Project 1
CS342 - Operating Systems
Bilkent University

Gokhan Simsek - 21401407

October 16, 2017

I measured the time of execution for each program with the built-in time command of Linux. E.g.

```
time ./integral 0 10 10 50

real 0m0.007s
user 0m0.000s
sys  0m0.000s
```

From these values, I have only used real time, since this shows how much time has passed from the start of the program until its end.

For the experiments, I have kept the intervals and the subintervals which will be used by children processes or the threads fixed. These fixed values are:
Lower bound (L): 0
Upper bound (U): 100
Subintervals (K): 100000

Since Linux's built-in time command is not very reliable for very small time intervals, e.g. nanoseconds, I have picked K to be quite large, so that time command can execute better and the results, relations and comparisons can be shown more clearly.

I have conducted the experiment for the following functions:

- $y = x^2$
  `(y = x * x)`
- $y = lnx$
  `(y = log(x))`
- $y = \sqrt{x}$
  `(y = pow(x, 0.5))`
- $y = x^4 - 82x^3 - 3x^2 - 10x + 11$
  `(y = pow(x,4) - 82*pow(x,3) - 3*pow(x,2) - 10 * x + 11)`

I have run each of the functions first with `integral` and then with `tintegral`. To find results quicker, I have written the following scrips, to run the programs with values $1, 5, 9, ..., 49$.

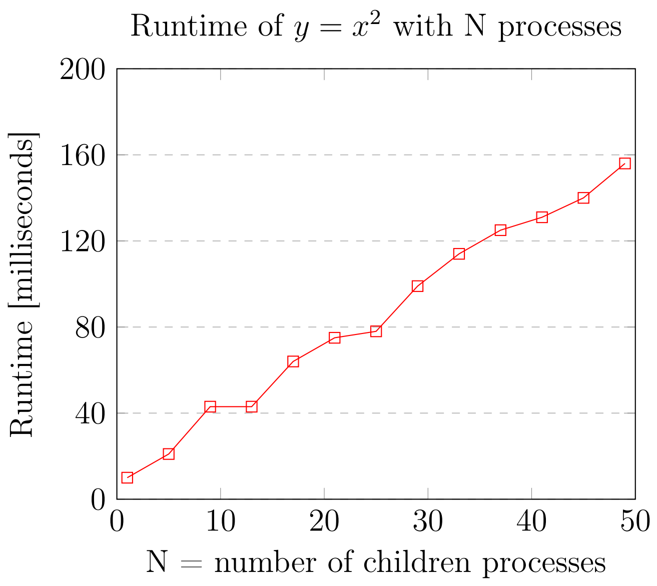
```
#!/bin/bash
for i in {1..50..4}
do
   echo N=$i
   time ./integral 0 100 100000 $i
done
```

The same script can be used by changing `./integral` to `./tintegral`.

1 $y = x^2$

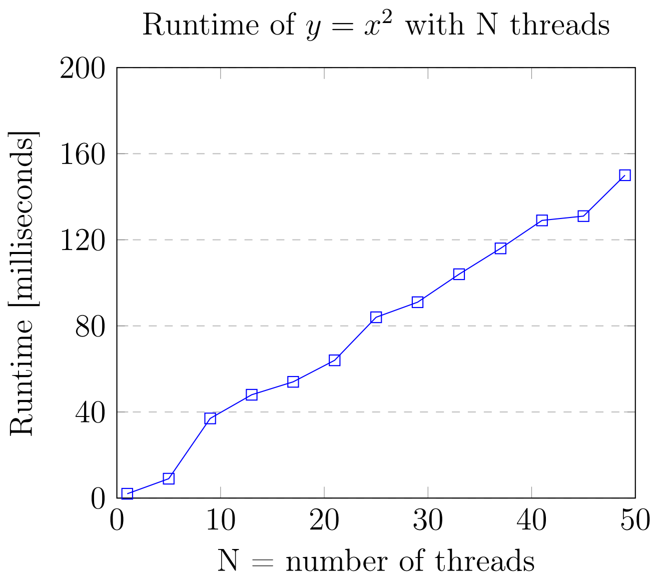
1.1 $y = x^2$ with integral.c (processes)

