

# Synchronising processes in Imagine Logo: Why and How

Peter Tomcsányi, [tomcsanyi@fmph.uniba.sk](mailto:tomcsanyi@fmph.uniba.sk)

Dept of Informatics Education, Comenius University, Bratislava, Slovakia

## Abstract

Several modern Logo implementations allow running concurrent processes. Imagine Logo is one of them (others are, for example, MicroWorlds EX and Terrapin Logo 2.3).

This feature is implemented not (only) because it is modern and recent operating systems allow its efficient implementation, but because it makes programming of many simple projects much easier.

On the other hand, from the theory of concurrent processes we know, that if we start using concurrent processes which share data, some specific challenges will inevitably emerge solving of which is easily possible only by introducing some means of interprocess synchronisation.

The article introduces the reader to processes in Imagine Logo. It shows a simple Logo project demonstrating the emerging challenges. The example is discussed to its details and the way how to solve it will be shown.

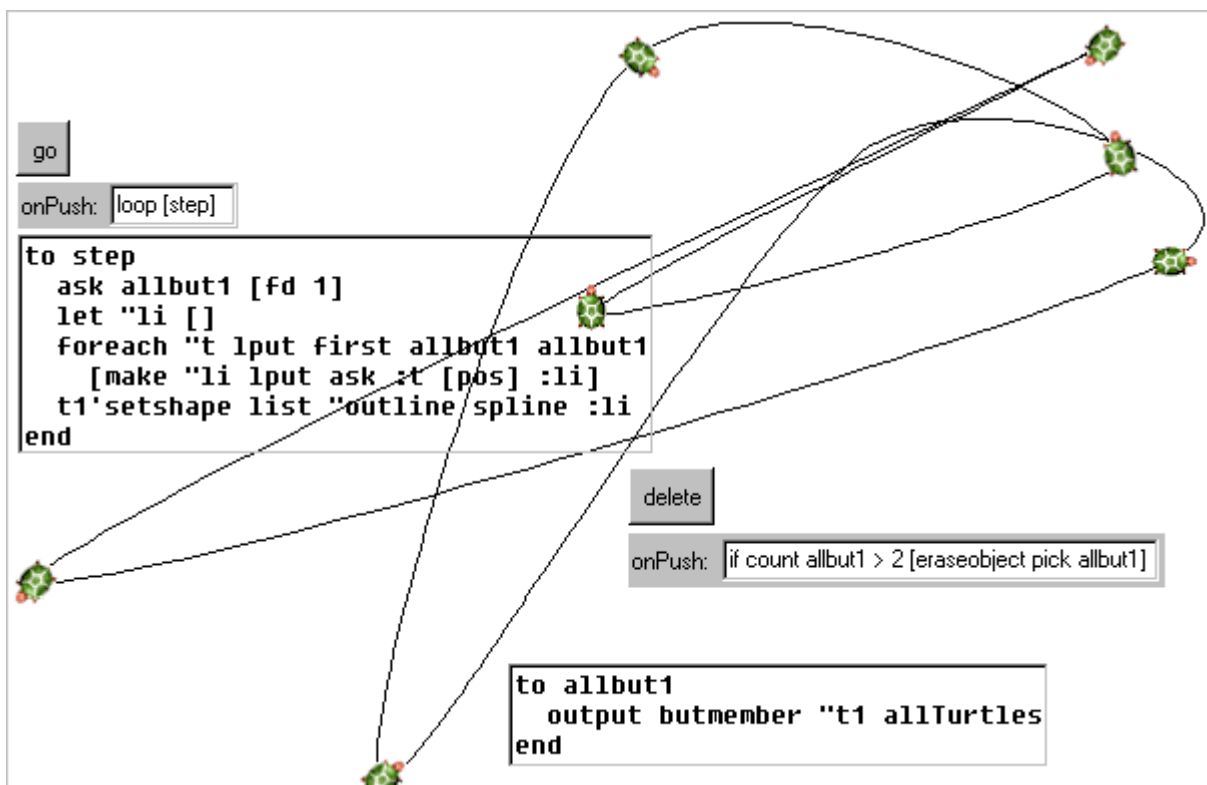


Figure 1. Our example project

## Keywords

Imagine Logo, concurrent processes, race condition

## Introduction

Several modern Logo Implementations allow running concurrent processes. Imagine Logo is one of them (others are, for example, MicroWorlds EX and Terrapin Logo 2.3).

