# Exercise session for lecture 3

Todays exercise is about working with threads, and specifically how to set the CPU affinity of your programs. This lets you pin threads to specific cores which allows you to e.g. limit thread-migration and data-transfer across core-caches.

We would like you to experiment with pinning the threads of your program to specific cores. Does pinning your threads to a specific set of cores improve performance? What behavior do you observe if you pin multiple threads to the same core? And does your performance decrease significantly if you pin across NUMA-nodes?

## Setting affinity through the shell

Using the `taskset` command, we can run a command with a specific CPU affinity. For example, running this in the terminal sets the CPU affinity of the script (and generally any processes launched by the it) to use cores #0,1,2,3:
`$ taskset -c 0-3 ./run.sh`

Alternatively, we can set the affinity of an existing process to cores #3,5,7 with:
`$ taskset -cp 3,5,7 <pid>`

The cores to pin a process to can be specified as a cpu-list as shown above, or as a mask. For more details look at the manual by running `man taskset`.

Alternately, if you want more control over scheduling and memory policies, you can use the numactl command.

You can see how many nodes are available on a server and which cores are associated with each node with:
`$ numactl -hardware`.

## Setting affinity programmatically

CPU affinity can also be set programmatically in some languages. You can set the affinity in C++ through the use of `std::thread` and `pthread` as seen in the following code.

**File: affinity.cpp**

```
1 #include <chrono>
2 #include <iostream>
3 #include <mutex>
4 #include <pthread.h>
5 #include <thread>
6 #include <vector>
7
```

```
8  int NUM_THREADS = 4;
9  std::mutex iomutex;
10
11 void work(int thread_index) {
12   std::this_thread::sleep_for(std::chrono::milliseconds(20));
13   while (1) {
14     {
15       // Thread-safe access to cout using a mutex:
16       std::lock_guard<std::mutex> iolock(iomutex);
17       std::cout << "Thread #" << thread_index << ": on CPU "
18                 << sched_getcpu() << "\n";
19     }
20     //while(1) ; //comment-in if you want to see things more visibly with
    htop
21     std::this_thread::sleep_for(std::chrono::milliseconds(900));
22   }
23 }
24
25 int main(int argc, const char** argv) {
26   std::vector<std::thread> threads(NUM_THREADS);
27   for (int i = 0; i < NUM_THREADS; ++i) {
28     threads[i] = std::thread(work, i);
29
30     // Create a cpu_set_t object representing a set of CPUs.
31     // Clear it and mark only CPU i as set.
32     cpu_set_t cpuset;
33     CPU_ZERO(&cpuset);
34     CPU_SET(i, &cpuset);
35     int rc = pthread_setaffinity_np(threads[i].native_handle(),
36                                     sizeof(cpu_set_t), &cpuset);
37     if (rc != 0) {
38       std::cerr << "Error calling pthread_setaffinity_np: " << rc << "\n";
39     }
40   }
41
42   for (auto& t : threads) {
43     t.join();
44   }
45 }
```

Note that this code is Linux-specific and adapted from this blog post which might be of interest to you: `https://eli.thegreenplace.net/2016/c11-threads-affinity-and-hyperthreading/`

On learnIT you'll find affinity.cpp as a file, as well as the no-affinity.cpp version seen below which doesn't set its own affinity. A makefile is also provided.

## The effects of setting affinity

On learnIT is the affinityperformanceexample.cpp file too. The affinity.cpp and no-affinity.cpp files gives a good example of how setting the affinity of a thread changes its behavior. The below example gives more insight to the effect on performance, when we pin threads to different cores.

**File: affinityperformanceexample.cpp**

```
#include <chrono>
#include <iostream>
#include <mutex>
#include <pthread.h>
#include <thread>
#include <vector>

int NUM_INCREMENTS = 10000;
std::mutex countermutex;
int counter = 0;

using namespace std;
using namespace std::chrono;

void work(int thread_index) {
  for (int i = 0; i < NUM_INCREMENTS; i++) {
    lock_guard<mutex> counterlock(countermutex);
    counter++;
  }
}

int main(int argc, const char** argv) {
  cout << "Example usage:" << endl << "./affinitynumaexample 4 0 8 16 24" <<
    endl;

  if (argc < 2) {
      cout << "Require atleast 2 arguments, number of threads and then CPU
   id for every thread." << endl;
      return -1;
  }
  auto n_threads = atoi(argv[1]);

  if (argc < n_threads + 1){
    cout << "CPU id must be specified for every thread" << endl;
    return -1;
  }

  cout << "Starting number of threads: " << n_threads << endl;

  auto start = high_resolution_clock::now();

  vector<thread> threads(n_threads);
  for (int i = 0; i < n_threads; ++i) {
    cout << "CPU: " << atoi(argv[2 + i]) << endl;
    threads[i] = thread(work, i);

    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(atoi(argv[2 + i]), &cpuset);
    int rc = pthread_setaffinity_np(threads[i].native_handle(),
                                    sizeof(cpu_set_t), &cpuset);
    if (rc != 0) {
      cerr << "Error calling pthread_setaffinity_np: " << rc << endl;
    }
```

```
53    }
54
55    for (auto& t : threads) {
56      t.join();
57    }
58
59    auto stop = high_resolution_clock::now();
60
61    auto duration = duration_cast<microseconds>(stop - start);
62    cout << "Counter value: " << counter << endl;
63    cout << "Duration: " << duration.count() << " microseconds" << endl;
64  }
```

The program takes as arguments the number of threads to start and then the affinity for each of the threads. It then measures how much time it takes for the threads to each increment a counter 10.000 times, while the counter is protected by a shared lock. As an example, the program can be called by ./affinitynumaexample 4 0 1 2 3.

