NAME

Sys::Virt − Represent and manage a libvirt hypervisor connection

SYNOPSIS

```
my $conn = Sys::Virt->new(uri => $uri);

my @domains = $conn->list_domains();

foreach my $dom (@domains) {
  print "Domain ", $dom->get_id, " ", $dom->get_name, "\n";
}
```

DESCRIPTION

The Sys::Virt module provides a Perl XS binding to the libvirt virtual machine management APIs. This allows machines running within arbitrary virtualization containers to be managed with a consistent API.

ERROR HANDLING

Any operations in the Sys::Virt API which have failure scenarios will result in an instance of the Sys::Virt::Error module being thrown. To catch these errors, simply wrap the method in an eval block:

```
eval { my $conn = Sys::Virt->new(uri => $uri); };
if ($@) {
  print STDERR "Unable to open connection to $addr" . $@->message . "\n";
}
```

For details of the information contained in the error objects, consult the Sys::Virt::Error manual page.

METHODS

my `$conn` = Sys::Virt->new(uri => `$uri`, readonly => `$ro`, flags => `$flags`);

Attach to the virtualization host identified by `uri`. The `uri` parameter may be omitted, in which case the default connection made will be to the local Xen hypervisor. Some example URIs include:

xen:///
: Xen on the local machine

test:///default
: Dummy ‘‘in memory’’ driver for test suites

qemu:///system
: System-wide driver for QEMU / KVM virtualization

qemu:///session
: Per-user driver for QEMU virtualization

qemu+tls://somehost/system
: System-wide QEMU driver on `somehost` using TLS security

xen+tcp://somehost/
: Xen driver on `somehost` using TCP / SASL security

For further details consult `http://libvirt.org/uri.html`

If the optional `readonly` parameter is supplied, then an unprivileged connection to the hypervisor will be attempted. If it is not supplied, then it defaults to making a fully privileged connection to the hypervisor. If the calling application is not running as root, it may be necessary to provide authentication callbacks.

If the optional `auth` parameter is set to a non-zero value, authentication will be enabled during connection, using the default set of credential gathering callbacks. The default callbacks prompt for credentials on the console, so are not suitable for graphical applications. For such apps a custom implementation should be supplied. The `credlist` parameter should be an array reference listing the set of credential types that will be supported. The credential constants in this module can be used as values in this list. The `callback` parameter should be a subroutine reference containing the code

necessary to gather the credentials. When invoked it will be supplied with a single parameter, an array reference of requested credentials. The elements of the array are hash references, with keys `type` giving the type of credential, `prompt` giving a user descriptive user prompt, `challenge` giving name of the credential required. The answer should be collected from the user, and returned by setting the `result` key. This key may already be set with a default result if applicable

As a simple example returning hardcoded credentials

```
my $uri  = "qemu+tcp://192.168.122.1/system";
my $username = "test";
my $password = "123456";

my $con = Sys::Virt->new(uri => $uri,
                         auth => 1,
                         credlist => [
                           Sys::Virt::CRED_AUTHNAME,
                           Sys::Virt::CRED_PASSPHRASE,
                         ],
                         callback =>
     sub {
           my $creds = shift;

           foreach my $cred (@{$creds}) {
              if ($cred->{type} == Sys::Virt::CRED_AUTHNAME) {
                  $cred->{result} = $username;
              }
              if ($cred->{type} == Sys::Virt::CRED_PASSPHRASE) {
                  $cred->{result} = $password;
              }
           }
           return 0;
     });
```

For backwards compatibility with earlier releases, the `address` parameter is accepted as a synonym for the `uri` parameter. The use of `uri` is recommended for all newly written code.

`$conn`->set_identity($identity, `$flags`=0)

Change the identity that is used for access control over the connection. Normally the remote daemon will use the identity associated with the UNIX domain socket that the app opens. Only a privileged client is usually able to override this. The `$identity` should be a hash reference whose keys are one of the IDENTITY constants. The `$flags` parameter is currently unused, and defaults to 0 if omitted.

my `$st` = `$conn`->new_stream($flags)

Create a new stream, with the given flags

my `$dom` = `$conn`->create_domain($xml, `$flags`);

Create a new domain based on the XML description passed into the `$xml` parameter. The returned object is an instance of the Sys::Virt::Domain class. This method is not available with unprivileged connections to the hypervisor. The `$flags` parameter accepts one of the DOMAIN CREATION constants documented in Sys::Virt::Domain, and defaults to 0 if omitted.

my `$dom` = `$conn`->create_domain_with_files($xml, `$fds`, `$flags`);

Create a new domain based on the XML description passed into the `$xml` parameter. The returned object is an instance of the Sys::Virt::Domain class. This method is not available with unprivileged connections to the hypervisor. The `$fds` parameter is an array of UNIX file descriptors which will be passed to the init process of the container. This is only supported with container based virtualization. The `$flags` parameter accepts one of the DOMAIN CREATION constants documented in

Sys::Virt::Domain, and defaults to 0 if omitted.

my $dom = $conn->define_domain($xml, $flags=0);
: Defines, but does not start, a new domain based on the XML description passed into the $xml parameter. The returned object is an instance of the Sys::Virt::Domain class. This method is not available with unprivileged connections to the hypervisor. The defined domain can be later started by calling the create method on the returned Sys::Virt::Domain object.

my $net = $conn->create_network($xml, $flags=0);
: Create a new network based on the XML description passed into the $xml parameter. The returned object is an instance of the Sys::Virt::Network class. This method is not available with unprivileged connections to the hypervisor.

