

Multiprocessing

This document contains the following sections:

[1.0 Multiprocessing introduction](#)

[1.1 Data types added to standard Common Lisp](#)

[1.2 The maximum number of simultaneous processes](#)

[2.0 Variables and functions](#)

[2.1 Threads and processes](#)

[2.2 Native threads and foreign functions](#)

[2.3 Waiting for input from a stream](#)

[2.3.1 mp:process-wait vs mp:wait-for-input-available](#)

[3.0 Process functions and variables](#)

[4.0 Processes and their dynamic environments](#)

[4.1 Lisp listeners and special variable bindings](#)

[5.0 Gates](#)

[6.0 Queues](#)

[7.0 Process locks](#)

[8.0 cl:sleep and minimum sleeping time](#)

[9.0 A simple example of multiprocessing](#)

[Appendix A. Wide binding](#)

[Appendix A.1. The bindstack index](#)

[Appendix B. Processes and the runtime analyzer](#)

1.0 Multiprocessing introduction

Allegro CL has extensions to support multiprocessing within a single executing Lisp image.

All processes share the same Lisp address space, sometimes called the *Lisp world*. Each process has its own execution stack (i.e. "call" stack) and dynamic variable bindings. All processes share everything else in the Lisp world, including packages, streams, global function bindings, and global values of special variables. (The global value of a special variable is its outermost value, outside any dynamic binding.) The Allegro CL compiler, interpreter, top level, and other subsystems all lambda bind their necessary special variables, and Allegro CL code itself is reentrant. Therefore, multiple processes can correctly maintain any number of correct and independent ongoing computations simultaneously within

the single Lisp world.

There are two implementations of multiprocessing in Allegro CL, the **native threads** model and the **virtual threads** model. **:os-threads** appears on the `*features*` list of the implementation using the native threads model and does not appear on the `*features*` list of the implementation using the virtual threads model. In earlier releases, the programming interface of the two models had significant differences. These differences are almost all gone in release 7.0.

In the **native threads** model, each process within Lisp is implemented by an operating system thread and management tasks such as scheduling are done by the operating system. In the **virtual threads** model, all processes are implemented internally by Lisp.

Although there are many subtle differences between these two implementations, in actual practice it is very rare for actual application code to have any dependency on these differences. Most code developed on one will run without modification on the other. This has been verified during the porting of several preexisting multiprocess-intensive subsystems.

Most symbol names for multiprocessing are in the `multiprocessing` package, nicknamed `mp`. Programmers must preface symbols with the package prefix `mp:` or execute

```
(use-package :multiprocessing)
```

before the first reference to any multiprocessing function. Depending how Lisp is installed the multiprocessing module may not be loaded into the initial Lisp system. To guarantee it is available, execute the form

```
(require :process)
```

before calling any multiprocessing functions, or place this top-level form

```
(eval-when (:compile-top-level :load-top-level :execute) (require :process))
```

near the start of any source file which uses multiprocessing.

1.1 Data types added to standard Common Lisp

Multiprocessing adds these user-visible data types (defstructs) to standard Common Lisp:

1. A *process* corresponds to the usual operating system notion of a thread. In the `:os-threads` implementation each process is associated with a single Operating System thread. In the non `:os-threads` implementation each process is associated with a virtual thread.
 2. A *process-lock* provides a mechanism for process synchronization. A process-lock is either free or seized by some process. Any process trying to seize a seized lock will block until the lock is free. See [Section 7.0 Process locks](#).
-

1.2 The maximum number of simultaneous processes

There can be only 350 simultaneously running processes. That is really a limit on the number of simultaneous threads: running processes are associated with threads. The garbage collector frees dead threads, so there can be many more than 350 threads used while an application runs, just no more than 350 used at one time.

If a process tries to run when the limit of running processes/threads is reached, an error will be signaled.

2.0 Variables and functions

The descriptions below provide only a brief introduction. Please follow the links to the individual description pages for details.

Name	Arguments	Notes
<code>symeval-in-process</code>	<i>symbol thread</i>	<p>This function returns two values. The first is the value of the special symbol in the given thread (which may be the current thread). It only looks for actual bindings on the thread; if the symbol has a global value but is not bound on the thread, the global value is not returned.</p> <p>The second returned value describes the status of the</p>

		binding. <code>t</code> is returned if the symbol is bound on the thread, <code>nil</code> if the symbol has no binding, and <code>:unbound</code> if there is a binding which has been subjected to <code>makunbound</code> . In the latter two cases, the first returned value is <code>nil</code> .
sys:global-symbol-value	<i>symbol</i>	This function returns two values. The first is the global value for the special variable named by <code>symbol</code> , ignoring any bindings on the current thread and the second is <code>t</code> if the variable has a value (in the sense of boundp). Otherwise the first value will be <code>nil</code> and the second will be the symbol <code>:unbound</code> .
profile-process-p	<i>process</i>	This function returns the value of the <code>profile-process-p</code> flag for the thread specified by <code>process</code> . If the value of the flag is <code>non-nil</code> , then the space and time runtime analyzers will collect samples when this process is executing.

2.1 Threads and processes

The process object implements the abstraction of independent processes running within the same Lisp world. Process objects are CLOS instances, allowing the application to define subclasses for whatever purpose. Processes are implemented on top of threads; each active process is associated with a particular thread. Thread objects are simple static structures that contain the information used to schedule. Active process threads are managed by the Operating System in native thread implementations and within Lisp in virtual thread implementations. `sys:*current-thread*` is bound in each thread to the thread object representing this (native or virtual) thread.

A process object is implemented as a CLOS class. Some of its slots are meaningful to user code, but

except for those explicitly noted they should be treated as read only.

Whether and when a process runs is controlled by several mechanisms. First the process' initial function and arguments must be specified, and then the process must be 'reset'. (The **process-preset** function combines these operations.)

Second, a process maintains two lists: its *run-reasons* and its *arrest-reasons*. These lists can contain Lisp objects of any type (whether or not each list is empty is the only relevant issue). For a process to be considered for execution, it must have at least one object on its run-reasons list and no objects on its arrest-reasons list.

Finally, a process that needs to wait for some arbitrary condition does so using the **process-wait** function. This function specifies a function and arguments. When the OS considers a waiting process for running, it causes the wait-function to be applied to the wait-arguments (in the process environment). If a non-null value is returned the process runs and the call to **process-wait** returns. **process-wait** is most efficient when the wait function is one of **gate-open-p**, **read-no-hang-p**, **write-no-hang-p**, or **stream-listen**, but any appropriate function can be used.

It is useful to define some terms for process states. A process is *active* if it has been preset, has not completed, has at least one run reason, and has no arrest reasons; otherwise, it is *inactive*. Active processes are further divided into two classes. An active process is *waiting* if it has executed a **process-wait** that has not yet completed. An active process is *runnable* if it is not waiting. In addition, the process actually running at any time is called the *current* process.