This feature is implemented not (only) because it is modern and recent operating systems allow its efficient implementation, but because it makes programming of many simple projects much easier. Especially multimedia Logo projects and projects with user interaction can benefit from concurrent execution of processes. Thanks to processes it is easy to make things happen in the same time without complicated concepts like simulation calendars or timers.

On the other hand, from the theory of concurrent processes we know, that if we start using concurrent processes which share data, some specific challenges will inevitably emerge. Computer scientists name them race condition and deadlock.

Are these challenges only hypothetical or they are real? Can we make a simple Logo program which can demonstrate them? It seems that not only many Logo users but also Logo designers do not think that Logo is immune to such problems. And maybe also that the solutions would be too technical and not belonging into the Logo language.

In this article we would like to dispute such opinions by showing an example of simple Logo program, which can exhibit a race condition. The problem can be easily explained to anybody with intermediate level of understanding processes in Logo. And the remedy is not at all that complicated and technical if the used Logo dialect contains well-designed means for it.

## Processes in Imagine Logo

A process is a running program. Whenever a computer starts to execute some code independently of other code it starts a new process.

In this section we will give a brief overview of processes in Imagine Logo so that the reader can better follow the example programs, which we will show later in this article.

In Imagine Logo there exists always one process, it is called the Commander process. It reads the text typed into the command line and then executes it. While it is executing we cannot type another line.

A new process can be started in two ways:

- Programmatically by using the primitives `launch`, `forever`, `every` and `after`.
- As a reaction to some events.

### Starting a process programmatically

It means that at least one process must be already running and as the result of execution of the mentioned commands another process is created and starts running in parallel to the original one.

If we type into the command line:

```
loop [fd 1 rt 1]
```

Then the turtle will start moving in a circle and the command line will be occupied by executing the commands so we cannot type anything else.

If we type

```
forever [fd 1 rt 1]
```

The effect on the turtle will be the same, but the commands were launched in another process and the command line becomes usable again.

## Reacting to events

A mouse click to a turtle or pushing a button on the screen are examples of events, which can launch new processes. These run in parallel to other already running ones.

Many objects define several events, which may happen to them and the programmer can attach some code, which starts running (i.e. becomes a process) when that event happens. We will name them event processes.

To avoid firing many processes by repeating the event (e.g. a mouse click) before the previously launched event process finished, there are special rules for most kinds of event processes: a new event process can be started only if no event process for the same object and the same event is running.

## Processes run simultaneously in a random way

It is quite clear that if we have one processor and more processes then they must not run really in parallel, but they must share the processor in such a way that a small piece of each process is executed first, then a small piece of next process is executed etc. So from the user's point of view they seem really running in parallel. The planning i. e. determining when which process is running is in Imagine Logo left to the operating system. This planning depends on many factors and we should not expect any regular way of executing.

For example, we launch two processes. One will forever turn the turtle right by one degree and another will forever move it forward by one step:

```
forever [rt 1]
forever [fd 1]
```

We may expect that these two processes will run in turn. First process performs `fd 1` second `rt 1` then again `fd 1` etc. So the turtle should move on a circle. This would be the most regular way to execute these two processes. But it is not the case. This is, what happens in the reality:



Figure 2. Two concurrent processes

It is nearly a circle. But not exactly. It is because many other things happen in the operating system and must be planned in-between these two processes. So sometimes it happens that one of the processes is allowed to go for several steps until the second one is allowed to go. Therefore the result is not an exact circle.

This simple example can serve as a test how a Logo implementation implements processes. The test can show whether it tries to implement them entirely by its own or leaves the implementation on the operating system. When running this test in MicroWorlds EX or Terrapin Logo we get a perfect circle, which indicates that these Logo implementations probably handle processes themselves and loops in two running processes strictly alternate. Although this seems to be more regular for the user, we prefer the way how processes are implemented in Imagine

Logo because it allows the user to learn something deeper about parallel processes in real operating systems.

### Stopping processes

A process can stop by itself. Processes launched by the `launch` or `after` commands as well as all kinds of event processes stop when their execution reaches the end of their code. On the other hand, processes launched by `forever` and `every` commands execute in an endless loop until stopped explicitly. Any kind of process can be stopped if it executes the `stopMe` command. All processes can be stopped by the `stopAll` command.

Any process can stop another process (or itself) using the `cancel` command. To use it we must know the name of the process to stop. Normally the process name is equal to the list of instructions, which it carries out. For longer lists it may be not convenient and therefore the programmer can add an optional input to the commands that starts processes to define the process' name, which can be used for subsequent `cancel` calls.

