

yduJ's MOO Lore Pamphlet

-- -- -- lag

yduJ's Theory Of Lag

What causes lag? This is a perennial question on LambdaMOO. There two basic types of lag: Net lag, and CPU lag. Net lag is lag you experience between your local computer (whatever's connected to the keyboard) and the remote computer (LambdaMOO), and can have lots of causes.

Net lag is usually different for different players, depending on their distance from the MOO. Distance isn't necessarily in miles---someone 10 miles from the MOO machine behind a 14.4kbaud SLIP line may have higher lag than someone in New York sitting on a machine with a T1 line. People on other continents than the MOO typically have higher netlag than those in North America, because their bits have to get funneled through some narrow channel under an ocean, competing with the propagation of alt.binaries.pictures.erotica, or have to go up to a satellite and back down again, which takes a while.

LambdaMOO, however, primarily suffers from CPU lag. This means the source of the problem is on the computer that's running the MOO itself, and no amount of closeness to the MOO is going to help fend it off. There are several things which can cause CPU lag, and lambdamoo suffers from them all.

When a program takes more memory than the machine has physical memory, it uses the disk to make memory look bigger, and does what's called "paging" to switch stuff back and forth. This makes stuff way slower. LambdaMOO's physical memory is 256 meg (that's bigger than most people's hard drives), but the program is about 330 meg, so it doesn't fit. Surprisingly, measurements have shown that LambdaMOO's machine isn't paging very much, though it is paging some.

But the main thing that's going on with LambdaMOO is that there is just too much work for the poor little thing to do. There are frequently 250 people logged in at once, and if they're all typing, in order to execute within 1 second lag, it has to do 250 commands per second. This is a lot, because to run the MOO language, the computer has to do a fair amount of work. To make matters worse, there are usually about 350 queued "forked tasks" competing with the 250 logged in users. Adding this all up makes for a very busy computer and very bad lag.

So what's my theory of what makes lag worse? My theory is that it takes a lot more time to create a forked task than you think. (Note: suspending creates a task too, but fork is the real killer.) My theory is that if one is doing a lot of work, and suspending at regular intervals, you should use up your whole quantum of ticks, and call `:suspend_if_needed(60)`, giving the machine a whole minute to ignore you and run someone else's stuff. It needs the break by then!

You should also be very careful when writing loops that you're not computing the same data twice. Store data in properties, and update the properties incrementally, rather than recomputing whole reams of stuff for different people.

If you're doing something that's periodic, emitting a message into a room, for example, make it with a really long period. Nobody cares to see the sky change color every 10 seconds; every 5 minutes is good enough. And make sure there's someone to *see* your text. Don't write tasks that sit in the background and change the color of the sky, even every 5 minutes, if there's nobody there to watch. Use `:enterfunc` to start tasks, and check on `:exitfunc` whether you should be stopping them. If you have something that changes its description based on a time factor, don't make a task to change the description. Instead, when the object is looked at, have it quickly compute

what *would have happened* had there been a task changing it. You may think "but my task only goes once every five minutes!" Yes, that's true. But you're sharing the machine with 8,000 other people, and once every five minutes is way more than your share. Stamp out periodic tasks!

If you're triggering on something like :tell, if you should fork() (a definite no-no!), make sure you are running only one task at a time. Record the currently running task in a property on the object. In the next iteration, check \$code_utils:task_valid and don't fork if it's valid.

-- -- -- permissions

Permissions are problematical in MOO. Generally you don't need to be concerned with permissions unless you are trying allow others to create children of your generic objects. However, most serious MOO programmers end up doing this, so it's worth knowing how the permissions work.

Let's use as example the Generic Wind-up Toy from our previous tutorial, only this time we'll have a different user, Eiram, create the child object, Wind-up Sushi.

```
@create #12221 named Wind-Up Sushi,Sushi
You now have Wind-Up Sushi (aka Sushi) with object number #15666 and
parent Generic Wind-Up Toy (#12221).
```

The default for creating a child object is that the new owner owns not only the object, but all of the properties as well. For example, let's look at .wind_down_msg, on the generic, and on our new child:

```
@show toy.wind_down_msg
#12221.wind_down_msg
Owner:      yduJ (#68)
Permissions: rc
Value:      "finishes."

@show sushi.wind_down_msg
#15666.wind_down_msg
Owner:      Eiram (#40068)
Permissions: rc
Value:      "spasms once and stops spinning."
```

We see that yduJ, who owns the generic, owns the property. But on the child, the owner of that object owns the property.

