29. Processes

The Lisp Machine supports *multi-processing*; several computations can be executed concurrently by placing each in a separate *process*. A process is like a processor, simulated by software. Each process has its own program counter, its own stack of function calls and its own special-variable binding environment in which to execute its computation. (This is implemented with stack groups; see chapter 13, page 256.)

If all the processes are simply trying to compute, the machine allows them all to run an equal share of the time. This is not a particularly efficient mode of operation since dividing the finite memory and processor power of the machine among several processes certainly cannot increase the available power and in fact wastes some of it in overhead. The typical use for processes is that at any time only one or two are trying to run. The rest are either *waiting* for some event to occur or *stopped* and not allowed to compete for resources.

A process waits for an event by means of the process-wait primitive, which is given a predicate function which defines the event being waited for. A module of the system called the process scheduler periodically calls that function. If it returns nil the process continues to wait; if it returns t the process is made runnable and its call to process-wait returns, allowing the computation to proceed.

A process may be *active* or *stopped*. Stopped processes are never allowed to run; they are not considered by the scheduler, and so can never become the current process until they are made active again. The scheduler continually tests the waiting functions of all the active processes, and those which return non-nil values are allowed to run. When you first create a process with make-process, it is inactive.

The activity of a process is controlled by two sets of Lisp objects associated with it, called its *run reasons* and its *arrest reasons*. These sets are implemented as lists. Any kind of object can be in these sets; typically keyword symbols and active objects such as windows and other processes are found. A process is considered active when it has at least one run reason and no arrest reasons.

To get a computation to happen in another process, you must first create a process, then say what computation you want to happen in that process. The computation to be executed by a process is specified as an *initial function* and a list of arguments to that function. When the process starts up it applies the function to the arguments. In some cases the initial function is written so that it never returns, while in other cases it performs a certain computation and then returns, which stops the process.

To *reset* a process means to exit its entire computation nonlocally using *unwind-stack (see page 82). Some processes are temporary and die when reset. The other, permanent functions start their computations over again when reset. Resetting a process clears its waiting condition, so that if it is active it becomes runnable. To *preset* a function is to set up its initial function (and arguments) and then reset it. This is how you start up a computation in a process.

All processes in a Lisp Machine run in the same virtual address space, sharing the same set of Lisp objects. Unlike other systems that have special restricted mechanisms for inter-process communication, the Lisp Machine allows processes to communicate in arbitrary ways through shared Lisp objects. One process can inform another of an event simply by changing the value of a global variable. Buffers containing messages from one process to another can be implemented as lists or arrays. The usual mechanisms of atomic operations, critical sections, and interlocks are provided (see store-conditional [page 688], without-interrupts [page 684], and process-lock [page 687]).

A process is a Lisp object, an instance of one of several flavors of process (see chapter 21, page 401). The remainder of this chapter describes the operations defined on processes, the functions you can apply to a process, and the functions and variables a program running in a process can use to manipulate its process.

29.1 The Scheduler

At any time there is a set of *active processes*; as described above, these are all the processes that are not stopped. Each active process is either currently running, runnable (ready to run), or waiting for some condition to become true. The active processes are managed by a special stack group called the *scheduler*, which repeatedly examines each active process to determine whether it is waiting or ready to run. The scheduler then selects one process and starts it up.

The process chosen by the scheduler becomes the *current process*, that is, the one process that is running on the machine. The scheduler sets the variable current-process to it. It remains the current process and continues to run until either it decides to wait, or a *sequence break* occurs. In either case, the scheduler stack group is resumed. It then updates the process's run time meters and chooses a new process to run next. This way, each process that is ready to run gets its share of time in which to execute.

Each process has a *priority* which is a number. Most processes have priority zero. Larger numbers give a process more priority. The scheduler only considers the highest priority runnable processes, so if there is one runnable process with priority 20 then no process with lesser priority can run.

The scheduler determines whether a process is runnable by applying the process's *wait-function* to its *wait-argument-list*. If the wait-function returns a non-nil value, then the process is ready to run; otherwise, it is waiting.

A process can wait for some condition to become true by calling process-wait (see page 685). This function sets the process's wait-function and wait-argument-list as specified by the caller, and resumes the scheduler stack group. A process can also wait for just a moment by calling process-allow-schedule (see page 686), which resumes the scheduler stack group but leaves the process runnable; it will run again as soon as all other runnable processes have had a chance.

