Grundlagen der Betriebssysteme

Praktische Einführung mit Virtualisierung

Stefan Bosse

Universität Koblenz - FB Informatik

Rechner- und Prozessorgrundlagen

Wie funktioniert ein Rechner?

Rechner- und Prozessorgrundlagen

Wie funktioniert ein Rechner?

Wie werden Programme von einem Rechner verarbeitet?

Grundmodell eines Rechners

Programm

Eine Sequenz von Maschinenbefehlen (= Prozessorinstruktionen) wird zusammen mit ihrer Datenhaltung als Programm bezeichnet

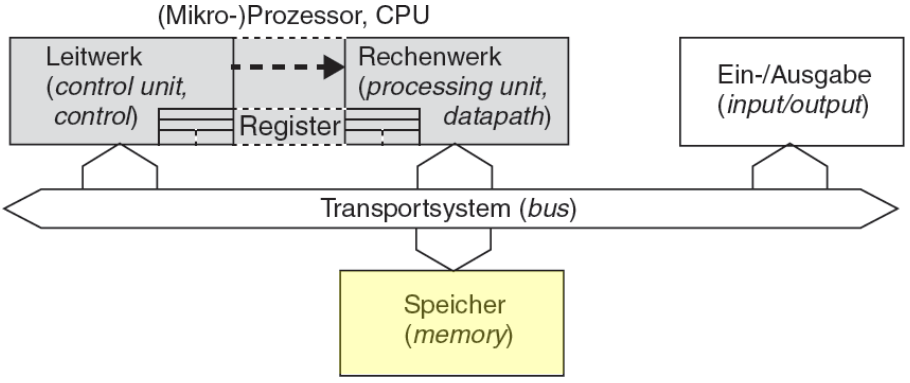
Programmausführung

Entsprechend die Ausführung durch den Prozessor als Programmausführung.

Die meisten heute gebauten Computersysteme beruhen auf der Aufbaustruktur des Von-Neumann-Rechners, die John von Neumann 1946 aufgestellt hat. Seltener kommt die alternative Struktur des Harvard-Rechners zum Zug, benannt nach der Struktur des Mark-I-Rechners an der Harvard University (1939-44).

Von-Neumann Rechner

Der Von-Neumann-Rechner besteht aus vier Funktionseinheiten:



[bsdp]

Abb. 1. Funktionsblöcke des Von-Neumann-Rechners

Leitwerk (Control Unit, CU)

Programme werden maschinenintern als Zahlen, auch Maschinenbefehle genannt, gespeichert. Die Maschinenbefehle legen die vom Prozessor auszuführenden Operationen fest. Das Leitwerk holt die Maschinenbefehle nacheinander aus dem Speicher, interpretiert sie und setzt sie in die zugehörigen Steueralgorithmen um. Das Leitwerk übernimmt damit als Befehlsprozessor die Steuerung der Instruktionsausführung. Unter Steueralgorithmen verstehen wir mögliche Prozessoroperationen, wie z.B. eine Addition, logische Oder-Verknüpfung oder Daten kopieren.

Rechenwerk (Processing Unit, PU)

Eingabedaten können neben den Maschinenbefehlen Bestandteile eines Programms sein (sie liegen dann im Speicher vor) oder werden während des Programmablaufs über die Funktionseinheit Ein-/Ausgabe von der Peripherie hereingeholt. Das Rechenwerk holt die Daten aus dem Speicher bzw. von der Eingabe, transformiert diese Daten mittels unterschiedlicher Steueralgorithmen und legt sie im Speicher ab bzw. übergibt sie der Ausgabe. Als eigentlicher Datenprozessor realisiert es die logischen und arithmetischen Operationen.

Speicher (Memory)

Er enthält die Maschinenbefehle und die zu verarbeitenden Daten. Eine Folge von logisch zusammengehörenden Maschinenbefehlen bezeichnen wir als Programm. Sowohl Befehle als auch Daten befinden sich in einem gemeinsamen Adressraum. Der Speicher dient somit der kombinierten Ablage von Programmen und Daten.

Ein-/Ausgabe (Input/Output, I/O)

Sie verbindet die Peripheriegeräte (z.B. Tastatur, Monitor, Drucker) mit dem Rechenwerk, stellt also eine oder mehrere Schnittstellen zur Umwelt dar. Mittels passender Maschinenbefehle werden über die Ein-/Ausgabe Daten von der Peripherie entgegengenommen oder dieser übergeben. Ursprünglich wurden das Eingabewerk und das Ausgabewerk als zwei getrennte Funktionsblöcke betrachtet. Heute werden sie zur Ein-/Ausgabe zusammengefasst.

Bussystem

Die Schaltzentrale um alle vier Werke miteinander zu verbinden. Gleichzeitig der Flaschenhals im gesamten Rechner (neben dem Speicher als zentrale Ressource für Programm und Daten)

[bsdp]

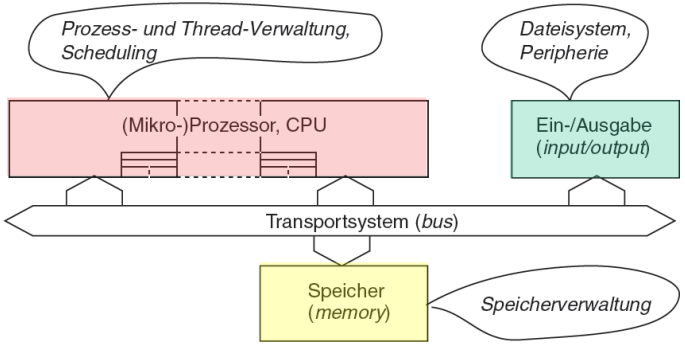
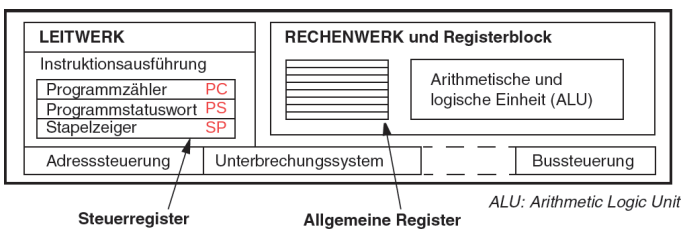


Abb. 2. Betriebssystem und Von-Neumann-Rechner

Prozessoraufbau

- Jede CPU-Familie besitzt einen spezifischen Instruktionssatz und Registeraufbau, mit dem sie sich von anderen Prozessorfamilien unterscheidet.
  - Deswegen muss ein Programm stets für die richtige CPU übersetzt sein, damit die Maschinenbefehle korrekt interpretiert werden.
  - Innerhalb einer sogenannten CPU-Familie kann jedoch der gleiche Code benutzt werden.
- Ein Prozessor besteht grob gesehen aus den Teilen Leitwerk, Rechenwerk, Register, Adress- und Bussteuerung



[bsdp]

Abb. 3. Schematischer Prozessoraufbau

Register

- Register sind Speicherzellen, aber im Mikroprozessor integriert, und teils mit spezieller Bedeutung, wie z.B.:
  - PC: Program Counter (Code Adresszeiger)
  - SP: Stack Pointer (Daten Adresszeiger)
- Die Registersätze verschiedener CPu Familien können sich erheblich unterscheiden
- Man unterscheidet grob drei Klassen CPU Architekturen, mit weitreichenden Folgen für Betriebssysteme und Programme (Erstellung):
  - **Complex Instrucion Set Computer** (CISC): Wenige Register, Operationen sind hauptspeicherzentriert, variables Instruktionsformat und Anzahl von Operanden ⇒ Intel x86 usw.
  - **Reduced Instruction Set Computer** (RISC): Viele Register, Operationen sind registerzentriert, teils fixes Instruktionsformat, wenige Operanden ⇒ ARM Cortex, Sun UltraSparc usw.
  - **Stack Processors**: Wenige Register, Operationen sind stackzentriert, 0-Operand Instruktionen ⇒ Green Array F18A (Dual stack Forth architecture), Virtual Machines!

