Lab 5

Problem 1

`deposit()` could be implemented as

```
 register0 = balance;
register0 = register0 + value;
balance = register0;
```

`withdraw()` could be implemented as

```
 register1 = balance;
register1 = register1 - value;
balance = register1;
```

Each function has its write back section, where `balance` is written. Thus the latter to write will hold the value.

With an initial value `balance = 5` and `value = 1`, we can see that `register0 = 6` and `register1 = 4`. If the program runs correctly, the final result should be 5. If for some reason `balance = register0` is executed later, `balance = 6`. And `balance = 4` vice versa.

Workaround

We implement a queue with control initialized to 1. For any process that leaves the queue, control will be decreased, then increased when the process finishes. No process can leave the queue if control is non positive.

Problem 2

- cond_usg
- nosynch

With synchronization, `p1` runs first but then hits cond signal and gives mutex permission to count threshold. `p2` runs with permission from cond signal, thus increasing `count` by `TCOUNT` or until `count == 20`. After reaching `count == 20`, cond signal is released so that `p1` can continue running, where `count += 80` then `count == 100`. `p1` releases mutex, letting `p2` and `p3` run, where `p2` runs the remaining (TCOUNT - (COUNT_LIMIT - 10)) iterations and `p3` runs all TCOUNT iterations.

Without synchronization, `p1`, `p2`, `p3` all try to write to `count` simultaneously, leading to inconsistent results in each execution. In the case two `inc_count` read and return `count` at the same time, `count` is increased by 1 unit where it should be increased by 2, hence the difference in final count.

Problem 3

For some reason `clock()` is not behaving properly. Regardless, using mutex on a global scope yields longer runtime as more mutex switch are needed, hence more instructions and more cycles for overhead.