A sequence break is a kind of interrupt that is generated by the Lisp system for any of a variety of reasons; when it occurs, the scheduler is resumed. The function si:sb-on (see page 687) can be used to control when sequence breaks occur. The default is to sequence break once a second. Thus even if a process never waits and is not stopped, it is forced to return control to the scheduler once a second so that any other runnable processes can get their turn.

The system does not generate a sequence break when a page fault occurs; thus time spent waiting for a page to come in from the disk is "charged" to a process the same as time spent computing, and cannot be used by other processes. It is done this way for the sake of simplicity; this allows the whole implementation of the process system to reside in ordinary virtual memory, so that it does not have to worry specially about paging. The performance penalty is small since Lisp Machines are personal computers, not multiplexed among a large number of processes. Usually only one process at a time is runnable.

A process's wait function is free to touch any data structure it likes and to perform any computation it likes. Of course, wait functions should be kept simple, using only a small amount of time and touching only a small number of pages, or system performance will be impacted since the wait function will consume resources even when its process is not running.

If a wait function gets an error, the error occurs inside the scheduler. If this enters the debugger, all scheduling comes to a halt until the user proceeds or aborts. Aborting in the debugger inside the scheduler "blasts" the current process by giving it a trivial wait function that always returns nil; this prevents recurrence of the same problem. It is best to write wait functions that cannot get errors, by keeping them simple and by arranging for any problems to be detected before the scheduler sees the wait function. process-wait calls the wait function once before giving it to the scheduler, and this often exposes an error before it can interfere with scheduling.

Note well that a process's wait function is executed inside the scheduler stack-group, *not* inside the process. This means that a wait function may not access special variables bound in the process. It is allowed to access global variables. It can access variables bound by a process through the closure mechanism (chapter 12, page 250). If the wait function is defined lexically within the caller of process-wait then it can access local variables through the lexical scoping mechanism. Most commonly any values needed by the wait function are passed to it as arguments.

current-process *Variable*

The value of current-process is the process that is currently executing, or nil while the scheduler is running. When the scheduler calls a process's wait-function, it binds current-process to the process so that the wait-function can access its process.

without-interrupts *body...* *Macro*

The *body* forms are evaluated with inhibit-scheduling-flag bound to t. This is the recommended way to lock out multi-processing over a small critical section of code to prevent timing errors. In other words the body is an *atomic operation*. The values of the last form in the body are ultimately returned.

In this example, list is presumed to be a global variable referred to from two places in the code which different processes will execute.

```
(without-interrupts
  (push item list))

(without-interrupts
  (cond ((memq item list)
         (setq list (delq item list))
         t)
        (t nil)))
```

inhibit-scheduling-flag *Variable*

The value of inhibit-scheduling-flag is normally nil. without-interrupts binds it to t, which prevents process-switching until inhibit-scheduling-flag becomes nil again. It is cleaner to use without-interrupts than to refer directly to this variable.

process-wait *whostate function* &rest *arguments*

This is the primitive for waiting. The current process waits until the application of *function* to *arguments* returns non-nil (at which time process-wait returns). Note that *function* is applied in the environment of the scheduler, not the environment of the process-wait, so special bindings in effect when process-wait was called are *not* be in effect when *function* is applied. Be careful when using any free references in *function*. *whostate* is a string containing a brief description of the reason for waiting. If the who-line at the bottom of the screen is looking at this process, it will show *whostate*.
Example:

```
(process-wait "Buffer"
        #'(lambda (b) (not (zerop (buffer-n-things b))))
        the-buffer)
```

process-sleep *interval*

Waits for *interval* sixtieths of a second, and then returns. It uses process-wait.

sleep *seconds*

Waits *seconds* seconds and then returns. *seconds* need not be an integer. This also uses process-wait.

process-wait-with-timeout *whostate interval function* &rest *arguments*

This is like process-wait except that if *interval* sixtieths of a second go by and the application of *function* to *arguments* is still returning nil, then process-wait-with-timeout returns anyway. The value returned is the value of applying *function* to *arguments*; thus, it is non-nil if the wait condition actually occurred, nil for a time-out.