Register

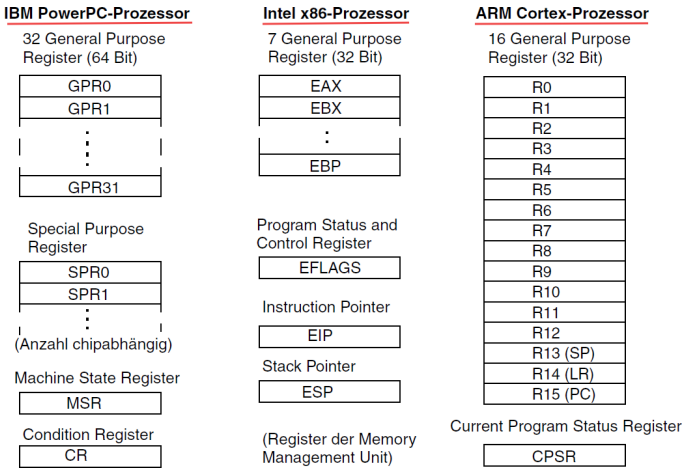


Abb. 4. Beispiele von CPU-Registern

Grundlagen der Programmausführung

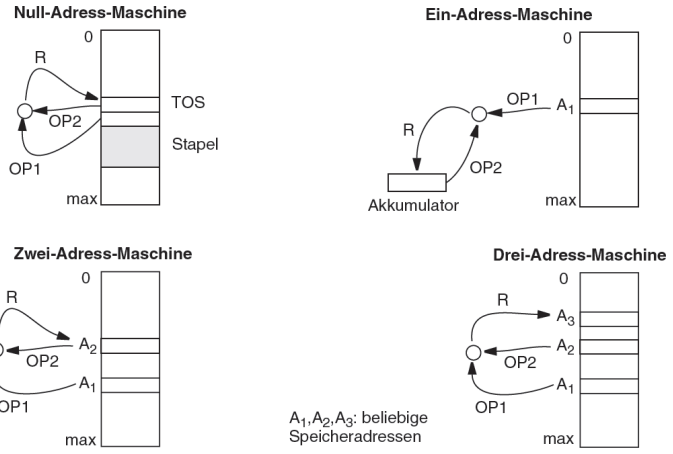


Abb. 5. Rechnereinteilung nach Adressanzahl: gezählt wird die Anzahl Operanden eines dyadischen Operators (z.B. einer Addition)

Grundlagen des Adressraums

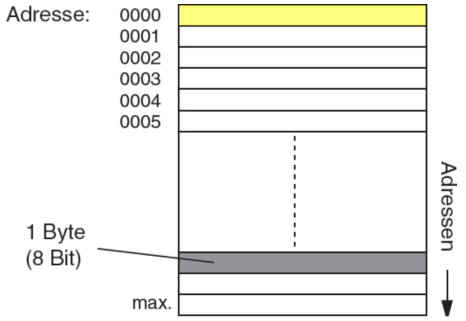
Der Zugriff auf die Befehle eines Programms erfolgt ebenso wie auf alle im Speicher abgelegten Operanden mittels Speicheradressen. Die Organisation des Adressraums stellt damit eine wichtige Eigenschaft und allenfalls auch Limitierung einer bestimmten Rechnerplattform sowie des Betriebssystems dar.

Ein Adressraum Σ ist gekennzeichnet durch:

0. Gesamtgröße in Bits (oder Bytes)
1. Anzahl $N=|\Sigma|$ von Speicherzellen
2. Anzahl der möglichen Adressen (u.U. $\neq$ Bytes!)
3. Anzahl der Bits pro Speicherzelle W, d.h. die Wortbreite

Grundlagen des Adressraums

- Adressen sind numerische (ganzzahlige) Werte und referenzieren Speicherzellen
- I.A. wird die erste Adresse mit 0, die letzte als *N*-1 bezeichnet.
- Achtung: Die Adresse kann sich entweder auf die Speicherzelle mit der Wortbreite *W* oder die Elementarzelle mit der Breite *B* Bits (i.A. 1 Byte) beziehen!



[bsdp]

Abb. 6. Grundprinzip des Adressraums, hier Byteadressierung

Adressraumtypen

- Der (Haupt-) Speicher *M* ist zunächst passiv und enthält Daten und Programmcode
- Aber neben dem Speicher müssen auch Ein- und Ausgabegeräte *P* "adressiert" (selektiert) und mit denen kommuniziert werden...
- Es gibt grob zwei Adressraumtypen bezüglich Speicher und Peripheriegeräten, die auch für das Betriebssystem sehr relevant sind:
  - Gemeinsamer Adressraum $\Sigma=\langle M,P\rangle$
  - Getrennte Adressräume $\Sigma_M=\langle M\rangle$, $\Sigma_P=\langle P\rangle$

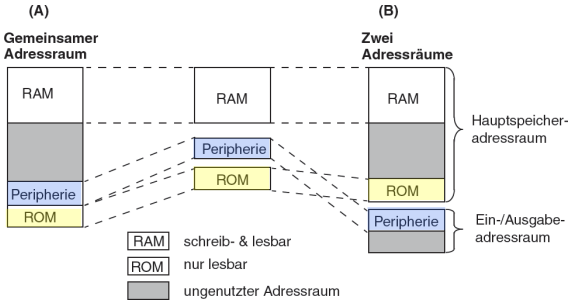


Abb. 7. Klassischer und erweiterter Von-Neumann-Rechner

Speicher und Speicherverwaltung

Wie ist das Speichermodell eines Programs und Prozesses organisiert?

Speicher und Speicherverwaltung

Wie ist das Speichermodell eines Programs und Prozesses organisiert?

Wie ist das Speichermodell eines Betriebssystems organisiert?

Speicher und Speicherverwaltung

Wie ist das Speichermodell eines Programs und Prozesses organisiert?

Wie ist das Speichermodell eines Betriebssystems organisiert?

Wie findet Speicherverwaltung statt?

Bytereihenfolge (byte ordering)

Ein ganz einfacher Prozessor könnte nur mit Datenwerten von 8 Bit Größe arbeiten, d.h., er würde über eine Speicheradresse stets nur ein einzelnes Byte lesen oder schreiben. Heutige Universalprozessoren unterstützen jedoch verschiedene Operandengrößen, die auch mehrere Byte umfassen können.