On Windows, even though a process may be waiting, it still typically watches for messages are processes operating system messages that require a response (because Windows expects threads to do this). However, there are some rare circumstances where this message handling should be suppressed. The macro **mp:with-message-interrupts-disabled** will do this.

Processes run until complete or interrupted. When a process is run, it is given an amount of processor time; when that time expires the operating system interrupts the process and looks about for other processes which can be run.

- Each unblocked process is examined and its priority is noted.
- The highest priority processes are considered (there may be several with the same high priority, or there may just be one).
- The process itself ensures that it can run, examining its arrest reasons (cannot run if there are any) and its run reasons (cannot run unless there is one).
- If the process has a wait function, it runs it. If the wait function returns nil, the process cannot run.
- If the process cannot be run, it checks to see if there are priority messages that must be processed, and if so processes them. (On some operating systems, every thread must respond to a priority message; all other threads will block until all priority messages are handled. This check is

to ensure that the system does not block because of such messages.)

- Once a process is determined to be runnable, it looks at its pending interrupts. (These are lisp interrupts queued via **process-interrupt**, not Operating System interrupts.) All lisp interrupts are processed.
- After all lisp interrupts are processed, normal process computation continues until another interrupt is received from the Operating System.

A process has two parameters to control scheduling. Its *priority* is an integer indicating its scheduling priority, higher numbers request more frequent processing. Its *quantum* indicates the minimum time (in the range 0.1 to 20.0 seconds inclusive) the process should be allowed to run before a clock tick might suspend it, assuming it does not give up execution voluntarily. This is to prevent thrashing between processes, particularly those that might have deep stacks.

2.2 Native threads and foreign functions

In the virtual thread (non **:os-threads**) model, foreign code is not interruptible and Lisp code cannot run until the foreign code returns control to Lisp, either by completing or by calling back to Lisp. In the native threads (**:os-threads**) model, Lisp code on one process may run while foreign code runs in another process. This means that certain properties of foreign code may no longer hold. Among them these are the most significant:

- Lisp pointers passed to foreign code are not guaranteed to be valid by default. In earlier releases, because Lisp code never ran while foreign code was running, a Lisp pointer passed to foreign code was guaranteed to be valid until the foreign code completed or called back to Lisp. In the **:os-threads** model, because Lisp code and foreign code can run in different OS processes, Lisp code may be run concurrently with foreign code. A garbage collection may move Lisp objects, rendering invalid pointers held by the foreign code. Each foreign function's definition specifies whether a lisp process calling that function retains exclusive control of the heap or releases the heap so that other lisp processes can run. Releasing the heap can improve overall application concurrency, especially if long-running or blocking foreign functions are involved, but it also introduces the danger that a lisp string passed as an argument may become invalid before the called code is through with it. Foreign function definitions by default perform non-releasing calls. A special keyword argument must be used in the **def-foreign-call** if a heap-releasing linkage is desired. We recommend storing all values used by both Lisp and foreign code in foreign (not garbage-collected) space or dynamically allocated on the stack, whenever those values may be used in heap-releasing calls. See *Releasing the heap when calling foreign functions* in *foreign-functions.htm* for more information.
- It is possible for a foreign function called from Lisp to explicitly start computation in additional threads, as supported by the OS (by having foreign code make the appropriate system calls to start the threads). One could imagine any of these new threads using the foreign function

interface to invoke a Lisp function defined as foreign-callable (see **defun-foreign-callable**). In an Allegro CL with the non **:os-threads** model of multiprocessing, doing this is almost certain to have disastrous consequences. It is wholly unsupported but there is no protection in the foreign function interface to prevent it from happening. The only legitimate calls to a foreign-callable function will occur in the Lisp's own thread of control, as call-backs from foreign code that was itself called from lisp.

In an **:os-threads** Allegro CL, however, it is legitimate for a thread started outside Lisp to call into Lisp via any foreign-callable function. Some extra work has to be done to create a Lisp process to represent that thread within the lisp world. That extra work is performed by a "customs agent" process that is started automatically (no specific programming is required).

- If foreign code spawns any new threads, or if it allows another thread to run, and the other thread attempts to call back into lisp, it will have to wait for the lisp heap. The danger is that the original thread may be waiting for results from its partner thread, but it has not yet given up the heap (this constitutes a deadlock situation). If this situation holds (foreign code does spawn new thread which call back into Lisp), `:when-ok` is the appropriate value for the *release-heap* argument to **def-foreign-call**. The default value for *release-heap* is `:never`, so in this situation, the value `:when-ok` must be explicitly specified.

2.3 Waiting for input from a stream

This section deals with the situation where no process can run until input arrives from outside of Lisp, typically via a stream. The issue is how to have Lisp not waste machine cycles waiting for the arrival of input, yet have Lisp wake up in a timely manner when the input becomes available.

Each waiting process has an associated wait function. Whenever the process' thread is given processor time, it executes the wait function and if the result is still false, it immediately releases the processor back to the OS. If the OS were to cycle repeatedly through all the waiting processes, the processor will be perpetually busy even though no useful work is being done. This might be reasonable on a dedicated machine -- the wasted processor cycles would not be valuable as there would be nothing else for the processor to do. But this would be bad computing citizenship on a host with other processes unrelated to Lisp, since it would consume processor cycles and paging bandwidth that might be in short supply. So Lisp tries to conserve machine resources by keeping waiting process threads completely quiescent when nothing is runnable, and allowing the OS to give threads processor time to check their wait functions only when something may have changed that could affect the result of one or more wait functions. However, Lisp needs a little help to do this.

There are only three things that can cause a wait function to return true after previously having returned `nil`: (1) some other running Lisp process has changed some piece of state in the Lisp world that the

wait function tests; (2) an asynchronous signal arrives; or (3) input becomes available on a file descriptor (generally associated with a Lisp stream) that the wait-function tests.

Case (1) requires that wait functions be tested periodically whenever (or at least immediately after) any Lisp process actually runs. The operating system thread management does this automatically. But what should happen when absolutely no processes are runnable? We want Lisp to stop running completely until either (2) or (3) happens.

When a process finds that its wait function is not yet satisfied, it releases the CPU by performing the equivalent of a Unix `select()` on the set of "interesting" file descriptors. This causes the process to block. The OS will not give the associated thread any CPU time until a signal arrives (2) or input becomes available on one of the interesting file descriptors (3). The process can run its wait-function again to determine whether it has actually become unblocked.

Unfortunately, the process machinery has no way to correlate wait functions with OS file descriptors, such that input becoming available on the file descriptor would (or might) cause the wait-function to return true. The system needs to be told explicitly that a file descriptor is interesting to some wait-function. The low-level input handlers (e.g., for `read-char`) for Allegro CL streams do this automatically, but any user implementing custom streams and/or making a low-level foreign-function interface to (for example) the underlying operating-system socket interface will need to write input functions in such a way that they inform the process machinery about input file descriptors. The description of **wait-for-input-available** describes how this is done.

