Betriebssysteme

Prozessmanagement

erstellt durch:

Name:	Karl Wohlrab
Telefon:	09281 / 409-279
Fax:	09281 / 409-55279
EMail:	mailto: Karl.Wohlrab@fhvr-aiv.de

Anmerkungen

Bei dem vorliegenden Scriptum handelt es sich um ein Rohmanuskript im Entwurfsstadium. Das Script wird begleitend zur Lehrveranstaltung fortgeschrieben und überarbeitet.

Es erhebt keinen Anspruch auf Vollständigkeit und Korrektheit. Inhaltliche Fehler oder Ungenauigkeiten können ebenso wenig ausgeschlossen werden wie Rechtschreibfehler.

Inhaltsverzeichnis

1. Grundlagen zum Prozessmanagement

Literaturhinweise: [Tann02] Kap. 2, Seite 87 ff

Die zentrale Komponente in jedem Betriebssystem ist der Prozess. Alle weiteren Komponenten hängen von diesem Konzept ab. Für das grundlegende Verständnis der Arbeitsweise eines Betriebssystems und damit auch des gesamten Computers ist deshalb auch ein Grundverständnis über die Prozesse erforderlich.

Einen Prozess kann man wie folgt beschreiben:

Ein Prozess ist vom Grundprinzip her, ein Programm in Ausführung.

Diesem Prozess können nach Bedarf verschiedene Komponenten desBitmaps;einfügen und bearbeiten Rechensystems zugeteilt werden, um die ordnungsgemäße Programmausführung zu gewährleisten.

1.1 Das Prozessmodell

Heutige Computer sind in der Lage mehrere Aufgaben "gleichzeitig" zu bearbeiten, d.h. mehrere Programme gleichzeitig abzuarbeiten. Man spricht vom Multiprocessing oderBitmaps;einfügen und bearbeiten Multitasking – der Rechner bearbeitet "zeitgleich" mehrere Prozesse (bzw. Tasks).

In Wirklichkeit läuft natürlich jeweils nur ein Programm auf der CPU – die anderen müssen warten bis sie an die Reihe kommen. Man spricht hier von **"Quasiparallelität"**. EchteBitmaps;einfügen und bearbeiten Parallelität kann man nur in Systemen mit mehreren Prozessoren erreichen.

Im Detail sind hier verschiedene Betriebsarten zu unterscheiden, die zum Teil auch historisch zu sehen sind.

1.1.1 Einprogrammbetrieb

Der Einprogrammbetrieb ist eigentlich nur auf einem klassischen "von-Neumann"-Rechner zu finden, bei dem noch keine Unterscheidung mehrere Benutzer möglich war. Hier hatte der Anwender wirklich den Rechner für seine Arbeiten exklusiv zur Verfügung – ein Betriebssystem war zu diesem Zeitpunkt eigentlich noch gar nicht erforderlich.

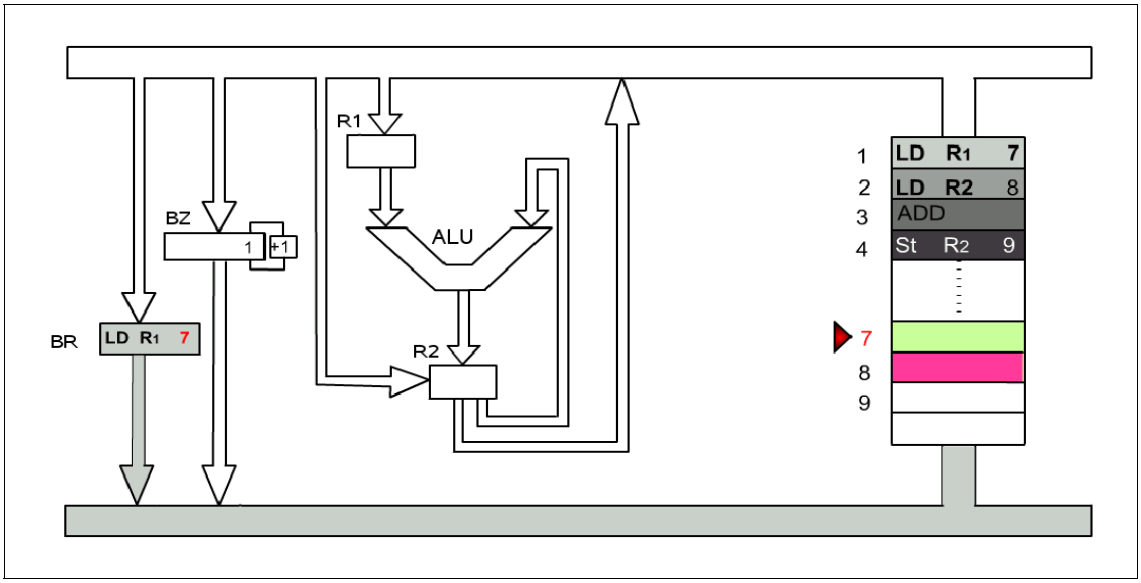

Abbildung 1: Ein einfaches Prozessormodell

1.1.2 Mehrprogrammbetrieb

Als mit zunehmender Leistungsfähigkeit der Rechner auch die Anforderung an eine bessere Auslastung der einzelnen (zum Teil sehr teueren) Komponenten entstand, mussten auf der gleichen Hardware mehrere Programme "gleichzeitig" ausgeführt werden. So konnte z.B. während auf der einen Seite die Druckausgabe für ein bereits abgearbeitetes Programm lief, schon auf der anderen Seite die Eingabe für das nächste Programm erfolgen, um den Durchsatz der Anlage zu verbessern.

Eine weitere Verbesserung des Durchsatzes konnte erzielt werden, indem man mehrere Eingabestationen parallel einsetzt (und sinngemäß auch die Ausgabestationen).

Spätestens jetzt war es erforderlich, die Aktivitäten der einzelnen Programme zu steuern und zu koordinieren. Da ja weiterhin nur eine CPU zur Verfügung stand, musste dafür Sorge getragen werden, dass jedes der Programme einen Anteil an der Nutzung der Ressourcen erhält, ohne dabei die anderen Programme zu sehr zu behindern. Deshalb wurden die Programme (und auch das Betriebssystem) in Form von **sequentiellen Prozessen** organisiert.

Konzeptionell besitzt jeder Prozess seine eigene (virtuelle) CPU; in der Realität schaltet natürlich die CPU zwischen den Prozessen hin und her. Dieses schnelle Hin- und Herschalten wird als **Multiprogramming** bezeichnet.

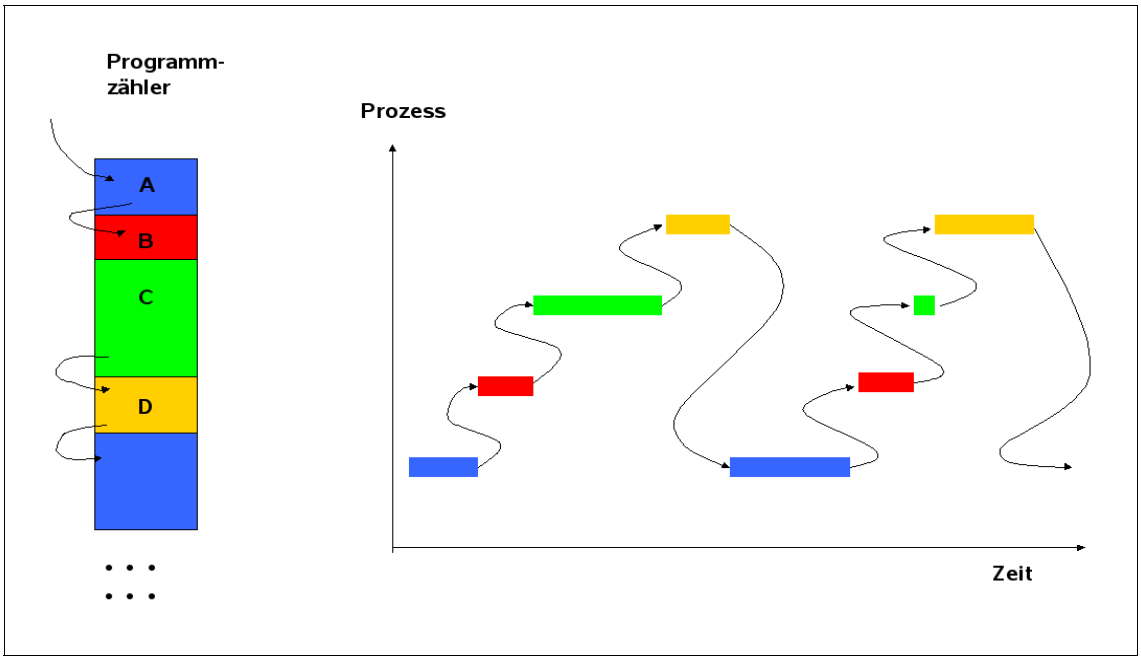

Abbildung 2: Grundprinzip des Multiprogramming

Das neue Verfahren "Multiprogramming" stellt jedoch an die Prozessverwaltung einige Herausforderungen:

Wenn ein Prozess den Zugriff auf die CPU abgibt (oder abgeben muss), ist dafür Sorge zutragen, dass der Prozess später an der Stelle fortsetzen kann, an der er unterbrochen wurde. Der CPU-Status (Befehlszähler und aktuelle Registerinhalte) müssen für die spätere Weiterbearbeitung gesichert werden. Jeder Prozess hat also seinen logischen Befehlszähler und seine logischen Register, die bei Bedarf vom Arbeitsspeicher in die CPU geladen bzw. entladen werden.

Dieses Verfahren hat für den Betrieb von Rechenanlagen entscheidende Vorteile. So können während Wartezeiten, während der z.B. ein Prozess auf einen Datentransfer von der Festplatte in den Arbeitsspeicher wartet andere Aufgaben (anderer Prozesse) in der CPU bearbeitet werden.

Zwangsläufig werden für die Aufgaben des Betriebssystems (was ja im Grunde auch nichts anderes als eine Menge von Programmen ist) ebenfalls entsprechende Prozesse bearbeitet.