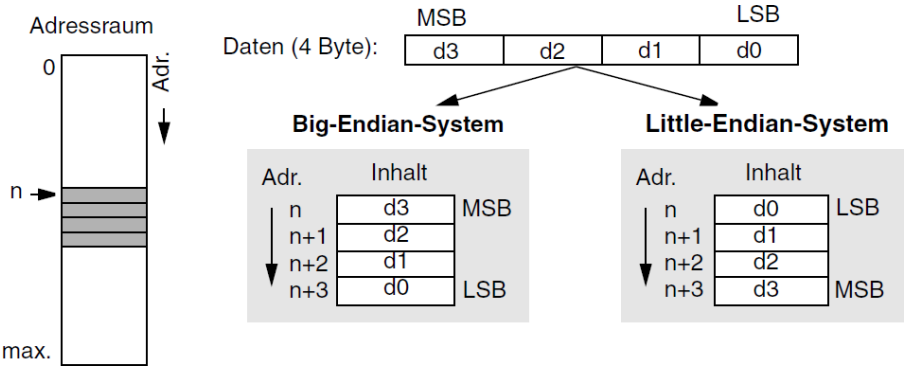


Abb. 8. Varianten der Bytereihenfolge bei Mehrbyte-Datenwerten (Beispiel für 4-Byte-Wert)

Ausrichtungsregeln im Adressraum

Die Ausrichtungsregeln (alignment rules) legen fest, auf welchen Adressen Variablen und Instruktionen liegen müssen. Sie sind maßgebend für die Programmübersetzung (Compiler, Assembler, Binder). Ihr Zweck liegt in der Erreichung optimaler Ausführungsgeschwindigkeiten auf dem benutzten Rechnersystem. Da sie von der Hardware abhängen, können sie entsprechend der benutzten Rechnerplattform variieren. Ausrichtungsregeln können sowohl für Code als auch Daten existieren.

[bsdp]

Abb. 9. Fehlausrichtung im Adressraum und ihre Folgen (Beispiel für einen 32-Bit-Datenbus und einen Zugriff auf einen Datenwert von 32 Bit Größe)

Ausrichtungsregeln im Adressraum

- Bedeutung für Instruktionscode: Abhängig von der Prozessorhardware gelten andere Ausrichtungsregeln, z.B.:
  - Instruktionen müssen immer auf geradzahligen Adressen liegen.
- Bedeutung für Daten: Die Datenbusbreite bestimmt die Ausrichtungsregeln für eine optimale Zugriffsgeschwindigkeit.
  - Z.B. ist der Datenbus 32 Bit breit. Damit ist eine optimale Ausrichtung für 4-Byte-Werte eine sogenannte Langwortgrenze, d.h. eine ohne Rest durch vier teilbare Adresse.
  - Im vorherigen Beispiel ist die Notwendigkeit von zwei Zugriffszyklen auf dem Datenbus gezeigt, wenn die Ausrichtungsregel nicht eingehalten wird (misalignment).
- Beim C-Strukturdatentyp (struct) bzw. dem C++-Klassendatentyp (class) gelten die Ausrichtungsregeln jeweils separat für die einzelnen Teilkomponenten. Neben dem Einfluss auf die Zugriffsgeschwindigkeit spielt dies wiederum sowohl für die Kompatibilität beim Datenaustausch wie auch für den effektiv belegten Platz im Adressraum eine Rolle.

Ausrichtungsregeln im Adressraum

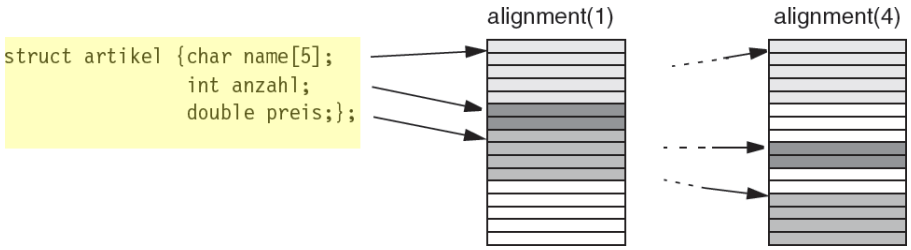


Abb. 10. Einfluss der Ausrichtung auf die resultierende Adressraumplatzierung (Beispiel)

Adressraumbelegung durch Programme

- Die Belegung des Adressraums durch ein Programm wird einerseits durch die Übersetzungswerkzeuge, andererseits durch das Betriebssystem festgelegt.
- Im Grundsatz können etwa fünf verschiedene Bereiche unterschieden werden:

1. **Code und Konstanten**: Die zugehörigen Speicherinhalte werden aus der ausführbaren Datei in den Hauptspeicher geladen. Dieser Adressbereich ändert seine Größe während der Programmausführung nicht.
2. **Initialisierte Daten**: Ein passender Bereich des Adressraums wird reserviert und anschließend werden die Initialwerte der Variablen aus der ausführbaren Datei dorthin geladen. Dieser Adressbereich ändert seine Größe während der Programmausführung nicht.
3. **Nicht initialisierte Daten**: Ein passender Speicherbereich wird reserviert und eventuell gelöscht (d.h. mit 0 geladen). Dieser Adressbereich ändert seine Größe während der Programmausführung nicht.

Adressraumbelegung durch Programme

4. Heap und Stack: Es wird ein gemeinsamer Bereich im Adressraum derart reserviert, dass die zwei Bereiche möglichst weit voneinander entfernt sind. Im Betrieb können sowohl Heap als auch Stack wachsen und schrumpfen.

Das Wachstum ist aber nur ungefährlich, solange sich die Heap- und Stack-Inhalte nicht überschneiden. Eine derartige Überschneidungssituation würde zu einem Fehler führen. Die Kunst liegt darin, diese Bereiche ausreichend groß zu wählen und eine mögliche Fehlsituation erkennen zu können. Ein Betriebssystem und/oder ein Laufzeitsystem einer Programmiersprache können hier allenfalls Hilfe anbieten, um Fehlsituationen zu erkennen.

- Weitere Lösung des Kollisonsproblem: Getrennte Segmente für Heap und Stackspeicher eines Programms.

Adressraumbelegung durch Programme

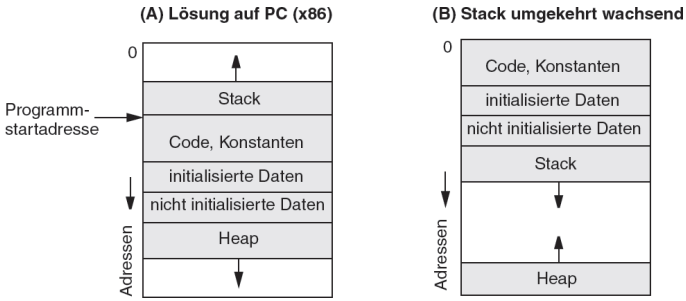


Abb. 11. Alternative Layouts des Adressraums eines Programms (oder Prozesses als Programm in Ausführung)

Adressraumnutzung durch C-Programme

Für die Programmierung ist es hilfreich, die Platzierungsregeln für Variablen in der Programmiersprache C zu kennen. Nachfolgend sind sie kurz aufgeführt:

Global (d.h. außerhalb einer Funktion) deklarierte Variablen

Ohne Initialisierung werden sie im Bereich nicht initialisierte Daten platziert. Mit einer Initialisierung jedoch im Bereich initialisierte Daten. Diese unterschiedliche Behandlung wird deshalb gemacht, weil die Anfangswerte für initialisierte Daten sich aus der ausführbaren Datei laden lassen. Das Betriebssystem kann so diese zwei Fälle klar voneinander unterscheiden.