2.3.1 `mp:process-wait` vs `mp:wait-for-input-available`

The purpose of **wait-for-input-available** is to wait on one or more input streams. **wait-for-input-available** takes a wait function just like **process-wait**, but before suspending the calling process it carefully records the input file descriptors for each stream or file descriptor argument. While the calling process is blocked inside the call to **wait-for-input-available**, these file descriptors will be passed to `select()` so that available input will immediately return from `select()` and awaken this thread.

If **process-wait** is used instead of **wait-for-input-available**, the thread may fail to notice when input becomes available and not run the wait function until some other possibly-unrelated interrupt or input causes the entire Lisp image to wake up and eventually run through all wait functions again.

3.0 Process functions and variables

The descriptions below provide only a brief introduction. Please follow the links to the individual description pages for details.

Name	Arguments	Notes
<code>*all-processes*</code>	[variable]	The value of this variable is a list of all processes that have ever been created and have never completed or been killed.
<code>*current-process*</code>	[variable]	The value of this variable is the process currently running (:os-threads) or which the scheduler is currently running (non :os-threads). In non :os-threads implementations, nil is returned if the scheduler itself is running.
<code>*default-process-quantum*</code>	[variable]	Default quantum given to each process.
start-scheduler	nil	<p>:os-threads: initializes multiprocessing (the function is misnamed since there is no scheduler but used for consistency with the non :os-threads implementation.)</p> <p>Non :os-threads: start the scheduler process and initialize multiprocessing.</p>

make-process	<i>&key name reset- action run-reasons arrest-reasons priority quantum resume-hook suspend- hook initial- bindings message- interrupt-function stack-allocation run- immediately</i>	<p>This function returns a new process object, but does nothing about making it runnable. Follow the link to the full description for details.</p>
process-initial-bindings	<i>process</i>	<p>This slot of a process stores an <i>alist</i> of initial special bindings which are established in a process when it is started. The value may be set with setf.</p>
process-property-list	<i>process</i>	<p>The <i>property-list</i> slot of a process implements a generalized property list as a convenient place to store additional information about a process.</p>
process-resume-hook	<i>process</i>	<p>It is normal for execution of a process to be interrupted many times. This is transparent to the process and usually it is not necessary for the process to know when its execution is suspended and resumed.</p> <p>However, if these slots are non-nil, they should be functions of no arguments which are called on the process' <i>stack-group</i> or <i>thread</i> each time the execution is suspended or resumed (but not when the process is first started or when it is killed).</p>
process-suspend-hook		

process-run-function	<i>name-or-keywords function &rest args</i>	This function does a make-process , then presets the new process with function and args. The first argument is either a string, which is the name of the process, or is a list of keyword arguments accepted by make-process . The new process is returned. By default, the process is killed when and if it completes.
process-run-restartable-function	<i>name-or-keywords function &rest args</i>	This function is just like process-run-function (just above), but automatically provides a <code>:reset-action</code> argument of <code>t</code> . The process thus started will restart if it is reset or completes.
process-enable	<i>process</i>	Makes process active by removing all its run and arrest reasons, then giving it a single run reason of <code>:enable</code> .
process-disable	<i>process</i>	This function makes process inactive by revoking all its run and arrest reasons. The effect is immediate if a process disables itself.
process-reset	<i>process &optional no-unwind kill</i>	This function causes <i>process</i> when it next runs to throw out of its present computation, if any, then apply its initial function to its initial argument.

process-preset	<i>process function &rest arguments</i>	This function sets the initial function and arguments of process, then resets any computation in progress in it. This does not make process active if it was not already active.
process-kill	<i>process</i>	This function resets the process to unwind it, then removes it from consideration by the scheduler and from the <code>*all-processes*</code> list.
process-interrupt	<i>process function &rest args</i>	This function forces process to apply function to args when it next executes. When function returns, the original computation of process continues. If process is waiting when interrupted, it runs the interrupt function and then continues waiting. If process is not active, process-interrupt makes it active for the interrupt function, then makes it inactive again.
process-name	<i>process</i>	This function returns the name of process, which must be a string. This value may be changed with <code>setf</code> .
process-name-to-process	<i>name &key :abbrev</i>	This function returns the process whose process-name is name. name must be a string. If the <code>:abbrev</code> keyword argument is specified non-nil, then name is matched to the beginning of each process-name to find a match.

process-thread	<i>process</i>	Returns the thread associated with <i>process</i> .
process-initial-form	<i>process</i>	This function returns a cons of the initial function of process and its argument list.
process-wait-function	<i>process</i>	This function returns the function used to determine when a waiting process becomes runnable.
process-wait-args	<i>process</i>	This function returns the list of arguments passed to the wait-function of <i>process</i> .
process-run-reasons	<i>process</i>	This function returns the list of run-reasons for <i>process</i> .
process-arrest-reasons	<i>process</i>	This function returns the list of arrest-reasons for <i>process</i> .
process-add-run-reason	<i>process object</i>	This function adds object to the list of run-reasons for process.
process-add-arrest-reason	<i>process object</i>	This function adds object to the list of arrest-reasons for process.
process-revoke-run-reason	<i>process object</i>	This function removes object from the list of run reasons for <i>process</i> .
process-revoke-arrest-reason	<i>process object</i>	This function removes object from the list of arrest reasons for process.
process-runnable-p	<i>process</i>	These functions return t if, respectively, <i>process</i> is runnable or active. A process is active if it has been reset and

process-active-p		not yet completed, and has at least one run reason and no arrest reasons. It is runnable if it is active and not waiting.
process-priority	<i>process</i>	This function returns the priority of <i>process</i> . It defaults to 0 and may be set to any fixnum with setf .
process-quantum	<i>process</i>	This function returns the quantum for <i>process</i> . The quantum may be specified when the process is created; it defaults to the value of <code>*default-process-quantum*</code> and may be set to any real value between 0.1 and 20 with setf .
process-whostate	<i>process</i>	This function returns the current who-line string of process.
without-scheduling	<i>&body body</i>	This macro inhibits the OS (in :os-threads) or the scheduler (non :os-threads) from suspending a process involuntarily (asynchronously) during the execution of <i>body</i> . This always works in non :os-threads versions since the scheduler is a Lisp process. However, in :os-threads versions, the OS will run another process if the current process blocks, waits, or executes a process-allow-schedule .

without-interrupts	<i>&body body</i>	This macro executes <i>body</i> protecting against any handling of asynchronous interrupts. Execution of <i>body</i> is guaranteed to complete without any other process running, or any asynchronous interrupt being dispatched, unless the process does something to block or otherwise explicitly yield to the scheduler (e.g., with process-allow-schedule). It is an error to call a heap-releasing foreign function within the scope of without-interrupts .
disallow-scheduling	[variable]	<p>This special variable is bound to <i>t</i> whenever multiprocessing scheduling is disabled. For example, the system binds this variable to <i>t</i> during the execution of the forms within a without-scheduling form.</p> <p>This variable should be treated as read-only and should never be set or bound by user code.</p>
process-sleep	<i>seconds &optional whostate</i>	process-sleep suspends the current process for at least the number of seconds specified. That number may be any non-negative, non-complex number. While the process sleeps, other processes are allowed to run. The <i>whostate</i> (default "Sleep") is a string which temporarily replaces the process' <i>whostate</i> during the sleep.

