

# Elements of Operating Systems

Chapter 8

## Where a Process Lives

J. Fornés    M. Farreras    Y. Becerra

Barcelona School of Informatics  
Universitat Politècnica de Catalunya

Bachelor's Degree in Informatics Engineering — Operating Systems

*Preview for readers · October 16, 2026*

© Universitat Politècnica de Catalunya

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International licence (CC BY-NC-SA 4.0):

«By Universitat Politècnica de Catalunya — BarcelonaTech (UPC), Any 2026»



<https://creativecommons.org/licenses/by-nc-sa/4.0/deed.en>

This text was written with the help of Claude, Anthropic's language model. Any errors that remain are the authors'.

# 8 Where a Process Lives

In Chapter 5 we said that `fork()` gives the child a copy of the parent's memory, and a footnote promised that no copy is really made at all. We also said that the PCB holds the description of the process's address space, without saying what that means. And in Chapter 4 we saw that C lets you add 1 to an address and write where you shouldn't, without explaining what stops that from taking the whole machine down with it.

All of this is the same subject: where a process's memory really is.

## 8.1 Two kinds of address

Let's start with the problem, which shows up better if you look at an executable. In Chapter 4 `gcc` made us a file; `objdump` tells us what is inside it\*:

```
alumni@labS0:~/S5$ objdump -d prova
0000000000401116 <lafuncio>:
 401116: 55                push    %rbp
 401117: 48 89 e5          mov     %rsp,%rbp
 40111a: 90                nop
 40111b: 5d                pop     %rbp
 40111c: c3                ret

000000000040111d <main>:
 40111d: 55                push    %rbp
 40111e: 48 89 e5          mov     %rsp,%rbp
```

---

\*Every session in this chapter comes from a lab room running openSUSE, which is most of them and the ones you will use for this subject. In the room running Ubuntu the addresses do not start the same way; the warning on page 8 says why.

```
401121: e8 f0 ff ff ff    call    401116 <lafuncio>
...
```

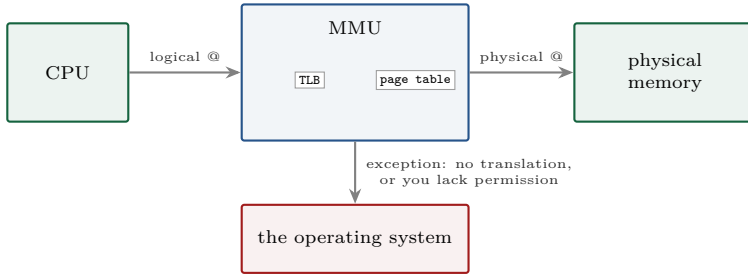
Look at the `call` line. The compiler has written the address 401116 inside the program: the one of the function it has to jump to. It had to settle on a number and put it there.

And here comes the awkward question. The program can be run three times at once, on a machine with fifty other processes on it and where the free memory is different every time. How can the compiler possibly know, months beforehand, which memory position will be free?

It doesn't.

The addresses the compiler writes into the program are **logical addresses**: a numbering system private to each process, which always starts the same way and says nothing about where the memory really is. The real positions in memory are the **physical addresses**, and the one that assigns them is the operating system, at the moment of setting the program going.

Between the one kind and the other there has to be somebody doing the translating, and it has to be fast enough to do it on every memory access — that is, several times per instruction. Software cannot do that: hardware does it, and the piece is called the **MMU**, the *memory management unit*.



The only address the program knows is the logical one. The MMU is what knows where it really is, and the operating system told it so when it allocated the memory.

**Figure 8.1.** Who asks, who answers, and what happens when there is no answer to give.

Of *how* the MMU does it we will say little, because you have seen it already: the address space is cut up into *pages* of a fixed size, the physical memory into *frames* of the same size, and a page table per process says which frame corresponds to each page, with which permissions and whether it is there at all. Processors add an associative memory to that, the TLB, so as not to have to look the table up on every access. What concerns us here is something else: what happens when the MMU **cannot** translate.

