# 23. PROCESSES

The Interlisp-D Process mechanism provides an environment in which multiple Lisp processes can run in parallel. Each executes in its own stack space, but all share a global address space. The current process implementation is cooperative; i.e., process switches happen voluntarily, either when the process in control has nothing to do or when it is in a convenient place to pause. There is no preemption or guaranteed service, so you cannot run something demanding (e.g., Chat) at the same time as something that runs for long periods without yielding control. Keyboard input and network operations block with great frequency, so processes currently work best for highly interactive tasks (editing, making remote files).

In Interlisp-D, the process mechanism is already turned on, and is expected to stay on during normal operations, as some system facilities (in particular, most network operations) require it. However, under exceptional conditions, the following function can be used to turn the world off and on:

(PROCESSWORLD *FLG*) [Function]

Starts up the process world, or if *FLG* = OFF, kills all processes and turns it off. Normally does not return. The environment starts out with two processes: a top-level EVALQT (the initial "tty" process) and the "background" process, which runs the window mouse handler and other system background tasks.

PROCESSWORLD is intended to be called at the top level of Interlisp, not from within a program. It does not toggle some sort of switch; rather, it constructs some new processes in a new part of the stack, leaving any callers of PROCESSWORLD in a now inaccessible part of the stack. Calling (PROCESSWORLD 'OFF) is the only way the call to PROCESSWORLD ever returns.

(HARDRESET) [Function]

Resets the whole world, and rebuilds the stack from scratch. This is "harder" than doing RESET to every process, because it also resets system internal processes (such as the keyboard handler).

HARDRESET automatically turns the process world on (or resets it if it was on), unless the variable AUTOPROCESSFLG is NIL.

## Creating and Destroying Processes

(ADD.PROCESS *FORM PROP$_1$ VALUE$_1$ ... PROP$_N$ VALUE$_N$*) [NoSpread Function]

Creates a new process evaluating *FORM*, and returns its process handle. The process's stack environment is the top level, i.e., the new process does not have access to the environment in which ADD.PROCESS was called; all such information must be passed as arguments in *FORM*. The process runs until *FORM* returns or the process is explicitly deleted. An untrapped error within the process also deletes the process (unless its RESTARTABLE property is T), in which case a message is printed to that effect.

The remaining arguments are alternately property names and values. Any property/value pairs acceptable to PROCESSPROP may be given, but the following two are directly relevant to ADD.PROCESS:

NAME Value should be a litatom; if not given, the process name is taken from (CAR *FORM*). ADD.PROCESS may pack the name with a number to make it unique. This name is solely for the convenience of manipulating processes at Lisp typein; e.g., the name can be given as the *PROC* argument to most process functions, and the name appears in menus of processes. However, programs should normally only deal in process handles, both for efficiency and to avoid the confusion that can result if two processes have the same defining form.

SUSPEND If the value is non-NIL, the new process is created but then immediately suspended; i.e., the process does not actually run until woken by a WAKE.PROCESS (below).

(PROCESSPROP *PROC PROP NEWVALUE*) [NoSpread Function]

Used to get or set the values of certain properties of process *PROC*, in a manner analogous to WINDOWPROP. If *NEWVALUE* is supplied (including if it is NIL), property *PROP* is given that value. In all cases, returns the old value of the property. The following properties have special meaning for processes; all others are uninterpreted:

NAME Value is a litatom used for identifying the process to the user.

FORM Value is the Lisp form used to start the process (readonly).

RESTARTABLE Value is a flag indicating the disposition of the process following errors or hard resets:

NIL or NO (the default): If an untrapped error (or Control-E or Control-D) causes its form to be exited, the process is

deleted. The process is also deleted if a HARDRESET (or Control-D from RAID) occurs, causing the entire Process world to be reinitialized.

T or YES: The process is automatically restarted on errors or HARDRESET. This is the normal setting for persistent "background" processes, such as the mouse process, that can safely restart themselves on errors.

HARDRESET: The process is deleted as usual if an error causes its form to be exited, but it *is* restarted on a HARDRESET. This setting is preferred for persistent processes for which an error is an unusual condition, one that might repeat itself if the process were simply blindly restarted.

RESTARTFORM — If the value is non-NIL, it is the form used if the process is restarted (instead of the value of the FORM property). Of course, the process must also have a non-NIL RESTARTABLE prop for this to have any effect.

