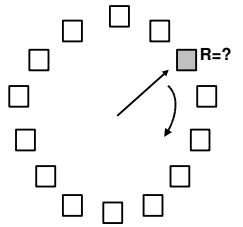
Speicherverwaltung (5)

Second-Chance-Algorithmus (1)

- Modifikation des FIFO-Algorithmus:
  Ist bei der Seitenersetzung das Referenz-Bit der ältesten Seite gesetzt, so wird
  - das Referenz-Bit gelöscht und die Seite am Ende der Liste eingereiht,
  - die gleiche Prüfung für die nächstälteste Seite durchgeführt.
- Es wird also
  - die älteste Seite ersetzt, deren Referenz-Bit gelöscht ist,
  - einer kürzlich benutzten Seite zunächst eine „zweite Chance“ gegeben.

Second-Chance-Algorithmus (2)

- Einfachere Implementierung: „Uhrzeiger“
  - Anordnung der Seiten in einer Ringliste, und Verschieben eines Zeigers statt Umpositionieren eines Listenelements.



Überprüfen der Seite, auf die der Zeiger zeigt:

- Ist R=0, wird die Seite ersetzt und der Zeiger weiterbewegt.
- Ist R=1, wird R gelöscht, der Zeiger weiterbewegt und die nächste Seite überprüft.

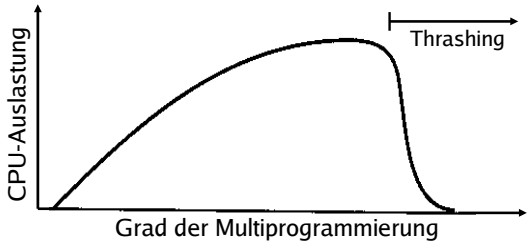
Seitenersetzung, Beispiele

Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
OPT	2	2 3	2 3	2 3 1	2 3 5 F	2 3 5	4 3 5 F	4 3 5	4 3 5	2 3 5 F	2 3 5	2 3 5
LRU	2	2 3	2 3	2 3 1	2 5 1 F	2 5 1	2 5 4 F	2 5 4	3 5 4 F	3 5 2 F	3 5 2	3 5 2
FIFO	2	2 3	2 3	2 3 1	5 3 1 F	5 2 1 F	5 2 4 F	5 2 4	3 2 4 F	3 2 4	3 5 4 F	3 5 2 F
CLOCK	2*	2* 3*	2* 3*	2* 3* 1*	5* 3 1 F	5* 2* 1 F	5* 2* 4* F	5* 2* 4*	3* 2 4 F	3* 2* 4	3* 2 5* F	3* 2* 5*

Thrashing (1)

- Thrashing bedeutet, dass ein Prozess exzessiv viele Page Faults macht (alle paar tausend Instruktionen).
- Thrashing entsteht, wenn ein Prozess mehr Seiten aktiv benutzt, als ihm Seitenrahmen zur Verfügung stehen.

Thrashing mehrerer Prozesse führt zu niedriger CPU-Auslastung:



Thrashing (2)

Lösungen

- Falls noch freier Speicher vorhanden: Zuteilung weiterer Seitenrahmen an den betreffenden Prozess (z. B. durch dynamische Working-Set-Anpassung oder globale Ersetzungsstrategie).
- Falls kein freier Speicher mehr: Auslagern (Swapping) von Prozessen.

Weitere Design-Möglichkeiten (1)

- Asynchrones Auslagern modifizierter Seiten:
  - Modifizierte, ersetzte Seiten werden z. B. erst dann ausgelagert, wenn die Platte mit dem Swap-Bereich wenig benutzt ist oder wenn der Vorrat freier Seiten zu klein wird.
  - Modifizierte, nicht ersetzte Seiten werden vorab ausgelagert und ihr Modify-Bit gelöscht.
- Vorrat an freien Seitenrahmen:
  - Durch frühzeitige (proaktive) Seitenersetzung wird dafür gesorgt, dass bei der Anforderung einer Seite ein freier Seitenrahmen verfügbar ist.

