OS_Mon

version 1.7

Typeset in LaTeX from SGML source using the DOCBUILDER 3.3.2 Document System.

Contents

OS_Mon Reference Manual

Short Summaries

- Application os_mon [page 4] – OS Monitoring Application
- Erlang Module cpu_sup [page 6] – A CPU Load and CPU Utilization Supervisor Process
- Erlang Module disksup [page 10] – A Disk Supervisor Process.
- Erlang Module memsup [page 12] – A memory Supervisor Process
- Erlang Module nteventlog [page 15] – This module implements a generic interface to the WIndows NT event log. The module is specific to Windows NT and in some ways replace the os_sup module, which is highly Unix specific.
- Erlang Module os_mon_mib [page 17] – Handles the OTP-OS-MON-MIB
- Erlang Module os_sup [page 18] – This module, together with some dedicated UNIX processes, implements a message passing service from the Solaris operating system to the error logger in the Erlang runtime system. The Solaris (SunOS 5.x) messages are retrieved from the syslog-daemon, syslogd.

os_mon

No functions are exported.

cpu_sup

The following functions are exported:

- `nprocs() -> UnixProcesses | {error, Reason}`
  [page 7] Get the number of UNIX processes running on this host
- `avg1() -> SystemLoad | {error, Reason}`
  [page 7] Get the system load average from the last minute
- `avg5() -> SystemLoad | {error, Reason}`
  [page 7] Get the system load average for the last five minutes
- `avg15() -> SystemLoad | {error, Reason}`
  [page 7] Get the system load average for the last fifteen minutes
- `util(ArgList) -> UtilSpec | {error, Reason}`
  [page 7] Gets the CPU utilization
- `util() -> CpuUtil | {error, Reason}`
  [page 9] Gets the CPU utilization

disksup

The following functions are exported:

- `get_check_interval() -> Time`
  [page 10] How often,in milliseconds, the disks are checked
- `get_disk_data() -> [DiskData]`
  [page 10] Get data for the disks in the system
- `get_almost_full_threshold() -> integer()`
  [page 10] Specify how much disk space can be used by each disk or partition before an alarm is sent

memsup

The following functions are exported:

- `get_check_interval() -> Time`
  [page 13] How often (in milliseconds) memory is checked
- `get_memory_data() -> MemData`
  [page 13] Get data for the memory in the system
- `get_system_memory_data() -> MemDataList`
  [page 13] Get system dependent memory data
- `get_procmem_high_watermark() -> integer()`
  [page 14] Specify how much memory can be allocated by one Erlang process before an alarm is sent
- `get_sysmem_high_watermark() -> integer()`
  [page 14] Specify how much memory can be allocated by one Erlang process before an alarm is sent
- `get_helper_timeout() -> integer()`
  [page 14] Returns the currently used `memsup_helper_timeout`
- `set_helper_timeout(Seconds) -> ok`
  [page 14] Sets the `memsup_helper_timeout` to use

nteventlog

The following functions are exported:

- `start(Identifier,MFA) -> Result`
  [page 15] Start the NT eventlog server
- `start_link(Identifier,MFA) -> Result`
  [page 16] Start and links the NT eventlog server
- `stop() -> Result`
  [page 16] Stop the message passing service

os_mon_mib

The following functions are exported:

- `load(Agent) -> ok | {error, Reason}`
  [page 17] Load the OTP-OS-MON-MIB.
- `unload(Agent) -> ok | {error, Reason}`
  [page 17] Unload the OTP-MIB.

os_sup

The following functions are exported:

- `start() -> Result`
  [page 18] Start the message passing service
- `start_link() -> Result`
  [page 18] Start and links the message passing service
- `stop() -> Result`
  [page 19] Stop the message passing service

os_mon

Application

This section describes the os_mon application in Erlang. The OS Monitoring application provides the following services:

- cpu_sup
- disksup
- memsup
- os_sup

Configuration

The following configuration parameters are defined for the OS Monitoring application. Refer to application(3) for more information about configuration parameters.

start_disksup = bool() <optional> Specifies if disksup should be started. The default is true.

start_memsup = bool() <optional> Specifies if memsup should be started. The default is true.

start_os_sup = bool() <optional> Specifies if os_sup should be started. The default is false.