Here's the tricky part. Only the owner of a property may change that property. This changing owner on a property makes a lot of sense for things like messages. You wouldn't want yduJ to be able to change Eiram's sushi's messages, just because she owns the generic.

However, we'll recall that the windup toy has a property .wound that it sets and decrements as it runs.

The way that verb permissions work is that they run as though the person who wrote the verb was doing all the work (with the author's permissions), not as though the person who issued the verb call was doing the work, or as the person who owns the object that initiated the verb. You could imagine any of these three possibilities. The second one would be more like Unix. In unix, for example, it matters who is typing "cat file", not who wrote the cat program. This allows for trojan horses to be installed in the cat program by people with questionable moral systems. The MOO system is somewhat less flexible, but more secure. Since it matters who wrote the verb, players know that their own properties and objects are safe from would-be crackers. (Unix folks can think of verbs as always running "setuid".)

However, this means that if the .wound property were treated the same as the .wind_down_msg property, then the :wind verb (which was written by yduJ, and thus can only change properties owned by yduJ) wouldn't be able to change the

property on Eiram's sushi. Thus, you'd wind and wind, but it wouldn't ever get wound up. Similarly, the verb that winds down wouldn't be able to decrement the .wound property, so the sushi would run forever! (Really, in both cases it would instead crash with a permissions error, but still, neither fate is acceptable.)

Let's talk about that Permissions: line in the @show above. Property permissions are stored as a string of characters, one or zero each from the set "rwc". R means readable; any verb can just do local = #15666.wind_down_msg and have the value assigned into the string. If the "r" bit is not set, only the owner of the property can look at it. W means writable; any verb can *change* the property. This is extremely rare, because of the possibility for abuse. C means "change in children", and is the one responsible for causing Eiram to have become the owner of the .wind_down_msg when he did @create. If you simply remove the "c" bit from a property, the copies in children don't get changed to the owner of the child object, but stay as the person who defined them on the generic. So, before letting Eiram make a child object, yduJ did:

```
@chmod toy.wound !c
Property permissions set to r.
```

And now when we use @show:

```
@show toy.wound
#12221.wound
Owner:      yduJ (#68)
Permissions: r
Value:      0
```

```
@show sushi.wound
#15666.wound
Owner:      yduJ (#68)
Permissions: r
Value:      0
```

We see that yduJ is the owner of this property. Note that this means Eiram can't change that property, despite owning the object it is on! Read the entry on Security in order to learn how to deal with this problem.

-- -- -- primitives

You can't really program on LambdaMOO just from reading the manual. Reading the manual gives you excellent background on how the underlying server works, but really, in the LambdaCORE, there are some verbs which take the place of (really, enhance the features of) the primitives defined by the server.

```
notify(person,message) => person:tell(message);
move(object,place) => object:moveto(place);

read() => $command_utils:read() or $command_utils:read_lines()
```

In addition to direct replacements defined for all objects, there are an enormous number of utility verbs designed to make programming tasks easier. 'Help Utilities' gives you a starting point to the documentation on the utilities; there are a lot of them!

Proselytizing here a bit:

```
create() => #4455:_create()
recycle() => #4455:_recycle()
```

Read recycling in #23456 for more information.

-- -- -- scheduler

How does the scheduler work?

The idea is there is a round robin queue of player tasks. That means that each player gets no more and no less CPU than any other. It just takes them one at a time, in order.

Within that timeslice, each player gets eir own round robin of forked tasks, including the interactive task that e may be typing at. (E may be logged out, and still have tasks, so it isn't necessary that there be an interactive task in the list.)

I think of it as a two dimensional array, or maybe a list each element of which is a bunch of lists. You go around the big list, taking the top item from the sublists each time. The sublists get added to by fork. You get a new interactive task for each command, so that adds to it, but also subtracts from it. Suspend moves the tasks to the end of your little list. The big list gets added to only when a player not currently in the list has a task started (e.g. with an automatic trigger in a room, or by connecting to the MOO).

So by having a large number of tasks ready to run, a player spams eirself, giving eirself a noticeable lag, which supposedly are not felt by players with only one task. Large numbers of cpu-intensive tasks can spam the entire MOO, though, impacting even those players who are being socially responsible with task use.

There's another wrinkle: If a player does a lot of computation even if e has few forks, that amount of computation is remembered, and when e comes up again for round robin scheduling, e may be skipped based on that large amount of computation, until someone else has built up a similar quantity, or there aren't any other tasks waiting. (I don't understand the exact algorithm here; the short of it is if you see "out of seconds", go get coffee---you're blackballed for a while---maybe several minutes.)