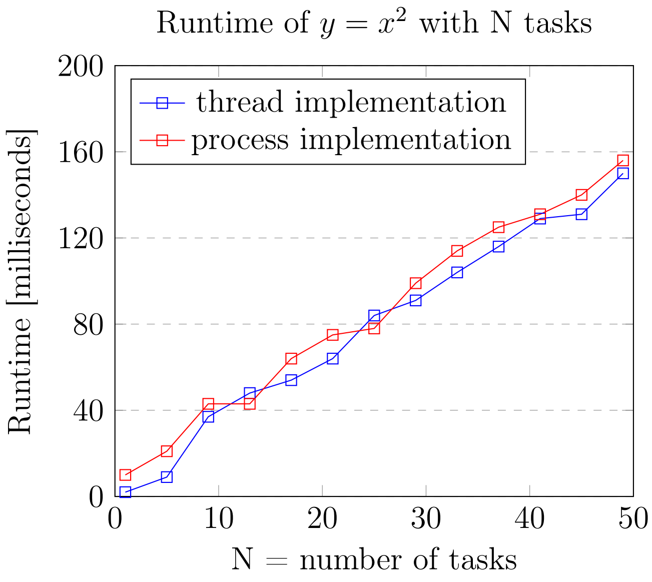
```
N=1  0m0.010s
N=5  0m0.021s
N=9  0m0.043s
N=13 0m0.043s
N=17 0m0.064s
N=21 0m0.075s
N=25 0m0.078s
N=29 0m0.099s
N=33 0m0.114s
N=37 0m0.125s
N=41 0m0.131s
N=45 0m0.140s
N=49 0m0.156s
```



1.2 $y = x^2$ with tintegral.c (threads)

```
N=1  0m0.002s
N=5  0m0.009s
N=9  0m0.037s
N=13 0m0.048s
N=17 0m0.054s
N=21 0m0.064s
N=25 0m0.084s
N=29 0m0.091s
N=33 0m0.104s
N=37 0m0.116s
N=41 0m0.129s
N=45 0m0.131s
N=49 0m0.150s
```



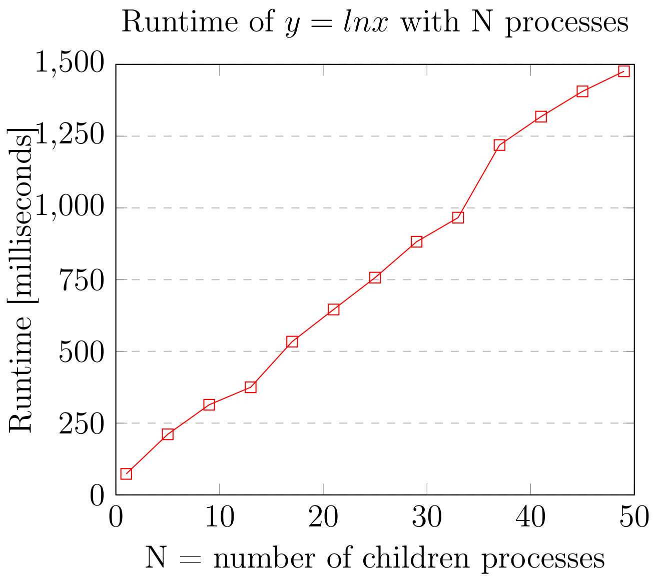


2 $y = ln x$

Since implementing $y = ln x$ requires the use of $log()$ function of <math.h>, and $log(0)$ returns $-inf$, this experiment has been conducted using 1 as the lower bound.

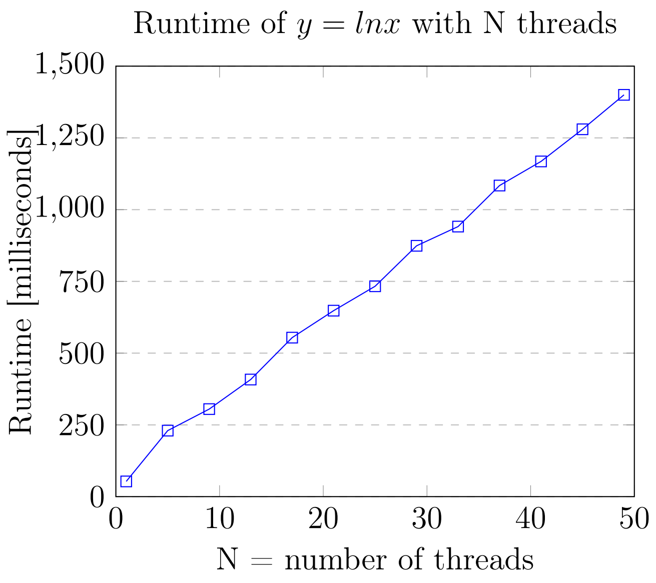
2.1 $y = ln x$ with integral.c (processes)

