Prozesse

Hans Jürgen Ohlbach

20. November 2017

Keywords: Prozesse, Speicherverwaltung, Prozesskommunikation, Pipes, Sockets, Race Conditions, Deadlocks

Empfohlene Vorkenntnisse: Prozessorarchitektur

Inhaltsverzeichnis

1 Einführung

In den ersten Computern wurden Programme nacheinander gestartet. Wenn ein Programm fertig war, konnte das nächste gestartet werden. Sowohl für die Benutzer, als auch für die Computer selbst ist das nicht sehr günstig. Wenn ein Programm sehr lange rechnet, dann muss der Benutzer des nächsten Programms sehr lange warten, bis sein Programm an der Reihe ist. Für den Computer kann es bedeuten, dass viele Hardwarecomponenten u.U. die meiste Zeit unbeschäftigt sind. Wartet das Programm z.B. auf eine Benutzereingabe, dann kann der Prozessor nichts tun bis die Eingabe ankommt.

In modernen Computern können dagegen mehrere Programme quasi gleichzeitig laufen (*Multiprogramming*). Jedesmal wenn ein Programm gestartet wird, erzeugt das Betriebssystem daher daraus einen *Prozess*. Wird das gleiche Programm mehrfach gestartet, dann existieren auch gleichzeitig mehrere Prozesse dafür.

Für den Benutzer sollte es so aussehen, dass alle gleichzeitig laufenden Prozesse tatsächlich auch gleichzeitig rechnen. Das ginge aber nur wenn es auch so viele Prozessoren wie Prozesse gäbe. Da das i.A. aber nicht der Fall ist, wird der Effekt durch *Pseudoparallelität* erreicht. D.h. die einzelnen Prozesse dürfen nur für ganz kurze Zeit rechnen, dann werden sie abgebrochen (suspendiert), und ein anderer Prozess darf rechnen. Durch den häufigen Wechsel wird der Anschein von echter Parallelität erweckt. In Mehrprozessorsystemen werden die Prozesse dann auf die verschiedenen Rechenkerne verteilt, so dass zumindest einige Prozesse echt parallel rechnen.

Prozesse können intern mehrere parallel ablaufende Programmteile haben, sog. *Threads*. Die Threads arbeiten das gleiche Programm ab, allerdings durch eigene Programmzähler an jeweils verschiedenen Stellen. Da es zu den Threads auch recht viel zu sagen gibt, wird ihnen ein eigenes Miniskript gewidmet. Allerdings sind etliche Phänomene bei parallel laufenden Threads ähnlich oder gleich wie bei parallel laufenden Prozessen. Dazu gehören insbesondere Race Conditions und Deadlocks (Kap. 8).

Damit mehrere Prozesse, egal ob vom gleichen oder verschiedenen Programmen, gleichzeitig laufen können, müssen eine Reihe von Fragen beantwortet werden:

- Welche Informationen zu jedem Prozess müssen verwaltet werden?
- Wie wird ein Prozess erzeugt?
- Wie verläuft die Lebensdauer eines Prozesses?
- In welchen Bereich des Arbeitsspeichers wird ein Prozess geladen?
- Was passiert, wenn nicht genug Platz im Arbeitsspeicher ist?
- Wie teilen sich die Prozesse die verfügbare Kapazität der Prozessoren auf?
- Wie können Prozesse miteinander kommunizieren?
- Was passiert, wenn mehrere Prozesse gleichzeitig auf dieselben Ressourcen zugreifen wollen?
- Was darf ein Prozess tun, und was nicht?

In diesem Miniskript gehen wir davon aus, dass ein compiliertes Programm fix und fertig als lauffähiger Objektfile vorliegt. Er kann nach dem Laden unmittelbar ausgeführt werden. In der Praxis ist das nicht so. Auf dem Weg von einem Quellcode bis zum lauffähigen Maschinencode sind noch etliche Probleme zu lösen, die im Miniskript zur Prozesserzeugung beschrieben werden. Dazu gehören insbesondere das separate compilieren von Programmteilen, das Einbinden von statischen und dynamischen Bibliotheken sowie von Plugins, und die gleichzeitige Verwendung desselben Codes in verschiedenen Prozessen.

2 Prozesstabelle und Prozesskontrollblock

Jedes Betriebssystem verwaltet die gerade aktiven Prozesse in einer *Prozesstabelle*. Die Prozesstabelle ist eine Liste aller *Prozesskontrollblöcke*. Jeder Prozesskontrollblock enthält alle Informationen, die das Betreibssystem haben muss, um den Prozess verwalten zu können. Das sind insbesondere

- der Eigentümer des Prozesses,
- die *ProzessId*, d.h. eine eindeutige Nummer für den Prozess,
- der momentane *Prozesszustand* (siehe Kap. 4),
- die Registerbelegungen wenn der Prozess gerade nicht arbeitet.
- der Stackpointer. Jeder Prozess benötigt ein Stack, in dem die Daten beim Aufruf von Unterprogrammen gespeichert werden. Der Stackpointer zeigt auf den *top of stack*;
- die Speicherbelegung.

Zu den einzelnen Datenblöcken, insbesondere der Speicherbelegung gibt es im folgenden eigene Abschnitte.

3 Erzeugung von Prozessen

Sobald der Rechner gebootet ist, und das Betriebssystem läuft, gibt es schon eine Reihe von Prozessen. Wie man dann neue Prozesse erzeugt, hängt im Detail vom Betriebssystem ab.

In Windows gibt es die Funktion `CreateProcess`, und Variationen davon. Ruft man in einem *Vaterprozess* diese Funktion auf, dann wird ein *Kindprozess* erzeugt. CreateProcess hat eine Reihe von Parametern, mit denen man steuern kann, was der Kindprozess alles von seinem Vaterprozess erbt.

In Unix und Linux gibt es das Kommando `fork`, welches aus einem *Vaterprozess* einen zunächst identischen *Kindprozess* kopiert. Ein typisches Codefragement dazu ist:

```
...
pid_t $child = fork();
if($child == 0)
   {             # Kindprozess
    ...          # weiter im Kindprozess
    exit(0);     # Ende des Kindprozesses
}
else {           # Vaterprozess
     ...         # weiter im Vaterprozess
     }
```

`fork()` verdoppelt zunächst den Prozess. Im Kindprozess liefert die Funktion fork als Wert die Zahl 0. Im Vaterprozess liefert sie stattdessen eine Nummer, die ProzessId. Durch die if-Abfrage kann man dann erreichen, dass beide Prozesse unterschiedlich weiter machen. Ein Kindprozess kann natürlich wiederum „forken“, so dass man einen „Enkelprozess“ bekommt. Damit kann man eine ganze Prozesshierarchie erzeugen.

Eine typische Anwendung wäre ein Serverprogramm, welches unterschiedliche Clients bedienen soll. Sobald ein neuer Client erscheint, wird ein Kindprozess erzeugt, der diesen Client bedient. Der Vaterprozess kann in der Zwischenzeit auf weitere Clients warten. Damit kann man mehrere Clients parallel bedienen. Allerdings würde man das heute nicht mehr so machen, sondern im Prozess mehrere Threads erzeugen.

Einen Prozess komplett zu verdoppeln wäre natürlich sehr aufwendig. Konkret wird aber nur der Prozesskontrollblock verdoppelt. Einzelne Bereiche der Daten werden erst dann verdoppelt, wenn sie von Vater- oder Kindprozess überschrieben werden (siehe Abschnitt *Copy On Write* im Miniskript zur Prozesserzeugung).

Um ein komplett neues Programm zu starten, gibt es in Unix und Linux die Familie der `exec` Funktionen. Mit exec wird der aktuelle Prozess vollständig durch den neuen Prozess ersetzt. Will man im aktuellen Prozess zusätzlich ein neues Programm starten, dann kopiert man den aktuellen Prozess zunächst mit fork, und ruft im Kindprozess dann exec auf. Dann laufen der Vaterprozess und der neue Prozess parallel weiter.