-- -- -- security

Read the entry on Permissions to understand how property permissions work.

In many cases, both the property owner (verb/generic author) and the owner of the child object will need to be able to change a property. MOO does not have "multiple ownership", or "permission groups", or any of those sorts of things.

The typical solution to this problem is for the generic owner to provide a callable verb :set_whatever. Sometimes it is appropriate to also provide a command line verb such as @set-whatever, depending on the application. These setting verbs then generally need to have security in them, to make sure that nobody but the child owner and generic owner actually make any changes. So here's some blathering on what are the available options:

The variable "caller" is like "this" from the calling verb. If you check for caller == this, you can be sure that the calling verb was defined either on this object or on one of its parent hierarchy. For verbs which are pass(led) to, caller == this is true. If there wasn't a caller because this is the first verb executed in the command, then caller is the same as player. You might use caller == this as your security check if you want to make sure that the :set_whatever verbs are only called by the child objects, or your own verbs. caller doesn't let you distinguish between being called by a verb defined on the generic or a verb defined on the child.

caller_perms() gives you the permissions of the calling verb (usually the person who wrote the calling verb). Note that permissions are represented by player objects, so when I say "the verb was called with my permissions" that means caller_perms() == #68. Only a wizard-authored verb can change the permissions that a verb runs with; see documentation on the function set_task_perms() if interested. Basically, you can use caller_perms() to see if someone is trying to spoof your verb. Check caller_perms() == this.owner

to make sure only the proper person is calling the verb. `Caller_perms()` is #-1 if your verb was invoked directly from the command line; sometimes you want to have a verb that's callable by another verb, *or* straight from the command lines, and still be able to do security checks. So, you use:

```
valid(caller_perms()) ? caller_perms() | player
```

where you'd have just used `caller_perms()` in a verb not command line callable. This idiom can also be used in a wizardly `set_task_perms()` call.

Wizards think it's nice if your verbs that do various permissions checkings allow wizards to use your verbs, even if they don't own the object. You can use `$perm_utils:controls(person,object)` instead of using a `== this.owner` type check... Wizards can bash your object anyway, it's just more convenient for them if you don't get in their way. :-) Seriously, on LambdaMOO, a lot of people make errors in their code and their object needs to be rescued; it's nice if you make these janitorial jobs easier.

`callers()` is another way you can do security checking. It tells you a lot of information about exactly how this verb got called. Do `help callers()` for more info... I always forget how it works, so whenever I'm writing a verb that uses `callers()`, I get the help, and then I write the verb with a `player:tell($string_utils:print(callers()))` so I can see just what I'm getting... I recommend this sort of experimental data gathering. Some people hate `callers()`. They think that for every use of `callers()` there is one verb that's doing two jobs, and it should be separated out into two verbs. In most cases they're right. If you're using `callers()` to determine how your verb should behave, rather than as a security thing, think about restructuring your code.

One sweeping method of security is to make your verb !x. This means it can *only* be invoked from the command line. However, this also means it can neither be called from a program nor be pass()ed to, which is sad, because someone might want to customize the behavior of this command line verb. Remember, use args of "this none this" to prevent a verb from being invoked via the command line. A !x "this none this" verb can't be called in any way (and thus is totally pointless!) A +x verb with a normal arglist can be called from anywhere. Read about arglists in the programmer's manual.

You'll note that I've never used "player" except in the `caller_perms()` case where I knew the verb had been invoked from the command line. This is because `player` is basically completely useless, from a security point of view. `Player` is the person who typed the original command which set in motion the thread of control which eventually lead to your verb getting called. This doesn't mean "player called your verb".

I'm going to describe how to spoof `player` in a `:tell`. I wrestled with myself for a long time about whether I would do this in this document. So, here it is. This type of spoofing is considered at least an order of magnitude more rude than the usual sort. You've been warned.

```
.program me:tell
pass(@args); /* do the normal stuff */
if (player == <my enemy>)
    dobj = player;
    #9060:bonk();
endif
.
```

So, everytime my enemy makes a noise in my presence, e bonks eirself with the carrot, because it's really too hard to eschew the use of "player" in VR settings such as the bonker. 99% of the time it's correct; indeed, it's only incorrect in cases where someone is trying some funny business. This demonstration shows how using "player" for security checking is totally useless; the whole point of security checking is to prevent people from doing things they shouldn't, which means that you have to assume these people will be trying tricks such as the above. For example, if the name setting code didn't have `caller/caller_perms()` security checks, I could have called `player:set_name("IamAnIdiot")` instead of merely being a pain with the bonker.