		<p>When multiprocessing is initialized, Common Lisp function <code>sleep</code> is changed to be equivalent to <code>process-sleep</code>. Instead of causing the entire Lisp world to suspend execution for the indicated time, only the executing process is suspended. This is usually the desired action.</p>
<p>process-wait</p>	<p><i>whostate function &rest arguments</i></p>	<p>This function suspends the current process (the value of <code>*current-process*</code>) until applying function to arguments yields true. The <code>whostate</code> argument must be a string which temporarily replaces the process' <code>whostate</code> for the duration of the wait. This function returns nil.</p> <p>See the discussion under the headings Section 2.3 Waiting for input from a stream and Section 2.3.1 mp:process-wait vs mp:wait-for-input-available.</p>
<p>process-wait-with-timeout</p>	<p><i>whostate seconds function &rest args</i></p>	<p>This function is similar to process-wait, but with a timeout. The units of time are seconds. The value of <code>seconds</code> may be any real number. Negative values are treated the same as 0. The wait will timeout if function does not return true before the timeout period expires.</p>

wait-for-input-available	<i>streams &key :wait-function : whostate :timeout</i>	This lower-level function extends the capabilities of process-wait and process-wait-with-timeout to allow a process to wait for input from multiple streams and to wait for input from a file.
with-timeout	<i>(seconds . timeout-forms) &body body</i>	This macro evaluates the body as a progn body. If the evaluation of body does not complete within the specified interval, execution throws out of the body and the timeout-forms are evaluated as a progn body, returning the result of the last form. The timeout-forms are not evaluated if the body completes within <i>seconds</i> .
process-allow-schedule	<i>&optional other-process</i>	This function resumes multiprocessing, allowing other processes to run. All other processes of equal or higher priority will have a chance to run before the executing process is next run. If the optional argument is provided, it should be another process.

4.0 Processes and their dynamic environments

The multiprocessing system described in this chapter is a nonstandard but upward-compatible extension to Common Lisp. Although multiprocessing can be used within a single application towards performing a single, integrated task, it is an important design feature that multiprocessing allows multiple unrelated tasks to reside in the same Lisp image. A user might run a window system, several independent Lisp listeners, a Lisp editor, and an expert system application (with network connections to processes in other

hosts) all in the same Lisp world.

For independent tasks to coexist in a single world, the names of functions and special variables in separate systems cannot interfere with one another. This is the problem the package system is intended to solve, and with a little care it is quite possible to keep names separate. However, there are a number of public common-lisp package special variables which are implicitly used by all Lisp applications. Each process may want its own values assigned to these special variables. This section discusses how conflicting demands on these variables can be supported.

For example, according to the Common Lisp standard, the default value of the special variable `*print-base*` is 10. A stand-alone application may assume that if it never changes `*print-base*` its value will be 10. Or, if a stand-alone application always wanted to print in octal, it might set `*print-base*` to 8. The effect on its own calculation is well defined by the Common Lisp standard, but changing the global value of `*print-base*` may have an unfortunate effect on other unrelated applications running in the same Lisp world. If one application changes `*print-base*` what about others that assume it will always have the default value, or want to exercise independent control?

A solution to this problem is to make a process that uses public variables maintain its own bindings for those variables. When a process is first started, a stack-group or thread is created on which it will run. Whenever the process' computation binds special variables, those bindings are recorded on its stack-group or thread. *Binding* hides the value of special variables with (possibly) different values. Inside the scope of a binding, code will only see the bound value, not the original value, of a variable. Special variable bindings are local to the process that creates them and are never seen by other processes.

Note that there is only one global "binding" for a variable. While it always exists and is shared by all processes, if the variable has been bound in the process to another value, code inside that binding in that process will only see the bound value, not the global value. This is important to understand when there are multiple processes. Sometimes it is useful for processes to communicate through the value in a global binding. Other times processes must be protected from seeing the value other processes have bound a variable to.

Assume you have several processes and none of them binds the special variable `*print-base*`. If one process sets the variable, the global binding is the one affected and all of the processes will see the new value. However, any process that binds the variable as a special will not see changes to the global binding while its own binding is in effect, and conversely, any `setq` it does will not affect the value seen by the others.

The multiprocessing system provides a mechanism to allow processes to bind special variables at process startup time. The process-initial-bindings slot of a process is examined when the process is first started. If not nil, it should be an alist of symbols and forms to evaluate for value. The symbols are bound in the process as special variables with the given initial values, effectively wrapping those bindings around the entire execution of that process. If a particular variable appears more than once on

the alist, entries after the first are ignored.

By default, **make-process** and **process-run-function** create processes with a null process-initial-bindings list. Such action is not appropriate for processes which may do arbitrary computations - for instance, a Lisp listener which accepts computations from a user - or for applications which need to be isolated from others. In such cases, the multiprocessing system provides a list of special variables with appropriate default values. The variable `excl:*cl-default-special-bindings*` is bound to that list.

In the **:os-threads** implementation, `*current-process*` is always added to each process' special binding list; when a process is running that process is the value of `*current-process*`. In the non **:os-threads** model the scheduler maintains this as a global variable.

The standard CL stream variables are bound to the value of `*terminal-io*` but that variable itself is not given a dynamic binding. This can cause problems because sometimes `*terminal-io*` may be set to a stream that will signal an error when used (see *debugging.htm*). The variable `excl:*initial-terminal-io*` holds the original `*terminal-io*` stream when Lisp starts up. It may be useful for processes that aren't connected to a usable `*terminal-io*` but wish to produce some output, for example for debugging.

Note that the value forms are evaluated in the dynamic environment of the new process, not the process that created it, and this new dynamic environment has no special bindings in effect. Those value forms that themselves depend upon special variable values, e.g., `cltl1:*break-on-warnings*`, will therefore see the global values of those variables. The intention of this mechanism is that the new process should not inherit variable values from the process that started it without something explicit being done to pass the value. Some other ways to achieve the same end are shown below.