4 Prozesszustände

Ein Prozess wird nicht einfach gestartet, arbeitet dann eine Weile und ist dann fertig. Wenn man genauer hinsieht, kann man eine Reihe von Zwischenzuständen beobachten, in denen sich der Prozess zeitweise befinden kann:

new: Der Prozess wird gerade geladen und seine Verwaltungsdaten werden berechnet.

ready: der Prozess kann loslegen. Er wartet, dass er an die Reihe kommt.

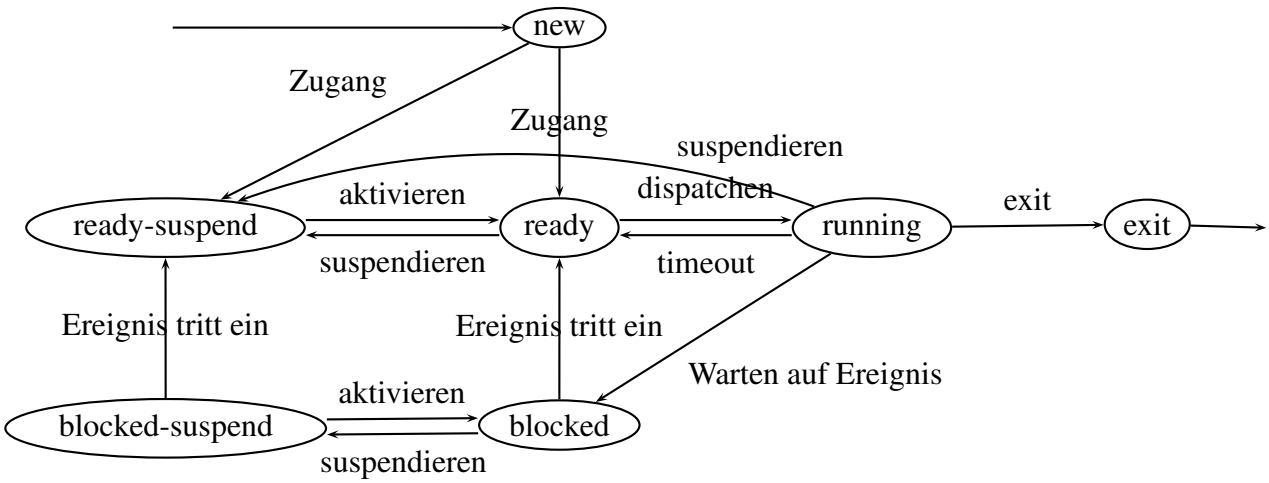
running: der Prozess ist vom sog. *Scheduler* ausgewählt worden, und vom sog. *Dispatcher* aktiviert worden. Er arbeitet jetzt.

blocked: der Prozess wartet auf ein externes Ereignis, im Moment kann er nicht weitermachen.

suspend: andere Prozesse sind im Moment dringender. Daher wird der Prozess unterbrochen bis er weiter machen darf. Seine Daten werden u.U. auf die Festplatte ausgelagert.
Es kann auch passieren, dass der Prozess, wenn er ready oder blocked ist, auf Festplatte ausgelagert werden muss, um Platz zu schaffen. Dann ist er im Zustand ready-supend oder blocked-suspend.

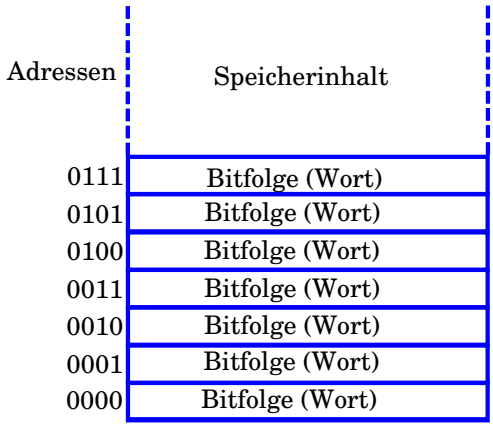
exit: der Prozess ist fertig. Alle Ressourcen, die er beansprucht hat, z.B. offene Dateien, werden frei gegeben.

Die Zustände mit ihren Übergängen sind in folgendem Diagramm aufgezeichnet:



5 Speicherverwaltung

Der Programmcode und die Daten für einen Prozess liegen zunächst auf einem externen Speicher; das ist meistens eine Festplatte. Wenn der Prozess gestartet wird und laufen soll, dann muss der Code und die Daten in den Arbeitsspeicher geladen werden. Der Arbeitsspeicher besteht aus einer Folge von *Worten* (4 oder 8 Bytes lang), die aufsteigend adressiert sind:



Der Programmcode kann aber nicht einfach an beliebige Positionen im Arbeitsspeicher geladen werden. Der Grund liegt darin, dass die Maschinenbefehle *Adressen* enthalten. Insbesondere Sprungbefehle müssen die Zieladresse für den Sprung enthalten. Diese Adressen müssen vom Compiler aus dem Quellcode des Programms erzeugt werden. Am einfachsten wäre es natürlich, wenn der Compiler davon ausgehen könnte, dass das Programm, wenn es in den Arbeitsspeicher geladen wird, immer ab Adresse 0 geladen wird. Dann ergeben sich alle Adressen aus der Logik des Programms, und nicht aus der Organisation der Prozesse im Betriebssystem.

Leider kann das nur funktionieren, wenn immer nur ein einziges Programm ausgeführt wird, welches komplett am Stück in den Arbeitsspeicher *ab Adresse 0* geladen wird, und das auch noch ganz ohne Betriebssystem. Da das völlig unpraktikabel ist, hat man bessere Organisationsprinzipien für die Prozessausführung entwickelt. Bis auf die modernste Version, *Virtuelle Speicherverwaltung mit Paging* (Kap. 5.2), gehen alle davon aus, dass der Quellcode so compiliert wird, dass keine *absoluten* Adressen mehr vorkommen, sondern die Adressierung so gemacht wird, dass die Anfangsadresse des Prozesses an beliebiger Stelle im Arbeitsspeicher sein kann. Der gesamte Prozess liegt *am Stück* oberhalb dieser Adresse.

5.1 Speicherverwaltung mit Partitionierung

Hierbei geht man davon aus, dass ein Prozess „am Stück“ in den Arbeitsspeicher geladen wird. D.h. wenn das Programm incl. Daten z.B. 100 KByte groß ist, dann muss ein Bereich der Größe 100 KByte im Arbeitsspeicher gefunden werden, wohin der Prozess geladen wird. Dafür gibt es verschiedene Möglichkeiten:

Feste Partitionierung:
Der Arbeitsspeicher wird in *Partitionen* fester Größe aufgeteilt. Z.B. könnte 1 GByte in Partitionen der Größe 100 MByte aufgeteilt werden. Dann passen maximal 10 Prozesse dieser Maximalgröße gleichzeitig in den Speicher.

Falls ein Prozess aber weniger Platz braucht, wird der Rest in der Partition verschwendet (*interne Fragmentierung*). Braucht der Prozess tatsächlich nur 1 MByte, dann werden im obigen Beispiel 99 BMyte verschwendet. Macht man die Partitionen aber zu klein, dann gibt es zwar mehr davon,

aber Prozesse mit großem Speicherbedarf passen dann überhaupt nicht mehr hinein. In modernen Betriebssystemen gibt es oft hunderte kleiner Prozesse gleichzeitig, aber auch einige große Prozesse. Feste Partitionierung ist daher keine Option.