Weitere Stichworte:

- Multitaskingbetrieb
- Multiuserbetrieb

1.1.3 Erzeugung von Prozessen

Literaturhinweise: [Tann02] Kap. 2.1.2, Seite 89 ff

In komplexeren Systemen wird ein Verfahren benötigt, um Prozesse je nach Bedarf zu erzeugen und (ggf. später auch wieder) zu beenden. Im Prinzip gibt es vier Ereignisse, die die Erzeugung eines Prozesses veranlassen:

- Initialisierung des Systems
- Systemaufruf zum Erzeugen eines Prozesses durch einen anderen Prozess
- Benutzeranfrage, einen neuen Prozess zu erzeugen
- Initiierung einer Stapelverarbeitung (Batch-Job)

Beim Booten eines Betriebssystem werden im Normalfall mehrere Prozesse erzeugt. Einige Prozesse laufen im Vordergrund - das sind Prozesse die mit Benutzern interagieren. Für systeminterne Arbeiten werden andere Prozesse - die Hintergrundprozesse erzeugt; diese führen spezielle Funktionen des Betriebssystems aus und lassen sich nicht bestimmten Benutzern zuordnen.

Hintergrundprozesse bearbeiten z.B. den Eingang von E-Mails, den Zugriff auf Web-Seiten oder auch die Verwaltung der unterschiedlichen Hardware-Ressourcen des Rechners. Ein typischer Hintergrundprozess ist auch der Scheduler der die Umschaltung der CPU auf die unterschiedlichen Prozesse koordiniert. In großen Systemen gibt es eine Vielzahl von Hintergrundprozessen die permanent aktiv sind.

Zusätzlich zu den, beim Systemstart erzeugt Prozessen können auch später weitere Prozesse erzeugt werden. So führen z.B. laufende Prozesse Systemaufrufe aus, um einen oder mehrere neue Prozesse zu erzeugen die ihnen bei der Verarbeitung Helfer. Dadurch kann die Arbeitslast eines Prozesses auf mehrere Prozesse verteilt werden.

Beispiel:

Beim Transfer großer Datenmengen über ein Netzwerk ist es hilfreich einen Prozess zu erzeugen der die Daten in einen gemeinsamen Puffer schreibt, während ein zweiter Prozess die Daten aus dem Puffer liest und weiter verarbeitet.

Eingaben eines Kommandos oder das Klicken eines Icons startet im interaktiven Systemen einen neuen Prozess der das gewählte Programm im Rahmen dieses Prozesses ablaufen lässt. In kommandobasierenden Systemen z.B. Unix auf denen X-Window läuft, übernimmt der Prozess das Fenster in dem er gestartet wurde. Bei Microsoft-Windows hat ein gestarteter Prozess kein Fenster er kann aber eines (oder auch mehrere) Fenster erzeugen. Beide Systeme können Benutzern mehrere Fenster in denen je ein Prozess läuft gleichzeitig geöffnet haben.

In den Stapelverarbeitungssystemen großer Mainframes können Benutzer Batch-Jobs an das System übertragen. Sobald das Betriebssystem entscheidet, dass genügen Betriebsmittel zu Verarbeitung der neuen Aufgabe zu Verfügung stehen erzeugt es einen neuen Prozess und bearbeitet darin die nächste Aufgabe auf der Warteschlange.

Im Grunde wird immer dann ein neuer Prozess erzeugt wenn ein bestehender Prozess einen Systemaufruf zur Erzeugung eines neuen Prozesses ausführt. Alle oben angeführten Beispiele werden per Systemaufruf verwirklicht.

Der Systemaufruf zur Erzeugung eines neuen Prozesses lautet unter Unix/Linux: **`fork`**; unter Windows gibt es hier für den Funktionsaufruf **`CreateProcess`**.

Sowohl in Unix als auch im Windows haben Vater und Kind nach einer Prozesserzeugung je einen eigenen getrennten Adressraum. In Unix ist der initiale Adressraum des Kind-Prozesses zwar eine Kopie des Adressraumes des Vater Prozesses es handelt sich dabei aber um zwei getrennte Adressräume. Änderungen die z.B. der Kind-Prozess in seinem eigenen Adressraum durchgeführt haben keine Auswirkung auf den Adressraum des Vater-Prozesses und umgekehrt.

1.1.4 Beendigung von Prozessen

Literaturhinweise: [Tann02] Kap. 2.1.3, Seite 91 ff

Nach seiner Erzeugung erledigt ein Prozess seine Aufgabe und nach Fertigstellung der Aufgabe terminiert der Prozess wieder.

Es sind vier verschiedene Bedingungen möglich unter denen ein Prozess terminiert:

- Freiwillige Terminierung nach Abschluss der zu bearbeitenden Aufgabe.
  Im Normalfall terminieren Prozesse freiwillig nach Beendigung ihrer Aufgabe. Die Terminierung teilt der Prozess dem Betriebssystem mit Hilfe eines System Aufrufs mit. Unter Unix heißt dieser Systemaufruf **`exit`**; unter Windows heißt der Systemaufruf **`ExitProcess`**.

- Freiwillige Terminierung auf Grund eines eigenständig erkannten Fehlers
  (z.B. Fehlende oder falsche Eingabedaten).

- Zwangsweise Terminierung auf Grund eines schwerwiegenden Fehlers
  Der Prozess wird hier im Regelfall vom Betriebssystem zwangsweise abgebrochen. Beispiel: Programmfehler oder Aufruf eines unzulässigen Befehls.

- Zwangsweise Terminierung durch einen anderen Prozess.
  Die zwangsweise Prozessbeendigung durch einen anderen Prozess wird ebenfalls mit Hilfe eines System Aufrufs durchgeführt. Der abbrechende Prozess weist das Betriebssystem mit Hilfe eines System Aufrufs (Unix: **`kill`**; Windows: **`TerminateProcess`**) an den abzubrechen der Prozess zu terminieren. Um diese Systemaufruf ausführen zu können ist eine entsprechende Berechtigung erforderlich – diese steht im Normalfall nur für ausgewählte Benutzer (z.B. Systemadministratoren) zur Verfügung.

1.1.5 Prozesshierarchien

Literaturhinweise: [Tann02] Kap. 2.1.4, Seite 92 ff

In manchen Systemen bestehen nach der Erzeugung eines neuen Prozesses zwischen Vater- und Kind-Prozess weitere Zusammenhänge. Wenn der Kind-Prozess weitere Prozesse erzeugt entsteht eine Prozesshierarchie.

In Unix bildet ein Prozess zusammen mit allen seinen Kindern und den weiteren Nachkommen eine Prozessfamilie.

Ein typisches Beispiel für die Rolle der Prozesshierarchien ist die Initialisierung eines Unix-Systems beim Systemstart:

- Ein spezieller Prozess mit dem Namen **`init`** ist im Boot-Image vorhanden.
- Der init-Prozess hat immer die Prozessnummer 1.
- Der init-Prozess liest aus der Datei **/etc/inittab**, wie viele Terminals es geben soll;
- danach erzeugt er pro Terminals einen neuen Prozess.
- Diese Terminal-Prozesse warten bis sich jemand am Terminal anmeldet (Login).
- Bei einer erfolgreichen Anmeldung führt der Login-Prozess eine Shell aus (z.B. die bash), um Kommandoeingaben annehmen zu können.
- Diese Kommandoeingaben können wiederum weitere Prozesse starten.
- … und so weiter.

Somit gehören alle Prozesse im gesamten System zu einem einzigen Baum mit dem init-Prozess an seiner Wurzel (ProcessID = 1).

Unter Windows gibt es kein Konzept einer Prozesshierarchie. Alle Prozesse sind gleichwertig. Bei der Erzeugung eines Kind-Prozess erhält der Vater-Prozess lediglich ein spezielles Token (Handle) um den Kind-Prozess zu steuern. Dieses Token kann auch an einen anderen Prozess weitergegeben werden - die Prozesshierarchie wird dadurch außer Kraft gesetzt – der Kind-Prozess wird quasi enterbt.

1.1.6 Prozesszustände

Bei der Verwaltung von Prozessen sind insbesondere drei Zustände (stark vereinfacht) zu unterscheiden. Die Zustände und die Möglichkeiten der Zustandsübergänge sind dem folgenden Zustands-Übergangsdiagramm zu entnehmen:

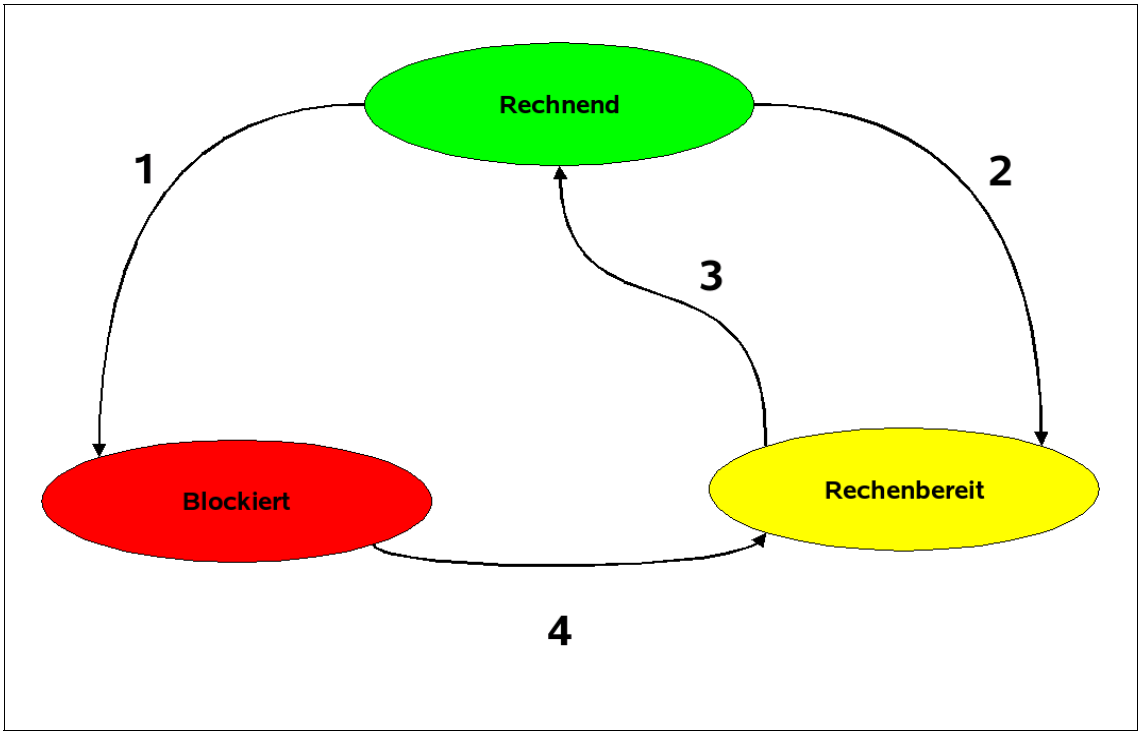

Abbildung 3: Prozesszustände und Zustandsübergänge

1.1.6.1 Status RECHNEND

- es kann sich immer nur genau ein Prozess ist im Status RECHNEND befinden (bei einer CPU).
- Dieser Prozess verfügt über all diejenigen Ressourcen (CPU, Hautspeicher, Daten, Programme, ...), die er für seine Arbeit benötigt
- fehlt den Prozess eine Ressource (z.B.: die vorhandenen Daten sind abgearbeitet und es werden neue Daten benötigt, die sich noch nicht im Hautspeicher befinden) wechselt der Prozess in den Status BLOCKIERT (Pfeil 1),
- Fall die Zeitscheibe des Prozesses abgelaufen ist obwohl der Prozess immer noch in der Lage wäre zu arbeiten, wechselt der Prozess in den Status RECHENBEREIT (Pfeil 2).

