Problem Sheet #3

Problem 3.1: *readers / writers problem* (1+1+1 = 3 points)

Below are three incorrect solutions of the readers-writers problem. Explain why the solutions works or in which situations the solutions fail to work correctly. The solutions use the following common definitions:

```
shared object data;
shared int readcount = 0;
semaphore mutex = 1, writer = 1;
```

a)

```
void reader()
{
    down(&mutex);
    readcount = readcount + 1;
    if (readcount == 1) down(&writer);
    up(&mutex);
    read_shared_object(&data);
    down(&mutex);
    readcount = readcount - 1;
    up(&mutex);
    if (readcount == 0) up(&writer);
}
```

```
void writer()
{
    down(&writer);
    write_shared_object(&data);
    up(&writer);
}
```

b)

```
void reader()
{
    down(&mutex);
    readcount = readcount + 1;
    if (readcount == 1) down(&writer);
    up(&mutex);
    read_shared_object(&data);
    down(&mutex);
    readcount = readcount - 1;
    if (readcount == 0) {
        up(&mutex);
        up(&writer);
    } else {
        up(&mutex);
    }
}
```

```
void writer()
{
    down(&writer);
    write_shared_object(&data);
    up(&writer);
}
```

c)

```
void reader()
{
    down(&mutex);
    readcount = readcount + 1;
    if (readcount == 1) down(&writer);
    up(&mutex);
    read_shared_object(&data);
    down(&mutex);
    readcount = readcount - 1;
    if (readcount == 0) up(&writer);
    up(&mutex);
}
```

```
void writer()
{
    down(&writer);
    down(&mutex);
    write_shared_object(&data);
    up(&mutex);
    up(&writer);
}
```

Problem 3.2: *perfect numbers (multi-threading)* (2+3+2 = 7 points)

A *perfect number* is a positive integer that is equal to the sum of its positive divisors, excluding the number itself. For example, 6 has the positive divisors $\{1, 2, 3\}$ and $1 + 2 + 3 = 6$.

Write a C program called `perfect` that finds perfect numbers in a range for numbers. The default number range is $[1, 10000]$. The program accepts the `-s` option to set the lower bound and the `-e` option to set the higher bound. Hence, the invocation `perfect -s 100 -e 1000` will search for perfect numbers in the range $[100, 1000]$.

The following function can be used to test whether a given number is a perfect number:

```c
#include <stdint.h>

static int
is_perfect(uint64_t num)
{
    uint64_t i, sum;

    if (num < 2) {
        return 0;
    }

    for (i = 2, sum = 1; i*i <= num; i++) {
        if (num % i == 0) {
            sum += (i*i == num) ? i : i + num / i;
        }
    }

    return (sum == num);
}
```

a) Write a program that searches for perfect numbers in a range of numbers. Your program must support the `-s` and `-e` options to define non-default search intervals.

```
./perfect -s 100 -e 10000
496
8128
```

b) Implement an option `-t` that can be used to define how many concurrent threads should be used to execute the search. If the `-t` option is not present, then a single thread is used to carry out the search. For debugging purposes, implement an option `-v` that writes trace information to the standard error. Below is an invocation with two threads and a verbose trace.

```
./perfect -t 2 -v
perfect: t0 searching [1,5000]
perfect: t1 searching [5001,10000]
6
28
496
8128
perfect: t0 finishing
perfect: t1 finishing
```

c) Determine how the `-t` option impacts the execution time. Pick a search interval that is a reasonable load for your computer hardware and then increase the threading level and determine how the execution time changes. Produce a plot presenting the measurements you have obtained and discuss the results.