

# Practical Work: Process Affinity

Master CSMI  
Compilation & Performance  
Bérenger Bramas

October 8, 2024

## 1 Summary

In this practical work, you will analyze how the pinning of the processes works and can be beneficial, but also what aligned memory allocation means.

## 2 Ressources

- `taskset` command man page: <https://linux.die.net/man/1/taskset>
- `sched_getcpu` man page: [https://linux.die.net/man/3/sched\\_getcpu](https://linux.die.net/man/3/sched_getcpu)
- `sched_getaffinity` man page: [https://linux.die.net/man/2/sched\\_getaffinity](https://linux.die.net/man/2/sched_getaffinity)
- `aligned_alloc` man page: [https://en.cppreference.com/w/c/memory/aligned\\_alloc](https://en.cppreference.com/w/c/memory/aligned_alloc)

### 2.1 Get the practical work

Consider you are in your project directory do the following:

```
# Clone my repo
# If you use SSH, use:
# git clone git@git.unistra.fr:bbramas/csmi-tp-2024.git --branch=TP2 csmi-tp2
# With https
git clone https://git.unistra.fr/bbramas/csmi-tp-2024.git --branch=TP2 csmi-tp2
# Go in the newly created directory
cd csmi-tp2
```

### 2.2 Add your repository as remote

You will push on your own repository:

```
# Rename my remote
git remote rename origin old-origin
# Add your own remote
# If you use an SSH key:
# git remote add origin git@git.unistra.fr:[YOU LOGIN HERE]/cnp-tp-2024.git
# If you use https:
git remote add origin https://git.unistra.fr/[YOU LOGIN HERE]/cnp-tp-2024.git
# Push the current branch and active the tracking
git push -u origin TP2
```

### 2.3 Compilation

To compile, we use CMake:

```
cd TP2
mkdir build
```

```

cd build
cmake ..
make # Will make all
make something # Will build only something
VERBOSE=1 make # Will show the commands used to compile (including the flags)

```

By default CMake will not use any optimization flag, and looking at the output of `VERBOSE=1 make` will show you that there is `-Ox`. To enable optimization you have to specify to cmake to generate a make file with the correct flags. To so, you can use `ccmake`. (in the build directory, note that `ccmake` should be installed) and edit `DCMAKE_BUILD_TYPE`, or directly set the variable with `cmake.. -DCMAKE_BUILD_TYPE = Release` in the build directory.

## 3 Reminder

### 3.1 Processor affinity

As we have seen in class, a CPU can have multiple cores, where each core can execute a different process. The OS is in charge of distributing the processes on the cores and decides based on criteria (priority, etc.) and heuristics which processes should be executed. Being preempted - remove from a core or move to another core - have significant execution penalty in terms of performance, because it means that the context of the process has to be stored and restored and that the data that were moved close to the core (in the different levels of the cache) might need to be reloaded from higher-levels. Therefore, when you execute simulations and try to get the best performance, or if you need to do an accurate execution time measure, the process must be pinned (bind) to a core. To pin a process, one has to inform the Linux scheduler about it, and there are several ways to do so. The first approach is to inform the scheduler before an execution starts, it has the advantage of being transparent for the target application (this can be achieved with `taskset`). The second method is to use the programming interface to the Linux scheduler to express the fact that a given process should run on some specific core(s). The good news is that this can be done directly from the process itself (see `sched_getaffinity`).

### 3.2 Aligned memory

Memory address  $p$  is say to be  $X$ -aligned if  $p \bmod X$  is 0. In current systems only power of two alignment are meaningful, and  $p$  is aligned to  $2^Y$  if the first non zero bit in  $p$  is at position  $Y$ . For instance, 1101.0101 is aligned to 1, but 1010.1000 is aligned to 8.

By default the classical allocation functions do not perform any specific alignment, and thus the addresses they return should be considered as aligned to 1 even if they might be aligned to a higher degree. One of the reasons is that the regular allocators focus more on filling the blank (having efficient memory filling pattern) rather than returning aligned blocks. Having aligned memory matters when we deal with performance because the caches work with lines of 64 Bytes (that must be 64 aligned). If a value at position  $@v$  is used, the cache line that is 64 aligned and which includes  $v$  will be moved  $((@v/64) * 64)$ . Moreover, if the compiler knows that a block is aligned, it can use specific instructions to load them into register, as we will see when we will vectorize a code.