my $net = $conn->define_network($xml, $flags = 0);
: Defines, but does not start, a new network based on the XML description passed into the $xml parameter. The returned object is an instance of the Sys::Virt::Network class. This method is not available with unprivileged connections to the hypervisor. The defined network can be later started by calling the create method on the returned Sys::Virt::Network object.

my $nwfilter = $conn->define_nwfilter($xml, $flags=0);
: Defines a new network filter based on the XML description passed into the $xml parameter. The returned object is an instance of the Sys::Virt::NWFilter class. This method is not available with unprivileged connections to the hypervisor.

my $secret = $conn->define_secret($xml, $flags=0);
: Defines a new secret based on the XML description passed into the $xml parameter. The returned object is an instance of the Sys::Virt::Secret class. This method is not available with unprivileged connections to the hypervisor.

my $pool = $conn->create_storage_pool($xml);
: Create a new storage pool based on the XML description passed into the $xml parameter. The returned object is an instance of the Sys::Virt::StoragePool class. This method is not available with unprivileged connections to the hypervisor.

my $pool = $conn->define_storage_pool($xml, $flags=0);
: Defines, but does not start, a new storage pol based on the XML description passed into the $xml parameter. The returned object is an instance of the Sys::Virt::StoragePool class. This method is not available with unprivileged connections to the hypervisor. The defined pool can be later started by calling the create method on the returned Sys::Virt::StoragePool object.

my $pool = $conn->create_interface($xml);
: Create a new interface based on the XML description passed into the $xml parameter. The returned object is an instance of the Sys::Virt::Interface class. This method is not available with unprivileged connections to the hypervisor.

my $binding = $conn->create_nwfilter_binding($xml, $flags=0);
: Create a new network filter binding based on the XML description passed into the $xml parameter. The returned object is an instance of the Sys::Virt::NWFilterBinding class.

my $iface = $conn->define_interface($xml, $flags=0);
: Defines, but does not start, a new interface based on the XML description passed into the $xml parameter. The returned object is an instance of the Sys::Virt::Interface class. This method is not available with unprivileged connections to the hypervisor. The defined interface can be later started by calling the create method on the returned Sys::Virt::Interface object.

my $dev = $conn->create_node_device($xml, $flags=0);
: Create a new virtual node device based on the XML description passed into the $xml parameter. The $flags parameter is currently unused and defaults to zero. The returned object is an instance of the Sys::Virt::NodeDevice class. This method is not available with unprivileged connections to the hypervisor.

my $dev = $conn->define_node_device($xml, $flags=0);

Defines, but does not start, a new node dev based on the XML description passed into the $xml parameter. The $flags parameter is currently unused and defaults to zero. The returned object is an instance of the Sys::Virt::NodeDevice class. This method is not available with unprivileged connections to the hypervisor. The defined node device can be later started by calling the create method on the returned Sys::Virt::NodeDevice object.

my @doms = $conn->**list_domains()**

Return a list of all running domains currently known to the hypervisor. The elements in the returned list are instances of the Sys::Virt::Domain class. This method requires O(n) RPC calls, so the list_all_domains method is recommended as a more efficient alternative.

my $nids = $conn->**num_of_domains()**

Return the number of running domains known to the hypervisor. This can be used as the maxids parameter to list_domain_ids.

my @domIDs = $conn->list_domain_ids($maxids)

Return a list of all domain IDs currently known to the hypervisor. The IDs can be used with the get_domain_by_id method.

my @doms = $conn->**list_defined_domains()**

Return a list of all domains defined, but not currently running, on the hypervisor. The elements in the returned list are instances of the Sys::Virt::Domain class. This method requires O(n) RPC calls, so the list_all_domains method is recommended as a more efficient alternative.

my $ndoms = $conn->**num_of_defined_domains()**

Return the number of running domains known to the hypervisor. This can be used as the maxnames parameter to list_defined_domain_names.

my @names = $conn->list_defined_domain_names($maxnames)

Return a list of names of all domains defined, but not currently running, on the hypervisor. The names can be used with the get_domain_by_name method.

my @doms = $conn->list_all_domains($flags)

Return a list of all domains currently known to the hypervisor, whether running or shutoff. The elements in the returned list are instances of the Sys::Virt::Domain class. The $flags parameter can be used to filter the list of returned domains.

my @nets = $conn->**list_networks()**

Return a list of all networks currently known to the hypervisor. The elements in the returned list are instances of the Sys::Virt::Network class. This method requires O(n) RPC calls, so the list_all_networks method is recommended as a more efficient alternative.

my $nnets = $conn->**num_of_networks()**

Return the number of running networks known to the hypervisor. This can be used as the maxids parameter to list_network_ids.

my @netNames = $conn->list_network_names($maxnames)

Return a list of all network names currently known to the hypervisor. The names can be used with the get_network_by_name method.

my @nets = $conn->**list_defined_networks()**

Return a list of all networks defined, but not currently running, on the hypervisor. The elements in the returned list are instances of the Sys::Virt::Network class. This method requires O(n) RPC calls, so the list_all_networks method is recommended as a more efficient alternative.

my $nnets = $conn->**num_of_defined_networks()**

Return the number of running networks known to the host. This can be used as the maxnames parameter to list_defined_network_names.

my @names = $conn->list_defined_network_names($maxnames)
: Return a list of names of all networks defined, but not currently running, on the host. The names can be used with the `get_network_by_name` method.

my @nets = $conn->list_all_networks($flags)
: Return a list of all networks currently known to the hypervisor, whether running or shutoff. The elements in the returned list are instances of the Sys::Virt::Network class. The `$flags` parameter can be used to filter the list of returned networks.