Event processes get special names composed of the special character `@`, the name of the object and the name of the event. For example, a process started by pushing button `b2` will be named `@b2onPush`.

Note that stopping the Commander process means something different than stopping any other process. Stopping the Commander process means that the currently executed input line of code stops executing but the Commander process does not disappear, it will prompt us for typing another line. If we stop any other process then it completely disappears.

### Listing processes and waiting for processes

The primitive `allProcesses` outputs a list of names of all existing processes.

The primitive `done?` outputs `false` if the process, which name is given to the command, exists. Otherwise it outputs `true`. `Done?` in combination with the `waitUntil` primitive allows us to wait until a process finishes.

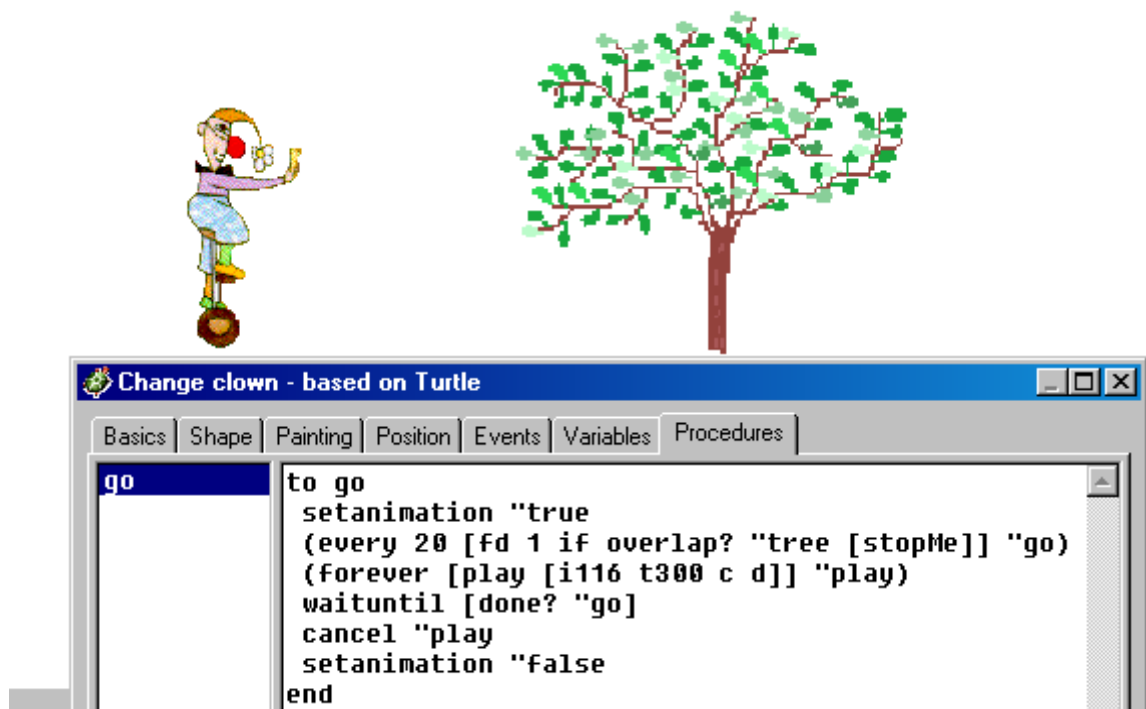


Figure 3. The clown is animated, he is moving and playing

The procedure `go` of the clown starts animating him (animating is made by Imagine Logo itself without the need of running any process), then starts moving (the process named `"go`) until hits the tree and in the same time some sound effects are started. Then the program just waits until the process `"go` stops and then it stops the playing process and also stops animation.

## The Animated Curve project

We have seen that processes are powerful and easy to use. And there seem to be no problems. So let's try another example.

We will try to make an animation, which is used commonly in screen savers. We will generate an animated curve, which changes its shape and bounces around the screen.

Start with a new project. The goal is to make some turtles on the screen with random headings, which will bounce off the page boundaries. Therefore we define an `onClick` event for `Page1`:

```
new "turtle [pos ( mousepos ) heading any rangestyle bounce pen pu]
```

Now click three or four times to the page to create some new turtles.