If, for example, you want your new process to share the readtable of the invoking process, putting an entry

```
(*readtable* . *readtable*)
```

on the alist would not work. The value form would be evaluated on the new process, and the global value of `*readtable*` would result. Instead, you should do something like this (note that this is a code fragment - you must add forms where there are suspension points):

```
(process-run-function
  `(:name ...
    :initial-bindings
      ((*readtable* . ',*readtable*)
       ,@excl:*cl-default-special-bindings*))
  ...)
```

Since the `:initial-binding` list is treated as an alist, the first entry shadows all succeeding ones. The effect of the above will be to set the alist cons for `*readtable*` to something like:

```
(*readtable* . (quote #<readtable @ #x504a1>))
```

where the `readtable` object is the `readtable` of the invoking process, and the quote is stripped off by the evaluation when the new process begins.

Using `process-run-function` and related functions it is possible to run any kind of Lisp computation as a separate quasi-parallel process. All processes share the same Lisp world, so the only thing that differentiates one process from another (besides the state of its computation, of course) is the set of special-variable dynamic bindings of its binding stack. For example, in an environment with multiple windows it is possible to run multiple top-level Lisp listeners or other interactive command loops, each process in its own window; `*terminal-io*` would be bound in each process to a stream object connected to its particular input-output window.

There are two ways to create special bindings in a new process. The simple way is just to place the special variable on the lambda list of the process' initial function, and pass the value as an argument (note that this is a code fragment - the suspension points indicate where additional forms are required):

```
(defun my-command-processor (*terminal-io* *page-width*)
  (declare (special *page-width*))
  ...)
```

```
(let ((my-window (create-a-window ...)))
  (process-run-function "Bizarre Command Interpreter"
    #'my-command-processor my-window
    (page-size my-window)))
```

However, this requires the process' initial function to provide for each variable that will be bound. The following more general idiom permits establishment of arbitrary bindings for the new process:

```
(defun establish-bindings (symbol-list value-list function args)
  (progv symbol-list value-list (apply function args)))
```

```
(let ((my-win (create-a-window ...)))
  (process-run-function "Bizarre Command Interpreter"
    #'establish-bindings
    '(*user-name* *phone-number*)
    (list "Fred" "555-1234")
    #'my-command-processor
```

```
(list my-win (window-size my-win)))
```

Here the `establish-bindings` function is *wrapped* around the application of `my-command-interpreter` function to its arguments; while the function is executing it will see bindings of the special variables `user-name` and `phone-number`.

4.1 Lisp listeners and special variable bindings

The problem of starting a real Lisp listener is actually somewhat more complex than the above discussion and sample code fragments suggest. For one thing, the Lisp system defined by Common Lisp depends on a large number of special variables, such as `*readtable*`, `*package*`, and `*print-level*`. Since commands executed by one Lisp listener might side effect some variables and thereby inappropriately affect other processes, each process should maintain separate bindings for these variables. Allegro CL provides a wrapping function, `tpl:start-interactive-top-level`, which automatically binds to reasonable default initial values all the special variables in the Lisp system. Within a call to **`tpl:start-interactive-top-level`**, a read-eval-print loop can be started with **`top-level-read-eval-print-loop`**.

Thus, the Allegro CL idiom for running a standard Lisp listener communicating with `*terminal-io*` (this might be done in a `excl:*restart-app-function*`) looks something like this:

```
(tpl:start-interactive-top-level *terminal-io*
 #'tpl:top-level-read-eval-print-loop
 nil)
```

And the idiom for running a standard Lisp listener inside a window looks something like this:

```
(process-run-function "My Lisp Listener"
 #'tpl:start-interactive-top-level
 my-window-stream
 #'tpl:top-level-read-eval-print-loop
 nil)
```

Entry to **`tpl:top-level-read-eval-print-loop`** establishes additional bindings for certain variables used by the top level loop. These bindings are established inside any bindings established by **`start-interactive-top-level`**. The variables and initial binding values are taken from the alist bound to the variable `*default-lisp-listener-bindings*`.

`tpl:top-level-read-eval-print-loop` also provides a hook to customize individual Lisp listeners. It calls the function which is the value of `*top-level-read-eval-print-loop-wrapper*`, with two

arguments, the internal function to invoke the read-eval-print loop, and the argument it requires.

In this rather trivial example, we define the wrapper so that the command character (the value of `*command-char*`, initially `#\:`) is `#\$`. Of course, in a real example, something more complex would be done, but the form would be similar to this example.

```
(defun my-lisp-listener-wrapper (function args)
  (let ((tpl:*command-char* #\$))
    (apply function args)))

;; After we evaluate the following forms, the command character will
be $
;; (rather than :) in any new lisp listener process.

(setq tpl:*top-level-read-eval-print-loop-wrapper*
  'my-lisp-listener-wrapper)
```

What about setting up personal customizations? When Allegro CL is first started it searches for `.clinit.cl` files as described in *startup.htm*. The typical purpose of a `.clinit.cl` file is to load other files and to define personalized functions and top-level commands, but it is possible for a `.clinit.cl` file to **setq** special variables. It is important to understand what it means to do so.

A `.clinit.cl` file is loaded into a Lisp before multiprocessing is started and even before the initial Lisp listener is created. If a `.clinit.cl` file sets any special variables, the value affected will (in general) be the global value of the special. If the purpose is to customize Lisp for some particular application this is probably the wrong way to do it. Someday an unrelated system may be loaded into the same Lisp world which may depend on reasonable default global variable values. Furthermore, the various default binding mechanisms described above will generally keep the global value even from being seen.

For example, if it is necessary to set `*print-escape*` to `nil` for some application, it is better for the application to set up its own binding of the variable in the application code itself, or if that is impossible, to have the code that starts the process wrap a binding of the variable around execution of the application with the mechanisms illustrated above. The worst way is to `setq` `*print-escape*` in a `.clinit.cl` file. Processes that assume the documented Common Lisp standard simply might not work properly if `*print-escape*` is `nil`.

A dynamic binding - that is, a 'location' where a value is stored - cannot be shared between processes. Each process may store the same (i.e. eql) value in its binding, but if one process does a `setf` of `*print-base*` it cannot affect the value of the binding of `*print-base*` seen by another process. If that value is a Lisp object that can be modified, side effects to that object will obviously be seen by both processes. Of the variables listed above, those that typically hold obviously side-effectable objects are `*package*`, `*read-table*` and the several stream variables. Numbers (such as the value of `*print-base*`) and the

boolean values `t` and `nil` (such as might be the value of `*print-escape*`) are not objects that can be side affected.

Unlike the standard common-lisp package special variables, it is quite reasonable to place in your `.clinit.cl` file personal customizations for top-level Lisp listener variables documented in `top-level.htm`, such as `*prompt*`, that are not bound per process. No standard Common Lisp code should depend on these.

Since the multiprocessing system tries hard to insulate variable bindings between processes, the macro `tpl:setq-default` is provided to make it easier for a user to change the default value of some standard variable when that is what is really desired. It is intended primarily for inclusion in `.clinit.cl` files.

setq-default is convenient for simple customization and suffices for simple environments. However, because it sets the global value of a symbol which is seen by *all* processes, it may not be appropriate in Lisp environments where the user may not have control over the needs of some processes. In such circumstances it may be preferable not to change a global symbol value in a `.clinit.cl` file with **tpl:setq-default**. Instead, users may just push a new cons onto the `*default-lisp-listener-bindings*` alist. Such action will have much the same effect with regard to any code run inside a Lisp listener but will not affect non-listener processes.