Innerhalb einer Funktion deklarierte Variablen

Ohne Angabe der Speicherklasse (implizit gilt auto) oder mit Angabe von auto: Es erfolgt eine Platzierung auf dem Stapel. Mit Angabe der Speicherklasse static: Platzierung wie global deklarierte Variablen.

Adressraumnutzung durch C-Programme

Konstanten (const)

Platzierung im Bereich Code, Konstanten. Sowohl Code als auch Konstanten sind funktional gesehen nur lesbare Speicherinhalte. Da sie separat ausgezeichnet werden, kann das Betriebssystem die so belegten Speicherbereiche mit einem Schreibschutz belegen.

Dynamisch allozierte Daten

Platzierung auf der Halde (Heap). Reservation mittels der Bibliotheksfunktionen `malloc()` oder `calloc()`, Freigabe mittels der Bibliotheksfunktion `free()`. In C++ leisten die Operatoren `new` und `delete` Vergleichbares.

Aufrufparameter von Funktionen

Die Parameter stehen auf dem Stapel (Stack) bereit oder werden über allgemeine Prozessorregister übergeben (compilerabhängig konfigurierbar). Zu beachten ist, dass bei der Übergabe von Zeigern die Daten selbst anderswo liegen, d.h. nicht auf dem Stapel!

Adressraumnutzung durch C-Programme

C2JS

Adressraumnutzung durch C-Programme

```
int x=0;
int foo(int x) {
  int t=x;
  return t+1;
}
int main() {
  int y;
  int *yp;
  y=foo(x);
  yp=(int*)malloc(sizeof(int));
  *yp=y;
}
```

Bsp. 1. Beispiel eines C Programms

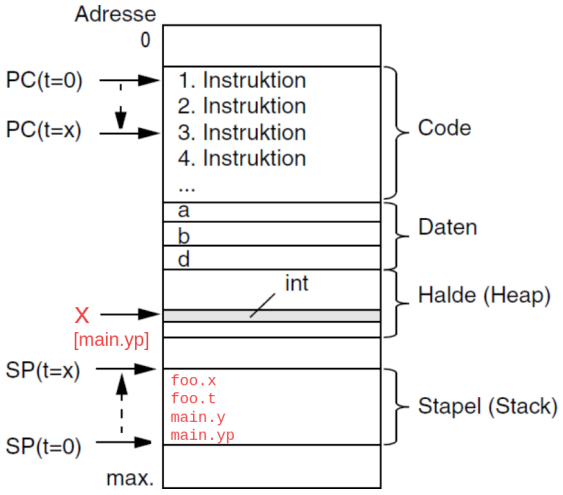


Abb. 12. Variablenplatzierung im Adressraum (entsprechend Programmbeispiel)

Heap versa Stack Speicher

1. Beide nehmen Datenvariablen auf (nur in speziellen Fällen Programmcode, z.B. bei Virtuellen Maschinen);
2. Der Heap Speicher braucht eine explizite Speicherverwaltung
   - Verwaltung von belegten und freien Bereich im Heap
   - Verwendung von Tabellen- oder Listenstrukturen
   - Speicherallokation und Speicherrückgabe erfolgt durch das Programm (oder Betriebssystem) explizit;
3. Der Stack Speicher braucht keine explizite Speicherverwaltung, er verwaltet sich automatisch über einen einzigen Zeiger (Stack Pointer SP)
   - Die Allokation ist lediglich eine Verschiebung des SP
4. Der Heap kann fragmentieren (Verlöcherung), der Stack nicht.

Funktionsweise des Stapels und Stapelzeigers (SP)

Wenn Funktionen (Unterprogramme) aufgerufen werden wird:

1. Ein Funktionsrahmen angelegt, der u.A. die Rücksprungadresse enthält (return);
2. Werden alle lokalen Variablen

> Die dynamischen Daten eines Unterprogrammaufrufs werden auf dem Stapel als Teil des sogenannten Aufrufrahmens bzw. Aktivierungsrahmens (stack frame) gespeichert. Sie stehen dort nur für die Dauer der Unterprogrammausführung zur Verfügung.

Funktionsweise des Stapels und Stapelzeigers (SP)

- Der bei Unterprogrammaufrufen benutzte Aktivierungsrahmen betrifft direkt die Programmierschnittstelle des Betriebssystems und die Prozess-/Thread-Verwaltung!
- Der Stapelspeicher (stack) besteht genau genommen aus einem dafür reservierten Teil des Adressraums und dem Prozessorsteuerregister Stapelzeiger (Stack Pointer, SP).
  - Der Stapelzeiger enthält jederzeit die Adresse des aktuellen oberen Endes des Stapels (Top Of Stack, TOS).
  - Im Stapel selbst befinden sich die Aufrufrahmen (stack frames) aller nicht beendeten Unterprogrammaufrufe sowie eventuell andere zwischengespeicherte Werte. Entsprechend sieht man oft den Begriff des call stack, d.h. des Aufrufstapels.
  - Der Stapel wird im deutschen Sprachraum teilweise als Kellerspeicher bzw. Keller bezeichnet.

Funktionsweise des Stapels und Stapelzeigers (SP)

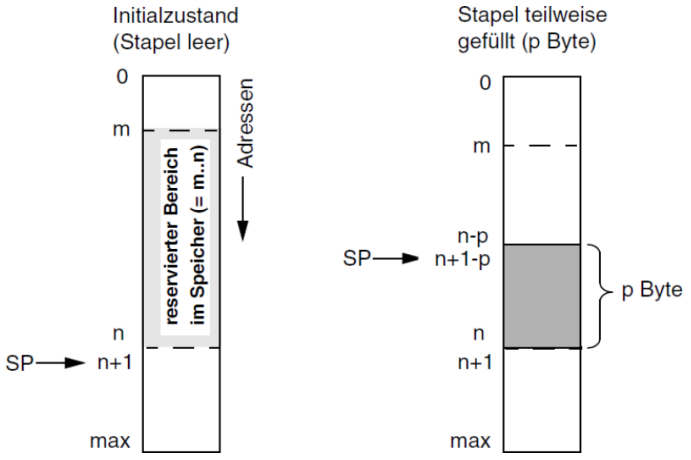


Abb. 13. Prinzip des Stapelspeichers

Funktionsweise des Stapels und Stapelzeigers (SP)

- Der Stapel stellt die Grundinfrastruktur für die Unterprogrammtechniken dar.
- Pro Unterprogrammaufruf wird ein Aktivierungsrahmen (stack frame), d.h. ein Datenbereich auf dem Stapel, angelegt.
- Bei Beendigung des Unterprogramms wird der Datenbereich wieder freigegeben, was auch als Abräumen bezeichnet wird. Ein Aktivierungsrahmen enthält eines oder mehrere der folgenden Elemente:
  - Rücksprungadresse des Unterprogramms (d.h. Adresse, an der im aufrufenden Programm nach Unterprogrammende fortzufahren ist)
  - Unterprogramm-Aufrufparameter (Funktionsargumente)
  - Unterprogramm-Rückgabewerte
  - Lokale Variablen (in C sog. automatic variables)
  - Nach Bedarf weitere zwischengespeicherte Werte (computersprachabhängig)

Unterprogrammaufruf

Jeder Universalprozessor stellt für den Aufruf von Unterprogrammen und die Rückkehr in das Oberprogramm spezifische Maschinenbefehle zur Verfügung, die diesen Vorgang automatisieren helfen.