disk_space_check_interval = integer() <optional> Defines how often, in minutes, the disksup process should check the disk space. The default is 30 minutes.

disk_almost_full_threshold = float() <optional> Defines what percentage of total disk space can be utilized before the disk_almost_full alarm is set. The default is 0.80 (80%).

memory_check_interval = integer() <optional> Defines how often, in minutes, the memsup process should check the memory. The default is one minute.

system_memory_high_watermark = float() <optional> Defines what percentage of the available system memory can be allocated before the corresponding alarm is set. The default is 0.8 (80%).

process_memory_high_watermark = float() <optional> Defines what percentage of the available system memory can be allocated by one Erlang process before the corresponding alarm is set. The default is 0.05 (5%).

memsup_helper_timeout = integer() (>= 1) <optional> Defines how long timeout time, in seconds, the memsup process should use when communicating with the memsup_helper process (an application internal server). Defaults to 30 seconds.

`os_sup_own = string()` Defines which directory contains the backup copy and the Erlang specific configuration files for `syslogd`, and the named pipe to receive the messages from `syslogd`.
Usually, this parameter has the value "/etc".

`os_sup_syslogconf = string()` Defines the full file name of the configuration file for `syslogd`.
Usually, this parameter has the value "/etc/syslog.conf".

`os_sup_errortag = atom()` Defines the atom with which to tag messages received from `syslogd` before forwarding them to the error logger in the Erlang runtime system.

SNMP MIBs

The following MIBs are defined in OS_MON:

OTP-OS-MON-MIB This MIB contains objects for instrumentation of disk, memory and cpu usage of the nodes in the system.

The MIB is stored in the `mibs` directory. It is defined in SNMPv2 SMI syntax. An SNMPv1 version of the mib is delivered in the `mibs/v1` directory.

The compiled MIB is located under `priv/mibs`, and the generated `.hrl` file under the `include` directory. To compile a MIB that IMPORTS the `OTP-OS-MON-MIB`, give the option `{il, ["os_mon/priv/mibs"]}` to the MIB compiler.

If the MIB should be used in a system, it should be loaded into an agent with a call to `os_mon_mib:load(Agent)`, where `Agent` is the Pid or registered name of an SNMP agent. Use `os_mon_mib:unload(Agent)` to unload the MIB. The implementation of this MIB uses Mnesia to store a cache with data needed. This means that Mnesia must run if this implementation of the MIB should be used. It also use functions defined for the `OTP-MIB`, thus that MIB must be loaded as well.

See Also

cpu_sup(3) [page 6], memsup(3) [page 12], disksup(3) [page 10], os_sup(3) [page 18], application(3), snmp(6)

cpu_sup

Erlang Module

cpu_sup is part of the os_mon application and all configuration parameters are defined in the reference documentation for the os_mon application.

cpu_sup is a process which supervises the CPU load and CPU utilization.

The load values are proportional to how long time a runnable UNIX process has to spend in the run queue before it is scheduled. Accordingly, higher values mean more system load. The returned value divided by 256 produces the figure displayed by rup and top. What is displayed as 2.00 in rup, is displayed as as load up to the second mark in xload.

For example, rup displays a load of 128 as 0.50, and 512 as 2.00.

If the user wants to view load values as percentages of machine capacity, then this way of measuring presents a problem, because the load values are not restricted to a fixed interval. In this case, the following simple mathematical transformation can produce the load value as a percentage:

PercentLoad = 100 * (1 - D/(D + Load))

D determines which load value should be associated with which percentage. Choosing D = 50 means that 128 is 60% load, 256 is 80%, 512 is 90%, and so on.

Another way of measuring system load is to divide the number of busy CPU cycles by the total number of CPU cycles. This produces values in the 0-100 range immediately. However, this method hides the fact that a machine can be more or less saturated. CPU utilization is therefore a better name than system load for this measure.

A server which receives just enough requests to never become idle will score a CPU utilization of 100%. If the server receives 50% more requests, it will still scores 100%. When the system load is calculated with the percentage formula shown previously, the load will increase from 80% to 87%.

The avg1/0, avg5/0, and avg15/0 functions can be used for retrieving system load values, and the util/0, and util/1 functions can be used for retrieving CPU utilization values. System load values can currently be retrieved on Solaris, Linux, FreeBSD, and OpenBSD. CPU utilization values can currently be retrieved on Solaris and Linux.