my @pools = $conn->**list_storage_pools()**
: Return a list of all storage pools currently known to the host. The elements in the returned list are instances of the Sys::Virt::StoragePool class. This method requires O(n) RPC calls, so the `list_all_storage_pools` method is recommended as a more efficient alternative.

my $npools = $conn->**num_of_storage_pools()**
: Return the number of running storage pools known to the hypervisor. This can be used as the `maxids` parameter to `list_storage_pool_names`.

my @poolNames = $conn->list_storage_pool_names($maxnames)
: Return a list of all storage pool names currently known to the hypervisor. The IDs can be used with the `get_network_by_id` method.

my @pools = $conn->**list_defined_storage_pools()**
: Return a list of all storage pools defined, but not currently running, on the host. The elements in the returned list are instances of the Sys::Virt::StoragePool class. This method requires O(n) RPC calls, so the `list_all_storage_pools` method is recommended as a more efficient alternative.

my $npools = $conn->**num_of_defined_storage_pools()**
: Return the number of running networks known to the host. This can be used as the `maxnames` parameter to `list_defined_storage_pool_names`.

my @names = $conn->list_defined_storage_pool_names($maxnames)
: Return a list of names of all storage pools defined, but not currently running, on the host. The names can be used with the `get_storage_pool_by_name` method.

my @pools = $conn->list_all_storage_pools($flags)
: Return a list of all storage pools currently known to the hypervisor, whether running or shutoff. The elements in the returned list are instances of the Sys::Virt::StoragePool class. The `$flags` parameter can be used to filter the list of returned pools.

my @devs = $conn->list_node_devices($capability)
: Return a list of all devices currently known to the host OS. The elements in the returned list are instances of the Sys::Virt::NodeDevice class. The optional `capability` parameter allows the list to be restricted to only devices with a particular capability type. This method requires O(n) RPC calls, so the `list_all_node_devices` method is recommended as a more efficient alternative.

my $ndevs = $conn->num_of_node_devices($capability[, `$flags`])
: Return the number of host devices known to the hypervisor. This can be used as the `maxids` parameter to `list_node_device_names`. The`capability` parameter allo ws the list to be restricted to only devices with a particular capability type, and should be left as `undef` if the full list is required. The optional <flags> parameter is currently unused and defaults to 0 if omitted.

my @devNames = $conn->list_node_device_names($capability, `$maxnames`[, `$flags`])
: Return a list of all host device names currently known to the hypervisor. The names can be used with the `get_node_device_by_name` method. The `capability` parameter allows the list to be restricted to only devices with a particular capability type, and should be left as `undef` if the full list is required. The optional <flags> parameter is currently unused and defaults to 0 if omitted.

my @devs = $conn->list_all_node_devices($flags)
: Return a list of all node devices currently known to the hypervisor. The elements in the returned list are instances of the Sys::Virt::NodeDevice class. The `$flags` parameter can be used to filter the list of returned devices.

my `@ifaces` = `$conn`->**list_interfaces()**
Return a list of all network interfaces currently known to the hypervisor. The elements in the returned list are instances of the Sys::Virt::Interface class. This method requires O(n) RPC calls, so the `list_all_interfaces` method is recommended as a more efficient alternative.

my `$nifaces` = `$conn`->**num_of_interfaces()**
Return the number of running interfaces known to the hypervisor. This can be used as the `maxnames` parameter to `list_interface_names`.

my `@names` = `$conn`->list_interface_names($maxnames)
Return a list of all interface names currently known to the hypervisor. The names can be used with the `get_interface_by_name` method.

my `@ifaces` = `$conn`->**list_defined_interfaces()**
Return a list of all network interfaces currently known to the hypervisor. The elements in the returned list are instances of the Sys::Virt::Interface class. This method requires O(n) RPC calls, so the `list_all_interfaces` method is recommended as a more efficient alternative.

my `$nifaces` = `$conn`->**num_of_defined_interfaces()**
Return the number of inactive interfaces known to the hypervisor. This can be used as the `maxnames` parameter to `list_defined_interface_names`.

my `@names` = `$conn`->list_defined_interface_names($maxnames)
Return a list of inactive interface names currently known to the hypervisor. The names can be used with the `get_interface_by_name` method.

my `@ifaces` = `$conn`->list_all_interfaces($flags)
Return a list of all interfaces currently known to the hypervisor, whether running or shutoff. The elements in the returned list are instances of the Sys::Virt::Interface class. The `$flags` parameter can be used to filter the list of returned interfaces.

my `@ifaces` = `$conn`->**list_secrets()**
Return a list of all secrets currently known to the hypervisor. The elements in the returned list are instances of the Sys::Virt::Secret class. This method requires O(n) RPC calls, so the `list_all_secrets` method is recommended as a more efficient alternative.

my `$nuuids` = `$conn`->**num_of_secrets()**
Return the number of secrets known to the hypervisor. This can be used as the `maxuuids` parameter to `list_secrets`.

my `@uuids` = `$conn`->list_secret_uuids($maxuuids)
Return a list of all secret uuids currently known to the hypervisor. The uuids can be used with the `get_secret_by_uuid` method.

my `@secrets` = `$conn`->list_all_secrets($flags)
Return a list of all secrets currently known to the hypervisor. The elements in the returned list are instances of the Sys::Virt::Network class. The `$flags` parameter can be used to filter the list of returned secrets.

my `@nwfilters` = `$conn`->**list_nwfilters()**
Return a list of all nwfilters currently known to the hypervisor. The elements in the returned list are instances of the Sys::Virt::NWFilter class. This method requires O(n) RPC calls, so the `list_all_nwfilters` method is recommended as a more efficient alternative.

