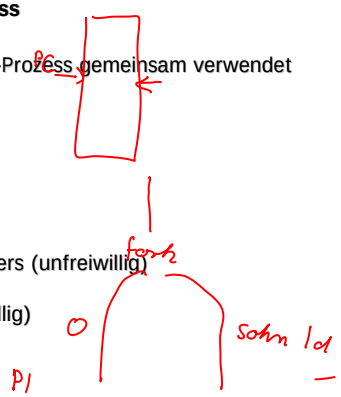
ein aktives Programm zu einem Zeitpunkt

Prozess-Erzeugung und -Beendigung

- **In sehr einfachen Systemen können alle Prozesse bei Systemstart vorliegen**
  - Z.B. Steuerung eines Mikrowellenherds
- **I. Allg. werden Prozesse dynamisch erzeugt und beendet**
- **Prozesse werden erzeugt bei**
  - Initialisierung des Systems
  - Systemaufruf zum Erzeugen eines Prozesses durch einen anderen Prozess
  - Benutzeranfrage, einen neuen Prozess zu erzeugen
  - Initiierung einer Stapelverarbeitung (Batch-Job)
- **Deamon**
  - Systemprozesse, die im Hintergrund arbeiten
  - Z.B. zur Verwaltung von Emails, Webseiten, Druckaufträgen, etc. (siehe UNIX)

Prozess-Erzeugung und –Beendigung

- **Technische Prozess-Erzeugung**
  - UNIX: **fork-Kommando** erzeugt identische Kopie des aufrufenden Prozesses
  - Windows: Win32-Funktionsaufruf **CreateProcess**
  - Getrennte Adressräume werden erzeugt
  - Einige Ressourcen können von Kind- und Eltern-Prozess gemeinsam verwendet werden, z.B. geöffnete Dateien



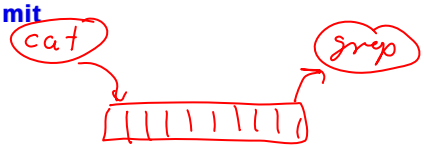
- **Prozesse werden beendet durch**
  - Normales Beenden (freiwillig)
  - Beenden aufgrund eines Fehlers (freiwillig)
    - z.B. falsche Daten, Eingabe-Datei nicht existent
  - Beenden aufgrund eines schwerwiegenden Fehlers (unfreiwillig)
    - Z.B. Adressraumverletzung, Division durch 0
  - Beenden durch einen anderen Prozess (unfreiwillig)
    - UNIX: kill-Kommando
    - Win32: TerminateProcess

Prozess-Hierarchien

- **Zusammenhänge zwischen Eltern- und Kind-Prozess**
  - Jeder Prozess hat genau EIN Elternteil
- **Prozess-Baum (vgl. Baum-Struktur der Datei-Systeme)**
- **Z.B. Initialisierung von UNIX**
  - Spezieller Prozess init ist Teil des Boot-Image
  - init –Prozess liest aus einer Datei die Anzahl benötigter Terminals und erzeugt pro Terminal einen Prozess
  - Jeder Terminal-Prozess wartet, bis sich jemand am System anmeldet
  - Nach erfolgreicher Anmeldung führt der Login-Prozess eine Shell aus, um Kommando-Eingaben annehmen zu können
  - Die Kommando-Eingaben (bzw. gestarteten Programme) können weitere Prozesse starten
  - Etc.
  - Prozessbaum mit init als Wurzelknoten
  - Windows kennt dieses Konzept der Prozess-Hierarchie nicht

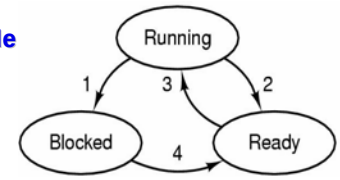
Prozess-Zustände

- **Jeder Prozess ist eine eigene Einheit mit**
  - eigenem internem Befehlszähler
  - internem Zustand



- **Kommunikation zwischen Prozessen**
  - Beispiel: cat file1 file2 file3 | grep tree
  - Erster Prozess P1 fügt die drei Dateien aneinander und übergibt das Ergebnis an den Prozess P2, der grep ausführt
  - Möglicherweise ist P2 ausführungsbereit, aber es sind noch keine Eingabe-Daten vorhanden ➔ P2 muss blockiert werden

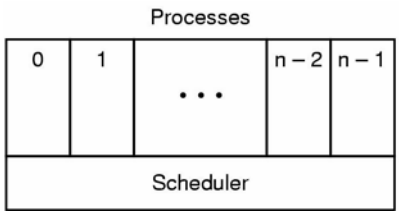
- **Wesentliche Zustände eines Prozesses**
  - Rechnend
  - Rechenbereit
  - Blockiert



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Prozess-Zustände

- **Prozess-Scheduler**
  - Teil des Betriebssystems
  - Bestimmt, welcher Prozess wie lange die CPU zugeteilt bekommt (Details später in diesem Kapitel)
  - Aufgaben
    - Unterbrechungsbehandlung
    - Starten und Stoppen von Prozessen
  - Vereinfachtes Modell



Implementierung von Prozessen

- **Prozesstabelle**
  - Enthält pro Prozess einen Eintrag, den sogenannten **Prozesskontrollblock** (PCB – Process Control Block, Prozessleitblock)
  - Informationen über den Zustand des Prozesses, Befehlszähler, Kellerzeiger, Speicherbelegung, Zustand der geöffneten Dateien, etc.
  - Informationen, die gespeichert werden müssen, wenn der Prozess vom Zustand `rechnend` in die Zustände `rechenbereit` oder `blockiert` übergeht
  - Damit kann der Prozess zu einem späteren Zeitpunkt weiterlaufen, als wäre er nie unterbrochen worden

Implementierung von Prozessen

- **Einige Felder eines Eintrags in der Prozesstabelle**

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

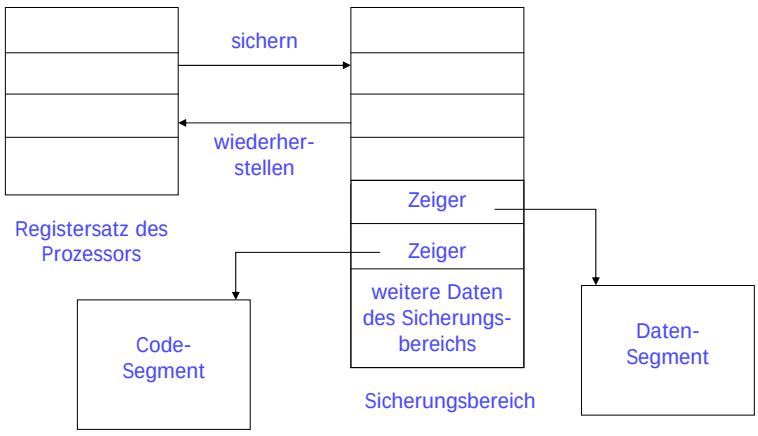
Der Kontext eines Prozesses

- **Inhalt aller Prozessor-Register, darunter insbesondere**
  - den aktuellen Befehlszähler (program counter)
  - den Stapelzeiger (stack pointer)
  - das Prozessor-Status-Wort (PSW)
- **Inhalt eventuell vorhandener FPU-Register**
- **Inhalt eines Teils von eventuell vorhandenen MMU-Registern bzw. von zugehörigen Adreßübersetzungstabellen**

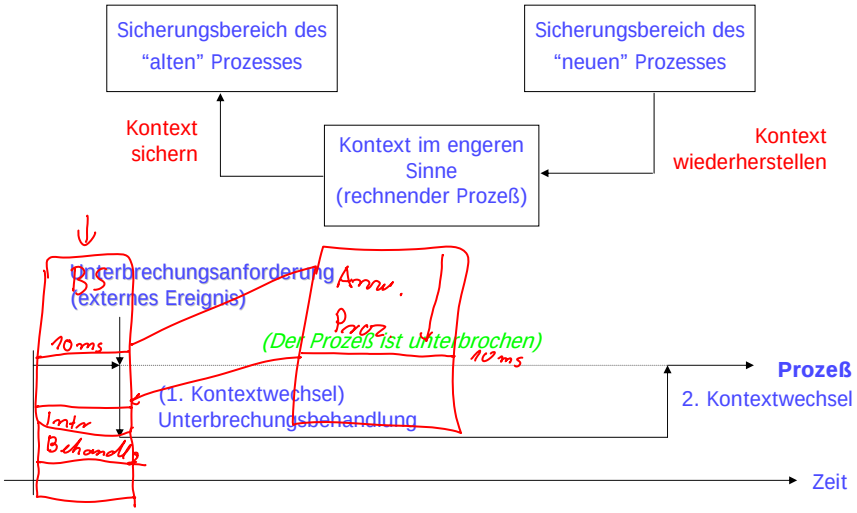
- **sämtliche vom Prozeß belegte Speichersegmente für**
  - prozeßeigene Daten, darunter den prozeßeigenen Stapel (Stack)
- **mit anderen Prozessen gemeinsam benutzte Speichersegmente für**
  - den Programm-Code und
  - eventuell auch für prozeßübergreifend gemeinsam genutzte Daten

- **betriebssysteminterne, aber prozeßspezifische Datenstrukturen, darunter**
  - Prozeßleitblock
  - Prozeßkopf
  - Daten über prozeßeigene Betriebsmittel wie Liste belegter Speichersegmente und Liste geöffneter Dateien usw.

Prozeßwechsel

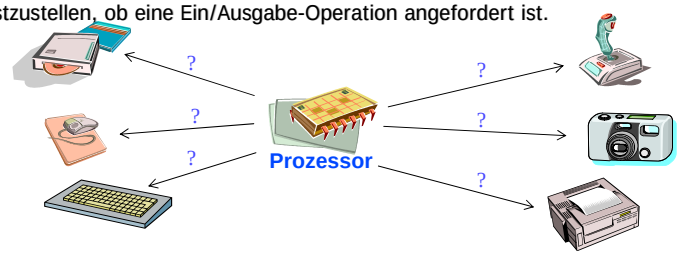


Kontextwechsel



Exkurs Rechnerarchitektur: Unterbrechungen im Prozessor (Interrupts)

- **Realisierung durch Polling (wiederholtes Abfragen)**
  - „Aktives Warten“ (busy waiting)
  - Prozessor überprüft in periodischen Abständen die Status-Bits der Komponenten, um festzustellen, ob eine Ein/Ausgabe-Operation angefordert ist.



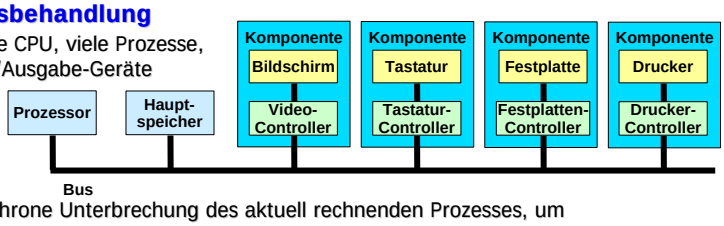
  - 100 % Auslastung, aber nur mit dem einen Prozess, der meistens vergeblich abfragt!
  - Nachteil: Prozessorzeit wird verschwendet, da der Prozessor deutlich schneller ist als die angeschlossenen Ein/Ausgabe-Komponenten und die Abfrage häufig nicht zu einer Ein/Ausgabe-Operation führt.
  - Nachteil: schlechte Reaktionszeit bei seltener Abfrage.

