Lab 9 Daniel Hjelm:

Task 1:

Kolla denna igen. Får konstiga errors.

Task -2:

If we remove DoItNow we will have an infinite loop since the while loop in thread\_func is waiting for it to be set to 1.

If we declare a global variable DoItNow we waste CPU time since the other core will just loop the whole time. This can be fixed with a conditional variable. We need to use the four more functions, so when I want to use this, I should go back to the lab instructions.  
  
When I did not do this the CPU time was double the real time, but when I used this the CPU time was the same as the real time. This is not something that is done to reduce the real time, but rather something one does to reduce the CPU time.

If I remove the signal-call the thread will never wake up and the program wont terminate.

Task-3:

The difference between joinable and detached is that the main program will wait until the joinable threads are finished before finishing. If we have detached (but still calls pthread\_exit(NULL) in main) the main will finish before the threads but the threads will still finish and print after the main has ended. If we also remove the exit call in the main the threads will not print when finished.

Joinable: The main will wait for it

Detached: The main can finish before the threads

Detached without pthread\_exit in main: The threads are broken.

Task-4:

Here the barrier is used to separate the hello-calls and the bye-calls. With the barrier in place all hello-calls will be printed before the bye-calls. When we remove the barrier the hello-calls and bye-calls will be in any random order depending on which thread is faster.

The internal order of the hello and bye-calls will still be random ofc.

With barrier: separated calls

Without barrier: Mixed calls

The spin-method also works, but we can’t compile it with any of the O1, O2 and O3 and still expect it to work, it will not even terminate. We can however compile the Synch (barrier method) with –O3 with no problem.

Check on Volatile:

<https://barrgroup.com/embedded-systems/how-to/c-volatile-keyword>

<https://stackoverflow.com/questions/246127/why-is-volatile-needed-in-c>

Task-5:

Fixa så att stop och start inte kan bli negativa.

Task-6:

The task should be n^2 since for every element we need to go through the array once.

Time enumsort: 2.785256 s

Time enumerate\_better: 0.991059 s

So by creating fewer threads that can do more work each we make it faster.

Since I only got two cores, would 2 threads be faster? No, it was actually almost 2 s.

So there is a trade-off.

Task-7:

We could probably do half the matrix on one thread and the other half on the other thread.  
Or if we want to use more threads we can split the matrix into more parts. We just need to change the i-loop.

For n = 1000:

Without parallelization: Elapsed time: 3.355969 wall seconds

2 threads: Elapsed time: 1.704158 wall seconds

4 threads: Elapsed time: 2.392642 wall seconds

10 threads: Elapsed time: 2.438229 wall seconds

For small matrices, such as n = 10, the parallel approach is bad since it will take longer time to create the threads than we gain in speed.