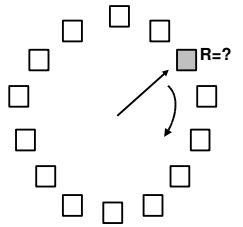
English Edition

Memory Management (5)

Second Chance Algorithm (2)

- Simpler implementation: „clock hand“
  - Put pages into a circular list, move the hand instead of repositioning a list element.



Check the page that the clock hand points to:

- if R=0, replace the page and move the hand.
- if R=1, clear R, move the hand and check the next page.

Second Chance Algorithm (1)

- Modified FIFO algorithm:
  When replacing a page: if the reference bit of the oldest page is set, then
  - clear the reference bit and put the page back to the end of the queue,
  - continue with the same check for the next oldest page.
- So the algorithm
  - replaces the oldest page with a cleared reference bit,
  - Gives a page that has recently been used a "second chance".

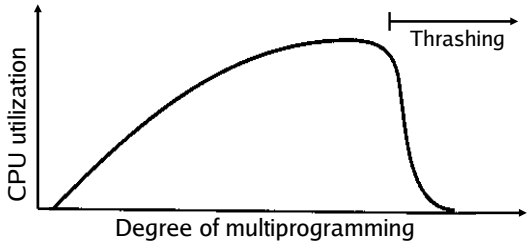
Page Replacement, Examples

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 means that a process causes excessively many Page Faults (every few thousand instructions).
- Thrashing occurs, when a process actively uses (i.e. Accesses) more pages than page frames were allocated for it.

Thrashing of several processes leads to low CPU utilization:



Thrashing (2)

Solutions

- If there is free memory available: Give the process further page frames (e.g. by dynamically adjusting the working set or with a global replacement strategy).
- If no free memory remains: Swapping of processes.

Further Design Issues (1)

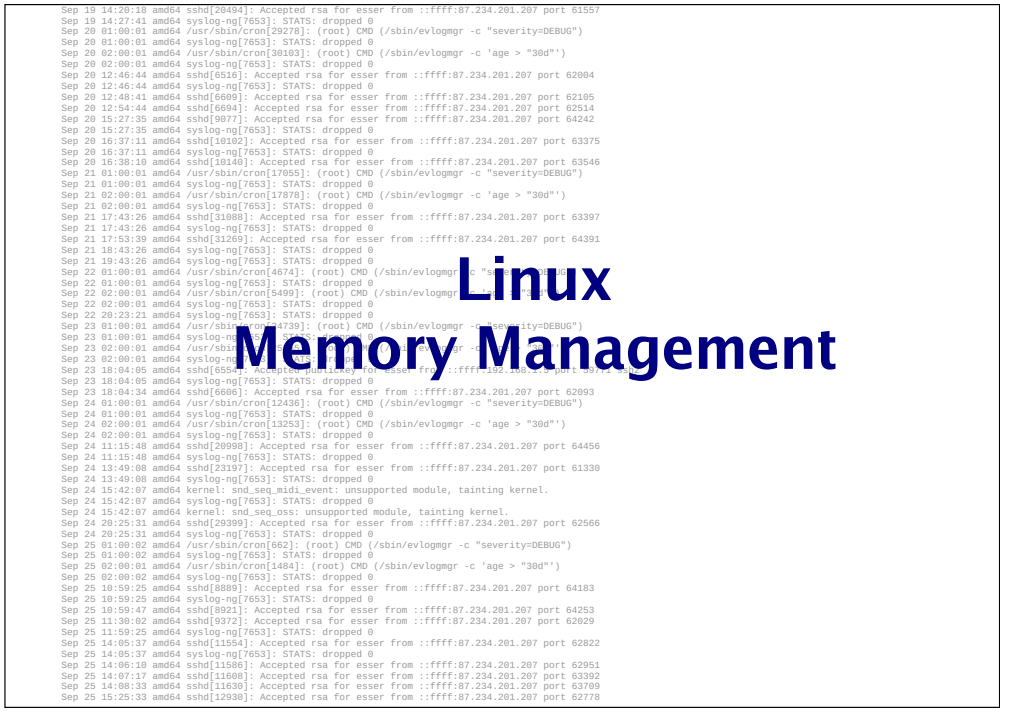
- Asynchronous Paging of modified pages:
  - Modified, replaced pages are paged with delay: E.g. wait until the disk containing the swap area is idle or when the stock of free page frames becomes small.
  - Write modifizied, non-replaced pages to disk in advance and clear their modify bit.
- Stock of free page frames:
  - Through early page replacement make sure that in case of a page request a free page frame will be available.