Dynamische Partitionierung:
Soll ein Prozess gestartet werden, dann wird für ihn gerade soviel Platz reserviert, wie er braucht. Wenn er beendet ist, wird der Platz wieder frei gegeben, und kann dann wiederverwendet werden. Zunächst einmal braucht man dazu eine Verwaltung, mit der man immer schnell herausfinden kann, an welcher Stelle wieviel Platz frei ist.

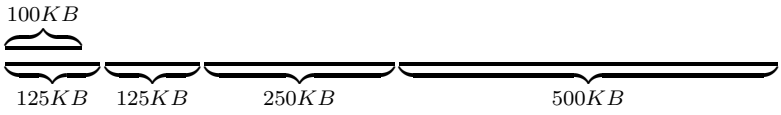
Sobald ein Prozess beendet ist, und sein Platz frei gegeben wurde, entsteht an dieser Stelle eine Lücke bestimmter Größe. Passt ein neu zu startender Prozess genau in diese Lücke, perfekt. Falls er weniger Platz braucht, kann man ihn auch in diese Lücke positionieren. Es bleibt aber dann wiederum eine Lücke, kleiner als die bisherige.

Im Laufe der Zeit beobachtet man dann:

- es gibt viele verteilte Lücken im Speicher, die meisten davon relativ klein (*externe Fragmentierung*),
- es gibt Prozesse, die größer sind als die größte Lücke. Obwohl die Summe der Lücken ausreichend wäre, passt der Prozess trotzdem nicht in den Speicher. Er muss dann warten, bis eine genügend große Lücke frei wird.

Auch diese Vorgehensweise hat zuviele Nachteile, um in modernen Computern anwendbar zu sein.

Buddy Systeme:
Hierbei reserviert man wiederum feste Partitionen. Diese werden aber erst bei Bedarf berechnet und zugewiesen. Als Beispiel, angenommen, der Arbeitsspeicher hat 1 MByte. Wenn der erste Prozess mehr als die Hälfte davon benötigt, wird das ganze MByte als eine Partition zugewiesen. Braucht er weniger als die Hälfe, dann wird solange halbiert, bis der Prozess gerade noch hinein passt. Angenommen, der braucht 100 KByte. Die Halbierung ergibt 1 Mbyte $\mapsto$ 500 KByte $\mapsto$ 250 KByte $\mapsto$ 125 KByte. In 62,5 KByte würde der Prozess nicht mehr hinein passen. Daher wird die 125 KByte große Partition reserviert. Die Aufteilung würde dann so aussehen:



Kommt jetzt wieder ein 100 KByte großer Prozess, kann er in die zweite 125 KByte große Partition geladen werden. Kommt dann noch ein dritter 100 KByte großer Prozess, dann wird die 250 KByte Partition wiederum aufgeteilt in zwei 125 KByte große Partitionen, und in die erste davon kommt der Prozess.

Das allgemeine Verfahren ist so, dass die kleinste freie Partition, in die der Prozess hineinpasst, durch Halbierung soweit aufgeteilt wird, dass der Prozess gerade noch hinein passt. Der Vorteil dieses Verfahrens ist, dass die Partitionsgröße dynamisch angepasst wird, und auch die Verwaltung der freien

Partitionen recht einfach ist. Darüber hinaus können benachbarte frei gewordene Partitionen gleicher Größe (die „Buddies“) wieder zu einer größeren zusammengefügt werden.

Man erhält aber trotzdem wieder interne Fragmentierung (Lücken innerhalb einer Partition) und externe Fragmentierung (Lücken zwischen den Partitionen). Auch dieses System hat zu viele Nachteile für moderne Betriebssysteme.

Alle diese Partitionierungsverfahren gehen von der impliziten Annahme aus, dass der Speicherbedarf der Prozesse zu Beginn bekannt ist, und sich im Lauf der Rechnung nicht ändert, insbesondere nicht größer wird. Das ist aber für viele Anwendungen nicht der Fall. Ein Editor, z.B., lädt Texte in den Speicher, deren Größe beim Start der des Editors unbekannt ist. Wird der Editor in eine Partition fester Größe geladen, kann diese sich erst während der Aktivitäten als zu klein herausstellen.

Bei jedem der vorgestellten Ansätze war der Schluss, dass sie ungeeignet sind für moderne Betriebssysteme. Das stimmt für die üblichen Computer. Für Prozessoren, die in technischen Systemen eingebaut sind, z.B. in einer Waschmaschine, muss das aber nicht gelten. Solche Systeme sind einfach genug, dass eines der Verfahren passen könnte.

5.2 Virtuelle Speicherverwaltung mit Paging

Das jetzt vorgestellte Verfahren löst eine ganze Reihe von Probleme auf einmal:

- es erlaubt dem Compiler die Annahme, dass die Adressierung bei 0 anfängt,
- es minimiert die Fragmentierungen,
- es optimiert die Suspendierung inaktiver Prozesse,
- es optimiert das „forken“ von Prozessen,
- es erlaubt, Bibliotheken einzubinden.

Zunächst einmal compiliert der Compiler die Programme so, dass die Adressierung tatsächlich von 0 anfängt. Die Adressen, die in dem compilierten Programm auftauchen bilden den *virtuellen Adressraum*.

5.2.1 Virtuelle Seitenaufteilung (Paging)

Der virtuelle Adressraum wird jetzt (virtuell) in Seiten (pages) von meist 1-4 KByte Größe aufgeteilt. Die Seitengröße muss dabei so sein, dass man die binären Adressen in zwei Blöcke aufteilen kann, die *Seitennummer*, und den *Offset* innerhalb der Seite. Bei einer Adressbreite von 32 Bit könnte man z.B. die ersten 24 Bit für die Seitennummer reservieren, und die letzten 8 Bit für den Offset zur Adressierung innerhalb der Seite.

Die folgende Adresse repräsentiert dann die Seitennummer 10 und den Offset 15:

Adresse: $\underbrace{0000000000001010}_{\text{Seitennummer}}\ \underbrace{00001111}_{\text{Offset}}$

In einem konkreten compilieren Programm kommen also Adressen vor, die Seitennummern von 0 bis zu einer Maximalgröße entsprechen. Hat das Programm z.B. einen Speicherbedarf von 1 MByte, und die Seitengröße ist 1 KByte (d.h. bei wortweise Adressierung mit 32 Bit, 8 Bits für den Offset in den Adressen), dann werden die Seitennummern von 0 bis 1000 (dezimal) gebraucht.

5.2.2 Physische Seitenaufteilung

Parallel dazu wird der Arbeitsspeicher in Seitenrahmen (*Frames*) der gleichen Größe wie die Seiten aufgeteilt. Auch hier gilt dann exakt die gleiche Aufteilung der Adressen. Man spricht hier von der *Framenummer* und dem Offset innerhalb des Frames. Die Anzahl der Frames ergibt sich dann aus der Größe des Arbeitsspeichers. Ist er z.B. 1 GByte groß, und man hat wiederum 8 Bits für den Offset, dann sind 1 Million Frames verfügbar.

5.2.3 Virtuelle Adresse $\mapsto$ physische Adresse, Seitentabelle

Die Idee ist jetzt, nicht ein ganzes Programm in eine Partition zu stecken, sondern jede virtuelle Seite separat in ein freies Frame zu stecken. Dazu braucht man nur eine Tabelle, in der steht, welche Seitennummer in welchem Frame steckt. Z.B. kann Seite 0 in Frame 5 stecken, die Seite 1 in Frame 2, die Seite 3 in Frame 6 usw. Diese *Seitentabelle* muss es einmal für jeden Prozess geben.

Die Abbildung von einer virtuellen Adresse auf eine physische Adresse mittels Seitentabelle ist jetzt ganz einfach: man extrahiert die Seitennummer, schaut in der Seitentabelle die Framenummer nach, und ersetzt die Seitennummer durch die Framenummer. Der Offset bleibt gleich.