BEFOREEXIT — If the value is the atom DON'T, it will not be interrupted by a LOGOUT. If LOGOUT is attempted before the process finishes, a message will appear saying that Interlisp is waiting for the process to finish. If you want the LOGOUT to proceed without waiting, you must use the process status window (from the background menu) to delete the process.

AFTEREXIT — Value indicates the disposition of the process following a resumption of Lisp after some exit (LOGOUT, SYSOUT, MAKESYS). Possible values are:

DELETE: Delete the process.

SUSPEND: Suspend the process; i.e., do not let it run until it is explicitly woken.

An event: Cause the process to be suspended waiting for the event (See the Events section below).

INFOHOOK — Value is a function or form used to provide information about the process, in conjunction with the INFO command in the process status window (see the Process Status Window section below).

WINDOW — Value is a window associated with the process, the process's "main" window. Used to switch the tty process to this process when you click in this window (see the

Switching the TTY Process section below).

Setting the WINDOW property does not set the primary I/O stream (NIL) or the terminal I/O stream (T) to the window. When a process is created, I/O operations to the NIL or T stream will cause a new window to appear. TTYDISPLAYSTREAM (see Chapter 28) should be used to set the terminal i/o stream of a process to a specific window.

TTYENTRYFN — Value is a function that is applied to the process when the process is made the tty process (see the Switching the TTY Process section below).

TTYEXITFN — Value is a function that is applied to the process when the process ceases to be the tty process (see the Switching the TTY Process section below).

(THIS.PROCESS) [Function]

Returns the handle of the currently running process, or NIL if the Process world is turned off.

(DEL.PROCESS *PROC* —) [Function]

Deletes process *PROC*. *PROC* may be a process handle (returned by ADD.PROCESS), or its name. If *PROC* is the currently running process, DEL.PROCESS does not return!

(PROCESS.RETURN *VALUE*) [Function]

Terminates the currently running process, causing it to "return" *VALUE*. There is an implicit PROCESS.RETURN around the *FORM* argument given to ADD.PROCESS, so that normally a process can finish by simply returning; PROCESS.RETURN is supplied for earlier termination.

(PROCESS.RESULT *PROCESS WAITFORRESULT*) [Function]

If *PROCESS* has terminated, returns the value, if any, that it returned. This is either the value of a PROCESS.RETURN or the value returned from the form given to ADD.PROCESS. If the process was aborted, the value is NIL. If *WAITFORRESULT* is true, PROCESS.RESULT blocks until *PROCESS* finishes, if necessary; otherwise, it returns NIL immediately if *PROCESS* is still running. *PROCESS* must be the actual process handle returned from ADD.PROCESS, not a process name, as the association between handle and name disappears when the process finishes (and the process handle itself is then garbage collected if no one else has a pointer to it).

(PROCESS.FINISHEDP *PROCESS*) [Function]

True if *PROCESS* has terminated. The value returned is an indication of how it finished: NORMAL or ERROR.

(PROCESSP *PROC*) [Function]

True if *PROC* is the handle of an active process, i.e., one that has not yet finished.

(RELPROCESSP *PROCHANDLE*) [Function]

True if *PROCHANDLE* is the handle of a deleted process. This is analogous to RELSTKP. It differs from PROCESS.FINISHEDP in that it never causes an error, while PROCESS.FINISHEDP can cause an error if its *PROC* argument is not a process at all.

(RESTART.PROCESS *PROC*) [Function]

Unwinds *PROC* to its top level and reevaluates its form. This is effectively a DEL.PROCESS followed by the original ADD.PROCESS.

(MAP.PROCESSES *MAPFN*) [Function]

Maps over all processes, calling *MAPFN* with three arguments: the process handle, its name, and its form.

(FIND.PROCESS *PROC ERRORFLG*) [Function]

If *PROC* is a process handle or the name of a process, returns the process handle for it, else NIL. If *ERRORFLG* is T, generates an error if *PROC* is not, and does not name, a live process.

## Process Control Constructs

(BLOCK *MSECSWAIT TIMER*) [Function]

Yields control to the next waiting process, assuming any is ready to run. If *MSECSWAIT* is specified, it is a number of milliseconds to wait before returning, or T, meaning wait forever (until explicitly woken). Alternatively, *TIMER* can be given as a millisecond timer (as returned by SETUPTIMER, Chapter 12) of an absolute time at which to wake up. In any of those cases, the process enters the *waiting* state until the time limit is up. BLOCK with no arguments leaves the process in the *runnable* state, i.e., it returns as soon as every other runnable process of the same priority has had a chance.