Weitere Design-Möglichkeiten (2)

- Page Buffering (Aufbewahren des Inhalts einer ersetzten Seite):
  - Eine ersetzte Seite behält zunächst ihren Inhalt, wird jedoch dem Vorrat an freien Seitenrahmen zugeordnet bzw. zum Auslagern vorgemerkt.
  - Wenn der Prozess diese Seite kurz danach wieder anspricht, ist sie noch im Speicher und kann ohne Plattenzugriff wieder dem Prozess zugeordnet werden (soft page fault).
  - Es muss bekannt sein, ob die Seite zwischenzeitlich „wiederverwendet“, d. h. einem anderen Prozess zugeteilt wurde.

Weitere Design-Möglichkeiten (3)

- Prepaging:
  - Eine gewisse Anzahl von Seiten wird vorab (z. B. beim Aufruf eines Programms) in den Speicher gebracht, noch bevor sie von dem Prozess angesprochen werden.
- Clustering:
  - Bei einem Page Fault wird nicht eine einzelne Seite, sondern ein Cluster aus mehreren Seiten in den Speicher gebracht (spezielles Prepaging).
  - Modifizierte Seiten werden nicht einzeln, sondern in Clustern von mehreren Seiten auf Platte geschrieben.

Weitere Design-Möglichkeiten (4)

- Locking von Seiten im Speicher:
  - Einzelne Seiten eines Prozesses werden vom Paging ausgenommen.
  - Implementierung über einen (privilegierten) System Call.
  - Beim Swapping des ganzen Prozesses werden auch diese Seiten mit ausgelagert.
- Auch das Design der Programme hat Auswirkungen auf die Zahl der Page Faults und somit auf die Laufzeit des Programms selbst wie auch auf die Leistung des gesamten Systems!

Speicher unter Linux (1)

- Speicherverwaltung auf mehreren Ebenen:
  1. Nodes, `pg_data_t`, Prozessor + dessen Speicher
  2. Zonen:
     `ZONE_DMA`, `ZONE_NORMAL`, `ZONE_HIGHMEM`
  3. Jede Zone hat eine Tabelle (`zone_mem_map`), die alle Speicherseiten dieser Zone aufführt
  4. Seiten: `struct page`

Speicher unter Linux (2)

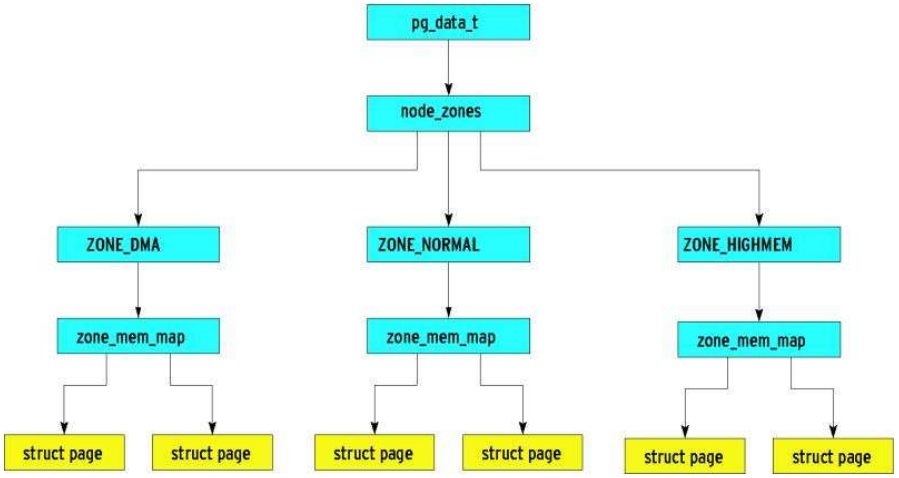


Bild: Linux-Magazin 09/2003

Speicher unter Linux (3)

- Auflösung virtueller Adressen in drei Stufen:
  - PGD: page global directory (Array vom Typ `pgd_t`)
  - PMD: page middle directory (Array vom Typ `pmd_t`)
  - PTE: page table entry (Array vom Typ `pte_t`)

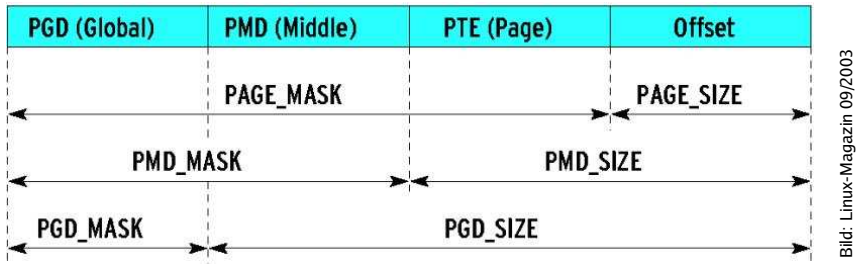


Bild: Linux-Magazin 09/2003

Speicher unter Linux (4)

