This page aims to describe how the game's scripting engine executes scripts - when a thread's code is executed - and how to measure script performance. This is a work in progress. Pointers to any existing work would be appreciated.

**Analysis**

We can learn a little about how the scripting engine works by running these two threads concurrently (i.e. by starting them both using **thread**):

testing1:

while (1) {

wait 10

iprintln\_noloc "start time: " level.time

for (local.i = 1; local.i < 4 \* 1024 \* 1024; local.i++) {

// we are just doing something insignificant here to take up time

local.dummy = local.i

}

iprintln\_noloc "end time: " level.time

}

end

testing2:

while (1) {

locprint 0 80 level.time

wait 0.2

}

end

When running these threads in a map script on a non-dedicated Spearhead server in objective mode, I observed that every 10 seconds, the 'for' loop would cause a delay of a number of seconds during which the game completely froze. Initially the "testing2" thread would be displaying the time in the lower-left corner of the screen every 0.2 seconds, but at the point where the "testing1" thread started its 'for' loop, the "testing2" thread would stop. Once the game had continued (due to the 'for' loop terminating), I observed that the "start time:" and "end time:" displayed on the screen were the same, and the time displayed in the lower-left corner of the screen by the "testing2" thread continued where it had left off rather than skipping ahead.

This shows us that:

* when a thread runs for a long time, the game's concept of "time" distorts from the real "time" you'd see on your wall clock - if your thread takes 5 seconds to run before it 'end's or uses a 'wait'-type command, the time in the game will not change during those 5 seconds, causing rounds to last an extra 5 seconds, bombs to not go off for an extra 5 seconds, etc. This makes sense as code in the game might not be able to handle time skipping forward. Probably level.time is incremented by the duration of a frame after every frame, i.e. if the server is running at 20 frames per second, once a frame has been processed, level.time probably has 1 second / 20 = 0.05 seconds added to it.
* threads are not scheduled pre-emptively: the CPU in your computer cannot run the code from every thread (or every program) simultaneously. In modern operating systems, programs are scheduled pre-emptively; in other words, every N milliseconds, one program's code will stop executing and another's will be started from where it left off last time it was executing. Here we see that the code in the "testing1" thread was allowed to run for a number of seconds without the "testing2" thread having a chance to run; to be more certain of this, I replaced the 'locprint' with an 'iprintln\_noloc' and certainly did not see any output from "testing2" in between the "start time:" and "end time:" outputs of "testing1". When, then, does the scripting engine allow another thread to run? The answer to that is obviously that one event which causes another thread to be allowed to run is when the currently running thread executes 'end', because the currently running thread is going to cease to exist. Otherwise, it would make sense for the game to allow another thread to execute when the currently running thread uses some kind of 'wait' command, such as 'wait' itself (which pauses for a certain number of seconds), 'waitframe', or 'waittill'. 'waitthread' is a special case - when I replaced the 'local.dummy = local.i' statement with a 'waitthread' call to a thread with the statement 'local.i = 1' in it (and also decreased the number of iterations to 1024 \* 512, since with even 1024 \* 1024 I got a "possible infinite loop" error [1]) I still saw that the "testing2" thread was not executed while the "testing1" thread was in the 'for' loop, so 'waitthread' apparently only causes the called thread to be executed, then causes the calling thread to continue executing when the called thread terminates, assuming no 'wait'-type statements in the called thread.
* from [1] above we see there may be significant cost involved in calling a thread - in this case it appears to be at least 4 times, and perhaps around 8 times (4 \* 1024 \* 1024 / 1024 \* 512 = 8), or even more, slower to call another thread to do an assignment than to do it ourselves, given that we would do 4 \* 1024 \* 1024 assignments in the "testing1" thread without getting a "possible infinite loop" error, whereas when we called another thread we got such an error with only 1024 \* 1024 assignments in a called thread.

Assuming we had some code which, like this contrived 'for' loop, needed to do quite a bit of processing, how would we make it not cause the game to freeze? We would of course need to make it 'wait'. However, if you put a 'waitframe' statement in the code above - you can't wait for any less amount of time than a frame, which by default is 1/20th of a second since the server runs at 20 frames per second (**which cvar controls this?**) - the loop will take 4 \* 1024 \* 1024 frames = 4 \* 1024 \* 1024 / 20 seconds = 209715.2 seconds = 58 hours, which is clearly too long. We would need to only do a 'waitframe' every N iterations, where N is chosen so that N iterations doesn't take too long, and "too long" means long enough to cause the game to freeze for a moment (which players would perceive as lag or packet loss and probably not realize is actually the server freezing for a moment). For example:

local.total\_iterations = 4 \* 1024 \* 1024

local.iterations\_per\_frame = 4096 // what I refer to as "N" above

local.frame\_count = local.total\_iterations / local.iterations\_per\_frame

for (local.frame = 1; local.i < local.frame\_count; local.frame++) {

for (local.i = 1; local.i < local.iterations\_per\_frame; local.i++) {

local.dummy = local.i

}

// will be executed after local.iterations\_per\_frame

// executions of "local.dummy = local.i"

