An Approach to Creating Strongly Interruption-Safe Programs in Bloxd.io

In this document I would introduce a kind of design choice that would guarantee **strong interruption safety**(which I will define later in the document) of a program, prove any program can be written in such a manner, introduce an engineering trick that helps reduce the difficulty of coding in such a manner, and introduce a framework that implements this design choice along with the engineering trick, and supports delay and concurrent execution.

Definition

Before we introduce the design choice, I would like to define a few concepts. Since Interruption is a phenomenon unique to bloxd.io programming, terms describing it are not very well defined and don't usually align with the usual definition in the programming industry. In the scope of this document,

- A piece of code that is **Uninterruptable** is a piece of code that includes no **interruption point**, so that it is **impossible to interrupt** on the QuickJS interpreter level. Scripts of this kind can only consist of extremely low-level javascript operations and mostly lack real-life functionality as most core language functionalities including function calls and branching and not allowed. FrostyCaveman and pythonSnake have dug deeper into this topic in a discussion on the bloxd.io discord server

  For Example:

  ```
  let a,b;
  a = 0+1; b = 1*2;
  ```

  would be uninterruptable because value assignment operations, variable declarations and basic math operations don't have any interruption point

- An operation that is **atomic** is an operation that will either succeed or fail, and have no other possible state. In other words, if an interruption happens during the execution of an atomic operation, it will either cause the operation to **fail completely and have no side effect, or not affect the operation at all.** This includes

  - operations that are **atomic on the QuickJS interpreter level** including
    - extremely low-level javascript operations that are uninterruptable
    - native function calls that don't use a user-space callback, e.g. `Array.prototype.push()`
    - bloxd.io API calls, e.g. `api.setStandardChestItemSlot()`

  as well as

  - **user-coded operations that are carefully constructed to be atomic**. In a lot of cases, the **atomicity** of a user-coded operation needs to be fulfilled by adding additional checking code elsewhere.

    Here's a simple example of a function that changes 2 blocks atomically. It caches the blocks it needs to change before setting them, allowing another callback to check if the previous operation partially failed and try again if so. This means the 2 blocks are either both set, or neither set, as long as the chunks are loaded

```
//world code
const block1Pos = [0,0,0];
const block2Pos = [0,3,0];
let blockName_cache;
function atomically_change_2_blocks(blockName) {
    blockName_cache = blockName;
    api.setBlock(block1Pos, blockName);
    api.setBlock(block2Pos, blockName);
    blockName_cache = undefined;
}
function check_block_change() {
    if(blockName_cache!==undefined) {
        api.setBlock(block1Pos, blockName_cache);
        api.setBlock(block2Pos, blockName_cache);
        blockNamecache = undefined;
    }
}

function tick() {
    check_block_change();
}
```

```
//somewhere in a code block
atomically_change_2_blocks("Dirt");
```

- **Interruption Safety** is a quality of a program or a part of a program where it is able to recover from an unpredictable interruption that occurs at any time or anywhere in the code and still function correctly. This term is borrowed from concurrency programming but should not be confused with the original word. There are two types of interruption safety in the scope of this document.
    - A program or a part of a program with **Weak Interruption Safety** has a chance of losing track of its previous action in case an interruption occurs, but **avoids data corruption so future operations can still proceed** as usual.

      For example, a simple program that adds to a counter whenever a player gets killed by another player satisfies week interruption safety because interruptions can make it fail sometimes, but never cause data corruption

      ```
      //world code
      let counter;
      onPlayerKilledOtherPlayer=()=>counter++;
      ```

    - A program or a part of a program with **Strong Interruption Safety** always keeps track of its previous action when an interruption occurs. In other words, it creates the illusion that **interruptions can only slow down its execution(causing lag), but not affect its behaviour**. Once a program or part of a program that satisfies this quality gets initiated, it cannot be terminated by any interruption. This quality can also be described as **Fully Interruption-Safe** or **Completely Interruption-Safe**.

For example, this program that creates a 1000-block-high stone column satisfies strong interruption safety, because even if an interruption prevented it from setting a block at a certain y level, it will just come back in the next tick.

```
//world code
const starting_pos = [0,0,0];
const max_y = 3000;
let current_pos = starting_pos;

function tick() {
    for(let i=0;i<300;i++) {
        if(current_pos[1]>max_y)return ;
        api.setBlock(current_pos, "Stone");
        if(!api.isBlockInLoadedChunk(...current_pos))return ;
        current_pos[1]++;
    }
}
```

Generally, when referring to Interruption Safety, we are referring to **weak interruption safety** unless otherwise specified.