Bildet z.B. die Seitentabelle die Seitennummer 10 (binär 1010) auf die Framenummer 30 (binär 1 1110) ab, dann ergibt sich folgende Umrechnung:

Virtuelle Adresse: $\underbrace{0000000000001010}_{\text{Seitennummer}}\ \underbrace{00001111}_{\text{Offset}}$

$\downarrow$

Physische Adresse: $\underbrace{0000000000011110}_{\text{Framenummer}}\ \underbrace{00001111}_{\text{Offset}}$

Die Ersetzung der Seitennummer durch die Framenummer wird bei jedem Speicherzugriff automatisch durch die *Memory Mangement Unit* (MMU) durchgeführt.

Dieses Verfahren hat zunächst folgende Vorteile:

- man hat wieder eine feste Partitionierung des Arbeitsspeichers in Frames, so dass die Verwaltung der freien Frames ziemlich einfach ist,

- man ist für neue Prozesse nicht auf eine ausreichend große Partitionsgröße angewiesen, sondern kann einfach genügend viele freie Frames zusammensuchen,
- ein Prozess kann dynamisch mehr Speicher anfordern. Dazu müssen nur freie Frames gefunden, und die Seitentabelle entsprechend erweitert werden.

5.2.4 Suspendierung, Paging

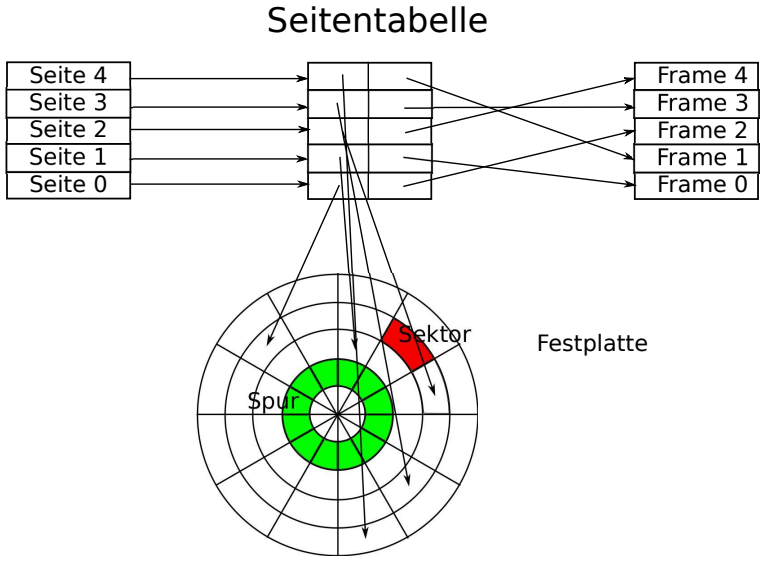
Ein weiteres Problem muss noch gelöst werden: was passiert, wenn der Arbeitsspeicher komplett voll ist, und trotzdem noch ein wichtiger Prozess gestartet werden muss?

In diesem Fall müssen entsprechend viele Frames für den neuen Prozess frei gemacht werden. Diese dürfen natürlich nicht verloren gehen, sondern sie müssen auf einem externen Speicher, meist der Festplatte, zwischengelagert werden. Hierbei hilft es, dass die Festplatten in *Sektoren* aufgeteilt sind, und die Daten sektorweise gelesen oder geschrieben werden. Die Idee ist also, jeder Seite im virtuellen Adressraum nicht nur ein Frame im Arbeitsspeicher zuzuordnen, sondern auch einen Sektor auf der Festplatte. Damit hat man folgende Möglichkeiten:

- Zunächst wird eine Seite mit der Nummer n_p in ein Frame mit der Nummer n_f geladen.
- Falls Platz gebraucht wird, und daher das Frame ausgelagert werden muss, wird es auf die Festplatte in den Sektor mit der Nummer n_s kopiert (*geswapped*). Der Eintrag $n_p \to n_f$ in der Seitentabelle wird gelöscht. Der dazugehörige Prozess muss dann natürlich suspendiert werden.
- Sobald wieder Frames frei geworden sind, kann der Sektor mit der Nummer n_s in ein jetzt freies Frame mit der Nummer n'_f kopiert werden. In der Seitentabelle wird $n_p \to n'_f$ eingetragen. Sind alle ausgelagerten Frames wieder im Arbeitsspeicher, kann der Prozess wieder aktiviert werden.

Die Aufteilung in Seiten bzw. Frames hat auch hier den großen Vorteil, dass man auf der Festplatte nicht einen zusammenhängen Bereich für einen kompletten Prozess braucht, sondern die Frames auf beliebige freie Sektoren im sog. *Swapspace* verteilen kann.

Schematisch sieht das dann so aus:



5.2.5 Seitenersetzungsstrategien

Wenn der Arbeitsspeicher voll ist, und ein wichtiger Prozess gestartet werden muss, dann muss Platz geschaffen werden, indem Frames in den Swapspace ausgelagert werden. Sobald ein Frame ausgelagert ist, kann der zugehörige Prozess nicht mehr arbeiten. Er wird daher suspendiert. Es muss also eine Entscheidung über die auszulagernden Frames, und damit über den zu suspendierenden Prozess getroffen werden. Hierzu können sowohl Informationen über den Prozess, als auch Informationen über die Frames herangezogen werden. In den meisten Betriebssystemen kann man Prozessen eine *Priorität* zuordnen. Je höher die Priorität ist, desto wichtiger ist er, und desto seltener sollte er suspendiert werden. Andere Informationen, die man für die Entscheidung verwenden könnte, wären z.B. die schon verbrauchte Rechenzeit, oder der aktuelle Speicherbedarf.

Auch für jedes einzelne Frame hat man Informationen, die für die Entscheidung nützlich sind. Am besten wäre natürlich, solche Frames frei zu machen, die am längsten nicht mehr gebraucht werden. Dazu bräuchte man aber einen Hellseherchip. Den gibt es leider noch nicht.

Es gibt aber verschiedene realistischere Möglichkeiten, mit Vor- und Nachteilen. Beispiele sind:

First In First Out (FIFO): Das Frame, welches am längsten im Speicher ist, fliegt als erstes raus. Damit fliegen aber auch häufig gebrauchte Frames raus.

Least Recently Used(LRU): Das am längsten nicht mehr benutzte Frame kommt raus

Last Frequently Used (LFU): Das bisher am wenigsten häufig benutzte Frame fliegt raus.

Climb: Hierbei steigt ein Frame bei jedem Aufruf eine Position höher, wenn sie bereits im Speicher vorhanden ist, d.h. sie tauscht ihre Position mit der vor ihr stehenden Frame. Ist ein neues Frame nicht im Speicher vorhanden, so wird das unterste Frame ausgelagert und das neue Frame dorthin geladen.

Es gibt auch noch komplexere Strategien. Wichtig beim Einsatz solcher Strategien ist auch der Verwaltungsaufwand, den man treiben muss, um die richtige Entscheidung zu treffen. Welche Strategie tatsächlich in aktuellen Betriebssystemen angewendet wird ist oft tief in deren Code verborgen.

5.2.6 Mehrstufige Seitentabellen

Die Seitengröße sollte nicht zu groß seine, sonst ist der Aufwand für das Paging, d.h. das Aus- und Einlagern der Frames groß und verlangsamt den Computer. Das wird allerdings ein Problem bei modernen Prozessoren mit 64 Bit Wortgröße. Mit 64 Bit kann man 2^{64} Wörter adressieren. Bei einer Seitengröße von 2^{10} hätte man dann $2^{64}/2^{10} = 2^{54} \sim 1.8 \cdot 10^{16}$ Seiten. Wollte man den ganzen Adressraum ausschöpfen, dann bräuchte man Seitentabellen mit so vielen Einträgen.

Das wäre viel zu groß!

