Problem 1

This is how timing constraints work in the system proposed in the Handout 10. Consider a user thread with actions A1 and A2 that generates some set of traces:

```
  A1 A2 A2 A1 A2 A1 ...
  A2 A1 A1 A1 A2 A2 ...
```

The Clock thread adds the clock ticks T to these traces by interleaving ticks T with the other actions in the trace:

```
  A1 T A2 A2  T T  A1 A2 A1  T ...
  T A2 A1 T  A1 A1  T A2 T  A2 ...
```

To get the execution of this trace in physical time, arrange the clock ticks equidistantly so that each happens every unit of physical time:

```
  A1    T A2 A2    T          T A1 A2 A1   T ...
        T A2 A1    T A1 A1    T A2        T A2 ...
```

So if there are more ticks interleaved with thread execution, this means that program actions take longer to execute. If a clock reaches a deadline D, then further ticks are disabled until the deadline is removed.

```
 ... T (deadline reached) A1 A2 A2 A1 (deadline released) T ...
```

This ensures that after assigning the physical time according to clock ticks, the time assigned to the part of the program where the deadline holds will never have time larger than deadline. In other words, deadlines guarantee that program actions will be packed densely enough to satisfy the timing constraints.

(The reason why this scheme may be confusing is that it does not say anything about how to achieve such schedule in practice, it just specifies the desired timing behaviors.
It lacks the constraints specifying the execution time of program basic program actions and a methodology for verifying that time taken according to these actions corresponds to the desired timing constraints which are stated. So all are talking about in the problem is a way of specifying how the program should execute in time.)

For part b), assume there is only Clock and one thread, so there can be no delay actions executing simultaneously. Also assume that the set of deadlines or time are not modified by the user program, the only modification is due to "delay" procedure.

By "time passes only in delay(k) actions" I mean that there should be no clock ticks allowed outside "delay" procedures.

The system in part b) should have roughly the following property. Consider a trace

```
  ... A delay(k1) ... delay(kn) B ...
```

If A occurs at time t and B at time t+p then

```
  p = k1 + ... + kn
```

In the presence of multiple threads, the analogous property would ideally hold. Consider any thread X and extract from the trace (preserving order) all actions generated by X and Clock. Then the same property should hold for such trace "projection":

```
  ... A delay(k1) ... delay(kn) B ...
```

If A occurs at time t and B at time t+p in the extracted trace, then

```
  p = k1 + ... + kn
```

Now in part c) you can check whether this property holds for your implementation for b). If it does not hold, you do not need to fix it, just explain why it does not hold. Alternatively, if your solution has some other major problem in multithreaded case, you can point to that problem. If your solution for part b) works fine in multithreaded case, explain how it achieves the "desirable property" that I just described. Again it is not an error if in c) the answer is that there are problems.

Problem 2

The criterion for when an optimization is better is the same throughout the problem, even in b) case.

Problem 3

Do not worry too much about the Poisson distribution, just use the formula for the expected service time from the handout.

Problem 4

You should add the minimal integer number of disks so that disk is not a bottleneck of the system.

Assume that writing x amount of data requires time

```
  latency + x/bandwidth
```

When you do a batching then you pay latency only once for the entire batch.

The "steady state" in this context is just a funny way of saying that you may use all simple formulas from the handout.