Dor Gross, 039344999

Operating Systems
Exercise 3

Part 2

A.

The algorithm doesn't satisfy mutual exclusion.
Below is a counterexample showing step by step the progress of 2 threads and the values of the lock and the threads' "value" variables (in red, the affected values):

thread 1:
```
do {
  value = id;              // value = 1
  swap(&lock, &value);     // lock = 1, value = 0
} while (value != 0 && value != id);  // false

critical section
```

thread 2:
```
do {
  value = id;              // value = 2
  swap(&lock, &value);     // lock = 2, value = 1
} while (value != 0 && value != id);  // true
do {
  value = id;              // value = 2
  swap(&lock, &value);     // lock = 2, value = 2
} while (value != 0 && value != id);  // false

critical section
```

The chain of execution above would lead, even if swap is done atomically, to have both threads in the critical section in the same time, and therefore the algorithm does not satisfies mutual exclusion.

And in a more general note, every thread that performs the *while* loop twice without being interrupted by a context switch, will enter the critical section (as its *value* is overridden by the *id* in every loop and then compared to the *id* in the *while*'s condition).

B.

The *lock* variable is initially set with *lock=0*. Since *swap* is performed atomically, in every single moment there's at least one thread that pass the *swap* line hold *value=0*. W.L.O.G, let T1 be the first thread to hold *value=0* (at least one thread must pass the *swap* line first). Since *value* is local and not shared with the other threads, it won't be changed when T1 reaches the *while* condition, and therefore T1 will pass to the *critical section*. And since T1 will also reset the *lock* to hold *lock=0* when it leaves the *critical section*, the same proof holds for the next change of the *lock*, and another thread could enter the critical section. And since there's always a thread that can enter the *critical*

section, the algorithm is deadlock free.

C.

The algorithm doesn't satisfy starvation freedom.
Below is a counterexample showing step by step the progress of 2 threads and the values of the lock and the threads' "value" variables (in red, the affected values):

thread 1:
```
do {
  value = id;              // value = 1
  swap(&lock, &value);     // lock = 1, value = 0
} while (value != 0 && value != id);  // false
```

```
critical section
```

thread 2:
```
do {
  value = id;              // value = 2
  swap(&lock, &value);     // lock = 2, value = 1
} while (value != 0 && value != id);  // true
```

thread 1:
```
lock = 0;  // lock = 0
```

thread 3:
```
do {
  value = id;              // value = 3
  swap(&lock, &value);     // lock = 3, value = 0
} while (value != 0 && value != id);  // false
```

```
critical section
```

thread 2:
```
do {
  value = id;              // value = 2
  swap(&lock, &value);     // lock = 2, value = 3
} while (value != 0 && value != id);  // true
```

thread 3:
```
lock = 0;  // lock = 0
```

thread 4:
```
do {
  value = id;              // value = 4
  swap(&lock, &value);     // lock = 4, value = 0
} while (value != 0 && value != id);  // false
```

```
critical section
```

thread 2: ...

And thread 2 could be starved, while the other threads constantly enters the critical section.

Part 3

A.

The suggested solution **does not** satisfy the requirements, as the call *B.count--* is not an atomic operation and 2 (or more) threads could concurrently perform the operation and decrease the *B.count* counter by less the required, causing all threads to wait forever in the *while* loop afterwards.

The 2 disadvantages are:

1. The wait is implemented using a **busy-wait** technique which consumes a lot of CPU and postpones other threads' running time.
2. The barrier's *count* counter **might** never reach 0 when 2 threads (or more) address it concurrently, and that will cause a deadlock.

B.

The suggested solution **does not** satisfy the requested behavior. Whenever the first thread will finish the first part and invoke the wait operation, it will hold the monitor and perform a busy-wait loop which will not release the monitor while doing so, and by doing it, the first thread will prevent other threads from entering the synchronized block (the barrier's critical section) and decreasing the number working threads (those which are still in the first part).

The disadvantage is clear - the barrier does not perform what it should, and a **deadlock would occur**.

C.

The suggested solution **satisfies** the requirements, as each thread invoking the *wait* operation will decrease the amount of non-waiting threads by 1 (even when addressed concurrently, due to the monitor).

The disadvantage of this implementation is that the wait is implemented using a **busy-wait** technique which consumes a lot of CPU and postpones other threads' running time.

D.

The suggested solution **satisfies** the requirements, as each thread invoking the *wait* operation will decrease the amount of non-waiting threads by 1, and wait (using the monitor) until all threads had invoked the *wait* operation. Whenever the last thread will enter the *wait* method, it will decrease the counter to 0 and will wake up (notify) all other waiting threads.

There is no obvious disadvantage with this implementation, but for some usages, when the order of threads might have some importance, this implementation would not necessarily promise the order of notifying the other threads.