

LispCourse #14: Multiple Processes and the Process Status Window

Processes and Multiple Processes

Consider how you start up a DEdit:

You type "(DF FOO)" in the Lisp Exec window. This opens a DEdit window separate from the Exec window. You can then carry out your editing actions in this DEdit window.

But note: as long as the DEdit window is active, the Exec window is inactive. If you try to type into the Exec window, the input goes into the DEdit input buffer.

Only after you exit the DEdit, does a value (i.e., the name of the edited object) get returned in the Exec window.

After the value is returned, the Exec window becomes active again and the DEdit window (if its still on the screen) becomes inactive.

Conclusion: the Exec and DEdit use seprate windows but can not run simultaneously. They must "take turns".

Contrast this behavior with that of TEdit:

You type "(TEDIT 'FOO)" in the Lisp Exec window. This opens a TEdit window separate from the Exec window. You can then carry out your editing actions in this TEdit window.

But note: TEdit reads in the file FOO (which make take some time if the file server is slow) and then immediately returns a value (usually a "nonsense" value!).

The Exec window remains active. After the value is returned, you can switch back and forth between the Exec and TEdit windows, simultaneously doing edits in the TEdit window while starting another TEdit, COPYFILE, doing a DEdit, etc. in the Exec window.

Conclusion: the Exec and the TEdit use separate windows and can be run simultaneously.

In technical terms, TEdit runs in a separate process from the Exec while DEdit runs in the same process as the Exec.

A ***process*** in computer jargon is a sequence of activities or tasks that must be carried out one after another.

Interlisp-D supports ***multiple processes*** -- that is you can be carrying out several sequences of activities simultaneously.

Within each of these processes or activity sequences, the activities are carried out in a strict sequential order.

Across processes, several such activity sequences can be going on simultaneously.

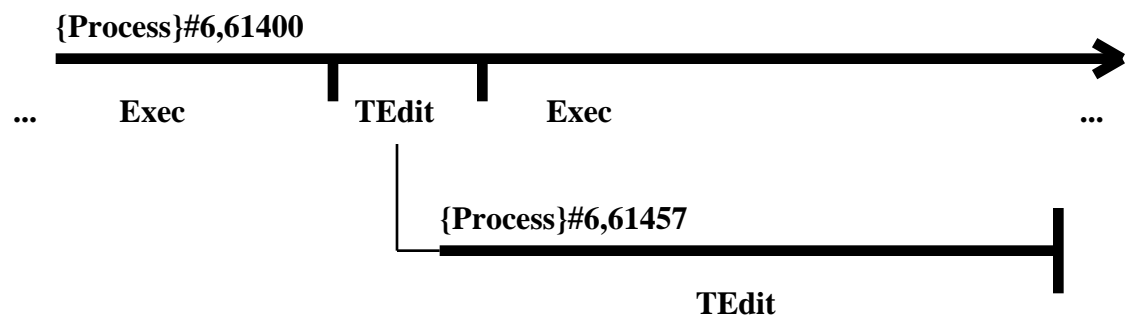
For complicated reasons, DEdit has been designed to run in the same activity sequence (i.e., process) as the Exec. When DEdit starts, the Exec goes into suspension until DEdit is finished.

Diagrammatically:

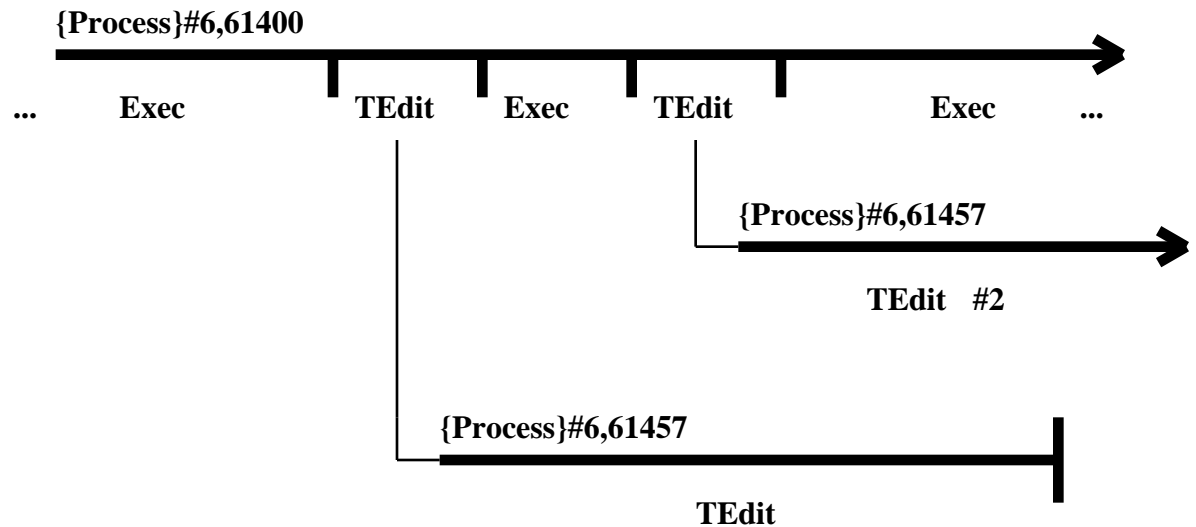


TEdit has been designed to run in a separate activity sequence (i.e., process) than the Exec. When TEdit starts, it starts up a new process. The Exec process then proceeds simultaneously with the Exec process.

Diagrammatically:



Or if you start up two TEdits:



The process of starting a new process from an old one is (sometimes) called *forking a process*.

Processes Scheduling: Time-sharing and all that

Time-sharing

Interlisp-D supports multiple processes in the sense that it (sometimes) appears that you can do multiple activities at once.

But, your D-machine can only do ONE thing at a time.

What actually happens is that Interlisp runs a little bit of one process, then a little bit of another process, then a little bit of a third process, and so on until the first process gets its turn again.

It all works because most of the time in any one process is spent *waiting*: waiting for the user to type the next character, waiting for the file server to respond, waiting for the disk to position to the right page, etc.

While the first process is waiting, the rest of the processes can run. While the rest of the processes are waiting, the first process can run.

This nifty little trick is called *time-sharing*.

The Scheduler

There is a part of Interlisp called the ***scheduler*** that determines which process gets its turn to run next.

The scheduler works as follows:

It runs through each process in the system giving each process control (i.e., permission to run) in turn. When a process gets control, it runs until it gives up that control. When a process gives up control, the control is passed to the next process in line.

This is called a ***cooperative scheduler*** because each process has to take the effort to give up control when it is waiting for something or when it feels like it has hogged-up too much time. The act of giving up control is called ***blocking***.

There are systems with ***preemptive schedulers*** that force processes to give up control at regular intervals, so that every process gets a little bit each and every second. Processes need not worry about blocking.

One feature of a cooperative scheduler is that some processes don't play by the rules: they hog as much time as they like without blocking. When this happens, other processes don't get their turns to run.

For example, try starting up a TEdit. Then go back to the Exec window and start a LISTFILES on a fairly large Lisp file. Go back a starting editing in TEdit. At some point TEdit will just stop working for about 2 to 3 minutes. This is because LISTFILES goes into a part where it doesn't block for along period of time - it gets control and just hangs on to it until its work is finished, not allowing the TEdit and the other processes to have their turn to run.

E.F.S. -- An Experiment:

Try the following:

Start a TEdit.

Back in the Exec window, type "(FOR I FROM 1 do
(PROMPTPRINT I))"

Try to use TEdit.

To kill the FOR: point in the Exec window, then type Ctrl-E.

Now try the following: (Note: (BLOCK) causes a process to block!!)

Start a TEdit.

