Programmation Systèmes
Cours 9 — Memory Mapping

Stefano Zacchiroli
zack@pps.jussieu.fr

Laboratoire PPS, Université Paris Diderot - Paris 7

24 novembre 2011

URL http://upsilon.cc/zack/teaching/1112/progsyst/
Copyright © 2011 Stefano Zacchiroli
License Creative Commons Attribution-ShareAlike 3.0 Unported License
http://creativecommons.org/licenses/by-sa/3.0/

Outline

1. Memory mapping
2. File mapping
3. `mmap` memory management
4. Anonymous mapping

Outline

Virtual memory again

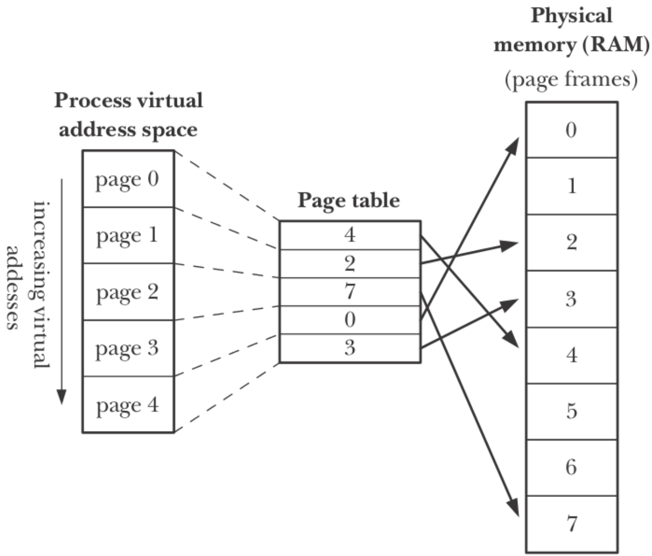
With virtual memory management (VMM) the OS adds an indirection layer between *virtual memory pages* and *physical memory frames*. The address space of a process is made of virtual memory pages, decoupling it from direct access to physical memory.

We have seen the main ingredients of VMM:

1. virtual pages, that form processes' virtual address space
2. physical frames
3. the page table maps pages of the resident set to frames

when a page $p \notin$ resident set is accessed, VMM swap it in from disk.

Virtual memory again (cont.)



TLPI, Figure 6-2

Backing store

Definition (datum, backing store)

In memory cache arrangements:

- a datum is an entry of the memory we want to access, passing through the cache
- the backing store is the (usually slower) memory where a datum can be retrieved from, when it cannot be found in the (usually faster) cache, i.e. when a cache miss happens

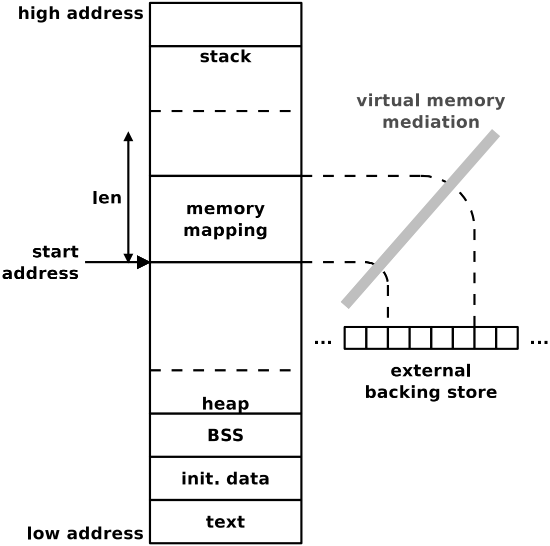
On UNIX, the *ultimate backing store* of virtual memory pages is usually the set of non-resident pages available on disk, that have been swapped out in the past.

Memory mapping

A UNIX memory mapping is a virtual memory area that has an extra backing store layer, which points to an external page store.

Processes can manipulate their memory mappings—request new mappings, resize or delete existing mappings, flush them to their backing store, etc.

Alternative intuition: a memory mapping is dynamically allocated memory with peculiar read/write rules.



Memory mapping types

Memory mappings can be of two different types, depending on the ultimate page backing store.

1. a file mapping maps a memory region to a region of a file
   - backing store = file
   - as long as the mapping is established, the content of the file can be read from or written to using direct memory access ("as if they were variables")
2. an anonymous mappings maps a memory region to a fresh "virtual" memory area filled with 0
   - backing store = zero-ed memory area

Having memory mapped pages in common

Thanks to virtual memory management, different processes can have mapped pages in common.

More precisely, mapped pages in different processes can refer to physical memory pages that have the same backing store.