When run on Linux, cpu_sup assumes that the /proc file system is present and accessible by cpu_sup. If it is not, cpu_sup will fail.

Exports

`nprocs() -> UnixProcesses | {error, Reason}`

Types:

- UnixProcesses = integer()
- Reason = term()

Returns the number of UNIX processes running on this machine. This is a crude way of measuring the system load, but it may be of interest in some cases.

`avg1() -> SystemLoad | {error, Reason}`

Types:

- SystemLoad = integer()
- Reason = term()

Returns the average system load in the last 60 seconds, as described above. 0 represents no load, 256 represents the load reported as 1.00 by `rup`.

`avg5() -> SystemLoad | {error, Reason}`

Types:

- SystemLoad = integer()
- Reason = term()

Returns the average system load from the last 300 seconds, as described above. 0 represents no load, 256 represents the load reported as 1.00 by `rup`.

`avg15() -> SystemLoad | {error, Reason}`

Types:

- SystemLoad = integer()
- Reason = term()

Returns the average system load from the last 900 seconds, as described above. 0 represents no load, 256 represents the load reported as 1.00 by `rup`.

`util(ArgList) -> UtilSpec | {error, Reason}`

Types:

- ArgList = [Arg]
- Arg = atom()
- UtilSpec = UtilDesc | [UtilDesc]
- UtilDesc = {Cpus, Busy, NonBusy, Misc}
- Cpus = integer() | [integer()] | atom()
- Busy = CpuStateDesc
- NonBusy = CpuStateDesc
- CpuStateDesc = float() | [{atom(), float()}]
- Misc = [{atom(), term()}]
- Reason = term()

Returns a CPU utilization specification of the CPU utilization since the last call to `util/0` or `util/1` by the calling process.

> **Note:**
> The returned value of the first call to `util/0` or `util/1` by a process will on most systems be the CPU utilization since system boot, but this is not guaranteed and the value should therefore be regarded as garbage. This also applies to the first call to `util/0` or `util/1` by a process after a restart of `cpu_sup`.

Currently recognized `Arguments`:

`detailed` The returned `UtilDesc`(s) will be more detailed.

`per_cpu` Each CPU will be specified separately (assuming this information can be retrieved from the operating system), i.e. one `UtilDesc` per CPU will be returned.

If the `per_cpu` has been passed as `Argument`, a list of `UtilDescs` will be returned; otherwise, only one `UtilDesc` will be returned.

Description of the `UtilDesc`:

`Cpus` If `detailed` and/or `per_cpu` has been passed as arguments, this element will contain the CPU number, or a list of CPU numbers of the CPU or CPUs that the `UtilDesc` contains information about.

If neither `detailed` nor `per_cpu` has been passed as arguments, this field will contain the atom `all` which implies that the `UtilDesc` contains information about all CPUs.

`Busy` If `detailed` has been passed as argument, this element will contain a list of `{atom(), float()}` tuples. Each tuple in the list contains information about a processor state that has been identified as a busy processor state. The first element is the name of the state, and the second element contains a float representing the percentage share of the CPU cycles spent in this state since the last call to `util/0` or `util/1`.

If `detailed` hasn't been passed as argument, this element will contain the sum of the percentage shares of the CPU cycles spent in all states identified as busy.

If `per_cpu` hasn't been passed, the value(s) presented are the average of all CPUs.

`NonBusy` The same as for the `Busy` element, but for processor states that has been identified as non-busy.

`Misc` Miscellaneous information. Currently unused; reserved for future use.

Currently these processor states are identified as busy:

`user` Executing code in user mode.

`nice_user` Executing code in low priority (nice) user mode. This state is currently only identified on Linux.

`kernel` Executing code in kernel mode.

Currently these processor states are identified as non-busy:

`wait` Waiting. This state is currently only identified on Solaris.

`idle` Idle.

> **Note:**
> Identified processor states may be different on different operation systems and may change between different versions of `cpu_sup` on the same operating system. The sum of the percentage shares of the CPU cycles spent in all busy and all non-busy processor states will always add up to 100%, though.