Bei 64 Bit-Architektur hat man daher die Adressen nochmal weiter unterteilt, in Segmentnummer, Seitennummer und Offset. Die folgende Adresse

$$\underbrace{00000000000000000000000000001}_{Segmentnummer}\ \underbrace{0000000000000000000000010}_{Seitennummer}\ \underbrace{0000000011}_{Offset}$$

ist damit aufgeteilt in: Segment 1, Seite 2, Offset 3.

Mit der Segmentnummer adressiert man die jeweilige Seitentabelle, in der dann die konkrete Framenummer steht. Pro Segment hat man jetzt eine eigene Seitentabelle. Der Vorteil ist, dass man nicht tatsächlich für jedes Segment eine Seitentabelle braucht, sondern nur für die Segmente, die ein Programm auch wirklich braucht.

Als Beispiel: Seitengröße 1 KByte, Segmentgröße 100 Mbyte, Programmgröße 300 MByte. Dann braucht man eine Segmenttabelle mit 3 Einträgen für 3 Seitentabellen. Wenn das Programm z.B. Videos bearbeitet, und jetzt ein Video mit 4 GByte lädt, dann werden noch 40 Seitentabellen erzeugt, und die Segmenttabelle entsprechend erweitert.

Adressumrechnung: Die Adressumrechnung einer 64-Bit Adresse ist jetzt folgendermaßen:

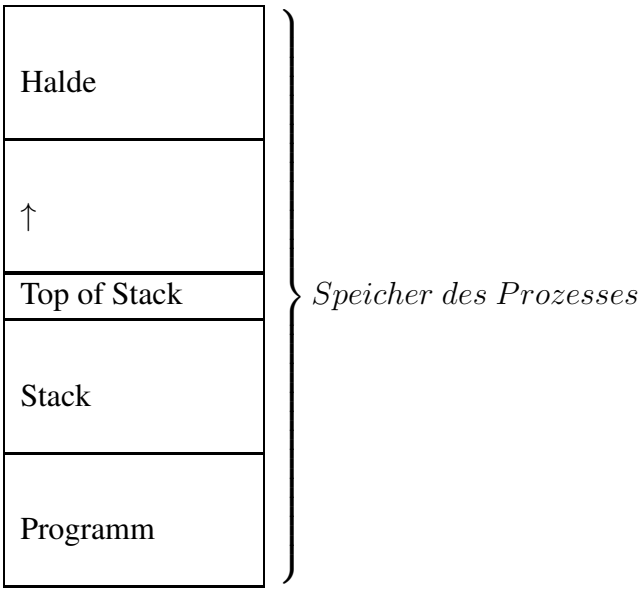
- Extrahiere die Segmentnummer, die Seitennummer und den Offset.
- Für die Segmentnummer greife auf die dazugehörige Seitentabelle zu (die könnte u.U. auch ausgelagert sein, und muss dann wieder in den Arbeitsspeicher geladen werden).
- Extrahiere aus dieser Seitentabelle die zur Seitennummer gehörige Framenummer.
- Ersetze in der Adresse Segmentnummer und Seitennummer durch die Framenummer. Der Offset bleibt wieder gleich.

Die Aufteilung in Segment-, Seitennummer und Offset kann man noch weiter verfeinern und erhält dann mehrstufige Seitentabellen. Der Mechanismus wird dann aber immer komplizierter und zeitaufwendiger.

6 Speicheraufteilung, Unterprogrammaufruf, Prozesswechsel

Der einem Prozess zugeteilte Bereich des Arbeitsspeichers wird i.A. aufgeteilt in *Programm, Stack* und *Halde*. Im Programmteil steckt der Maschinencode. Er sollte eigentlich im Verlauf der Abarbeitung nicht verändert werden, obwohl das möglich wäre, und in manchen Anwendungen vielleicht sogar gewünscht. Der Stack wird für Unterprogrammaufrufe gebraucht. Er kann wachsen und schrumpfen. Die Halde (engl. Heap) für globale Daten. In Java z.B. wird für jedes Objekt, welches mit einem Konstruktor erzeugt wird, Platz auf der Halde reserviert. In anderen Sprachen gibt es dazu spezielle Befehle, in C z.B. den Befehl `memalloc`.

Eine Aufteilung könnte z.B. so aussehen:



In diesem Bild könnte der Stack nach oben wachsen, und die Halde nach unten. Sobald sie sich treffen, ist der Speicherplatz voll. Alternativ könnte man für den Stack eine Grenze setzen, bis dahin es wachsen kann, und die Halde oberhalb dieser Grenze beginnen lassen. Die Halde könnte sich dann nach oben ausdehnen. In beiden Fällen kann es aber zum berüchtigten *Stack Overflow*-Fehler kommen. Der Fehler entsteht aber meist nicht durch wirklichen Platzmangel, sondern dadurch, dass ein fehlerhaftes rekursives Programm aus dem Ruder läuft.

Dieses Stück Speicher ist natürlich nur im virtuellen Adressraum zusammenhängend. Über die Seitentabelle kann es physisch wild verteilt sein.

6.1 Unterprogrammaufruf

Kein vernünftiges Programm besteht nur aus einer Folge von elementaren Anweisungen. Stattdessen wird ein großes Programm in kleine Teile aufgeteilt, die sich gegenseitig aufrufen können. In objektorientierten Sprachen heißen sie meist *Methoden*. In imperativen Sprachen heißen sie *Unterprogramme* oder *Subroutines*.

Sie können von beliebigen Stellen eines Programms mit Argumenten aufgerufen werden, berechnen etwas, und liefern kein, ein, oder gar mehrere Ergebnisse zurück. Unterprogramme können wiederum Unterprogramme, oder auch sich selbst aufrufen, und das kann (nahezu) beliebig tief verschachtelt werden.

Ein Beispiel ist die rekursive Version der Fakultätsfunktion $n! = 1 \cdot 2 \cdot \ldots \cdot n$:

$$fakultaet(n) = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot fakultaet(n-1) & \text{sonst} \end{cases}$$

Das Unterprogramm $fakultaet$ ruft sich selbst wieder auf.

Bei einem Unterprogrammaufruf muss:

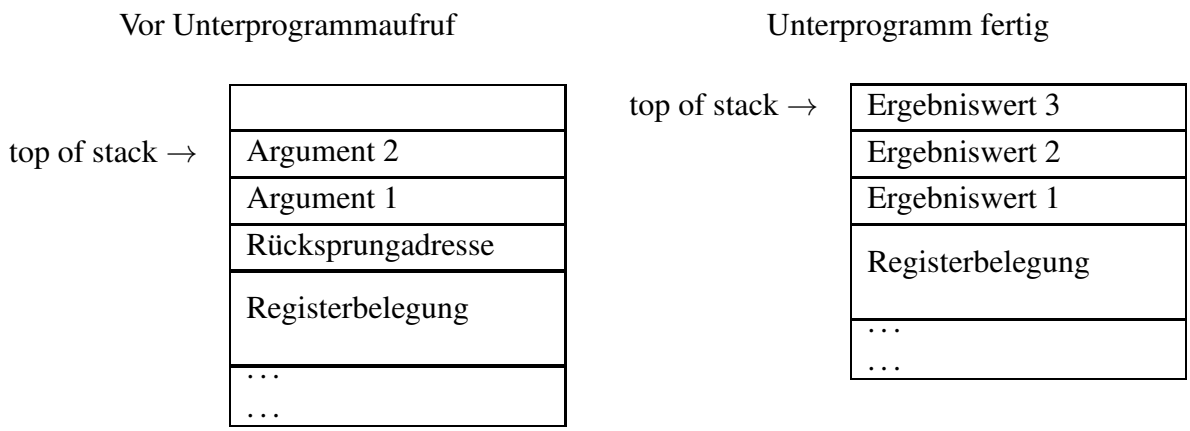
- der momentane Zustand der Berechnung im aufrufenden Programm gesichert werden,
- eine Rücksprungadresse vereinbart werden, damit das Unterprogramm, wenn es fertig ist, auch wieder an die richtige Stelle zurückkehrt,
- die Argumenten für das Unterprogramm an eine Stelle gespeichert werden, wo sie das Unterprogramm abholen kann,
- an den Anfang des Unterprogramms gesprungen werden,
- die Berechnung im Unterprogramm durchgeführt werden,
- die Ergebnisse an eine Stelle gespeichert werden, wo sie das aufrufende Programm abholen kann,
- zur Rücksprungadresse zurück gesprungen werden.
- der Ausgangszustand im aufrufenden Programm wieder hergestellt werden,
- die Ergebnisse des Unterprogrammaufrufs abgeholt und dann weitergerechnet werden.