Back in the Exec window, type "(FOR I FROM 1 do
(PROMPTPRINT I) (BLOCK))"

Try to use TEdit.

To kill the FOR: point in the Exec window, then type Ctrl-E.

Cautionary Note: control hogging processes are just one reason for delays in Interlisp. Slow file servers are another. Don't automatically assume that if a process is slow, there is another process hogging the run time!

Job Control: Starting, Killing, Restarting, Suspending, & Waking Processes

You can do a variety of things to help you manage the processes running in your environment. The most important of these are the following:

Starting a process (& Process Names)

Most processes are started by programs forking a process to carry out a task.

Examples:

1. The function TEDIT opens a TEdit window and starts a TEdit process in that window.
2. The function call (LAFITE 'ON) starts a process that runs in the Lafite Status window and checks your mail every now and then.
3. The function CROCK starts a process that updates a clock on the screen once a minute.

You can also start a process by yourself from the Exec. The function ADD.PROCESS forks a new process. ADD.PROCESS takes as its first argument the form to be evaluated in the new process.

Example:

```
10_ ( ADD.PROCESS '(COPYFILE 'A 'B))
      {PROCESS}#3,65600
```

11_

Normally, COPYFILE runs as part of the Exec process, i.e., it doesn't automatically fork a process to do its work. Using ADD.PROCESS will start a new process and do a COPYFILE within that process. The COPYFILE will just run until its done. While its running, you can progress with whatever you want in the Exec, including starting another COPYFILE.

Process Names: Every process has a name. The name is usually the CAR of the form in the call to ADD.PROCESS. Thus the process created in the example above would have the name *COPYFILE*. If there is already a process with that name, a number is tacked onto the end of the name. For example, *COPYFILE#2*.

You can also give the process a name using (ADD.PROCESS *Form* 'NAME *ProcessName*). For example, (ADD.PROCESS '(COPYFILE 'A 'B) 'NAME *'AtoB*) would start a process named "AtoB".

Killing a process

Most processes just evaluate a single function call as in the COPYFILE example above. These processes normally just run until the evaluation is completed. Then they just disappear.

Sometimes a process takes to long, gets hung up, or goes into a break. In this case, you may want to **kill** the process, i.e., stop the evaluation in progress and force the process to disappear.

Example:

You start a LISTFILES to print a file. This starts a process to send the files to the printer. But the printer is down. The process will just sit there printing in the PromptWindow *Quake not responding*. This can drive you crazy. So you kill the process started by LISTFILES and redo the LISTFILES when you know the printer is up again.

Killing is usually done using the Process Status window (described below). But can also be done by calling DEL.PROCESS from the Exec with the process name as an argument. E.g., (DEL.PROCESS 'COPYFILE)

Restarting a process

Some processes can be restarted from the beginning at any time before they finish. Restarting a process is just like killing it and then redoing the `ADD.PROCESS` that started the process to begin with.

Example:

You start a `HARDCOPY` in your TEdit window that never seems to finish because the file servers are slow and TEdit can't find the right fonts or some such nonsense. You just want to kill the `HARDCOPY` command and get on with your editing. Solution: Restart the TEdit process.

Note that you could kill the TEdit process. But then you'd have to go through the trouble of restarting TEdit in order to get back to editing. Moreover, if you hadn't put your edits, restarting TEdit may lose your edits. Therefore, restarting is much safer.

Restarting is usually done using the Process Status window (described below). But can also be done by calling `RESTART.PROCESS` from the Exec with the process name as an argument. E.g., *(RESTART.PROCESS 'TEdit#4)*

Suspending a process

Suspending a process stops a running process, but doesn't make it disappear. It stays around and inactive until it is either awoken (i.e., unsuspended) or killed.

Suspending a process is usually done by programs. But it can be useful in other ways.

Example:

You try to print something out with a LISTFILES. But the printer is down and the LISTFILES keeps printing *Quake not responding* in the Prompt window. You can stop this annoying printout by suspending the LISTFILES process until you know the printer is back up.