When the MMU cannot complete a translation — because the address is not the process’s, because the process has no permission for it, or because no frame has been assigned yet — it raises an exception and the operating system has to resolve it. Sometimes by killing the process, and sometimes by doing the work that was missing.

That sentence is the key to the whole chapter. Half of what the kernel does with memory consists of not doing it until the MMU complains.

## 8.2 The map of a process

A process's address space is not a uniform mass: it is shared out among **regions**, each one with a size, a starting point and a set of permissions.

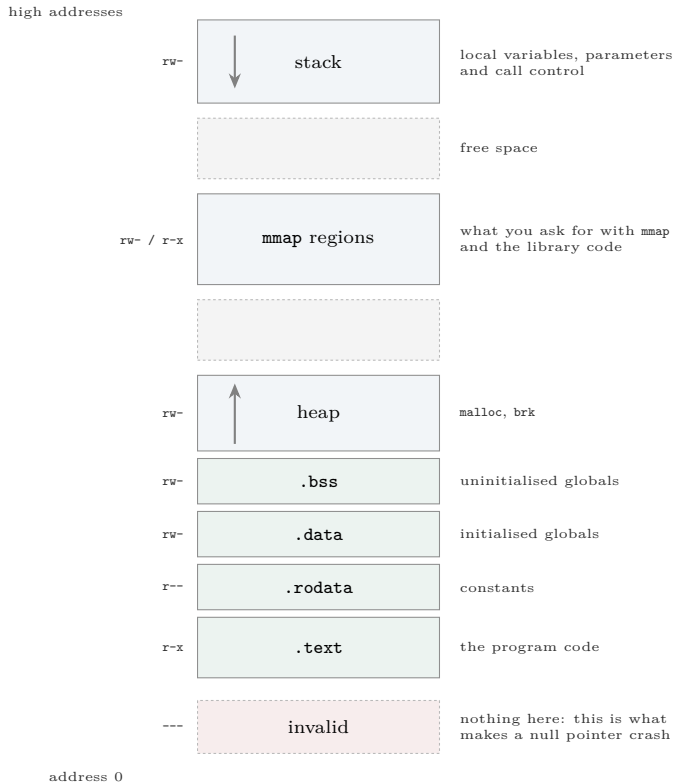


**Figure 8.2.** A process's address space. The regions at the bottom come from the executable; the ones at the top are made at run time.

This picture says four things.

**The regions at the bottom come from the file.** An executable is not a block: it is a format — on Unix, ELF — with **sections** declared in the header. `.text` carries the code; `.data`, the global

variables with an initial value; `.rodata`, the constants. `.bss` is the odd one: it holds the globals that are *not* initialised, and since they are all zero, only their size is kept in the file. An `objdump -h` will show you all of them.

**The ones at the top are not in the file.** The stack and the heap are made when the program runs and change size while it lasts. They grow towards each other: the stack towards low addresses, every time you call a function; the heap towards high ones, every time you ask for memory.

**Every region has permissions.** The code is readable and executable but not writable, and that is why you cannot rewrite your own program halfway through running it. The constants are read-only, and that is why writing over a string literal blows up. The MMU enforces them page by page, and it is the same MMU that does the translating.

**And address 0 belongs to nobody.** The first page is deliberately left unassigned, so that any access through a null pointer raises the exception and the process dies with a `SIGSEGV*` instead of trampling on the code.

