Computer architecture and operating systems (DAT4, SW4)

Exercises

1) Consider a program that execute the code fragment below. Would there be any benefit from using multiple-threads in this program? On a unicore-machine? On a multi-core machine? Explain precisely!

```
data1=read(File); //20ms
sum1=compute(data1);//10ms
data2=fetch(url);    //50 ms
sum2= compute(data2);//10ms
printf(sum1+sum2);
```

2) Home-banking: Consider the pseudo code below where two threads transfers money from one back account to another.

```
   int kontoA=10; int kontoB=20;
   transfer(int * from,int * to, int amount){
       *from-= amount;
       *to+=amount;
   }
   Thread mand(){transfer(&kontoA,&kontoB,7));
   Thread kone(){transfer(&kontoA,&kontoB,4));
```

2.1) List all pairs of values (kontoA,kontoB) that the program can produce after the two threads have terminated

2.2) **(særlig vigtig)**Add a lock per account and lock them such that the bank does not have race-conditions: `transfer(int * from, mutex *from_lock, int * to, mutex*to_lock, int amount){`

2.3) Rewrite the version of transfer below to work correctly and nicely (short critical section and few exit points) with your locks.

```
transfer(int * from,int * to,amount){
    if(amount <0 || *from<amount)
      return false
    *from-= amount;
    *to+=amount;
    return true;
}
```

2.4) Implement your solution as executable pthreads code

2.5) will your code in step 2.2 and 2.3 also work correctly (the requirement of a bank is that money are never lost, ie. in a bank where money only can be transferred among its accounts, the sum of all accounts are constant) if there is concurrent call to the same accounts with a "int getBalance(int * accountA, mutex *accountA_lock, int * accountB, mutex*accountB_lock)" defined as follows

```
   int getBalance(int * accountA, mutex *accountA_lock, int * accountB, mutex * accountB_lock) {
    lock(accountA_lock);
    lock(accountB_lock);
    int balance=*accountA+*accountB;
    unlock(accountA_lock);
    unlock(accountB_lock);
    return balance;
   }
```

2.6) Will your code in step 2.2 and 2.3 also work if the threads use opposing order of the accounts? Hint: consider carefully the order in which your locks are acquired

```
Thread mand(){transfer(&kontoA,&lockA, &kontoB,&lockB,7));
Thread kone(){transfer(&kontoB,&lockB, &kontoA, &lockA,4));
```

3. Consider the get method of the approximative counter. In the current version it returns the value of the global count.
3.1) Design an alternative get(counter_t *c) method that also lock all local counters and return a sum of global and the local counters: getAll(counter_t *c) . **(særlig vigtig)**
3.2) Under what circumstances would this give (and would not give) an up-to-date count with good performance?
3.3) implement your solution and try it out in the virtual machine. You will need to change to main function to create a printCount thread that runs concurrently with the run_counter threads. The printCount function is to iteratively call getAll foloowed by a sleep(1); The books code can be fetched from below.

https://github.com/jonathantorres/ostep/blob/master/ch29/slop_counter.c
https://github.com/remzi-arpacidusseau/ostep-code/tree/master/include

4) CSPP 12.6 (p1031)

Challenge 13: Parallel Matrix Multiplication.

(see below about hardware/OS assumptions; the first two exercises can be done in the current VM)

Here is given a pthreads program that does matrix-multiplication according to the ijk method (CSPP3 page 681) and measures its execution time (in clock-cycles.) as function of the number of worker threads.

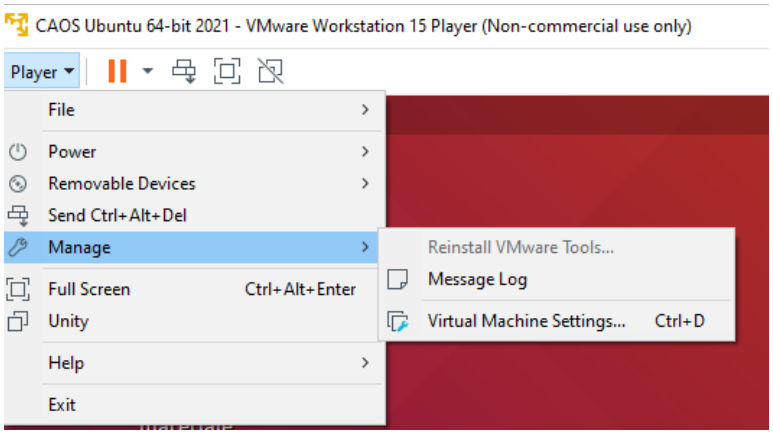
1) Why are no mutexes required in this solution?
2) What is the overhead of using more and more threads when you exceed the number of cores in your machine?
3) Plot the execution time as function of the number of worker threads. Also plot the speedup (here you may use the execution time of this program with one worker thread as approximation)
4) Also vary the problem size: try with matrix sizes: 500*500, 200*200, 100*100, 50*50, 20*20, 10*10, 5*5. At which point do you no longer gain from parallel execution?
5) Add the matrix-multiplication procedure using kij-method, and benchmark that as well. Beware your solution does not have race-conditions.

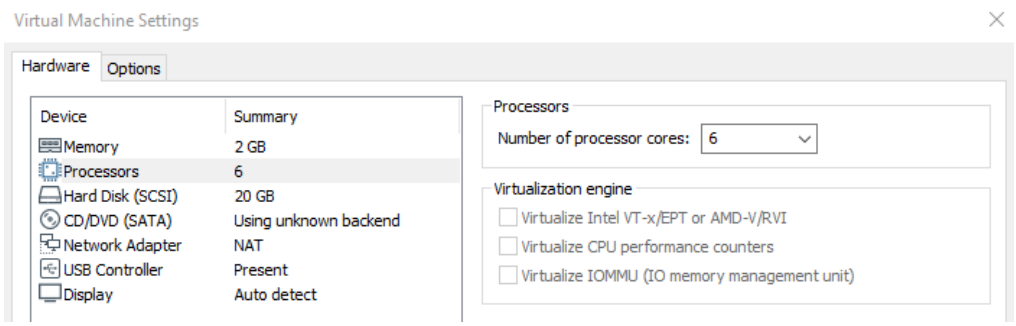
Hardware OS assumptions:

This kind of parallel benchmarking is normally best done using your native OS, but also works using the CAOS VM.

If you want to go native, Linux should work directly, and O expect macOS as well. On windows the easiest method is probably to enable and use the Windows Linux Sub-system and install the ubuntu shell (https://ubuntu.com/tutorials/ubuntu-on-windows#1-overview), then in the ubunto shell: gcc (`sudo apt-get install build-essential`), and fiinally the pthread librarry (apt-get install libpthread-stubs0-dev). There are pthread libs available, depending on what C-compiler environment you may have running.

If you use the CAOS Virtual Machine you may need to configure it with a number of processors corresponding to the number of cores in your physical processor. In vmware this is done under "Manage VM". This should be done while the machine is powered down (suspended is likely not enough); or at minimum requires a reboot of the VM.





O

Last modified: Wednesday, 13 April 2022, 11:28 AM

◄ 11-Slides

Jump to...

Solutions Sketch ►