If *interval* is nil, there is no timeout, and this function is then equivalent to process-wait.

with-timeout (*interval timeout-forms...*) *body...* *Macro*

body is executed with a timeout in effect for *interval* sixtieths of a second. If *body* finishes before that much time elapses, the values of the last form in *body* are returned.

If after *interval* has elapsed *body* has not completed, its execution is terminated with a throw caught by the with-timeout form. Then the *timeout-forms* are evaluated and the values of the last one of them are returned.

For example,

```
(with-timeout ((* 60. 60.) (format *query-io* " ... Yes.") t)
  (y-or-n-p "Really do it? (Yes after one minute) "))
```

is a convenient way to ask a question and assume an answer if the user does not respond promptly. This is a good thing to do for queries likely to occur when the user has walked away from the terminal and expects an operation to finish without his attention.

process-allow-schedule

Resumes the scheduler momentarily; all other processes will get a chance to run before the current process runs again.

sys:scheduler-stack-group *Constant*

This is the stack group in which the scheduler executes.

sys:clock-function-list *Variable*

This is a list of functions to be called by the scheduler 60 times a second. Each function is passed one argument, the number of 60ths of a second since the last time that the functions on this list were called. These functions implement various system overhead operations such as blinking the blinking cursor on the screen. Note that these functions are called inside the scheduler, just as are the functions of simple processes (see page 695). The scheduler calls these functions as often as possible, but never more often than 60 times a second. That is, if there are no processes ready to run, the scheduler calls the clock functions 60 times a second, assuming that, all together, they take less than 1/60 second to run. If there are processes continually ready to run, then the scheduler calls the clock functions as often as it can; usually this is once a second, since usually the scheduler gets control only once a second.

sys:active-processes *Variable*

This is the scheduler's data-structure. It is a list of lists, where the car of each element is an active process or nil and the cdr is information about that process.

sys:all-processes *Variable*

This is a list of all the processes in existence. It is mainly for debugging.

si:initial-process *Constant*

This is the process in which the system starts up when it is booted.

si:sb-on &optional *when*

Controls what events cause a sequence break, i.e. when rescheduling occurs. The following keywords are names of events which can cause a sequence break.

:clock	This event happens periodically based on a clock. The default period is one second. See **sys:%tv-clock-rate**, page 293.
:keyboard	Happens when a character is received from the keyboard.
:chaos	Happens when a packet is received from the Chaosnet, or transmission of a packet to the Chaosnet is completed.

Since the keyboard and Chaosnet are heavily buffered, there is no particular advantage to enabling the **:keyboard** and **:chaos** events, unless the **:clock** event is disabled.

With no argument, **si:sb-on** returns a list of keywords for the currently enabled events.

With an argument, the set of enabled events is changed. The argument can be a keyword, a list of keywords, **nil** (which disables sequence breaks entirely since it is the empty list), or a number, which is the internal mask, not documented here.

29.2 Locks

A *lock* is a software construct used for synchronization of two processes. A lock is either held by some process, or is free. When a process tries to seize a lock, it waits until the lock is free, and then it becomes the process holding the lock. When it is finished, it unlocks the lock, allowing some other process to seize it. A lock protects some resource or data structure so that only one process at a time can use it.

In the Lisp Machine, a lock is a locative pointer to a cell. If the lock is free, the cell contains **nil**; otherwise it contains the process that holds the lock. The **process-lock** and **process-unlock** functions are written in such a way as to guarantee that two processes can never both think that they hold a certain lock; only one process can ever hold a lock at one time.

process-lock *locative* &optional (*lock-value* **current-process**) (*whostate* **"Lock"**) *timeout*

This is used to seize the lock that *locative* points to. If necessary, **process-lock** waits until the lock becomes free. When **process-lock** returns, the lock has been seized. *lock-value* is the object to store into the cell specified by *locative*, and *whostate* is passed on to **process-wait**.

If *timeout* is non-**nil**, it should be a fixnum representing a time interval in 60ths of a second. If it is necessary to wait more than that long, an error with condition name **sys:lock-timeout** is signaled.

process-unlock *locative* &optional (*lock-value* **current-process**)

This is used to unlock the lock that *locative* points to. If the lock is free or was locked by some other process, an error is signaled. Otherwise the lock is unlocked. *lock-value* must have the same value as the *lock-value* parameter to the matching call to **process-lock**, or else an error is signaled.

sys:lock-timeout (error) *Condition*

This condition is signaled when **process-lock** waits longer than the specified timeout.

