OS_Mon

version 2.1
Typeset in \LaTeX from SGML source using the DocBuilder-0.9.8.4 Document System.
Contents

1 OS_Mon Reference Manual ....................................... 1
  1.1 os_mon ............................................. 4
  1.2 cpu_sup .......................................... 6
  1.3 disksup ........................................... 10
  1.4 memsup ........................................... 13
  1.5 nteventlog ........................................ 17
  1.6 os_mon_mib ...................................... 19
  1.7 os_sup ........................................... 20
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 13] – A Memory Supervisor Process
- Erlang Module **nteventlog** [page 17] – Interface to Windows Event Log
- Erlang Module **os_mon_mib** [page 19] – Loading and Unloading of OTP-OS-MON-MIB
- Erlang Module **os_sup** [page 20] – Interface to OS System Messages

**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 for 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()** -> CpuUtil | {error, Reason}  
  [page 7] Get the CPU utilization
- **util(Opts)** -> UtilSpec | {error, Reason}  
  [page 8] Get the CPU utilization
disksup

The following functions are exported:

- `get_disk_data()` -> [DiskData]
  [page 11] Get data for the disks in the system

- `get_check_interval()` -> MS
  [page 11] Get time interval, in milliseconds, for the periodic disk space check

- `set_check_interval(Minutes)` -> ok
  [page 11] Set time interval, in minutes, for the periodic disk space check

- `get_almost_full_threshold()` -> Percent
  [page 11] Get threshold, in percent, for disk space utilization

- `set_almost_full_threshold(Float)` -> ok
  [page 11] Set threshold, as percentage represented by a float, for disk space utilization

memsup

The following functions are exported:

- `get_memory_data()` -> {Total, Allocated, Worst}
  [page 14] Get data for the memory in the system

- `get_system_memory_data()` -> MemDataList
  [page 15] Get system dependent memory data

- `get_check_interval()` -> MS
  [page 15] Get time interval, in milliseconds, for the periodic memory check

- `set_check_interval(Minutes)` -> ok
  [page 16] Set time interval, in minutes, for the periodic memory check

- `get_procmem_high_watermark()` -> int()
  [page 16] Get threshold, in percent, for process memory allocation

- `set_procmem_high_watermark(Float)` -> ok
  [page 16] Set threshold, as percentage represented by a float, for process memory allocation

- `get_sysmem_high_watermark()` -> int()
  [page 16] Get threshold, in percent, for system memory allocation

- `set_sysmem_high_watermark(Float)` -> ok
  [page 16] Set threshold, given as a float, for system memory allocation

- `get_helper_timeout()` -> Seconds
  [page 16] Get the timeout value, in seconds, for memory checks

- `set_helper_timeout(Seconds)` -> ok
  [page 16] Set the timeout value, in seconds, for memory checks
nteventlog

The following functions are exported:

- `start(Identifier, MFA) -> Result`
  [page 17] Start the NT eventlog server
- `start_link(Identifier, MFA) -> Result`
  [page 17] Start the NT eventlog server
- `stop() -> stopped`
  [page 18] Stop the NT eventlog server

os_mon_mib

The following functions are exported:

- `load(Agent) -> ok | {error, Reason}`
  [page 19] Load the OTP-OS-MON-MIB
- `unload(Agent) -> ok | {error, Reason}`
  [page 19] Unload the OTP-OS-MON-MIB

os_sup

The following functions are exported:

- `enable() -> ok | {error, Res}`
  [page 21] Enable the service (Solaris only)
- `enable(Dir, Conf) -> ok | {error, Error}`
  [page 21] Enable the service (Solaris only)
- `disable() -> ok | {error, Res}`
  [page 22] Disable the service (Solaris only)
- `disable(Dir, Conf) -> ok | {error, Error}`
  [page 22] Disable the service (Solaris only)
The operating system monitor, OS\_Mon, provides the following services:

- **cpu\_sup** [page 6] CPU load and utilization supervision (Unix)
- **disksup** [page 10] Disk supervision (Unix, Windows)
- **memsup** [page 13] Memory supervision (Unix, Windows, VxWorks)
- **os\_sup** [page 20] Interface to OS system messages (Solaris, Windows)

To simplify usage of OS\_Mon on distributed Erlang systems, it is not considered an error trying to use a service at a node where it is not available (either because OS\_Mon is not running, or because the service is not available for that OS, or because the service is not started). Instead, a warning message is issued via error\_logger and a dummy value is returned, which one is specified in the man pages for the respective services.