Further Design Issues (2)

- Page Buffering (Keeping the contents of a replaced page):
  - A replaced page first keeps its content, but is added to the list of free page frames or becomes reserved for swap-out.
  - If the process accesses the page again briefly after this, it is still in memory and can be reassigned to the process without disk access (soft page fault).
  - It must be known, whether the page frame has been reused meanwhile, i.e. has been assigned to a different process.

Further Design Issues (3)

- Prepaging:
  - A certain number of pages is brought into memory in advanced (e.g. when starting a new process) – even before the process tries to access it.
- Clustering:
  - When a Page Fault occurs, a cluster of several pages (instead of just the requested one) are brought into memory (special prepaging).
  - Modifed pages are not written to disk alone, but in clusters of several pages.



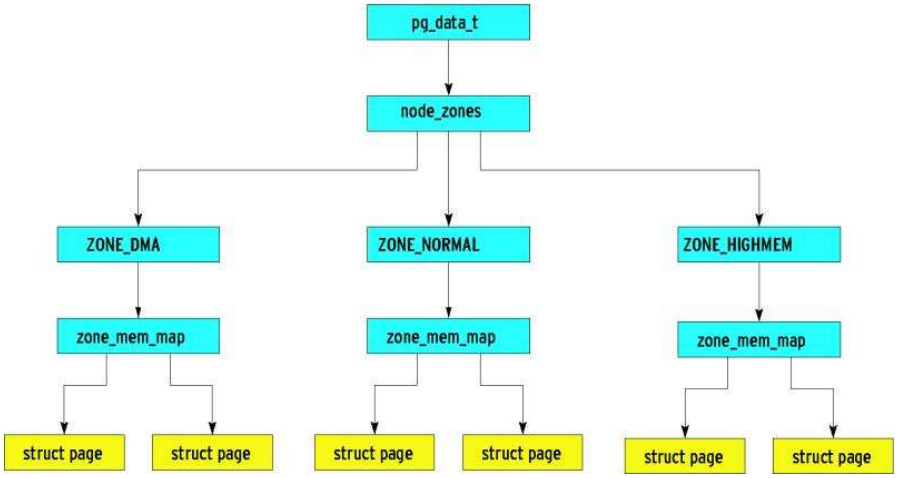
Further Design Issues (4)

- Locking pages in memory:
  - Some pages of a process are excempt from paging.
  - Implementation via a (privileged) System Call.
  - When swapping out the whole process, these pages are swapped as well.
- Design of the programs influences the number of page faults and thus the runtime of the program as well as the overall system performance!

Linux view on Memory (1)

- Memory management on several layers:
  1. Nodes, `pg_data_t`, Processor + its memory
  2. Zones:
     `ZONE_DMA`, `ZONE_NORMAL`, `ZONE_HIGHMEM`
  3. Each zone has its own table (`zone_mem_map`), listing all memory pages of this zone
  4. pages: `struct page`

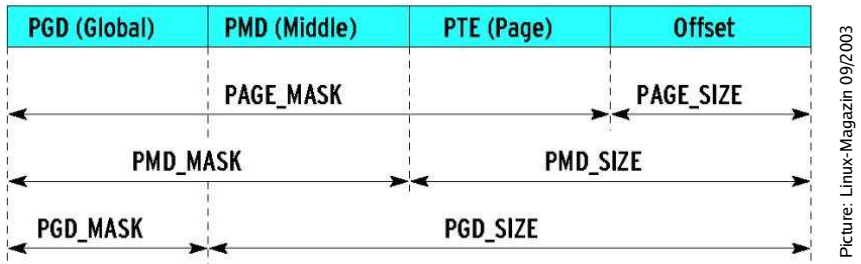
Linux view on Memory (2)