It is a good idea to use **unwind-protect** to make sure that you unlock any lock that you seize. For example, if you write

```
(unwind-protect
   (progn (process-lock lock-3)
          (function-1)
          (function-2))
   (process-unlock lock-3))
```

then even if **function-1** or **function-2** does a **throw**, **lock-3** will get unlocked correctly. Particular programs that use locks often define special forms that package this **unwind-protect** up into a convenient stylistic device.

A higher level locking construct is **with-lock**:

with-lock (*lock* &key *norecursive*) *body...* *Macro*

Executes the *body* with *lock* locked. *lock* should actually be an expression whose value would be the status of the lock; it is used inside **locf** to get a locative pointer with which the locking and unlocking are done.

It is OK for one process to lock a lock multiple times, recursively, using **with-lock**, provided *norecursive* is not **nil**.

norecursive should be literally t or **nil**; it is not evaluated. If it is t, this call to **with-lock** signals an error if the lock is already locked by the running process.

A lower level construct which can be used to implement atomic operations, and is used in the implementation of **process-lock**, is **store-conditional**.

store-conditional *location oldvalue newvalue*

This stores *newvalue* into *location* iff *location* currently contains *oldvalue*. The value is **t** iff the cell was changed.

If *location* is a list, the cdr of the list is tested and stored in. This is in accord with the general principle of how to access the contents of a locative properly, and makes **(store-conditional (locf (cdr x)) ...)** work.

An even lower-level construct is the subprimitive **%store-conditional**, which is like **store-conditional** with no error checking, but is faster.

29.2.1 Process Queues

A process queue is a kind of lock which can record several processes which are waiting for the lock and grant them the lock in the order that they requested it. The queue has a fixed size. If the number of processes waiting remains less than that size then they will all get the lock in the order of requests. If too many processes are waiting then the order of requesting is not remembered for the extra ones, but proper interlocking is still maintained.

si:make-process-queue *name size*
Makes and returns a process queue object named *name*, able to record *size* processes. The count of *size* includes the process that owns the lock.

si:process-enqueue *process-queue* &optional *lock-value who-state*
Attempts to lock *process-queue* on behalf of *lock-value*. If *lock-value* is nil then the locking is done on behalf of current-process.

If the queue is locked, then *lock-value* or the current process is put on the queue. Then this function waits for that lock value to reach the front of the queue. When it does so, the lock has been granted, and the function returns. The lock is now locked in the name of *lock-value* or the current process, until si:process-dequeue is used to unlock it.

who-state appears in the who line during the wait. It defaults to "Lock".

si:process-deqeueue *process-queue* &optional *lock-value*
Unlocks process-queue. *lock-value* (which defaults to the current process) must be the value which now owns the lock on the queue, or an error occurs. The next process or other object on the queue is granted the lock and its call to si:process-enqueue will therefore return.

si:reset-process-queue *process-queue*
Unlocks the queue and clears out the list of things waiting to lock it.

si:process-queue-locker *process-queue*
Returns the object in whose name the queue is currently locked, or nil if it is not now locked.

29.3 Creating a Process

There are two ways of creating a process. One is to create a permanent process which you will hold on to and manipulate as desired. The other way is to say simply, "call this function on these arguments in another process, and don't bother waiting for the result." In the latter case you never actually use the process itself as an object.

make-process *name* &key ...
Creates and returns a process named *name*. The process is not capable of running until it has been reset or preset in order to initialize the state of its computation.