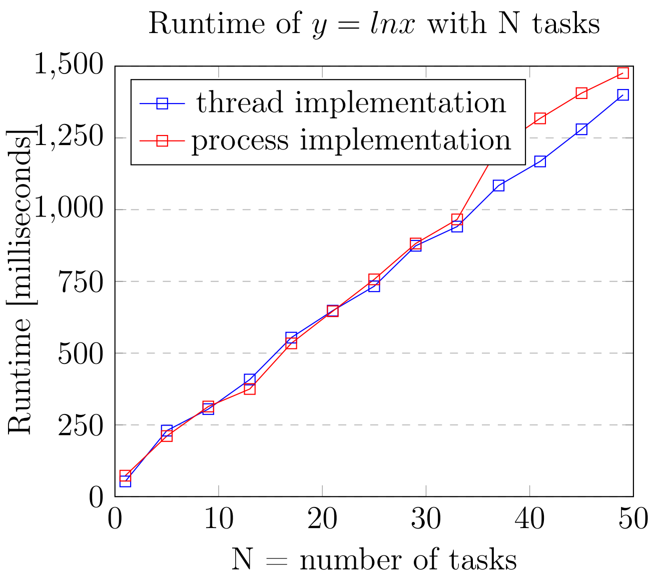
```
N=1  0m0.073s
N=5  0m0.211s
N=9  0m0.314s
N=13 0m0.375s
N=17 0m0.534s
N=21 0m0.646s
N=25 0m0.757s
N=29 0m0.882s
N=33 0m0.966s
N=37 0m1.219s
N=41 0m1.318s
N=45 0m1.406s
N=49 0m1.476s
```



2.2 $y = lnx$ with tintegral.c (threads)

```
N=1  0m0.053s
N=5  0m0.230s
N=9  0m0.305s
N=13 0m0.408s
N=17 0m0.554s
N=21 0m0.648s
N=25 0m0.733s
N=29 0m0.874s
N=33 0m0.941s
N=37 0m1.084s
N=41 0m1.168s
N=45 0m1.280s
N=49 0m1.400s
```

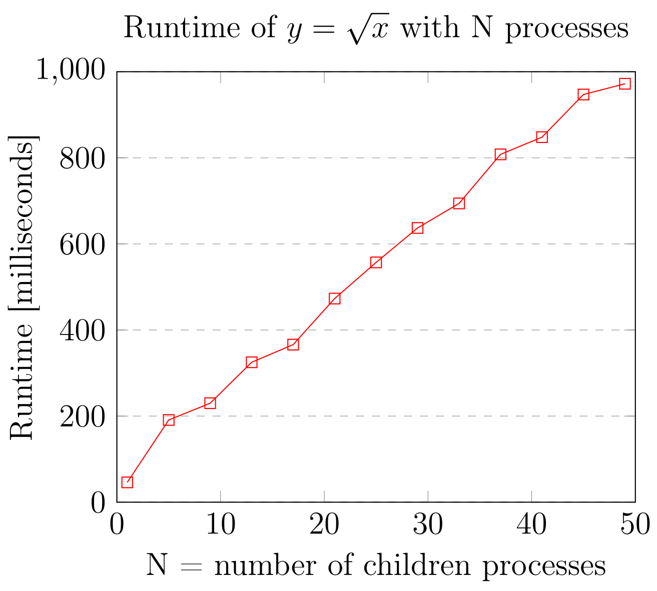




3 $y = \sqrt{x}$

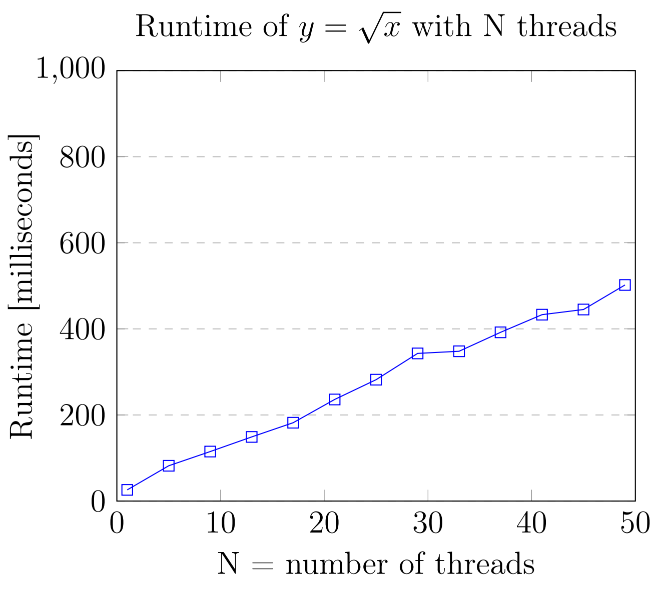
3.1 $y = \sqrt{x}$ with integral.c (processes)