Suspending is usually done using the Process Status window (described below). But can also be done by calling SUSPEND.PROCESS from the Exec with the process name as an argument. E.g., (*SUSPEND.PROCESS* '*FILELISTING*').

Waking a process

A suspended process can be started at the exact point at which it was suspended by **waking** it.

Note that waking a process is a little like restarting a process. BUT a suspended process must be *woken*, it cannot be *restarted*. Restarting a suspended process, restarts the process from the beginning but does not unsuspend it. Therefore, doing a restart followed by a wake is basically the same as doing a wake followed by a restart. The wake unsuspends, the restart makes things start from the beginning again.

Example:

The printer is now fixed and you want your LISTFILES to pick up where it left off. You just wake the process you suspended earlier.

Waking a process requires that you return a value from the suspension. For processes suspended by the user, this value is usually NIL.

Waking is usually done using the Process Status window (described below). But can also be done by calling `WAKE.PROCESS` from the Exec with the process name as an argument. E.g., (`WAKE.PROCESS 'FILELISTING`).

Processes and Breaks

Every process has its own, separate stack.

When an error occurs and a break is entered, the break occurs *within the process* causing the error. The Break Exec that is started is run in that process and the stack that you can examine using the Break Exec is the stack of that process.

Thus you can have as many Break Execs running as there are processes to break.

Example:

Your TEdit breaks due to some wierd error. This causes a break in the TEdit process, but will not effect any other running process. For example, if you have a COPYFILE going at the same time. The COPYFILE will progress as usual since the break has no effect on its process. If the COPYFILE then has an error, it will enter its own, independent Break window.

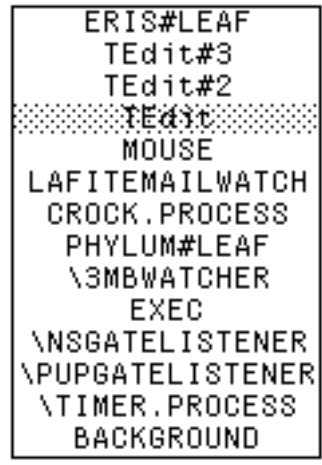
You can force a process to enter a break at any time from the Process Status window (described below). You cannot do this (easily) from the Exec.

You can also just observe the stack of a running process from the Process Status window. You cannot do this (easily) from the Exec.

The Standard Processes

Most of the time there will be 5 to 10 processes running in your system. Most of these processes are special system processes that handle things like the Ethernet, the mouse, the Lisp exec, file servers and so on.

Below is a list of the processes currently running on my system:



```
ERIS#LEAF
TEdit#3
TEdit#2
TEdit
MOUSE
LAFITEMAILWATCH
CROCK.PROCESS
PHYLUM#LEAF
\3MBWATCHER
EXEC
\NSGATELISTENER
\PUPGATELISTENER
\TIMER.PROCESS
BACKGROUND
```

The processes that I was directly responsible for were the three TEdit processes, the CROCK.PROCESS that is running my clock, and the LAFITEMAILWATCH process that is checking my mail every now and again.

The rest are all system processes carrying out some sort of system monitoring task. In general, I don't worry much about these system processes.

The TTY Process

There may be many processes running that require input from a keyboard. However, there is only one keyboard. In order to determine where the type-in from that keyboard goes, there is a notion called the *TTY process*.

The TTY process is a special designation given to only one process at a time. The process with this designation is the one that receives the keyboard input. The rest of the processes requiring keyboard input have to wait until they become the TTY process before getting any type-in.

For example, when you have multiple TEdits running, all but the one you are currently typing into are just sitting there waiting to become the TTY process.

If the process that is the TTY process has a window (as is almost always the case), then that window contains a blinking caret indicating where the typed input will appear. When the process loses the TTY process designation, then the caret may remain, but it no longer blinks. Thus, the blinking caret almost always designates the window whose process is the TTY process.