That can happen in two ways:

1. through `fork`, as memory mappings are inherited by children
2. when multiple processes map the same region of a file

Shared vs private mappings

With mapped pages in common, the involved processes *might* see changes performed by others to mapped pages in common, depending on whether the mapping is:

private mapping
: in this case modifications are not visible to other processes.
  - pages are *initially* the same, but modification are not shared, as it happens with *copy-on-write* memory after fork
  - private mappings are also known as copy-on-write mappings

shared mapping
: in this case modifications to mapped pages in common are visible to all involved processes
  - i.e. pages are *not* copied-on-write

Memory mapping zoo

Summarizing:

- memory mappings can have files or “zero” as backing store
- memory mappings can be private or shared

A total of 4 different flavors of memory mappings:

visibility / backing store	file mapping	anon. mapping
private	private file mapping	private anon. mapping
shared	shared file mapping	shared anon. mapping

Each of them is useful for a range of different use cases.

mmap

The mmap syscall is used to request the creation of memory mappings in the address space of the calling process:

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

Returns: *starting address of the mapping if OK, MAP_FAILED on error*

The mapping specification is given by:

- length, that specifies the length of the desired mapping
- flags, that is a bit mask of flags that include

MAP_PRIVATE	request a private mapping
MAP_SHARED	request a shared mapping
MAP_ANONYMOUS	request an anonymous mapping

  - MAP_ANONYMOUS ⇔ anonymous mapping
    - fd must be -1 for anonymous mappings
  - one of MAP_PRIVATE, MAP_SHARED must be specified

mmap (cont.)

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

addr gives an address hint about where, in the process address space, the new mapping should be placed. It is just a hint and it is very seldomly used. To not provide one, pass NULL.

For file mappings, the mapped file region is given by:

- `fd`: file descriptor pointing to the desired backing file
- `offset`: absolute offset pointing to the *beginning* of the file region that should be mapped
  - the end is given implicitly by length
  - to map the entire file, use `offset == 0`

mmap (cont.)

```c
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

The desired memory protection for the requested mapping must be given via `prot`, which is a bitwise OR of:

PROT_READ	pages may be read
PROT_WRITE	pages may be write
PROT_EXEC	pages may be executed
PROT_NONE	pages may not be accessed at all

either PROT_NONE or a combination of the others must be given.

mmap (cont.)

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

The desired memory protection for the requested mapping must be given via prot, which is a bitwise OR of:

PROT_READ	pages may be read
PROT_WRITE	pages may be write
PROT_EXEC	pages may be executed
PROT_NONE	pages may not be accessed at all

either PROT_NONE or a combination of the others must be given.

What can PROT_NONE be used for?

mmap (cont.)

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

The desired memory protection for the requested mapping must be given via prot, which is a bitwise OR of:

PROT_READ	pages may be read
PROT_WRITE	pages may be write
PROT_EXEC	pages may be executed
PROT_NONE	pages may not be accessed at all

either PROT_NONE or a combination of the others must be given.

PROT_NONE use case: put memory fences around memory areas that we do not want to be trespassed inadvertently

mmap — example

```
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#include "apue.h"

int main(int argc, char **argv) {
        int fd;
        struct stat finfo;
        void *fmap;

        if (argc != 2)  err_quit("Usage: mmap-cat FILE");
        if ((fd = open(argv[1], O_RDONLY)) < 0)
                err_sys("open error");
        if (fstat(fd, &finfo) < 0)
                err_sys("fstat error");

        fmap = mmap(NULL, finfo.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
        if (fmap == MAP_FAILED)
                err_sys("mmap error");
        if (write(STDOUT_FILENO, fmap, finfo.st_size) != finfo.st_size)
                err_sys("write error");
        exit(EXIT_SUCCESS);
} /* end of mmap-cat.c */
```

mmap — example (cont.)

Demo

munmap

The converse action of mmap—unmapping—is performed by munmap:

#include <sys/mman.h>

int munmap(**void** *addr, size_t length);

Returns: *0 if OK, -1 otherwise*

The memory area between the addresses addr and addr+length will be unmapped as a result of munmap. Accessing it after a successful munmap will (very likely) result in a segmentation fault.

Usually, an entire mapping is unmapped, e.g.:

```
if ((addr = mmap(NULL, length, /* ... */)) < 0)
        err_sys("mmap error");