Failure: `badarg` if the `ArgList` is not a list of recognized `Arguments`.

`util() -> CpuUtil | {error, Reason}`

Types:

- CpuUtil = float()
- Reason = term()

Returns CPU utilization since the last call to `util/0` or `util/1` by the calling process.

> **Note:**
> The returned value of the first call to `util/0` or `util/1` by a process will on most systems be the CPU utilization since system boot, but this is not guaranteed and the value should therefore be regarded as garbage. This also applies to the first call to `util/0` or `util/1` by a process after a restart of `cpu_sup`.

The CPU utilization is defined as the sum of the percentage shares of the CPU cycles spent in all busy processor states (see `util/1`) in average on all CPUs.

See Also

os_mon(3) [page 4]

disksup

Erlang Module

disksup is part of the os_mon application and all configuration parameters are defined in the reference documentation for the os_mon application.

disksup is a process which supervises the available disk space in the system. Once every disk_space_check_interval minutes, the disks are checked and an alarm is generated for each disk which uses more than the disk_almost_full_threshold of available space.

On UNIX All (locally) mounted disks are checked, including the swap disk if it is present.

On WIN32 All logical drives of type "FIXED_DISK" are checked.

The disksup process defines one alarm which it sends using alarm_handler:set_alarm(Alarm). Alarm is defined as follows:

{{disk_almost_full, MountedOn}, []} This alarm is sent when a disk uses more than disk_almost_full_threshold of its available disk space, and it is cleared automatically when disksup observes that the disk space is back to normal.

Exports

get_check_interval() -> Time

Types:

- Time = integer()

Time interval, in milliseconds, which defined how often the disks are checked.

get_disk_data() -> [DiskData]

Types:

- DiskData = {Id, KByte, Capacity}
- Id = string()
- KByte = integer()
- Capacity = integer()

Gets data for the system disks or partitions. Id is a string that identifies the disk or partition. KByte is the total size of the disk or partition in kbytes. Capacity is the percentage of disk space used.

get_almost_full_threshold() -> integer()

Threshold as a percentage of the available disk space. It specifies how much disk space can be used by each disk or partition before an alarm is sent.

See Also

alarm_handler(3), os_mon(3)

memsup

Erlang Module

memsup is part of the os_mon application and all configuration parameters are defined in the reference documentation for the os_mon application.

memsup is a process which supervises the memory usage for the system and for individual processes, as follows:

- If more than system_memory_high_watermark of available system memory is allocated, as reported by the underlying operating system, the alarm system_memory_high_watermark is set.
- If any Erlang process in the system has allocated more than process_memory_high_watermark of total system memory, the alarm process_memory_high_watermark is set.

The total system memory reported under UNIX is the number of physical pages of memory times the page size, and the available memory is the number of available physical pages times the page size. This is a reasonable measure as swapping should be avoided anyway, but the task of defining total memory and available memory is difficult because of virtual memory and swapping.

The memsup process defines two alarms which are set by the alarm_handler:set_alarm(Alarm) function. Alarm is defined as:

{system_memory_high_watermark, []}. This alarm is set when the used system memory exceeds system_memory_high_watermark of the total available memory.

{process_memory_high_watermark, Pid}. This alarm is set when an Erlang process exceeds process_memory_high_watermark of the total available memory.

These alarms are cleared automatically when the alarm cause is no longer valid.

There is also a interface to system dependent memory data, get_system_memory_data/0. The output is highly dependent on the underlying operating system and the interface is targeted primarily for systems without virtual memory (e.g. VxWorks). The output on other systems is however still valid, although sparse.

A call to get_system_memory_data/0 is more costly than a call to get_memory_data/0 as data is collected synchronously when this function is called.

Exports

`get_check_interval() -> Time`

Types:

- Time = integer()

A time interval, in milliseconds, which defines how often memory is checked. The `get_system_memory_data()` function is in no way affected by this interval.

`get_memory_data() -> MemData`

Types:

- MemData = {TotalMemorySize, AllocatedBytes, {LargestPid, PidAllocatedBytes}}
- TotalMemorySize = integer()
- AllocatedBytes = integer()
- LargestPid = pid()
- PidAllocatedBytes = integer()

Returns data about the memory in the system. `LargestPid` is the Pid of the largest Erlang process in the system. `PidAllocatedBytes` is the amount of memory the `LargestPid` has allocated.