5.0 Gates

A gate is an object with two states, open and closed. It is created with **make-gate**. Its state can be open (see **open-gate**) or closed (see **close-gate**) and can be tested with the **gate-open-p** function. A waiting process whose wait function is `#'gate-open-p` places much lower cpu demands on an os-threaded lisp than a waiting process that has a general wait function. That is because **process-wait** threading code recognizes this function (either the name or the function itself) specially and knows how to treat it much more efficiently than other arbitrary Lisp functions.

Therefore, it is often worthwhile to add gates to a program (expending the additional programming effort) to reduce the overhead of blocked processes, even though gates might not be logically necessary. The example just below illustrates this. Logically, the gate object is not necessary, in that the example also depends on process-locks and could be written more simply using just the **process-lock**. But the example is more efficient because it uses gates.

Here is an example of using a gate to control execution.

```
;; an instance of xqueue represents a server and its queue.
;; a gate is used to indicate that data needing to be handled
;; is on the queue.
```

```

(defstruct xqueue
  data
  lock
  gate
  process)

;; anyone with access to an xqueue item can add to its work queue by
calling
;; xqueue-add. After the data is added, the associated gate is opened
;; (an open gate indicates that data is waiting to be handled).

(defun xqueue-add (xq item)
  (mp:with-process-lock ((xqueue-lock xq))
    (setf (xqueue-data xq) (nconc (xqueue-data xq) (list item))))
  (mp:open-gate (xqueue-gate xq)))
  item)

;; create-xqueue-server starts a server process running and returns
;; the associated xqueue item to which work items can be queued.
;; The server process calls server-function on each element it
retrieves
;; from the queue. When server-function returns :exit, the server
;; process exits.
;;
;; note that the main loop (the process that is set to
;; the value of (xqueue-process xq)) waits on the gate being open
;; (indicating unhandled data is present) and closes the gate when
;; all available data is handled.

(defun create-xqueue-server (server-function &key (name "Server"))
  (let ((xq (make-xqueue)))
    (setf (xqueue-lock xq) (mp:make-process-lock)
          (xqueue-gate xq) (mp:make-gate nil)
          (xqueue-process xq)
          (mp:process-run-function
            name
            #'(lambda (lxq sf)
                (loop
                 (mp:process-wait "Waiting for data"
                                   #'mp:gate-open-p
                                   (xqueue-gate lxq))
                 (let (e run)

```

```

      (mp:with-process-lock ((xqueue-lock lxq))
        (if* (null (xqueue-data lxq))
          then (mp:close-gate (xqueue-gate lxq))
          else (setq e (pop (xqueue-data lxq))
                    (setq run t)))
        (when (and run (eq :exit (funcall sf e)))
          (return))))
    xq server-function))
xq))

```

6.0 Queues

Conceptually, a queue is a first in, first out (FIFO) list. (Note that queues may not be implemented as lists, but details of its actual implementation are not necessary for understanding the concept.)

Enqueuing an object is conceptually appending an object to the end of the list, dequeuing an object is conceptually equivalent to returning the car (first element of the list and replacing the list with its cdr).

Queues are often useful, and because certain Allegro CL features required queues, we decided to make the queue implementation public. The advantage of using queues over using lists (and appending to and popping from that list) are:

- The **enqueue** and **dequeue** operations are atomic in a multiprocessing sense: once started, the operations will complete without interruption.
- Dequeuing is provided with a waiting facility, so a process that tries to dequeue an object from a queue will (optionally) wait, if the queue is empty, until something is placed on it.

Queues are instances of the class `queue`. Create a queue with `(make-instance 'mp:queue)`. Objects are added to queues with the **enqueue** generic function and removed from queues with the **dequeue** generic function. Queues have no intrinsic size limit.

7.0 Process locks

A *process-lock* is a defstruct which provides a mechanism for interlocking process execution. Lock objects are created with **make-process-lock**. A process-lock is either *free* or it is *seized* by exactly one process. When a process is seized, a non-`nil` value is stored in the lock object (in the slot named

locker). Usually this is the process which seized the lock, but can be any Lisp object other than `nil`. Any process which tries to seize the lock before it is released will block. This includes the process which has seized the lock; the **with-process-lock** macro protects against such recursion.

The locker slot (accessed with **process-lock-locker**) indicates whether the lock is seized or free. If the value of the locker slot is `nil`, the lock is free. If it is non-`nil`, the lock is seized. Both **process-lock** and **process-unlock** take a lock-value first optional argument. This argument defaults to the current process (the value of `*current-process*`). This value, whatever it is, is stored in the locker slot of the lock object by **process-lock**. When **process-unlock** is called, the lock-value argument is compared to the value in the locker slot. If the values are the same, the lock is unlocked. If the values are different, **process-unlock** signals an error.

The normal operation is as follows. Suppose that a lock object has been created and is the value of the variable `my-lock`:

```
(setq my-lock (mp:make-process-lock :name "my-lock"))
```

Suppose as well that at least two processes are running, the FOO process and the BAR process. When the FOO process is current, **process-lock** is called on `my-lock`:

```
(mp:process-lock my-lock)
```

Now the value in the locker slot is `#<FOO process>`. Now **process-unlock** is called in the FOO process

```
(mp:process-unlock my-lock)
```

The `#<FOO process>` is passed as the value of the lock-value optional argument, and, since it matches the value in the locker slot, the lock is unlocked and the value of the locker slot is set to `nil`.

Meanwhile, suppose in process BAR, **process-lock** is called:

```
(mp:process-lock my-lock)
```

If this call occurs while the lock is seized by the FOO process, the BAR process blocks (waits) until the lock is freed and then itself seizes the lock (we assume no other process is waiting for the lock). As soon as the FOO process gives up the lock, the call to **process-lock** in the BAR process completes, with the value in the locker slot being set to `#<BAR process>`.

Both **process-lock** and the **without-scheduling** macro protect a segment of code from interleaving execution with other processes. Neither has significant execution overhead, although **without-scheduling** is somewhat more efficient. However, the mechanisms have different ranges of applicability. A process-lock blocks only those other processes which request the same lock; **without-scheduling**

unconditionally blocks all other processes, even those completely unrelated to the operation being protected. This might include high-priority processes that need to field interrupts with low latency. Therefore, the **without-scheduling** macro should not be used around a code body that might require significant time to execute.

The descriptions below provide only a brief introduction. Please follow the links to the individual description pages for details.