BLOCK can be aborted by interrupts such as Control-D, Control-E, or Control-B. BLOCK will return before its timeout is completed, if the process is woken by

WAKE.PROCESS, PROCESS.EVAL, or PROCESS.APPLY.

(DISMISS *MSECSWAIT TIMER NOBLOCK*) [Function]

DISMISS is used to dismiss the current process for a given period of time. Similar to BLOCK, except that:

- DISMISS is guaranteed not to return until the specified time has elapsed
- *MSECSWAIT* cannot be T to wait forever
- If *NOBLOCK* is T, DISMISS will not allow other processes to run, but will busy-wait until the amount of time given has elapsed.

(WAKE.PROCESS *PROC STATUS*) [Function]

Explicitly wakes process *PROC*, i.e., makes it *runnable*, and causes its call to BLOCK (or other waiting function) to return *STATUS*. This is one simple way to notify a process of some happening; however, note that if WAKE.PROCESS is applied to a process more than once before the process actually gets its turn to run, it sees only the latest *STATUS*.

(SUSPEND.PROCESS *PROC*) [Function]

Blocks process *PROC* indefinitely, i.e., *PROC* will not run until it is woken by a WAKE.PROCESS.

The following three functions allow access to the stack context of some other process. They require a little bit of care, and are computationally non-trivial, but they do provide a more powerful way of manipulating another process than WAKE.PROCESS allows.

(PROCESS.EVALV *PROC VAR*) [Function]

Performs (EVALV *VAR*) in the stack context of *PROC*.

(PROCESS.EVAL *PROC FORM WAITFORRESULT*) [Function]

Evaluates *FORM* in the stack context of *PROC*. If *WAITFORRESULT* is true, blocks until the evaluation returns a result, else allows the current process to run in parallel with the evaluation. Any errors that occur will be in the context of *PROC*, so be careful. In particular, note that

```
(PROCESS.EVAL PROC '(NLSETQ (FOO)))
```

and

```
(NLSETQ (PROCESS.EVAL PROC '(FOO)))
```

behave quite differently if FOO causes an error. And it is quite permissible to intentionally cause an error in proc by performing

```
(PROCESS.EVAL PROC '(ERROR!))
```

If errors are possible and *WAITFORRESULT* is true, the caller should almost certainly make sure that *FORM* traps the errors; otherwise the caller could end up waiting forever if *FORM* unwinds back into the pre-existing stack context of *PROC*.

After *FORM* is evaluated in *PROC*, the process *PROC* is woken up, even if it was running BLOCK or AWAIT.EVENT. This is necessary because an event of interest may have occurred while the process was evaluating *FORM*.

(PROCESS.APPLY *PROC FN ARGS WAITFORRESULT*) [Function]

Performs (APPLY *FN ARGS*) in the stack context of *PROC*. Note the same warnings as with PROCESS.EVAL.

## Events

An "event" is a synchronizing primitive used to coordinate related processes, typically producers and consumers. Consumer processes can "wait" on events, and producers "notify" events.

(CREATE.EVENT *NAME*) [Function]

Returns an instance of the EVENT datatype, to be used as the event argument to functions listed below. *NAME* is arbitrary, and is used for debugging or status information.

(AWAIT.EVENT *EVENT TIMEOUT TIMERP*) [Function]

Suspends the current process until *EVENT* is notified, or until a timeout occurs. If *TIMEOUT* is NIL, there is no timeout. Otherwise, timeout is either a number of milliseconds to wait, or, if *TIMERP* is T, a millisecond timer set to expire at the desired time using SETUPTIMER (see Chapter 12).

(NOTIFY.EVENT *EVENT ONCEONLY*) [Function]

If there are processes waiting for *EVENT* to occur, causes those processes to be placed in the running state, with *EVENT* returned as the value from AWAIT.EVENT. If *ONCEONLY* is true, only runs the first process waiting for the event (this should only be done if the programmer knows that there can only be one process capable of responding to the event at once).