/* access memory mapped region via addr */

if (munmap(addr, length) < 0)
        err_sys("munmap error");
```

munmap on region (sub-)multiples

munmap does not force to unmap entire regions, one by one

1. unmapping a region that contains no mapped page will have no effect and return success
2. unmapping a region that spans several mappings will unmap all contained mappings
   - and ignore non mapped areas, as per previous point
3. unmapping only part of an existing mapping will
   - either reduce mapping size, if the unmapped part is close to one edge of the existing mapping;
   - or split it in two, if the unmapped part is in the middle of the existing mapping

Outline

File mappings

File mapping recipe

1. `fd = open(/* .. */);`
2. `addr = mmap(/* .. */, fd, /* .. */);`

Once the file mapping is established, access to the (mapped region of the) underlying file does not need to pass through `fd` anymore. However, the FD might still be useful for other actions that can still be performed only via FDs:

- changing file size
- file locking
- `fsync` / `fdatasync`
- ...

Thanks to file pervasiveness, we can use file mapping on device files

- e.g.: disk device files, /dev/mem, ... (not all devices support it)

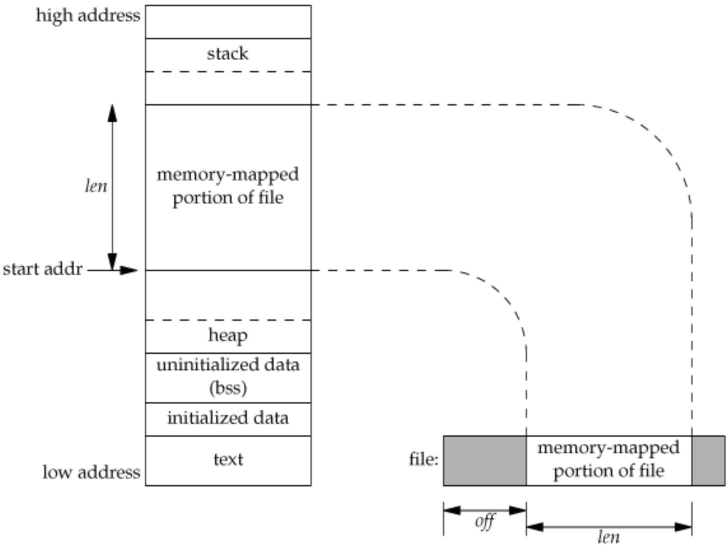
File mapping and memory layout

All kinds of memory mappings are placed in the large memory area in between the heap and stack segments.

The return value of `mmap` points to the start of the created memory mapping (i.e. its lowest address) and the mapping grows above it (i.e. towards higher addresses).

For file mappings, the mapped file region starts at `offset` and is `length` bytes long. The mapped region in memory will have the same size (modulo alignment issues...).

File mapping and memory layout (cont.)



APUE, Figure 14.31

File mapping and memory protection

The requested *memory protection* (`prot`, `flags`) must be compatible with the *file descriptor permissions* (O_RDONLY, etc.).

- FD must always be open for reading
- if PROT_WRITE and MAP_SHARED are given, the file must be open for writing

Private file mapping (#1)

Effects:

- the content of the mapping is initialized from file by reading the corresponding file region
- subsequent modifications are handled via copy-on-write
  - they are invisible to other processes
  - they are not saved to the backing file

Use cases

1. Initializing process segments from the corresponding sections of a binary executable
   - initialization of the text segment from program instructions
   - initialization of the data segment from binary data

   either way, we don't want runtime changes to be saved to disk

This use case is implemented in the dynamic loader/linker and rarely needed in other programs.

Private file mapping (#1)

Effects:

- the content of the mapping is initialized from file by reading the corresponding file region
- subsequent modifications are handled via copy-on-write
  - they are invisible to other processes
  - they are not saved to the backing file

Use cases

2. Simplifying program *input* logic
   - instead of a big loop at startup time to read input, one `mmap` call and you're done
   - runtime changes won't be saved back, though

Private file mapping — example (segment init.)

A real life example can be found in: glibc's dynamic loading code.

Demo

Private file mapping — example (input logic)

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include "apue.h"

int main(int argc, char **argv) {
        int fd;
        struct stat finfo;
        void *fmap;
        char *match;
        if (argc != 3)  err_quit("Usage: substring STRING FILE");

        if ((fd = open(argv[2], O_RDONLY)) < 0)  err_sys("open error");
        if (fstat(fd, &finfo) < 0)  err_sys("fstat error");
        /* "input" all file at once */
        fmap = mmap(NULL, finfo.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
        match = strstr((char *) fmap, argv[1]);

        printf("string%s found\n", match == NULL ? " NOT" : "");
        exit(match == NULL ? EXIT_FAILURE : EXIT_SUCCESS);
} /* end of grep-substring.c */
```

Demo

Notes:

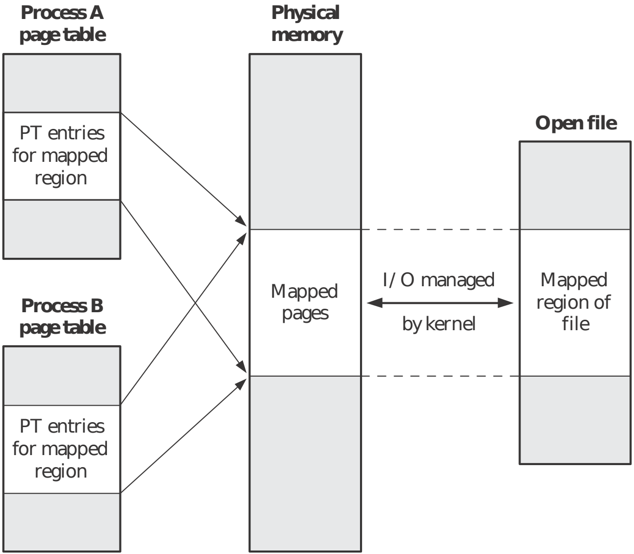
- example similar to `mmap-cat.c`, but here we put it into use
- thanks to the byte array abstraction we can easily look for a substring with `strstr` without risking that our target gets split in two different BUFSIZ chunks