- Falls der Prozess durch einen Interrupt unterbrochen wurde ist er ja eigentlich noch rechenfähig, da er unmittelbar vor der Unterbrechung noch über alle erforderlichen Ressourcen verfügte. er wechselt somit direkt in den Zustand "RECHENBEREIT (Pfeil 2)

1.1.6.2 Status RECHENBEREIT

- Ein Prozess im Status RECEHENBEREIT hat alle für seine weitere Bearbeitung erforderlichen Ressourcen verfügbar (z.B.: Daten und Programmcode stehen im Hauptspeicher bereit)
- Der Prozess wartet in Status RECENBEREIT solange bis er die Ressource CPU zugeteilt bekommt – er nimmt quasi aktiv am Scheduling teil:
- Im Status RECHENBEREIT stehen im Regelfall mehrere wartende Prozesse
- Falls die CPU frei wird und einen der wartenden und rechenbereiten Prozesse zugewiesen wird, wechselt der betreffende Prozess in den Zustand RECHNEND (Pfeil 3)

1.1.6.3 Status BLOCKIERT

- Ein blockierter Prozess benötigt zur weiteren Bearbeitung noch verschiedene Ressourcen.
- Solange die benötigten Ressourcen nicht im erforderlichen Umfang zur Verfügung stehen, verbleibt der Prozess im Zustand BLOCKIERT.
- Wenn alle erforderlichen Ressourcen für einen Prozess bereitgestellt wurden, wechselt dieser Prozess in den Status RECHENBEREIT (Pfeil 4).
- Im Status BLOCKIERT befinden sich im Regelfall de meisten Prozesse.

Achtung!

Ein Prozess wird nie direkt aus dem Status BLOCKIERT in den Status RECHNEND wechseln – der Weg führt immer über den Status RECHENBEREIT.

1.1.7 Implementierung von Prozessen

Literaturhinweise: [Tann02] Kap. 2.1.6, Seite 95 ff

1.1.7.1 Die Prozesstabelle

Zur Realisierung des Prozessmodells und zur Verwaltung der Prozesse wird vom Betriebssystem eine Tabelle (die **Prozesstabelle**) verwendet.

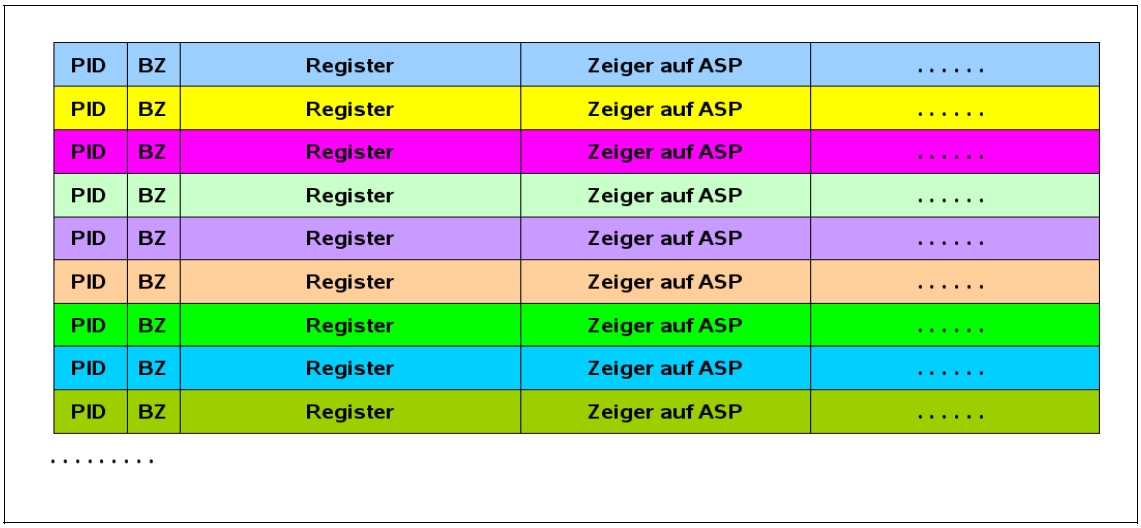

Abbildung 4: Prozesstabelle

Diese Prozesstabelle ist ein Feld von Datenstrukturen (**Prozesskontrollblock**).

- Für jeden Prozess besteht ein Eintrag in dieser Prozesstabelle.
- Dieser Eintrag enthält Informationen über den aktuellen Zustand des Prozesses, seinen Befehlszähler, seinen Stack-Pointer (Keller Zeiger), seine Speicherbelegung, den Zustand seiner geöffneten Dateien, Verwaltungs- und Scheduling-Informationen und sowie eine Vielzahl weiterer Informationen über den Prozess.
- Die Informationen des Prozesskontrollblocks müssen immer dann gespeichert werden wenn der Prozess vom Zustand ***rechnend*** in den Zustand ***rechenbereit*** oder in den Zustand ***blockiert*** übergeht. Wechselt der Prozess später wieder in den Zustand ***rechnend***, werden die gespeicherten Daten des Prozesskontrollblocks wieder an die entsprechenden Stellen (Register, etc.) der CPU geladen - der Prozess kann somit weiterlaufen als wäre er nie unterbrochen worden.

1.1.7.2 Der Prozesskontrollblock

Prozessmanagement	Speichermanagement	Dateiverwaltung
Prozess-ID (PID)	Anfang Textsegment	Wurzelverzeichnis
Elternprozess (PPID)	Anfang Datensegment	Arbeitsverzeichnis
Program-Counter	Anfang Kellersegment	Dateideskriptor
Register (alle)	...	Benutzer-ID
Prozessstatus		Gruppen-ID
Priorität		...
Kellerzeiger		
div. Signale		
Startzeit des Prozesses		
Benutzte CPU-Zeit		
Zeitpunkt nächster Alarm		
...		

Abbildung 5: Prozesskontrollblock

1.1.7.3 Unterbrechungen (Interrupts)

Die Prozesstabelle ist auch ein gutes Hilfsmittel, um konkurrierende Zugriffe auf Ein- und Ausgabegeräte (z.B. Diskettenlaufwerke, Festplatten, …) geordnet zu koordinieren. Mit jeder Klasse von Ausgabegeräten ist eine (feste) Speicherstelle verbunden. Häufig werden diese festen Speicherstellen am Ende des Arbeitsspeichers verwaltet. Dieser Bereich wird deshalb auch als **Interruptvektor** bezeichnet. Der Interruptvektor enthält die Adressen der Routinen zur Unterbrechungsbehandlung.

Beispiel:

Nehmen wir an, es läuft gerade der Benutzerprozess 3 während ein Festplatten-Interrupt auftritt. Dann wird der Prozesskontrollbock des Benutzerprozesses 3 durch die Unterbrechungs-Hardware in der Prozesstabelle gespeichert. Anschließend springt die Verarbeitung zu derjenigen Adresse, die im Festplatten-Interruptvektor spezifiziert wird. Von diesem Punkt an wird alles weitere durch die Software (die Routine zu Unterbrechungsbehandlung) ausgeführt.

Alle Unterbrechungen speichern zunächst die Register im Kellerspeicher, erst danach werden die Einträge in den Prozesskontrollbock der Prozesstabelle übertragen. Anschließend werden die Informationen, die durch die Unterbrechung auf dem Keller Speicher gesichert wurden dort wieder entfernt. Diese Abläufe sind meist in Maschinensprache oder Assembler programmiert

die Unterbrechungsverarbeitung erfolgt im Grunde auf der untersten Schicht des Betriebssystems.

1. Hardware sichert Befehlszähler und Registerinhalte im Kellerspeicher
2. Hardware holt neuen Befehlszähler vom Interruptvektor
3. Assemblerroutine speichert Register im Prozesskontrollblock
4. Assemblerroutine erzeugt neuen Kellerspeicher
5. **Unterbrechungsroutine wird abgearbeitet**
6. Scheduler sucht den nächsten zur Verarbeitung anstehenden Prozess
7. Assemblerroutine lädt Befehlszähler vom Prozesskontrollblock in die CPU
8. neuer aktueller Prozess wird gestartet

Abbildung 6: Ablauf der Interrupt-Behandlung

1.2 Prozeß-Scheduling

Literaturhinweise: [Tann02] Kap. 2.5, Seite 148 ff

In multiprogrammierbaren Rechnern kann es vorkommen, dass mehrere Prozesse zur selben Zeit die CPU nutzen möchten. Diese (gar nicht so seltene) Situation tritt immer dann ein, wenn zwei oder mehr Prozesse gleichzeitig rechenbereit sind. Falls nur eine CPU verfügbar ist, muss deshalb vom Betriebssystem entschieden werden, welcher Prozess als nächstes den Zugriff auf die CPU erhält. Diese Komponente des Betriebssystems die diese Entscheidung trifft ist der **Scheduler**; der Algorithmus der vom Scheduler verwendet wird heißt **Scheduling-Algorithmus**.

Zur Zeit der Stapelverarbeitungssysteme mit Eingabe von Lochkarte oder Magnetband, war der Scheduling-Algorithmus noch relativ einfach - es wurde einfach der nächste Job vom Band ausgeführt; jeder Job hatte den Rechner exklusiv für sich zur Verfügung - ein konkurrierender Zugriff auf Ressourcen wie z.B. der CPU konnte eigentlich gar nicht entstehen.

Erst mit Einführung von Time-Sharing-Systemen bei denen wirklich gleichzeitig mehrere (teilweise viele) Jobs laufen, wurde die Anforderung an den Scheduling-Algorithmus wesentlich komplexer. Mit Einführung der Dialogsysteme wurde diese Komplexität noch einmal um ein Vielfaches gesteigert.