Als Zwischenspeicher eignet sich hervorragend ein Stack. Mit seiner Hilfe kann man diese Schritte einfach durchführen:

- Speichere die Inhalte der Register auf dem Stack,
- Speichere die Rücksprungadresse auf dem Stack,
- Speichere die Argumente für das Unterprogramm auf dem Stack.

Das Unterprogramm holt sich dann die Argumenten vom Stack, rechnet, holt die Rücksprungadresse vom Stack und speichert die Ergebnisse auf den Stack. Nach dem Rücksprung kann sich das aufrufende Programm die Ergebnisse vom Stack abholen, die Originalinhalte der Register wieder herstellen, und normal weiter rechnen.

Das folgende Bild zeigt eine möglich Stackbelegung für einen Unterprogrammaufruf, welcher zwei Argumente braucht, und drei Ergebniswerte zurückliefert.



Die meisten höheren Programmiersprachen erlauben allerdings nur die Rückgabe von *einem* Ergebniswert. Das ist aber keine prinzipielle Grenze.

Konkrete Prozessoren können zusätzliche Register für die Rücksprungadresse und die Argumente für das Unterprogramm haben, so dass weniger Daten im Stack gespeichert werden müssen.

Der Vorteil des Stacks ist, dass das Unterprogramm wiederum Unterprogramme aufrufen kann, indem die Daten dem Stack einfach hinzugefügt werden. Das kann so oft geschehen, bis der Speicher für den Stack voll ist (Stack Overflow Fehler).

6.2 Prozesswechsel

Ein Prozess, der nicht mehr weiter rechnen kann oder nicht mehr weiter rechnen darf, wird abgebrochen und suspendiert. Damit er zu einem späteren Zeitpunkt wieder weiter machen kann, müssen die relevanten Daten ebenfalls zwischengespeichert werden. Dazu gehören insbesondere die Registerinhalte. Diese können jedoch nicht auf dem Stack gespeichert werden, sondern werden in den Prozesskontrollblock übertragen, von wo sie der *Dispatcher* wieder holen kann, wenn der Prozess wieder aktiviert wird.

Das Sichern und Wiederherstellen der Registerinhalte ist jedoch der leichtere Teil des Prozesswechsels. Zusätzlich muss noch

- Frames in den Swapspace ausgelagert werden, wenn Platz gebraucht wird,
- der neu zu aktivierende Prozess vom Scheduler bestimmt werden,
- für den neu zu aktivierenden Prozess Platz für seine Frames gefunden werden,
- diese Frames wieder in den Speicher geladen werden,
- die Segmenttabelle angepasst werden.

Ist der Arbeitsspeicher genügend groß (was in modernen Computer oft der Fall ist), dann erübrigt sich meist die Auslagerung in den Swapspace. Trotzdem ist ein Prozesswechsel noch eine aufwendige Operation. Sie ist aber unvermeidbar, wenn man dem Benutzer das Gefühl geben will, dass alle Prozesse quasi parallel laufen, und wenn auch die Hardware des Rechners ausgelastet werden soll.

7 Prozesskommunikation

Einfache Programme werden gestartet, rechnen etwas, und werden dann wieder beendet. Die nächste Stufe ist, dass die Programme mit dem Benutzer kommunizieren. Dazu gibt es in allen Programmiersprachen Befehle zum lesen und schreiben von Daten. Es gibt aber auch Situationen, wo Programme miteinander kommunizieren müssen. Dafür wurden verschiedene Mechanismen entwickelt.

7.1 Interrupts

Die primitivste Art der Kommunikation besteht darin, einem Prozess einen *Interrupt* zu schicken. Ein Interrupt bewirkt, dass der Prozess sofort in seiner normalen Befehlsfolge unterbrochen wird, und einen sog. *Interrupthandler* ausführt. Der Interrupthandler ist ein frei programmierbares Stück Code, welches auf den Interrupt in geeigneter Weise reagieren soll. Nachdem der Interrupthandler fertig ist, kehrt das Programm wieder an die Stelle zurück, wo es unterbrochen wurde, und rechnet dort weiter.

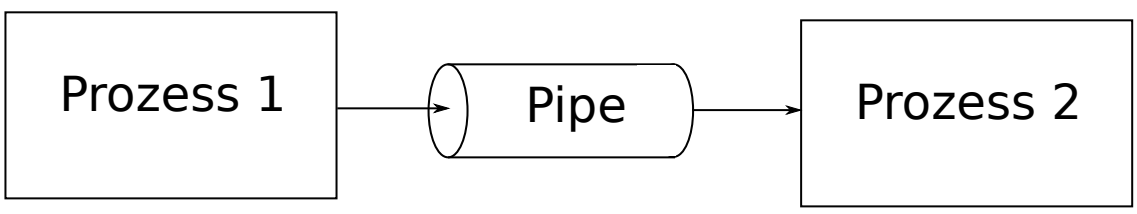
Ein Anwendungsbeispiel wäre: das Programm führt eine stundenlange Rechnung durch, und man möchte ab und zu wissen, wo es gerade steht. Dann könnte man einen Interrupthandler programmieren, der den momentanen Stand der Rechnung ausdruckt. Jetzt kann man zu beliebiger Zeit den Interrupt schicken, und bekommt dann den Stand der Rechnung ausgedruckt.

In den meisten Betriebssystemen wird neben dem Interruptsignal eine Interruptnummer mitgeschickt, meist zwischen 0 und 31. Für jede Interruptnummer kann man einen separaten Interrupthandler programmieren, so dass der Prozess flexibler reagieren kann. Einige Nummern werden allerdings schon vom Betriebssystem selbst abgefangen, um z.B. den Prozess zu killen.

Die Programmierung von Interrupts ist sehr betriebssystemabhängig. Sie wird meist durch C- oder C++-Bibliotheken verfügbar gemacht. In Java kann man bisher allerdings keine Interrupthandler programmieren. Das Problem in Java ist, dass ein Java Programm mehrere parallele Threads ausführen kann. Bisher ist es nicht möglich, von außerhalb eines Java Prozesses einen bestimmten Thread anzusprechen, und ihm einen Interrupt zu schicken.

7.2 Pipes

Pipes sind Kommunikationswege zwischen Prozessen *innerhalb eines Rechners*. Zwei Prozesse können eine gemeinsame Pipe einrichten, über die man Daten *in eine Richtung* schicken kann.



Für bidirektionale Kommunikation muss man zwei verschiedene Pipes einrichten.

Unnamed Pipes: Ein Prozess kann eine Pipe erzeugen, dann einen Kindprozess „forken“, und über die Pipe dem Kindprozess Daten schicken.

Named Pipes: Mit dieser Version können unabhängige Prozesse innerhalb des Computers miteinander kommunizieren. Man kann sich eine Named Pipe als *File* vorstellen, in den ein Prozess schreibt, und von dem der andere liest.

Pipes über die Kommandozeile in Unix und Linux: Hierfür gibt es das Symbol `|`. Damit kann man die Ausgabe eines Prozesses in die Eingabe eines anderen Prozesses schicken.