If a process has a window, you can usually make that process be the TTY process by clicking a left or middle mouse button in its window. Basically, you point into the window you want to type into, then you type into that window.

You can also move the TTY process around using the Process Status window as described below.

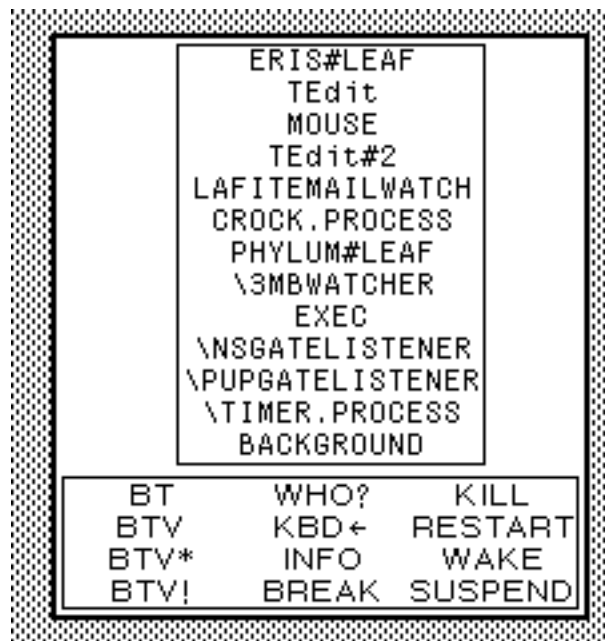
Finally, many programs move the TTY process around. For example, when you start a new TEdit, it grabs the TTY process for itself. When a TEdit quits, it gives the TTY process to the Lisp Exec.

The Process Status Window

The Process Status window provides an easy-to-use mouse/window-based interface to process management.

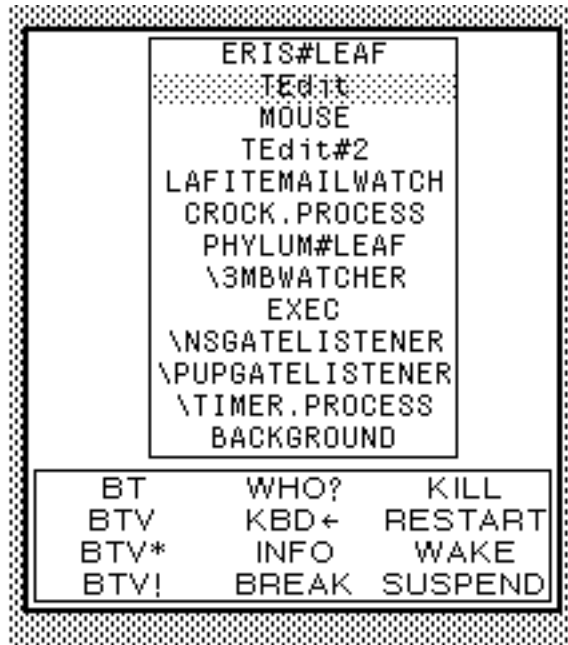
To open a PSW, select the PSW command from the background menu OR use the function call (PROCESS.STATUS.WINDOW) in the exec. Either way, you'll be asked to place the window on the screen.

The window will look like the following:



The upper box in this window contains a list of the names of all the processes currently in the system, both running and suspended.

You can *select* any one of these processes by clicking over the process name. This will shade this process name.



The lower box is a menu of commands that you can apply to the selected process.

The commands are: (starting from the right column)

KILL ž kills the selected process

RESTART ž restarts the selected process

WAKE ž wakes the selected process if its suspended. You will be given a menu of values that can be returned. Choosing NIL from this menu will generally work.

SUSPEND ž suspends the selected process

WHO? ž moves the selection to the process that has the TTY process.