- Wir bezeichnen nachfolgend diese zwei Maschineninstruktionen mit den Namen JSR (jump to subroutine) und RET (return from subroutine).
  - Abhängig vom benutzten Prozessor können diese Befehle etwas abweichend bezeichnet sein, arbeiten aber stets nach dem hier vorgestellten Verfahren.
- Eine wichtige Grundeigenschaft des Unterprogrammaufrufs besteht darin, dass egal woher ein Unterprogramm aufgerufen wird, stets an die dem Aufruf direkt nachfolgende Programmstelle zurückgekehrt wird, wenn das Unterprogramm endet.
  - Dies wird technisch mithilfe des Stapels realisiert, indem die Rücksprungadresse für die Dauer der Unterprogrammausführung auf dem Stapel zwischengespeichert wird.

Unterprogrammaufruf

Für den Unterprogrammaufruf dient der JSR-Befehl, der die Rücksprungadresse auf dem Stapel ablegt und den Programmzähler mit der Startadresse des Unterprogramms lädt. Die Rückkehr in das Oberprogramm wird durch den RET-Befehl veranlasst, indem er das oberste Element vom Stapel entfernt und als neuen Wert in den Programmzähler, womit der Programmablauf an der Rücksprungstelle fortfährt.

Unterprogrammaufruf

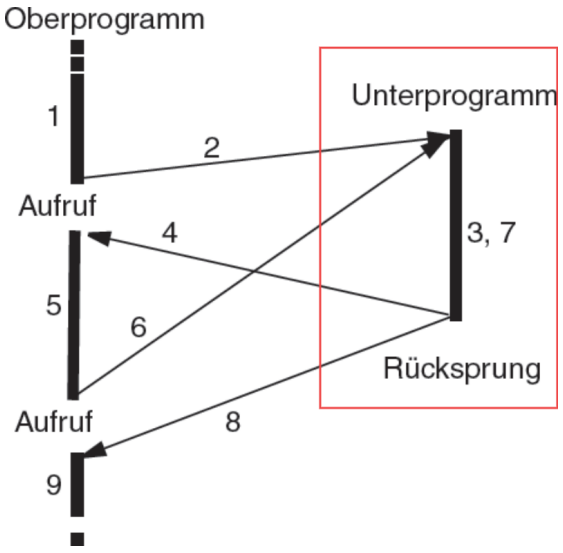


Abb. 14. Verwendung von Unterprogrammen (Beispiel)

Aufruf

Maschinenbefehl JSR

- sichert Rücksprungadresse auf Stapel
- lädt PC mit UP-Adresse

Rücksprung

Maschinenbefehl RET

- holt gesicherte Rücksprungadresse vom Stapel und lädt sie in den Programmzähler PC (damit: Rückkehr zu OP)

OP: Oberprogramm
UP: Unterprogramm

Unterprogrammaufruf

Durch die Verwendung des Stack Speichers und der LIFO Semantik wird auch Funktionsrekursion und geschachtelte Azfrufe unterstützt!

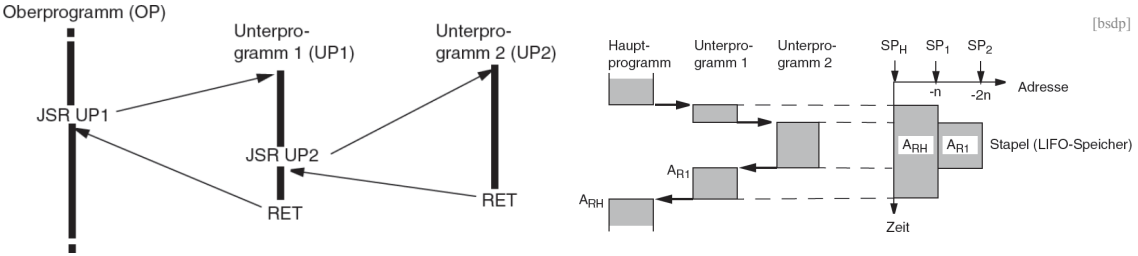


Abb. 15. Geschachtelter Unterprogrammaufruf (Beispiel)

Paging und Segmentierung

Bisher gab es nur einen gesamten Speicherbereich mit der Unterteilung in die Bereiche Programm, statische Daten, dynamische Daten und temporäre Daten.

- In einem Betriebssystem werden (außer bei Mikrokontrollerrechnern) mehrere Prozesse, d.h. Programme in Ausführung, in einem Multiplexbetrieb quasi parallel betrieben.
  - Jetzt muss jeder Prozess eigene Speichersegmente besitzen

Es bedarf jetzt einer hierarchischen Speicherverwaltung: 1. Segmente / alle Prozesse, 2. Speicherbereiche in Segmenten eines Prozesses.

Paging und Segmentierung

- Aber halt: schauen wir uns von einer Programmdatei die Symboltabelle an so finden wir z.B.:

```
objdump ball.exe  -h

ball.exe:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00006c68  80000000  80000000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       0000000b  80007000  80007000  00008000  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .eh_frame     00001038  8000700c  8000700c  0000800c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .data         00000004  8000a000  8000a000  0000a000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  4 .bss          00001238  8000a020  8000a020  0000a004  2**5
                  ALLOC
```

Bsp. 2. Die Segmente eines Programms bevor es in den Speicher geladen wird. VMA: Virtual Mmeory Address, LMA: Linear Memory Address. Stack und Heap gibt es hier noch nicht! .text: Programmcode, .rodata: Konstante Daten, .data: statische globale und initialisierte Daten, .bss: statische globale nicht initialisierte Daten

[bsdp]