The meaning of an event is up to the programmer. In general, however, the notification of an event is merely a hint that something of interest to the waiting process has happened; the process should still verify that the conceptual event actually occurred. That is, *the process should be written so that it operates correctly even if woken up before the timeout and in the absence of the notified event.* In particular, the completion of

PROCESS.EVAL and related operations in effect wakes up the process in which they were performed, since there is no secure way of knowing whether the event of interest occurred while the process was busy performing the PROCESS.EVAL.

There is currently one class of system-defined events, used with the network code. Each Pup and NS socket has associated with it an event that is notified when a packet arrives on the socket; the event can be obtained by calling PUPSOCKETEVENT or NSOCKETEVENT, respectively (see Chapter 32).

## Monitors

It is often the case that cooperating processes perform operations on shared structures, and some mechanism is needed to prevent more than one process from altering the structure at the same time. Some languages have a construct called a monitor, a collection of functions that access a common structure with mutual exclusion provided and enforced by the compiler via the use of monitor locks. Interlisp-D has taken this implementation notion as the basis for a mutual exclusion capability suitable for a dynamically-scoped environment.

A monitorlock is an object created by you and associated with (e.g., stored in) some shared structure that is to be protected from simultaneous access. To access the structure, a program waits for the lock to be free, then takes ownership of the lock, accesses the structure, then releases the lock. The functions and macros below are used:

(CREATE.MONITORLOCK *NAME* —) [Function]

Returns an instance of the MONITORLOCK datatype, to be used as the lock argument to functions listed below. *NAME* is arbitrary, and is used for debugging or status information.

(WITH.MONITOR *LOCK* $FORM_1$ ... $FORM_N$) [Macro]

Evaluates $FORM_1$ ... $FORM_N$ while owning *LOCK*, and returns the value of $FORM_N$. This construct is implemented so that the lock is released even if the form is exited via error (currently implemented with RESETLST).

Ownership of a lock is dynamically scoped: if the current process already owns the lock (e.g., if the caller was itself inside a WITH.MONITOR for this lock), WITH.MONITOR does not wait for the lock to be free before evaluating $FORM_1$ ... $FORM_N$.

(WITH.FAST.MONITOR *LOCK* $FORM_1$ ... $FORM_N$) [Macro]

Like WITH.MONITOR, but implemented without the RESETLST. User interrupts

(e.g., Control-E) are inhibited during the evaluation of $FORM_1$ ... $FORM_N$.

Programming restriction: the evaluation of $FORM_1$ ... $FORM_N$ must not error (the lock would not be released). This construct is mainly useful when the forms perform a small, safe computation that never errors and need never be interrupted.

(MONITOR.AWAIT.EVENT *RELEASELOCK EVENT TIMEOUT TIMERP*) [Function]

For use in blocking inside a monitor. Performs (AWAIT.EVENT *EVENT TIMEOUT TIMERP*), but releases *RELEASELOCK* first, and reobtains the lock (possibly waiting) on wakeup.

Typical use for MONITOR.AWAIT.EVENT: A function wants to perform some operation on *FOO*, but only if it is in a certain state. It has to obtain the lock on the structure to make sure that the state of the structure does not change between the time it tests the state and performs the operation. If the state turns out to be bad, it then waits for some other process to make the state good, meanwhile releasing the lock so that the other process can alter the structure.

```
(WITH.MONITOR FOO-LOCK
  (until CONDITION-OF-FOO
  do (MONITOR.AWAIT.EVENT FOO-LOCK EVENT-FOO-
CHANGED TIMEOUT))
    OPERATE-ON-FOO)
```

It is sometimes convenient for a process to have WITH.MONITOR at its top level and then do all its interesting waiting using MONITOR.AWAIT.EVENT. Not only is this often cleaner, but in the present implementation in cases where the lock is frequently accessed, it saves the RESETLST overhead of WITH.MONITOR.

Programming restriction: There must not be an ERRORSET between the enclosing WITH.MONITOR and the call to MONITOR.AWAIT.EVENT such that the ERRORSET would catch an ERROR! and continue inside the monitor, for the lock would not have been reobtained. (The reason for this restriction is that, although MONITOR.AWAIT.EVENT won't itself error, you could have caused an error with an interrupt, or a PROCESS.EVAL in the context of the waiting process that produced an error.)