`get_system_memory_data() -> MemDataList`

Types:

- MemDataList = [TaggedValue ...]
- TaggedValue = {Tag, Value}
- Value = integer()
- Tag = atom()

Gets system dependent memory data. The result is returned as a list containing tagged tuples, where the tag can be one of the following:

`total_memory` The total amount of memory (in bytes) available to the Erlang emulator, allocated and free. May or may not be equal to the amount of memory configured in the system.

`free_memory` The amount of free memory available to the Erlang emulator for allocation.

`system_total_memory` The amount of memory available to the whole operating system. This may well be equal to `total_memory` but not necessarily.

`largest_free` The size of the largest contiguous free memory block available to the Erlang emulator.

`number_of_free` The number of free blocks available to the Erlang runtime system. This gives a fair indication of how fragmented the memory is.

As with `get_memory_data()`, the values on Unix-like systems indicate the amount of *physical* memory that is configured and free. The `largest_free` and `number_of_free` tags are currently only returned on a VxWorks system.

All memory sizes are presented as number of *bytes*.

`get_procmem_high_watermark() -> integer()`

Threshold as a percentage of the total available system memory. It specifies how much memory can be allocated by one Erlang process before an alarm is sent.

`get_sysmem_high_watermark() -> integer()`

Threshold as a percentage of the total available system memory. It specifies how much memory can be allocated by the system before an alarm is sent.

`get_helper_timeout() -> integer()`

Returns how long timeout time, in seconds, the `memsup` process uses when communicating with the `memsup_helper` process (an application internal server).

`set_helper_timeout(Seconds) -> ok`

Types:

- Seconds = integer() (>= 1)

Sets how long timeout time, in seconds, the `memsup` process should use when communicating with the `memsup_helper` process (an application internal server).

Failure: `badarg` if `Seconds` is not an integer or is an integer less than `1`.

See Also

alarm_handler(3), os_mon(3)

nteventlog

Erlang Module

The nteventlog module is a server which will inform your erlang application of all events written to the Windows NT event log. This is implemented with a port program that monitors the eventlog file and reacts whenever a new record is written to the log.

Your Erlang application is informed of each record in the eventlog through a user supplied callback function (an “MFA”). This function can do whatever filtering and formatting is necessary and then deploy any type of logging suitable for your application. When the user supplied function returns, the log record is regarded as accepted and the port program updates its persistent state so that the same event will not be sent again (as long ar the server is started with the same identifier).

When the service is started, all events that arrived to the eventlog since the last accepted message (for the current identifier) are sent to the user supplied function. This can make your application aware of operating system problems that arise when your application is not running (like the problem that made it stop the last time). The interpretation of the log records is entierly up to the application.

When starting the service, a identifier is supplied, which should be reused whenever the same application (or node) wants to start the server. The identifier is the key for the persistent state telling the server which events are delivered to your application. As long as the same identifier is used, the same eventlog record will not be sent to Erlang more than once (with the exception of when grave system failures arise, in which case the last records written before the failure may be sent to Erlang more again after reboot).

If the event log is configured to wrap around automatically, records that has arrived to the log and been overwritten when the server was not running are lost. The server however detects this state and loses no records that are not overwritten.

Exports

`start(Identifier,MFA) -> Result`

Types:

- Identifier = string() | atom()
- MFA = {Mod, Func, Args}
- Mod = atom()
- Func = atom()
- Args = list()
- Result = {ok, Pid} | {error, {already_started, Pid}}
-
- LogData = {Time,Category,Facility,Severity,Message}
- Time = {MegaSecs, Secs, Microsecs}

- MegaSecs = integer()
- Secs = integer()
- Microsecs = integer()
- Category = string()
- Facility = string()
- Severity = string()
- Message = string()

This function starts the server. The supplied function should take at least one argument of type LogData, optionally followed by the arguments supplied in Args.