- **Verwalten des freien Speichers**
  Linux verwendet Buddy-System; größte Blockgröße: 2^9 (= 512 Seiten = 2 MByte)
- **Rückgabe von Speicher**
  - Kernel 2.4:
    sofortiges „Verschmelzen“ der Buddies
  - Kernel 2.6:
    „lazy“ – erst mal abwarten, ob einer der kleinen Blöcke kurzfristig wieder genutzt wird

Speicher unter Linux (5)

- Prozess-Adressraum: `mm_struct` (in *linux/sched.h*)
- Jeder Prozess hat genau eine solche Struktur (mehrere Threads eines Prozesses nutzen die gleiche)
- Adressraum in zusammenhängende, nicht überlappende **Regionen** (`vm_area_struct`) unterteilt
- Anfang/Ende von Regionen auf Seitengrenzen
- Informationen zu Regionen mehrfach gespeichert: als Liste und als sog. Red Black Tree

Speicher unter Linux (6)

Red Black Tree:
binärer Suchbaum
mit folgenden Regeln:

- Knoten sind rot oder schwarz
- Wurzel ist schwarz
- Alle Blätter schwarz
- Beide Kinder jedes roten Knotens sind schwarz
- Für jeden Knoten gilt: Alle Pfade zu Blättern enthalten gleich viele schwarze Knoten

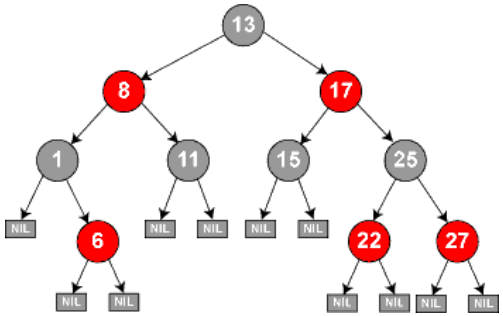


Bild: Wikipedia, http://en.wikipedia.org/wiki/Red-black_tree

-> weitgehend ausgeglichener Baum; effiziente Suche nach Adressen möglich

Speicher unter Linux (7)

```
struct mm_struct {
  struct vm_area_struct * mmap;       /* list of VMAs */
  struct rb_root mm_rb;
  struct vm_area_struct * mmap_cache; /* last find_vma result */
  pgd_t * pgd;
```

- `mmap`: verkettete Liste der Regionen (`mmap` ist der Listenkopf)
- `mm_rb`: Wurzel des Red-Black-Baums
- `mmap_cache`: Suche auch mit Red-Black-Baum aufwändig; hier Ergebnis der letzten Suche
- `pgd`: Das Page Global Directory für diesen Prozess

Speicher unter Linux (8)

```
struct mm_struct {
  [...]
  unsigned long start_code, end_code, start_data, end_data;
  unsigned long start_brk, brk, start_stack;
  [...]
  unsigned long rss, [...]
```

- `start_code/end_code`: Anfang und Ende der Code-Section
- `start_data/end_data`: Anfang/Ende der Data-Section
- `start_brk/brk`: Anfang/Ende des Heap
- `start_stack`: Anfang des Stacks
- `rss`: Anzahl der Prozessseiten, die im Speicher sind

Speicher unter Linux (9)

Speicher-Regionen:
`vm_area_struct` (in *linux/mm.h*)

```
struct vm_area_struct {
    struct mm_struct * vm_mm;
    unsigned long vm_start;
    unsigned long vm_end;

    /* linked list of VM areas per task, sorted by address */
    struct vm_area_struct *vm_next;

    pgprot_t vm_page_prot;
    unsigned long vm_flags;

    rb_node_t vm_rb;
    [...]
```

Speicher unter Linux (10)

```
struct vm_area_struct {
    [...]
    struct vm_area_struct *vm_next_share;
    struct vm_area_struct **vm_pprev_share;

    /* Function pointers to deal with this struct. */
    struct vm_operations_struct * vm_ops;

    /* Information about our backing store: */
    unsigned long vm_pgoff;
    struct file * vm_file;
    unsigned long vm_raend;
    void * vm_private_data;
};
```