**Configuration**

When OS\_Mon is started, by default all services available for the OS, except os\_sup, are automatically started. This configuration can be changed using the following application configuration parameters:

- **start\_cpu\_sup** = bool() Specifies if cpu\_sup should be started. Defaults to true.
- **start\_disksup** = bool() Specifies if disksup should be started. Defaults to true.
- **start\_memsup** = bool() Specifies if memsup should be started. Defaults to true.
- **start\_os\_sup** = bool() Specifies if os\_sup should be started. Defaults to false.

Configuration parameters effecting the different OS\_Mon services are described in the respective man pages.

See [config(4)] for information about how to change the value of configuration parameters.
SNMP MIBs

The following MIBs are defined in OS\texttt{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 \texttt{mibs} directory. It is defined in SNMPv2 SMI syntax. An SNMPv1 version of the MIB is delivered in the \texttt{mibs/v1} directory.

The compiled MIB is located under \texttt{priv/mibs}, and the generated .hrl file under the \texttt{include} directory. To compile a MIB that IMPORTS the OTP-OS-MON-MIB, give the option \{il, ["os\texttt{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 \texttt{os\_mon\_mib:load(Agent)}, where \texttt{Agent} is the pid or registered name of an SNMP agent. Use \texttt{os\_mon\_mib:unload(Agent)} to unload the MIB. The implementation of this MIB uses Mnesia to store a cache with data needed, which implicates that Mnesia must be up and running. The MIB also use functions defined for the OTP-MIB, thus that MIB must be loaded as well.

See Also

cpu\_sup(3) [page 6], disksup(3) [page 10], memsup(3) [page 13], os\_sup(3) [page 20], nteventlog(3) [page 17], \[snmp(3)].
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 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 percentage 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:

\[
\text{PercentLoad} = 100 \times (1 - \frac{D}{D + \text{Load}})
\]

\(D\) 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 \text{avg1/0}, \text{avg5/0}, and \text{avg15/0} functions can be used for retrieving system load values, and the \text{util/0} and \text{util/1} functions can be used for retrieving CPU utilization values.

When run on Linux, \text{cpu_sup} assumes that the /proc file system is present and accessible by \text{cpu_sup}. If it is not, \text{cpu_sup} will terminate.
Exports

nprocs() -> UnixProcesses | {error, Reason}
  Types:
  • UnixProcesses = int()
  • 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.
  Returns 0 if cpu_sup is not available.

avg1() -> SystemLoad | {error, Reason}
  Types:
  • SystemLoad = int()
  • Reason = term()
  Returns the average system load in the last minute, as described above. 0 represents no load, 256 represents the load reported as 1.00 by rup.
  Returns 0 if cpu_sup is not available.

avg5() -> SystemLoad | {error, Reason}
  Types:
  • SystemLoad = int()
  • Reason = term()
  Returns the average system load in the last five minutes, as described above. 0 represents no load, 256 represents the load reported as 1.00 by rup.
  Returns 0 if cpu_sup is not available.

avg15() -> SystemLoad | {error, Reason}
  Types:
  • SystemLoad = int()
  • Reason = term()
  Returns the average system load in the last 15 minutes, as described above. 0 represents no load, 256 represents the load reported as 1.00 by rup.
  Returns 0 if cpu_sup is not available.

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 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` below) in average on all CPUs.

Returns 0 if `cpu_sup` is not available.

