Adventures in Multiprocessing:
Controlling Multiple Windows within a Single Application

Shelly Evans
UNISYS Corporation
Systems Technology Operations
San Diego, CA

Abstract

This paper presents the multiprocessing architecture used to control the multiple window environment of a LISP–based Computer Aided Design tool. The multiprocessing control is implemented using the Lucid Common LISP Multitasking Facility. This paper will discuss the system motivation and goals, system architecture, and system development, including problems encountered, performance enhancements made, and areas for further improvement. This illustrates how using high–level, explicit LISP multitasking control within the LISP development environment can simplify the programmer's task.

System Description

The GLL (Geometric Layout Language) System is a CAD tool used to generate physical layout of integrated circuit designs. The layout is described using the Geometric Layout Language, and the GLL System is used to display the generated layout, generate GLL code, check design rules, create output files for verification and fabrication, etc. The GLL System is implemented (mostly) using Common Lisp. The Geometric Layout Language and the GLL System Command Language are also LISP–based. The GLL System is an internal tool presently at use in UNISYS. It has approximately twenty active users, typically layout designers and design engineers.

The GLL System has two input areas: a Command Menu and a LISP Input/Transcript pad (see Fig. 1). Most commands can be entered either through the Menu or by using the appropriate GLL Command Language command. In addition, large or small chunks of GLL code are often input through the LISP pad in order to incrementally update the generator.

In its earliest incarnations, the GLL System was implemented on Apollo workstations and the Menu interface used Apollo native graphics calls. Users had to indicate with a command when the input mode was to be switched from the Command Menu to the LISP input pad, and vice–versa. In order to support the Sun and HP workstation platforms, the Menu was rewritten using (foreign function interfaces to) Motif/Xt/X libraries. At this time, the top level looping strategy was also reconsidered.