### 8.2.1 Having a look

None of this is theory: the kernel will show you the map if you ask it for it. This program writes the address of half a dozen things of its own and then stays still, so that you can compare it with the real map:

**Listing 8.1.** Where each thing lives

```
1 int global_initialised = 5;           /* .data section */
2 int global_empty;                    /* .bss section */
3 const char constant[] = "text";      /* .rodata section */
4
5 static void where(const char *what, const void *address)
6 {
```

---

\*From Chapter 7. If you do not catch the signal, the default action is to kill the process and leave a `core` file behind.

```

7     char buf[96];
8
9     sprintf(buf, "%-24s %p\n", what, address);
10    write(1, buf, strlen(buf));
11 }
12
13 int main(void)
14 {
15     int local = 1;                                ❶
16     int *dynamic = malloc(sizeof(int));            ❷
17     char buf[80];
18
19     if (dynamic == NULL)
20         error_and_exit("malloc", 1);
21
22     where("code (main)", (void *)main);            ❸
23     where("constant (.rodata)", constant);
24     where("initialised (.data)", &global_initialised);
25     where("zeroed (.bss)", &global_empty);
26     where("heap (malloc)", dynamic);
27     where("end of heap", sbrk(0));                 ❹
28     where("stack (local)", &local);
29
30     sprintf(buf, "\nI am %d: look at /proc/%d/maps\n",
31               (int)getpid(), (int)getpid());
32     write(1, buf, strlen(buf));
33     sleep(30);
34
35     free(dynamic);
36     return 0;
37 }

```

- ❶ A local variable: it lives on the stack.
- ❷ `malloc` returns an address in the heap. We talk about that in the next section.
- ❸ A function's name is the address of its first instruction: the same one `objdump` showed us.
- ❹ `sbrk(0)` with a zero asks for nothing: it only says where the heap ends right now.

```
alumne@labS0:~/S5$ ./mem-map &
code (main)          0x401217
constant (.rodata)   0x402008
initialised (.data)  0x404060
zeroed (.bss)        0x404068
heap (malloc)        0x39bfa010
end of heap          0x39c1b000
stack (local)        0x7ffec08e7b74

I am 8385: look at /proc/8385/maps
alumne@labS0:~/S5$ cat /proc/8385/maps
00400000-00401000 r--p ... /home/alumne/S5/mem-map
00401000-00402000 r-xp ... /home/alumne/S5/mem-map
00402000-00403000 r--p ... /home/alumne/S5/mem-map
00403000-00404000 r--p ... /home/alumne/S5/mem-map
00404000-00405000 rw-p ... /home/alumne/S5/mem-map
39bfa000-39c1b000 rw-p ... [heap]
7f5f4e400000-7f5f4e428000 r--p ... /usr/lib64/libc.so.6
7f5f4e428000-7f5f4e59e000 r-xp ... /usr/lib64/libc.so.6
...
7ffec08c8000-7ffec08ea000 rw-p ... [stack]
```

Now pair them up. `main` is at `0x401217`, which falls inside the executable's `r-xp` line: read and execute, no write, exactly as it should be. The constant falls in the `r--p` line below it. The two globals both fall in the program's one and only `rw-p` line. The `malloc` falls inside `[heap]`, and the end of the heap matches the end of that line exactly. And the local variable is in `[stack]`, more than a hundred terabytes away from all the rest.

#### Question 1

There are five lines of the executable in the map and only four kinds of thing in the program. And in the middle up comes `libc`, which we never asked for anywhere. Where did it spring from?

 **Answer**

From the shared libraries, which we will see further on, among the four tricks. Your executable does not carry `libc`'s code inside it: it carries a note saying that it needs it, and it is on starting up that somebody maps it into the process's address space. That "somebody" is a program too, and you will see it in the map as well: `ld-linux.so`.

## 8.3 What happens inside an `exec()`

With that we can now finish what Chapter 5 left half done. When a process does an `execvp()`, the kernel:

1. **Interprets the file:** it reads the ELF header and sees which sections it has, what size they are and where they are inside the file.
2. **Defines the address space:** it decides which logical addresses each region takes up and with what permissions, and notes it down in the PCB.
3. **Assigns physical memory** and sets the MMU up so that it knows how to translate.
4. **Brings the contents in:** it reads the sections from the disk and writes them into the memory it assigned.
5. **Sets the program counter** to the first instruction.

And from here the process carries on, with another program inside it. Of step 4 we will say further on that it is a lie: not all of it is read, and hardly ever.