This is very handy for telling multiple processes running the same program apart. For example, if there are 4 TEdits running, there will be

processes named TEdit, TEdit#2, TEdit#3, and TEdit#4. To find out which TEdit you want to operate on you can bug inside that TEdit's window. This will pass the TTY process to that TEdit. Then choose the WHO? command in the PSW. This will select the TEdit corresponding to the window you clicked in.

KBD_ ž gives the TTY process to the selected process.

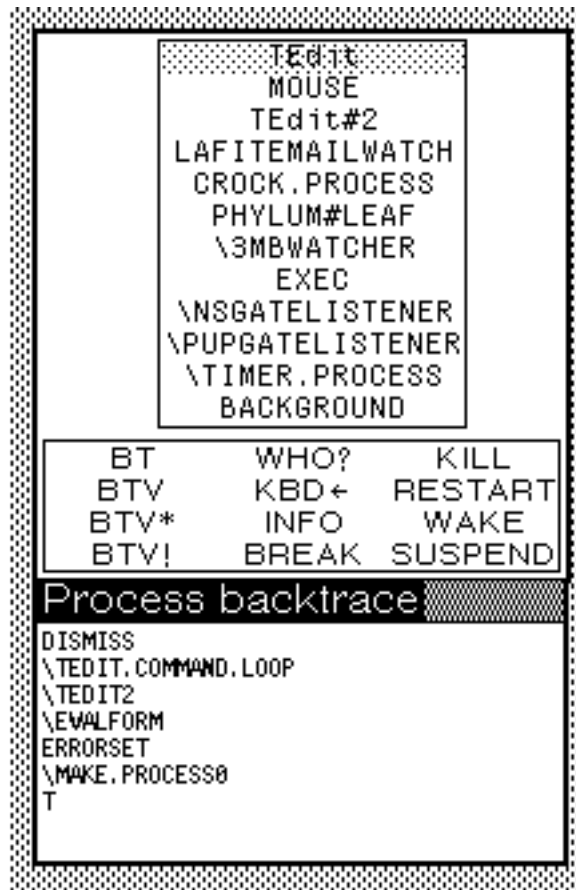
Example: If TEdit has the TTY process, then selecting the EXEC process and using the KBD_ command will move the TTY Process (i.e., the blinking cursor) to the Exec window.

INFO ž returns any information the selected process wants to give. I have never seen a process that wanted to give any information, but ...

BREAK ž forces the selected process into a break, thus opening a break window, starting a break exec, etc.

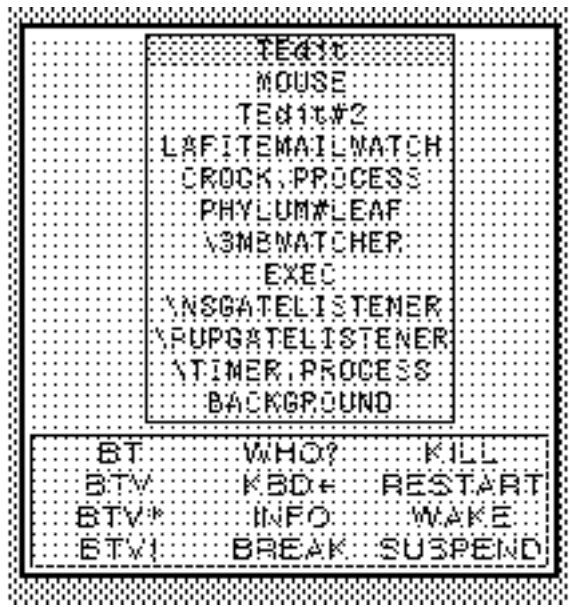
This is handy to examine the state of a computation that has run amok:
Break the process and then muck around the stack using the Break Exec.

BT (BTV, ...) ž prints a backtrace of the stack of the selected process in an attached window just above (below) the PSW. BT prints standard backtrace with just the names of the functions called. BTV and the rest print backtraces with some more information about each frame.