my `$nnwfilters` = `$conn`->**num_of_nwfilters()**
Return the number of running nwfilters known to the hypervisor. This can be used as the `maxids` parameter to `list_nwfilter_names`.

my `@filterNames` = `$conn`->list_nwfilter_names($maxnames)
Return a list of all nwfilter names currently known to the hypervisor. The names can be used with the `get_nwfilter_by_name` method.

my @nwfilters = $conn->list_all_nwfilters($flags)
Return a list of all nwfilters currently known to the hypervisor. The elements in the returned list are instances of the Sys::Virt::NWFilter class. The $flags parameter is currently unused and defaults to zero.

my @bindings = $conn->list_all_nwfilter_bindings($flags)
Return a list of all nwfilter bindings currently known to the hypervisor. The elements in the returned list are instances of the Sys::Virt::NWFilterBinding class. The $flags parameter is currently unused and defaults to zero.

$conn->define_save_image_xml($file, $dxml, $flags=0)
Update the XML associated with a virtual machine's save image. The $file parameter is the fully qualified path to the save image XML, while $dxml is the new XML document to write. The $flags parameter is currently unused and defaults to zero.

$xml = $conn->get_save_image_xml_description($file, $flags=1)
Retrieve the current XML configuration associated with the virtual machine's save image identified by $file. The $flags parameter accepts the same constants as Sys::Virt::Domain::managed_save_get_xml_description.

my $dom = $conn->get_domain_by_name($name)
Return the domain with a name of $name. The returned object is an instance of the Sys::Virt::Domain class.

my $dom = $conn->get_domain_by_id($id)
Return the domain with a local id of $id. The returned object is an instance of the Sys::Virt::Domain class.

my $dom = $conn->get_domain_by_uuid($uuid)
Return the domain with a globally unique id of $uuid. The returned object is an instance of the Sys::Virt::Domain class.

my $net = $conn->get_network_by_name($name)
Return the network with a name of $name. The returned object is an instance of the Sys::Virt::Network class.

my $net = $conn->get_network_by_uuid($uuid)
Return the network with a globally unique id of $uuid. The returned object is an instance of the Sys::Virt::Network class.

my $pool = $conn->get_storage_pool_by_name($name)
Return the storage pool with a name of $name. The returned object is an instance of the Sys::Virt::StoragePool class.

my $pool = $conn->get_storage_pool_by_uuid($uuid)
Return the storage pool with a globally unique id of $uuid. The returned object is an instance of the Sys::Virt::StoragePool class.

my $pool = $conn->get_storage_pool_by_volume($vol)
Return the storage pool with a storage volume $vol. The $vol parameter must be an instance of the Sys::Virt::StorageVol class. The returned object is an instance of the Sys::Virt::StoragePool class.

my $pool = $conn->get_storage_pool_by_target_path($path)
Return the storage pool with a target path of $path. The returned object is an instance of the Sys::Virt::StoragePool class.

my $vol = $conn->get_storage_volume_by_path($path)
Return the storage volume with a location of $path. The returned object is an instance of the Sys::Virt::StorageVol class.

my $vol = $conn->get_storage_volume_by_key($key)
Return the storage volume with a globally unique id of $key. The returned object is an instance of the Sys::Virt::StorageVol class.

my $dev = $conn->get_node_device_by_name($name)
Return the node device with a name of $name. The returned object is an instance of the Sys::Virt::NodeDevice class.

my $dev = $conn->get_node_device_scsihost_by_wwn($wwnn, $wwpn, $flags=0)
Return the node device which is a SCSI host identified by $wwnn and $wwpn. The$flags parameter is unused and defaults to zero. The returned object is an instance of the Sys::Virt::NodeDevice class.

my $iface = $conn->get_interface_by_name($name)
Return the interface with a name of $name. The returned object is an instance of the Sys::Virt::Interface class.

my $iface = $conn->get_interface_by_mac($mac)
Return the interface with a MAC address of $mac. The returned object is an instance of the Sys::Virt::Interface class.

my $sec = $conn->get_secret_by_uuid($uuid)
Return the secret with a globally unique id of $uuid. The returned object is an instance of the Sys::Virt::Secret class.

my $sec = $conn->get_secret_by_usage($usageType, $usageID)
Return the secret with a usage type of $usageType, identified by $usageID. The returned object is an instance of the Sys::Virt::Secret class.

my $nwfilter = $conn->get_nwfilter_by_name($name)
Return the domain with a name of $name. The returned object is an instance of the Sys::Virt::NWFilter class.

my $nwfilter = $conn->get_nwfilter_by_uuid($uuid)
Return the nwfilter with a globally unique id of $uuid. The returned object is an instance of the Sys::Virt::NWFilter class.

my $binding = $conn->get_nwfilter_binding_by_port_dev($name)
Return the network filter binding for the port device $name. The returned object is an instance of the Sys::Virt::NWFilterBinding class.

my $xml = $conn->find_storage_pool_sources($type, $srcspec[, $flags])
Probe for available storage pool sources for the pool of type $type. The$srcspec parameter can be undef, or a parameter to refine the discovery process, for example a server hostname for NFS discovery. The $flags parameter is optional, and if omitted defaults to zero. The returned scalar is an XML document describing the discovered storage pool sources.

my @stats = $conn->get_all_domain_stats($stats, \@doms=undef, $flags=0);
Get a list of all statistics for domains known to the hypervisor. The$stats parameter controls which data fields to return and should be a combination of the DOMAIN STATS FIELD CONSTANTS.

