Betriebs-systeme II

FH München
WS 2006/07
Hans-Georg Eßer
hans-georg.esser@fhm.edu

Die Folien zu „Betriebssysteme II“ basieren auf den Ausarbeitungen von
Prof. Christian Vogt, http://www.cs.fhm.edu/~vogt und
Prof. Claudius Schnörr, http://www.cs.fhm.edu/~schnoerr

Content Overview

- Contiguous memory allocation
  - fix-sized partitions
  - variably-sized partitions
  - methods for administration of free memory
  - segmentation
- Non-contiguous memory allocation
  - Virtual memory management (paging)
  - multi-level paging
  - segmentation plus paging
- Demand paging
  - Page Faults and what to do about them
  - page replacement strategies
  - further design possibilities
- Swapping

Motivation

- What does a memory address consist of?
- What happens when accessing an address?
- How can administrators and developers use their knowledge about memory management?
  - How does Shared Memory work?
  - What are memory-mapped files?
- How does a Segmentation Fault occur?
- How to create a virtual address space?

Kinds of memory management

Two principle types

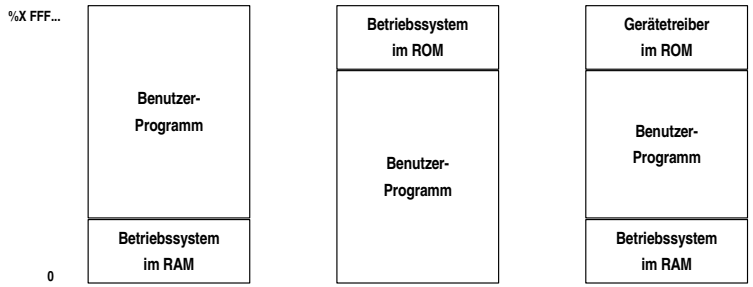
- Contiguous memory management
  - Whenever a process requests some memory, OS must satisfy this request with a contiguous memory block
- Non-contiguous memory management
  - OS can reserve several smaller memory blocks, giving the requested size when added up
  - Searching the (distributed) process' memory is an OS task.

Kinds of memory management

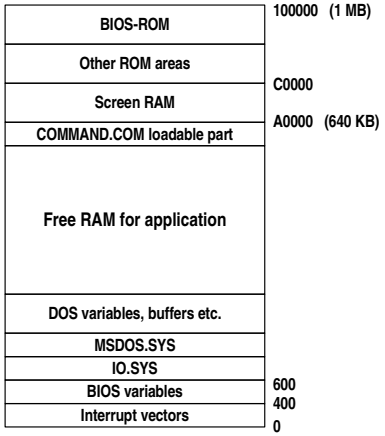
- Today: Main memory almost always non-contiguous (virtual memory management).
- Cases of contiguous memory management:
  - disk space management,
  - management of space in page and swap files

Single tasking without swapping or paging

- Divide memory (RAM and ROM) into partitions for
  - the operating system
  - a (one) user program
- When program finishes: Reuse its memory area for next program



Memory usage in MS-DOS

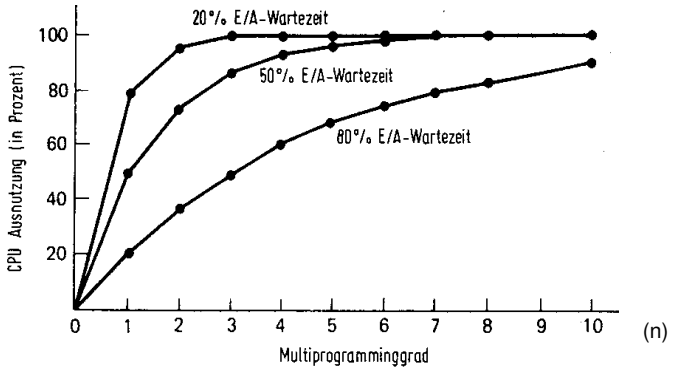


Multiprogramming

- Programs spend a lot of their time waiting (for I/O etc.).
- Swapping to disk every time is inefficient.
- Solution: several programs in main memory simultaneously
- Preconditions: movable (relocatable) code and memory protection

Multiprogramming

- When *n* programs spend a fraction *p* of their time waiting, then CPU usage is = $1-p^n$.