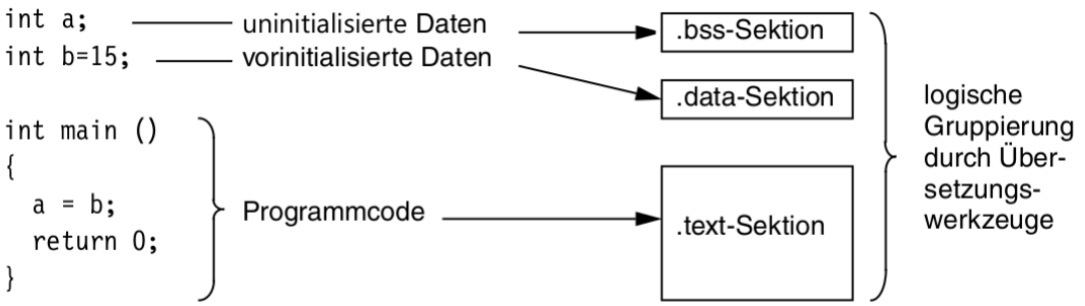


Abb. 16. Gruppierung verschiedener Programmteile

Paging und Segmentierung

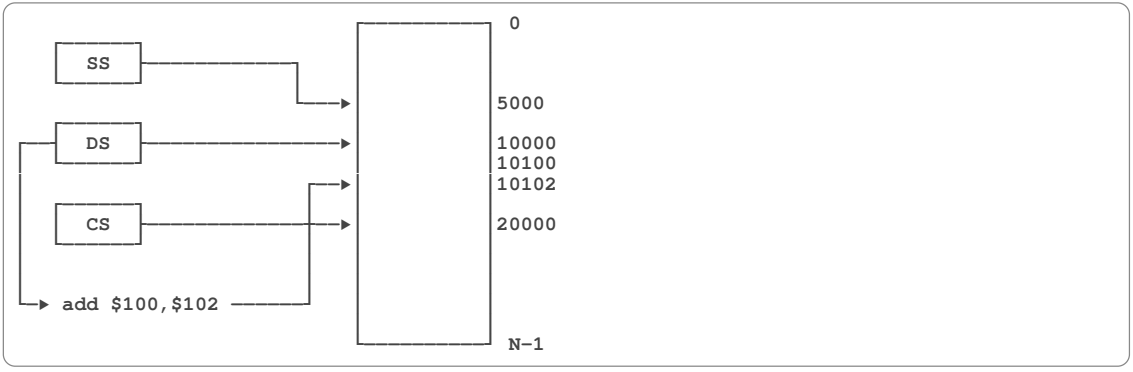
- Würde man alle Programme nach dieser Anweisung in den Speicher laden würden sich alle Programme im Speicher überdecken.
  - Man könnte alle Programmadressen umrechnen (Offset), aber das müsste man nicht nur für die Segmente machen, sondern für alle Maschineninstruktionen die mit absoluten Adressen arbeiten. Keine gute Idee.
  - Man könnte **Virtuelle Speicherräume** und Virtuellen Speicher einführen. Gute Idee, aber...

Es nützt uns nichts wenn wir (oder das Betriebssystem) die physikalischen Speicheradressen (P Raum) auf virtuelle Adressen (für jeden Prozess, der V Raum) abbilden. Die Hardware muss das tun, und das geschieht durch die Memory Management Unit (MMU).

- Die MMU arbeitet mit Tabellen die Adressen von Segmenten vom V in den P Raum transformieren. Damit die Tabellen um Umfang nicht zu groß werden werden Speichersegmente nur in Seiten (Pages) verwaltet, und Page Units (Größe z.B. 4096 Bytes).

Paging und Segmentierung

- Dynamische Segmentierung kann auch direkt vom Mikroprozessor unterstützt werden, so dass kein virtueller Speicherraum benötigt wird.
- Dazu werden für einen Speicherzugriff Segmentregister verwendet, die dann den oberen Teil einer Speicheradresse zusammen mit der von einer Maschineninstruktion gelieferten Adresse zusammengerechnet werden:



Bsp. 3. Die Programmadresse ergibt sich aus PC+CS, die Datenadressen aus DS+$addr, und die Stackadressen aus SS+SP

Verwaltung von Prozessadressräumen

- Dynamische Segmentierung kann erfolgen durch:
  - MMU (Tabellen)
  - Prozessor (Register)
- Egal welche Methode, bei jedem Prozesswechsel müssen die Tabellen oder Register umgeladen werden.
- Nachdem ein Programm übersetzt wurde, steht es dem Betriebssystem in Form einer ausführbaren Datei zur Verfügung.
  - Beim Laden eines Programms müssen Bereiche im Adressraum zum Beispiel für Code, Daten, Heap und Stack reserviert werden.
  - Ein laufendes Programm kann zudem Threads erzeugen, gemeinsame Speicherbereiche mit anderen Prozessen einrichten und weitere Aktionen ausführen, die wiederum Bereiche im Prozessadressraum betreffen.

Das Betriebssystem muss über alle diese Belegungen des Adressraums Buch führen. Nur so ist es in der Lage, neue Reservierungen ohne Kollisionen mit bestehenden Belegungen durchzuführen.

Verwaltung von Prozessadressräumen

- Prozessadressräume sind also i.A. virtuell, so dass jeder Prozess aus seiner Sicht immer die gleiche "Startadresse" hat.
  - Vorteile: Einfache Kompilierung und Mehrfachausführung einer Programmdatei ist möglich
  - Nachteile: Hardwareunterstützung (MMU), Verwaltungsaufwand, Prozesswechsel
- Segmente sind Speicherregionen die vom Betriebssystem verwaltet werden müssen (Partitionierung des physikalischen Speichers).
  - Auch hier muss zwischen Anwendungsprozessen und dem Betriebssystemprozess unterschieden werden. Der virtuelle Adressraum ist auch zu partitionieren:

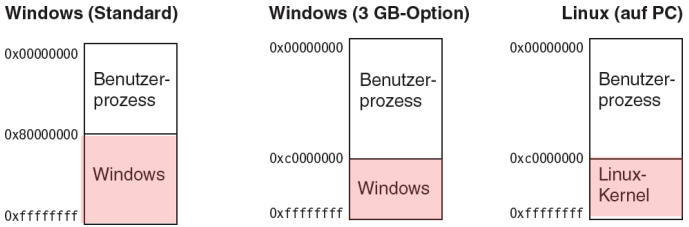


Abb. 17. Beispiele von Adressraumaufteilungen

Verwaltung von Prozessadressräumen

Für jeden Prozess benötigt das Betriebssystem eine Buchhaltung der im Adressraum belegten und freien Bereiche. Da bei vielen Prozessen der Adressraum nur schwach belegt ist, wird typischerweise nur die Belegung festgehalten.

- Dies ist die Voraussetzung dafür, dass sich während des Prozessablaufs zusätzliche Bereiche kollisionsfrei reservieren lassen.
- Steht eine passende Hardwareunterstützung (MMU) zur Verfügung, so können zudem Fehlzugriffe auf unbelegte Bereiche festgestellt werden.

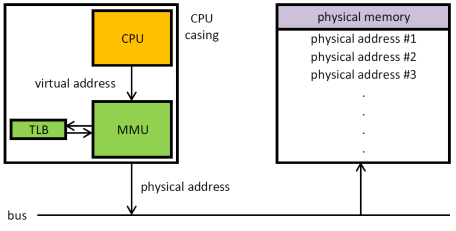


Abb. 18. Funktionsweise einer MMU+CPU Architektur

Dynamische Speicherreservierung von Regionen oder Erweiterungen

- Anlässe für nachträgliche Reservierungen können sein:
  - **Thread**-Erzeugung: Es wird ein Stack-Bereich bereitgestellt (evtl. sogar zwei Stack-Bereiche, getrennt für Benutzer- und Kernmodus).
  - Einrichten gemeinsamen Speichers (**shared memory**): Es wird ein Adressbereich bereitgestellt, der den Datenaustausch zwischen mehreren Prozessen ermöglicht.
  - Laden einer **Bibliotheksdatei**: Ihr Inhalt wird geladen und in einem Bereich des Adressraums sichtbar gemacht.
  - Gemeinsame Bibliothek (**shared library**) nutzen: Von mehreren Prozessen gemeinsam genutzte Bibliotheken werden einmal geladen und für alle beteiligten Prozesse sichtbar gemacht.
  - Speicherbasierte Datei einrichten (**memory mapped file**): Der Dateiinhalt oder Teile davon werden über einen Bereich im Adressraum sicht- und änderbar gemacht.