The optional @doms parameter is a list of Sys::Virt::Domain objects to return stats for. If this is undefined, then all domains will be returned. The $flags method can be used to filter the list of returned domains.

The return data for the method is a list, one element for each domain. The element will be a hash with two keys, dom pointing to an instance of Sys::Virt::Domain and data pointing to another hash reference containing the actual statistics.

$conn->interface_change_begin($flags)
Begin a transaction for changing the configuration of one or more network interfaces

$conn->interface_change_commit($flags)
Complete a transaction for changing the configuration of one or more network interfaces

`$conn`->interface_change_rollback($flags)
Abort a transaction for changing the configuration of one or more network interfaces

`$conn`->restore_domain($savefile)
Recreate a domain from the saved state file given in the `$savefile` parameter.

`$conn`->get_max_vcpus($domtype)
Return the maximum number of vcpus that can be configured for a domain of type `$domtype`

my `$hostname` = `$conn`->**get_hostname()**
Return the name of the host with which this connection is associated.

my `$uri` = `$conn`->**get_uri()**
Return the URI associated with the open connection. This may be different from the URI used when initially connecting to libvirt, when 'auto-probing' or drivers occurrs.

my `$xml` = `$conn`->**get_sysinfo()**
Return an XML documenting representing the host system information, typically obtained from SMBIOS tables.

my `$type` = `$conn`->**get_type()**
Return the type of virtualization backend accessed by this hypervisor object. Currently the only supported type is `Xen`.

my `$xml` = `$conn`->domain_xml_from_native($format, `$config`);
Convert the native hypervisor configuration `$config` which is in format <$format> into libvirrt domain XML. Valid values of `$format` vary between hypervisor drivers.

my `$config` = `$conn`->domain_xml_to_native($format, `$xml`)
Convert the libvirt domain XML configuration `$xml` to a native hypervisor configuration in format `$format`

my `$ver` = `$conn`->**get_version()**
Return the complete version number as a string encoded in the formula `(major * 1000000) + (minor * 1000) + micro`.

my `$ver` = `$conn`->get_major_version
Return the major version number of the libvirt library.

my `$ver` = `$conn`->get_minor_version
Return the minor version number of the libvirt library.

my `$ver` = `$conn`->get_micro_version
Return the micro version number of the libvirt library.

my `$ver` = `$conn`->get_library_version
Return the version number of the API associated with the active connection. This differs from `get_version` in that if the connection is to a remote libvirtd daemon, it will return the API version of the remote libvirt, rather than the local client.

`$conn`->**is_secure()**
Returns a true value if the current connection is secure against network interception. This implies either use of UNIX sockets, or encryption with a TCP stream.

`$conn`->**is_encrypted()**
Returns a true value if the current connection data stream is encrypted.

`$conn`->**is_alive()**
Returns a true value if the connection is alive, as determined by keep-alive packets or other recent RPC traffic.

`$conn`->set_keep_alive($interval, `$count`)
Change the operation of the keep alive protocol to send `$count` packets spaced `$interval` seconds apart before considering the connection dead.

my $info = $con->**get_node_info()**

Returns a hash reference summarising the capabilities of the host node. The elements of the hash are as follows:

memory
: The amount of physical memory in the host

model
: The model of the CPU, eg x86_64

cpus
: The total number of logical CPUs.

mhz
: The peak MHZ of the CPU

nodes
: The number of NUMA cells

sockets
: The number of CPU sockets

cores
: The number of cores per socket

threads
: The number of threads per core

NB, more accurate information about the total number of CPUs and those online can be obtained using the `get_node_cpu_map` method.

my ($totcpus, $onlinemap, $totonline) = $con->**get_node_cpu_map()**;

Returns an array containing information about the CPUs available on the host. The first element, `totcpus`, specifies the total number of CPUs available to the host regardles of their online stat. The second element, `onlinemap`, provides a bitmap detailing which CPUs are currently online. The third element, `totonline`, specifies the total number of online CPUs. The values in the bitmap can be extracted using the `unpack` method as follows:

```
my @onlinemap = split(//, unpack("b*", $onlinemap));
```

my $info = $con->get_node_cpu_stats($cpuNum=-1, $flags=0)

Returns a hash reference providing information about the host CPU statistics. If <$cpuNum> is omitted, it defaults to `Sys::Virt::NODE_CPU_STATS_ALL_CPUS` which causes it to return cumulative information for all CPUs in the host. If `$cpuNum` is zero or larger, it returns information just for the specified number. The `$flags` parameter is currently unused and defaults to zero. The fields in the returned hash reference are

kernel
: The time spent in kernelspace

user
: The time spent in userspace

idle The idle time

iowait
: The I/O wait time

utilization
: The overall percentage utilization.

my $info = $con->get_node_memory_stats($cellNum=-1, $flags=0)

Returns a hash reference providing information about the host memory statistics. If <$cellNum> is omitted, it defaults to `Sys::Virt::NODE_MEMORY_STATS_ALL_CELLS` which causes it to return cumulative information for all NUMA cells in the host. If `$cellNum` is zero or larger, it returns

information just for the specified number. The `$flags` parameter is currently unused and defaults to zero. The fields in the returned hash reference are

total
: The total memory

free
: The free memory

buffers
: The memory consumed by buffers

cached
: The memory consumed for cache

my `$params` = `$conn`->get_node_memory_parameters($flags=0)
: Return a hash reference containing the set of memory tunable parameters for the node. The keys in the hash are one of the constants MEMORY PARAMETERS described later. The `$flags` parameter is currently unused, and defaults to 0 if omitted.