The design choice described in this document will be able to ensure **strong interruption safety** in a program, or at least core parts of the program that do not involve taking user input from callbacks.

Design

General Idea

Since all basic javascript operations, most native functions, and all API calls are considered atomic, no matter how complex a program is, it should always be possible to split it into multiple atomic parts linked with each other. In other words, it should be possible to draw a call graph representation of the program, where each node is an atomic operation.

Therefore, by performing each atomic operation step by step, letting them decide which operation to do next, and repeating the atomic operations if an interruption happens, we can get a program whose behavior cannot be altered by any interruption.

A simple framework that does the above can be written like this:

```
currently_running_function=undefined;
function tick() {
    if(typeof currently_running_function==="function")currently_running_function =
currently_running_function();
}
```

Each tick, this framework tries to run the current function(node in the call graph). If the function call succeeds, it will prepare to run the function returned by this function(the next node in the call graph) next tick. If the function call gets interrupted before it returns, the value assignment will not happen so the value of `currently_running_function` will not change, allowing the function to be called again next tick.

Atomicity is Not Enough

If you looked closely into the definition of atomicity in the above section, you might have noticed a problem here. By definition, if an interruption happens during the execution of an atomic operation, it will either cause the operation to **fail completely and have no side effect**, or **not affect the operation at all.** This means that when an atomic operation gets interrupted, **it might have already been successfully executed**. And since our example framework will call the operation again **whenever an interruption happens before value assignment**, every single function might be executed **multiple times**. This could lead to unexpected behaviour.

An example of this will be a simple function that appends a value to an array.

```
let array = [];
currently_running_function=function(){
    array.push(1);
    return undefined;
}
```

When ran in our simple framework,

```
currently_running_function=undefined;
function tick() {
    if(typeof currently_running_function==="function")currently_running_function =
currently_running_function();
}
```

When there is no interruption this function works fine. It's going to push an element into `array`, and set the `currently_running_function` to `undefined`, so nothing is ran next tick. However, if an interruption happens between

```
    array.push(1);
```

and

```
    return undefined;
```

the push operation will be successful, but the framework will think it isn't, since an interruption occurred and the value of `currently_running_function` is not changed. This means that the function will be called again next tick, so one extra element will end up in the array.

How do we solve this problem? well, one way to solve it is to make every single operation not only **atomic**, but also **idempotent**. an operation being **idempotent** means that repeating it multiple times has the same effect as running it once.

In the case of an array push operation, it can be split into two operations that are both **atomic** and **idempotent**. Code:

```
let array = [];
currently_running_function = function(){
    let len = array.length;
    return function () {
        array[len] = 1;
    };
}
```

The first operation records the length of the array, while the second operation sets the array's index at its previous length to a certain value. This takes advantage over the fact that `Array.prototype.push()` is not the only way to change the length of an array in javascript, as setting the value of a key that is out-of-bound will increase the `length` automatically. Both the first and second operation are **atomic**, as they only consist of one single value assignment operation. They are also both **idempotent**, as they are both assigning a pre-determined value to a pre-determined memory location. Thus, we achieve **strong interruption safety** in appending elements to an array.

Proof of Turing Completeness

NOTE: If you are just reading this article to figure out what the new framework is about, you can skip this section. It's just here to convince myself that any program can be made strongly interruption-safe with the new framework.

To prove that a program made solely out of operations that are both atomic and idempotent has **Turing completeness**, the easiest way is by writing an interpreter for a Turing-complete programming language. For this proof, I chose to write an interpreter for the BrainFuck programming language because of its simplicity.