This is a handy way to find out if a process is progressing or if its stuck at some point: click BT many times in succession. If the stack is constantly changing, then things are progressing. If the stack is unchanging with `AWAIT.EVENT` or some such function at the top, then the process is probably stuck waiting for a file server to respond or some other external event to occur.

Whenever a process starts or finishes, the PSW gets shaded over. This indicates that the information is no longer current.



To update the information, just click the left or middle mouse buttons anywhere in the PSW. This will redisplay the PSW with the current information.

Multiple Exec Processes

In default mode, Interlisp has only one Exec process. There are, however, a number of Lisp Users packages that allow for multiple Exec windows and multiple Exec processes to run simultaneously.

One such package is called EXEC.

Load {eris}<LispUsers>Exec.DCOM. This will add a "EXEC" choice to the background menu.

Choosing EXEC from the background menu will prompt you for a region and then open a new Exec window in that region running an independent Exec process. You can use this new Exec simultaneously with and in exactly the same manner as the original Exec process.

For example, you can run a long COPYFILE in the original Exec window. While its running, you can carry out your normal tasks (e.g., starting TEdits, doing DIRs, etc.) using the second Exec.

You can open as many Exec windows and start as many Exec processes as you like.

To terminate a given Exec, evaluate the function call (*EXEC.QUIT*) in its Exec window.

Note: There is one way in which the multiple Execs are not independent: they share a common history list. Each event in every Exec updates the event number in all running Execs. From any Exec window you can redo or undo events originally carried out in any other Exec window.

This shared history list can sometimes be a great help. But it can also be problematic if you forget the shared nature of the list. In particular, nevent numbers printed in the Exec windows are not always up-to-date because they may change constantly due to events in other Exec windows. You very often tend to undo or redo the wrong event number because you rely on incorrect information printed in the Exec window and don't check the history list for the correct event number.

The Mouse Process and (SPAWN.MOUSE)

The process that sits in the background and watches your mouse movements and mouse button clicks is called the *MOUSE process*.

When you click inside a menu or a window, the mouse process is responsible for getting a decision about what to do from the menu or window, and then evaluating the correct functions necessary to do it. Thus much work in Interlisp gets done within the mouse process as button presses lead to the evaluation of various functions.

For example, when you choose TEDIT from the background menu, the MOUSE process evaluates the function call (*TEDIT*) just as if you had typed it into an Exec window.

In general, you shouldn't have to worry too much about the mouse process unless you are programming.

Sometimes, however, something goes wrong and the mouse process gets stuck. For example, a break can occur in a function being evaluated in the mouse process. Or a function can go into an infinite loop when being evaluated in the mouse process.

When this happens, you can't use your mouse for anything. You can often, however, still type in to an Exec or a Break Exec.

When this happens, the function call (SPAWN.MOUSE) comes in very handy. SPAWN.MOUSE will start up a new mouse process, replacing the old.

The old function keeps going and finishes whatever it was doing when SPAWN.MOUSE was called, except that it is renamed OLDMOUSE.

The new mouse process takes over control of the mouse and hence all of the mouse functions are restored.

Example:

You choose an item from a Lafite Browser menu. This immediately causes a break. However, in the Break window you discover that the mouse doesn't work at all. In fact, no where on the screen does the mouse have any effect.

If the flashing cursor is in the Break window or in a Exec window, then you could type (*SPAWN.MOUSE*). This would free up the mouse by starting a new MOUSE process. The break in the Lafite menu command would remain unaffected, although its process would be renamed from MOUSE to OLDMOUSE.

References

Processes are documented in Section 18.20 of the IRM.

Subsections 18.20.1, 18.20.2, & 18.20.5 are generally relevant for non-programming users.

Subsection 18.20.6 covers the TTY process.

Subsection 18.20.7 covers the MOUSE process and SPAWN.MOUSE.

Subsection 18.20.8 covers the Process Status window.