Exkurs Rechnerarchitektur: Unterbrechungen im Prozessor (Interrupts)

- **Beispiel: Matrixdrucker druckt ca. 100 Zeichen pro Sekunde**
  - benötigt also alle 10 ms die Übergabe eines Zeichens
  - Datenübergabe durch kleines Unterprogramm (< 100 Maschinenbefehle pro Aufruf)
  - Abarbeitung eines zweitaktigen Maschinenbefehls dauert 10 ns bei 200 MHz
  - CPU/Belastung pro Zeichen: 0,001 ms aktiv; 9,999 ms oder 99,99 % der Rechenzeit wartet der Prozessor auf den Drucker!
- **Realisierung durch Polling (wiederholtes Abfragen)**
  - CPU zu 100% ausgelastet!
- **Abhilfe: externe Unterbrechungsanforderungen (interrupt requests):**
  - Prozessor macht „etwas anderes" (z.B. Abarbeitung eines anderen Prozesses)
  - wird durch einen speziellen Hardwaremechanismus benachrichtigt, wenn das Gerät zur Übernahme neuer Daten bereit ist (Interrupt-Signal am Unterbrechungseingang des Prozessors)
  - Weitere Beispiele: Eingabe eines Zeichens an der Tastatur oder wenn der Festplattenkopf die gewünschte Position erreicht hat

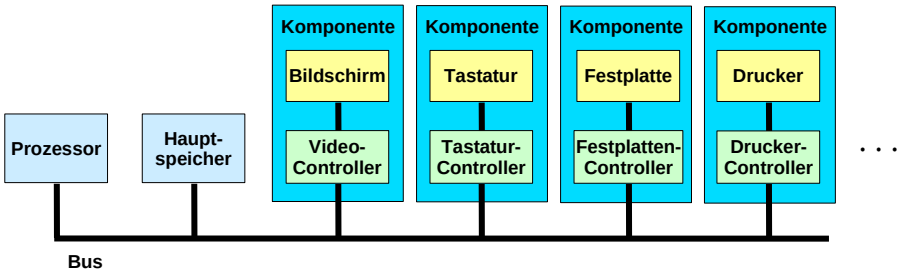
Implementierung von Prozessen

- **Unterbrechungsbehandlung**
  - Situation: eine CPU, viele Prozesse, mehrere Ein-/Ausgabe-Geräte



  - Zweck: asynchrone Unterbrechung des aktuell rechnenden Prozesses, um bereitstehende Ein-/Ausgabe Anforderungen zu bearbeiten
  - Jeder Klasse von Ein-/Ausgabe-Geräten ist eine Speicherstelle namens Interrupt-Vektor zugeordnet
  - Interrupt-Vektor enthält Adressen der Routinen zur Unterbrechungsbehandlung
  - Beispiel: Benutzerprozess 4 läuft, während ein Festplatten-Interrupt auftritt
    - Befehlszähler, Programmstatuswort und Registerinhalte werden durch die Unterbrechungs-Hardware auf den aktuellen Keller gespeichert
    - CPU springt zu der Adresse, die im Festplatten-Interrupt-Vektor angegeben ist
    - Alles weitere wird durch die Software, d.h. durch die Routine zur Unterbrechungsbehandlung ausgeführt

Exkurs Rechnerarchitektur: Unterbrechungen im Prozessor (Interrupts)



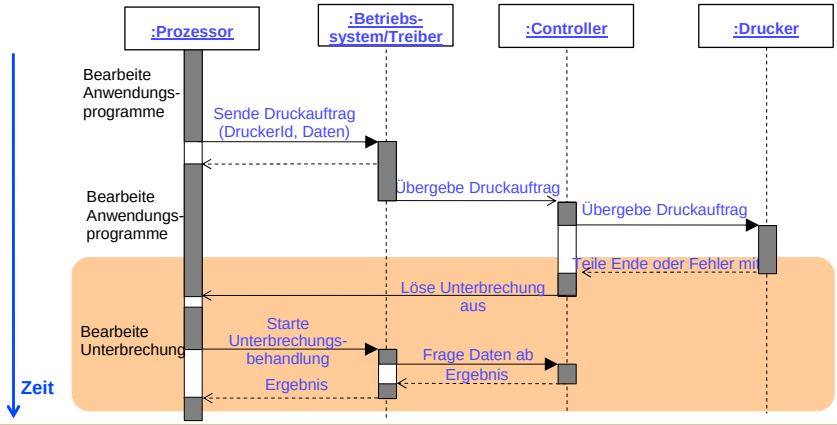
- **Es muss Mechanismen und Techniken geben für die effiziente Kommunikation und Interaktion zwischen dem Prozessor und den weiteren Komponenten**
  - Polling
  - Unterbrechungen (Interrupts)

Exkurs Rechnerarchitektur: Unterbrechungen im Prozessor (Interrupts)

- **Prozessor unterbricht daraufhin die Befehlsfolge und ruft statt dessen eine spezielle Unterbrechungsprozedur, ISR = Interrupt Service Routine) wie ein Unterprogramm ohne Übergabe von Argumenten auf**
  - Unterbrechungsprozedur hängt nur vom anfordernden Peripheriegerät, nicht aber von der unterbrochenen Befehlsfolge ab
  - Dabei wechselt der Prozessor gegebenenfalls aus der Normal- in die Systembetriebsart
  - auf dem Stapel (stack) werden die Rückkehradresse und das Prozessorstatuswort (für die spätere Wiederherstellung des ursprünglichen Prozessorzustands) abgelegt
  - Völlig analog zum eingangs behandelten Software-Interrupt-Befehl
- **Unterbrechungsprozedur wird ausgeführt**
  - z.B. die wenigen Befehle zum Übergeben eines weiteren Zeichens
- **Rückkehr zur (gerade unterbrochenen) Programmabarbeitung**
  - RTI (return from Interrupt)
  - Der Prozessor wechselt hierbei ggf. wieder zurück in die Normalbetriebsart

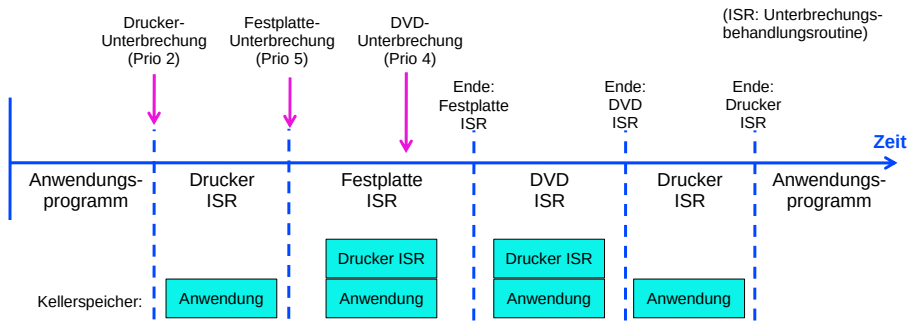
Exkurs Rechnerarchitektur: Unterbrechungen im Prozessor (Interrupts)

- **Unterbrechungen in einem Rechnersystem sind eine wesentliche Technik für die Kommunikation zwischen Prozessor und angeschlossenen Komponenten.**

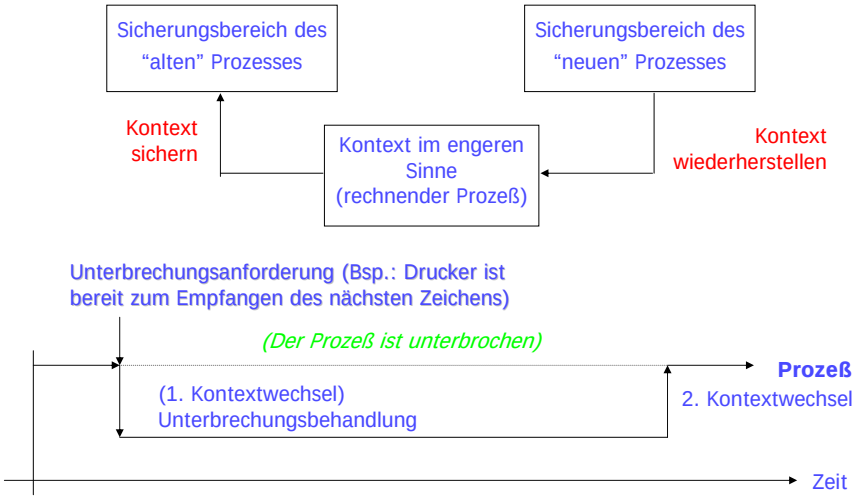


Exkurs Rechnerarchitektur: Unterbrechungen im Prozessor (Interrupts)

- **Priorisierung von Unterbrechungen.**
  - Bei gleichzeitigem Auftreten mehrerer Unterbrechungen wird die Unterbrechung mit der höchsten Priorität ausgewählt.
    - Hohe Priorität: schnelle Komponenten, z.B. Festplatte.
    - Niedrige Priorität: langsame Komponenten, z.B. Tastatur.
  - Es ist möglich, dass eine Unterbrechung hoher Priorität eine augenblicklich bearbeitete Unterbrechungsbehandlung zu einer Unterbrechung niedrigerer Priorität unterbricht.

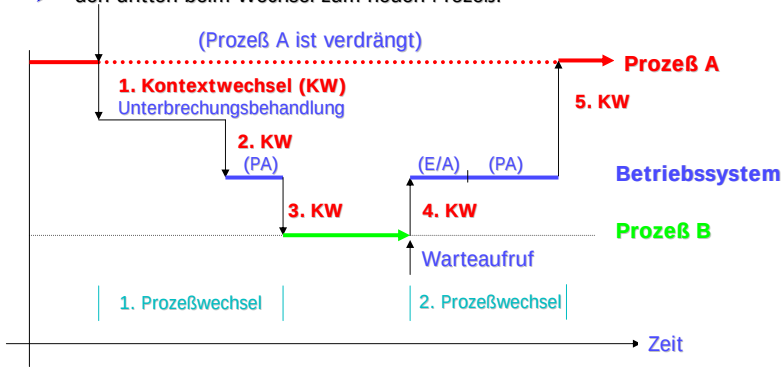


Kontextwechsel



Prozesswechsel

- **Prozesswechsel schließt genaugenommen sogar drei Kontextwechsel ein:**
  - den zur Durchführung der Unterbrechungsbearbeitung,
  - einen zweiten beim Übergang in den Betriebssystem-Kontext und
  - den dritten beim Wechsel zum neuen Prozeß.