Turtle `t1` will have a special task - it will change its shape to a smooth curve connecting the positions of all the other turtles. We will need several times to instruct all the turtles except `t1` to do something so we define a function to get the names of all turtles except `t1`:

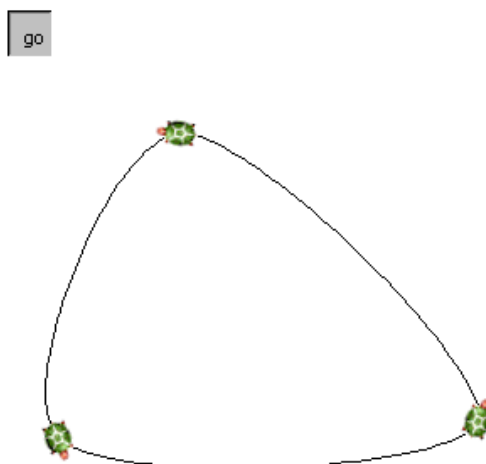
```
to allbut1
  output butmember "t1 allTurtles
end
```

Now let's add a button named `go` and program its `onPush` event to:

```
loop [step]
```

and define its procedure `step`:

```
to step
  ask allbut1 [fd 1]
  let "li []
  foreach "t lput first allbut1 allbut1
    [make "li lput ask :t [pos] :li]
  t1'setshape list "outline spline :li
end
```



*Figure 4. A curve defined by three turtles*

Now when we push the button, it will run its own process. The process at first moves all turtles a bit and then it will make t1's shape to be a smooth curve going through the positions of the three other turtles.

As the process is programmed for any number of turtles we can still click into the page. It will create another turtle and it will automatically participate in the curve.

After creating some more turtles by clicking the page we can write this to the command line:

```
ask allbut1 [ht]
```

and we will see only the bouncing curve.

Our next idea is to program another button, which effect will be deleting a randomly chosen turtle. Not to delete all of them we will at first check whether there are more than 2 moving turtles. So we create a button, name it "delete" and define its `onPush` event as follows:

```
if count allbut1 > 2 [eraseobject pick allbut1]
```

If we try the button then a few times it may work correctly, but then we will see an error like this:

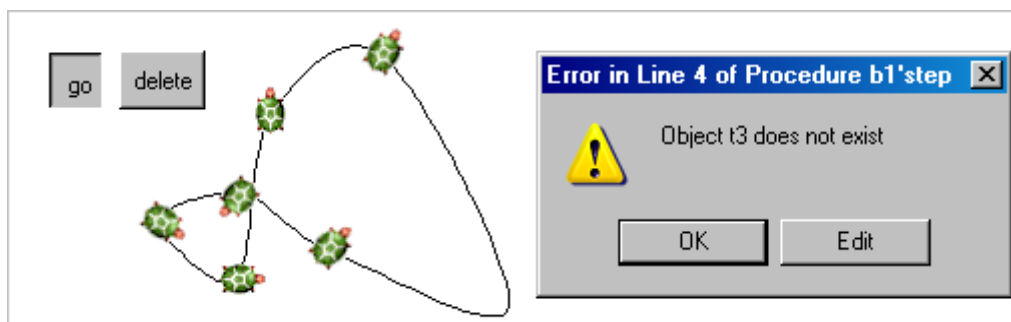


Figure 5. The error message after trying to delete a turtle while all the turtles are moving

Another strange thing may happen: after deleting a turtle the movement stops and the `go` button is released.

What did happen? If we look into the code then we can see that the third and fourth line of procedure `step` is a loop going through a list of turtle names. The list is constructed by calling `allbut1` and then inserting the first item of that list again at its end. So if we have turtles `t1`, `t2`, `t3` and `t4` then the result of `lput first allbut1 allbut1` will be `[t2 t3 t4 t2]`. So the loop will start with variable `t` having at first the value `"t2`, then `"t3` then `"t4` and then `"t2`. Inside the loop we use the value of variable `t` as an input to the function `ask` to get the position of the turtle whose name is stored in `t`.

If we manage to push the button at the moment when this loop is being executed and we delete a turtle, which has not yet been examined in the loop, then later the code in the loop's body will want to read its position. And the turtle will not exist. So we get an "object does not exist" error. That's it. The presence or non-presence of the error depends on when we delete the turtle i.e. when we push the "delete" button.

After all we understand what has happened. But what can we do?

We found the part of the code, which is responsible for the bad behaviour. If we were able to achieve that the process of deleting does not run while the process of moving is executing the loop in third and fourth line of procedure `step` then we would be able to solve the problem. But how to achieve it?