In früheren Zeiten war die CPU Zeit eine sehr knappe (und teuere) Ressource, die es galt möglichst optimal auszunutzen; das bedeutet der Auslastungsgrad der CPU sollte möglichst hoch sein. Deshalb sollten natürlich möglichst viele aktive Prozesse im System sein, damit immer ausreichend Arbeit für die CPU zur Verfügung stand. Im Gegensatz dazu bedeutete ein hoher Auslastungsgrad der CPU, dass Anwendungsprozesse (z.B. Dialogprozesse) zum Teil sehr lange warten mussten, bis sie die CPU zugeteilt bekamen - lange Antwortzeiten waren die Folge. Das bedeutet das immer bessere und effizientere Scheduling-Algorithmen entwickelt wurden.

In einem neueren Computern wie z.B. auch den Personalcomputer ist im Normalfall ausreichend CPU Zeit verfügbar - die CPU ist normalerweise so leistungsfähig das ihre Leistungsgrenze durch einen einzelnen Benutzer nicht erreicht wird man sagt die CPU ist **idle**. Eine Scheduling-Strategie ist deshalb in einem normalen Standalone-PC nicht so sonderlich von Bedeutung. Erst durch die Vernetzung der Rechner mit der Verlagerung von Leistungen auf einen oder mehrere Server oder auch andere PCs erhielt der Scheduling-Algorithmus eine neue wichtige Bedeutung.

Beispiel:

> Bei einem normalen vernetzten PC (kein Server) laufen ebenfalls mehrere Prozesse die z.B. die Netzwerkkarte überwachen und bei Eintreffen einer Nachricht über das Netzwerk den gerade laufenden Prozess abbrechen und dafür sorgen, dass zunächst diese eingegangene Nachricht bearbeitet wird.
> Ebenso können vom Anwender parallel mehrere Prozesse gestartet werden:
> So kann zum Beispiel in einem Fenster ein größerer Kopiervorgang zwischen zwei Festplatten ablaufen,

während in einem anderen Fenster der Anwender einen Text bearbeitet
und in noch einen weiteren Fenster zum Beispiel eine Tabellenkalkulation gerade ausgeführt wird.

All diesen Prozessen muss der Zugriff auf die CPU ermöglicht werden ohne dass sich diese Prozesse gegenseitig allzu sehr behindert. Der Scheduling-Algorithmus muss dabei auch Sorge tragen das z.B. Benutzerprozesse vorrangig bearbeitet werden denn hier sind sehr kurze Antwortzeiten gefragt, wogegen andere Prozesse z.B. Versand von E-Mails ohne weiteres auch mit einigen Sekunden Zeitverzug abgearbeitet werden können ohne dass hier der laufende Betrieb nachhaltig beeinträchtigt wird.

1.2.1 Prozessverhalten

Durch Untersuchungen an unterschiedlichen Prozessen zeigte sich, dass bei fast allen Prozessen sich Zeiten mit sehr hoher Rechenlast mit Zeiten mit wenig Rechenlast (z.B. Zeiten für Ein-/Ausgabe) abwechseln

Beispiel:

Die CPU arbeitet eine gewisse Zeit ohne anzuhalten dann erfolgt ein Systemaufruf um Daten aus einer Datei zu lesen - während dieser Zeit hat die CPU keine Arbeit zu verrichten. Wenn der Systemaufruf beendet wird und die Daten z.B. von der Platte in den Arbeitsspeicher gelesen sind, kann der Prozess weiter arbeiten d.h. jetzt kommt wieder eine rechenintensive Phase. Diese Phase dauert wieder solange entweder weitere Daten benötigt werden oder z.B. Ergebnisse ausgegeben werden sollen.

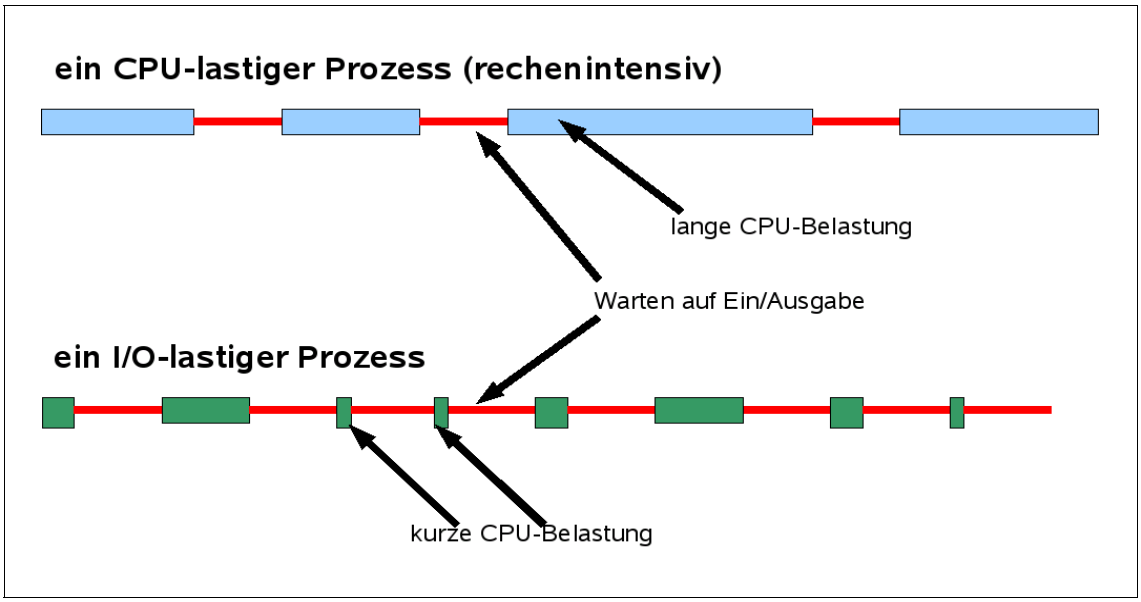

Abbildung 7: Prozessverhalten und CPU-Auslastung

Es zeigt sich noch jedoch dass es sehr unterschiedliche Prozesse gibt nämlich solche mit hoher und lang anhaltender CPU Belastung (rechenintensive Prozesse) und andere mit nur sehr kurzen CPU Belastungen und dafür langen Pausen für die Ein-/Ausgabe (I/O-intensive Prozesse). Mit der zunehmenden Leistungsfähigkeit der CPUs (schnellere Rechengeschwindigkeit) werden die Prozesse immer mehr von der wesentlich langsameren Ein-/Ausgabe abhängig - die Rechenleistung verliert der gegen immer mehr an Bedeutung.

Beim Scheduling müssen die unterschiedlichen Scheduling Algorithmen verschiedene Anforderungen berücksichtigen. Die Anforderungen sind nicht in allen Systemen gleich. Im Grunde sind drei verschiedene Umgebungsarten zu unterscheiden:

1. **Stapelverarbeitung (Batchbetrieb)**

2. **Interaktion (Dialogbetrieb)**

3. **Echtzeitverarbeitung**

In **Stapelverarbeitungssystemen** kommt es nicht auf eine sofortige schnelle Antwort (z.B. an den Benutzer der an einem Dialogterminal auf Antwort artet) an, sondern um die möglichst schnelle Bewältigung großer Datenmengen oder langwieriger Berechnungen. Das bedeutet es ist ohne weiteres möglich dass ein Prozess auch für längere Zeit die Ressource CPU benutzt ohne diese zwischenzeitlich wieder abzugeben. Da durch wird die Anzahl der Prozesswechsel verringert und die gesamte Verarbeitungsgeschwindigkeit etwas verbessert.

In Umgebungen mit **interaktiven Benutzern** ist dagegen ein häufiges Unterbrechen eines Prozesses notwendig, um z.B. auch die schnelle Beantwortung von Anfragen oder Aufträgen anderer Benutzer zu ermöglichen. Jede Interaktion eines Benutzers (z.B.: Tastendruck oder Mausklick) kann eine Anforderung an die CPU bedeuten.

In Systemen mit **Echtzeiteigenschaften** wird das Unterbrechen häufig gar nicht gebraucht denn die Prozesse müssen keine langen Zeiten arbeiten. Im Unterschied zu interaktiven Systemen arbeiten in Echtzeitsysteme nur Programme die bekannte Anwendungen darstellen. Echtzeitsysteme werden im Rahmen dieser Lehrveranstaltung jedoch nicht weiter betrachtet.

1.2.2 Wichtige Scheduling-Algorithmen für Stapelverarbeitungssysteme

1.2.2.1 First-Come-First-Served (FCFS)

Dies ist der einfachste aller Scheduling-Algorithmen der allerdings keinerlei Unterbrechungsmechanismen vorsieht. Hier wird die CPU den Prozessen in der Reihenfolge zugewiesen in der sie diese anfordern. Es ist im Grunde eine einzige Warteschlange mit rechenbereiten Prozessen; wenn der erste Prozess gestartet wird kann dieser so lange rechnen wie er will - erst wenn er terminiert kann der nächste Prozess gestartet werden.

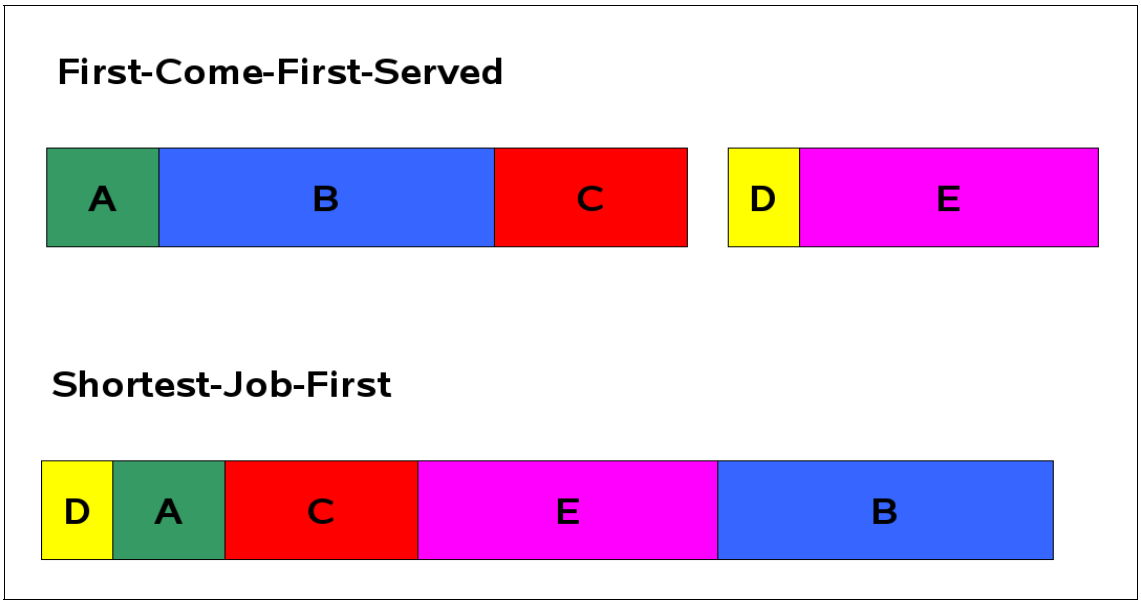

Abbildung 8: Scheduling-Algorithmen (Teil 1)

1.2.2.2 Shortest Job First (SJF)

Dieser Ansatz geht davon aus dass die Laufzeiten der Jobs vorher bereits bekannt sind. Bei der Abarbeitung startet der Scheduler dann zunächst denjenigen Job mit der kürzesten Laufzeit dann den nächst längeren und so weiter. Dieser Algorithmus arbeitet eigentlich nur dann optimal in alle Aufgaben gleichzeitig verfügbar denn nur dann kann ja der kürzeste Job ausgewählt werden.