Prozeß als Betriebsmittelbesitzer

- **Das Betriebssystem muß über die Ausstattung eines Prozesses an Betriebsmitteln Buch führen**
  - Speichersegmente für Daten
  - exklusiv belegte Peripheriegeräte wie serielle Schnittstellen, Drucker und Tastatur sowie die mit diesen verbundenen Schnittstellen-Bausteine
  - exklusiv geöffnete Dateien
  - Mit anderen Prozessen gemeinsam genutzte Betriebsmittel wie z. B. Kommunikations- und Synchronisationshilfsmittel wie Semaphore, Briefkästen, gemeinsame Speicher, aber auch den Bildschirm
- **Bedeutung der Betriebsmittelverwaltung**
  - Verhinderung, daß exklusiv belegte Betriebsmittel ein zweites Mal vergeben werden
  - Bei erzwungenem Abbruch oder aus anderen Gründen (z.B. Verklemmungs-verhinderung) Entziehen der Betriebsmittel

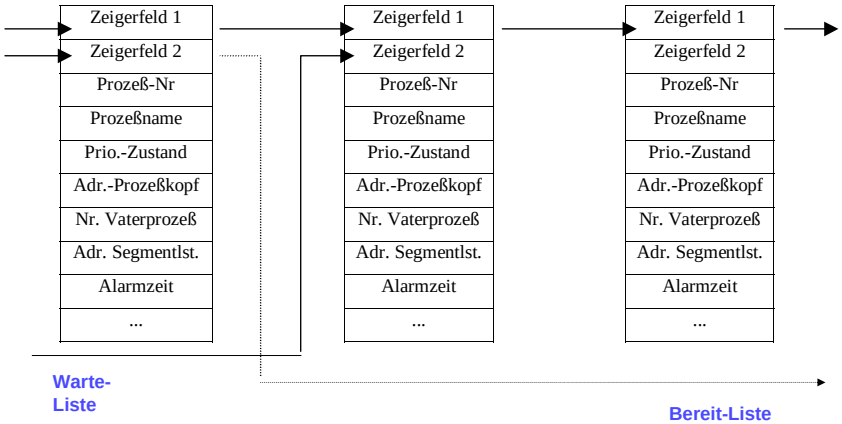
Prozessleitblock (PCB) / Prozesskopf

- **Repräsentierung von Prozessen durch Listen von Datenstrukturen**
- **Prozeßbezogene Teile davon gehören ebenfalls zum erweiterten Kontext**
  - **Prozessleitblock**: vom Betriebssystem zur Prozeßverwaltung benötigte Daten (muß immer im Hauptspeicher verbleiben!)
  - **Prozesskopf**: Daten, die das Betriebssystem dem Prozeß zur Verfügung stellt, weil er sie für seinen korrekten Ablauf benötigt (kann zusammen mit dem Prozeß ausgelagert werden)
- **Z.T. enthält der PCB auch Platz für den zu sichernden Kontext**
- **Anordnung der Prozessleitblöcke in einer festen Tabelle oder in einer verzeigerten Liste**
- **Evtl. weitere Zeigerfelder für das Einhängen des PCB in andere Listen (Bereit-, Wartelisten) des Betriebssystems**

Prozeßleitblock

- Prozeß-Nummer
- Prozeß-Name (alphanumerisch)
- Zeiger auf den Prozeßkopf
- (Arbeits-) Zustand
- Liste der Ereignisse, auf die der Prozeß wartet
- Alarmzeit für Weckaufruf
- Priorität
- CPU-Zeitverbrauch und andere Angaben für Zwecke der Ablaufplanung
- Nummer des Vaters des Prozesses
- Liste der Sohnprozesse
- Nummer der Prozessgruppe
- Liste der prozesseigenen Speichersegmente (Basisadressen und Segmentlängen)
- Liste der geöffneten Dateien bzw. Geräte
- Listen für sonstige Betriebsmittel
- Verweis auf einen prozesseigenen Briefkasten
- in Mehrbenutzerystemen: Benutzernummer des Prozesseigentümers
- Privilegien (Ausführungs- und Zugriffsrechte des Prozesses)
- Betriebsmittelkonten für (kommerzielle) Abrechnungszwecke

Beispielhafte Belegung eines Prozeßleitblocks mit Ausschnitten aus der Prozeß-Tabelle, der Bereit- und Warte-Liste

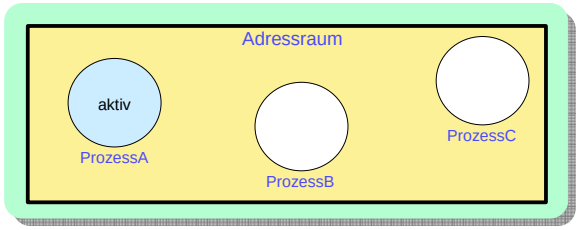


Prozeßkopf

- **Enthält die Daten, die das Betriebssystem für den Prozeß bereitstellt und aktuell hält, weil er sie für seinen korrekten Ablauf braucht:**
  - aktueller Katalog im Dateisystem (working directory)
  - Liste der beim Prozeßstart übergebenen Argumente
  - Identifikation des dem Prozeß zugeordneten Terminals
  - Daten über ablaufende E/A-Vorgänge (Pufferadressen, Transferrichtung)
  - Parameterblock für die bei einem Systemaufruf übergebenen Argumente und danach zurückgegebenen Ergebnisse
  - Fehlermeldungen nach Systemaufrufen
  - Platz zur Aufnahme des zu sichernden Kontextes
  - Betriebsmittelkonten für (kommerzielle) Abrechnungszwecke
- **inhaltliche Trennung zwischen Prozeßleitblock und Prozeßkopf ist nicht immer eindeutig abgegrenzt**
- **Bisweilen werden beide Datenstrukturen auch zusammengefaßt**

Prozesse und Threads

- **Prozesse arbeiten auf disjunkten Adressräumen.**
  - Speicherbelegung, Befehlszähler, Kellerspeicher, geöffnete Dateien, etc.



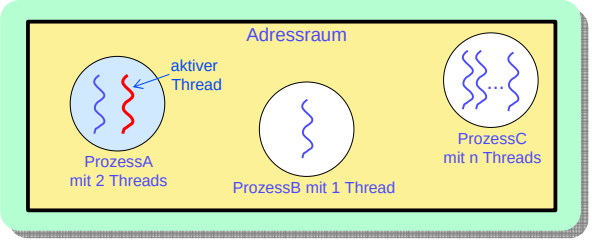
EIN aktiver Prozess wird vom Prozessor bearbeitet.

(Pseudoparallelität)

- **Prozess-Wechsel : abwechselnde Zuteilung der CPU an verschiedene Prozesse**
  - Sicherung des CPU Kontexts :
    Registerwerte, Programmzähler, Stack-Pointer, ...
  - Auslagerung und Einlagerung von Prozessen zwischen Hauptspeicher und Festplatte
  - Zeitaufwändig

Threads

- **Feinere Granularität durch Bereitstellung mehrerer Threads pro Prozess**
- **Threads (Ausführungsfäden) sind leichtgewichtige Prozesse, die genau einem Prozess zugeordnet sind.**
  - Threads eines Prozesses arbeiten auf dem gemeinsamen Adressraum des Prozesses und teilen sich gemeinsame globale Variablen.
  - Für jeden Thread werden bei seiner Unterbrechung Informationen gespeichert, so dass er zu einem späteren Zeitpunkt wieder fortgesetzt werden kann
    - Befehlszähler, lokale Variablen und Kellerspeicher

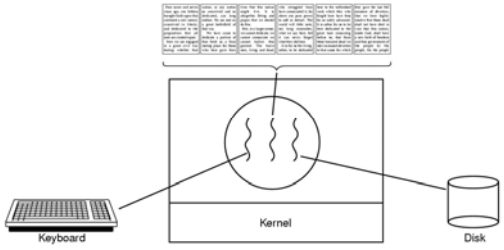


Multi-Threading: mehrere Threads arbeiten im Adressraum eines Prozesses.

Ein aktiver Prozess enthält genau einen aktiven Thread. (Pseudoparallelität)

Threads - Verwendung

- **Multi-Threading ermöglicht es, Parallelität innerhalb eines Programms (d.h. in einer Anwendung) zu beschreiben.**
  - Einige Aufgaben können blockiert werden (z.B. wegen Ein-/Ausgabe-Operationen)
- **Beispiel: Textverarbeitungssystem, in dem gleichzeitig**
  - der Benutzer editiert (Benutzer-Interaktion),
  - die Darstellung am Bildschirm (Layout) neu berechnet wird und
  - im Hintergrund eine automatische Sicherung der Datei erfolgt

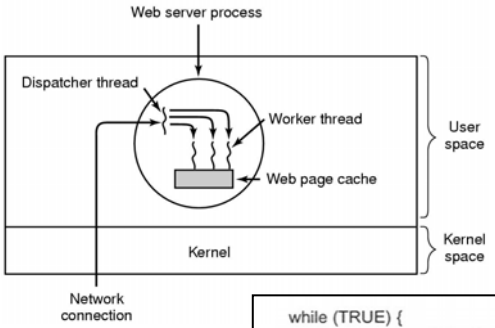


Threads - Verwendung

- **Der Benutzer hat den Eindruck, dass diese Vorgänge gleichzeitig erfolgen, da sie von mehreren Threads pseudoparallel ausgeführt werden.**
- **Threads arbeiten auf denselben Daten -> editiertes Dokument**
  - Keine explizite Kommunikation und Datenübertragung erforderlich
  - Synchronisation der Thread-Zugriffe auf dieselben Daten muss bei Bedarf synchronisiert werden
- **Schnelle Thread-Wechsel**
- **Ähnlichkeit zwischen Thread und Prozess**
  - Ausführung eines Programms auf einem virtuellen Prozessor
  - Geringere Nebenläufigkeits-Transparenz zur Steigerung der Effizienz

Threads - Verwendung

- **Beispiel: Web-Server mit mehreren Threads**




```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

Dispatcher Thread

```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
      read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

Worker Thread

Threads

- **Thread-Kontext**
  - CPU Kontext
  - Informationen zur Thread-Verwaltung
  - schnellere Thread-Wechsel als Prozess-Wechsel, da Thread-Kontext weniger umfangreich als Prozess-Kontext
  - kein automatischer Schutz des Datenzugriffs zwischen Threads innerhalb eines einzelnen Prozesses
  - Leistungsgewinn durch Multi-Threading
  - Erhöhter Aufwand bei Programmierung von multi-threaded Applikationen wegen expliziter Nebenläufigkeit

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Subprozesse (Threads)

- **Mehrere Ablaufeinheiten arbeiten mit derselben Ausstattung an Betriebsmitteln, insbesondere im selben Adreßraum**
  - Diese Ablaufeinheiten werden dann als Sub- oder Mikroprozesse (Leichtgewichtsprozesse, lightweight processes, threads, microtasks)
  - "private" Ausstattung: lediglich ein eigener Registersatz und ein eigener Stapel
  - alle anderen Betriebsmittel, insbesondere den Adreßraum, werden mit den anderen anderen Subprozessen desselben "Hauptprozesses“ geteilt
  - der Kontext eines Subprozesses hat einen kleineren Umfang als der eines "normalen Prozesses".
- **Zeitliche Effizienz: Spezielle Formen der Interprozeßkommunikation und -synchronisation**
  - Synchronisation durch binäre Semaphore, die mit schnellen, besonderen Maschinenbefehlen implementiert sind
  - Interprozeßkommunikation über gemeinsam genutzte Speicherbereiche

Vor- und Nachteile von Subprozessen

- **Vorteile:**
  - Prozeßwechselzeiten sind kürzer als zwischen normalen Prozessen (kleinerer Kontext, keine Umschaltung zwischen unterschiedlichen Betriebsmittelausstattungen)
  - Gemeinsame Nutzung von Betriebsmitteln wird erleichtert (insbesondere eines gemeinsamen Speicherbereichs); Beispiele:
    - Tabellenkalkulationsprogramm mit Subprozessen für Eingabe und Berechnung
    - Serverprozesse in Client/Server-Anwendungen: Bereitstellung bestimmter Betriebsmittel
  - Mit Wartezeiten verbundene Ein/Ausgabe-Aufträge können von eigenen Subprozessen abgewickelt werden. Der "Hauptstrang" der Ablaufeinheit arbeitet ohne Verzögerung weiter
- **Nachteile bei der Strukturierung eines Anwendungssystems:**
  - Koordination der Subprozesse und Zugriff auf gemeinsame Betriebsmittel nur durch Programmierer
  - Ohne Nutzung von Wartezeiten für Ein-/Ausgabe nehmen sich Subprozesse nur gegenseitig Rechenzeit weg, ohne den Gesamtprozeßablauf zu beschleunigen
  - Ein Wechsel zwischen Subprozessen verschiedener Prozesse kostet genauso viel Zeit wie ein normaler Prozeßwechsel.