 **Careful**

The starting addresses of the regions are not always the same. If you compare two runs of `mem-map` you will see some numbers moving about: the system applies a random offset on purpose,

so that an attacker who wants to jump to a particular address cannot guess it. In the rooms running openSUSE the code and the globals do not move—the compiler there makes executables that always have to be loaded at 0x400000—, but the heap, the libraries and the stack do. In the room running Ubuntu, the one used for SO2 and CASO, the compiler makes *PIE* executables, which can be loaded anywhere: there the numbers start with 0x55 and everything moves, the code included. So here, and for the rest of the chapter, the only thing you can take for granted about an address is which region it falls in.

## 8.4 Dynamic memory

So far, all the memory we have used was settled before running: the globals are sized by the compiler and the locals by the call to the function. But there are things that are not known until the program runs. If the program is given a number on the command line and has to make that many children, how many `ints` does it need?

Fixing that at compile time has two answers and both of them are bad: either you reserve too many and squander them, or you reserve too few and it blows up. The good one is to ask for them when you know.

### 8.4.1 `brk()` and `sbrk()`: moving the limit

The classic way is to move the heap's upper limit.

System call `sbrk`

```
void *sbrk(intptr_t bytes);
```

`sbrk()` moves the heap's limit by as many bytes as you tell it and returns *where it was before*, which is precisely the address of the piece

you have just gained\*. With a negative value it makes the limit go back; with a zero it does not move it and only tells you where it is.

**Listing 8.2.** The heap's limit, by hand

```

1 static void limit(const char *when)
2 {
3     char buf[96];
4
5     sprintf(buf, "%-26s %p\n", when, sbrk(0));      ❶
6     write(1, buf, strlen(buf));
7 }
8
9 int main(void)
10 {
11     char buf[96];
12     void *mine;
13
14     limit("at the start");
15
16     mine = sbrk(4096);                               ❷
17     if (mine == (void *)-1)
18         error_and_exit("sbrk", 1);
19     sprintf(buf, "%-26s %p\n", "sbrk(4096) returned", mine);
20     write(1, buf, strlen(buf));
21     limit("after the sbrk");
22
23     if (sbrk(-4096) == (void *)-1)                   ❸
24         error_and_exit("sbrk", 1);
25     limit("after giving them back");
26
27     free(malloc(1));                                  ❹
28     limit("after malloc(1)");
29     return 0;
30 }
```

❶ `sbrk()` with a zero is the way to ask without requesting anything.

\*`sbrk` is this book's exception: it is not in POSIX—it was in it as `LEGACY` and ended up being taken out—and on Linux it is not a call either. The call is `brk()`, which wants the limit's new address, whole; `sbrk` is the convenience of asking in bytes and being told where it was. It is in the book because it is what you will find written everywhere, and because it explains itself.

- ② We ask for a page. What it returns is the address of the beginning.
- ③ And we give it back.
- ④ This `malloc` of a single byte is the most interesting thing in the program: look at where it leaves the limit.

```
alumni@labS0:~/S5$ ./mem-sbrk
at the start          0x326e7000
sbrk(4096) returned   0x326e7000
after the sbrk        0x326e8000
after giving them back 0x326e7000
after malloc(1)       0x32708000
```

A one-byte `malloc` has moved the limit by 132 kB. That is no mistake: the library asks for a lot of them at once so that it does not have to bother the kernel again next time.

#### Careful

This program mixes `sbrk()` and `malloc` on purpose, to teach that both of them touch the same limit. In a real program, never do it: the library takes it for granted that the heap is its own, and moving it from under it is an excellent way to get errors nobody can make sense of.

`sbrk()` is honest and uncomfortable. It gives you raw bytes, it has no idea what you keep in them, and all the bookkeeping — what is free, what can be given back, when — is left on your side. That is why hardly anybody uses it directly.

## 8.4.2 malloc: the exception

Library function    `malloc` and `free`

```
void *malloc(size_t bytes);  
void free(void *region);
```

`malloc` and `free` are **not system calls**: they are C library functions, and that is why in this book they do not come out in red. What they do is the bookkeeping `sbrk()` does not do.