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

Types:
- Opts = [detailed | per_cpu]
- UtilSpec = UtilDesc | [UtilDesc]
- UtilDesc = {Cpus, Busy, NonBusy, Misc}
- Cpus = all | int() | [int()]()
- Busy = NonBusy = {State, Share} | Share
- State = user | nice_user | kernel
- | wait | idle | atom()
- Share = float()
- Misc = []
- Reason = term()

Returns CPU utilization since the last call to `util/0` or `util/1` by the calling process, in more detail than `util/0`.

**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 after a restart of `cpu_sup`.

Currently recognized options:
- detailed The returned `UtilDesc(s)` will be even more detailed.
- per_cpu Each CPU will be specified separately (assuming this information can be retrieved from the operating system), that is, a list with one `UtilDesc` per CPU will be returned.

Description of `UtilDesc` = `{Cpus, Busy, NonBusy, Misc}: 

8 OS_Mon
**Cpus**  If the detailed and/or per_cpu option is given, this is the CPU number, or a list of the CPU numbers.

If not, this is the atom all which implies that the UtilDesc contains information about all CPUs.

**Busy**  If the detailed option is given, this is a list of {State, Share} tuples, where each tuple contains information about a processor state that has been identified as a busy processor state (see below). The atom State is the name of the state, and the float Share represents the percentage share of the CPU cycles spent in this state since the last call to util/0 or util/1.

If not, this is the sum of the percentage shares of the CPU cycles spent in all states identified as busy.

If the per_cpu is not given, the value(s) presented are the average of all CPUs.

**NonBusy**  Similar to Busy, but for processor states that have been identified as non-busy (see below).

**Misc**  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 operating 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.

Returns \{all,0,0,[]\} if cpu_sup is not available.

**See Also**

os_mon(3) [page 4]
disksup is an Erlang module which supervises the available disk space in the system. It is part of the OS_Mon application, see os_mon(6) [page 4]. Available for Unix and Windows.

Periodically checks the disks. For each disk or partition which uses more than a certain amount of the available space, the alarm \{disk\_almost\_full, MountedOn, []\} is set.

**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.

Alarms are reported to the SASL alarm handler, see [alarm\_handler(3)]. To set an alarm, alarm\_handler:\set\_alarm(Alarm) is called where Alarm is the alarm specified above. The alarms are cleared automatically when the alarm cause is no longer valid.

**Configuration**

The following configuration parameters can be used to change the default values for time interval and threshold:

- **disk\_space\_check\_interval** = int() > 0 The time interval, in minutes, for the periodic disk space check. The default is 30 minutes.

- **disk\_almost\_full\_threshold** = float() The threshold, as percentage of total disk space, for how much disk can be utilized before the disk\_almost\_full alarm is set. The default is 0.80 (80%).

See [config(4)] for information about how to change the value of configuration parameters.
Exports

get_disk_data() -> [DiskData]
  Types:
  - DiskData = {Id, KByte, Capacity}
  - Id = string()
  - KByte = int()
  - Capacity = int()
  Returns the result of the latest disk check. 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.
  The function is asynchronous in the sense that it does not invoke a disk check, but returns the latest available value.
  Returns [{"none",0,0}] if disksup is not available.

get_check_interval() -> MS
  Types:
  - MS = int()
  Returns the time interval, in milliseconds, for the periodic disk space check.

set_check_interval(Minutes) -> ok
  Types:
  - Minutes = int()>1
  Changes the time interval, given in minutes, for the periodic disk space check.
  The change will take effect after the next disk space check and is non-persistant. That is, in case of a process restart, this value is forgotten and the default value will be used.
  See Configuration [page 10] above.

get_almost_full_threshold() -> Percent
  Types:
  - Percent = int()
  Returns the threshold, in percent, for disk space utilization.

set_almost_full_threshold(Float) -> ok
  Types:
  - Float = float(), 0=<Float=<1
  Changes the threshold, given as a float, for disk space utilization.
  The change will take effect during the next disk space check and is non-persistant. That is, in case of a process restart, this value is forgotten and the default value will be used.
  See Configuration [page 10] above.
See Also

[alarm_handler(3)], os_mon(3) [page 4]
memsup

Erlang Module

memsup is a process which supervises the memory usage for the system and for
individual processes. It is part of the OS Mon application, see os_mon(6) [page 4].
Available for Unix, Windows and VxWorks.