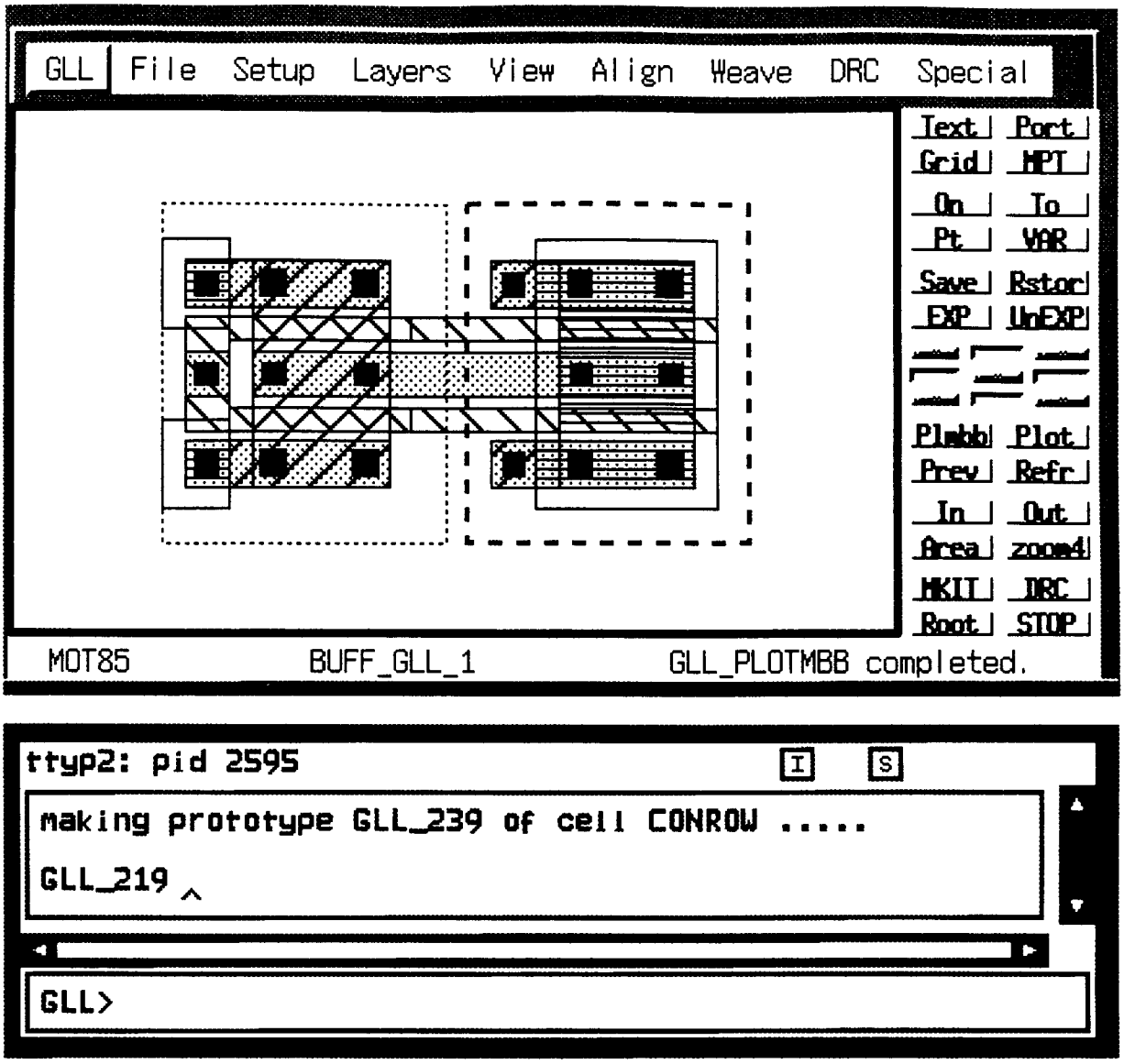

Fig. 1: The GLL System has two Input Areas

The main objective was to allow the user to freely interleave commands in the Menu and LISP input pad areas, without having to indicate the area of input to the system. Other major goals were to maintain good response time (< 10% downgrade from the previous system), maintain data integrity, be transparent to the user, and be error free. A further goal was extensibility, as in the future an integrated editor will be added to the system.

Multiprocessing Architecture

We chose to use a multiprocessing architecture to control the multiple window environment within the GLL system (see Fig. 2). In this model, one process controls each window area, and one process is dedicated to data processing.

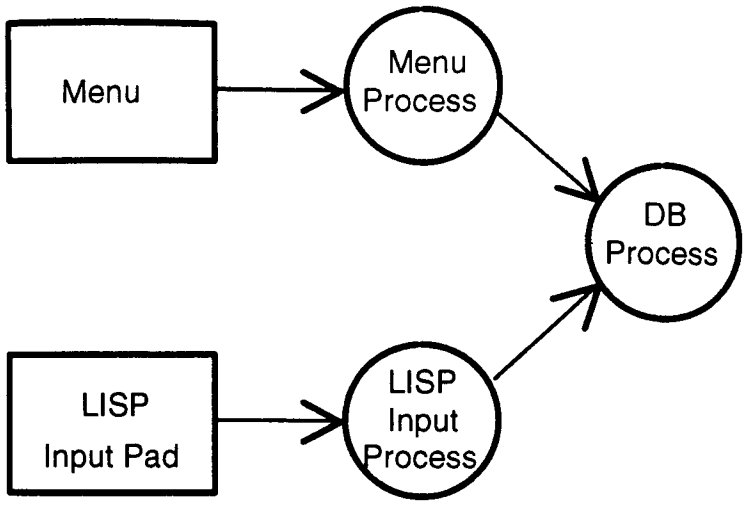

Fig. 2: Processes and Input Paths

This architecture was chosen because it is a natural partition of the multiple window problem, and we believed it could satisfy all of our goals as well as provide extensibility. This model allows the system to receive in–order input, to queue up work, and also to interrupt processing when needed. For example, a user may interrupt a long graphics plot by pressing a STOP button.

Other approaches were considered. A CLIM implementation on the Apollo platform was not available to us. This problem aside, a Command Menu/Interactor application frame could be implemented in CLIM. The only functionality missing would be interruptability (which could be accomplished with an OS interrupt), and possibly extensibility to use an integrated editor. We intend to revisit this option once the Apollo platform is no longer supported.

Implementation Options

Having chosen the multiprocessing architecture model, there were several options for implementation. We chose to use the Lucid Common LISP Multitasking Facility because it provides good high–level support for multiprocessing within a single LISP environment.[1] The Multitasking Facility provides operators for creating, killing, suspending, and interrupting processes, as well as inspecting process state, etc. It has a built–in scheduler which cycles between processes using a prioritized time–slice algorithm. The scheduler automatically saves and restores each process' state when stopping and restarting it. The Multitasking Facility also provides locking operators. The Multitasking Facility offers a lot of control over processes at a high level, which made the implementation of the multiprocessing control both easy and natural.

1. To the operating system, the LISP environment is a single process. The 'processes' referred to herein are wholly created and managed by the Multitasking Facility, and unknown to the operating system.