Code relocation: Z80 assembler

```
#base 0xA000

ld hl,data
ld a,(hl)
call print
ret

:print
cmp a,' '
jne ende
call OS_PRINT
:ende
ret

:data
char 'x'
```

```
        ; Program starts at 0xA000

A000    ld hl,0xA011   ; data
A003    ld a,(hl)
A004    call 0xA008    ; print
A007    ret

        ; Function print
A008    cmp a,' '
A00A    jne 0xA010     ; ende
A00D    call OS_PRINT
        ; Label ende
A010    ret

        ; Data
A011    char 'x'
```


Relocator for Z80, 19 years old

[no, no, I won't translate this slide...]

(Nicht ganz ernst gemeinter) Literaturhinweis:

H.-G. Eßer: *MC-Relocator – Programmverschiebung leicht gemacht*, Schneider CPC International 03/1988, DMV-Verlag, S. 67 f.

```
für 464-664-6128

1000 ' ****************************          [1164]
1010 ' ***                          ***      [51]
1020 ' ***      RELOCATOR V1.0      ***      [918]
1030 ' ***                          ***      [51]
1040 ' ***                          ***      [51]
1050 ' ***    Geschrieben im        ***      [2098]
1060 ' ***    Oktober 1987 von      ***      [960]
1070 ' ***                          ***      [51]
1080 ' ***    Hans-Georg Esser      ***      [697]
1110 ' ***                          ***      [51]
1120 ' ****************************          [1164]
1130                                          [117]
1140 MODE 2:ZONE 3                            [1209]
1150 WINDOW #1,48,76,7,10                     [1139]
1160 WINDOW #2,48,76,12,18                    [1223]
1170 FOR N=1 TO 2:PEN #N,0:PAPER #N,1:NEXT    [2164]
1180 PRINT CHR$(24);STRING$(80,216);"         [8496]
 Relocator V1.0      (c) 1987 by Hans-Geor
g Esser      Relocator V1.0      ";STRING$
(80,216);CHR$(24)
1190 LOCATE 1,22                              [730]
1200 PRINT CHR$(24);STRING$(80,216);"         [8529]
 Relocator V1.0      (c) 1987 by Hans-Geor
g Esser      Relocator V1.0      ";STRING$
(80,216);CHR$(24);
1210 WINDOW 1,48,5,20                         [1040]
1220 CLS #1:CLS #2                            [576]
1230 LOCATE #1,2,2:PRINT #1,"Aktuelle Adre    [2624]
sse:"
1240 LOCATE #2,2,2:PRINT #2,"Geaenderte Ad    [3236]
ressen:":PRINT #2
1250 LOCATE 5,3:LINE INPUT "Name des Binae    [4640]
r-Files     ) ",file$
1260 LOCATE 5,5:INPUT "Ursprungsadresse       [4581]
     ) ",adr1
1270 LOCATE 5,7:INPUT "Programm-Laenge (in     [3793]
 Bytes) ) ",length
1280 LOCATE 5,9:INPUT "Neue Adresse            [3384]
     ) ",adr2
1290 LOCATE 5,11:LINE INPUT "Name des neue     [2549]
n Files     ) ",file2$
1300 LOCATE 5,14:PRINT "Relokation wird du     [5399]
rchgefuehrt."
1310 ' Einlesen                               [1453]
1320 MEMORY adr1-1                            [108]
1330 LOAD file$,adr1                          [518]
1340 diff=adr2-adr1                           [566]
1350 ' Alle Adressen muessen um diff nach     [4774]
oben verschoben werden.
1360 FOR n=adr1 TO adr1+length                [2180]
1370 LOCATE #1,20,2:PRINT #1,"&";HEX$(n,4)    [2013]
;
1380 GOSUB 1480 ' Ueberpruefen und ggf. ae    [2409]
ndern
```

```
1390 NEXT n                                   [366]