Periodically performs a memory check:

- If more than a certain amount 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 Pid in the system has allocated more than a certain amount of
  total system memory, the alarm \{process_memory_high_watermark, Pid\} is set.

Alarms are reported to the SASL alarm handler, see [alarm_handler(3)]. To set an
alarm, alarm_handler: set_alarm(Alarm) is called where Alarm is either of the alarms
specified above.

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

The function get_memory_data() [page 14] can be used to retrieve the result of the
latest periodic memory check.

There is also a interface to system dependent memory data, get_system_memory_data()
[page 15]. The result 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.

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.
**Configuration**

The following configuration parameters can be used to change the default values for time intervals and thresholds:

- `memory_check_interval = int() > 0` The time interval, in minutes, for the periodic memory check. The default is one minute.

- `system_memory_high_watermark = float()` The threshold, as percentage of system memory, for how much system memory can be allocated before the corresponding alarm is set. The default is 0.80 (80%).

- `process_memory_high_watermark = float()` The threshold, as percentage of system memory, for how much system memory can be allocated by one Erlang process before the corresponding alarm is set. The default is 0.05 (5%).

- `memsup_helper_timeout = int() > 0` A timeout, in seconds, for how long the `memsup` process should wait for a result from a memory check. If the timeout expires, a warning message "OS Mon (memsup) timeout" is issued via `error_logger` and any pending, synchronous client calls will return a dummy value. Normally, this situation should not occur. There have been cases on Linux, however, where the pseudo file from which system data is read is temporarily unavailable when the system is heavily loaded. The default is 30 seconds.

- `memsup_system_only = bool()` Specifies whether the `memsup` process should only check system memory usage (true) or not. The default is false, meaning that information regarding both system memory usage and Erlang process memory usage is collected.

  It is recommended to set this parameter to false on systems with many concurrent processes, as each process memory check makes a traversal of the entire list of processes.

See [config(4)] for information about how to change the value of configuration parameters.

**Exports**

```erlang
get_memory_data() -> {Total, Allocated, Worst}
```

Types:
- `Total = Allocated = int()`
- `Worst = {Pid, PidAllocated} | undefined`
- `Pid = pid()`
- `PidAllocated = int()`

Returns the result of the latest memory check, where `Total` is the total memory size and `Allocated` the allocated memory size, in bytes. `Worst` is the pid and number of allocated bytes of the largest Erlang process on the node. If `memsup` should not collect process data, that is if the configuration parameter `memsup_system_only` was set to true, `Worst` is undefined.

The function is normally asynchronous in the sense that it does not invoke a memory check, but returns the latest available value. The one exception if is the function is
called before a first memory check is finished, in which case it does not return a value until the memory check is finished.
Returns \( \{0,0,\text{pid()},0\} \) or \( \{0,0,\text{undefined}\} \) if \texttt{memsup} is not available, or if all memory checks so far have timed out.

\begin{verbatim}
get_system_memory_data() -> MemDataList

Types:
- MemDataList = [{Tag, Size}]
  - Tag = atom()
  - Size = int()

Invokes a memory check and returns the resulting, system dependent, data as a list of tagged tuples, where \texttt{Tag} can be one of the following:

\begin{itemize}
  \item \texttt{total_memory} \hspace{1em} The total amount of memory available to the Erlang emulator, allocated and free. May or may not be equal to the amount of memory configured in the system.
  \item \texttt{free_memory} \hspace{1em} The amount of free memory available to the Erlang emulator for allocation.
  \item \texttt{system_total_memory} \hspace{1em} The amount of memory available to the whole operating system. This may well be equal to \texttt{total_memory} but not necessarily.
  \item \texttt{largest_free} \hspace{1em} The size of the largest contiguous free memory block available to the Erlang emulator.
  \item \texttt{number_of_free} \hspace{1em} The number of free blocks available to the Erlang runtime system. This gives a fair indication of how fragmented the memory is.
  \item \texttt{buffered_memory} \hspace{1em} The amount of memory the system uses for temporary storing raw disk blocks.
  \item \texttt{cached_memory} \hspace{1em} The amount of memory the system uses for cached files read from disk.
  \item \texttt{total_swap} \hspace{1em} The amount of total amount of memory the system has available for disk swap.
  \item \texttt{free_swap} \hspace{1em} The amount of memory the system has available for disk swap.
\end{itemize}

All memory sizes are presented as number of bytes.
The \texttt{largest_free} and \texttt{number_of_free} tags are currently only returned on a VxWorks system.
Returns the empty list [] if \texttt{memsup} is not available, or if the memory check times out.