`$conn`->set_node_memory_parameters($params, `$flags`=0)
: Update the memory tunable parameters for the node. The `$params` should be a hash reference whose keys are one of the MEMORY PARAMETERS constants. The `$flags` parameter is currently unused, and defaults to 0 if omitted.

`$info` = `$conn`->get_node_sev_info($flags=0)
: Get the AMD SEV information for the host. `$flags` is currently unused and defaults to 0 if omitted. The returned hash contains the following keys:

  Sys::Virt::SEV_CBITPOS
  : The CBit position

  Sys::Virt::SEV_CERT_CHAIN
  : The certificate chain

  Sys::Virt::SEV_PDH
  : Platform diffie-hellman key

  Sys::Virt::SEV_REDUCED_PHYS_BITS
  : The number of physical address bits used by SEV

  Sys::Virt::SEV_MAX_GUESTS
  : Maximum number of SEV guests that can be launched

  Sys::Virt::SEV_MAX_ES_GUESTS
  : Maximum number of SEV-ES guests that can be launched

`$conn`->node_suspend_for_duration($target, `$duration`, `$flags`=0)
: Suspend the the host, using mode `$target` which is one of the NODE SUSPEND constants listed later. The `$duration` parameter controls how long the node is suspended for before waking up.

`$conn`->domain_event_register($callback)
: Register a callback to received notifications of domain state change events. Only a single callback can be registered with each connection instance. The callback will be invoked with four parameters, an instance of `Sys::Virt` for the connection, an instance of `Sys::Virt::Domain` for the domain changing state, and a `event` and `detail` arguments, corresponding to the event constants defined in the `Sys::Virt::Domain` module. Before discarding the connection object, the callback must be deregistered, otherwise the connection object memory will never be released in garbage collection.

`$conn`->**domain_event_deregister()**
: Unregister a callback, allowing the connection object to be garbage collected.

`$callback` = `$conn`->domain_event_register_any($dom, `$eventID`, `$callback`)
: Register a callback to received notifications of domain events. The`$dom` parameter can be `undef` to request events on all known domains, or a specific `Sys::Virt::Domain` object to filter events.

The `$eventID` parameter is one of the EVENT ID constants described later in this document. The `$callback` is a subroutine reference that will receive the events.

All callbacks receive a `Sys::Virt` connection as the first parameter and a `Sys::Virt::Domain` object indicating the domain on which the event occurred as the second parameter. Subsequent parameters vary according to the event type

EVENT_ID_LIFECYCLE
: Extra `event` and `detail` parameters defining the lifecycle transition that occurred.

EVENT_ID_REBOOT
: No extra parameters

EVENT_ID_RTC_CHANGE
: The `utcoffset` gives the offset from UTC in seconds

EVENT_ID_WATCHDOG
: The `action` defines the action that is taken as a result of the watchdog triggering. One of the WATCHDOG constants described later

EVENT_ID_IO_ERROR
: The `srcPath` is the file on the host which had the error. The`devAlias` is the unique de vice alias from the guest configuration associated with `srcPath`. The `action` is the action taken as a result of the error, one of the IO ERROR constants described later

EVENT_ID_GRAPHICS
: The `phase` is the stage of the connection, one of the GRAPHICS PHASE constants described later. The `local` and `remote` parameters follow with the details of the local and remote network addresses. The `authScheme` describes how the user was authenticated (if at all). Finally `identities` is an array ref containing authenticated identities for the user, if any.

The return value is a unique callback ID that must be used when unregistering the event.

`$conn`->domain_event_deregister_any($callbackID)
: Unregister a callback, associated with the `$callbackID` previously obtained from `domain_event_register_any`.

`$callback` = `$conn`->network_event_register_any($net, `$eventID`, `$callback`)
: Register a callback to received notifications of network events. The`$net` parameter can be `undef` to request events on all known networks, or a specific `Sys::Virt::Network` object to filter events. The `$eventID` parameter is one of the EVENT ID constants described later in this document. The `$callback` is a subroutine reference that will receive the events.

  All callbacks receive a `Sys::Virt` connection as the first parameter and a `Sys::Virt::Network` object indicating the network on which the event occurred as the second parameter. Subsequent parameters vary according to the event type

  EVENT_ID_LIFECYCLE
  : Extra `event` and `detail` parameters defining the lifecycle transition that occurred.

  The return value is a unique callback ID that must be used when unregistering the event.

`$conn`->network_event_deregister_any($callbackID)
: Unregister a callback, associated with the `$callbackID` previously obtained from `network_event_register_any`.

`$callback` = `$conn`->storage_pool_event_register_any($pool, `$eventID`, `$callback`)
: Register a callback to received notifications of storage pool events. The`$pool` parameter can be `undef` to request events on all known storage pools, or a specific `Sys::Virt::StoragePool` object to filter events. The `$eventID` parameter is one of the EVENT ID constants described later in this document. The `$callback` is a subroutine reference that will receive the events.

  All callbacks receive a `Sys::Virt` connection as the first parameter and a `Sys::Virt::StoragePool` object indicating the storage pool on which the event occurred as the

second parameter. Subsequent parameters vary according to the event type

EVENT_ID_LIFECYCLE
Extra `event` and `detail` parameters defining the lifecycle transition that occurred.

EVENT_ID_REFRESH
No extra parameters.

The return value is a unique callback ID that must be used when unregistering the event.

`$conn`->storage_pool_event_deregister_any($callbackID)
Unregister a callback, associated with the `$callbackID` previously obtained from `storage_pool_event_register_any`.