1.2.2.3 Shortest Remaining Time Next

Dieser Algorithmus funktioniert ähnlich wie der oben beschriebene Algorithmus "Shortest Job First" - hier wird jedoch vom Scheduler immer der Prozess ausgewählt dessen verbleibende Zeit am kürzesten ist.

1.2.3 Scheduling in interaktiven Systemen

In interaktiven Systemen gibt es ganz andere Anforderungen an das Prozess-Scheduling (z.B.: kurze Antwortzeiten im Dialogbetrieb); deshalb wurden hierfür noch weitaus komplexere Scheduling-Algorithmen entwickelt. Selbstverständlich können diese Algorithmen auch in Stapelverarbeitungssystemen eingesetzt werden.

1.2.3.1 Round-Robin Scheduling

Round-Robin ist einer der ältesten, einfachsten und am häufigsten verwendeten Algorithmen. Jeder Prozess erhält einen immer gleich großen Zeitabschnitt für seinen Ablauf zur Verfügung. Dieser Zeitabschnitt wird auch als **Quantum** bezeichnet. Falls der Prozess am Ende des Quantums immer noch läuft wird die CPU unterbrochen und an einen anderen Prozess weitergegeben. Falls der Prozess während der Laufzeit blockiert oder vor Ablauf des Quantums beendet ist, wechselt die CPU natürlich sofort an einen anderen Prozess.

Man nennt das Verfahren häufig auch **Zeitscheibenverfahren** oder Time**-Sharing**

Die Round Robin Methode ist relativ leicht zu realisieren - der Scheduler muss nur eine Liste der lauffähigen Prozesse vorhalten diese Liste wird reihum abgearbeitet.

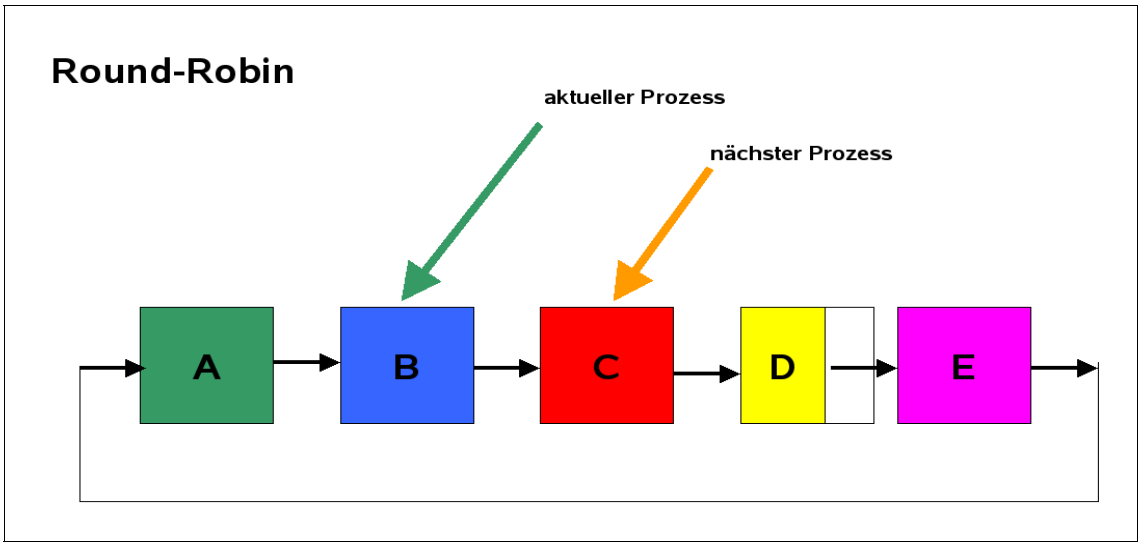

Abbildung 9: Scheduling-Algorithmen (Teil 2): Round-Robin

Einen großen Einfluss auf den Gesamtdurchsatz des Systems hat hierbei die Länge des Quantums. Bei jedem Prozesswechsel ist selbstverständlich auch ein Zeitbedarf für das Speichern bzw. Zurückladen der Registerinhalte und sonstiger Prozessinformationen mit einzuberechnen: je kürzer das Quantum desto häufiger muss der CPU Status entladen und wieder neu geladen werden. Das bedeutet bei zu kurzem Quantum und im Gegenzug bei zu großem Quantum ist mit einer unnötigen Verlängerung der Antwortzeiten zu rechnen. Zu

kurzes Setzen des Quantums verursacht zu viele Prozesswechsel und vermindert damit die CPU Effizienz; ein zu langes Setzen könnte andererseits schlechte Antworten erzeugen, da zu lange gewartet werden muss bis ein Prozess die CPU zurückerhält.
Hier ist ein vernünftiger Kompromiss erforderlich.

1.2.3.2 Prioritätenbasiertes Scheduling

Beim Round-Robin Scheduling geht man von der Annahme aus dass alle Prozesse gleich wichtig sind. Dies ist in der in der Realität häufig nicht so. Deshalb wird häufig ein sog. "**Prioritätenbasiertes Scheduling**" angewandt.

Jeder Prozess bekommt eine Priorität zugewiesen und der Prozess mit der höchsten Priorität bekommt den Zugriff auf die CPU.

> **Anmerkung:**
>
> **Auch bei PCs die nur von einem Benutzer benutzt werden kann dies Bedeutung haben, denn z.B. ein Dämon-Prozess der im Hintergrund E-Mails verschickt oder nur die Netzwerkschnittstelle überwacht, hat mit Sicherheit niedrigere Priorität als ein Prozess der in Echtzeit zur Darstellung von Videos auf dem Bildschirm verwendet wird.**

Damit Prozesse mit hoher Priorität nicht ewig laufen, kann der Scheduler die Priorität des gegenwärtigen Prozesses z.B. bei jedem Taktsignal um einen Schritt erniedrigen. Falls diese Aktion dazu führt, dass seine Priorität unter die des nächst wichtigen Prozesses fällt, kommt es zu einem Prozesswechsel. Eine andere Möglichkeit ist es jedem Prozess der laufen darf (das ist der Prozess mit der höchsten Priorität), ein maximales Zeitquantum zuzuweisen. Wenn dieses Quantum aufgebraucht ist, bekommt der Prozess mit der nächst höchsten Priorität den Zugriff auf die CPU. Ebenso kann ein Prozess in Wartezustand je Wartezyklus die Priorität um 1 erhöht bekommen.

Prioritäten können vom System auch dynamisch vergeben werden. Sehr Ein-/Ausgabelastige Prozesse, die eigentlich die CPU immer nur sehr kurz benötigen, würden z.B. mit sehr hoher Priorität laufen damit sie möglichst schnell die nächste Ein-/Ausgabeanfrage starten können; im Gegenzug würden sehr rechenintensive Prozesse eher mit niedriger Priorität laufen, damit durch sie die CPU nicht zu lange und zu oft belegt wird.

Unter Unix/LINUX gibt es den Befehl **nice**, der es einen Benutzer erlaubte freiwillig die Priorität seines Prozesses zu verringern (um nett (nice) zu den anderen Benutzern zu sein).

Häufig werden Prozesse nach ihrer Priorität in so genannte **Prioritätsklassen** zusammengefasst. Zwischen den Klassen wird prioritätsbasierendes Scheduling eingesetzt; innerhalb der Klassen die Round-Robin-Methode.

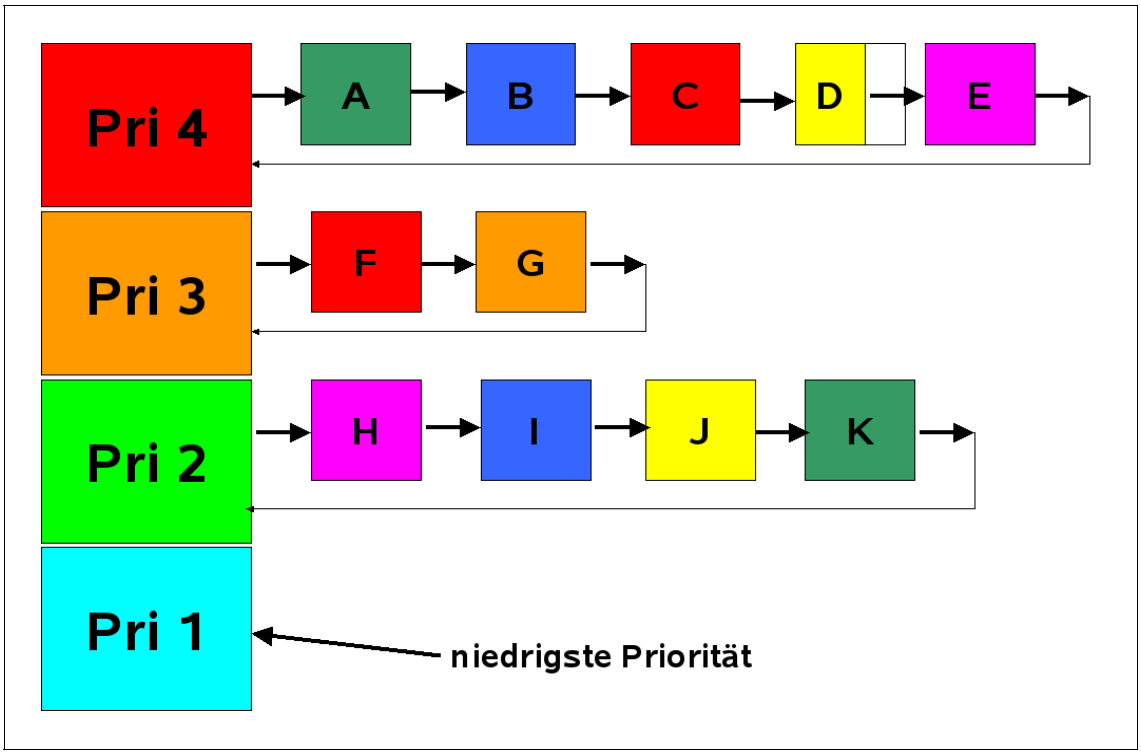

Abbildung 10: Scheduling-Algorithmen (Teil 3): Round-Robin mit Prioritäten

1.2.4 Scheduling in Echtzeitsystemen

Echtzeitsysteme sind entweder nur für genau eine Aufgabe entwickelt - hier hat Scheduling nahezu keine Bedeutung. In anderen Echtzeitsystemen (die häufig mehrere Aufgaben gleichzeitig verrichten sollen) ist dagegen Scheduling von sehr großer Bedeutung.