Speicher unter Linux (11)

System Calls, die den Adressraum verändern

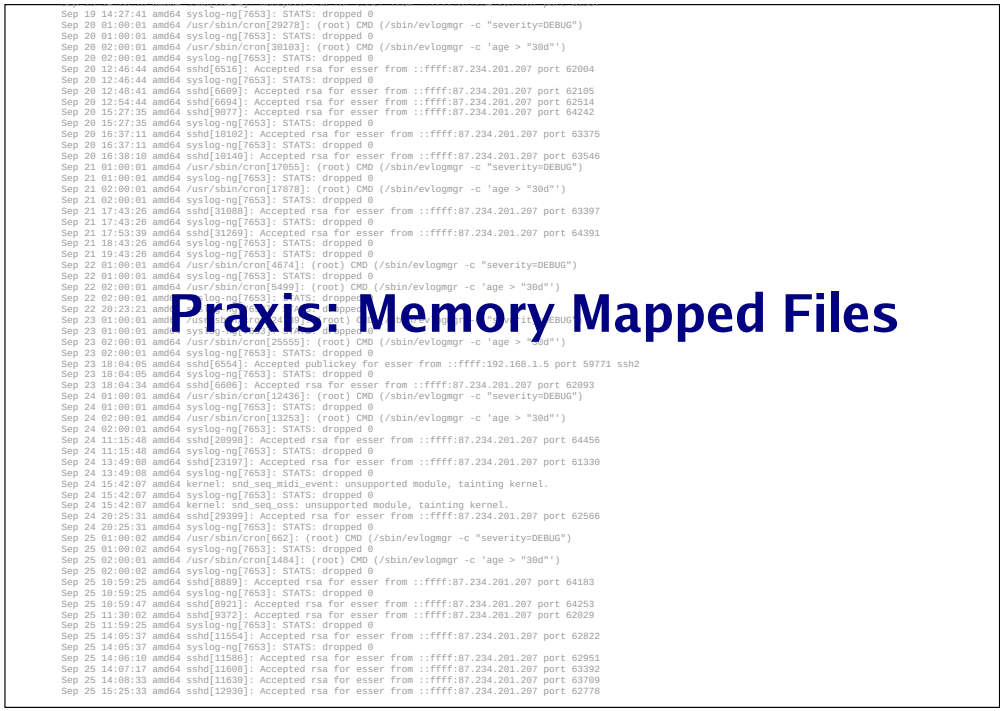
- Verändern der Größe der Datenregion mit
  - `brk(endds)` endds: höchste virtuelle Adresse der Datenregion (der sog. break value).
  - `oldendds = sbrk(increment)` Verändern des break value um `increment` Bytes.
- Der `fork` System Call beinhaltet
  - Duplizieren der prozess-privaten Regionen des Vaterprozesses,
  - Abbilden der neuen prozess-privaten sowie der gemeinsam benutzten (shared) Regionen in den Adressraum des Sohnprozesses.

Speicher unter Linux (12)

System Calls, die den Adressraum verändern

- Der `exec` System Call beinhaltet
  - das Aufheben der Abbildungen der bisherigen Regionen,
  - das Allokieren und Laden neuer Regionen,
  - das Abbilden der neuen Regionen in den Adressraum des Prozesses.
- Der `exit` System Call beinhaltet
  - das Aufheben aller Abbildungen von Regionen.

Praxis: Memory Mapped Files



Memory Mapped Files

- Idee: Inhalt einer Datei als Speicher einblenden
- Zugriff auf Datei anschließend über Variablen / Speicheradresse möglich,
  z. B. Abbildung einer ASCII-Datei auf
  - String-Variable (Python)
  - char* -Variable (C)
- Auslesen und Verändern der Datei durch Manipulieren der Speicherstellen
- Standard-POSIX-Funktion `mmap()`