Usually you do not need to specify any of the keyword arguments. The following keyword arguments are allowed:

:simple-p	Specifying t here gives you a simple process (see page 695).
:flavor	Specifies the flavor of process to be created. See section 29.5, page 695, for a list of all the flavors of process supplied by the system.
:stack-group	Specifies the stack group the process is to use. If this option is not specified, a stack group is created according to the relevant options below.
:warm-boot-action	What to do with the process when the machine is booted. See page 693.
:quantum	See page 692.
:priority	See page 693.
:run-reasons	Lets you supply an initial run reason. The default is **nil**.
:arrest-reasons	Lets you supply an initial arrest reason. The default is **nil**.
:sg-area	The area in which to create the stack group. The default is the value of **default-cons-area**.
:regular-pdl-area :special-pdl-area :regular-pdl-size :special-pdl-size	These are passed on to **make-stack-group**, page 259.
:swap-sv-on-call-out :swap-sv-of-sg-that-calls-me :trap-enable	Specify those attributes of the stack group. You don't want to use these.

If you specify **:flavor**, there can be additional options provided by that flavor.

The following functions allow you to call a function and have its execution happen asynchronously in another process. This can be used either as a simple way to start up a process that will run "forever", or as a way to make something happen without having to wait for it to complete. When the function returns, the process is returned to a pool of free processes for reuse. The only difference between these three functions is in what happens if the machine is booted while the process is still active.

Normally the function to be run should not do any I/O to the terminal, as it does not have a window and terminal I/O will cause it to make a notification and wait for user attention. Refer to section 13.5, page 264 for a discussion of the issues.

process-run-function *name-or-options function* &rest *args*

Creates a process, presets it to apply *function* to *args*, and starts it running. The value returned is the new process. The process is killed if *function* returns; by default, it is also killed if it is reset. Example:

```
(defun background-print (file)
  (process-run-function "Print" 'hardcopy-file file))
```

creates a background process that prints the specified file.

name-or-options can be either a string specifying a name for the process or a list of alternating keywords and values that can specify the name and various other parameters.

:name	This keyword should be followed by a string which specifies the name of the process. The default is "Anonymous".
:restart-after-reset	This keyword says what to do to the process if it is reset. nil means the process should be killed; anything else means the process should be restarted. nil is the default.
:warm-boot-action	What to do with the process when the machine is booted. See page 693.
:restart-after-boot	This is a simpler way of saying what to do with the process when the machine is booted. If the :warm-boot-action keyword is not supplied or its value is nil, then this keyword's value is used instead. nil means the process should be killed; anything else means the process should be restarted. nil is the default.
:quantum	See page 692.
:priority	See page 693.

process-run-restartable-function *name-or-keywords function* &rest *args*

This is the same as process-run-function except that the default is that the process will be restarted if reset or after a warm boot. You can get the same effect by using process-run-function with appropriate keywords.

29.4 Process Generic Operations

These are the operations that are defined on all flavors of process. Certain process flavors may define additional operations. Not all possible operations are listed here, only those of interest to the user.

29.4.1 Process Attributes

:name *Operation on* si:process

Returns the name of the process, which was the first argument to make-process or process-run-function when the process was created. The name is a string that appears in the printed-representation of the process, stands for the process in the who-line and the peek display, etc.

:stack-group *Operation on* si:process

Returns the stack group currently executing on behalf of this process. This can be different from the initial-stack-group if the process contains several stack groups that coroutine among themselves, or if the process is in the error-handler, which runs in its own stack group.

Note that the stack-group of a *simple* process (see page 695) is not a stack group at all, but a function.

:initial-stack-group *Operation on* si:process

Returns the stack group the initial-function is called in when the process starts up or is reset.

:initial-form *Operation on* si:process

Returns the initial "form" of the process. This isn't really a Lisp form; it is a cons whose car is the initial-function and whose cdr is the list of arguments to which that function is applied when the process starts up or is reset.

In a simple process (see page 695), the initial form is a list of one element, the process's function.

To change the initial form, use the :preset operation (see page 694).

:wait-function *Operation on* si:process

Returns the process's current wait-function, which is the predicate used by the scheduler to determine if the process is runnable. This is #'true if the process is running, and #'false if the process has no current computation (for instance, if it has just been created, its initial function has returned) or if the program has decided to wait indefinitely. The wait-function of a flushed process is si:flushed-process, a function equivalent to #'false but recognizably distinct.

:wait-argument-list *Operation on* si:process

Returns the arguments to the process's current wait-function. This is frequently the **&rest** argument to **process-wait** in the process's stack. The system always uses it in a safe manner, i.e. it forgets about it before **process-wait** returns.