Da Echtzeitsysteme im Rahmen der vorliegenden Lehrveranstaltung nicht weiter betrachtet werden sollen, wird an dieser Stelle auf die Darstellung entsprechender Scheduling-Algorithmen verzichtet.

Warteschlangenverwaltung
...
...

1.3 Interprozesskommunikation

Literaturhinweise: [Tann02] Kap. 2.3, Seite 117 ff

wird später ergänzt

Stichworte:

- Zeitkritische Abläufe
- Kritische Bereiche (evtl. Verweis auf Deadlock-Möglichkeiten; s.u..)
- Wechselseitiger Ausschluss
- Sleep-and-awake-Verfahren
- Semaphore
- Monitore
- …
- …

2. Prozessmanagement unter LINUX

Jeder Prozess, der im Betriebssystemkern aktiviert wird, erhält eine eindeutige Kennzeichnung, die so genannte **Prozessnummer** (**PID** = Process-ID). Über diese kann jeder Prozess eindeutig identifiziert werden. Da die Prozessnummer eine positive Integer-Zahl ist, gibt es eine maximale Prozessnummer. Wird diese erreicht, beginnt die Zählung wieder von vorne, wobei noch existierende Prozesse mit niedriger Nummer übersprungen werden.

Ein neuer Prozess kann nur von einem bereits laufenden Prozess erzeugt werden. Dadurch werden, ähnlich wie beim Dateibaum, die einzelnen Prozesse im Betriebssystemkern in einer baumartigen, hierarchischen Struktur verwaltet. Jeder Kind-Prozess ist genau einem Eltern-Prozess untergeordnet. Ein Eltern-Prozess kann jedoch beliebig viele Kind-Prozesse besitzen. Die Wurzel der Prozessstruktur wird durch den Systemstart geschaffen und als init-Prozess (PID 1) bezeichnet.

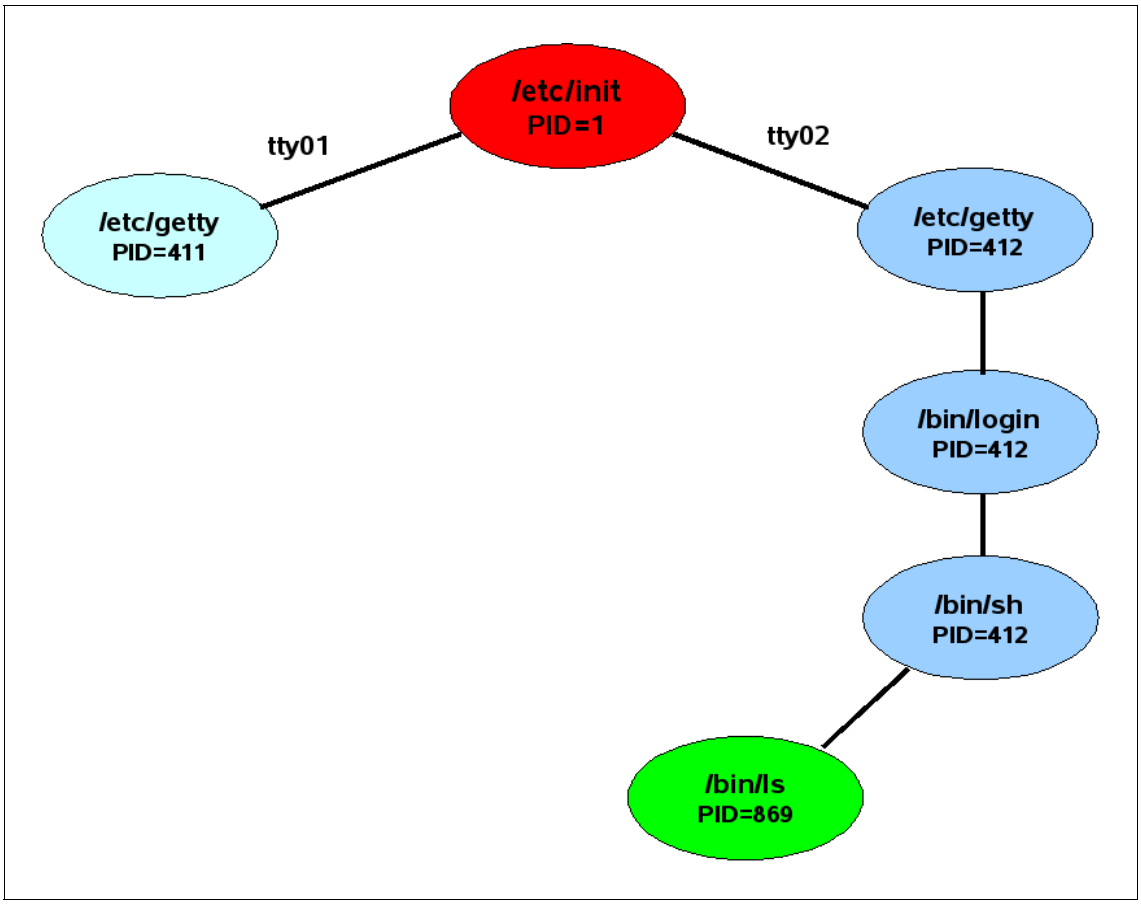

Abbildung 11: Prozessbaum unter LINUX

In der Regel wartet der Eltern-Prozess auf die Beendigung seiner Kind-Prozesse. Diese Art der Prozesssynchronisation wird als synchrone Ausführung bezeichnet, der Kind-Prozess wird als Vordergrundprozess ausgeführt. Bezogen auf einen Benutzer ist die Shell (Login-Shell) der Eltern-Prozess. Alle Kommandos, die der Benutzer startet, sind Kind-Prozesse. Während diese abgearbeitet werden ruht der Eltern-Prozess. Als asynchroner Prozess oder

Hintergrundprozess werden solche Prozesse bezeichnet, bei denen der Eltern-Prozess nicht auf das Ende seines Kind-Prozesses wartet, sondern parallel (quasiparallel auf einer Ein-Prozessor-Maschine) asynchron weiterläuft. Auf der Shell-Ebene kann jeder Prozess durch Anfügen von '&' (kaufm. UND) in der Kommandozeile als Hintergrundprozess gestartet werden.

Folgende Grundsätze gelten:

- Jeder Prozess kann beliebig viele neue Prozesse erzeugen
- Die Kindprozesse erben die gesamte Prozess-Umgebung (Home-Directory, Standard-Pfadnamen, Terminal-Typ, Systemvariablen, etc.)
- Eltern- und Kindprozesse laufen asynchron (Elternprozess kann auf Ende des/der Kindprozess(e) warten --> Synchronisation)
- Eltern- und Kindprozesse können miteinander kommunizieren
- Zwei Prozesse können miteinander kommunizieren und sich untereinander synchronisieren
- Ein Prozess kann einen Nachfolgeprozess starten: exec()
- 40 externe Prioritätsebenen, 2 interne Prioritätsebenen (System/User)

2.1 Der UNIX-Scheduler

UNIX ist ein Multitasking-Betriebssystem, d.h. mehrere Prozesse eines oder mehrerer Benutzer konkurrieren um die Vergabe der Rechenzeit des Prozessors. Wie viele andere Systeme auch, arbeitet UNIX nach dem Zeitscheiben-Prinzip (Time-Sharing in Kombination mit Prioritätenregelung)

Über einen Scheduling-Algorithmus zur Berechnung der Priorität erhält jeder einzelne Prozess einen bestimmten Teil der Rechenzeit zugewiesen. D.h. der Prozess mit der zur Zeit höchsten Priorität erhält die CPU, wird nach einen Zeitintervall suspendiert und, falls noch nicht beendet, zu einem späteren Zeitpunkt wieder reaktiviert. Die aktuelle Priorität eines Prozesses setzt sich aus dem Produkt des CPU-Faktors und der Grundpriorität zusammen.

Jeder Prozess besitzt eine bestimmte Priorität bzw. Rechenzeit. Diese Priorität verteilt Linux selber und im Normalfall haben alle Programme die Priorität 10. Die Priorität verteilt sich von -20 bis 19, dabei ist -20 die höchste Priorität und 19 die niedrigste Priorität[1]

1 Quelle: manpage zu "nice"

Im Normalfall sollte man an der Priorität nichts ändern, manchmal kann es dennoch nützlich sein.

Beispiel:
Im Hintergrund soll ein Backup-Programm laufen, die Arbeit im Dialogbetrieb soll aber nicht signifikant gestört werden. Dann kann man das Backup-Programm mit einer niedrigen Priorität starten:

```
nice -n 10 backup.sh      # Programm backup.sh mit der
                          # niedrigen Priorität 10 starten
```

Anzeigen kann man die Priorität mit

```
ps -l                     # Spalte NI = Priorität
```

oder

```
top                       # Spalte NI = Priorität
```

Mit dem Befehl `renice` kann root (sonst keiner) die Priorität eines laufenden Prozesses ändern. Auch darf nur root eine höhere Priorität als 0 erteilen.