1400 dum$="dummy.dum"                         [1270]
1410 SAVE dum$,b,adr1,length                  [1587]
1420 MEMORY adr2-1                            [128]
1430 LOAD dum$,adr2                           [683]
1440 !ERA,@dum$                               [988]
1450 SAVE file2$,b,adr2,length                [1961]
1460 LOCATE 5,14:PRINT "Die Relokation wur    [5148]
de durchgefuehrt."
1470 END ' Programm-Ende                      [882]
1480 ' Aktuelle Adresse ueberpruefen          [933]
1490 pr$=HEX$(PEEK(n),2)                      [1242]
1500 byte=VAL("&"+HEX$(PEEK(n+2),2)+HEX$(P    [2760]
EEK(n+1),2))
1510 RESTORE 1830                             [920]
1520 rd$=""                                   [183]
1530 WHILE rd$<>"XX"                          [1942]
1540 READ rd$                                 [483]
1550 IF pr$=rd$ THEN 1640                     [765]
1560 WEND                                     [390]
1570 ' Untersuchung auf 2-Byte-Befehl         [2528]
1580 pr$=HEX$(PEEK(n),2)+HEX$(PEEK(n+1),2     [1841]
1590 byte=VAL("&"+HEX$(PEEK(n+3),2)+HEX$(P    [1647]
EEK(n+2),2))
1600 RESTORE 1840                             [886]
1610 rd$="":WHILE rd$<>"XXXX":READ rd$:IF     [2416]
rd$=pr$ THEN 1730
1620 WEND                                     [390]
1630 RETURN                                   [555]
1640 ' Bezug auf Adresse innerhalb des Pro    [3010]
gramms ?
1650 IF byte<adr1 OR byte>adr1+length THEN    [2412]
 RETURN
1660 ' Verschieben                            [815]
1670 byte=byte+diff                           [564]
1680 b$=HEX$(byte,4)                          [1060]
1690 POKE n+2,VAL("&"+LEFT$(b$,2))            [1198]
1700 POKE n+1,VAL("&"+RIGHT$(b$,2))           [1323]
1710 PRINT #2," &";HEX$(n,4),                 [2014]
1720 RETURN                                   [555]
1730 ' 2-Byte Befehl auswerten                [1642]
1740 IF byte<adr1 OR byte>adr1+length THEN    [2412]
 RETURN
1750 ' Verschieben                            [815]
1760 byte=byte+diff                           [564]
1770 b$=HEX$(byte,4)                          [1060]
1780 POKE n+3,VAL("&"+LEFT$(b$,2))            [892]
1790 POKE n+2,VAL("&"+RIGHT$(b$,2))           [764]
1800 PRINT #2," &";HEX$(n,4),                 [2014]
1810 RETURN                                   [555]
1820 ' Daten                                  [442]
1830 DATA CD,DC,FC,D4,C4,F4,EC,E4,CC,C3,DA    [6017]
,FA,D2,C2,F2,EA,E2,CA,E9,3A,01,11,21,2A,31
,32,22,XX
1840 DATA ED48,ED5B,DD21,DD2A,FD21,FD2A,ED    [3306]
7B,ED43,ED53,DD22,FD22,ED73,XXXX
Listing Relocator
```


Code relocation: Z80 assembler

Problems with relocation:

- What is an address?
  - jump (branch) and call targets
  - accesses to data areas
- What just happens to look like an address?
- Multiply indirect addressing
  („load the value from the memory location whos address is in this memory location“)

Better: Loader plans relocation in advance, memorizes where addresses occur

Code relocation and memory protection (1/2)

- Program must be able to execute no matter where in the memory it is loaded.