\textbf{Note:}
On Linux the memory available to the emulator is \texttt{cached_memory} and \texttt{buffered_memory} in addition to \texttt{free_memory}.
\end{verbatim}

\begin{verbatim}
get_check_interval() -> MS

Types:
- MS = int()
\end{verbatim}
Returns the time interval, in milliseconds, for the periodic memory check.

```
set_check_interval(Minutes) -> ok
```

Types:
- `Minutes = int() > 0`

Changes the time interval, given in minutes, for the periodic memory check.

The change will take effect after the next memory check and is non-persistent. That is, in case of a process restart, this value is forgotten and the default value will be used. See Configuration [page 14] above.

```
get_prcmepm_high_watermark() -> int()
```

Returns the threshold, in percent, for process memory allocation.

```
set_prcmepm_high_watermark(Float) -> ok
```

Changes the threshold, given as a float, for process memory allocation.

The change will take effect during the next periodic memory check and is non-persistent. That is, in case of a process restart, this value is forgotten and the default value will be used. See Configuration [page 14] above.

```
get_sysmem_high_watermark() -> int()
```

Returns the threshold, in percent, for system memory allocation.

```
set_sysmem_high_watermark(Float) -> ok
```

Changes the threshold, given as a float, for system memory allocation.

The change will take effect during the next periodic memory check and is non-persistent. That is, in case of a process restart, this value is forgotten and the default value will be used. See Configuration [page 14] above.

```
get_helper_timeout() -> Seconds
```

Types:
- `Seconds = int()`

Returns the timeout value, in seconds, for memory checks.

```
set_helper_timeout(Seconds) -> ok
```

Types:
- `Seconds = int() (>= 1)`

Changes the timeout value, given in seconds, for memory checks.

The change will take effect for the next memory check and is non-persistent. That is, in the case of a process restart, this value is forgotten and the default value will be used. See Configuration [page 14] above.

See Also

[alarm_handler(3)], os_mon(3) [page 4]
nteventlog

Erlang Module

nteventlog provides a generic interface to the Windows event log. It is part of the OS_Mon application, see os_mon(6) [page 4]. Available for Windows versions where the event log is available. That is, not for Windows 98 and some other older Windows versions, but for most (all?) newer Windows versions.

This module is used as the Windows backend for os_sup, see os_sup(3) [page 20]. To retain backwards compatibility, this module can also be used to start a standalone nteventlog process which is not part of the OS_Mon supervision tree. When starting such a process, the user has to supply an identifier as well as a callback function to handle the messages.

The identifier, an arbitrary string, should be reused whenever the same application (or node) wants to start the process. nteventlog is informed about all events that have arrived to the eventlog since the last accepted message for the current identifier. As long as the same identifier is used, the same eventlog record will not be sent to nteventlog more than once (with the exception of when graved system failures arise, in which case the last records written before the failure may be sent to Erlang again after reboot).

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

The callback function works as described in os_sup(3).

Exports

\[
\begin{align*}
\text{start} & : \text{Identifier, MFA} \rightarrow \text{Result} \\
\text{start_link} & : \text{Identifier, MFA} \rightarrow \text{Result}
\end{align*}
\]

Types:
- Identifier = string() | atom()
- MFA = \{Mod, Func, Args\}
- Mod = Func = atom()
- Args = \[term()\]
- Result = \{ok, Pid\} | \{error, \{already_started, Pid\}\}
- Pid = pid()
This function starts the standalone nteventlog process and, if `start_link/2` is used, links to it.