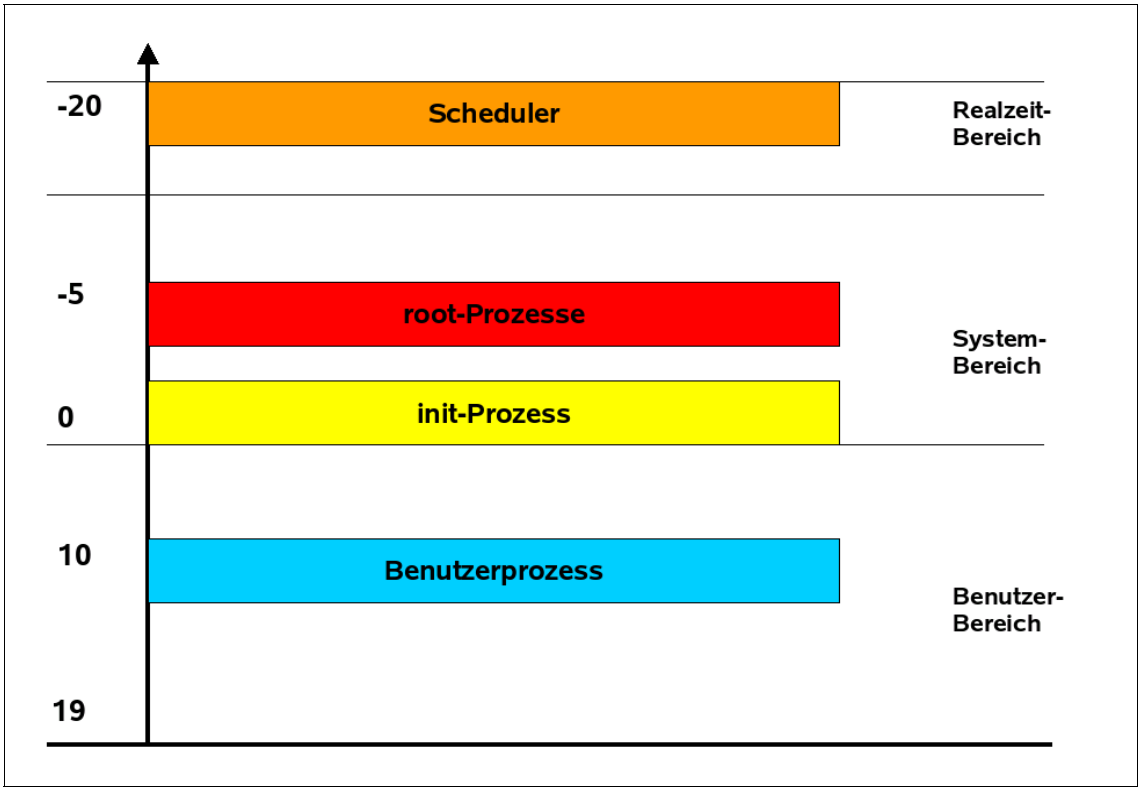

Abbildung 12: Prioritäten unter UNIX/LINUX

Die Prozessverwaltung und Prioritätssteuerung ist recht komplex. Hier deshalb nur kurze Stichpunkte:

- Round Robin mit Multilevel Feedback
- Neuberechnung aller Prioritäten einmal pro Sekunde
- Hohe CPU-Auslastung: Priorität sinkt
- länger nicht gerechnet: Priorität steigt

2.2 Systemaufrufe fork(), exec() und wait()

Diese Systemaufrufe haben mit der Generierung von Kindprozessen zu tun und erlauben die Synchronisation zwischen Eltern- und Kindprozessen.

2.2.1 fork()

fork() erzeugt einen Kindprozess, der ein vollständiges Abbild des Elternprozesses ist und der beim gleichen Stand des Befehlszählers fortgesetzt wird. Eltern- und Kindprozess wird jedoch die Möglichkeit geboten, festzustellen, ob es sich um Eltern- oder Kindprozess handelt: Der Kindprozess bekommt als Rückgabewert 0, der Elternprozess die PID des Kindprozesses. Durch bedingte Verzweigung nach dem Schema ("`.. if Elternprozess then ... else ...`") können beide Prozesse dann unterschiedlich weiterarbeiten.

- Etliche Systemprozesse schließen stdin, stdout und stderr, treten also in den Hintergrund. Solche Prozesse nennt man '**demon**' (Daemonprozesse).
- Terminiert der Elternprozeß vor dem Kindprozeß, wird dieser zum 'Waisenkind'. Normalerweise wird er dann vom Init-Prozeß 'adoptiert'.
- Hat der Kindprozeß dann auch noch den Kontakt zum Terminal (Standardausgabe und -eingabe) wird er zum 'Zombie'.

Einzelschritte beim Aufruf von fork():

1. Prozesstabelle überprüfen (Platz frei?)
2. Speicher für Kindprozess allokieren
3. Elternprozess-Speicher --> Kindprozess-Speicher kopieren
4. Prozesstabelleneintrag es Elternprozesses aktualisieren
5. PID für Kindprozess wählen, Kindp. in Prozesstabelle eintragen
6. Kernel und Dateisystem über Kindprozess informieren
7. Fertigmeldung an Eltern- und Kindprozess senden

2.2.2 wait()

wait() ermöglicht dem Elternprozess das Warten auf die Beendigung des/der Kindprozess(e). Der Elternprozess wird verdrängt und erst durch das Ende eines Kindprozesses wieder "aufgeweckt". Zur Unterscheidung mehrerer Kindprozesse liefert die Funktion wait() die PID des "gestorbenen" Kindprozesses zurück.

Gibt es keinen Kindprozess, ist das Ergebnis -1.

Beheben des Waisenkind/Zombie-Problems:

1. Elternprozess ruft wait() zu spät auf:
2. Beim Wait-Aufruf wird zuerst die Prozesstabelle nach terminierten Kindprozessen durchsucht und diese dann gelöscht.
3. Elternprozess ist terminiert:
4. Beim Terminieren des Elternprozesses werden dessen Kindprozesse zu Kindprozessen des Systemprozesses Init.

2.2.3 exec()

Bei **exec()** wird der ursprüngliche Prozess durch einen neuen Prozess ersetzt (eine Rückkehr zum aufrufenden Prozess ist daher logischerweise nicht möglich). exec() ist der Komplizierteste Aufruf, da der komplette Prozessadressraum ersetzt werden muss.

Dieser Aufruf ist auch als Kommando verfügbar (`exec [Programmname]`).

Der Ablauf im Schema:

1. Zugriffsrechte prüfen (Datei ausführbar?)
2. Größe der Speichersegmente feststellen
3. Aufrufparameter und Umgebung des Aufrufers festhalten
4. Speicher des Aufrufers freigeben, neuen Speicher allokieren
5. Neues Programm in den Speicher laden
6. UID-, GID-Bits bearbeiten
7. Prozesstabelle aktualisieren
8. Bereitmeldung an den Kernel senden

2.2.4 sleep()

Schließlich gibt es noch eine Systemfunktion, welche die zeitweise Blockierung eines Prozesses erzwingt: Mit **sleep()** kann ein Prozess für eine definierte Zeit "eingeschläfert" werden. Nach Ablauf der vorgegebenen Zeit, wird der Prozess wieder auf "bereit" gesetzt und kann weiterlaufen.

Auch sleep() ist als Kommando verfügbar (`sleep [Zeit in Sekunden]`).

2.3 Die Programme init und getty

Nach dem Start des Rechners und Laden von UNIX starten als erstes:

1. der Scheduler mit der **Prozessnummer 0** und
2. das Programm **init** mit der **Prozessnummer 1**

Es gilt grundsätzlich:

- Alle Prozesse, die auf dem Rechner laufen sind Kindprozesse von **init** (wobei "Kind" hier auch für alle weiteren Nachkömmlinge steht).
- Im Single-User-Modus werden von **init** nur noch einige Prozesse gestartet, im Multi-User-Modus sind wesentlich mehr Prozesse zu aktivieren (z.B. der Drucker-Spooler, die Abrechnung, etc.).
- Alle zu startenden Prozesse sind in der Datei /etc/inittab aufgeführt; **init** holt sich die Informationen über zu startende Prozesse aus der Datei /etc/inittab.
- Für uns ist eigentlich nur das Programm getty interessant, denn dieses Programm sorgt für den Kontakt mit den Terminals.

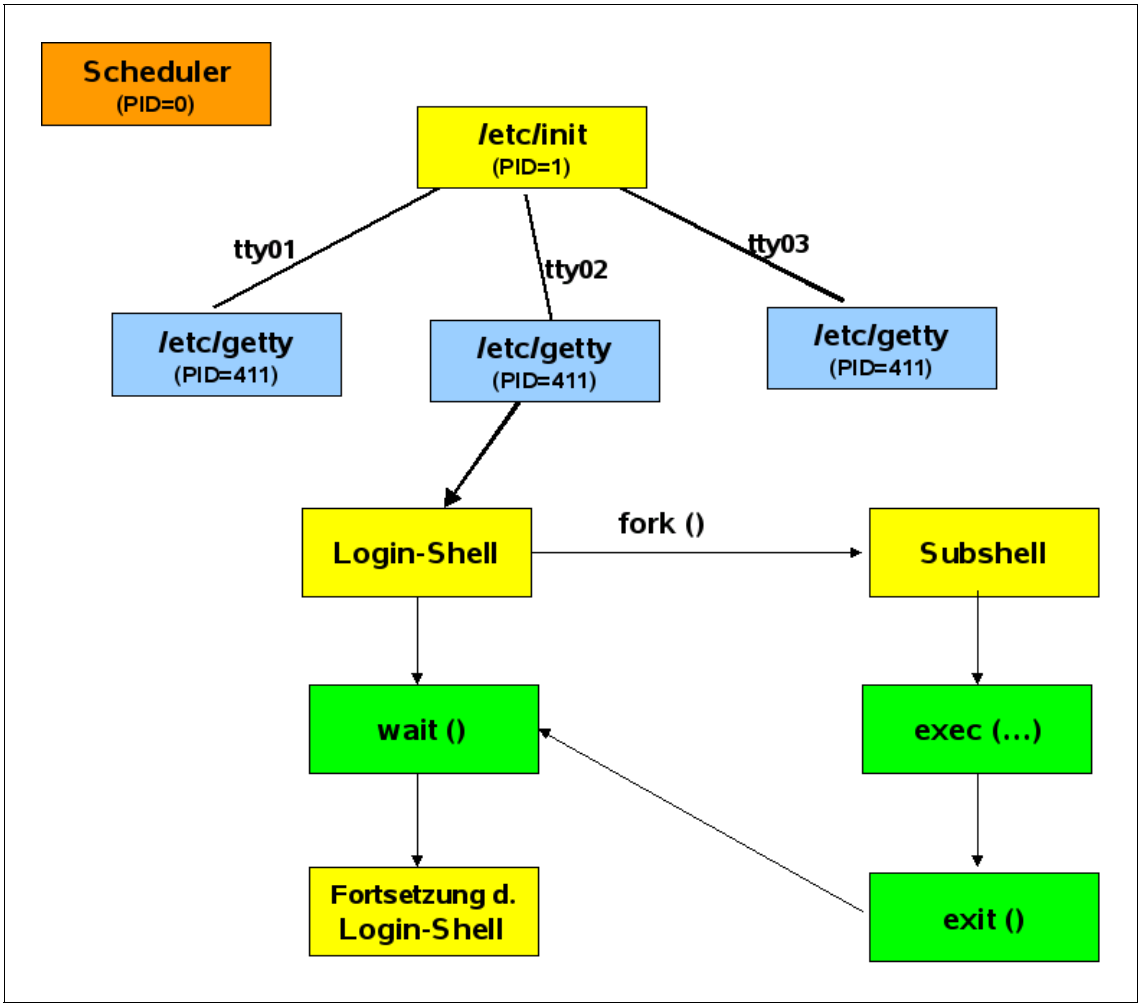

Abbildung 13: Prozessmanagement unter LINUX

Was geschieht nun weiter?