Shared file mapping (#2)

Effects:

- processes mapping the same region of a file share physical memory frames
  - more precisely: they have virtual memory pages that map to the same physical memory frames
- additionally, the involved physical frames have the mapped file as ultimate backing store
  - i.e. modifications to the (shared) physical frames are saved to the mapped file on disk

Shared file mapping (#2) (cont.)



TLPI, Figure 49-2

Shared file mapping (#2) (cont.)

Use cases

1. memory-mapped I/O, as an alternative to `read`/`write`
   - as in the case of private file mapping, but here it works for both reading and writing data
2. interprocess communication, with the following characteristics:
   - data-transfer (not byte stream)
   - with filesystem persistence
   - among unrelated processes

Memory-mapped I/O

Given that:

1. memory content is initialized from file
2. changes to memory are reflected to file

we can perform I/O by simply changing bytes of memory.

Access to file mappings is less intuitive than sequential `read`/`write` operations

- the mental model is that of working on your data as a huge byte array (which is what memory is, after all)
- a best practice to follow is that of defining `struct`-s that correspond to elements stored in the mapping, and copy them around with `memcpy` & co

Memory-mapped I/O — example

We will redo the distributed scheme to assign global sequential unique identifiers to concurrent programs.

- we will use memory-mapped I/O
- we will do `fcntl`-based locking as before

Memory-mapped I/O — example (protocol)

```c
#include <fcntl.h>
#include <fcntl.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>
#include "apue.h"

#define DB_FILE "counter.data"
#define MAGIC    "42"
#define MAGIC_SIZ          sizeof(MAGIC)

struct glob_id {
        char magic[3];   /* magic string "42\0" */
        time_t ts;       /* last modification timestamp */
        long val;        /* global counter value */
};
```

Memory-mapped I/O — example (library)

```c
int glob_id_verify_magic(int fd, struct glob_id *id) {
        struct flock lock;
        int rc;

        lock.l_type = F_RDLCK;          /* read lock */
        lock.l_whence = SEEK_SET;       /* abs. position */
        lock.l_start = 0;               /* from begin... */
        lock.l_len = MAGIC_SIZ;         /* ...to magic's end */
        printf("  acquiring read lock...\n");
        if (fcntl(fd, F_SETLKW, &lock) < 0)
                err_sys("fcntl error");

        rc = strncmp(id->magic, MAGIC, 3);
        lock.l_type = F_UNLCK;
        printf("  releasing read lock...\n");
        if (fcntl(fd, F_SETLK, &lock) < 0)
                err_sys("fcntl error");

        return rc;
}
```

Memory-mapped I/O — example (library) (cont.)

```
void glob_id_write(struct glob_id *id, long val) {
        memcpy(id->magic, MAGIC, MAGIC_SIZ);
        id->ts = time(NULL);
        id->val = val;
}

/* end of mmap-uid-common.h */
```

Memory-mapped I/O — example (DB init/reset)

```c
#include "mmap-uid-common.h"

int main(void) {
	int fd;
	struct stat finfo;
	struct glob_id *id;
	struct flock lock;

	if ((fd = open(DB_FILE, O_RDWR | O_CREAT | O_TRUNC,
			S_IRUSR | S_IWUSR)) < 0)
		err_sys("open error");
	if (ftruncate(fd, sizeof(struct glob_id)) < 0)
		err_sys("ftruncate error");
	if (fstat(fd, &finfo) < 0)
		err_sys("fstat error");
```

Memory-mapped I/O — example (DB init/reset) (cont.)

```
	id = (struct glob_id *) mmap(NULL, finfo.st_size,
			PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

	lock.l_type = F_WRLCK;			/* write lock */
	lock.l_whence = SEEK_SET;		/* abs. position */
	lock.l_start = 0;			/* from begin... */
	lock.l_len = 0;				/* ...to EOF */
	printf("acquiring write lock...\n");
	if (fcntl(fd, F_SETLKW, &lock) < 0)
		err_sys("fcntl error");

	glob_id_write(id, (long) 0);
	exit(EXIT_SUCCESS);
}
/* end of mmap-uid-reset.c */
```

Memory-mapped I/O — example (client)

```
#include "mmap-uid-common.h"

int main(void) {
        int fd;
        struct stat finfo;
        struct glob_id *id;
        struct flock lock;

        if ((fd = open(DB_FILE, O_RDWR)) < 0)
                err_sys("open error");
        if (fstat(fd, &finfo) < 0)
                err_sys("fstat error");
        id = (struct glob_id *) mmap(NULL, finfo.st_size,
                        PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
        printf("checking magic number...\n");
        if (glob_id_verify_magic(fd, id) < 0) {
                printf("invalid magic number: abort.\n");
                exit(EXIT_FAILURE);
        }
```