Picture: Linux-Magazin 09/2003

Linux view on Memory (3)

- Resolution of virtual addresses in three steps:
  - PGD: page global directory (Array of type `pgd_t`)
  - PMD: page middle directory (Array of type `pmd_t`)
  - PTE: page table entry (Array of type `pte_t`)



Picture: Linux-Magazin 09/2003

Linux view on Memory (4)

- **Managing the free memory**
  Linux uses the Buddy System; largest block size: 2^9 (= 512 pages = 2 MByte)
- **Returning memory**
  - Kernel 2.4:
    immediate merging of the buddies
  - Kernel 2.6:
    delayed – wait, whether one of the small blocks will be requested in the near future

Linux view on Memory (5)

- Process address space: `mm_struct` (in *linux/sched.h*)
- Each process has exactly one such structure (several threads of one process share the same)
- Address space divided into contiguous, non-overlapping **regions** (`vm_area_struct`)
- Start/End of regions aligned to page borders
- Information about regions is multiply saved:
  as list and as so-called Red Black Tree

Linux view on Memory (6)

Red Black Tree:
binary search tree
with following rules:

- Nodes are red or black
- Root is black
- All leaves are black
- Both children of each red node are black
- For each node holds:
  All paths to leaves contain the same number of black nodes.

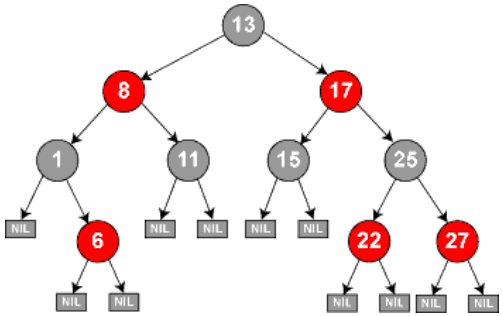


Bild: Wikipedia, http://en.wikipedia.org/wiki/Red-black_tree

-> mostly balanced tree; allows efficient search for addresses

Linux view on Memory (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`: linked list of regions (`mmap` is the list head)
- `mm_rb`: root of the Red Black Tree
- `mmap_cache`: Search is expensive, even with red black tree; here the last search result is cached
- `pgd`: Page Global Directory for this process

Linux view on Memory (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`: start and end of the code section
- `start_data/end_data`: start and end of the data section
- `start_brk/brk`: start/end of the heap
- `start_stack`: start of the stack
- `rss`: number of process pages which are currently in memory

Linux view on Memory (9)

Memory regions:
`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;
    [...]
```

Linux view on Memory (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;
};
```


Linux view on Memory (11)

System Calls which modify the address space

- Changing the size of a data region with
  - `brk(endds)` endds: highes virtual address of the data region (so called break value).
  - `oldendds = sbrk(increment)` increase the break value by `increment` bytes.
- The `fork` system call causes
  - Duplicating the process-private regions of the parent process,
  - Mapping the new process-private and the shared regions into the address space of the child process.

Linux view on Memory (12)

System Calls which modify the address space

- The `exec` system call causes
  - Undoing the mappings of the old regions (unmapping),
  - Allocation and loading of new regions,
  - Mapping the new regions into the process address space.
- The `exit` system call causes
  - Undoing all region mappings.