:whostate *Operation on* si:process

Returns a string that is the state of the process to go in the who-line at the bottom of the screen. This is "run" if the process is running or trying to run; otherwise, it is the reason why the process is waiting. If the process is stopped, then this whostate string is ignored and the who-line displays **arrest** if the process is arrested or **stop** if the process has no run reasons.

:quantum *Operation on* si:process

:set-quantum *60ths* *Operation on* si:process

Respectively return and change the number of 60ths of a second this process is allowed to run without waiting before the scheduler will run someone else. The quantum defaults to 1 second.

:quantum-remaining *Operation on* si:process
Returns the amount of time remaining for this process to run, in 60ths of a second.

:priority *Operation on* si:process
:set-priority *priority-number* *Operation on* si:process
Respectively return and change the priority of this process. The larger the number, the more this process gets to run. Within a priority level the scheduler runs all runnable processes in a round-robin fashion. Regardless of priority a process will not run for more than its quantum. The default priority is 0, and no normal process uses other than 0.

:warm-boot-action *Operation on* si:process
:set-warm-boot-action *action* *Operation on* si:process
Respectively return and change the process's warm-boot-action, which controls what happens if the machine is booted while this process is active. (Contrary to the name, this applies to both cold and warm booting.) This can be nil or :flush, which means to *flush* the process (see page 695), or can be a function to call. The default is si:process-warm-boot-delayed-restart, which resets the process, causing it to start over at its initial function. You can also use si:process-warm-boot-reset, which throws out of the process' computation and kills the process, or si:process-warm-boot-restart, which is like the default but restarts the process at an earlier stage of system reinitialization. This is used for processes like the keyboard process and chaos background process, which are needed for reinitialization itself.

:simple-p *Operation on* si:process
Returns nil for a normal process, t for a simple process. See page 695.

:idle-time *Operation on* si:process
Returns the time in seconds since this process last ran, or nil if it has never run.

:total-run-time *Operation on* si:process
Returns the amount of time this process has run, in 60ths of a second. This includes cpu time and disk wait time.

:disk-wait-time *Operation on* si:process
Returns the amount of time this process has spent waiting for disk I/O, in 60ths of a second.

:cpu-time *Operation on* si:process
Returns the amount of time this process has spent actually executing instructions, in 60ths of a second.

:percent-utilization *Operation on* si:process
Returns the fraction of the machine's time this process has been using recently, as a percentage (a number between 0 and 100.0). The value is a weighted average giving more weight to more recent history.

:reset-meters *Operation on* si:process

Resets the run-time counters of the process to zero.

29.4.2 Run and Arrest Reasons

:run-reasons *Operation on* si:process

Returns the list of run reasons, which are the reasons why this process should be active (allowed to run).

:run-reason *object* *Operation on* si:process

Adds *object* to the process's run reasons. This can activate the process.

:revoke-run-reason *object* *Operation on* si:process

Removes *object* from the process's run reasons. This can stop the process.

:arrest-reasons *Operation on* si:process

Returns the list of arrest reasons, which are the reasons why this process should be inactive (forbidden to run).

:arrest-reason *object* *Operation on* si:process

Adds *object* to the process's arrest reasons. This can stop the process.

:revoke-arrest-reason *object* *Operation on* si:process

Removes *object* from the process's arrest reasons. This can activate the process.

:active-p *Operation on* si:process
:runnable-p *Operation on* si:process

These two operations are the same. t is returned if the process is active, i.e. it can run if its wait-function allows. nil is returned if the process is stopped.

29.4.3 Bashing the Process

:preset *function* &rest *args* *Operation on* si:process

Sets the process's initial function to *function* and initial arguments to *args*. The process is then reset so that any computation occuring in it is terminated and it begins anew by applying *function* to *args*. A :preset operation on a stopped process returns immediately, but does not activate the process; hence the process will not really apply *function* to *args* until it is activated later.

:reset &optional *no-unwind kill* *Operation on* si:process

Forces the process to throw out of its present computation and apply its initial function to its initial arguments, when it next runs. The throwing out is skipped if the process has no present computation (e.g. it was just created), or if the *no-unwind* option so specifies. The possible values for *no-unwind* are:

:unless-current	
nil	Unwind unless the stack group to be unwound is the one currently executing, or belongs to the current process.