Memory-mapped I/O — example (client) (cont.)

```
        lock.l_type = F_WRLCK;              /* write lock */
        lock.l_whence = SEEK_SET;           /* abs. position */
        lock.l_start = MAGIC_SIZ;           /* from magicno... */
        lock.l_len = 0;                     /* ...to EOF */
        printf("acquiring write lock...\n");
        if (fcntl(fd, F_SETLKW, &lock) < 0)
                err_sys("fcntl error");

        printf("got id: %ld\n", id->val);

        sleep(5);

        glob_id_write(id, id->val + 1);

        exit(EXIT_SUCCESS);
}
/* end of mmap-uid-get.c */
```

Demo

Notes:

- the `glob_id` structure now includes the magic string
  - TBH, there was no real reason for not having it before...
  - as a consequence, file sizes increases a bit, due to padding for memory alignment reasons
- we keep the FD around
  - to do file locking
  - resize the file upon creation
    - no, `lseek` is not enough to change file *size*
- the I/O logics is simpler

Demo

Notes:

- the `glob_id` structure now includes the magic string
  - TBH, there was no real reason for not having it before...
  - as a consequence, file sizes increases a bit, due to padding for memory alignment reasons
- we keep the FD around
  - to do file locking
  - resize the file upon creation
    - no, `lseek` is not enough to change file *size*
- the I/O logics is simpler

let's see the diffs...

Memory-mapped I/O — advantages

- performance gain: 1 memory copy
  - with `read`/`write` I/O each action involves 2 memory copies: 1 between user-space and kernel buffers + 1 between kernel buffers and the I/O device
  - with memory-mapped I/O only the 2nd copy remains
  - *flash exercise: how many copies for standard I/O?*

Memory-mapped I/O — advantages

- performance gain: 1 memory copy
  - with `read`/`write` I/O each action involves 2 memory copies: 1 between user-space and kernel buffers + 1 between kernel buffers and the I/O device
  - with memory-mapped I/O only the 2nd copy remains
  - *flash exercise: how many copies for standard I/O?*
- performance gain: no context switch
  - no syscall and no context switch is involved in accessing mapped memory
  - page faults are possible, though

Memory-mapped I/O — advantages

- performance gain: 1 memory copy
  - with `read`/`write` I/O each action involves 2 memory copies: 1 between user-space and kernel buffers + 1 between kernel buffers and the I/O device
  - with memory-mapped I/O only the 2nd copy remains
  - *flash exercise: how many copies for standard I/O?*
- performance gain: no context switch
  - no syscall and no context switch is involved in accessing mapped memory
  - page faults are possible, though
- reduced memory usage
  - we avoid user-space buffers → less memory needed
  - if memory mapped region is shared, we use only one set of buffers for all processes

Memory-mapped I/O — advantages

- performance gain: 1 memory copy
  - with `read`/`write` I/O each action involves 2 memory copies: 1 between user-space and kernel buffers + 1 between kernel buffers and the I/O device
  - with memory-mapped I/O only the 2nd copy remains
  - *flash exercise: how many copies for standard I/O?*
- performance gain: no context switch
  - no syscall and no context switch is involved in accessing mapped memory
  - page faults are possible, though
- reduced memory usage
  - we avoid user-space buffers → less memory needed
  - if memory mapped region is shared, we use only one set of buffers for all processes
- seeking is simplified
  - no need of explicit `lseek`, just pointer manipulation

Memory-mapped I/O — disadvantages

- memory garbage
  - the size of mapped regions is a multiple of system page size
  - mapping regions which are way smaller than that can result in a significant waste of memory

Memory-mapped I/O — disadvantages

- memory garbage
  - the size of mapped regions is a multiple of system page size
  - mapping regions which are way smaller than that can result in a significant waste of memory
- memory mapping must fit in the process address space
  - on 32 bits systems, a large number of mappings of various sizes might result in memory fragmentation
  - it then becomes harder to find continuous space to grant large memory mappings
  - the problem is substantially diminished on 64 bits systems

Memory-mapped I/O — disadvantages

- memory garbage
  - the size of mapped regions is a multiple of system page size
  - mapping regions which are way smaller than that can result in a significant waste of memory
- memory mapping must fit in the process address space
  - on 32 bits systems, a large number of mappings of various sizes might result in memory fragmentation
  - it then becomes harder to find continuous space to grant large memory mappings
  - the problem is substantially diminished on 64 bits systems
- there is kernel overhead in maintaining mappings
  - for small mappings, the overhead can dominate the advantages
  - memory mapped I/O is best used with large files and random access

Outline

Memory alignment

The notion of memory alignment, refers to the relation between memory addresses as used by programs and memory addresses as used by the hardware.