`$callback` = `$conn`->node_device_event_register_any($dev, `$eventID`, `$callback`)
Register a callback to received notifications of node device events. The`$dev` parameter can be `undef` to request events on all known node devices, or a specific `Sys::Virt::NodeDevice` object to filter events. The `$eventID` parameter is one of the EVENT ID constants described later in this document. The `$callback` is a subroutine reference that will receive the events.

All callbacks receive a `Sys::Virt` connection as the first parameter and a `Sys::Virt::NodeDevice` object indicating the node device on which the event occurred as the second parameter. Subsequent parameters vary according to the event type

EVENT_ID_LIFECYCLE
Extra `event` and `detail` parameters defining the lifecycle transition that occurred.

The return value is a unique callback ID that must be used when unregistering the event.

`$conn`->node_device_event_deregister_any($callbackID)
Unregister a callback, associated with the `$callbackID` previously obtained from `node_device_event_register_any`.

`$callback` = `$conn`->secret_event_register_any($secret, `$eventID`, `$callback`)
Register a callback to received notifications of secret events. The`$secret` parameter can be `undef` to request events on all known secrets, or a specific `Sys::Virt::Secret` object to filter events. The `$eventID` parameter is one of the EVENT ID constants described later in this document. The `$callback` is a subroutine reference that will receive the events.

All callbacks receive a `Sys::Virt` connection as the first parameter and a `Sys::Virt::Secret` object indicating the secret on which the event occurred as the second parameter. Subsequent parameters vary according to the event type

EVENT_ID_LIFECYCLE
Extra `event` and `detail` parameters defining the lifecycle transition that occurred.

EVENT_ID_VALUE_CHANGED
No extra parameters.

The return value is a unique callback ID that must be used when unregistering the event.

`$conn`->secret_event_deregister_any($callbackID)
Unregister a callback, associated with the `$callbackID` previously obtained from `secret_event_register_any`.

`$conn`->register_close_callback($coderef);
Register a callback to be invoked when the connection is closed. The callback will be invoked with two parameters, the `$conn` it was registered against, and the reason for the close event. The reason value will be one of the `CLOSE REASON CONSTANTS` listed later in this document.

`$conn`->**unregister_close_callback()**;
Remove the previously registered close callback.

my $xml = $con->baseline_cpu(\@xml, $flags=0)

Given an array ref whose elements are XML documents describing host CPUs, compute the baseline CPU model that is operable across all hosts. The XML for the baseline CPU model is returned. The optional $flags parameter can take one of

Sys::Virt::BASELINE_CPU_EXPAND_FEATURES

Expand the CPU definition to list all feature flags, even those implied by the model name.

Sys::Virt::BASELINE_CPU_MIGRATABLE

Only include features which can be live migrated.

my $xml = $con->baseline_hypervisor_cpu($emulator, $arch, $machine, $virttype, \@xml, $flags=0)

Given an array ref whose elements are XML documents describing host CPUs, compute the baseline CPU model that is operable across all hosts. The XML for the baseline CPU model is returned. Either $emulator or $arch must be a valid string referring to an emulator binary or an architecture name respectively. The $machine parameter is an optional name of a guest machine, and $virttype is an optional name of the virtualization type. The optional $flags parameter accepts the same values as baseline_cpu.

@names = $con->get_cpu_model_names($arch, $flags=0)

Get a list of valid CPU models names for the architecture given by $arch. The $arch value should be one of the architectures listed in the capabilities XML document. The $flags parameter is currently unused and defaults to 0.

my $info = $con->**get_node_security_model()**

Returns a hash reference summarising the security model of the host node. There are two keys in the hash, model specifying the name of the security model (eg 'selinux') and doi specifying the 'domain of interpretation' for security labels.

my $xml = $con->**get_capabilities()**;

Returns an XML document describing the hypervisor capabilities

my $xml = $con->get_domain_capabilities($emulator, $arch, $machine, $virttype, flags=0);

Returns an XML document describing the capabilities of the requested guest configuration. Either $emulator or $arch must be a valid string referring to an emulator binary or an architecture name respectively. The $machine parameter is an optional name of a guest machine, and $virttype is an optional name of the virtualization type. $flags is unused and defaults to zero.

my $xml = $con->get_storage_pool_capabilities($flags=0);

Returns an XML document describing the storage pool driver capabilities (e.g. which storage pool types are supported and so on). $flags is currently unused and defaults to zero.

my $result = $con->compare_cpu($xml, $flags=0);

Checks whether the CPU definition in $xml is compatible with the current hypervisor connection. This can be used to determine whether it is safe to migrate a guest to this host. The returned result is one of the constants listed later The optional $flags parameter can take one of the following constants

Sys::Virt::COMPARE_CPU_FAIL_INCOMPATIBLE

Raise a fatal error if the CPUs are not compatible, instead of just returning a special error code.

Sys::Virt::COMPARE_CPU_VALIDATE_XML

Validate input XML document against the RNG schema.

my $result = $con->compare_hypervisor_cpu($emulator, $arch, $machine, $virttype, $xml, $flags=0);

Checks whether the CPU definition in $xml is compatible with the current hypervisor connection. This can be used to determine whether it is safe to migrate a guest to this host. Either $emulator or $arch must be a valid string referring to an emulator binary or an architecture name respectively. The $machine parameter is an optional name of a guest machine, and $virttype is an optional name of the virtualization type. The returned result is one of the constants listed later The optional $flags

parameter can take the same values as the `compare_cpu` method.