The `LogData` tuple contains:

1. The message `Time` is represented as by the erlang:now() bif.
2. The message `Category` which usually is one of the strings “System”, “Application” or “Security”. Note that the NT eventlog viewer has another notion of category, which in most cases is totally meaningless and therefor not imported into erlang. What this module calls a category is one of the main three types of events occuring in a normal NT system.
3. The message `Facility` is the source of the event, usually the name of the application that generated it. This could be almost any string. When matching events from certain applications, the version number of the application may have to be accounted for. What this module calls facility, the NT event viewer calls “source”.
4. The message `Severity` is one of the strings “Error”, “Warning”, “Informational”, “Audit_Success”, “Audit_Faulure” or, in case of a currently unknown Windows NT version “Severity_Unknown”.
5. The `Message` string is formatted exactly as it would be in the NT eventlog viewer. Binary data is not imported into erlang.

`start_link(Identifier,MFA) -> Result`

Types:

- Identifier = string() | atom()
- MFA = {Mod, Func, Args}
- Mod = atom()
- Func = atom()
- Args = list()
- Result = {ok, Pid} | {error, {already_started, Pid}}

Behaves as start/2 but also links the server.

`stop() -> Result`

Types:

- Result = stopped

Stops a started server, usually only used during development. The server does not have to be shut down gracefully to maintain its state.

See Also

os_sup(3) [page 18] and the Windows NT documentation.

os_mon_mib

Erlang Module

The snmp application should be used to start an snmp agent then the API functions below can be used to load/unload the OTP-OS-MON-MIB into/from the agent. The instrumentation of the OTP-OS-MON-MIB uses mnesia, hence mnesia must be started prior to loading the OTP-OS-MON-MIB.

Exports

`load(Agent) -> ok | {error, Reason}`

Types:

- Agent - pid() | atom()
- Reason - term()

Load the OTP-OS-MON-MIB.

`unload(Agent) -> ok | {error, Reason}`

Types:

- Agent - pid() | atom()
- Reason - term()

Unload the OTP-OS-MON-MIB.

See Also

snmp(3)

os_sup

Erlang Module

This module starts a server written in Erlang (and later referenced only as server), which receives messages from the Solaris operating system. The messages are tagged with an atom and subsequently forwarded to the *error_logger* in the Erlang runtime system. If the atom is `std_error`, the messages are handled the same way as the bulk of internal error messages in the Erlang runtime system.

This module, together with the dedicated UNIX-processes, makes a number of reconfigurations to the Solaris operating system when the service is enabled. These configurations include:

- the installation of a new configuration file for `syslogd`
- the creation of a named pipe
- the start of a port program.

As a consequence of these modifications:

1. `syslogd` writes messages of interest to the named pipe
2. the port program reads messages from the named pipe and forwards them to the server
3. the server delivers them to the error logger of the Erlang runtime system.

When the service is disabled, the original configuration is restored.

Exports

`start() -> Result`

Types:

- Result = {ok, Pid} | {error, {already_started, Pid}}
- Pid = pid()

This function starts the server together with its dedicated UNIX processes. It returns {`ok, Pid`} if the start was successful, otherwise {`error, already_started`}.

`start_link() -> Result`

Types:

- Result = {ok, Pid} | {error, {already_started, Pid}}
- Pid = pid()

This function starts the server together with its dedicated UNIX processes. The caller is also linked to the server. It returns {ok, Pid} if the start was successful, otherwise {error, already_started} .

stop() -> Result

Types:

- Result = ok | not_started

This function stops the server and restores the original configuration of the operating system. It returns ok if successful, otherwise not_started.

Operation

1. This module is normally started by the *supervisor* and *supervisor_bridge* behaviours. Consequently, the user should not call the functions described above.
2. This module cannot be run in multiple instances on the same hardware. Special care must be taken if two or more Erlang nodes execute on the same hardware platform so that only one node uses this service *in any one instance*.
3. This module requires that a number of actions be been taken prior to starting it. These actions must be performed with *root* privileges on SunOS 5 and include change of ownership and file privileges of an executable binary file, and copying and creating a modified copy of the configuration file for the syslog-daemon *syslogd*. In addition, a the following configuration parameters must be set.
   (a) implement the server using *gen_server*.
   (b) implement protection against starting two or more instances of the service on the same hardware platform.

See also

- os_mon(3), error_logger(3), Installation Guide for your platform.
- syslogd(1M), syslog.conf(4) in the Solaris documentation.

Index of Modules and Functions

Modules are typed in *this way*.
Functions are typed in `this way`.