A variable that is located at a memory address which is a multiple of the variable size is said to be naturally aligned, e.g.:

- a 32 bit (4 bytes) variable is naturally aligned if its located at an address which is a multiple of 4
- a 64 bit (8 bytes) variable is naturally aligned if its located at an address which is a multiple of 8

All memory allocated "properly" (i.e. via the POSIX APIs through functions like `malloc`, `calloc`, `mmap`, `...`) is naturally aligned.

The risks of unaligned memory access depend on the hardware:

- it might result in traps, and hence in signals that kill the process
- it might work fine, but incur in performance penalties

Portable applications should avoid unaligned memory access.

posix_memalign

The notion of alignment is more general than natural alignment.
We might want to allocate memory aligned to chunks larger than the variable size.
POSIX.1d standardized the posix_memalign function to allocate heap memory aligned to arbitrary boundaries:

```c
#include <stdlib.h>

int posix_memalign(void **memptr, size_t alignment, size_t size);
```

Returns: *0 if OK; EINVAL or ENOMEM on error*

- we request an allocation of size bytes...
- ...aligned to a memory address that is a multiple of alignment
  - alignment must be a power of 2 and a multiple of sizeofvoid *

posix_memalign (cont.)

The notion of alignment is more general than natural alignment.
We might want to allocate memory aligned to chunks larger than the variable size.
POSIX.1d standardized the posix_memalign function to allocate heap memory aligned to arbitrary boundaries:

```
#include <stdlib.h>

int posix_memalign(void **memptr, size_t alignment, size_t size);
```

Returns: *0 if OK; EINVAL or ENOMEM on error*

- on success, memptr will be *filled* with a pointer to freshly allocated memory; otherwise the return value is either EINVAL (conditions on alignment not respected) or ENOMEM (not enough memory to satisfy request)
  - note: the above are *return values*, errno is *not* set
- allocated memory should be freed with free

posix_memalign — example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "apue.h"

int main(int argc, char **argv) {
	int rc;
	void *mem;

	if (argc < 3)  err_quit("Usage: memalign SIZE ALIGNMENT");

	if ((rc = posix_memalign(&mem,
				atoi(argv[2]), atoi(argv[1]))) != 0) {
		printf("posix_memalign error: %s\n", strerror(rc));
		exit(EXIT_FAILURE);
	}
	printf("address: %ld (%p)\n", (long) mem, mem);
	exit(EXIT_SUCCESS);
}
/* end of memalign.c */
```

`posix_memalign` — example (cont.)

Demo

Page alignment

As we have seen, the size of memory pages defines the granularity at which virtual memory operates.
A memory page is the smallest chunk of memory that can have distinct behavior from other chunks.

- swap-in / swap-out is defined at page granularity
- ditto for memory permission, backing store, etc.

`mmap` operates at page granularity

- each mapping is composed by a discrete number of pages
- new mappings are returned page aligned

Determining page size

There are various way to determine the page size.

A *non-portable* way of determining page size at compile-time is to rely on implementation-specific constants, such as Linux's PAGE_SIZE that is defined in `<asm/page.h>`

As *it might change between compilation and execution*, it is better to determine the page size at runtime. To determine that and many other limits at runtime, POSIX offers `sysconf`:

```
#include <unistd.h>

long sysconf(int name);
```

Returns: *the requested limit if name is valid; -1 otherwise*

The `sysconf` name to determine page size is `_SC_PAGESIZE`, which is measured in bytes.

Determining page size — example

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
        printf("page size: %ld btyes\n", sysconf(_SC_PAGESIZE));

        exit(EXIT_SUCCESS);
} /* end of pagesize.c */
```

On a Linux x86, 64 bits system:

```
$ ./pagesize
page size: 4096 btyes
$
```

mmap and page alignment

For maximum portability, `mmap`'s arguments addr and offset must be page aligned.[1]

In the most common case, the requirement is trivial to satisfy:

- `addr == NULL` — no address hint, do as you please
- `offset == 0` — map since the beginning of the file

[1] SUSv3 *mandate* page-alignment. SUSv4 states that implementations can *decide whether it's mandatory* or not. The net result is the same: for maximum portability, be page-aligned.

mmap and page alignment

For maximum portability, `mmap`'s arguments addr and `offset` must be page aligned.[1]

Exercise

Note that `offset` is the file offset. Contrary to addr it does not refer directly to memory. Why should it be page aligned then? Why can't we map unaligned file regions to aligned memory regions?