When you ask it for memory, `malloc` looks to see whether there is a free piece of the heap big enough already; if there is, it marks it as taken and returns you its address, with no system call in between. If there isn't, it has to ask the kernel for some, and here it does something you have to know about: **it chooses between two system calls according to the size**. For ordinary requests it moves the heap's limit with `brk()` — and asks for more than it needs, so as not to have to bother the kernel again straight away. For the big ones, from a threshold the system fixes\* upwards, it does not even touch the heap: it asks for a region of its own with `mmap()`. And `free` does not always do the same thing either: what comes from the heap is marked as free in the library's structures and is not given back to the system — it will be put to use again —, whereas what came from an `mmap()` really is given back, with `munmap()`.

The heap you see in `/proc/PID/maps` is not the one your program is using: it is the one the library has asked for. Between the two things is everything `malloc` has kept back in reserve.

## 8.4.3 Why malloc and not printf

You should have this question on the tip of your tongue. All through the book we have written with `write()` and never with `printf`, on

\*In `glibc` the threshold is configurable and, by default, is of the order of 128 kB. It can be read and changed with `mallopt(M_MMAP_THRESHOLD, ...)`.

the grounds that `printf` is a library function and hides the system from us; and now it turns out we are using `malloc`, which is one as well.

This is no inconsistency: it is the same rule, applied twice.

**`printf` replaces** the system call we want to show: it puts a buffer in the middle that decides on its own account when the writing really happens, and that buffer hides precisely what we are studying — the order things happen in when there is more than one process writing.

**`malloc` replaces nothing: it is the client** of the calls we want to show. Underneath it does a `brk()` or an `mmap()` — one or the other according to the size you ask it for —, and what it adds, which is grouping requests together and remembering what is free, is work the kernel does not do and that somebody has to do anyway.

The rule, then: **we avoid what hides the mechanism from us and we accept what leans on it.**

`sprintf` is in the second group, and that is why we use it without any qualms: it formats text into an array of `char` and stops there. The `write()` that comes after it is still ours, and it is where the system shows.

#### 8.4.4 `mmap()`: the modern way

System call `mmap` and `munmap`

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

`mmap()` moves no limit: it creates a new region, wherever the system likes, of the size you say and with the permissions you say. It has six

parameters and it looks frightening, but three of them are zero nearly every time.

**addr** where you want it. With `NULL` the system decides, which is what you want 99% of the time.

**bytes** how much.

**prot** the permissions: `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`, or `PROT_NONE`. They are the same ones you will read afterwards in the `maps`.

**flags** what it is filled with and who sees it. `MAP_ANONYMOUS` means “blank memory”; without it, the region is filled with the contents of a file. And `MAP_PRIVATE` or `MAP_SHARED` decide whether anybody else sees the changes.

**fd and offset** which file and from where. With `MAP_ANONYMOUS` they are ignored and you put `-1` and `0` in them.

To ask for memory and nothing more, then, the formula is always this one:

### Listing 8.3. Memory outside the heap

```

1  /* The BSDs, and therefore macOS, call it MAP_ANON. The
2     reference platform of this book is Linux (see
3     codi/PORTABILITAT.md), but this line is free. */
4  #ifndef MAP_ANONYMOUS
5  #define MAP_ANONYMOUS MAP_ANON
6  #endif
7
8  #define HOW_MANY 4096
9
10 int main(void)
11 {
12     char buf[96];
13     int *region;
14
15     region = mmap(NULL, HOW_MANY,                ❶
16                  PROT_READ | PROT_WRITE,          ❷
17                  MAP_ANONYMOUS | MAP_PRIVATE,     ❸
18                  -1, 0);

```

```

19     if (region == MAP_FAILED)                                ❹
20         error_and_exit("mmap", 1);
21
22     region[0] = 42;
23     sprintf(buf, "region at %p, I stored a %d in it\n",
24             (void *)region, region[0]);
25     write(1, buf, strlen(buf));
26     sprintf(buf, "the heap still ends at %p\n", sbrk(0));
27     write(1, buf, strlen(buf));
28
29     if (munmap(region, HOW_MANY) < 0)                        ❺
30         error_and_exit("munmap", 1);
31     return 0;
32 }