Identifier is an identifier as described above.

MFA is the supplied callback function. When nteventlog receives information about a new event, this function will be called as `apply(Mod, Func, [Event|Args])` where Event is a tuple.

```
stop() -> stopped
```

Types:
- Result = stopped

Stops nteventlog. Usually only used during development. The server does not have to be shut down gracefully to maintain its state.

**See Also**

`os_mon(6)` [page 4], `os_sup(3)` [page 20]

Windows NT documentation
os_mon_mib

Erlang Module

Functions for loading and unloading the OTP-OS-MON-MIB into/from an SNMP 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()

Loads the OTP-OS-MON-MIB.

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

Types:
- Agent = pid() | atom()
- Reason = term()

Unloads the OTP-OS-MON-MIB.

See Also

os_mon(6) [page 4], [snmp(3)]
Erlang Module

os_sup is a process providing a message passing service from the operating system to the error logger in the Erlang runtime system. It is part of the OS_Mon application, see os_mon(6) [page 4]. Available for Solaris and Windows.

Messages received from the operating system results in an user defined callback function being called. This function can do whatever filtering and formatting is necessary and then deploy any type of logging suitable for the user’s application.

Solaris Operation

The Solaris (SunOS 5.x) messages are retrieved from the syslog-daemon, syslogd. Enabling the service includes actions which require root privileges, such as change of ownership and file privileges of an executable binary file, and creating a modified copy of the configuration file for syslogd. When os_sup is terminated, the service must be disabled, meaning the original configuration must be restored. Enabling/disabling can be done either outside or inside os_sup, see Configuration [page 21] below.

Warning:
This process cannot run in multiple instances on the same hardware. OS_Mon must be configured to start os_sup on one node only if two or more Erlang nodes execute on the same machine.

The format of received events is not defined.

Windows Operation

The Windows messages are retrieved from the eventlog file.

The nteventlog module is used to implement os_sup. See nteventlog(3) [page 17]. Note that the start functions of nteventlog does not need to be used, in this case the process is started automatically as part of the OS_Mon supervision tree.

OS messages are formatted as a tuple {Time, Category, Facility, Severity, Message}:

  Time = {MegaSecs, Secs, MicroSecs} A time stamp as returned by the BIF now().
Category = string() Usually one of "System", "Application" or "Security". Note that the NT eventlog viewer has another notion of category, which in most cases is totally meaningless and therefore not imported into Erlang. What is called a category here is one of the main three types of events occurring in a normal NT system.

Facility = string() The source of the message, usually the name of the application that generated it. This could be almost any string. When matching messages from certain applications, the version number of the application may have to be accounted for. This is what the NT event viewer calls “source”.

Severity = string() One of "Error", "Warning", "Informational", "Audit_Success", "Audit_Failure" or, in case of a currently unknown Windows NT version "Severity_Unknown".

Message = string() Formatted exactly as it would be in the NT eventlog viewer. Binary data is not imported into Erlang.

Configuration

os_sup_mfa = {Module, Function, Args} The callback function to use. Module and Function are atoms and Args is a list of terms. When an OS message Msg is received, this function is called as apply(Module, Function, [Msg | Args]). Default is [os_sup, error_report, [Tag]] which will send the event to the error logger using [error_logger:Error_report(Tag, Msg)]. Tag is the value of os_sup_errortag, see below.

os_sup_errortag = atom() This parameter defines the error report type used when messages are sent to error logger using the default callback function. Default is std_error, which means the events are handled by the standard event handler.

os_sup_enable = bool() Solaris only. Defines if the service should be enabled (and disabled) inside (true) or outside (false) os_sup. For backwards compatibility reasons, the default is true. The recommended value is false, as the Erlang emulator should normally not be run with root privileges, as is required for enabling the service.

os_sup_own = string() Solaris only. Defines the directory which contains the backup copy and the Erlang specific configuration files for syslogd, and a named pipe to receive the messages from syslogd. Default is "/etc".

os_sup_syslogconf = string() Solaris only. Defines the full name of the configuration file for syslogd. Default is "/etc/syslog.conf".