WinAPI: Erzeugen von Threads - 1

```
#include <windows.h>
#include <iostream>
using namespace std;

CRITICAL_SECTION g_cs;


DWORD WINAPI Thread1(void *param)
{
    int th = * ((int *) param);
    for (int i=0; i<10; i++)
    {
       EnterCriticalSection(&g_cs);
       cout << th << '-' ;
       LeaveCriticalSection(&g_cs);
       Sleep(rand() % 10);
    }
    return th;
}
```

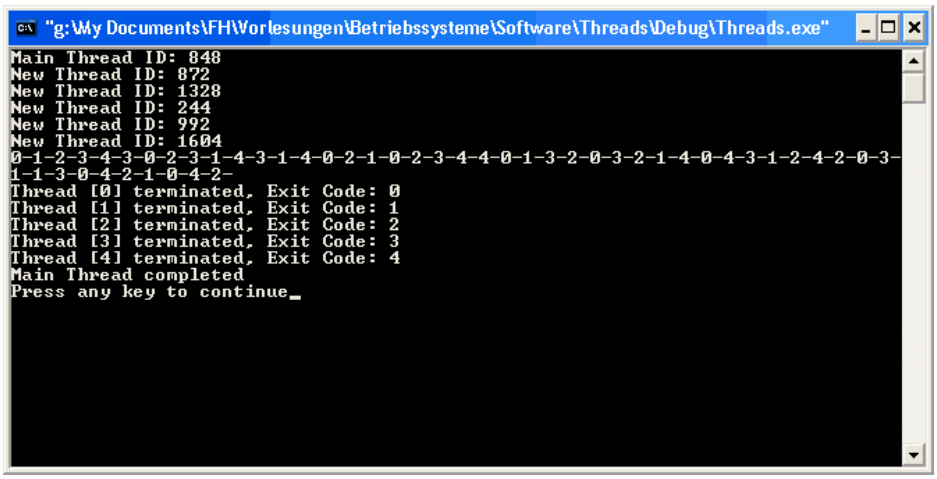
WinAPI: Erzeugen von Threads - 2

```
void main()
{
    InitializeCriticalSection(&g_cs);

    DWORD dwMainThreadId = GetCurrentThreadId();
    cout << "Main Thread ID: " << dwMainThreadId << endl;

    HANDLE hThread[5];
    DWORD dwThreadId;
    int b;

    for (int i=0; i<5; i++) {
        hThread[i] = CreateThread(
            (SECURITY_ATTRIBUTES *) 0,
            0,                          // Stack-Size: default
            & Thread1,                  // Thread-Function
            (void *) &i,                // Parameter passed to the thread
            CREATE_SUSPENDED,           // Thread creation status
            &dwThreadId);               // Thread id
        cout << "New Thread ID: " << dwThreadId << endl;
    }
:
```

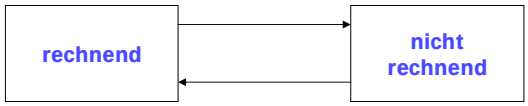

WinAPI: Erzeugen von Threads – 3
main - Fortsetzung

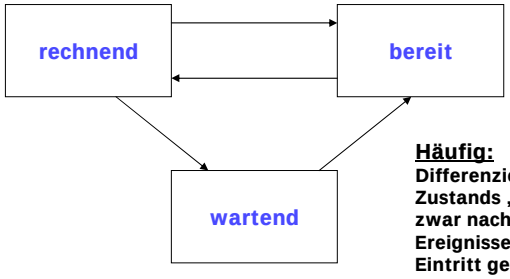
```
:
    for (i=0; i<5; i++)
       b = ResumeThread(hThread[i]);

    DWORD dwExitCode;
    DWORD dwStatus = WaitForMultipleObjects(5,hThread, true, INFINITE);
    cout << endl;

    for (i=0; i<5; i++) {
       b = GetExitCodeThread(hThread[i], &dwExitCode);
       cout << "Thread ["<<i<<"] terminated, Exit Code: " << dwExitCode << endl;
       b = CloseHandle(hThread[i]);
    }

    cout << "Main Thread completed\n";
    DeleteCriticalSection(&g_cs);
}
```

WinAPI: Erzeugen von Threads - 4
Bildschirmausgabe



Prozeßzustände





Häufig:
Differenzierung des Zustands „wartend“, und zwar nach dem Typ des Ereignisses auf dessen Eintritt gewartet wird

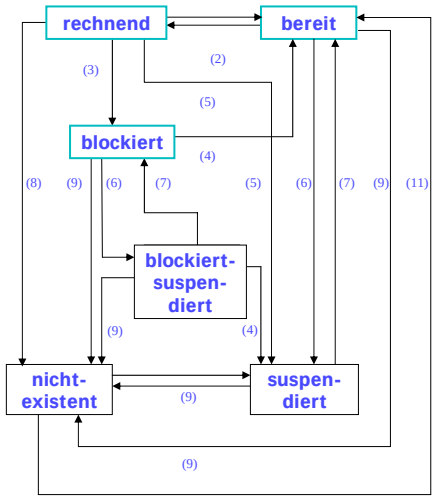
Prozeßzustände

- **"rechnend", "bereit" und "blockiert" sind die elementaren Prozeßzustände**
- **Differenzierung des Zustands "blockiert" nach dem Typ des Ereignisses**
  - alle diesbezüglichen Prozesse werden in einer zustandsspezifischen Liste geführt
    - Eigene Liste für Prozesse, die auf Uhrzeit-Weckvorgange warten
    - Eigene Liste für Warten auf E/A Ausgänge
    - Evtl. Eigene Liste für Warten auf bestimmtes E/A-Gerät
- **Zustandsübergang: Umhängen von einer Liste auf eine andere**
- **Wartestellung/Blockierung läuft der Prozeß „aus freien Stücken“ an**
  (z.B. durch Systemaufruf wie E/A-Anforderung)
- **Wartezustand wird nie durch einen anderen Prozeß ausgelöst**
  - Versetzen vom Warte- in den Bereitzustand durch andere Prozesse möglich!
- **Führen der rechenbereiten Prozesse in einer Bereitliste;**
  - rechnender Prozeß befindet sich dort an ausgezeichneter Position
- **Entzug des Prozessors**
  - Aufgabe des Prozessors durch einen Warte-Aufruf --> Warteliste
  - Verdrängung durch einen anderen Prozeß --> Bereitliste
  - Beendigung

Weitere Prozeßzustände

- **Zustand "geladen" oder "ruhend" (inactive)**
  - In den Hauptspeicher laden, **ohne** sie damit auch zu starten
  - Vorverlagern des zeitlich aufwendigen Ladevorgangs
- **Zustand "suspendiert" (suspended)**
  - Laden ohne Starten
  - durch sich selbst oder durch einen anderen Prozeß mit Hilfe eines Systemaufrufs
  - Schutzmechanismen (Recht nur für Superuser / Vaterprozesse)
  - Einfache Form der Synchronisation zwischen Prozessen
    (Alle anderen Prozesse - die auf dasselbe BM zugreifen wollen - werden suspendiert
- **Kombinierter Zustand "blockiert/suspendiert"**
  - Wenn danach das erwartete Ereignis eintritt, bleibt er suspendiert
  - wenn er stattdessen fortgesetzt wird, muß er weiter auf das Ereignis warten
- **Zustand "nichtexistent" (nonexistent)**
  - Prozesse, die dem Betriebssystem gar nicht bekannt sind
    - Prozesse, die noch auf der Festplatte ruhen, und erst geladen werden müssen
    - beendete Prozesse, nachdem sie aus der Prozeßtabelle entfernt wurden.
- **Weitere (Parallel-)Zustände: etwa für Prozeßauslagerung**

Erweitertes Zustandsdiagramm



Nr.	Aktion	Verursacher
(1)	Verdrängen	Betriebssystem
(2)	Zuordnen	Betriebssystem
(3)	blockierender Aufruf	Prozeß selbst
(4)	Erwartetes Ereignis tritt ein (Warte-Bed. erfüllt)	externes Ereignis, fremder Prozeß
(5)	Suspendier-Aufruf	Prozeß selbst
(6)	Suspendier-Aufruf	fremder Prozeß
(7)	Fortsetz-Aufruf	fremder Prozeß
(8)	Prozeßende	Prozeß selbst
(9)	Prozeß-Abbruch	fremder Prozeß, Betriebssystem
(10)	Laden ohne Starten	fremder Prozeß
(11)	Laden mit Starten	fremder Prozeß

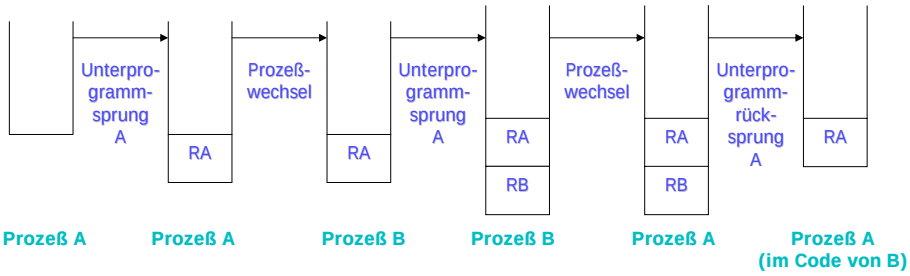
Prozeßwechsel

- **Bei einem Mehrprozeßsystem konkurrieren Prozesse um Rechenzeit, also um das Betriebsmittel Prozessor**
- **Prozessauswahl**
  - mittels Prioritätenvon Prozessen aus der Bereitliste
  - Restfristen für die Prozeßantworten ~ Abbildbar auf Prioritäten
  - Der lauffähige Prozeß mit der besten Priorität bekommt den Prozessor. Falls es mehrere derartige Prozesse gibt, kommt der dran, der am längsten auf der Bereitliste wartet.
- **Prozessverdrängung und Unterbrechbarkeit**
  - der rechnende Prozeß gibt den Prozessor freiwillig auf, indem er
    - sich beendet oder
    - einen blockierenden Systemaufruf abgibt (eine Wartestelle anläuft)
  - oder er wird verdrängt, d. h., ihm wird das Betriebsmittel "Prozessor" vom Betriebssystem zwangsweise entzogen.
- **Unterscheidung: Systeme mit und ohne Prozessverdrängung (Preemption)**

Verdrängende Betriebssysteme

- **Verdrängungsereignisse**
  - Zeitscheibenverfahren: die gegenwärtige Zeitscheibe ist abgelaufen und ein Prozeß gleicher Priorität ist bereit
  - ein Prozeß besserer Priorität wurde lauffähig
    - eine Prioritätsverbesserung eines bereiten Prozesses (z.B. bei Unix)
    - ein externes Ereignis (letztlich durch eine Unterbrechungsanforderung ausgelöst) oder
    - eine Maßnahme der Interprozesskommunikation oder -synchronisation oder aber durch einen Fortsetzaufruf an einen suspendierten Prozeß.
- **Unterbrechungen/Verdrängungen durch externe Ereignisse:**
  - Unterbrechung kann an jeder beliebigen Stelle im Code erfolgen
    --> Verdrängter Prozeß kann nicht für Kontextsicherung und -wiederherstellung verantwortlich gemacht werden
  - Notwendigkeit pro Prozeß einen Stapel zu führen

Notwendigkeit für prozeßeigene Stapel, Verdrängbarkeit



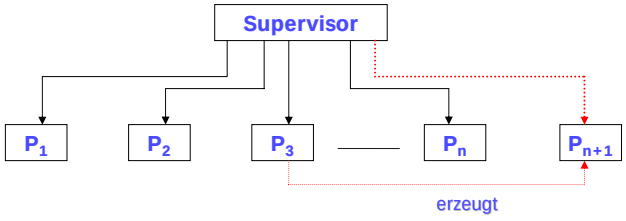
- **Verdrängbarkeit (Preemptability), Abstufungen:**
  - Verdrängung ist nur zum Ende einer Zeitscheibe
  - Verdrängung ist beim Durchlaufen von Anwendungscode jederzeit, beim Durchlaufen von Betriebssystemcode nur vor der Rückkehr in den Anwendungscode möglich
  - Verdrängung ist auch beim Durchlaufen von Betriebssystemcode, abgesehen von kurzen Sperrabschnitten, jederzeit möglich