```

- ❶ NULL: you decide.
- ❷ Read and write. If you asked for `PROT_READ` alone, the line that comes next would kill your process.
- ❸ Blank memory, and for me alone.
- ❹ `mmap()` does not return NULL when it fails, but `MAP_FAILED`. It is one of the few calls that do not follow the `-1` convention.
- ❺ And it is given back when it is no longer needed. Unlike `free`, this really does give it back to the system.

```

alumne@labS0:~/S5$ ./mem-mmap
region at 0x7f3fe4d7d000, I stored a 42 in it
the heap still ends at 0x39e17000

```

Compare the two numbers with the picture in Figure 8.2: the new region is not in the heap, it is much further up, between the heap and the stack, next to the libraries.

Why do we prefer this one and not `brk()`? Because `brk()` only knows how to do one thing — move a limit — and `mmap()` knows how to do four. It lets you choose the permissions region by region; it lets you free something in the middle without having to wait for everything above it to be freed; it lets you fill the region with a file, which we will see in Chapter 9; and it lets you share it, which is the next section. `brk()` was not a bad idea in 1975; it has simply been

outgrown.

## 8.5 Sharing memory

By default, a process cannot touch another one's memory. It is the most basic protection there is, and it is the MMU that enforces it: if an address is not in your page table, it does not exist for you.

But sometimes sharing it is exactly what you want, and then you have to ask for it. The flag is what says so:

**Listing 8.4.** One flag of difference

```

1  /* The BSDs, and therefore macOS, call it MAP_ANON. The
2     reference platform of this book is Linux (see
3     codi/PORTABILITAT.md), but this line is free. */
4  #ifndef MAP_ANONYMOUS
5  #define MAP_ANONYMOUS MAP_ANON
6  #endif
7
8  int main(int argc, char *argv[])
9  {
10     char buf[96];
11     int *value, flag;
12     pid_t ret;
13
14     if (argc != 2)
15         Usage(argv[0]);
16     if (strcmp(argv[1], "shared") == 0)
17         flag = MAP_SHARED;
18     else if (strcmp(argv[1], "private") == 0)
19         flag = MAP_PRIVATE;
20     else
21         Usage(argv[0]);
22
23     value = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
24                 MAP_ANONYMOUS | flag, -1, 0);
25     if (value == MAP_FAILED)
26         error_and_exit("mmap", 1);
27
28     *value = 100;
29 
```

```

30     if ((ret = fork()) < 0)
31         error_and_exit("fork", 1);
32     if (ret == 0) {
33         *value = 42; ❸
34         exit(0);
35     }
36     waitpid(ret, NULL, 0);
37
38     sprintf(buf, "%s: the parent reads %d\n", argv[1], *value);
39     write(1, buf, strlen(buf)); ❹
40     munmap(value, sizeof(int));
41     return 0;
42 }

```

- ❶ The only line that changes between the two cases.
- ❷ The parent writes a 100 into it before it has any child.
- ❸ The child writes a 42 into it.
- ❹ And the parent looks at what is in there.

```

alumne@labS0:~/S5$ ./mem-shared private
private: the parent reads 100
alumne@labS0:~/S5$ ./mem-shared shared
shared: the parent reads 42

```

With `MAP_PRIVATE`, the region behaves like the rest of the process's memory: `fork()` makes a copy of it for the child, and each one writes into its own. With `MAP_SHARED`, the two page tables point to the same frame, and what one of them writes into it the other sees immediately.

This is communication between processes already, and it is the fastest there is: there is no copy, no system call, no kernel in between. Writing to shared memory is writing to memory. The price is that there is nothing to synchronise anything. If both of them write into it at once, the result is whatever it turns out to be. With the pipes of Chapter 10 that does not happen, because the kernel is there; here it is not, and the risk is yours.