Regionen

Im Adressraum belegte Bereiche werden Regionen genannt. Sie repräsentieren lückenlos zusammenhängende Adressbereiche mit unterscheidbaren Eigenschaften. Wichtige **Attribute** einer Region sind:

- **Startadresse**: Dies ist die Adresse, mit der die Region beginnt (niedrigster Adresswert)
- **Größe**: Die Anzahl zugehöriger Adresswerte (als Anzahl Byte)
- **Schutzattribute**: Les-, schreib- oder ausführbar bzw. Kombinationen davon (die Durchsetzung benötigt spezielle Hardwareunterstützung)
- **Zugehöriger Hintergrundspeicher**: Dateiname und Position innerhalb der Datei (kann undefiniert sein; benötigt zudem Softwareunterstützung)

Für das Suchen von Lücken im Adressraum sind nur die Attribute 1 und 2 von Belang. Wird ein virtuelles Speichersystem realisiert, so erlaubt das Attribut 3 die Erkennung von Fehlzugriffen und das Attribut 4 die Zuordnung zu einer Datei. Eine Region kann einem Hintergrundspeicher zur Auslagerung zugeordnet sein

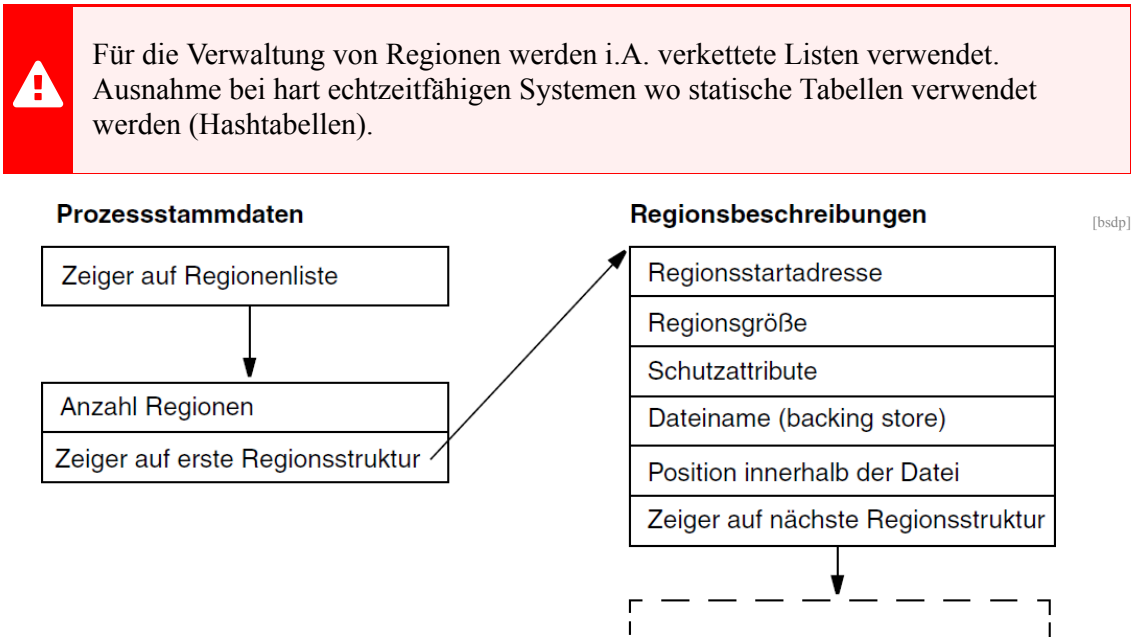


Abb. 19. Regionen pro Prozess

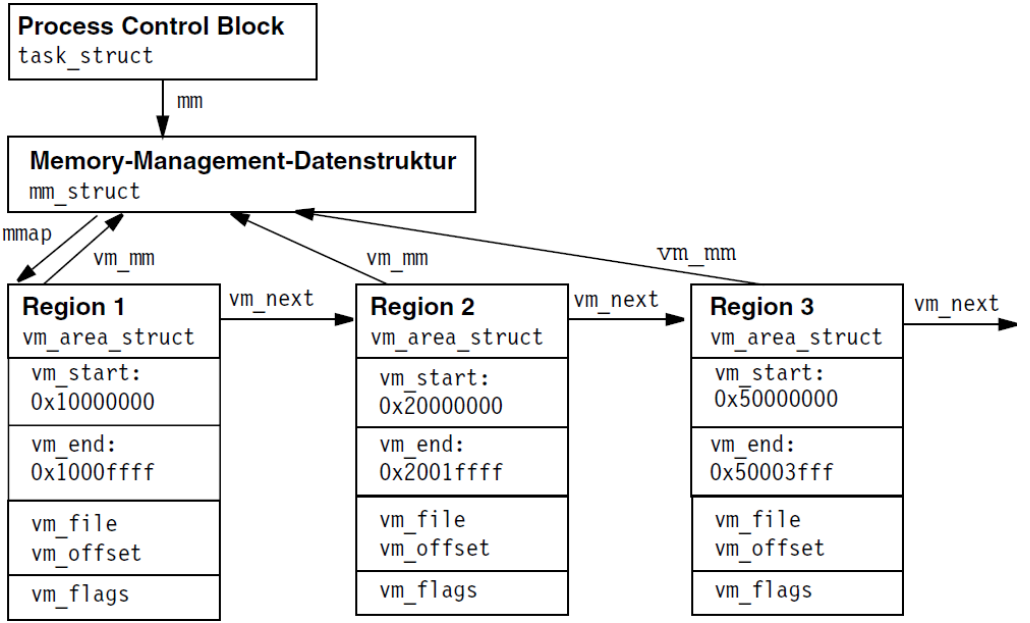


Abb. 20. Prozessadressraumbeschreibung unter Linux (Beispiel)

... wenn wir schon virtuelle Speicherräume haben:

Swapping

Die Idee liegt darin, ganze Prozesse oder Speicherbereiche auszulagern, wenn sie im Moment keine Rechenzeit benötigen. Für diesen Zweck wird auf der Festplatte ein spezieller Bereich (Swapping Area) reserviert.