Here's the code for a strongly interruption-safe BrainFuck interpreter. It runs the BrainFuck code given by a string, and then outputs the entire output when the program finishes in a weakly interruption-safe manner. It's compatible with the extremely simple framework I just provided. Note that this interpreter is only a work of concept. It does not have performance in mind.

```
//I'm trying to make a "nice" interpreter as defined by
https://www.muppetlabs.com/~breadbox/bf/standards.html
//reading at the end of the input gives u 0
let p, RAM, output, input, compiled_function_list;
const InstructionLookUpTable = {
    "+":function() {
        return changeRAM.bind({
            nextFunc: this.nextFunc,
            newVal: RAM[p]+1,
        })
    },
    "-":function() {
        return changeRAM.bind({
            nextFunc: this.nextFunc,
            newVal: RAM[p]-1,
        })
    },
    ">":function() {
        return changeP.bind({
            nextFunc: this.nextFunc,
            newVal: p+1,
        })
    },
    "<":function() {
        return changeP.bind({
            nextFunc: this.nextFunc,
            newVal: p-1,
        })
    },
    "[":function() {
        return RAM[p]?
compiled_function_list[this.nextFunc]:compiled_function_list[this.GotoFunc];
    },
    "]":function() {
        return RAM[p]?
compiled_function_list[this.GotoFunc]:compiled_function_list[this.nextFunc];
    },
    ".":function() {
        return changeOutput.bind({
            nextFunc: this.nextFunc,
            newVal: output+String.fromCharCode(RAM[p]),
        })
    },
    ",":function(){
        RAM[p] = input.getCharAt(0) || 0;
        return readInput.bind({
            nextFunc: this.nextFunc,
            newVal: input.substr(1),
        })
    }
}
function InterpretBrainFuck(code, inp) {
    //once the function get successfully called, it becomes strongly interruption-safe
    currently_running_function = function() {
        //initialize a few things
        RAM = new Uint8Array(9999); //minimum RAM requirement by the "nice" standard
        p = 0;
        output = "";
        input = inp;
        //do the actual compiling here
        let stack = []; //stack for analyzing square brackets
        let gotoLocations = [];
        compiled_function_list = [];
        //analyze matching brackets, parsing syntax
        for(let i=0;i<code.length;i++) {
            if(!(code[i] in InstructionLookUpTable)) {
                throw SyntaxError("Invalid Character in BrainFuck: "+code[i]);
            }
            if(code[i]==="[") {
                stack.push(i);
            } else if (code[i]==="]") {
                let index = stack.pop();
                if(index===undefined) {
                    throw SyntaxError("Unmatching Brackets in BrainFuck");
                }
                gotoLocations[index] = i;
                gotoLocations[i] = index;
            }
        }
        for(let i=0;i<code.length;i++) {
            compiled_function_list[i] = InstructionLookUpTable[code[i]].bind(
                {
                    nextFunc:i+1,
                    GotoFunc: gotoLocations[i],
                }
            )
        }
        //the last function that outputs the output
        //note that it is not idempotent
        compiled_function_list.push(()=>api.broadcastMessage(output));
        return compiled_function_list[0];
    }
}

function changeRAM() { //bind this function with an object to specify what to do
    let nextFunc = this.nextFunc;
    let newVal = this.newVal;
    RAM[p] = newVal;
    console.log("changing RAM", this, nextFunc, compiled_function_list[nextFunc]);
    return compiled_function_list[nextFunc];
}

function changeP() {
    let nextFunc = this.nextFunc;
    let newVal = this.newVal;
    p = newVal;
    console.log("changing P", this, nextFunc, compiled_function_list[nextFunc]);
    return compiled_function_list[nextFunc];
}

function changeOutput() {
    let nextFunc = this.nextFunc;
    let newVal = this.newVal;
    output = newVal;
    console.log("changing output", this, nextFunc, compiled_function_list[nextFunc]);
    return compiled_function_list[nextFunc];
}

function readInput() {
    let nextFunc = this.nextFunc;
    let newVal = this.newVal;
    input = newVal;
    console.log("reading input", this, nextFunc, compiled_function_list[nextFunc]);
    return compiled_function_list[nextFunc];
}
```

Anyways, I tried to make this thing run a hello world sample program I found online, and it worked pretty well (why the weird `\u{}` notation? it's to avoid the swear filter from filtering `Fuck`)

```
InterpretBrain\u{46}\u{75}\u{63}\u{6b}(">++++++++[<+++++++++>-]<.>++++[<+++++++>-]
<+.+++++++..+++.>>++++++[<+++++++>-]<++.------------.>++++++[<+++++++++>-]<+.<.+++.------.--
------.>>>++++[<++++++++>-]<+.","")
```

If you are in the bloxd.io discord you can see the results here. This can actually be a pretty big moment in bloxd.io development history because it proves that **strong interruption safety does not contradict Turing completeness**. Therefore **any program that does not directly interact with the outer environment can be made strongly interruption safe**.