Memory Mapped Files in C (1)

```
NAME
       mmap, munmap - map or unmap files or devices into memory

SYNOPSIS
       #include <sys/mman.h>

       void  * mmap(void *start, size_t length, int prot , int flags, int fd, off_t off-
       set);

       int munmap(void *start, size_t length);

DESCRIPTION
       The mmap() function asks to map length bytes starting at offset offset  from  the
       file  (or  other object) specified by the file descriptor fd into memory, prefer-
       ably at address start.  This latter address is a hint only, and is usually speci-
       fied  as  0.   The actual place where the object is mapped is returned by mmap(),
       and is never 0.

       The prot argument describes the desired memory protection (and must not  conflict
       with  the  open mode of the file). It is either PROT_NONE or is the bitwise OR of
       one or more of the other PROT_* flags.
       PROT_EXEC  Pages may be executed.
       PROT_READ  Pages may be read.
       PROT_WRITE Pages may be written.
       PROT_NONE  Pages may not be accessed.

       The flags parameter specifies the type of the mapped object, mapping options  and
       whether modifications made to the mapped copy of the page are private to the pro-
       cess or are to be shared with other references.  It has bits
[...]
```


Memory Mapped Files in C (2)

```
       MAP_FIXED  Do not select a different address than the one specified.  If the mem-
                  ory  region  specified by start and len overlaps pages of any existing
                  mapping(s), then the overlapped part of the existing  mapping(s)  will
                  be  discarded.   If  the specified address cannot be used, mmap() will
                  fail.  If MAP_FIXED is specified, start must  be  a  multiple  of  the
                  pagesize.  Use of this option is discouraged.

       MAP_SHARED Share  this  mapping  with  all  other processes that map this object.
                  Storing to the region is equivalent to writing to the file.  The  file
                  may not actually be updated until msync(2) or munmap(2) are called.

       MAP_PRIVATE
                  Create  a  private copy-on-write mapping.  Stores to the region do not
                  affect the original file.  It is unspecified whether changes  made  to
                  the file after the mmap() call are visible in the mapped region.

       You must specify exactly one of MAP_SHARED and MAP_PRIVATE.
[...]
       fd should be a valid file descriptor, unless MAP_ANONYMOUS is set, in which  case
       the argument is ignored.

       offset should be a multiple of the page size as returned by getpagesize(2).

       Memory mapped by mmap() is preserved across fork(2), with the same attributes.
[...]
RETURN VALUE
       On success, mmap() returns a pointer to the mapped area.   On  error,  the  value
       MAP_FAILED  (that  is,  (void *) -1) is returned, and errno is set appropriately.
       On success, munmap() returns 0, on failure -1, and errno is set (probably to EIN-
       VAL).
```


Memory Mapped Files in C (3)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <asm/fcntl.h>
#include <sys/stat.h>

int main(void) {
  char filename[]="testdatei.txt";
  struct stat buffer;
  stat( filename, &buffer );
  int len=buffer.st_size;

  printf("Laenge der Datei: %1d \n",len);
  int fd;
  if ((fd = open(filename, O_RDWR)) == -1)
     return ((int) (caddr_t) -1);

  char *result = (char *) mmap(0, len,
     PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);

  /* und jetzt den Anfang überschreiben */
  printf ("%s",result);
  strcpy(result,"Ein kleiner Versuch ---\n");

  (void) close(fd);
  return 0;
}
```

```
$ ls -l testdatei.txt
[...]   66  [...]  testdatei.txt

$ cat testdatei.txt
Das ist eine kleine Testdatei.
Sie enthaelt nur zwei Zeilen Text.

$ gcc -o mmap-beispiel \
  mmap-beispiel.c

$ ./mmap-beispiel
Laenge der Datei: 66
Das ist eine kleine Testdatei.
Sie enthaelt nur zwei Zeilen Text.

$ cat testdatei.txt
Ein kleiner Versuch ---
atei.
Sie enthaelt nur zwei Zeilen Text.
```

Memory Mapped Files in Python

```
#!/usr/bin/python
# mmap-beispiel.py
import mmap, os

filename = "testdatei.txt"
f = file(filename, "r+")
size = os.path.getsize(filename)
data = mmap.mmap(f.fileno(), size)

print data
print len(data), size

# Mit Slicing wie auf Strings zugreifen:
print repr(data[:15]), repr(data[:15])

# oder über Datei-Funktionen (read):
print repr(data.read(15)), \
      repr(data.read(15))

# Auch Schleifen gehen:
counter=0
for i in data:
  counter+=1
  print counter,":",i
  if counter==10: break
```

```
$ python mmap-beispiel.py
<mmap.mmap object at 0x403a33a0>
66 66

'Das ist eine kl' 'Das ist eine kl'
'Das ist eine kl' 'eine Testdatei.'

1 : D
2 : a
3 : s
4 :
5 : i
6 : s
7 : t
8 :
9 : e
10 : i
```