Exports

enable() -> ok | {error, Res}
enable(Dir, Conf) -> ok | {error, Error}

Types:
- Dir = Conf = Res = string()
Enables the os_sup service. Needed on Solaris only.

If the configuration parameter os_sup_enable is false, this function is called automatically by os_sup, using the values of os_sup_syslogdir and os_sup_syslogconf as arguments.

If os_sup_enable is true, this function must be called before OS_Mon/os_sup is started. Dir defines the directory which contains the backup copy and the Erlang specific configuration files for syslogd, and a named pipe to receive the messages from syslogd. Defaults to "/etc". Conf defines the full name of the configuration file for syslogd. Default is "/etc/syslog.conf".

Results in a OS call to:

<PRIVDIR>/bin/mod_syslog otp Dir Conf

where <PRIVDIR> is the priv directory of OS_Mon, code:priv_dir(os_mon).

Returns ok if this yields the expected result "0", and {error, Res} if it yields anything else.

Note:
This function requires root privileges to succeed.

disable() -> ok | {error, Res}
disable(Dir, Conf) -> ok | {error, Error}

Types:
• Dir = Conf = Res = string()

Disables the os_sup service. Needed on Solaris only.

If the configuration parameter os_sup_enable is false, this function is called automatically by os_sup, using the same arguments as when enable/2 was called.

If os_sup_enable is true, this function must be called after OS_Mon/os_sup is stopped. Dir defines the directory which contains the backup copy and the Erlang specific configuration files for syslogd, and a named pipe to receive the messages from syslogd. Defaults to "/etc". Conf defines the full name of the configuration file for syslogd. Default is "/etc/syslog.conf".

Results in a OS call to:

<PRIVDIR>/bin/mod_syslog nootp Dir Conf

where <PRIVDIR> is the priv directory of OS_Mon, code:priv_dir(os_mon).

Returns ok if this yields the expected result "0", and {error, Res} if it yields anything else.

Note:
This function requires root privileges to succeed.
See also

[error_logger(3)], os_mon(3) [page 4]

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.

avg1/0
   cpu_sup, 7
avg15/0
   cpu_sup, 7
avg5/0
   cpu_sup, 7
cpu_sup
   avg1/0, 7
   avg15/0, 7
   avg5/0, 7
   nprocs/0, 7
   util/0, 7
   util/1, 8
disable/0
   os_sup, 22
disable/2
   os_sup, 22
disksup
   get_almost_full_threshold/0, 11
   get_check_interval/0, 11
   get_disk_data/0, 11
   set_almost_full_threshold/1, 11
   set_check_interval/1, 11
enable/0
   os_sup, 21
enable/2
   os_sup, 21
get_almost_full_threshold/0
   disksup, 11
get_check_interval/0
   disksup, 11
   memsup, 15
get_disk_data/0

load/1
   os_mon_mib, 19

memsup
   get_check_interval/0, 15
   get_chunk_timeout/0, 16
   get_memory_data/0, 14
   get_procmem_high_watermark/0, 16
   get_system_high_watermark/0, 16
   get_system_memory_data/0, 15
   set_check_interval/1, 16
   set_chunk_timeout/1, 16
   set_procmem_high_watermark/1, 16
   set_system_high_watermark/1, 16
nprocs/0
   cpu_sup, 7
nteventlog
   start/2, 17
   start_link/2, 17
   stop/0, 18

os_mon_mib
   load/1, 19
   unload/1, 19
os_sup
  disable/0,22
  disable/2,22
  enable/0,21
  enable/2,21

set_almost_full_threshold/1
  disksup, 11

set_check_interval/1
  disksup, 11
  memsup, 16

set_helper_timeout/1
  memsup, 16

set_procmem_high_watermark/1
  memsup, 16

set_sysmem_high_watermark/1
  memsup, 16

start/2
  nteventlog, 17

start_link/2
  nteventlog, 17

stop/0
  nteventlog, 18

unload/1
  os_mon_mib, 19

util/0
  cpu_sup, 7

util/1
  cpu_sup, 8