Question 1

If the function that forms the body of a process returns, it means that the process had completed to its execution. Since processes are usually expected to run indefinitely, the return of a process's main function might indicate an unintended termination.

What could have happened in the background could have been a myriad of things, such as system halt or an error being raised or maybe the process might be restarting.

Question 2

If a process tries to send a message to itself, either it can successfully be queued, or it could be ignored or blocked. It depends on the implementation of the messaging system embedded in the `micro:bian`.

A potential problem that can occur with doing this could be that a deadlock may occur. For example, if the process blocks waiting for a message from itself that can only be processed when the process is running - a deadlock will occur in such a scenario.

Question 3

Imagining the processes as a node in a directed graph, removing cycles from graphs can be done in this abstract way.

Abstracting to graphs in general, removing cycles consists of removing all paths that begin and end with the same node. This can be done by traversing through such graph and marking each node visited. If the traversal algorithm stumbles upon an already visited node, that edge should be deleted.

So, in this context, knowing that, we can do the following. We can mark processes with some kind of pointer or variable. If the subroutine or the processes are well defined in the system, this can be done relatively easily. Then, keep following the flow of the processes with a routine similar to a Depth-first search. Mark nodes visited as visited and unmark them as the algorithm backtracks. Even in this case of unmarking - a cycle won't be unmarked and can be detected. Such cycles can be deleted by invalidating such a process.

Since this algorithm is a process itself, it should be run periodically to ensure no such cycles can occur in the program.

Question 4

Given the nature of concurrent execution - while not being synchronised - the output loses its correctness. Since we can't determine whether or not the value of the stack remained consistent during the concurrent execution, we can't be certain about the correctness of the output.

Although, we can get some output that might be predictable.

- `proc1` may execute its printing loop before `proc2`, causing a sequence of zeros to be printed.
- `proc1` could simultaneously work with `proc2`, and print an increasing sequence of `r` from 0 to 100,000.
- `proc2` may run before `proc1` prints anything, causing a sequence of all 100,000s to be printed.
- Or, we may get a somewhat increasing sequence, but `proc1` and `proc2` might run before each other, printing a partially ordered increasing sequence.

Replacing `start` and `init` only helps to alter the initial scheduling of the processes. If no such priority schemes exist, reordering the `start` calls could influence the observed behaviour in a consistent manner. For example, starting `proc2` before `proc1` might slightly increase the chance that `proc2` increments `r` a few times before `proc1` starts printing.

Question 5

Part a:
When a process sends a character to the serial driver, the following context switches occur when sending process to the Kernel/Driver, or when the Kernel/Driver is sending back to the process. This means there are at least 2 context switches for each character being sent.

Part b:

Let T = time per character, b = Baud Rate. Therefore

$$T = \frac{10 \text{ bits}}{b}$$

and rearranging for b, we get

$$b = \frac{10 \text{ bits}}{T}$$

since T = 40 microseconds (due to part a), we get

$$b = \frac{10 \text{ bits}}{0.00004} = 250,000 \text{ bps}.$$

Part c:

To reduce the number of context switches per character output, we can try buffering. Instead of sending characters individually, the sending process can accumulate multiple characters into a buffer. When the buffer is full, the entire buffer is sent to the serial driver in one go. One other potential solution is to increase the payload from one character to a string of many characters, decreasing the average time spent doing context switches.

Question 6

The output is garbled due to the same reason as we saw in Question 4. Both the processes are running concurrently, but accessing the same data. This causes both of the processes running at the same time and resulting in the two slogans alternating with each other.

One way to solve this issue is to introduce a coordinating process that controls each speaker and when they are allowed to transmit their slogan. Alternatively, we can let each speaker complete a sentence before the other one intervenes, without having to resort to another process. We may use a flag to signify which turn it currently is.

```
#include "microbian.h"

static volatile int turn = 0;

void put_string(char *s, int myTurn) {
    while (1) {
        if (turn == myTurn) {
            for (char *p = s; *p != '\0'; p++) serial_putc(*p);
            turn = (myTurn + 1) % 2;
            return;
        }
    }
}

void speaker(int n) {
    while (1) {
        put_string(slogan[n], n);
    }
}

void init(void) {
    serial_init();
    start("May", speaker, 0, STACK);
    start("Farage", speaker, 1, STACK);
}
```

Even though this solution outputs the correct solution, the CPU may end up waiting whenever the program is checking who's turn it is to speak. This could be mitigated with the use of some pipelining and scheduling.