Name	Arguments	Notes
make-process-lock	<i>&key name</i>	This function creates a new lock object. The value of the : name keyword argument should be a string which is used for documentation and in the whostate of processes waiting for the lock. (There are additional keyword argument for internal use not lized. They should not be set by user code.)
process-lock	<i>lock &optional lock-value whostate timeout</i>	This function seizes lock with the value lock-value (which must be non-nil).
process-unlock	<i>lock &optional lock-value</i>	This function unlocks lock, setting the value in the locker slot to nil. The value of the locker slot of the lock must be the same as the lock-value argument. If it is not, an error is signaled.
process-lock-locker	<i>lock</i>	This function returns the value of the locker slot of lock. This value is usually the process holding the lock, but can be any Lisp value. If the value is nil, the lock is not locked.

process-lock-p	<i>object</i>	Returns true if object is a lock (as returned by make-process-lock) and returns nil otherwise.
with-process-lock	<i>(lock &key norecursive) &body body</i>	This macro executes the body with lock seized.

8.0 cl:sleep and minimum sleeping time

While not strictly a multiprocessing issue, **cl:sleep** functionality is actually implemented as part of the multiprocessing (though it is in a module always present in a runnign Lisp).

A call to **cl:sleep** is in fact effective a call to **mp:process-sleep** (with the *whostate* argument set to "sleeping"). **cl:sleep** requires a non-negative argument. It will error when passed a negative argument.

As to how short a time one can sleep, that is the minimum effective argument to sleep is an internal parameter in Allegro CL. For a variety of reasons, mostly historic, the initial value is 75 (meaning 75 milliseconds). To allow for shorter waiting times, you can change the internal value. This form:

```
(sys::thread-control :clock-event-delta)
```

returns the value the system is currently using, as an integer number of milliseconds. (Note that **sys::thread-control** is an internal function and is not further documented. It should not be used for any purpose other than determining or changing the minimum sleep time.) The form is a setf-able location. To make the delta 10 milliseconds, evaluate this:

```
(setf (sys::thread-control :clock-event-delta) 10)
```

Reasonable values range from 0 to 75. The value is not saved in a **dumplisp** image. Each fresh invocation of Lisp must reset it if something other than the default value is desired.

9.0 A simple example of multiprocessing

The example below can be loaded into Lisp either compiled or interpreted. It shows simple use of **mp:process-run-function** and **mp:with-process-lock**. Three parallel processes compute the number of trailing

zeroes in factorial N for different ranges of N.

```
(in-package :cl-user)
;; in the IDE (in-package :cg-user)
(require :process)

(defun factorial (n) (if (< n 2) n (* n (factorial (1- n)))))

;; This lock is used to prevent output interleaving.
(defvar moby-output-lock (mp:make-process-lock))

;; Print to the stream the number of trailing
;; zeros in (factorial n) from n=from up to n=to.
;; This is a *very* inefficient way to do this computation,
;; but the point is to make it run slow enough to see.

(defun process-test (stream from to)
  (do ((n from (1+ n))
      ((>= n to))
      (do ((x (factorial n) (/ x 10))
          (zeros -1 (1+ zeros)))
          ((not (integerp x))
           (mp:with-process-lock (moby-output-lock)
            (format stream "factorial(~d) has ~d trailing zeros~%"
                      n zeros)))))))

;; This starts three processes in parallel.
;; The original Lisp listener returns immediately,
;; and will accept types forms while the other processes run.

(defun moby-process-test ()
  (mp:process-run-function "Test 1" #'process-test t 400 440)
  (mp:process-run-function "Test 2" #'process-test t 440 470)
  (mp:process-run-function "Test 3" #'process-test t 470 400)
  t)

;; Make sure factorial itself is compiled
;; because large factorials exceed the interpreter's stack.

(unless (compiled-function-p #'factorial) (compile 'factorial))

(format t "Type (moby-process-test) to test multi-processing.~%")
```

Appendix A: Wide binding

The value of a variable depends on its current binding. In a multiprocessing environment, the current binding can be different in different processes. Programmers must be aware of the fact that they must establish bindings in different processes for variables for which different processes require different values. For example, a process printing a value to a file for debugging purposes may want `*print-length*` and `*print-level*` both to be `nil` so that all the information is printed, but a listener process may want those variables to be small integers so that the user is not overwhelmed with unneeded data. Without some form of per-process binding, the process doing the printing to a file could mess up the process managing the listener.

While the need for binding is important for programmers to understand, the multiprocessing binding implementation in theory is not. So long as it works as described, implementation details are at best simply distracting, and at worst actually misleading. (They can be misleading because a programmer might try to take advantage of details of the implementation that are either misunderstood or once properly understood but since modified, perhaps by a patch or a new release. For example, in this section, we talk of the virtual vector of bindings associated with the value of a symbol. It would be a mistake to try to get a handle on this vector and manipulate it directly instead of using a documented tool such as `symeval-in-process`.)

However, certain features of the binding model are visible to programmers and users, such as the *bind index* (which we describe below). In order to understand these features, it is necessary to understand some of the multiprocessing binding model. And, so long as the information is not used to get around proper ways of doing things, general understanding of the model is probably desirable.

There are various binding models that can be used. Broadly, there is *shallow* binding, where the value slot contains the current binding in some form, and *deep* binding, where a processes binding stack is searched for a symbols value.

Releases of Allegro CL prior to 6.2 used a standard (narrow) shallow binding model: the value slot of a symbol held the current binding of the current process. (The global value was held in a separate slot.) Each process had an associated bindstack with bindings recorded. Switching processes required unwinding out the current set of bindings and winding in the newly current processes bindings. This made accessing the current value in a process fast, but made both process switching and accessing the binding in another process slow.

Allegro CL uses a *wide* binding variant of shallow binding. A symbol has a logically wide value cell; it is conceptually a vector of values, one for each process. Each process's bindstack has an index by which the symbol's value vector is accessed, so that every process has immediate access to its own set of

current bindings. Because of the way the conceptual value-vector is implemented, the bindstacks need not be unwound and rewound when the current process changes.

One consequence of the new model is symbol-value access for wide binding is slightly slower than for narrow binding; an extra indirection is needed into the value vector in order to access a symbol's value. However, since wide binding is still a shallow-binding technique, it is still much faster than deep binding.

The advantages to wide binding over narrow binding are:

- Because the bindstacks do not have to be unwound/rewound during process switches, process-switching overhead is reduced, making processes lighter weight and faster.
- Because symbol values are immediately accessible, wide binding becomes a step toward making concurrent lisp thread execution possible.

We wish to emphasize again that the details of the vector of values implementation are internal. Programs should not try to access or modify the vector of values associated with a symbol.

Appendix A.1 The bindstack index

Under the normal multiprocessing model, each process has one implementation structure. Each of these implementation structures normally have one bindstack. When a bindstack is first allocated, it is given an index not used by any other bindstack in the system (including dead bindstacks which have however not been garbage-collected). This index serves as the index into each symbol's value-vector. When a process has had a bindstack allocated for it, the bindstack's index is shown by the process's print-object method. Note that this index cannot be considered an identifier; if a process dies and is gc'd, another process will eventually take on the same index. However, the index can be useful in distinguishing between currently running processes, especially if these processes have been given the same name.