On rare occasions it may be useful to manipulate monitor locks directly. The following two functions are used in the implementation of WITH.MONITOR:

(OBTAIN.MONITORLOCK *LOCK DONTWAIT UNWINDSAVE*) [Function]

Takes possession of *LOCK*, waiting if necessary until it is free, unless *DONTWAIT* is true, in which case it returns NIL immediately. If *UNWINDSAVE* is true, performs a RESETSAVE to be unwound when the enclosing RESETLST exits. Returns *LOCK* if *LOCK* was successfully obtained, T if the current process already owned *LOCK*.

(RELEASE.MONITORLOCK *LOCK EVENIFNOTMINE*) [Function]

Releases *LOCK* if it is owned by the current process, and wakes up the next process, if any, waiting to obtain the lock.

If *EVENIFNOTMINE* is non-NIL, the lock is released even if it is not owned by the current process.

When a process is deleted, any locks it owns are released.

## Global Resources

The biggest source of problems in the multi-processing environment is the matter of global resources. Two processes cannot both use the same global resource if there can be a process switch in the middle of their use (currently this means calls to BLOCK, but ultimately with a preemptive scheduler means anytime). Thus, user code should be wary of its own use of global variables, if it ever makes sense for the code to be run in more than one process at a time. "State" variables private to a process should generally be bound in that process; structures that are shared among processes (or resources used privately but expensive to duplicate per process) should be protected with monitor locks or some other form of synchronization.

Aside from user code, however, there are many *system* global variables and resources. Most of these arise historically from the single-process Interlisp-10 environment, and will eventually be changed in Interlisp-D to behave appropriately in a multi-processing environment. Some have already been changed, and are described below. Two other resources not generally thought of as global variables—the keyboard and the mouse—are particularly idosyncratic, and are discussed in the next section.

The following resources, which are global in Interlisp-10, are allocated per process in Interlisp-D: primary input and output (the streams affected by **INPUT** and **OUTPUT**), terminal input and output (the streams designated by the name **T**), the primary read table and primary terminal table, and dribble files. Thus, each process can print to its own primary output, print to the terminal, read from a different primary input, all without interfering with another process's reading and printing.

Each process begins life with its primary and terminal input/output streams set to a dummy stream. If the process attempts input or output using any of those dummy

streams, e.g., by calling (READ T), or (PRINT & T), a tty window is automatically created for the process, and that window becomes the primary input/output and terminal input/output for the process. The default tty window is created at or near the region specified in the variable DEFAULTTTYREGION.

A process can, of course, call TTYDISPLAYSTREAM explicitly to give itself a tty window of its own choosing, in which case the automatic mechanism never comes into play. Calling TTYDISPLAYSTREAM when a process has no tty window not only sets the terminal streams, but also sets the primary input and output streams to be that window, assuming they were still set to the dummy streams.

(HASTTYWINDOWP *PROCESS*) [Function]

> Returns T if the process *PROCESS* has a tty window; NIL otherwise. If *PROCESS* is NIL, it defaults to the current process.

Other system resources that are typically changed by RESETFORM, RESETLST, or RESETVARS are all global entities. In the multiprocessing environment, these constructs are suspect, as there is no provision for "undoing" them when a process switch occurs. For example, in the current release of Interlisp-D, it is not possible to set the print radix to 8 inside only one process, as the print radix is a global entity.

Note that RESETFORM and similar expressions are perfectly valid in the process world, and even quite useful, when they manipulate things strictly within one process. The process world is arranged so that deleting a process also unwinds any RESETxxx expressions that were performed in the process and are still waiting to be unwound, exactly as if a Control-D had reset the process to the top. Additionally, there is an implicit RESETLST at the top of each process, so that RESETSAVE can be used as a way of providing "cleanup" functions for when a process is deleted. For these, the value of RESETSTATE (see Chapter 14) is NIL if the process finished normally, ERROR if it was aborted by an error, RESET if the process was explicitly deleted, and HARDRESET if the process is being restarted after a HARDRESET or a RESTART.PROCESS.

## Typein and the TTY Process

There is one global resource, the keyboard, that is particularly problematic to share among processes. Consider, for example, having two processes both performing (READ T). Since the keyboard input routines block while there is no input, both processes would spend most of their time blocking, and it would simply be a matter of chance which process received each character of typein.

To resolve such dilemmas, the system designates a distinguished process, termed the *tty process*, that is assumed to be the process that is involved in terminal interaction. Any typein from the keyboard goes to that process. If a process other than the tty process requests keyboard input, it blocks until it becomes the tty process. When the

tty process is switched (in any of the ways described further below), any typeahead that occurred before the switch is saved and associated with the current tty process. Thus, it is always the case that keystrokes are sent to the process that is the tty process at the time of the keystrokes, regardless of when that process actually gets around to reading them.