Ablauf des Prozeßwechsels

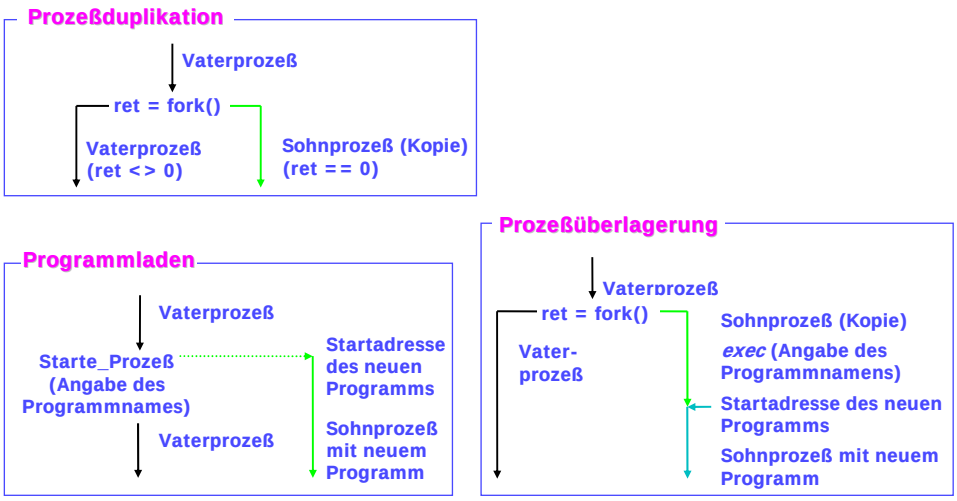
- **Ablaufplaner (Scheduler)**
  - Verwaltung der Bereitliste
- **Prozeßumschalter (Dispatcher)**
  - Durchführung des Prozeßwechsels, d.h. Kontextsicherung und -wiederherstellung
- **Durchführung des Prozeßwechsels**
  - Aktuellen Befehlszähler (PC, program counter) auf dem Stapel sichern
  - Kontext (und damit auch den Stapelzeiger) sichern
  - Neuen Zustand in den Prozeßleitblock des alten Prozesses eintragen
  - Prozeßleitblock des alten Prozesses "weghängen" (an Ablaufplaner übergeben)
  - Prozeßleitblock des neuen, vom Ablaufplaner gewählten Prozesses adressieren
  - Kontext des neuen Prozesses wiederherstellen (schaltet auf neuen Stapel um)
  - Rücksprungbefehl durchführen

Prozeßsysteme

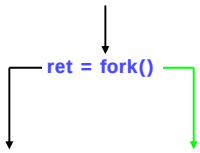
- **Varianten von Prozeßsystemen**
  - statisch vs. dynamisch
  - hierarchisch vs. flach
- **Vater- / Sohnbeziehungen**
  - besondere Rechte des Vaters:
    Abbruch, Ablaufverfolgung, Suspendieren, Prioritätsänderung
  - einfachere Interprozeßkommunikation zwischen verschiedenen Söhnen
- **Beispiel: flaches Prozeßsystem:**



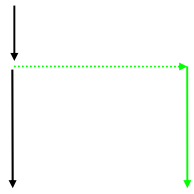
Prozeßerzeugung



Formen der nebenläufigen Abarbeitung



- **Synchroner Ablauf dieser Varianten**
  - Vater wartet auf die Beendigung des Sohnes
  - ~ Unterprogrammaufruf mit dynamischem Binden
    Unterprogramm braucht beim Programmieren noch nicht bekannt zu sein
  - Kommunikation zwischen Vater- und Sohnprozeß während der Laufzeit des Sohnprozesses weder möglich noch nötig (z.B. Programmausführung im Vordergrund der Shell)



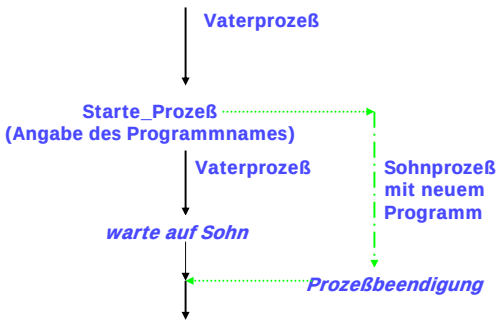
- **Asynchronfall**
  - Vater und Sohn arbeiten parallel weiter, wobei sie grundsätzlich zeitlich voneinander unabhängig ablaufen
  - Sie können während des Ablaufs miteinander kommunizieren und sich synchronisieren
    (über entsprechende Systemaufrufe)

Schritte bei der Prozeßerzeugung

- Sicherstellen, daß noch genügend Speicherplatz (Platz in der Prozeßtabelle, Hauptspeicher usw.) zur Verfügung steht
- einen Eintrag in der Prozeßtabelle (Prozeßleitblock) reservieren
- eine freie Prozeßnummer bestimmen und in den Prozeßleitblock eintragen
  - bei Variante 1 (Prozeßduplikation): Speicher allozieren und den Datenbereich sowie den Stapel des Vaterprozesses in den des Sohnprozesses kopieren; Verweis vom Prozeßleitblock auf das beim Vaterprozeß schon vorhandene Codesegment einrichten
  - bei Variante 2 (Programmladen): Speicher allozieren, Programm (Code- und Datenteil) vom Massen- in den Hauptspeicher kopieren
- die zu vererbenden Teile des Prozeßleitblocks und des Prozeßkopfes in den Kontext des Sohnes kopieren (damit wird der Kontext im engeren Sinne aufgebaut, außerdem werden dabei auch die übrigen Betriebsmittel vererbt)
- den Sohnprozeß ablaufbereit setzen, im asynchronen Fall den Vaterprozeß fortsetzen und ihm in jedem Fall die Prozeßnummer des Sohnes mitteilen

- Schritte bei der Prozeßüberlagerung: zusätzlich benötigten Speicher allozieren oder überflüssigen freigeben, Programm in den Hauptspeicher kopieren

Prozeßbeendigung

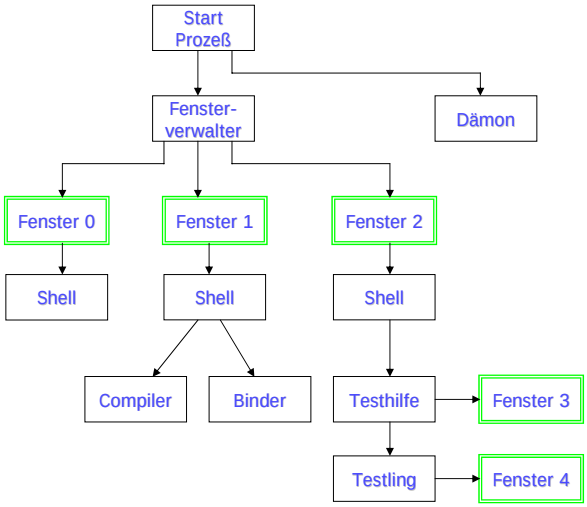


- Möglichkeit im Asynchronfall: Vater wird vor dem Sohn beendet
  - Sohn dem Großvater oder (wie in Unix) dem Initialisierungsprozeß als neuer Sohn zugeordnen
  - Sohn (und weitere Abkömmlinge) werden mit beendet
- Nachteil bei 2. Variante:
  - Sohnprozesse können nicht im Hintergrund weiterlaufen

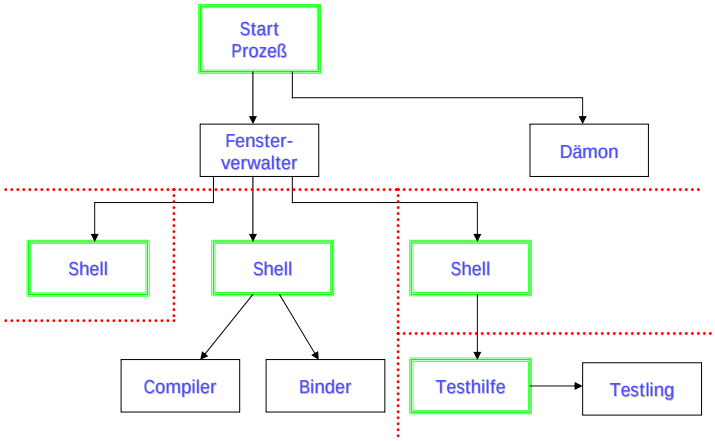
Aktionen bei Prozeßbeendigung

- geöffnete Dateien schließen
- Semaphore und andere Betriebsmittel freigeben;
  auf Kommunikation wartende Fremdprozesse mit entsprechender Fehlermeldung fortsetzen
- vom Prozeß belegten Hauptspeicher freigeben
- Sohnprozesse umhängen oder abbrechen (gegebenenfalls rekursiv!)
- in Systemen mit Betriebsmittelkontierung die Verbrauchsdaten (etwa Prozessor-Zeit, Ein/Ausgabe-Zeit, Speicherverbrauch) feststellen und unter der Benutzernummer abspeichern
- falls nötig, den auf die Beendigung wartenden Vaterprozeß bereitsetzen bzw. benachrichtigen
- Prozeßleitblock freigeben

Beispiel eines hierarchischen Prozeßsystems mit Zuordnung von Fenstern



Prozeßfamilien



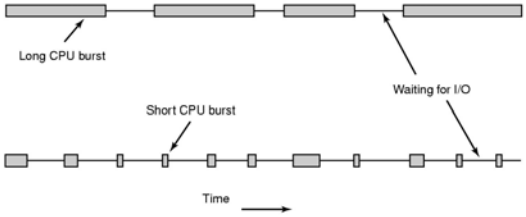
Ablaufplanung (Scheduling)

- **Ziele**
  - Durchsatzoptimierung (Stapel-Betrieb)
  - Optimierung der Auslastung der Betriebsmittel (Stapel-Betrieb)
  - kurze Antwortzeiten (Dialog-Betrieb)
  - garantierte Reaktionszeiten (Echtzeit-Betrieb)
  - Verklemmungsvermeidung
  - sparsamer Umgang mit wertvollen Betriebsmitteln
- **Ebenen der Ablaufplanung**
  - **langfristige Ablaufplanung**
    Wann werden Programme gestartet? Strategie kann durchaus zeitlich variabel sein
  - **mittelfristige Ablaufplanung**
    Koordinierungsfragen bei der Betriebsmittelnutzung und -überbeanspruchung, Hauptspeicherbelegung und Prozeß-Auslagerung)
  - **kurzfristige Ablaufplanung**
    Auswahl eines der Prozesse auf der Bereitliste

Scheduling

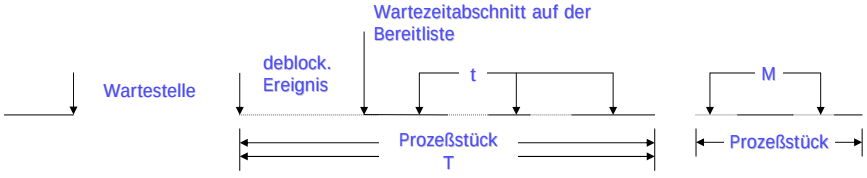
- **Häufige Prozesswechsel erhöhen den Verwaltungsaufwand zur Sicherung und Wiederherstellung der Prozess-Kontexte**
  - Z.B. mehrere Prozess-Wechsel in kurzer Zeit verbrauchen viel CPU-Rechenzeit