In the most common case, the requirement is trivial to satisfy:

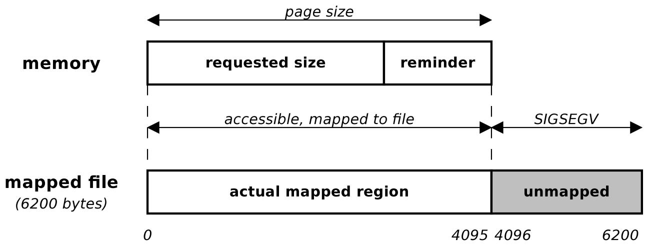
- `addr == NULL` — no address hint, do as you please
- `offset == 0` — map since the beginning of the file

[1] SUSv3 *mandate* page-alignment. SUSv4 states that implementations can *decide whether it's mandatory* or not. The net result is the same: for maximum portability, be page-aligned.

File mapping and page alignment

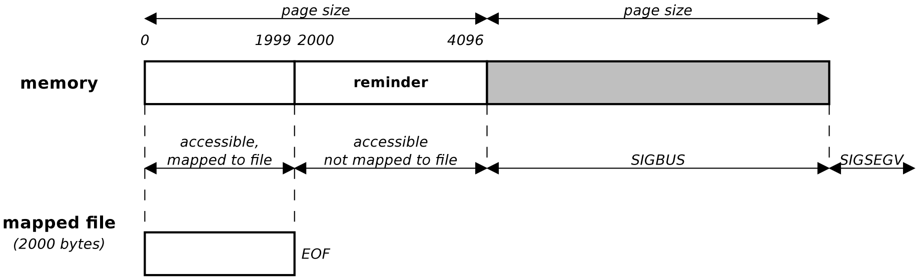
Interactions among page alignment and mapping or file sizes might be tricky. Two cases deserve special attention.

1. requested mapping size < page size
   - as mappings are made of entire pages, the size of the mapping is rounded up to the next multiple of page size
   - access beyond the *actual* mapping boundary will result in SIGSEGV, killing the process by default



File mapping and page alignment (cont.)

2. mapping extends beyond EOF
   - due to explicit request of mapping size rounding, a mapping might extend past end of file
   - the reminder of the page is accessible, initialized to 0, and shared with other processes (for MAP_SHARED)
   - changes past EOF will not be written back to file
     - ...until the file size changes by other means
   - *if* more entire pages are included in the mapping past EOF, accessing them will result in SIGBUS (as a warning)



Memory synchronization

As an IPC facility, we use file mapping for filesystem-persistent data transfer. To that end, we need to be concerned about:

- interprocess synchronization
  - when are page changes made visible to other processes?
- memory-file synchronization
  - when are modified pages written to file?
  - when are pages read from file?

these questions are particularly relevant for applications that mix memory-mapped with `read`/`write` I/O

Memory synchronization (cont.)

As an IPC facility, we use file mapping for filesystem-persistent data transfer. To that end, we need to be concerned about:

- interprocess synchronization
  - when are page changes made visible to other processes?

easy: given that processes have virtual frames that point to the same page, changes are immediately visible to all involved processes

- memory-file synchronization
  - when are modified pages written to file?
  - when are pages read from file?

  these questions are particularly relevant for applications that mix memory-mapped with read/write I/O

msync

The msync syscall can be used to control synchronization between memory pages and the underlying mapped file:

```
#include <sys/mman.h>

int msync(void *addr, size_t length, int flags);
```

Returns: *0 if OK, -1 otherwise*

- addr and length identify (part of) the mapping we want to sync
- flags is a bitwise OR of:

MS_SYNC	request synchronous file write
MS_ASYNC	request asynchronous file write
MS_INVALIDATE	invalidate cached coies of mapped data

addr must be page aligned; length will be rounded up

Memory → file synchronization

We can use `msync` to perform memory → file synchronization, i.e. to flush changes from the mapped memory to the underlying file.

- doing so ensures that applications `read`-ing the file will see changes performed on the memory mapped region

The degree of persistence guarantees offered by `msync` varies:

- with synchronous writes (MS_SYNC), `msync` will return only after the involved pages have been written to disk
  - i.e. the memory region is synced *with disk*
- with asynchronous writes (MS_ASYNC) `msync` ensures that subsequent `read`-s on the file will return fresh data, but only *schedules* disk writes without waiting for them to happen
  - i.e. the memory region is synced *with kernel buffer cache*

Intuition

`msync(..., MS_SYNC)` ≈ `msync(..., MS_ASYNC) + fdatasync(fd)`

File → memory synchronization

Regarding the initial loading, the behavior is implementation dependent.
The only (obvious) guarantee is that loading from file to memory will happen in between the mmap call and 1st memory access.

- portable applications should not rely on any specific load timing
- on Linux, page loading is lazy and will happen at first page access, at page-by-page granularity

Subsequent loading — e.g. a process `write` to a mapped file, when will the change be visible to processes mapping it? — can be controlled using `msync`'s `MS_INVALIDATE` flag.

- access to pages invalidated with `MS_INVALIDATE` will trigger page reload from the mapped file

Unified virtual memory system

Several UNIX implementations provide Unified Virtual Memory (UVM) (sub)systems. With UVM memory mappings and the kernel buffer cache share physical memory pages.