It is less immediately obvious how to handle keyboard interrupt characters, as their action is asynchronous and not always tied to typein. Interrupt handling is described in the Handling of Interrupts section below.

## Switching the TTY Process

Any process can make itself be the tty process by calling TTY.PROCESS.

(TTY.PROCESS *PROC*) [Function]

Returns the handle of the current tty process. In addition, if *PROC* is non-NIL, makes it be the tty process. The special case of *PROC* = T is interpreted to mean the executive process; this is sometimes useful when a process wants to explicitly give up being the tty process.

(TTY.PROCESSP *PROC*) [Function]

True if *PROC* is the tty process; *PROC* defaults to the running process. Thus, (TTY.PROCESSP) is true if the caller is the tty process.

(WAIT.FOR.TTY *MSECS NEEDWINDOW*) [Function]

Efficiently waits until (TTY.PROCESSP) is true. WAIT.FOR.TTY is called internally by the system functions that read from the terminal; user code thus need only call it in special cases.

If *MSECS* is non-NIL, it is the number of milliseconds to wait before timing out. If WAIT.FOR.TTY times out before (TTY.PROCESSP) is true, it returns NIL, otherwise it returns T. If *MSECS* is NIL, WAIT.FOR.TTY will not time out.

If *NEEDWINDOW* is non-NIL, WAIT.FOR.TTY opens a TTY window for the current process if one isn't already open.

WAIT.FOR.TTY spawns a new mouse process if called under the mouse process (see SPAWN.MOUSE, in the Keeping the Mouse Alive section below).

In some cases, such as in functions invoked as a result of mouse action or a user's typed-in call, it is reasonable for the function to invoke TTY.PROCESS itself so that it can take subsequent user type in. In other cases, however, this is too undisciplined; it is desirable to let the user designate which process typein should be directed to. This is most conveniently done by mouse action.



The system supports the model that "to type to a process, you click in its window." To

cooperate with this model, any process desiring keyboard input should put its process handle as the PROCESS property of its window(s). To handle the common case, the function TTYDISPLAYSTREAM does this automatically when the ttydisplaystream is switched to a new window. A process can own any number of windows; clicking in any of those windows gives the process the tty.

This mechanism suffices for most casual process writers. For example, if a process wants all its input/output interaction to occur in a particular window that it has created, it should just make that window be its tty window by calling TTYDISPLAYSTREAM. Thereafter, it can PRINT or READ to/from the T stream; if the process is not the tty process at the time that it calls READ, it will block until the user clicks in the window.

For those needing tighter control over the tty, the default behavior can be overridden or supplemented. The remainder of this section describes the mechanisms involved.

There is a window property WINDOWENTRYFN that controls whether and how to switch the tty to the process owning a window. The mouse handler, before invoking any normal BUTTONEVENTFN, specifically notices the case of a button going down in a window that belongs to a process (i.e., has a PROCESS window property) that is not the tty process. In this case, it invokes the window's WINDOWENTRYFN of one argument (*WINDOW*). WINDOWENTRYFN defaults to GIVE.TTY.PROCESS:

(GIVE.TTY.PROCESS *WINDOW*) [Function]

> If *WINDOW* has a PROCESS property, performs (TTY.PROCESS (WINDOWPROP *WINDOW* 'PROCESS)) and then invokes *WINDOW*'s BUTTONEVENTFN function (or RIGHTBUTTONFN if the right button is down).

There are some cases where clicking in a window does not always imply that the user wants to talk to that window. For example, clicking in a text editor window with a shift key held down means to "shift-select" some piece of text into the input buffer of the *current* tty process. The editor supports this by supplying a WINDOWENTRYFN that performs GIVE.TTY.PROCESS if no shift key is down, but goes into its shift-select mode, without changing the tty process, if a shift key is down. The shift-select mode performs a BKSYSBUF of the selected text when the shift key is let up, the BKSYSBUF feeding input to the current tty process.