There are different possibilities to have aligned allocations. The first one is to use existing functions, such as the C11 `aligned_alloc`. The second is to create your own allocator on top of `malloc/new`.

## 4 Pinning

### 4.1 Print the available CPU core

The file `print_av_cores.cpp` contains the code to know the pinning of the current process, and print the list on the standard output. Using the `taskset` command try to do the following:

- pin the process on a single core
- pin the process on two cores

You should obtain output like:

```

$ ./print_av_cores
Available cores =
0  1  2  3  4  5  6  7

$ taskset [something] ./print_av_cores
Available cores =
0

$ taskset [something] ./print_av_cores
Available cores =
0  1

```

## 4.2 Develop a kernel to print the used cores

Update the `print_moves.cpp` file to print everytime the process is moved on a different core. To ensure it does, run several instance of the application on multiple terminals. To know on which core is the current process, you can use the function `sched_getcpu`.

Example of desired output:

```

$ ./print_moves
>> Move to core 3
>> Core count : [0] 1    [3] 1
>> Move to core 7
>> Core count : [0] 1    [3] 1    [7] 1
>> Move to core 3
>> Core count : [0] 1    [3] 2    [7] 1
>> Move to core 7
>> Core count : [0] 1    [3] 2    [7] 2
>> Move to core 3
>> Core count : [0] 1    [3] 3    [7] 2
>> Move to core 7
>> Core count : [0] 1    [3] 3    [7] 3
>> Move to core 3
>> Core count : [0] 1    [3] 4    [7] 3
>> Move to core 7
>> Core count : [0] 1    [3] 4    [7] 4

```

## 4.3 Measure the performance cost of changing cores

Update the file `dot_with_move.cpp` such that in the second scope the process is moved on a different core after each call to `dot`. Notice that a list of available cores is obtained at the beginning of the main.

Example of desired output:

```

$ ./dot_with_move
Check dot
TestSize = 500000
>> Without move : 1.22568
>> With move : 4.4557

```

# 5 Memory alignment

## 5.1 Memory alignment evaluation

Fill the function `alignementOfPtr` in the first part of the code in file `aligned_malloc.cpp` such that the main will print the alignment of several imaginary addresses but also the addresses returned by C11 aligned malloc function. You can consider only alignment power of 2.

```

Test with hard numbers
Address 1

```

```

>> Alignement 1
Address 2
>> Alignement 2
Address 4
>> Alignement 4
Address 8
>> Alignement 8
Address 6
>> Alignement 2
Address 7
>> Alignement 1
Test with C11
alignment = 1
Address 0x5585a4a86280
>> Alignement 128
...
alignment = 2
Address 0x5585a4a86280
>> Alignement 128
...
alignment = 4
Address 0x5585a4a86280
>> Alignement 128
...
alignment = 8
Address 0x5585a4a86280
>> Alignement 128
...
alignment = 16
Address 0x5585a4a86280
>> Alignement 128
...

```

## 5.2 Create an allocator for aligned memory

Create your own function that will allocate aligned memory. It is advised to do it in two steps. In the first one, you can simply allocate more memory than asked, and return a pointer that is correctly aligned (but that will not be compatible with the regular free leading to memory leaks). In a second step, you will store the address of the real block just before the aligned pointer and use it to free as usual.

```

// First step
function aligned-malloc(alignment, size)
    size_with_extra_space = extra_need_space(alignment, size)
    unaligned_ptr = malloc(size_with_extra_space)
    ptr = get_first_aligned_address_in_block(alignment, unaligned_ptr)
    return ptr
end

// Second step
function aligned-malloc(alignment, size)
    size_with_extra_space = extra_need_space(alignment, size)
    unaligned_ptr = malloc(size_with_extra_space)
    ptr = get_first_aligned_address_in_block(alignment, unaligned_ptr)
    *(ptr-size_of_pointer) = unaligned_ptr
    return ptr
end

function aligned-free(ptr)
    unaligned_ptr = *(ptr-size_of_pointer)

```

```
        free(unaligned_ptr)
end
```