Beispiel: das Kommando

```
ls -l | wc
```

leitet die Ausgabe des `ls` Kommandos (welches die Files auflistet) in die Eingabe des Programms `wc` (welches die Buchstaben, Wörter und Zeilen zählt) um.

7.3 Sockets

Zur Kommunikation zwischen Prozessen, die auf verschiedenen Computern laufen, benutzt man *Sockets*. Dazu benötigt man

- die Internet Adresse des jeweils anderen Computers,
- eine sog. *Portnummer*.

Internet Adresse und Portnummer kann man vergleichen mit der Adresse eines Hochhauses, und der Stockwerksnummer. Die Portnummern sind Zahlen zwischen 0 und 65535.

Mit Internet Adresse und Portnummer kann man dann einen bidirektionalen Kommunikationskanal zwischen zwei Prozessen aufbauen. Jeder Prozess, der einen Socket einrichten will, muss eine *freie* Portnummer wählen. Diese muss dem Partner bekannt sein.

Ist in beiden Prozessen der Socket eingerichtet und die Kommunikation eröffnet, dann können sie lesen und schreiben wie auf Files. Jeder Prozess kann soviele Sockets einrichten, wie freie Portnummern zur Verfügung stehen. Eine ganze Reihe von Portnummern sind allerdings fest vergeben, und können nicht benutzt werden.

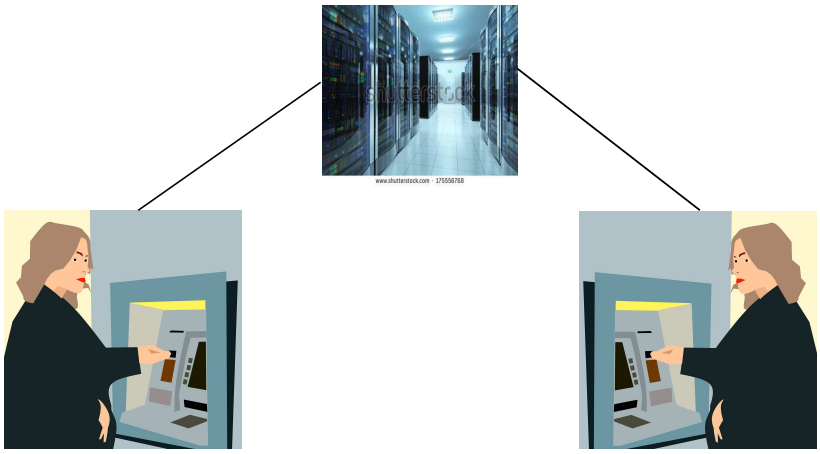
8 Probleme paralleler Prozesse

Die Möglichkeit, Prozesse parallel auszuführen, macht unsere Computer überhaupt praktisch benutzbar. Leider führt die Parallelisierung zu einer Reihe von Problemen, die die praktische Programmierung sehr tückisch machen.

8.1 Race Conditions

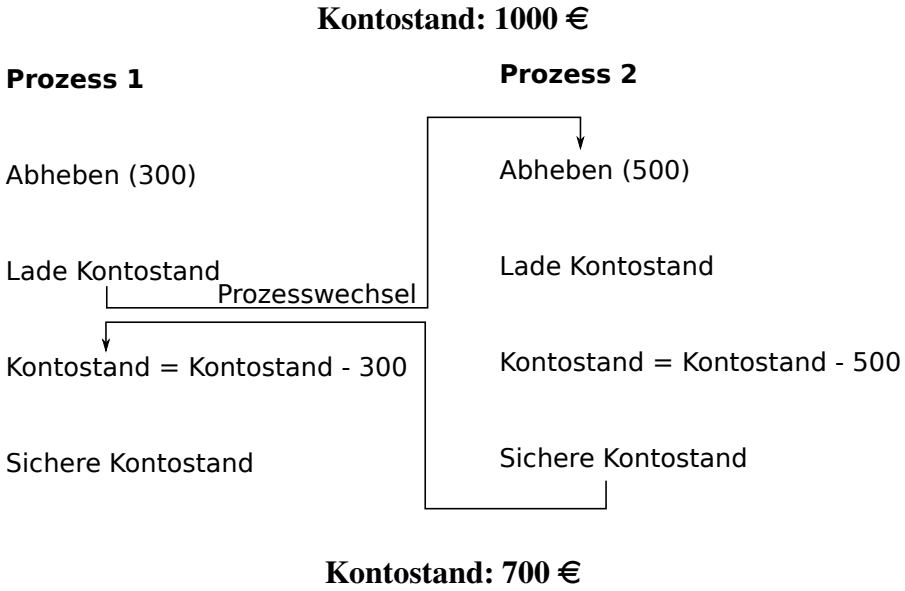
Wenn verschiedene Prozesse *gleichzeitig* auf *die gleichen* Ressourcen, z.B. Daten, zugreifen, kann es zu Problemen kommen.

Man betrachte folgendes Szenario: Zwei Personen greifen *gleichzeitig* auf dasselbe Konto zu. Eine will 300€ abheben, und eine zweite will 500€ abheben. Auf dem Konto sind 1000€. Am Ende sollten also noch 200€ übrig sein.



Die Geldautomaten kontaktieren das Rechenzentrum der Bank. Dort werden zwei Prozesse, Prozess 1 und Prozess 2 aktiviert, die die Anforderungen der Geldautomaten bedienen. Sie laden jeweils den aktuellen Kontostand von der Datenbank, subtrahieren den abgehobenen Betrag, und sichern den Kontostand wieder in der Datenbank. Die Prozesse arbeiten allerdings nicht echt parallel, sondern können mitten in der Rechnung suspendiert werden, damit andere Prozesse ebenfalls eine Chance haben.

Es könnte dabei zu folgender Abfolge kommen:



Nachdem Prozess 1 den Kontostand von 1000€ von der Datenbank geladen hat, kommt unglücklicherweise ein Prozesswechsel zu Prozess 2. Dieser lädt jetzt wiederum den Kontostand von 1000€, subtrahiert die 500€ und sichert den neuen Kontostand von 500€ in die Datenbank. Danach wird Prozess 1 wieder aktiviert. Er hat aber noch den alten Kontostand von 1000€ in seinem internen Speicher. Davon subtrahiert er die 300€. Das Ergebnis, 700€ wird in die Datenbank gespeichert, und überschreibt dabei die 500€ von Prozess 2. (Die Bank, und dann der Programmierer eines solchen Programms würden jetzt ernsthafte Probleme bekommen.)

Das Problem ist dabei, dass beide Prozesse *auf lokalen Kopien* der Daten in der Datenbank arbeiten. Sobald sie eine Kopie gezogen haben, werden Änderungen in der Datenbank nicht mehr war genommen.

Das Phänomen nennt man *Race Condition*. (Mehrere Prozesse wetteifern um gemeinsame Daten). Es kann immer dann auftreten wenn mehrere parallel laufende Prozesse (oder Threads) lokale Kopien von gemeinsamen Daten ziehen, diese ändern, und dann die geänderte Kopie wieder zurückschreiben.

Typische Situationen sind:

- Daten aus Datenbanken oder Files in den Arbeitsspeicher kopieren,
- Daten aus dem Arbeitsspeicher in Register des Prozessors kopieren,
- Daten aus dem Arbeitsspeicher in verschiedene Bereiche des Caches kopieren.

Das Heimtückische an Race Conditions ist, dass sie nur sehr selten, und in sehr großen Zufällen, zu Fehlern führen. Ein Programm kann 1000 mal fehlerfrei arbeiten, und beim nächsten mal tritt der Fehler auf.