Other options considered were implementing our own multiprocessing system in LISP, from a simple model such as input polling to a more full–fledged implementation such as the Multitasking Facility. We considered the simple models too restrictive and the more advanced ones too time–consuming to implement, especially as the needed functionality was already available. We also could have implemented multiprocessing control using C operators such as fork and exec. This approach would have been unnatural within the context of the LISP system—for example, it would not allow sharing of the address space, meaning all communication would have to be done using message–passing. Also, the C operators lack the clarity and high–level control features of the Multitasking Facility operators.

System Architecture

We chose to implement three processes within the GLL system:

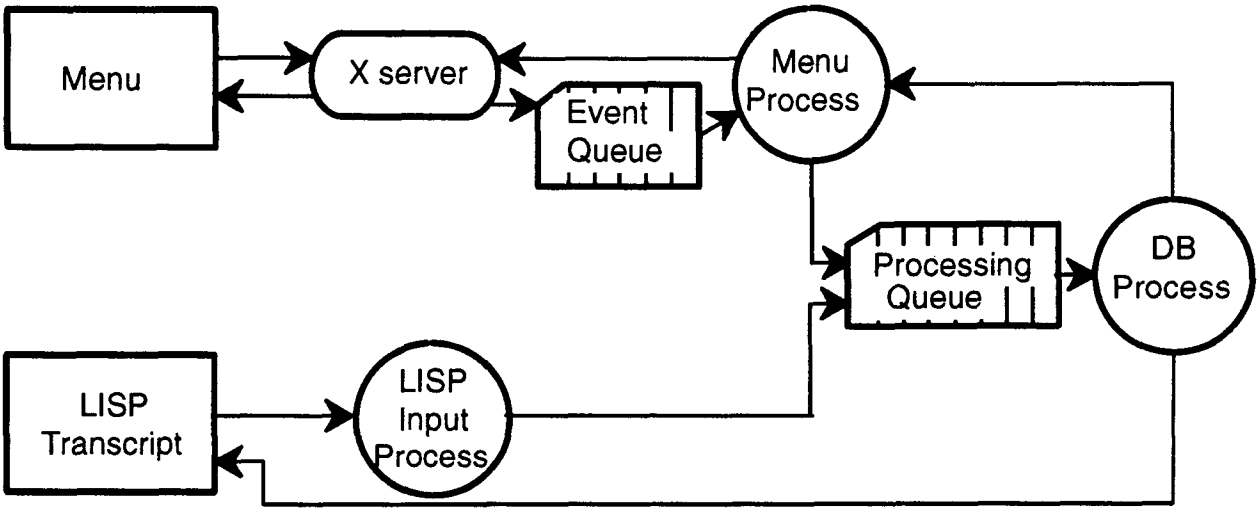

Fig. 3: GLL Multiprocessing Architecture

The Database (DB) Process does all the GLL database Processing. It processes forms that have been inserted by the other two processes into the Processing Queue, in a first in, first out manner.

The Lisp Input (LI) Process listens to the LISP input pad and places the forms that it reads into the Processing Queue.

The Menu Process listens to the Menu via the X server Event Queue, and places forms that need to be processed into the GLL Processing Queue. It also handles sending any results back to the Menu via X server calls.[2]

2. The X server is an application which runs on the local workstation and acts as an intermediary between the user programs (e.g. the GLL client) and the resources of the workstation, in order to perform low–level I/O.

Architecture Description

The multiprocessing architecture is now described in more detail.

The Processing Queue

The Processing Queue is a LISP structure which contains a list of forms to be processed (the queue), and a count of the number of forms currently in the queue. The queue is accessed in a first in, first out manner.

The Processing Queue structure also contains a lock. The lock is a named slot which is nil, if no–one currently holds the lock, or else equal to the process id of the process which currently holds the lock. To ensure integrity, any process which wants to access or change the Processing Queue must first acquire the lock, using the Multitasking Facility's locking operators. These operators ensure that only one process may hold the lock at any time. All others are locked out.

The Database (DB) Process

All accesses to the GLL database must be performed through the DB Process.