- **Prozess-Verhalten**
  - Hohe Rechenlast -> rechenlastige Prozesse
    - Lange CPU-Nutzungszeiten
  - Viele Ein-/Ausgabe-Anforderungen -> Ein/Ausgabe-lastige Prozesse
    - Kurze CPU-Nutzungszeiten, häufige Ein/Ausgabe-Wartezeiten
    - Z.B. Festplatte-Zugriffe, Warten auf Benutzereingabe



Wichtige Parameter der Ablaufplanung

t	**benötigte Ausführungszeit (reine Rechenzeit) des Prozeßstücks (Unix: ~ "user time" + "system time")**
T	**gesamte Antwortzeit inklusive Wartezeiten (Unix: ~ "real time")**
M := T – t	**Wartezeit auf Bereitliste (missed time)**
R := t/T	**response ratio, Antwortrate**
P := T/t	**penalty ratio, Verzögerungsrate**

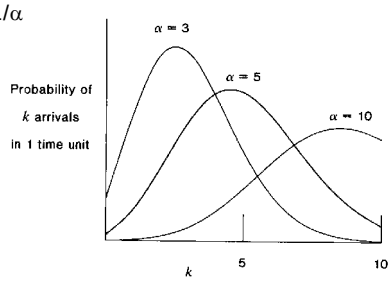


Grundlagen zur Analyse von Auswahlstrategien

Ankunftsprozeßstrom der Prozesse auf der Bereitliste und Ablaufzeit der Prozesse

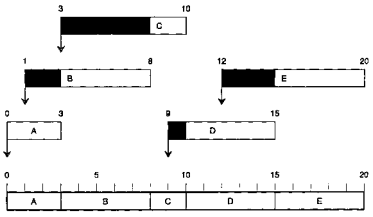
Beide: Exponentialverteilung mit den Parametern α und ß

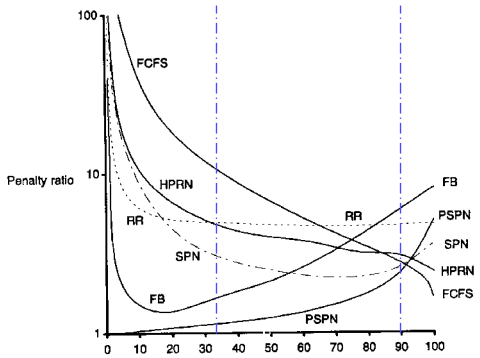
- Wahrscheinlichkeit, daß irgendein Prozeß innerhalb von τ bereit wird
  $W(X=\tau) = 1 - e^{-\alpha\tau}$
- Wahrscheinlichkeit, daß ein Prozeß innerhalb τ abgearbeitet wird
  $W(X=\tau) = 1 - e^{-ß\tau}$
- Erwartungswert der Exponentialverteilung: $EX = 1/\alpha$
- Ankünfte der Prozesse bilden einen Poisson-Prozeß
  $W(X=k) = e^{-\alpha}\alpha^k/k!$
- Sättigungs- oder Auslastungsgrad des Rechners
  $\rho := \alpha/\beta$



Ankunftsreihenfolge (FCFS; First Come, First Served)

Prozeßname	Ankunftszeit	Rechenzeit
A	0	3
B	1	5
C	3	2
D	9	5
E	12	5





Ankunftsreihenfolge (FCFS: first come, first served)

- Auswahl nach der Ankunftsreihenfolge auf der Bereitliste
- Neuzugänge kommen immer ans Ende der Bereitliste
- Prioritäten werden nicht verwendet!
- Rechnende Prozesse werden nicht verdrängt
  das hilft Prozeßwechselzeiten sparen und optimiert den Gesamtdurchsatz
- jeder Prozeß kommt zum Rechnen, es gibt kein "Aushungern" (starvation, livelock)

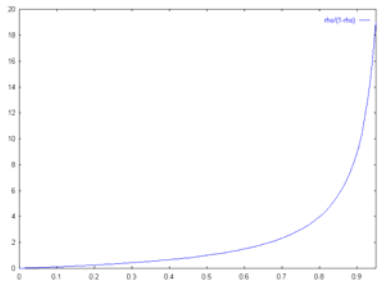
- Bevorzugung / Benachteiligung:
  FCFS begünstigt lange Prozesse und benachteiligt kurze

Process name	Arrival time	Service required	Start time	Finish time	T	M	P
A	0	1	0	1	1	0	1.00
B	0	100	1	101	101	1	1.01
C	0	1	101	102	102	101	102.00
D	0	100	102	202	202	102	2.02
Mean					101.5	51.0	28.1

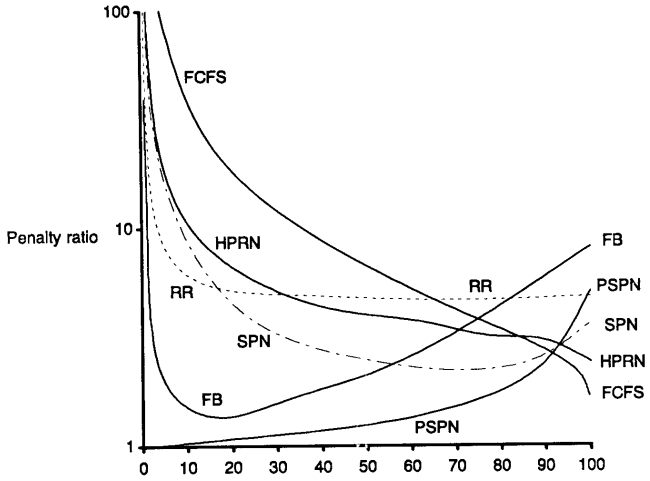
Ankunftsreihenfolge (FCFS: first come, first served)

$$M = \frac{\rho}{\beta\,(1-\rho)} \qquad T(t) = t + \frac{\rho}{\beta\,(1-\rho)} \qquad P(t) = 1 + \frac{\rho}{t\,\beta\,(1-\rho)}$$

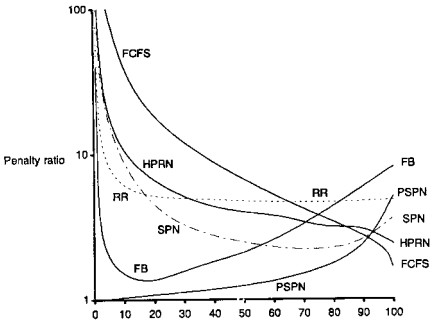
- Mit $\rho = 0.8$ und $\beta = 1$ erhält man $M = 4$, $T(t) = t + 4$ und $P(t) = 1 + 4/t$

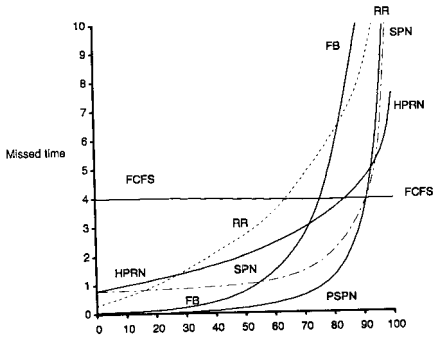


Prozessorvergabestrategien
Verzögerungsraten im Vergleich

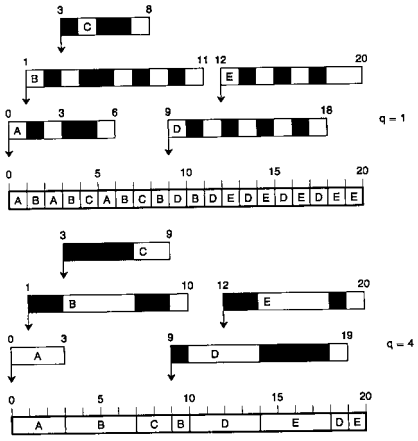


Verzögerungsraten und
Wartezeiten auf der Bereitliste im Vergleich





Round Robin (RR)



$q = 1$

Process name	Arrival time	Service required	Finish time	T	M	P
A	0	3	6	6	3	2.0
B	1	5	11	10	5	2.0
C	3	2	8	5	3	2.5
D	9	5	18	9	4	1.8
E	12	5	20	8	3	1.6
Mean				7.6	3.6	1.98

$q = 4$

Process name	Arrival time	Service required	Finish time	T	M	P
A	0	3	3	3	0	1.0
B	1	5	10	9	4	1.8
C	3	2	9	6	4	3.0
D	9	5	19	10	5	2.0
E	12	5	20	8	3	1.6
Mean				7.2	3.2	1.88

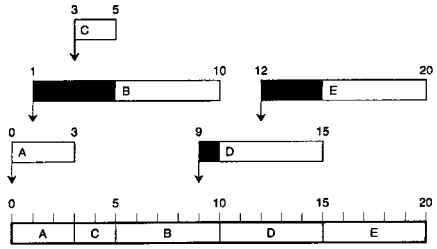
Round Robin (RR)

- **günstige Antwortzeiten für kurze Prozesse, benachteiligt aber lange Prozesse**
- **Wahl des Zeitquantums:**
  - q → ∞ : RR geht in FCFS über
  - q → 0 : der Zeitaufwand für die Prozeßwechsel nimmt überhand.
  - ein kürzeres q ergibt kürzere Antwortzeiten, kostet jedoch mehr Prozeßwechselzeit
  - Wahl von q kann Sache der langfristigen Ablaufplanung sein
- **Für q → 0 :**

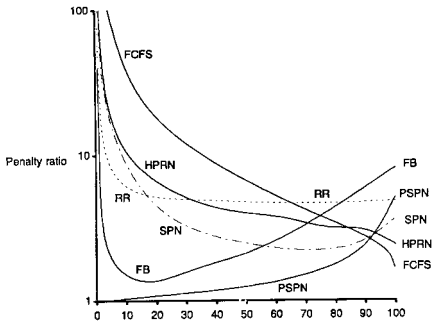
$$T(t) = \frac{t}{1-\rho} \qquad P = \frac{1}{1-\rho} \qquad M = T(t) - t = t\,\frac{\rho}{1-\rho}$$

- **ρ = 0.8 ergibt hier P = 5!**
- **18 Prozeßwechsel mit q=1; 7 Prozeßwechsel mit q=4**

Shortest Process next (SPN)



Process name	Arrival time	Service required	Start time	Finish time	T	M	P
A	0	3	0	3	3	0	1.0
B	1	5	5	10	9	4	1.8
C	3	2	3	5	2	0	1.0
D	9	5	10	15	6	1	1.2
E	12	5	15	20	8	3	1.6
Mean					5.6	1.6	1.32



Shortest Process next (SPN)

- Zusatzinformation ist für jeden Prozeß notwendig:
  die (voraussichtliche) Rechenzeit für das aktuelle Prozeßstück
- Wichtig, ob es sich um ein "kurzes" (E/A-intensives) oder "langes" (rechenintensives) Prozeßstück handelt
- diese Information kann statistisch beim Prozeßablauf gesammelt werden, da ein Prozeß i.a. mehrmals auf die Bereitliste gerät
- Hierzu wird für ein Prozeßstück, beginnend mit einem Durchschnittswert für alle Prozeßstücke als Anfangswert, das exponentielle Mittel eP gebildet gemäß

  e'P := 0,9eP + 0,1s
- guter Kompromiß zwischen FCFS und RR, wobei das jeweils ungün-stigere Extremverhalten der beiden anderen Verfahren vermieden wird
- (4 Prozeßwechsel bei der Modellprozeßmenge, da verdrängungsfrei)

"Kürzester Prozeß zuerst" mit Verdrängung (PSPN, Preemptive Shortest Process Next)

- der jeweils rechnende Prozeß wird verdrängt, wenn ein Prozeß bereit wird, dessen Rechenzeitanforderung geringer ist als der Restrechenzeitbedarf des rechnenden Prozesses.
- P(t) und M(t) sind in der Simulation – abgesehen von den längsten zehn Prozent – deutlich besser als bei allen anderen Verfahren
- Neigung zum Aushungern sehr langer Prozesse
- beste durchschnittliche Verzögerungsrate P
  Bereitliste wird so kurz wie möglich gehalten
- Vergabestrategie verhält sich besser als RR, kommt aber mit deutlich weniger Prozeßwechseln aus.
- theoretisch nahezu optimal; in der Praxis verhält sie sich wegen der statistisch zu ermittelnden Rechenzeitprognose nicht so ideal.