- In seiner Grundform basiert das Swapping auf der Multiprogrammierung mit festen Partitionen, die um das Ein- und Auslagern von Prozessen erweitert wird.
- Das Swapping wird vor allem dann benutzt, wenn mehrere Benutzer auf einem Rechner arbeiten (z.B. ältere Unix-Systeme).
  - Bei sehr vielen gleichzeitigen Benutzern kann es dann passieren, dass das Betriebssystem nur noch mit Swappen beschäftigt ist. In dieser Situation könnte es hilfreich sein, wenn das Betriebssystem den Grad der Multiprogrammierung reduziert bzw. beschränkt, was jedoch in Dialogsystemen unakzeptabel ist.
  - In einem Stapelverarbeitungssystem wäre hingegen das längere Anhalten eines Prozesses kein Problem, womit für die übrigen Aufträge ausreichend Speicher verfügbar wäre.

Swapping

Abb. 21. Swapping Prinzip

- Bei der Partitionierung von Speicher legt man Regionen vorab fest die dann an Prozesse vergeben werden. Mit Swapping findet dann Austausch zur Laufzeit statt.

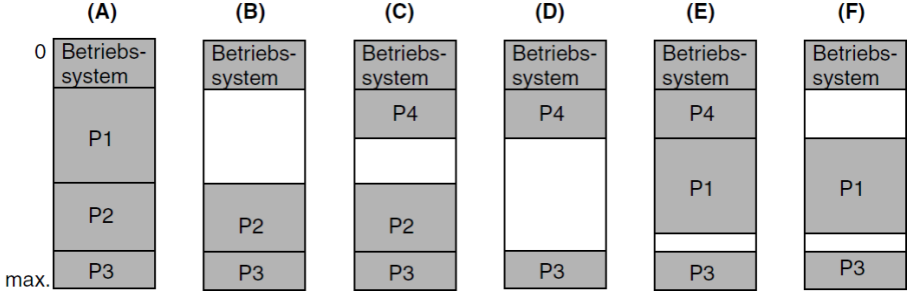


Abb. 22. Dynamische Partitionierung beim Swapping (Beispielsabfolge)

Swapping

Idee gut, aber zeitintensiv, gerade wenn ein Prozess nur kurz rechnet ...

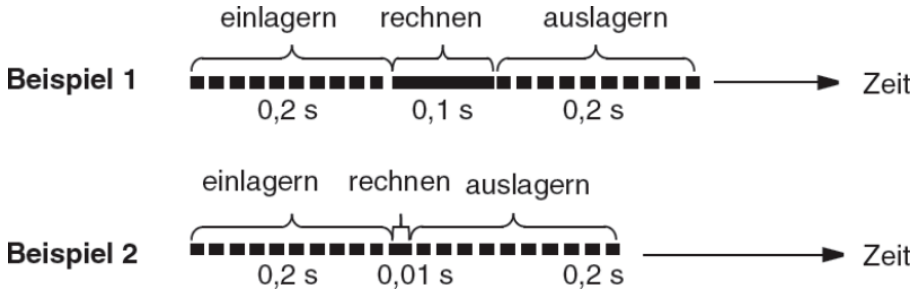


Abb. 23. Beispiele für Zeitbedarfe beim Swapping-Verfahren

Virtueller Speicher

Zusammengefasst sind dies die wesentlichsten Eigenschaften:

- Jeder Prozess kann den Adressraum fast beliebig belegen, d.h. teilt ihn höchstens mit dem Betriebssystem selbst.
- Jeder Prozess ist gegen Fehlzugriffe aller anderen Prozesse geschützt (Schreibund Leseschutz!), auch das Betriebssystem ist geschützt.
- Alle Prozesse sind von der vorhandenen Hauptspeichergröße weitgehend unabhängig (mehr Adressraum als real vorhanden).

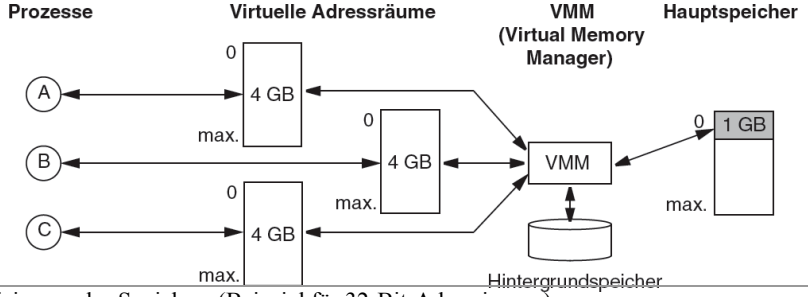


Abb. 24. Virtualisierung des Speichers (Beispiel für 32-Bit-Adressierung)

Virtueller Speicher

Flexibilität ...

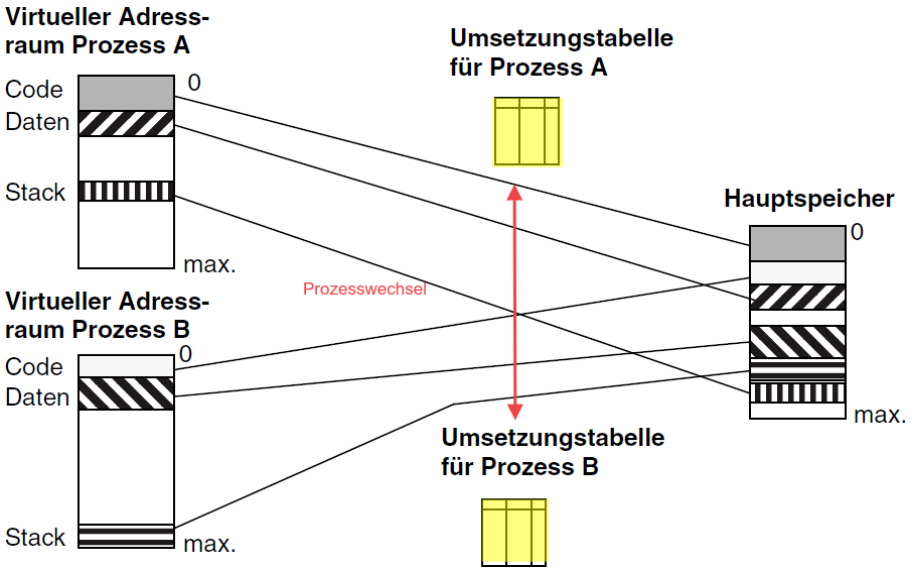


Abb. 25. Flexible Platzzuteilung dank Adressumsetzung.

Zusammenfassung

1. Jedes Programm besteht aus verschiedenen Speicherbereichen: Heap, Stack, Datenbereiche, Code

2. C bildet das Prozesspeichermodell sehr konkret durch globale, lokale, und temporäre Variablen sowie Funktionen ab, und ermöglicht den Speicherzugriff über Zeiger ("Specihermodell")

3. Die einzelnen Programmbereiche können auf verschiedene Regionen im Speicher abgebildet werden.

4. Dynamische Speicherverwaltung benötigt Datenstrukturen die belegte und freie Regionen verwalten, der Prozess kann ebenfalls in seinen Regionen eine solche Speicherverwaltung haben (Verwaltungshierarchie)

5. Das Problem von Adresskollision kann durch Virtuelle Adressraume gelöst werden, die über eine MMU oder einen VMM auf physikalische Adressen abgebildet werden.

6. Was noch fehlt: Wie wird Speicher effizient verwaltet? Also die Algorithmik . coming soon.