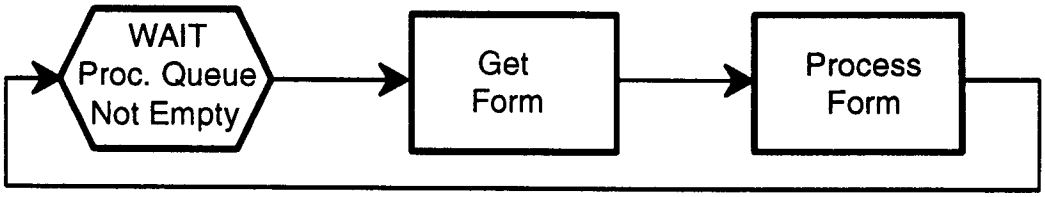

Fig. 4: DB Process Top Loop

The DB Process' top loop executes a process–wait on the condition that the Processing Queue's count is not 0. When this condition is met, the DB Process tries to acquire the Processing Queue lock, by executing the process–lock operator, such that the DB Process will wait if another process holds the lock. When the lock is acquired, the DB Process pops the top form from the Queue, deactivates the LISP Input Process, and releases the lock. The LISP Input Process is deactivated to avoid competition in case the DB Process needs to request input from the LISP input pad in the course of processing the form. The LISP Input Process is reactivated after the form has finished processing. At this time, the DB Process also acquires the Queue lock again to decrement the count. The DB Process then returns to the top of the loop, and executes a process–wait on the Processing Queue's count not equal to 0.

The LISP Input (LI) Process

The LI Process is responsible for listening to the LISP input pad, and placing forms that it reads into the Processing Queue.

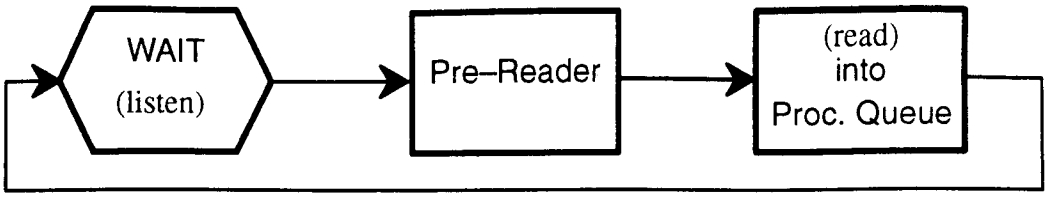

Fig. 5: LI Process Top Loop

The LI Process' top loop does a process–wait on the condition that (listen) returns true. The LI Process must then acquire the Processing Queue lock. Once the lock is acquired, the input must again be checked through a (listen). This is because another process may have consumed the input while the LI Process was waiting to acquire the lock. If this (listen) returns nil, the LI Process releases the lock and waits again. Note that the DB Process and the LI Process share the LISP input pad, and the DB Process must be able to preempt the LI Process in order to request and receive input when necessary. (See "Improvement Areas", below, for more on this subject). Note also that when a break occurs the Condition Handler deactivates the LI Process for the same reason.

The LI Process now has the lock and something to read. It now enters a Pre–Reader loop. The Pre–Reader throws away garbage input, such as spaces, which may precede the form to be read. This is because if the LI Process performs a (read), and the input is all garbage, the (read) will wait until a valid form is entered. Since the LI Process holds the lock, this effectively hangs the system until a form is entered in the LISP pad. The Pre–Reader seeks to avoid this by getting and throwing away characters in the set ("space", "newline", "backspace", ")", etc.). When a ")" is encountered, the Pre–Reader prints the same message the Lucid Interpreter prints: "Ignoring an unmatched right parenthesis", and when a ";" is encountered, it throws away all characters up to and including EOLN.

When the Pre–Reader is finished, either a good character has been encountered or nothing is left, so another (listen) is done to ensure input is available. If so, then a (read) is performed, and the form is placed in the Processing Queue. The Processing Queue lock is released. Then the LI Process loops back to wait on (listen) again.

It was found undesirable to have the LI Process queue up input beyond one form. When the user types ahead, he may be entering input intended as a response to some command. For example, he may type in 'Y' in anticipation of a question from a command he has entered. For this reason, the LI Process will only accept input when the Processing Queue length is 0, so it will wait on both the (listen) and on the Processing Queue length = 0 at the top of the loop. This tends to give preference to input coming from the Menu, but it does not cause a problem in practice, as users do not tend to interleave queued input. (But see "LI Process Improvement Areas", below).

The Menu Process

The Menu Process is responsible for all communication with the Motif Menu. It was found necessary to have a single Multitasking Facility 'process' handle all communication, both input and output, with the X server. This avoids situations in which two LISP processes are trying to send messages to the X server at the same time, which may generate out–of–order requests to the X server, causing hard errors.