"Höchste Verzögerungsrate zuerst" (HPRN, Highest Penalty Ratio Next)

- für jeden Prozeß P := (w+t)/t berechnen
- Der Prozeß mit dem höchsten Wert von P bekommt den Prozessor
- P beginnt mit einem Wert von 1 und wächst im Laufe der Wartezeit und führt endlich dazu, daß der Prozeß unter allen bereiten den höchsten Wert von P hat und rechnen darf.

- Kein Prozeß wird ausgehungert, solange $\rho < 1$ ist
- Im simulierten Verhalten liegt HPRN zwischen SPN und FCFS

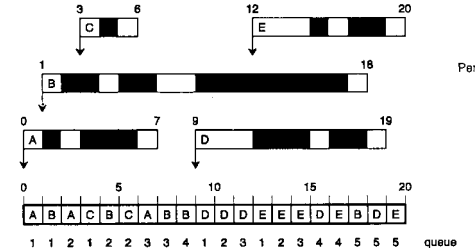
- Nachteile:
  - Da HPRN ohne Verdrängung auskommt, wird ein kurzer Prozeß direkt nach einem sehr langen ebenso benachteiligt wie bei FCFS und SPN.
  - Die Auswahlentscheidung kostet Rechenzeit (Gleitkomma-Division, Messung von t und w, Umordnung der Bereitliste).
  - SPN ist außer bei sehr langen Prozessen immer überlegen, so daß sich der Mehraufwand kaum lohnt.

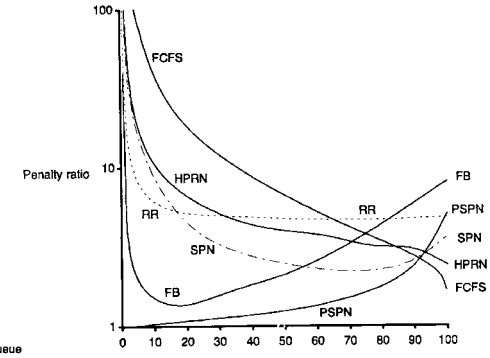
Mehrebenen-Rücklauf (FB, Multiple-level Feedback)

- **Hier wird die Bereitliste in einige Teillisten (Ebenen) sich verschlechternder Priorität aufgespalten: Ebene 1, Ebene 2, Ebene 3 usw.**
- **Wenn ein Prozeß eine bestimmte Zahl von Zeitscheiben auf seiner Ebene verbraucht hat, wird er ans Ende der nächstschlechteren Ebene gesetzt**
- **Dieses Verfahren verhält sich ähnlich wie das RR, jedoch bekommen kürzere Prozesse erheblich bessere Priorität als längere**
- **FB ist für 80 % der Prozesse besser als RR, für die langsamsten 20 % schlechter. Dies führt vor allem im Dialogbetrieb zu kurzen Antwortzeiten.**
- **keine Daten zur Vorhersage der benötigten Rechenzeit benötigt**
- **Die Charakteristik eines Prozesses (E/A- oder rechenbetont) führt automatisch und dynamisch zur Plazierung auf der passenden Teilliste**
- **Varianten:**
  - Die Quantengröße (Zeitscheibenlänge) hängt von der Ebene ab, also z. B. $2^n q$ oder nq auf Ebene n
  - Ein Prozeß der Ebene n wird erst nach n oder 2^n Zeitscheiben auf die nächste Teilliste abgesenkt
  - Ein Prozeß wird nach einer bestimmten Wartezeit wieder auf eine bessere Teilliste angehoben.
  - Statt Prioritäten zwischen den Ebenen festzulegen, wird jeder Ebene eine Zeitscheibe für alle Prozesse zusammen zugeordnet (wachsend mit Ebenen-Nummer).

Mehrebenen-Rücklauf (FB, Multiple-level Feedback)

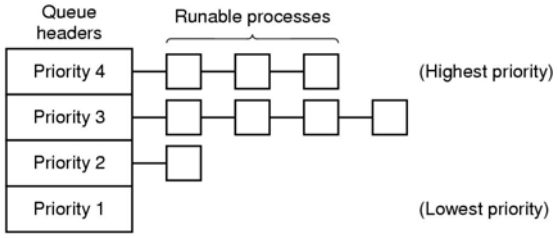
Process name	Arrival time	Service required	Finish time	T	M	P
A	0	3	7	7	4	2.3
B	1	5	18	17	12	3.4
C	3	2	6	3	1	1.5
D	9	5	19	10	5	2.0
E	12	5	20	8	3	1.6
Mean				9.0	5.0	2.16





Prioritäten-Basiertes Scheduling

- **Vorgehensweise**
  - Jeder Prozess besitzt eine Priorität, die zur Bestimmung des Scheduling dient
  - Ein Prozess mit der besten Priorität bekommt die CPU zugeteilt
- **Arten von Prioritäten**
  - externe Prioritäten (vom Benutzer, typisch für Echtzeit-Systeme) oder vom Betriebssystem bestimmt (wie in Unix, VMS)
  - Statische oder dynamische Zuteilung der Prioritäten
    - Z.B. anhand der Benutzer (Process-Owner), Warte- oder Laufzeit des Prozesses

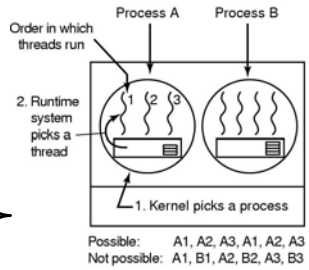


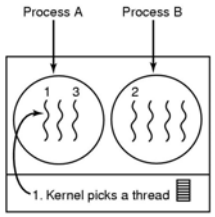
Prioritäten-Basiertes Scheduling

- **Entscheidung innerhalb der gleichen Prioritätsebene**
  - Abarbeitung strikt nach Reihenfolge (FCFS); ggf. auch mit Verdrängung (Schlechtes Time-Sharing-Verhalten; Einsatz eher im Echtzeit-Betrieb)
  - Umlaufverfahren (RR): Zuteilung der CPU an die Prozesse mit der höchsten Priorität; gleichmäßigere Verteilung der Rechenzeit.
- **Aufteilung der Prioritäten in Klassen**
  - 1. Verdrängende Prioritäten
    - Jederzeitige Verdrängung anderer Prozesse mit schlechteren Prioritäten möglich
  - 2. Nicht verdrängende, meistens dynamisch veränderliche Prioritäten
    - Teilnehmer-Betrieb, zeitscheiben-gesteuert
  - 3. Hintergrund-Prioritäten
    - Produktive Nutzung überschüssiger Rechenzeit; ggf. auch Auslagerung auf Platte.
    - Häufig dynamisch priorisiert, keine Zeitscheiben
  - 4. Leerlaufpriorität
    - Priorität des Leerlaufprozesses
    - Halt-Zustand des Prozessors oder Spezialdienste wie Freispeicherverwaltung, System-Selbsttest

Thread-Scheduling

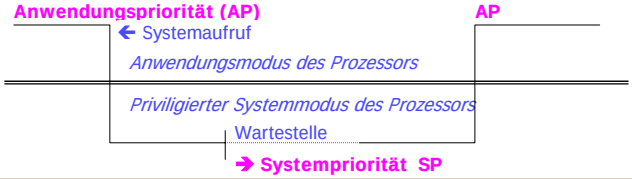
- **Zwei Ebenen von Parallelität**
  - Prozesse
  - Threads
- **Threads auf Anwenderebene**
  - Betriebssystem-Kernel kennt keine Threads
  - reines Prozess-Scheduling des Kernels
  - Anwendungs-spezifische Thread-Scheduler
- **Threads auf Kernel-Ebene**
  - Betriebssystem-Kernel kennt Threads
  - Thread-Scheduling mit oder ohne Berück-sichtigung der Prozess-Zugehörigkeit
  - Bevorzugung eines Thread-Wechsels im selben Prozess, da dieser günstiger ist als ein Prozess-Wechsel





Fallstudie 1: Ablaufplanung bei Unix System V

- **Unix System V: modifizierte Variante des Mehrebenen-Rücklaufs**
- **Implementierung mit Hilfe von Prioritäten (0 - beste ... 127)**
  - 0-60 Systemprioritäten; 61-127 Anwendungsprioritäten
- **Ablaufplanung im Rhythmus von Zeitscheiben von je einer Sekunde**
  - Eine Zeitscheibe besteht meistens aus 50 oder 60 Systemtakten (Ticks)
  - Auswahlentscheidungen können auch innerhalb einer Zeitscheibe getroffen werden, wenn entsprechende Ereignisse eingetreten sind.
- **Zuweisung einer festen Systempriorität, wenn eine Wartestelle angelaufen wird (Abhängig von der Art der Wartestelle)**
  - Ziel: Möglichst schnelles Verlassen des Systemkerns und Freigabe wertvoller BM

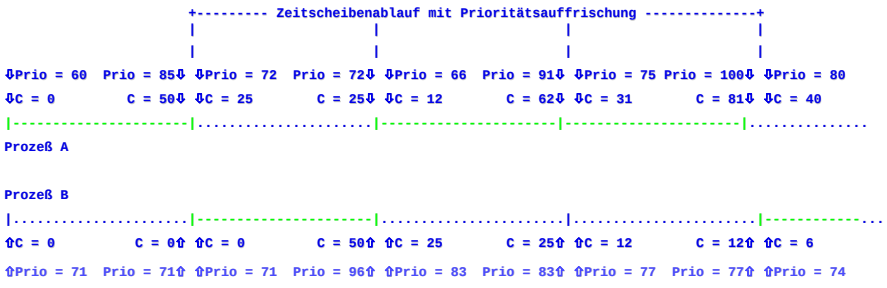


Treffen der Auswahlentscheidung

- **Prozeßstart mit fester Anfangspriorität (i. a. 60) ggf. “+ nice“**
- **Eine Auswahlentscheidung fällt an ...**
  - Wartestelle, Beendigung eines Systemaufrufs, externes Hardware-Ereignis (Beendigung der Unterbrechungsbehandlung), internes Ereignis (z.B. IPKS), Zeitscheibenablauf
  - Nicht jede Auswahlentscheidung führt auch zum Prozeßwechsel!
- **Auswahlentscheidung: Höchste Priorität+RR**
  - Prioritätsverschlechterung alle zwei Ticks um je eine Einheit, max 127
    Beim nächsten Auswahlzeitpunkt kann Prozeßwechsel erfolgen
  - Priorität := min (C DIV 2 + A + N, 127)
- **Prioritätsauffrischung: Halbierung der Prozessorzeit [C := C DIV 2] einmal pro Zeitscheibe**
  - neu gestartete Prozesse werden nicht unangemessen bevorzugt
  - rechenintensive Prozesse werden nicht ausgehungert
  - die Prioritäten länger laufender Prozesse wachsen nicht alle auf 127
  - eine Verhaltensänderung eines Prozesses von "lang" nach "kurz" wirkt sich mit der Zeit auch in der Prioritätseinstufung aus

Beispiel eines Zeitscheibenablaufs mit Prioritätsauffrischung

- **Start zweier Prozesse A und B am Anfang einer Zeitscheibe**
- **B erhält den *nice*-Zuschlag 11**
- **Ticklänge 20 ms (50 Ticks pro Sekunde)**