Sometimes a process wants to be notified when it becomes the tty process, or stops being the tty process. To support this, there are two process properties, TTYEXITFN and TTYENTRYFN. The actions taken by TTY.PROCESS when it switches the tty to a new process are as follows: the former tty process's TTYEXITFN is called with two arguments (*OLDTTYPROCESS NEWTTYPROCESS*); the new process is made the tty process; finally, the new tty process's TTYENTRYFN is called with two arguments (*NEWTTYPROCESS OLDTTYPROCESS*). Normally the TTYENTRYFN and TTYEXITFN need only their first argument, but the other process involved in the switch is supplied

for completeness. In the present system, most processes want to interpret the keyboard in the same way, so it is considered the responsibility of any process that changes the keyboard interpretation to restore it to the normal state by its TTYEXITFN.

A window is "owned" by the last process that anyone gave as the window's PROCESS property. Ordinarily there is no conflict here, as processes tend to own disjoint sets of windows (though, of course, cooperating processes can certainly try to confuse each other). The only likely problem arises with that most global of windows, PROMPTWINDOW. Programs should not be tempted to read from PROMPTWINDOW. This is not usually necessary anyway, as the first attempt to read from T in a process that has not set its TTYDISPLAYSTREAM to its own window causes a tty window to be created for the process (see the Global Resources section above).

## Handling of Interrupts

At the time that a keyboard interrupt character (see Chapter 30) is struck, any process could be running, and some decision must be made as to which process to actually interrupt. To the extent that keyboard interrupts are related to typein, most interrupts are taken in the tty process; however, the following are handled specially:

RESET (initially Control-D)
ERROR (initially Control-E)
: These interrupts are taken in the mouse process, if the mouse is not in its idle state; otherwise they are taken in the tty process. Thus, Control-E can be used to abort some mouse-invoked window action, such as the Shape command. As a consequence, note that if the mouse invokes some lengthy computation that the user thinks of as "background", Control-E still aborts it, even though that may not have been what the user intended. Such lengthy computations, for various reasons, should generally be performed by spawning a separate process to perform them. The RESET interrupt in a process other than the executive is interpreted exactly as if an error unwound the process to its top level: if the process was designated RESTARTABLE = T, it is restarted; otherwise it is killed.

HELP (initially Control-G)
: A menu of processes is presented to the user, who is asked to select which one the interrupt should occur in. The current tty process appears with a * next to its name at the top of the menu. The menu also includes an entry "[Spawn Mouse]", for the common case of needing a mouse because the mouse process is currently tied up running

someone's BUTTONEVENTFN; selecting this entry spawns a new mouse process, and no break occurs.

BREAK (initially Control-B) — Performs the HELP interrupt in the mouse process, if the mouse is not in its idle state; otherwise it is performed in the tty process.

RUBOUT (initially DELETE) — This interrupt clears typeahead in *all* processes.

RAID, STACK OVERFLOW, STORAGE FULL — These interrupts always occur in whatever process was running at the time the interrupt struck. In the cases of STACK OVERFLOW and STORAGE FULL, this means that the interrupt is more likely to strike in the offending process (especially if it is a "runaway" process that is not blocking). Note, however, that this process is still not necessarily the guilty party; it could be an innocent bystander that just happened to use up the last of a resource prodigiously consumed by some other process.

## Keeping the Mouse Alive

Since the window mouse handler runs in its own process, it is not available while a window's BUTTONEVENTFN function (or any of the other window functions invoked by mouse action) is running. This leads to two sorts of problems: (1) a long computation underneath a BUTTONEVENTFN deprives the user of the mouse for other purposes, and (2) code that runs as a BUTTONEVENTFN cannot rely on other BUTTONEVENTFNs running, which means that there some pieces of code that run differently from normal when run under the mouse process. These problems are addressed by the following functions:

(SPAWN.MOUSE —) [Function]

Spawns another mouse process, allowing the mouse to run even if it is currently "tied up" under the current mouse process. This function is intended mainly to be typed in at the Lisp executive when the user notices the mouse is busy.