- getty wird von **init** für jedes angeschlossene Terminal gestartet.
- getty wartet, bis das Terminal eingeschaltet wird, liest den Usernamen ein und ruft dann mittels exec das login-Programm auf, das seinerseits das Passwort einliest.
- Wenn sich nun ein Benutzer anmeldet, wird das in der Benutzerliste spezifizierte Programm per exec aktiviert (in der Regel der Kommandointerpreter, die Shell).
- Die Shell (oder das anstelle der Shell aufgerufene Programm) arbeitet, bis es terminiert (bei der Shell durch Eingabe von EOF = CTRL-D). Da auch die Shell immer noch Kindprozess von **init** ist, erhält nun **init** ein Signal, dass der Kindprozess terminiert ist (siehe fork).
- Nun "wacht **init** auf", stellt fest, welcher seiner Kindprozesse terminiert ist und durchsucht die Datei `/etc/inititab` - uns interessiert wieder nur getty.

- In `/etc/inititab` steht beim Prozess getty die Steuerinformation respawn (spawn = "laichen"). Das heißt für **init**, dass dieser Prozess nach seiner Terminierung neu gestartet werden soll - und genau das geschieht.

2.4 Einflussnahme des Benutzers auf Prozesse

- Der Benutzer kann Prozesse erzeugen
- Der Benutzer kann eigene Prozesse jederzeit terminieren
- Prozesse können "im Hintergrund" laufen (mit hoher oder niedriger Priorität)
- Prozesse können als Batch (d.h. ohne Bindung an ein Terminal) laufen
- Prozesse können regelmäßig zu bestimmten Zeiten gestartet werden (z. B. jede Nacht, jede Woche, etc.)
- Prozesse können zu einem bestimmten Zeitpunkt gestartet werden

2.5 Zeitgesteuerte Prozessausführung

Aufgaben, die zu bestimmten Zeiten ausgeführt werden sollen, müssen vom System zeitlich gesteuert werden. Dies könnten beispielsweise Datensicherungen oder große Druckaufträge sein. Oft handelt es sich um Vorgänge, die sehr viel Systemleistung benötigen oder zumindest eine Ressource das System dermaßen stark in Anspruch nehmen, dass ein vernünftiges Arbeiten der übrigen Benutzer nicht mehr möglich ist. Denkbar sind auch Sicherungen, die außerhalb der üblichen Benutzungszeiten durchgeführt werden müssen, um Inkonsistenzen zu vermeiden.

Linux bietet zu diesem Zweck drei Programme an, die in der Lage sind, solche Aufgaben zeitgesteuert abzuarbeiten.

at	für die Ausführung von einmaligen Jobs
batch	für das Abarbeiten von Jobs, die nicht an einen festgelegten Zeitpunkt gebunden sind, sondern sich nach der Belastung der Systemressourcen richten
cron	für Aufgaben, die wiederholt auftreten

2.5.1 Der Befehl "at"

Die Syntax lautet wie folgt.

```
at [-l -d <jobnummer> -f <dateiname>] zeit
```

Das Kommando at kennt noch einige weitere Optionen siehe man-Pages.

Alternativ zu **`at -l`** kann man auch das Kommando **`atq`**,
zu **`at -d`** kann auch das Kommando **`atrm`** verwendet werden.

Nach der Festlegung der Zeit wird der Anwender aufgefordert, die gewünschten Aktionen anzugeben. Sind alle Anweisungen eingegeben worden, so kann Sie die Eingabe mit Strg + D beendet werden.

2.5.2 Der Befehl "batch"

Dieser Befehl entspricht in seiner Anwendung dem bereits besprochenen Befehl at, nur dass hierbei auf eine Zeitangabe verzichtet werden kann, da die eingegebenen Kommandos ausgeführt werden, wenn das System nur wenig belastet wird.

2.5.3 Prozessteuerung mit cron

Es gibt Programme, die sollten in regelmäßigen Abständen laufen, z.B. um das Postfach auf neu ankommende Mail zu überprüfen. Hierfür existiert ein eigener Dämon **`crond`**, der alle in bestimmten Tabellen aufgeführten Maßnahmen zu gegebener Zeit veranlasst

Für den Systemverwalter besteht die Möglichkeit, bestimmte wiederkehrende Verwaltungsaufgaben in die Datei **`/etc/crontab`** einzutragen:

Zu Beginn einer solchen **`crontab`** können bestimmte Einstellungen vorgenommen werden, die die Ausführung der in der Tabelle enthaltenen Kommandos steuern. Im Beispiel werden die Shell, die zur Interpretation der Kommandos genutzt wird, die Suchpfade nach ausführbaren Programmen und die Ausgabeumleitung festgelegt

Es folgen die Einträge der Art "Wann ist was zu tun", wobei 5 Felder das "Wann" beschreiben und der Rest das "Was". Schauen wir uns die Zeitfelder genauer an; von links nach rechts besitzen sie folgende Bedeutung

Feld	Zeit	Zulässige Werte
1	Minute	0-59
2	Stunde	0-23
3	Tag	1-31
4	Monat	1-12, jan, feb, ..., dec
5	Wochentag	0-7, sun (entspricht 0 oder 7), mon, ..., sat

Jedes Feld erlaubt auch die Eingabe mehrerer Werte. Die nachfolgende Tabelle fasst die verschiedenen Syntaxvarianten zusammen, die Beispiele beziehen sich - sofern nicht anders angegeben- auf Minuten:

	Syntax	Beispiel/Bemerkung
Voller Bereich	*	0, 1, 2, ..., 59
Ausgewählte Bereiche	1-5	1, 2, 3, 4, 5
Liste	2,3,11,12	Nur an den angegebenen Werten
	2,3,30-40	Kombination aus Liste und Bereich
Schrittweite	*/2	aller zwei Minuten (0, 2, ..., 58)

Tag, Wochentag und Monat können auch als englische Namen (die ersten drei Buchstaben) angegeben werden. Klein- und Großschreibung werden dabei nicht unterschieden. Bereiche sind bei der Verwendung von Namen nicht erlaubt

Das Editieren der **/etc/crontab** ist selbstverständlich nur dem Systemadministrator gestattet, dem gewöhnlichen Nutzer steht das Kommando **crontab** zur Verfügung, das ihm eine eigene Tabelle im Verzeichnis **/var/crond/tabs** unter seinem Nutzernamen generiert.

Ein Nutzer kann mit

```
crontab -e
```

seine eigene crontab-Datei editieren und mit

```
crontab -l
```

die Einträge betrachten. Ein direktes Editieren ist nicht möglich, da die Datei root gehört!

Beispiel:

Um zum Beispiel den Feierabend am Rechner nicht zu verschlafen hilft:

```
crontab -e
no crontab for user - using an empty one
…
0 20 * * 1-5 echo -e "E.T. will nach Hause \a"
…
crontab: installing new crontab
```

Damit wird an jedem Wochentag um 20.00 Uhr an den rechtzeitigen Abschied erinnert, indem ein Piepton erklingt und wir vom **crond** eine Mail erhalten.

Wir überzeugen uns noch schnell vom richtigen Eintrag:

```
crontab –l
# DO NOT EDIT THIS FILE - edit the master and reinstall.
# (/tmp/crontab.1491 installed on Fri Nov 13 13:30:46 1998)
# (Cron version -- $Id: crontab.c,v 2.13 1994/01/17
03:20:37 vixie Exp $)
0 20 * * 1-5 echo -e "E.T. will nach Hause \a"
```

2.6 Prozesskommunikation und -Synchronisation

Zur Kommunikation zwischen Prozessen und der Synchronisation der Prozessaktivitäten gibt es in UNIX unterschiedliche Konzepte:

- **Dateien**
  Ein oder mehrere Prozesse schreiben in Dateien, die von anderen Prozessen gelesen werden (wenig effizient).

- **Pipes (Datenkanäle)**
  Eine Pipe ist ein FIFO-organisierter Speicher, in den ein Prozess schreibt und aus dem ein anderer Prozess liest (FIFO = first in, first out). Pipes werden wie Dateien angesprochen, sind jedoch in der Regel als Pufferbereich im Hauptspeicher organisiert (auf Datei muss erst ausgewichen werden, wenn der Puffer überläuft).

- **Signale**
  sind Software-Interrupts, die asynchron auftreten.
  Hauptsächlich zur Kommunikation zwischen Benutzerprogramm und den BS.
  Auslösung z. B. durch:

  - Aktionen des Benutzers (CTRL-C)
  - Programmfehler
  - durch andere Prozesse Im Prozess muß explizit festgelegt werden, welche Aktion auf ein bestimmtes Signal erfolgen soll:
  - Ignorieren
  - Bearbeiten durch eine Service-Routine

  Ist nichts Entsprechendes definiert, wird der Prozess abgebrochen.

- **Named Pipes**
  Bei einfachen Pipes gelten zwei Einschränkungen:

  - Lebensdauer an die Lebensdauer der beteiligten Prozesse gebunden
  - Kommunikation nur für Prozesse mit gemeinsamen Elternprozeß oder zwischen Eltern- und Kindprozeß

  Named Pipes werden als Spezialdateien angelegt (siehe Dateitypen), die sich wie Gerätedateien verhalten und beliebig lange leben können. Named Pipes können nur einmal gelesen werden und natürlich arbeiten sie als FIFO.

- **Message Queues**
  dienen dem Austausch von strukturierten Nachrichten über eine Dienstleistung des BS.

- **Semaphore**
  sind Zustandsvariablen als elementarer Mechanismus zur Synchronisation von Prozessen.

- **Shared Memory**
  ist ein gemeinsamer Datenbereich im Hauptspeicher, der von zwei Prozessen genutzt werden kann (sehr viel effizienter als Dateien).

3. Prozessmanagement unter WINDOWS

wird bei Bedarf später ergänzt

4. Abbildungsverzeichnis

Abbildungsverzeichnis