Therefore the implementation guarantees that the views of a file

1. as a memory-mapped region
2. and as a file accessed via I/O syscalls

are always coherent.

With UVM `MS_INVALIDATE` is useless and the only useful use of `msync` is to flush data to disk, to ensure filesystem persistence.

Whether a system offers UVM or not is an implementation-specific "detail".

- Linux implements UVM

Outline

Private anonymous mapping (#3)

Effects:

- each request of a new private anonymous mapping gives a fresh memory area that shares no pages with other mappings
- obtained memory is initialized to zero
- child processes will inherit private anonymous mappings, but copy-on-write will ensure that changes remain process-local
  - and that are minimized

Use cases

1. allocation of initialized memory, similar to `calloc`
   - in fact, `malloc` implementations often use `mmap` when allocating large memory chunks

Note: on several UNIX-es, anonymous mappings (both private and shared) can alternatively be obtained by file mapping `/dev/zero`.

mmap allocation — example (library)

```c
#include <errno.h>
#include <stdio.h>
#include <sys/mman.h>
#include <unistd.h>
#include "apue.h"

struct list {
        int val;
        struct list *next;
};

static struct list *list_bot;
static struct list *list_top;
static long list_siz;
```

mmap allocation — example (library) (cont.)

```
int list_init(long len) {
        list_top = (struct list *) mmap(NULL, len * sizeof(struct list),
                          PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS,
                          -1, 0);
        if (list_top == MAP_FAILED)
                return -1;
        list_bot = list_top;
        list_siz = len;
        printf("list_init: top=%p, len=%ld\n", list_top, len);
        return 0;
}

struct list *list_alloc() {
        long siz = (list_top - list_bot) / sizeof(struct list);
        if (siz >= list_siz) {
                errno = ENOMEM;
                return NULL;
        }
        list_top->next = NULL;
        printf("allocated %p\n", list_top);
        return list_top++;
}
```

mmap allocation — example (library) (cont.)

```
struct list *list_free() {
        /* left as an exercise */
        return NULL;
}

struct list *list_add(struct list *l, int val) {
        struct list *elt;
        if ((elt = list_alloc()) == NULL)
                return NULL;
        elt->val = val;
        elt->next = l;
        return elt;
}

void visit_list(const char *label, struct list *l) {
        printf("[%s] visit list: ", label);
        while (l != NULL) {
                printf("%d ", l->val);
                l = l->next;
        }
        printf("\n");
}

/* end of mmap-list.h */
```

mmap allocation — example

```c
#include "mmap-list.h"

int main(void) {
        struct list *l = NULL;
        pid_t pid;

        if (list_init(1000) < 0)  err_sys("list_init error");
        if ((l = list_add(l, 42)) == NULL
            || (l = list_add(l, 17)) == NULL
            || (l = list_add(l, 13)) == NULL)
                err_sys("list_add");
        visit_list("common", l);

        if ((pid = fork()) < 0)  err_sys("fork error");
        if (pid > 0) {  /* parent */
                l = list_add(l, 7);
                visit_list("parent", l);
        } else {        /* child */
                l = list_add(l, 5);
                visit_list("child", l);
        }

        exit(EXIT_SUCCESS);
} /* end of mmap-list.c */
```

Demo

Notes:

- proof of concept example of ad hoc memory management
  - ...the importance of being libc!
- virtual memory addresses are preserved through fork
- copy-on-write ensures that processes do not see changes made by others

Shared anonymous mapping (#4)

Effects:

- as with private anonymous mappings: each request gives a fresh area of initialized memory
- the main difference is that now pages are not copied-on-write
  - if the virtual pages become shared among multiple processes, the underlying physical memory frames become shared as well
  - note: the only way for this to happen is via `fork` inheritance

Use cases

1. Interprocess communication
   - data-transfer
   - among related processes (no longer unrelated)
   - with process persistence (no longer filesystem)

mmap-based IPC — example

```
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <unistd.h>
#include "apue.h"

int main(void) {
        int *addr;

        addr = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
                    MAP_SHARED | MAP_ANONYMOUS, -1, 0);
        if (addr == MAP_FAILED)
                err_sys("mmap error");
        *addr = 42;

        switch (fork()) {
        case -1:
                err_sys("fork error");
                break;
```

mmap-based IPC — example (cont.)

```
        case 0:             /* child */
                printf("child: %d\n", *addr);
                (*addr)++;
                break;
        default:            /* parent */
                if (wait(NULL) == -1)
                        err_sys("wait error");
                printf("parent: %d\n", *addr);
        }

        if (munmap(addr, sizeof(int)) == -1)
                err_sys("munmap error");
        exit(EXIT_SUCCESS);
}

/* End of mmap-ipc.c */
```

Demo

Notes:

- the mapping is shared through fork
- it is used as a data transfer facility from child to parent
- as with all shared memory solutions, synchronization is mandatory: in this case it is obtained via `wait`