(ALLOW.BUTTON.EVENTS) [Function]

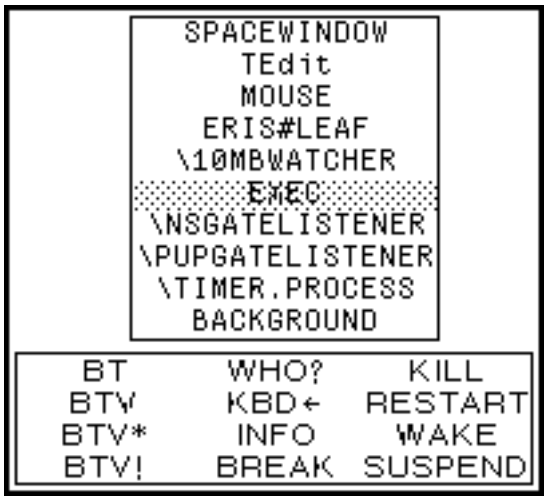
Performs a (SPAWN.MOUSE) only when called underneath the mouse process. This should be called (once, on entry) by any function that relies on BUTTONEVENTFNs for completion, if there is any possibility that the function will itself be invoked by a mouse function.

It never hurts, at least logically, to call SPAWN.MOUSE or ALLOW.BUTTON.EVENTS needlessly, as the mouse process arranges to quietly kill itself if it returns from the

user's BUTTONEVENTFN and finds that another mouse process has sprung up in the meantime. (There is, of course, some computational expense.)

## Process Status Window

The background menu command PSW (see Chapter 28) and the function PROCESS.STATUS.WINDOW (below) create a "Process Status Window", that allows the user to examine and manipulate all of the existing processes:




The window consists of two menus. The top menu lists all the processes at the moment. Commands in the bottom menu operate on the process selected in the top menu (EXEC in the example above). The commands are:

BT, BTV, BTV*, BTV! — Displays a backtrace of the selected process.

WHO? — Changes the selection to the tty process, i.e., the one currently in control of the keyboard.

KBD← — Associates the keyboard with the selected process; i.e., makes the selected process be the tty process.

INFO — If the selected process has an INFOHOOK property, calls it. The hook may be a function, which is then applied to two arguments, the process and the button (LEFT or MIDDLE) used to invoke INFO, or a form, which is simply EVAL'ed. The APPLY or EVAL happens in the context of the selected process, using PROCESS.APPLY or PROCESS.EVAL. The INFOHOOK process property can be set using PROCESSPROP (see the

Creating and Destroying Processes section above).

BREAK — Enter a break under the selected process. This has the side effect of waking the process with the value returned from the break.

KILL — Deletes the selected process.

RESTART — Restarts the selected process.

WAKE — Wakes the selected process. Prompts for a value to wake it with (see WAKE.PROCESS).

SUSPEND — Suspends the selected process; i.e., causes it to block indefinitely (until explicitly woken).

(PROCESS.STATUS.WINDOW *WHERE*) [Function]

Puts up a process status window that provides several debugging commands for manipulating running processes. If the window is already up, PROCESS.STATUS.WINDOW refreshes it. If *WHERE* is a position, the window is placed in that position; otherwise, the user is prompted for a position.

Currently, the process status window runs under the mouse process, like other menus, so if the mouse is unavailable (e.g., a mouse function is performing an extensive computation), you may be unable to use the process status window (you can try SPAWN.MOUSE, of course).

## Non-Process Compatibility

This section describes some considerations for authors of programs that ran in the old single-process Interlisp-D environment, and now want to make sure they run properly in the Multi-processing world. The biggest problem to watch out for is code that runs underneath the mouse handler. Writers of mouse handler functions should remember that in the process world the mouse handler runs in its own process, and hence (a) you cannot depend on finding information on the stack (stash it in the window instead), and (b) while your function is running, the mouse is not available (if you have any non-trivial computation to do, spawn a process to do it, notify one of your existing processes to do it, or use PROCESS.EVAL to run it under some other process).

The following functions are meaningful even if the process world is not on: BLOCK (invokes the system background routine, which includes handling the mouse); TTY.PROCESS, THIS.PROCESS (both return NIL); and TTY.PROCESSP (returns T, i.e., anyone is allowed to take tty input). In addition, the following two functions exist in both worlds:

(EVAL.AS.PROCESS *FORM*) [Function]

Same as (ADD.PROCESS *FORM* 'RESTARTABLE 'NO), when processes are running, EVAL when not. This is highly recommended for mouse functions that perform any non-trivial activity.

(EVAL.IN.TTY.PROCESS *FORM WAITFORRESULT*) [Function]

Same as (PROCESS.EVAL (TTY.PROCESS) *FORM WAITFORRESULT*), when processes are running, EVAL when not.

Most of the process functions that do not take a process argument can be called even if processes aren't running. ADD.PROCESS creates, but does not run, a new process (it runs when PROCESSWORLD is called).