A bindstack index will be seen in several places:

- The **:processes** command will print the bindstack-index of the process, if any, under the new header named "Bix".

```
cl-user(3): :pro
P Bix Dis   Sec  dSec  Priority  State   Process Name, ...
*   1   3     0   0.0           0 runnable Initial Lisp Listener
cl-user(4):
```

- The print-object method for a process will include the bindstack index in square brackets after the name:

```
cl-user(1): (mp:start-scheduler)
nil
cl-user(2): sys:*current-process*
#<multiprocessing:process Initial Lisp Listener[1] @ #x7145d76a>
cl-user(3):
```

- The **Process dialog** in the IDE and the Process browser in *Allegro Composer* display the Bindstack index in the "Bix" column.
- Trace output shows the bindstack index in square brackets after the indentation (see **:trace**):

```
cl-user(1): (defun fact (n)
              (if (= n 1) 1 (* n (fact (1- n)))))
fact
cl-user(2): :trace fact
(fact)
cl-user(3): (fact 3)
  0[1]: (fact 3)
    1[1]: (fact 2)
      2[1]: (fact 1)
        2[1]: returned 1
      1[1]: returned 2
    0[1]: returned 6
  6
cl-user(4):
```

A full sample run of tracing in a multiprocessing context is given below. This example is only for demonstration of the wide-binding concepts and has no other useful purpose. In this example, a client starts up a server and then writes a buffer-full to it, after which it waits to read the data back again. The server reads the buffer from the client and then writes it back again, after which it goes into a break loop (so that the process does not die). The client is then able to read the buffer sent by the client, and prints it and enters a break loop itself.

Note that after the first two unrelated **device-read** entry/exit pairs, the traced device-read call appears to have been entered twice without the first of these two exiting. However, with the aid of the **:processes** command, we can see that the first entry is on behalf of the "run-client" process, which has bindstack-index 3, and the second device-read entry is for bindstack-index 5, or the "server" process. It may be hard to see the second return from **device-read** (although it actually corresponds to the first **device-read** entry of the pair), but it does appear after the server enters its break loop.

```

cl-user(1): (shell "cat socket-trace.cl")
(in-package :user)

(eval-when (compile load eval)
  (require :sock))

(defun start-server (buffer)
  (let ((sock (socket:make-socket :connect :passive)))
    (mp::process-run-function
     "server" #'server-get-connection sock buffer)
    (socket:local-port sock)))

(defun server-get-connection (sock buffer)
  (let ((stm (socket:accept-connection sock)))
    (close sock) ;; don't need it anymore
    (unwind-protect
     (progn
      (read-in-buffer stm buffer)

      (write-out-buffer stm buffer))
     (close stm))
    (break "server breaks"))))

(defun run-client (port buffer)
  (let ((stm (socket:make-socket :remote-host "localhost"
                                :remote-port port)))
    (unwind-protect
     (progn
      (dotimes (i (length buffer))
        (setf (aref buffer i) (mod i 256)))
      (write-out-buffer stm buffer)
      (socket:shutdown stm :direction :output)
      (read-in-buffer stm buffer)
      )
     (close stm))))))

(defun write-out-buffer (stm buffer)
  (dotimes (i (length buffer))
    (write-byte (aref buffer i) stm)))

(defun read-in-buffer (stm buffer)
  (dotimes (i (length buffer))
    (setf (aref buffer i)

```

```
(read-byte stm))))
```

```
(defun socket-run (&optional (message-length 10))
  (let ((client-buffer
        (make-array message-length :element-type '(unsigned-byte 8)))
        (server-buffer
        (make-array message-length :element-type '(unsigned-byte
8))))
    (run-client (start-server server-buffer) client-buffer)
    (format t "~s~%" client-buffer)
    (force-output)
    (break "client breaks"))))
```

```
0
```

```
cl-user(2): :cl socket-trace
; Fast loading [...] /socket-trace.fasl
```

```
cl-user(3): (require :acldns)
; Fast loading [...] /code/acldns.fasl
```

```
t
```

```
cl-user(4): (trace (device-read :inside (or server-get-connection run-
client) :not-inside break))
```

```
(device-read)
```

```
cl-user(5): (mp:process-run-function "run-client" #'socket-run)
#<multiprocessing:process run-client @ #x71b375fa>
```

```
cl-user(6):
```

```
0[3]: (device-read #<file-simple-stream
                  #p"/etc/resolv.conf" for input pos 0 @
                  #x71b397ea>
                  nil 0 nil t)
```

```
0[3]: returned 109
```

```
0[3]: (device-read #<file-simple-stream
                  #p"/etc/resolv.conf" for input pos 109 @
                  #x71b397ea>
                  nil 0 nil t)
```

```
0[3]: returned -1
```

```
0[3]: (device-read #<multivalent stream socket connected from
                  localhost/4137 to localhost/4136 @ #x71b3f912>
                  nil 0 nil t)
```

```
0[5]: (device-read #<multivalent stream socket connected from
                  localhost/4136 to localhost/4137 @ #x71b44e42>
                  nil 0 nil t)
```

```
0[5]: returned 10
```

```
Break: server breaks
```

```
Restart actions (select using :continue):
```

```
0: return from break.
```

```
1: Abort entirely from this process.
```

```
[Current process: server]
```

```
[1c] cl-user(1):
```

```
0[3]: returned 10
```

```
 #(0 1 2 3 4 5 6 7 8 9)
```

```
Break: client breaks
```

```
Restart actions (select using :continue):
```

```
0: return from break.
```

```
1: Abort entirely from this process.
```

```
[Current process: run-client]
```

```
[1c] cl-user(1): :pro
```

```
P Bix Dis   Sec  dSec  Priority  State  Process Name, Whostate,
```

```
Arrest
```

```
*   5   1     0   0.0          0 waiting  server, waiting for input
```

```
*   3   3     0   0.0          0 waiting  run-client, waiting for input
```

```
*   4   1     0   0.0          0 waiting  Domain Name Server Client,  
                                     waiting for input
```

```
*   1   2     0   0.0          0 runnable Initial Lisp Listener
```

```
cl-user(7):
```

Appendix B: Processes and the runtime analyzer

The time runtime analyzer in Allegro CL (documented in *runtime-analyzer.htm*) collects data by periodically recording which functions are on the stack. The space runtime analyzer records all requests for new allocations. It is possible to set a flag in a process object that prevents sampling when that process is the current process. This permits more accurate data for time spent or space used by the process and code of interest.

When a new process is created, the value of the flag is set to its value in the current process. The predicate **profile-process-p** polls the profiling flag and can be used with **setf** to change its value.