Alle Programmiersprachen, die Parallelisierung unterstützen, haben daher Mechanismen gegen Race Conditions. In Java z.B. ist es das Schlüsselwort *synchronized*. Damit kann man einen kritischen Bereich des Programms gegen Unterbrechungen schützen. Im obigen Kontoführungsbeispiel würde man das ganze Programm gegen Unterbrechungen schützen, so dass kein Prozesswechsel an der kritischen Stelle möglich wäre.

8.2 Deadlocks

Ein weitere Fehlerquelle bei parallel laufenden Prozessen sind *Deadlocks*. Man kennt das Phänomen auch aus dem Alltag, wenn an einer Kreuzung ohne Verkehrszeichen gleichzeitig an allen Einmündungen Autos stehen. Wegen der rechts-vor-links Regel darf dann eigentlich niemand fahren.

Das folgende Beispiel illustriert, dass das Phänomen auch im Computer-Kontext auftreten kann.

Diesmal betrachten wir zwei Konten einer Bank, Konto 1 und Konto 2. Jetzt möchte Karl einen Betrag von Konto 1 nach Konto 2 überweisen. Gleichzeitig möchte Kurt einen Betrag von Konto 2 nach Konto 1 überweisen.

Im Bankcomputer werden zwei Prozesse aktiv. Der erste öffnet Konto 1 und liest den Kontostand. Dummerweise genau gleichzeitig öffnet Prozess 2 das Konto 2 und liest dessen Kontostand. Damit Prozess 1 die Überweisung verbuchen kann, muss er jetzt Konto 2 öffnen. Dieses ist aber von Prozess 2 gesperrt. Prozess 2 muss für die Überweisung Konto 1 öffnen, welches von Prozess 1 gesperrt ist. Beide müssen warten, bis die Zielkonten entsperrt sind, was aber nicht passiert, da sich beide gegenseitig blockieren.

Ein Deadlock kann immer passieren, wenn parallele Prozesse auf zwei oder mehr Ressourcen zugreifen müssen, und dabei eine ringförmige Abhängigkeit entsteht:

Prozess 1 blockiert Ressource 1 und braucht Ressource 2.
Prozess 2 blockiert Ressource 2 und braucht Ressource 3.
. . .
Prozess n blockiert Ressource n und braucht Ressource 1.

Abhilfe: Priorisierung der Ressourcen
Eine Abhilfe, die in manchen Situationen helfen kann, ist, die Ressourcen zu priorisieren, und nach den Prioritäten zu blockieren.

Im obigen Kontobeispiel könnte die Kontonummer die Priorität bestimmen.

Prozess 1 könnte daher Konto 1 öffnen. Prozess 2 braucht Konto 1 und Konto 2. Nach der Kontopriorität muss er aber erst auf Konto 1 zugreifen, bevor er Konto 2 eröffnet. Da Konto 1 blockiert ist, muss er warten. Prozess 1 kann daher Konto 2 öffnen, die Überweisung durchführen und dann beide Konten schließen. Dann hat Prozess 2 Zugriff auf beide Konten.

Hier funktioniert es. Generell gibt es gegen Deadlocks aber leider kein Allheilmittel.

9 Sicherheitsmaßnahmen

Prozesse dürfen auf keinen Fall völlig unkontrolliert im Computer agieren. Damit sie das nicht tun können, gibt es eine Reihe von Sicherheitsmaßnahmen.

9.1 Erlaubter Speicherbereich

Jeder Prozess agiert in seinem ihm zugewiesenen Bereich des Arbeitsspeichers. Damit er nicht in den Bereichen anderer Prozesse lesen oder schreiben kann, gibt es das *Grenzregister*. Bei virtueller Speicherverwaltung sind die Programme ja so compiliert, dass die Adressen von 0 bis zu einer maximalen Adresse reichen.

Diese maximale virtuelle Adresse kann in einem Register, dem Grenzregister, gespeichert werden. Greift dann ein Maschinenbefehl auf eine Speicheradresse oberhalb des Grenzregisters zu, wird der Prozess abgebrochen.

Bei zusätzlichen Speicheranforderungen werden dem Prozess freie Frames zugewiesen. Über den Mechanismus der Seitentabelle kann dann die virtuelle Adressierung dieser Frames so gemacht werden, dass sie einfach an den bisherigen Adressbereich anschließen. Das Grenzregister wird dann angepasst, und der neue Speicherbereich ist dann für den Prozess verfügbar.

9.2 Modes

Einem beliebigen Prozess ist nicht jeder Maschinenbefehl erlaubt. Z.B. wäre es ein eklatantes Sicherheitsloch, wenn jeder Prozess einfach den Inhalt des Grenzregisters ändern dürfte. Um unerlaubte Aktionen unter Kontrolle zu halten, hat man das Konzept der *Modes* eingeführt. Man unterscheidet dabei *Kernel Mode* und *User Mode*. Im Kernel Mode sind alle Maschinenbefehle erlaubt. Im User Mode nur eine eingeschränkte Menge.

Wie wird nun eine komplexe Aktion, wie z.B. eine neue Speicheranforderung durchgeführt? Dafür stellt das Betriebssystem bestimmte Systemfunktionen zur Verfügung, die ein Benutzerprozess aufrufen kann. Diese Systemfunktionen können zunächst überprüfen, ob der Prozess die Operation durchführen darf. Wenn ja, dann wird in den Kernel Mode umgeschaltet, und alle Aktionen sind erlaubt. Man geht dann davon aus, dass die Implementierung der Betriebssystemfunktionen keinen Unfug macht. Nachdem die Systemfunktion fertig ist, wird wieder in den User Mode zurückgeschaltet, und der Prozess darf normal weiterrechnen.

Eine solche Systemfunktion ist der Filezugriff. Die Funktion kann dabei überprüfen, ob der Eigentümer des Prozesses überhaupt berechtigt ist, auf den angeforderten File zuzugreifen.

9.3 Benutzerrechte

Jeder Prozess gehört einem Eigentümer. Diese haben i.A. unterschiedliche Rechte. Meist gibt es einen *Administrator*, der die meisten Rechte hat, und normale Benutzer, die die wenigsten Rechte haben. Der Administrator darf z.B. neue Benutzeraccounts anlegen, die Files aller Benutzer ansehen usw. Das Betriebssystem kann also für jeden Prozess über seinen Eigentümer und dessen Rechte überprüfen, ob eine bestimmte Aktion, die im Kernel Modus läuft, erlaubt ist, oder nicht.

Leider zeigt die Praxis, dass all diese Sicherheitsmaßnahmen doch nicht ausreichen, um unerwünschte Aktionen auszuschließen. Immer wieder werden Sicherheitslücken gefunden, die dann von Hackern ausgenutzt werden.

10 Ausblick

Das bisherige Modell geht davon aus, dass ein Programm compiliert wird, der komplette Maschinencode auf der Festplatte liegt, und dann als Prozess gestartet werden kann. Jedes Programm adressiert ab Adresse 0. Durch den Mechanismus der Seitentabellen werden dann die virtuellen Adressen auf die physischen Adressen abgebildet. Das ist aber leider immer noch ein zu vereinfachtes Modell.

Was man wirklich möchte ist:

- größere Programme erstellen, indem einzelne Teile getrennt editiert und compiliert werden,
- einzelne compilierte Teile zu einem lauffähigen Programm zusammenbinden,
- lauffähige Programmfiles so klein wie möglich halten, d.h.
  - wiederverwendbare Progammteile als Bibliotheken organisieren,
  - Bibliotheken erst zur Startzeit des Programms dazu laden,
- *Plugins* noch im laufenden Programm nachladen,
- Code, welcher von mehreren Prozessen benutzt wird, möglichst nur einmal im Speicher haben,
- Programmteile aus verschiedenen Programmiersprachen zusammenbinden.

All dies wird in dem Miniskript zur Prozesserzeugung behandelt. Des weiteren gibt es ein eigenes Miniskript zum Thema Threads.

Stichwortverzeichnis