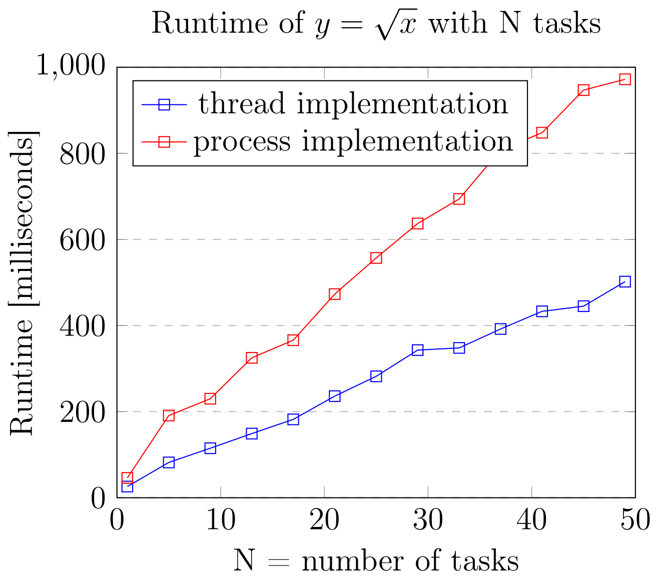
```
N=1  0m0.046s
N=5  0m0.191s
N=9  0m0.230s
N=13 0m0.325s
N=17 0m0.366s
N=21 0m0.473s
N=25 0m0.557s
N=29 0m0.637s
N=33 0m0.694s
N=37 0m0.808s
N=41 0m0.848s
N=45 0m0.947s
N=49 0m0.942s
```



3.2 $y = \sqrt{x}$ with tintegral.c (threads)

```
N=1  0m0.026s
N=5  0m0.082s
N=9  0m0.115s
N=13 0m0.149s
N=17 0m0.182s
N=21 0m0.236s
N=25 0m0.282s
N=29 0m0.343s
N=33 0m0.348s
N=37 0m0.392s
N=41 0m0.433s
N=45 0m0.445s
N=49 0m0.502s
```

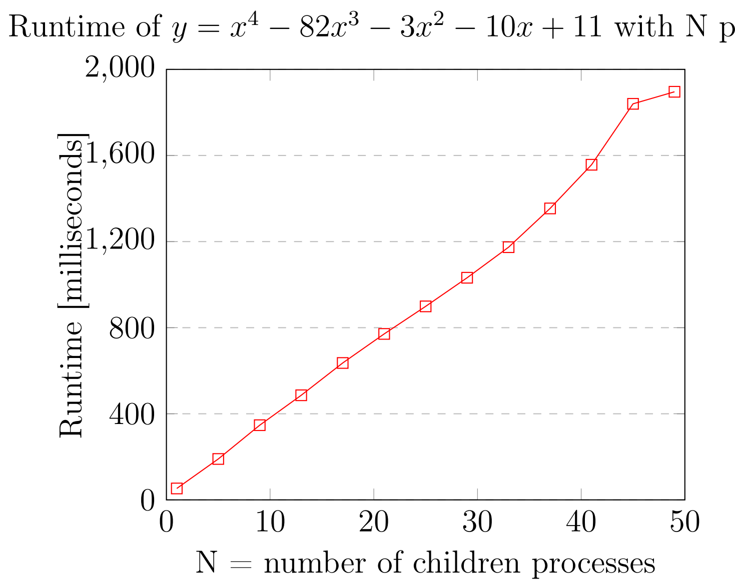




4 $y = x^4 - 82x^3 - 3x^2 - 10x + 11$

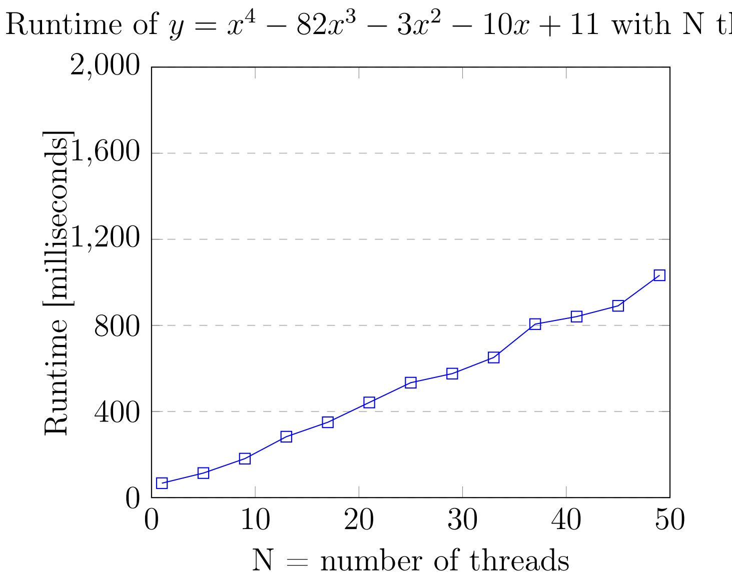
4.1 $y = x^4 - 82x^3 - 3x^2 - 10x + 11$ with integral.c (processes)