  Two possibilities:
  - Linker remembers which code blocks contain absolute references. When loading the program it modifies them.
  - Machine has a special hardware register, a basis register

  Every time an address is used (at runtime), the basis register is added to the address.

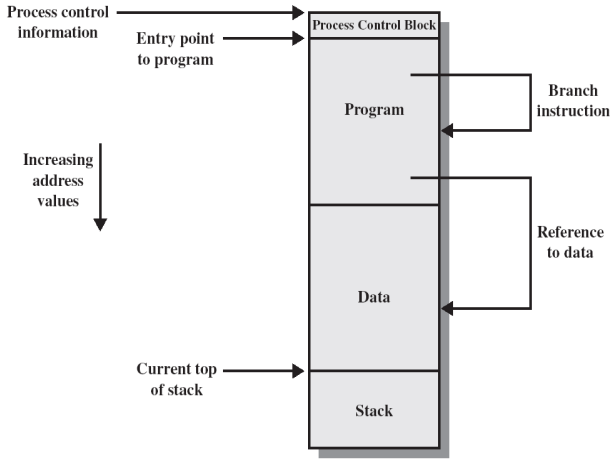
Code relocation and memory protection (2/2)

- Programs must not access other processes' memory areas.

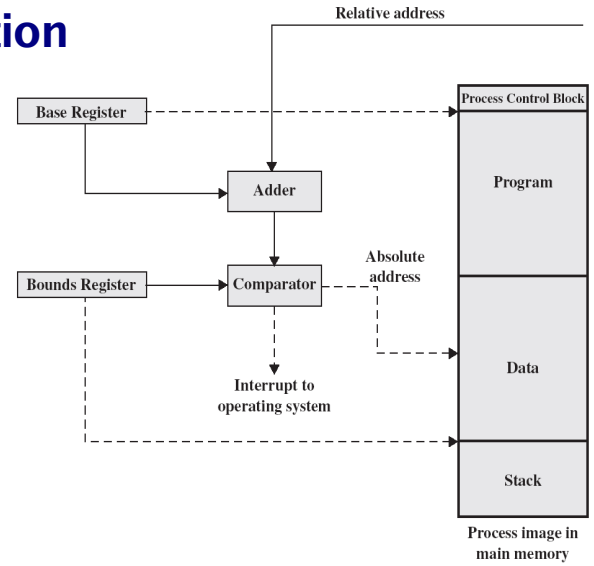
  Two ways to assure this:
  - protection code
  - Machine has (yet another) hardware register: limit register

  By checking the limit register OS can find out if the memory access is valid (i.e. inside the reserved address area).

Code relocation



Support for code relocation and memory protection



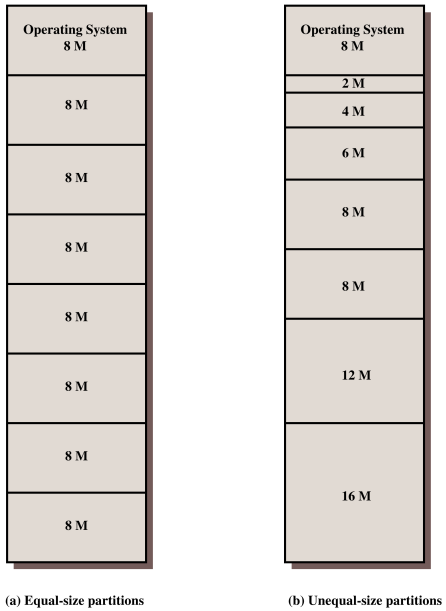
Contiguous Memory Management

Split into fixed partitions

- Create memory partitions of fixed (equal or different) size.
- Assign a process to a free partition. Alternatives:
  - First program that fits in the free partition (one queue)
  - FIFO for each partition (several queues)
  - Largest program that fits in the free partition
    - Disadvantage: small programs postponed
    - Solution: postpone program no more than *k* times