The solution would be not so easy without specific support for such situations, which is built into Imagine Logo. There is a primitive named `runCritical`. It has two inputs. The first input is a

word and the second input is a list of instructions. It runs its second input in such a way that no two `runCritical` commands with the same value of their first inputs can be run in the same time. It means that whatever is written inside the second input to `runCritical`, it will be protected against being interrupted by any other process running also a `runCritical` command with the same value of its first input.

We can now use the new command. We must insert it into two places. The first one is in the step procedure:

```
to step
  ask allbut1 [fd 1]
  let "li []
  runCritical 1 [
    foreach "t lput first allbut1 allbut1
      [make "li lput ask :t [pos] :li]
  ]
  t1'setshape list "outline spline :li
end
```

And the second place is in the `onPush` event of the delete button:

```
if count allbut1 > 2 [runCritical 1 [eraseobject pick allbut1]]
```

Now we can try again and we will see that we will never get the error message "object does not exist". But, if we try for a longer time (and it depends a bit on the speed of the computer) we can sometimes get another strange behaviour: after deletion of a turtle all the movement suddenly stops. Is this another mystery? No, it is the same kind of problem. Procedure `step` has another problematic place. It's its first line. Here we ask all turtles to move forward. Here it may happen that a turtle, which is currently executing `fd 1` is erased. And one of the rules in Imagine Logo says that if an object is being erased and any running process is being currently executed for that object, then that process must be stopped. Therefore the process of the `go` button stops.

The remedy is the same, the first line of the procedure `step` must be also protected by a `runCritical` command:

```
to step
  runCritical 1 [ask allbut1 [fd 1]]
  let "li []
  runCritical 1 [
    foreach "t lput first allbut1 allbut1
      [make "li lput ask :t [pos] :li]
  ]
  t1'setshape list "outline spline :li
end
```

Now our small project is complete and we should never get any errors regardless of how quickly we are trying to delete the turtles while the project is running.

## A bit more theory

As we already mentioned in the introduction, the situations shown in the previous section, are well-known to computer scientists and also to professional programmers - see, for example, Tanenbaum (2001).

So depending on the age and level of programming knowledge of our pupils or students, we can either stop at the level of details presented in the previous section, or we can go on and use the opportunity to tell them more from the theory of concurrent processes.

Computer scientists have studied such problems at the times when first operating systems were able to run several concurrent processes and programmers started to get into such problems. Since then our problem has a name. It's called **race condition**.

If the result of work of several processes depends on the exact order of execution of these processes by the operating system, we name it **race condition**. It means that running the processes for the same data we can many times get good results, but then suddenly we get a strange result or an error or a crash. So race conditions cause hard to debug bugs in our programs.

And exactly this happened in our previous example. We can push several times the delete button (which launches an event process running in parallel with the main process of `go` button) and a turtle is just deleted with no error. But then suddenly for another click on the button we get an error.

The reason of the race condition is that in our code we have **critical regions** - lines of our code that manipulate data, which can be shared by other running processes at the same time. We can eliminate the race condition if we identify all critical regions in our program and achieve that no two processes will be inside their critical regions at the same time. Such execution of critical regions is named **mutual exclusion**. Both operating systems and programming languages use to give some tools for achieving mutual exclusion. The tool found in Imagine Logo is the `runCritical` command, which can frame our critical sections and will arrange mutual exclusion of them. Some other Logo implementations, even if they include the ability of running several processes, do not contain such tools.

## Conclusion

We have shown one simple example of Logo program, where race condition happens. It demonstrates that in Logo implementations, which add concurrent processes some kind of synchronisation tools are needed as well. Imagine Logo provides synchronisation mechanisms for critical sections.

Showing such examples to intermediate and advanced learners can, in our opinion, enhance their understanding of processes, which can lead to both better using them in projects and getting some knowledge, which may be used also outside the Logo world.

The author of this article uses this example when teaching concepts of operating systems to future teacher in pre-service courses. These students already passed an intensive course of Imagine, several of them are advanced Imagine programmers and this example can well connect their previous knowledge with the new concepts found in the operating systems course.

## References

Blaho, A. and Kalas, I. (2001) *Object Metaphor Helps Create Simple Logo Projects*. In Proceedings of EuroLogo 2001. Edited by G. Futschek. Linz, August. pp. 55 – 66.

Tanenbaum, A. S. (2001) *Modern Operating Systems (2nd Edition)*, Prentice Hall, New Jersey