## 8.6 The four tricks

Everything we have explained so far would work already. What comes now is not needed at all, technically: they are four optimisations. It is just that without them the machine in front of you would not last ten minutes, and all four come out of the same idea, the one in the first section: *do nothing until the MMU complains*.

### 8.6.1 Demand loading

We said that `exec()` reads the sections from the disk and writes them into the memory it assigned. It does not.

What it does is define the regions and leave them with no frames assigned: the MMU has no translation for them. The first time the program touches a page, the exception goes off; the kernel looks in the PCB, sees that the address is valid but is not loaded, assigns it a frame, brings in the right piece of the file, updates the MMU and **resumes the instruction that had failed**, which this time works. The program hears nothing about any of it.

And the gain is double: starting up is much faster, because nobody waits for a whole executable to be read, and of a two-megabyte program of which only one function is ever run, only that function is loaded.

### 8.6.2 Shared libraries

Think how many programs on your machine use `libc`. If each one carried a copy of it inside its executable, it would be a disk full of identical copies, and a memory full of identical copies.

That is why, by default, the libraries' code is not part of the executable: there is only a note saying which of its functions it needs. On starting up, the kernel launches another program — the dynamic linker, which comes out in the map as `ld-linux.so` — which maps the libraries into the process's address space with `mmap()`, and since that code is read-only, every process that uses it shares the same frames.

One copy on the disk, one copy in memory, and updating it forces nobody to recompile anything. With `gcc -static` you would get the

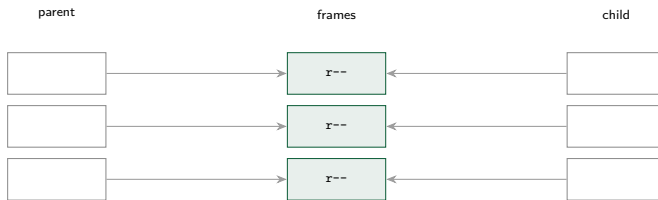
old behaviour; do it once and compare the size of the two executables.

### 8.6.3 Copy on write

Now we can pay off the debt from Chapter 5. We said that the child of a `fork()` gets a copy of the parent's memory. It is true from inside the program, and false from inside the kernel.

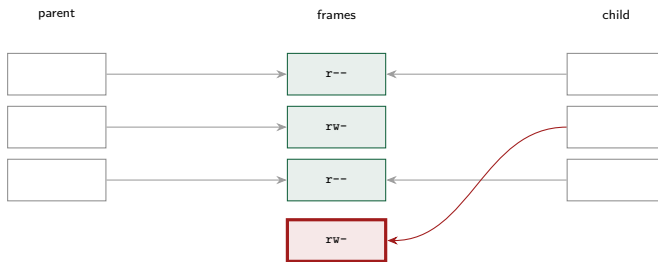
Copying a process's whole address space would be enormously expensive, and it would be squandered nearly every time: in Chapter 5, half our examples did a `fork()` and an `execvp()` straight after it, which throws out everything that had just been copied.

#### Just after `fork()`



No copy at all. Both page tables point to the same frames, and the kernel has marked them all *read-only*, in the parent too.

#### The child writes to the second page



The MMU will not let it write and tells the kernel, which copies *that* page and no more, restores the permissions and resumes the instruction. The other two stay shared.

**Figure 8.3.** `fork()` copies nothing. The copying is done page by page, and only of the ones somebody touches.

The mechanism is the same one as always. Instead of copying, the kernel makes the two page tables point to the same frames and marks them both **read-only** — the parent's too, which until now could write into them. While both of them only read, nothing happens and there is a single copy. When one of the two writes, the MMU will not let it and warns the kernel; the kernel looks at its structures, sees that this is a copy-on-write page, copies *that* page, gives the write permissions back — to both processes, and only for that page — and resumes the instruction.

Linux applies it to the dynamic memory you have asked for and not yet touched as well: all the new pages point to one and the same read-only frame full of zeros, until you write into it.