Practice: Memory Mapped Files

```
Sep 19 14:27:41 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 01:00:01 amd64 /usr/sbin/cron[29278]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 20 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 02:00:01 amd64 /usr/sbin/cron[30103]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 20 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 12:46:44 amd64 sshd[6516]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62004
Sep 20 12:46:44 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 12:48:41 amd64 sshd[6609]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62105
Sep 20 12:54:44 amd64 sshd[6694]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62514
Sep 20 15:27:35 amd64 sshd[9077]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64242
Sep 20 15:27:35 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 16:37:11 amd64 sshd[10102]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63375
Sep 20 16:37:11 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 16:38:10 amd64 sshd[10140]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63546
Sep 21 01:00:01 amd64 /usr/sbin/cron[17055]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 21 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 02:00:01 amd64 /usr/sbin/cron[17078]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 21 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 17:43:26 amd64 sshd[31088]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63397
Sep 21 17:43:26 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 17:53:39 amd64 sshd[31269]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64391
Sep 21 18:43:26 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 19:43:26 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 22 01:00:01 amd64 /usr/sbin/cron[4674]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 22 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 22 02:00:01 amd64 /usr/sbin/cron[5499]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 22 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 22 20:23:21 amd64 [illegible]
Sep 23 01:00:01 amd64 [illegible]
Sep 23 01:00:01 amd64 [illegible]
Sep 23 02:00:01 amd64 /usr/sbin/cron[25555]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 23 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 18:04:05 amd64 sshd[6554]: Accepted publickey for esser from ::ffff:192.168.1.5 port 59771 ssh2
Sep 23 18:04:05 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 18:04:34 amd64 sshd[6606]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62093
Sep 24 01:00:01 amd64 /usr/sbin/cron[12436]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 24 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 02:00:01 amd64 /usr/sbin/cron[13253]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 24 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 11:15:48 amd64 sshd[20998]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64456
Sep 24 11:15:48 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 13:49:08 amd64 sshd[23197]: Accepted rsa for esser from ::ffff:87.234.201.207 port 61330
Sep 24 13:49:08 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 15:42:07 amd64 kernel: snd_seq_midi_event: unsupported module, tainting kernel.
Sep 24 15:42:07 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 15:42:07 amd64 kernel: snd_seq_oss: unsupported module, tainting kernel.
Sep 24 20:25:31 amd64 sshd[29399]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62566
Sep 24 20:25:31 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 01:00:02 amd64 /usr/sbin/cron[662]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 25 01:00:02 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 02:00:01 amd64 /usr/sbin/cron[1484]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 25 02:00:02 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 10:59:25 amd64 sshd[8889]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64183
Sep 25 10:59:25 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 10:59:47 amd64 sshd[8921]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64253
Sep 25 11:30:02 amd64 sshd[9372]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62029
Sep 25 11:59:25 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 14:05:37 amd64 sshd[11554]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62822
Sep 25 14:05:37 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 14:06:10 amd64 sshd[11586]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62951
Sep 25 14:07:17 amd64 sshd[11608]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63392
Sep 25 14:08:33 amd64 sshd[11630]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63709
Sep 25 15:25:33 amd64 sshd[12930]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62778
```

Memory Mapped Files

- Idea: map contents of a file into memory
- then access the file through variables / memory addresses,
  e.g. mapping an ASCII file to
  - a string variable (Python)
  - a char* variable (C)
- read and manipulate file by modifying content of memory addresses
- standard POSIX function `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[]="testfile.txt";
  struct stat buffer;
  stat( filename, &buffer );
  int len=buffer.st_size;

  printf("Length of file: %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,"A little test ---\n");

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

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

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

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

$ ./mmap-example
Length of file: 66
Das ist eine kleine Testdatei.
Sie enthaelt nur zwei Zeilen Text.

$ cat testfile.txt
A little test ---
atei.
Sie enthaelt nur zwei Zeilen Text.
```

Memory Mapped Files in Python

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

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

print data
print len(data), size

# Access strings via slicing:
print repr(data[:15]), repr(data[:15])

# or via file functions (read):
print repr(data.read(15)), \
      repr(data.read(15))

# Loops are possible, too:
counter=0
for i in data:
  counter+=1
  print counter,":",i
  if counter==10: break
```

```
$ python mmap-example.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
```