Split into fixed partitions

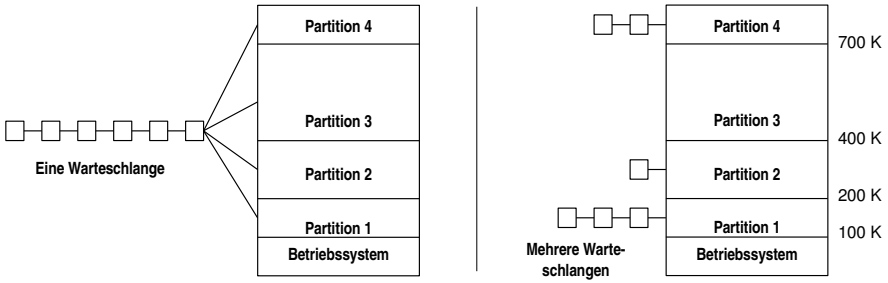
- **Equal size**
  - Waste memory when there are many small programs
  - Programs fit in any partition
- **Unequal size**
  - Better memory usage
  - Possibly inapt allocation



(a) Equal-size partitions (b) Unequal-size partitions

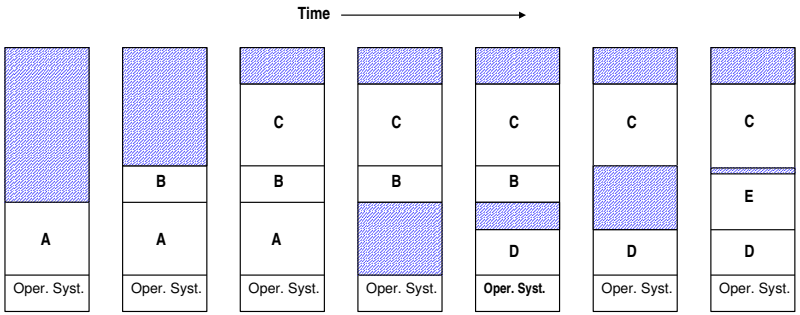
Split into fixed partitions

- Large free spaces within a partition can occur: internal fragmentation.



Split into variable partitions (1)

- Configure number and size of partitions dynamically.



Split into variable partitions (2)

- Possibly many small areas (holes) in memory remain unused.
  external fragmentation.
- Move partitions to get rid of the holes (memory compaction).

Split into variable partitions (3)

What happens when process needs more memory?

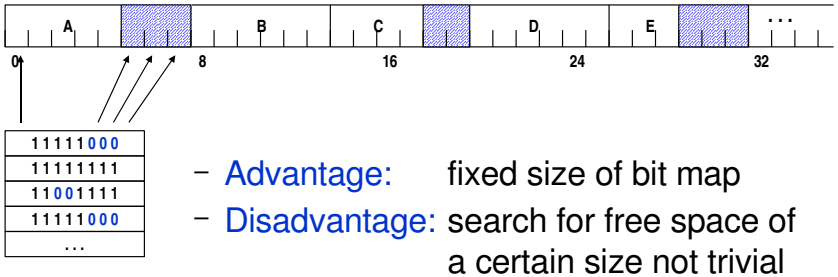
- Increase partition size if a neighboring partition is free.
- Allocate a larger (free) partition and move program there.

Split into variable partitions (4)

- When there is no sufficiently large partition, one or several process must be swapped to disk.
- Alternative: Allocate more memory than was requested by the process. Leads to
  - internal fragmentation
  - If larger memory block is not sufficient either, one of the other methods must be applied.

Which parts of the memory are free? a) Bit Maps

- partition memory in units (some bytes to several KByte)
- One bit per unit: 0 when unit is free
  1 when unit is in use



- Advantage: fixed size of bit map
- Disadvantage: search for free space of a certain size not trivial

Which parts of the memory are free? b) Linked Lists

- Chained list with descriptions of memory areas:
  - used by a process (P) or free (H=hole)
  - start address and length of area
  - pointer to next entry