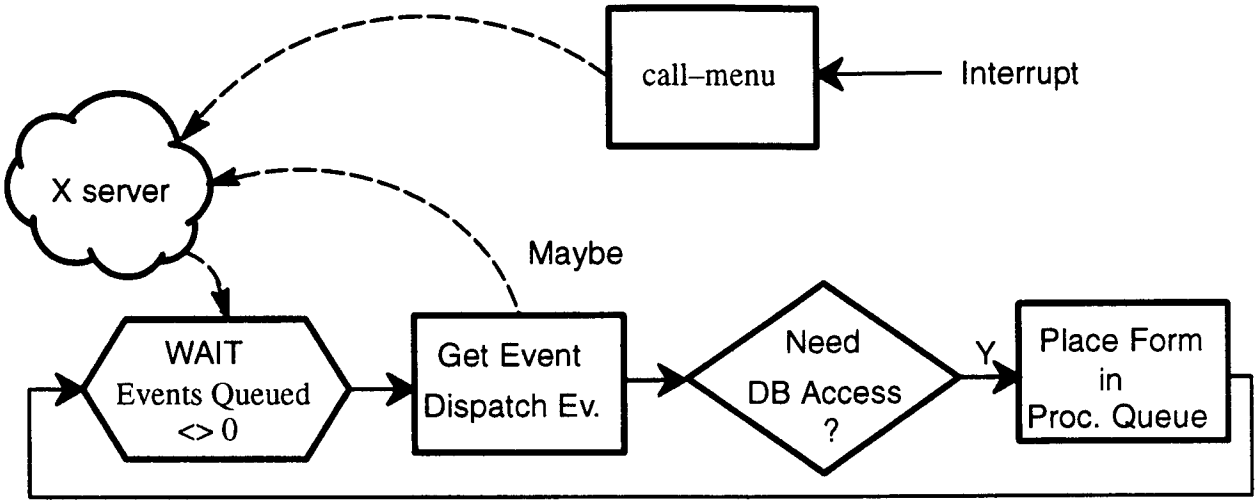

Fig. 6: The Menu Process' top loop and call–menu Interrupt

The Menu Process' top loop waits for events which the X server has placed in the X Event Queue by executing a process–wait on Events Queued <> 0. When an event is there, the Menu Process executes XtAppNextEvent, to get the next event, and then XtDispatchEvent to execute the code associated with the event.

Note: The Xt library has a top–level loop: XtAppMainLoop, which waits until there is something in the Event Queue, then gets it and dispatches it, performing the same operations as described above. We do not use this top loop because it was slowing execution time by 50% within the Multitasking Facility context. The reason is that even when the Menu Process was simply waiting for input using XtAppMainLoop it was taking its full quantum of time. For example, when the DB Process was executing a form it would get a quantum of time and then the Menu Process would get a quantum of time, which was typically spent waiting. Therefore we broke up the loop and performed our own wait using the Multitasking Facility's process–wait operator.

Some X events require local processing which does not need to go through the Processing Queue. An example of this is an expose event which needs only to refresh the Menu area.

However, most events do require Database access and must go through the Processing Queue. The Menu Process then places such forms to be executed into the Processing Queue.

When graphical output or Menu updates are needed in the course of DB processing, the Menu Process must send the request to the X server, as noted above. In these cases the DB Process invokes a function, call–menu, which interrupts the Menu Process, giving it the request to perform. The DB Process then waits until the Menu Process has completed its task, as there may be subsequent processing to do. It does this by waiting on the values returned by the requested function when the Menu Process has finished processing:

```
(defun call–menu (fcn &optional args)
 ...
 (let (hold–values)
   (lcl:interrupt–process *menu–process*
      #'(lambda nil (lcl:with–interruptions–allowed      ;; Allows pr. to be interrupted again
                        (setq hold–values (multiple–value–list (apply fcn args))))))
   (lcl:process–wait "waiting for interrupt to complete"
                     #'(lambda nil hold–values))
  (values–list hold–values)
```

After finishing the interrupt processing, the Menu Process automatically returns to what it was doing when interrupted. It should be noted that the Menu Process requests are made on a high level, for example by requesting a full screen plot, rather than making a request for each low–level operator which draws a line or a box, because of the overhead involved.

Improvement Ideas

As GLL development continues, we would naturally like to improve upon the Multiple Window Access implementation.

CLIM

As stated above, when the Apollo workstation platform is no longer supported, we will look more closely at implementing this kind of window manager using CLIM, or another high–level Graphical User Interface.