waitframe

}

However, how do we pick a value of N? One answer is to do some experimentation - try different values of N and see how it affects gameplay. It might, however, be hard to determine whether your script is actually causing "lag". A more complex solution is to profile your code to measure how long each iteration is taking. Since one iteration will often not take very long, you'd generally want to do a large number of iterations, measure how long it takes to perform them all, then divide the elapsed time by the number of iterations, e.g. if it takes 5 seconds to do 1000 iterations, each iteration is taking 5 seconds / 1000 = 5 milliseconds. Profiling is made difficult by the fact that level.time doesn't increase while your code is running, so you can't use it to measure how long your code is taking to execute. I believe that enabling un-buffered logging using 'logfile 2', writing messages to the console using e.g. 'println "START"' and 'println "END"', and then having a script monitor the log file to measure the amount of time between "START" and "END" appearing in it, might be a way of measuring the time it takes to execute code, but it may be inaccurate.

Assuming it is possible to determine how long each iteration is taking, you can choose a value for N such that N \* (amount of time per iteration) is no greater than the frame length which defaults to 0.05 seconds as discussed above, since N \* (amount of time per iteration) is the amount of time your thread will execute each frame. Remember that each frame the game will have other processing to do, and other threads may be executing too, so you will need to use less than 0.05 seconds every frame in order to completely avoid any freezing. However, probably some level of freezing might not be perceptible to players, I'm not really sure about this.

Obviously, not all code is structured as a loop with a large number of iterations - it may consist of a number of smaller nested loops, or perhaps a number of loops executed one after the other. This does not make the discussion above less applicable. In the case of a sequence of loops, we could execute one 'for' loop, then use 'waitframe', then execute the next 'for' loop. Of course, since we know that a 'wait'-type command allows other threads to be executed, we must be aware of the concurrency issues - could some other thread interfere with our thread's operation? You may need to set a variable to indicate your thread is currently operating so some other thread avoids interfering with it, e.g.:

// does something every 5 seconds that threadB might interfere with

threadA:

while (1) {

wait 5

level.currently\_in\_threadA = 1

// do something time consuming here

waitframe

// do something time consuming here

waitframe

// do something time consuming here

level.currently\_in\_threadA = 0

}

end

threadB: // does something every frame that can disturb threadA

while (1) {

if (!level.currently\_in\_threadA) {

// do something quick which might interfere with threadA and which

// we can afford to NOT run for 2-3 frames while threadA is active

}

waitframe

}

end

**Something simple we can take from this**

One basic lesson we can take from our observations here is that it's best to not do all of our processing in one frame - it's probably best to spread it out across frames. We might have a thread like this:

give\_axis\_snipers\_silenced\_pistols:

while (1) {

for (local.i = 1; local.i <= $player.size; local.i++) {

local.player = $player[local.i]

local.weapon = waitthread get\_player\_weapon local.player

if (local.weapon && local.weapon.model == "models/weapons/kar98sniper.tik") {

local.player give "models/weapons/silencedpistol.tik"

}

}

wait 1

}

end

The thread above will check all players every second. If there were quite a few players on the server, when that frame comes around every second that the thread does the checking, it might cause that frame to freeze for a bit of time. The situation would be worse if other things were done in the same thread. To spread the processing across frames, we could insert a 'waitframe' at the end of the 'for' loop so we only check one player per frame, and then replace the 'wait 1' with another 'waitframe'. If we thought that 'wait 1' was an acceptable delay, i.e. it would be okay if there was a delay of up to a second between when the player started using a sniper rifle and they got their silenced pistol, then the 'waitframe' will be fairly similar with regards to delay - if there are 20 players on the server, then with the server running at 20 frames per second, it will take 20 /20 = 1 second for the 'for' loop to complete. We need a 'waitframe' in place of the 'wait 1' so that if there are no players, we still do a 'wait', otherwise we'd loop continually without a 'wait' and prevent any other thread from executing.

Is the code totally correct with a 'wait' in the 'for' loop? The number of players, i.e. $player.size, could change in between frames due to players joining or leaving the server. If there were 20 players on the server when we started the loop (i.e. $player.size was 20), then after the 19th iteration when we did a 'waitframe' one player left the server, in the next iteration local.i would be 20 but $player.size would be 19. Will the body of the 'for' loop be executed with local.i equal to 20 even though it's not a valid index into the $player array? The script engine would execute the statement "local.i++" and evaluate the condition "local.i <= $player.size" after an iteration of execution of the body of the 'for' loop has completed, i.e. after the 'waitframe', so unless "$player.size" is only evaluated once, the code should still work correctly. Testing with a larger delay in the 'for' loop could verify this is the case.

--[dcoshea](http://gronnevik.se/rjukan/index.php?n=Profiles.Dcoshea)