You should experiment with different values for the affinity. What happens when you set the affinity to the first four cores? What about alternating cores (e.g. cores 0, 2, 4...)? What happens when you cross NUMA nodes? Which method do you expect to be fastest and why?

# Checking for info on your CPU

You can find info on the CPU you are using in multiple ways. One way is using `lscpu` command. Another way is to look at `/proc/cpuinfo` file. You can use `cat /proc/cpuinfo` to see its contents or use `grep` command to search for a specific info in this file.

Together with your CPU affinity exercise, observe the CPU speed using one of the ways mentioned above.

First, note down what `cpu MHz` says when you aren't running anything.

Then, note down what it says when you are running one of the CPP programs in the affinity exercise.

Finally, search for the processor in our class server (you can find the name of the model under `/proc/cpuinfo` as well) and see what its processor speed should be.

What do you observe? Was the results expected? Discuss among yourselves in class.

# System Monitoring Commands

## Top (`top`)

The `top` command is a system monitoring utility that provides a real-time view of the processes running on a system, including their resource usage and status. When executed, `top` displays a list of processes sorted by CPU usage, with the most CPU-intensive processes listed first. The display is updated regularly to provide an up-to-date view of the system.

## Htop (`htop`)

`htop` is similar to `top`, but provides a more user-friendly interface. `htop` displays a list of processes sorted by CPU usage, but also provides additional information about memory usage, system load, and process details such as the user running the process and its status. The interface is also more interactive, allowing users to perform actions such as killing processes directly from the `htop` interface.

## Cat `/proc/cpuinfo` (`cat /proc/cpuinfo`)

The `cat` command is used to display the contents of a file, in this case the file `/proc/cpuinfo`. This file contains information about the system's CPU, including its vendor, model, clock speed, and other details. The output of this command provides a snapshot of the current state of the CPU.

## Lscpu (`lscpu`)

The `lscpu` command is used to display information about the system's CPU architecture, including the number of CPUs, cores, and threads, the architecture type, and the clock speed. `lscpu` provides a more concise and organized view of the information contained in `/proc/cpuinfo`.

## Cat `/proc/meminfo` (`cat /proc/meminfo`)

Like `/proc/cpuinfo`, the `/proc/meminfo` file contains information about the system's memory, including the total amount of physical and swap memory, the amount of free and used memory, and other details. The `cat` command is used to display the contents of this file, providing a snapshot of the current state of the system's memory.

## Exploring `/sys/devices/system/node/node1/cpu10/cache/index0`

The `/sys/devices/system/node` directory contains information about the system's hardware, including information about the CPUs and memory. In this case, the path `/sys/devices/system/node/no` is being explored to obtain information about a specific CPU cache. The contents of this directory can be displayed using the `cat` command to inspect details about the cache, such as its size and associativity.

# Using Perf to track performance

In the `CoreAffinity` exercise, you have experimented with pinning threads to specific cores. In this experimentation, we have used time as an indication as to whether or not setting core affinity was beneficial or not. But what has *actually* changed? And how do we find out what actually changed?

Although time in general is a good indicator for performance, it can not be used on its own. In order to investigate exactly what has changed, we will utilize the Linux `perf` tool. This tool can be used in many settings, and in order to find the correct metric in a given case, one can type `perf list` in the command line in order to see all available metrics.

Although it may seem tempting to profile a program with all available events, this is rarely a good idea. Instead, one must hypothesize as to which metrics could be interesting in the given case. In the core affinity case, two metrics of interest could be context switches and CPU migrations. In order to profile a program for these specific metrics, one can simply type `perf stat -e migrations, context-switches <execute program>`. This will directly print the analysis performed by `perf` to stdout. If you instead wish to save the analysis, you can do `perf record -e migrations,context-switches -o <output name> python3 <execute program>`. This will produce a `.data` file. This can be converted to a .txt file with the following command `perf script -i <filename>.data> <filename>.txt`

# Other files

**File: no-affinity.cpp**: Creates 4 threads and prints the cpu that it's running on. Allows you to watch the threads migrate, and see whether your `taskset` commands worked.

```
#include <chrono>
#include <iostream>
#include <mutex>
#include <pthread.h>
#include <thread>
#include <vector>

int NUM_THREADS = 4;
std::mutex iomutex;

void work(int thread_index) {
  std::this_thread::sleep_for(std::chrono::milliseconds(20));
  while (1) {
    {
      // Thread-safe access to cout using a mutex:
      std::lock_guard<std::mutex> iolock(iomutex);
      std::cout << "Thread #" << thread_index << ": on CPU "
                << sched_getcpu() << "\n";
    }
    //while(1) ; //comment-in if you want to see things more visibly with htop
    std::this_thread::sleep_for(std::chrono::milliseconds(900));
  }
}

int main(int argc, const char** argv) {
  std::vector<std::thread> threads(NUM_THREADS);
  for (int i = 0; i < NUM_THREADS; ++i) {
    threads[i] = std::thread(work, i);
  }

  for (auto& t : threads) {
    t.join();
  }
}
```

**File: Makefile**: to compile the C++ programs. To compile, simply run `make`.

```
CXX = g++
CXXFLAGS = -std=c++11 -Wall -O3 -g -DNDEBUG -pthread
LDFLAGS = -lpthread -pthread

all: affinity no-affinity affinityperformanceexample

affinity: affinity.cpp
  $(CXX) $(CXXFLAGS) $^ -o $@ $(LDFLAGS)

no-affinity: no-affinity.cpp
  $(CXX) $(CXXFLAGS) $^ -o $@ $(LDFLAGS)

affinityperformanceexample: affinityperformanceexample.cpp
```

```
14  $(CXX) $(CXXFLAGS) $^ -o $@ $(LDFLAGS)
```