LI Process

A lot of effort goes into the coordination between the LI Process and the DB Process regarding the "sharing" of the LISP input/transcript pad. The LI Process could perform all of the input processing for the system and just place whatever it reads into the Processing Queue. This would require the DB Process to search the Processing Queue for input when needed and possibly to wait on this Queue. The DB Process would also have to differentiate between input from the Menu and the Lisp Input processes in this search.

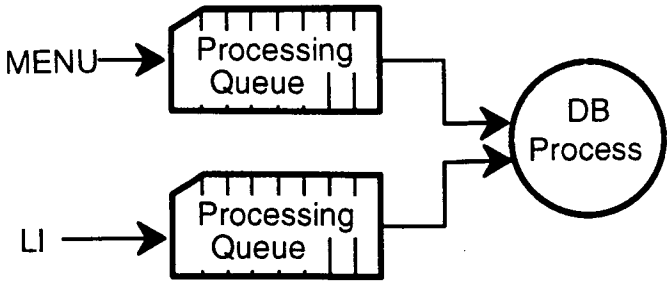

Fig. 7: Two Queues feeding the DB Process

For this reason, it could be beneficial to have two queues feeding the DB Process—one for input from the Menu and one for input from the LISP Input Pad. When the DB Process required input it would access (or wait on) the LI Processing Queue. Queue requests could be time–stamped so that the older of the requests would be picked for DB processing when needed and ready.

Menu Process

Whenever Database information is needed by the Menu Process, for example to display certain sub–menus, a request is placed in the Processing Queue. The results will be sent back to the Menu Process by the DB Process in the form of an update to the Menu. A different approach is to have Menu State Objects which represent information displayed by the Menu. These objects would be controlled by a lock. They would be updated by the Database Process, and could be accessed by either the Database Process or the Menu Process.

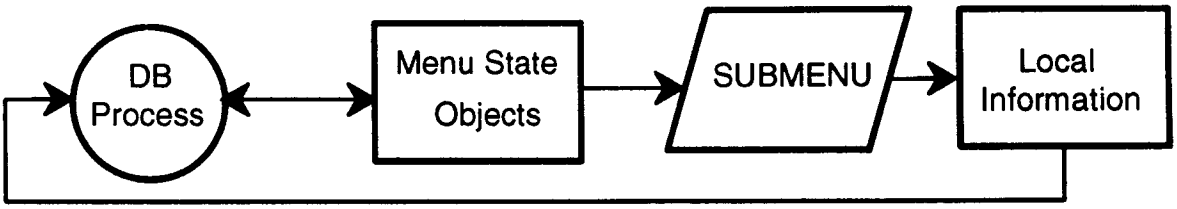

Fig. 8: Menu Process Improvement Ideas

Also, whenever the user enters information into the Menu, the Database and Menu update actions are performed immediately. Since the information is processed as soon as it is entered, none of the submenus have a "CANCEL" button to clear the state the user has entered. Each submenu should have a method of storing information locally and then updating when an "ACCEPT" button is pushed or clearing when a "CANCEL" button is pushed.

Conclusions

This system has been available to and used by our users for some time now, and the goals for this system were substantially met. Users can freely interleave commands between the two windows, and are blissfully unaware of the multiprocessing loops which control the system. Data integrity has been maintained. Response time has also been maintained, with the addition of a 5% overhead due to the multiprocessing control. Because of the multiprocessing control, the GLL system is now always performing some background processing on the workstation, even when otherwise idle, due to the wait loops which are checked every scheduling quantum. This processing typically takes between 1–2% of the available processing bandwidth on the workstation, which does not cause a problem. The final objective of having an error–free system has not yet been met. X crashes still occur for which the cause is not immediately identifiable. These problems will continue to be addressed.

In conclusion, we found the Lucid Multitasking Facility to be a good multiprocessing implementation which met all of our needs. Implementing the multiprocessing control within the LISP environment was fun and educational.

Acknowledgements

I would like to acknowledge my supervisor, Eli Schlachet, for the many ideas he contributed to the paper and to the design and implementation of this system. Also my co–workers, Kathy Herring and Diane Melendrez, who also worked on this project and contributed many valuable insights and ideas.

References

1. Lucid, Inc., Lucid Common LISP, Version 4.0, Advanced User's Guide, 1992

2. Flanagan, David, Ed., X Toolkit Intrinsics Reference Manual, O'Reilly and Associates, Inc., 1992

3. Steele, Guy L. Jr., Common LISP, The Language, 2nd Edition, Digital Press, 1990

4. Stevens, W. Richard, Advanced Programming in the UNIX Environment, Addison–Wesley Publishing Company, Inc., 1992