Vergleich verschiedener Unix-Versionen

- Verfahren verhält sich bei allen Unix-Systemen grundsätzlich gleich
- Auffrischungsfaktoren und -perioden können jedoch unterschiedlich sein

	System V	6. Ed.	7. Edition
Priorität	min (C DIV 2 + A + N, 127)	min (C DIV 16 + A + N, 127)	
Auffrischung C :=	C DIV 2	C*0.8	max (Min.zeit, C-10)

Fallstudie 2: Windows NT 4.0
Prioritätsgesteuertes preemtives Scheduling-System

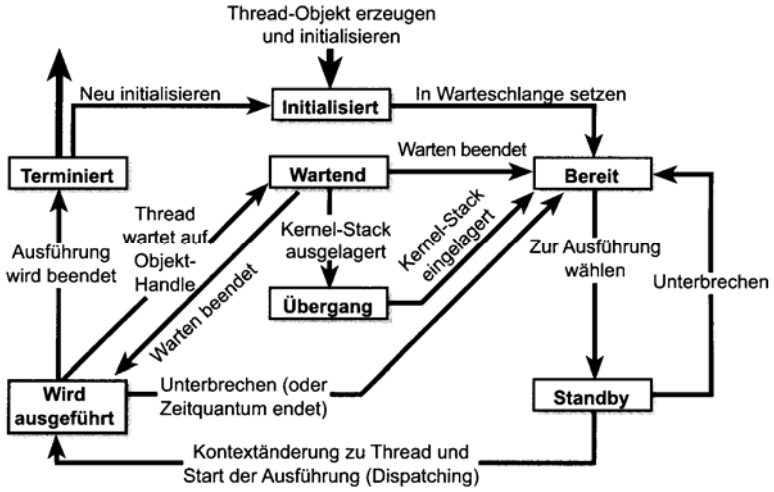
- 32 Prioritätsebenen

31 : 16	Echtzeitprioritäten
15 : 1	variable Prioritäten
0	Leerlaufprozeß

Win32 - API Prioritätsklassen

relative Thread-Prio.	Echtzeit	Hoch	Normal	Leerlauf
Zeitkritisch	31	15	15	15
Maximum	26	15	10	6
Angehoben	25	14	9	5
Normal	24	13	8	4
Abgesenkt	23	12	7	3
Minimum	22	11	6	2
Leerlauf	16	1	1	1

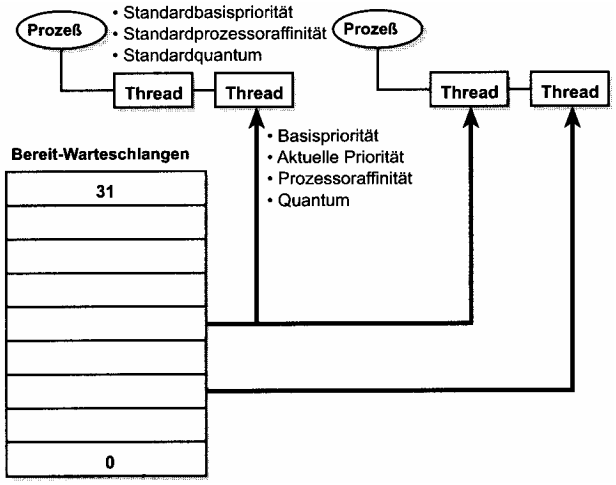
Thread Zustände



Quantum

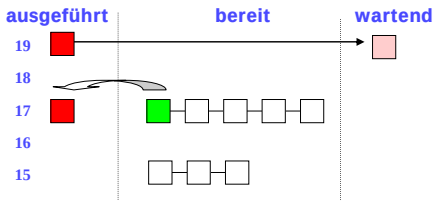
- Quantum-Wert pro Thread: Ausführungsdauer des Threads, Ganzzahlwert
- Standardwert Windows NT Workstation: 6, NT-Server: 36
- Pro Tick werden 3 Einheiten vom Quantum abgezogen
- Länge eines Ticks ist abhängig von der Hardware-Plattform
  (x86-Einprozessorsysteme: 10ms, x86-Multiprozessorsysteme: 15 ms)
- Standardausführungszeit auf Pentium:
  NT-Workstation: 30ms, NT-Server: 180ms
- NT Workstation: Quantum kann temporär für Vordergrund-Threads erhöht werden
- Ein Quantum wird verdoppelt, wenn Windows NT die Priorität eines Threads erhöht, um zu versuchen, eine Prioritätsinversion zu verhindern.
- Bei Verlassen eines Wartezustands: Rücksetzen des Quantums auf den Standardwert (Echtzeit-Prozesse) oder Reduktion um eine Einheit (normale Prozesse)

Scheduling-Datenstruktur



Scheduling-Szenarien

- **Freiwilliger Wechsel**
  - Anlauf einer Wartestelle
  - Prozeß wird wartend gesetzt
  - Priorität wird nicht verändert
  - Quantum wird nach Beendigung des Wartens um eine Einheit vermindert



- **Unterbrechung**
  - Thread wird an den **Anfang** der Bereit-Warteschlange der jeweiligen Priorität gesetzt
  - Nach Neuaktivierung kann er das Restquantum aufbrauchen
  - Unterbrechung kann auch bei im Kernel-Modus befindlichen Threads passieren
- **Quantum ist zu Ende**
  - Prozeß kommt an das **Ende** der Bereit-Warteschlange seiner Prio.-Ebene
  - Suche nach einem Prozeß mit höherer Priorität
  - Sonst: Erneute Zuweisung eines Quantums
- **Prozeßterminierung**

Anpassung des Schedulings (1)

- **Verlängerung des Quantums der Threads im Vordergrundprozeß**
  - Bei Windows NT kann das Quantum für den Vordergrundprozeß erhöht werden
  - Systemsteuerung > System > Leistungsmerkmale > Steigerungsvarianten: keine (Q*1), mittel (Q*2) und maximal (Q*3)
- **Prioritätssteigerung bei Beendigung des Wartezustands**
  - Temporäre Prio.-Erhöhung nach Beendigung bestimmter E/A- und Warteoperationen, um möglichst sofort mit der Bearbeitung fortfahren zu können
  - Höhe der Prioritätssteigerung hängt von der Art des Wartens ab; wird vom Gerätetreiber festgelegt (Vorschläge in Datei: \Ddk\Include\Ntddk.h)

Gerät oder Objekt	**Steigerung (max. 15!)**
Ereignis, Semaphore	1
Festplatte, CD-ROM, Parallele Schnittstelle, Graphikkarte	1
Netzwerk, Mailslot, Named Pipe, Serielle Schnittstelle	2
Tastatur, Maus	6
Sound	8

  - Bei Aktivierung ein Quantum lang Ausführung auf der erhöhten Prioritätsebene
  - Nach Ablauf des Quantums Reduktion der Prioritätsebene um eine Ebene bis die Basispriorität erreicht ist

Anpassung des Schedulings (2)

- **Prioritätssteigerung bei Threads, die in den Wartezustand übergehen**
  - Bei Threads, die auf auf Benutzereingaben oder Fensternachrichten warten, wird die aktuelle Priorität auf den Wert 14 gesetzt
  - Bei Aktivierung wird das Quantum verdoppelt und die Priorität bei Ablauf des Quantums sofort wieder auf die Basispriorität zurückgesetzt
- **Prioritätssteigerung bei Threads, die keine Prozessorzeit zugeteilt bekommen haben (Prioritätsinversion)**
  - Balance Set Manager durchsucht die Bereit-Warteschlangen nach Threads, die sich mehr als 300 Prozessor-Ticks (~ 3-4 Sekunden) teilen
  - Die Priorität eines solchen Threads wird auf 15 gesetzt und sein Quantum verdoppelt
  - Nach Ablauf von zwei Quanten fällt die Priorität des Threads sofort auf die ursprüngliche Basispriorität
  - Wiederholung nach weiteren 300 Prozessor-Ticks

Fallstudie 3: BS 2000

- **16 Prozeßkategorien, 12 vom Systemverwalter frei vergebbar, 4 fest zugeordnet**
  - SYS für Systemprozesse
  - TP für Transaktionsverarbeitung
  - DIALOG für Teilnehmer-Programme
  - BATCH für Stapelaufträge
- **Aufträge vergleichbarer Wichtigkeit werden in einer Prozeßkategorie zusammengefaßt**
  - Jeder Kategorie wird ein nichtnegatives Gewicht W_i zugeordnet
  - Abrechnungspreis kann von Kategorie und Gewicht abhängen
- **Externe Prioritäten (EPRI)**
  - Prozessen können vom Benutzer externe Prioritäten zugeordnet werden und zwar von 0 (beste) bis 255.

0 - 30	System-Prioritäten,
31 – 127	gute feste Prioritäten,
128 - x	variable Prioritäten, x-vom Systemverwalter festgelegt
x - 255	schlechte feste Prioritäten (für Hintergrundbetrieb)

Fallstudie 3: BS 2000

- **Feste interne Prioritäten (IPRI)**
  - Aus den externen Prioritäten und den Kategoriegewichten werden feste interne Prioritäten von 1 bis 256 berechnet gemäß

$$IPRI = \begin{cases} 256 - EPRI & \text{falls } EPRI < 128 \\ & \\ WSUM + W_i & \text{(feste ext. Pr.)} \\ 1 + (256 - EPRI) * \dfrac{WSUM + W_i}{2 * WSUM} & \text{falls } EPRI > 127 \\ & \text{(var./feste ext. Pr.)} \end{cases}$$

  - WSUM die Summe der Gewichte über alle Kategorien
  - Bei IPRI und VPRI ist 1 die schlechteste Priorität.
- **Variable interne Prioritäten (VPRI) in Abhängigkeit von**
  - Zeit (ET = elapsed time), seit dem letzten Transfer des Prozesses in den Hauptspeicher
  - Prozessor-Zeit-Verbrauch (CPU) seit demselben Zeitpunkt
  - einem rechnermodellspezifischen Konvergenzfaktor CF bei Zeitscheibenende und bei bestimmten Anlässen

Fallstudie 3: BS 2000

$$VPRI = \begin{cases} IPRI & \text{falls EPRI} < 128 \\ \left(\min\left(IPRI * \frac{ET}{CPU*CF}, 127\right)\right) & \text{falls EPRI} > 127 \end{cases}$$

(feste ext. Pr.)

(var./feste ext. Pr.)

- **modifiziertes HPRN-Verfahren im unteren Prioritätenbereich (wegen ET = WT + CPU, WT = Wartezeit)**
- **Zuschläge zur Verbesserung der Priorität:**
  - beim Durchlaufen von Kern-Code im privilegierten Prozessorzustand bekommen Prozesse einen Zuschlag von 5 Einheiten
  - Ein rechnender Prozeß erhält einen Zuschlag von 5 Einheiten
  - Bei einer Wartezeit auf der Bereitliste > 1 Sekunde erhält ein Prozeß einen von VPRI abhängigen Zuschlag zwischen 13 (bei VPRI = 128) und 45 Einheiten (bei VPRI = 0) als Auffrischung; dies wirkt dem Verhungern unterprivilegierter Prozesse entgegen.
- **Bei Bedarf findet Verdrängung statt**

Einflußmöglichkeiten des Systemverwalters auf das Systemverhalten

- **Vergabe und Gewichtung der Prozeßkategorien**
- **Festlegung der Prioritätsgrenze x**
- **Festlegung des Konvergenzfaktors CF**
- **Festlegung des Zeitscheibenquantums**

- **Da die Zusammenhänge zwischen diesen Einflußgrößen (und noch weiteren für die mittelfristige Ablaufplanung) sehr kompliziert sind, hat man auch schon ein Expertensystem entwickelt, das die Wahl dieser Größen in Abhängigkeit von der aktuellen Systemlast optimal durchführt.**