### Question 2

A process takes up seven pages: three of code, two of data, one of stack and one of heap. It does a `fork()` and, straight afterwards, the child does an `execvp()` of a program that takes up ten. How many pages of physical memory end up being spent over and above what is needed, with copy-on-write and without it?

### Answer

Without copy-on-write, seven are copied at the moment of the `fork()`, and straight afterwards the `execvp()` throws them all out and asks for ten: seven pages' worth of work and of memory completely lost.

With copy-on-write, not one has been copied between the `fork()` and the `execvp()`. The child ends up with its ten and the parent keeps its seven, which is where we had to get to. This case — a `fork()` and an `execvp()` one after the other — is the commonest of them all: it is what the shell does every time you type a command into it.

### 8.6.4 Virtual memory

The last trick goes the other way: it is not about bringing things into memory when they are needed, but about **taking out** the ones that are not.

If the physical memory runs out, the kernel picks a page it reckons will not be used soon, saves it in an area of disk — the *swap* area — and hands the frame over to whoever needs it. If somebody touches that page again, the MMU cannot translate it, and you know how it goes on already: exception, the kernel looks for a free frame, brings the contents back into it from the disk, updates the MMU and resumes the instruction. That is a **page fault**.

The consequence is a big one: the sum of all the processes' address spaces can be bigger than the machine's memory, and even a single process can have a space bigger than it. Memory stops being a hard limit and becomes a question of speed.

#### Careful

Of speed, mind you, and a great deal of it. An access to a resident page is nanoseconds; one to a page that is on the disk is six orders of magnitude more. If the system overdoes it and cannot keep in memory even the minimum each process needs in order to make progress, it goes into *thrashing*: it spends the day swapping pages about and runs nothing. The machine has not hung, but it might as well have. The way out is to take processes off it, and it is the system itself that has to detect this by looking at the page fault rate.

So that this happens rarely, the kernel picks its victims well — approximating the algorithm that throws out the page nobody has touched for longest — and does *prefetch*: if it sees you reading pages one after another, it brings a few of them in ahead of time.

## 8.7 What there is, then, on Linux

Gathering it all up, and keeping what has to be remembered:

- Virtual memory based on **paging**<sup>\*</sup>, with a replacement algorithm that approximates “least recently used” and with sequential prefetch.
- **Demand loading** of everything: of the executable and of the libraries.
- **Copy-on-write** at page level, both in the `fork()` and in freshly requested dynamic memory.
- **Shared libraries** by default.
- And, in the PCB, the base address of the process’s page table. In the context switch of Chapter 6 the incoming process’s one is loaded and the TLB is invalidated, which is no small part of what that switch costs.

## 8.8 Exercises

1. Run `mem-map` twice and compare the addresses. Which of them change and which do not? Now do it again with `setarch -R ./mem-map`, which turns the randomisation off.
2. In `mem-map`, add a write over the constant: `((char *)constant)[0] = 'X';`. Does it compile? Does it run? Which signal do you get, and why do you get it instead of seeing the constant changed?
3. Make `mem-mmap` ask for the region with `PROT_READ` alone and try to write into it. Compare what happens with the previous exercise.
4. Compile any program at all with `gcc` and with `gcc -static`, and compare the sizes of the two executables and the two `/proc/PID/maps` maps. Where has the difference ended up?

---

<sup>\*</sup>On Intel the hardware forces you to go through segmentation, but Linux sets it up so that it does nothing and works with pages alone.

5. Write a program that asks for ten megabytes with `mmap()` and does not touch them, and have a look at the `VmRSS` field of `/proc/self/status*`. Now write a byte into every page and go back to it. Which of this chapter's tricks have you just seen the effect of?
6. In `mem-shared`, take the `waitpid()` out. Does 42 still come out with `shared`? Always? Why is this a question for Chapter 7 and not for this one?

---

\*`VmSize` is the size of the address space; `VmRSS`, the physical memory it really has assigned to it.