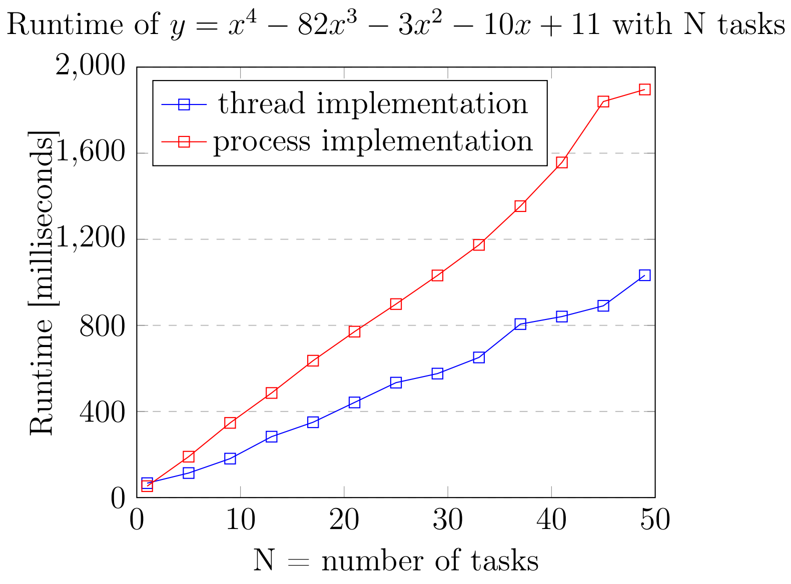
```
N=1  0m0.053s
N=5  0m0.190s
N=9  0m0.347s
N=13 0m0.486s
N=17 0m0.636s
N=21 0m0.771s
N=25 0m0.899s
N=29 0m1.032s
N=33 0m1.174s
N=37 0m1.354s
N=41 0m1.557s
N=45 0m1.840s
N=49 0m1.896s
```



4.2 $y = x^4 - 82x^3 - 3x^2 - 10x + 11$ with tintegral.c (threads)

```
N=1  0m0.067s
N=5  0m0.114s
N=9  0m0.181s
N=13 0m0.283s
N=17 0m0.350s
N=21 0m0.442s
N=25 0m0.534s
N=29 0m0.576s
N=33 0m0.651s
N=37 0m0.806s
N=41 0m0.841s
N=45 0m0.891s
N=49 0m1.033s
```





5 Discussion

5.1 Given different functions

We can see that different functions result in different run times. This depends on the complexity of the function. The functions that are not native in C, i.e. need a library and a function call to be calculated take much more time. This can be seen comparing $y = x^2$ to $y = lnx$. In implementing $y = x^2$, I haven't made any calls to `pow()` from `<math.h>`. Instead, I used `y=x*x`. However, the latter, $y = lnx$, requires the `log()` function from `<math.h>`, and so has a much greater runtime.

We can also see that as the number of function calls to different libraries increases, the runtime also increases, by comparing $y = \sqrt{2}$ and $y = x^4 - 82x^3 - 3x^2 - 10x + 11$. The latter calls the `pow()` function 3 times to calculate powers. So, it has a greater runtime compared to $y = \sqrt{2}$.

5.2 Processes vs. Threads

It is expected for threads to be faster, since they do not have to allocate additional resources for communicating with the main thread, and thus is more memory efficient and speedy. As process switching is more difficult since they require more memory, using processes is, generally speaking, slower.

In the first two experiments, conducted with $y = x^2$ and $y = lnx$, only one core is being utilized by the virtual machine. Thus, there isn't a big difference between the process and thread implementations.

However, the last two experiments, with $y = \sqrt{x}$ and $y = x^4 - 82x^3 - 3x^2 - 10x + 11$ are run by a virtual machine using two cores. A significant increase in the performance of the thread implementation (tintegral) can clearly be seen. Thread implementation (tintegral) outperforms process implementation (integral) by a factor of almost 2 to 1.