:always	Unwind in all cases. This may cause the operation to throw through its caller instead of returning.
t	Never unwind.

If *kill* is t, the process is to be killed after unwinding it. This is for internal use by the :kill operation only.

A :reset operation on a stopped process returns immediately, but does not activate the process; hence the process will not really get reset until it is activated later.

:flush *Operation on* si:process

Forces the process to wait forever. A process may not :flush itself. Flushing a process is different from stopping it, in that it is still active and hence if it is reset or preset it will start running again.

A flushed process can be recognized because its wait-function is si:flushed-process. If a process belonging to a window is flushed, exposing or selecting the window resets the process.

:kill *Operation on* si:process

Gets rid of the process. It is reset, stopped, and removed from sys:all-processes.

:interrupt *function* &rest *args* *Operation on* si:process

Forces the process to apply *function* to *args*. When *function* returns, the process continues the interrupted computation. If the process is waiting, it wakes up, calls *function*, then waits again when *function* returns.

If the process is stopped it does not apply *function* to *args* immediately, but will later when it is activated. Normally the :interrupt operation returns immediately, but if the process's stack group is in an unusual internal state :interrupt may have to wait for it to get out of that state.

29.5 Process Flavors

These are the flavors of process provided by the system. It is possible for users to define additional flavors of their own.

si:process *Flavor*

This is the standard default kind of process.

si:simple-process *Flavor*

A simple process is not a process in the conventional sense. It has no stack group of its own; instead of having a stack group that gets resumed when it is time for the process to run, it has a function that gets called when it is time for the process to run. When the wait-function of a simple process becomes true, and the scheduler notices it, the simple process's function is called in the scheduler's own stack group. Since a simple process does not have any stack group of its own, it can't save control state in between calls; any state that it saves must be saved in data structure.

The only advantage of simple processes over normal processes is that they use up less system overhead, since they can be scheduled without the cost of resuming stack-groups. They are intended as a special, efficient mechanism for certain purposes. For example, packets received from the Chaosnet are examined and distributed to the proper receiver by a simple process that wakes up whenever there are any packets in the input buffer. However, they are harder to use, because you can't save state information across scheduling. That is, when the simple process is ready to wait again, it must return; it can't call **process-wait** and continue to do something else later. In fact, it is an error to call **process-wait** from inside a simple process. Another drawback to simple processes is that if the function signals an error, the scheduler itself is broken and multiprocessing stops; this situation can be repaired only by aborting, which blasts the process. Also, when a simple process is run, no other process is scheduled until it chooses to return; so simple processes should never run for a long time without returning.

Asking for the stack group of a simple process does not signal an error, but returns the process's function instead.

Since a simple process cannot call **process-wait**, it needs some other way to specify its wait-function. To set the wait-function of a simple process, use **si:set-process-wait** (see below). So, when a simple process wants to wait for a condition, it should call **si:set-process-wait** to specify the condition, then return.

si:set-process-wait *simple-process wait-function wait-argument-list*

Sets the *wait-function* and *wait-argument-list* of *simple-process*. See the description of the **si:simple-process** flavor (above) for more information.

29.6 Other Process Functions

process-enable *process*

Activates *process* by revoking all its run and arrest reasons, then giving it a run reason of **:enable**.

process-reset-and-enable *process*

Resets *process*, then enables it.

process-disable *process*

Stops *process* by revoking all its run reasons. Also revokes all its arrest reasons.

The remaining functions in this section are obsolete, since they simply duplicate what can be done by sending a message. They are documented here because their names are in the **global** package.

process-preset *process function* &rest *args*

Sends a **:preset** message.

process-reset *process*
Sends a :reset message.

process-name *process*
Gets the name of a process, like the :name operation.

process-stack-group *process*
Gets the current stack group of a process, like the :stack-group operation.

process-initial-stack-group *process*
Gets the initial stack group of a process, like the :initial-stack-group operation.

process-initial-form *process*
Gets the initial form of a process, like the :initial-form operation.

process-wait-function *process*
Gets the current wait-function of a process, like the :wait-function operation.

process-wait-argument-list *p*
Gets the arguments to the current wait-function of a process, like the :wait-argument-list operation.

process-whostate *p*
Gets the current who-line state string of a process, like the :whostate operation.