`$mem` = `$con`->**get_node_free_memory()**;
Returns the current free memory on the host

`@mem` = `$con`->get_node_cells_free_memory($start, `$end`);
Returns the free memory on each NUMA cell between `$start` and `$end`.

`@pages` = `$con`->get_node_free_pages(\@pagesizes, `$start`, `$end`);
Returns information about the number of pages free on each NUMA cell between `$start` and `$end` inclusive. The `@pagesizes` parameter should be an arrayref specifying which pages sizes information should be returned for. Information about supported page sizes is available in the capabilities XML. The returned array has an element for each NUMA cell requested. The elements are hash references with two keys, 'cell' specifies the NUMA cell number and 'pages' specifies the free page information for that cell. The 'pages' value is another hash reference where the keys are the page sizes and the values are the free count for that size.

`$con`->node_alloc_pages(\@pages, `$start`, `$end`, `$flags`=0)
Allocate further huge pages for the reserved dev. The <\@pages> parameter is an array reference with one entry per page size to allocate for. Each entry is a further array reference where the first element is the page size and the second element is the page count. The same number of pages will be allocated on each NUMA node in the range `$start` to `$end` inclusive. The `$flags` parameter accepts two contants

Sys::Virt::NODE_ALLOC_PAGES_ADD
The requested number of pages will be added to the existing huge page reservation.

Sys::Virt::NODE_ALLOC_PAGES_SET
The huge page reservation will be set to exactly the requested number

CONSTANTS

The following sets of constants are useful when dealing with APIs in this package

CONNECTION

When opening a connection the following constants can be used:

Sys::Virt::CONNECT_RO
Request a read-only connection

Sys::Virt::CONNECT_NO_ALIASES
Prevent the resolution of URI aliases

CREDENTIAL TYPES

When providing authentication callbacks, the following constants indicate the type of credential being requested

Sys::Virt::CRED_AUTHNAME
Identity to act as

Sys::Virt::CRED_USERNAME
Identity to authorize as

Sys::Virt::CRED_CNONCE
Client supplies a nonce

Sys::Virt::CRED_REALM
Authentication realm

Sys::Virt::CRED_ECHOPROMPT
Challenge response non-secret

Sys::Virt::CRED_NOECHOPROMPT
Challenge response secret

Sys::Virt::CRED_PASSPHRASE
Passphrase secret

Sys::Virt::CRED_LANGUAGE
RFC 1766 language code

Sys::Virt::CRED_EXTERNAL
Externally provided credential

IDENTITY CONSTANTS

The following constants are useful to change the connection identity

Sys::Virt::IDENTITY_USER_NAME
The process user name

Sys::Virt::IDENTITY_UNIX_USER_ID
The process UNIX user ID

Sys::Virt::IDENTITY_GROUP_NAME
The process group name

Sys::Virt::IDENTITY_UNIX_GROUP_ID
The process UNIX group ID

Sys::Virt::IDENTITY_PROCESS_ID
The process ID.

Sys::Virt::IDENTITY_PROCESS_TIME
The process start time.

Sys::Virt::IDENTITY_SASL_USER_NAME
The SASL authenticated user name

Sys::Virt::IDENTITY_X509_DISTINGUISHED_NAME
The X509 certificate distinguished name for the TLS connection

Sys::Virt::IDENTITY_SELINUX_CONTEXT
The SELinux process context

CPU COMPARISON CONSTANTS

Sys::Virt::CPU_COMPARE_INCOMPATIBLE
This host is missing one or more CPU features in the CPU description

Sys::Virt::CPU_COMPARE_IDENTICAL
The host has an identical CPU description

Sys::Virt::CPU_COMPARE_SUPERSET
The host offers a superset of the CPU descriptoon

NODE SUSPEND CONSTANTS

Sys::Virt::NODE_SUSPEND_TARGET_MEM
Suspends to memory (equivalent of S3 on x86 architectures)

Sys::Virt::NODE_SUSPEND_TARGET_DISK
Suspends to disk (equivalent of S5 on x86 architectures)

Sys::Virt::NODE_SUSPEND_TARGET_HYBRID
Suspends to memory and disk (equivalent of S3+S5 on x86 architectures)

NODE VCPU CONSTANTS

Sys::Virt::NODE_CPU_STATS_ALL_CPUS
Request statistics for all CPUs

NODE MEMORY CONSTANTS

Sys::Virt::NODE_MEMORY_STATS_ALL_CELLS
Request statistics for all memory cells

MEMORY PARAMETERS

The following constants are used to name memory parameters of the node

Sys::Virt::NODE_MEMORY_SHARED_FULL_SCANS
How many times all mergeable areas have been scanned.

Sys::Virt::NODE_MEMORY_SHARED_PAGES_SHARED
How many the shared memory pages are being used.

Sys::Virt::NODE_MEMORY_SHARED_PAGES_SHARING
How many sites are sharing the pages

Sys::Virt::NODE_MEMORY_SHARED_PAGES_TO_SCAN
How many present pages to scan before the shared memory service goes to sleep

Sys::Virt::NODE_MEMORY_SHARED_PAGES_UNSHARED
How many pages unique but repeatedly checked for merging.

Sys::Virt::NODE_MEMORY_SHARED_PAGES_VOLATILE
How many pages changing too fast to be placed in a tree.

Sys::Virt::NODE_MEMORY_SHARED_SLEEP_MILLISECS
How many milliseconds the shared memory service should sleep before next scan.

Sys::Virt::NODE_MEMORY_SHARED_MERGE_ACROSS_NODES
Whether pages can be merged across NUMA nodes

CLOSE REASON CONSTANTS

The following constants related to the connection close callback, describe the reason for the closing of the connection.

Sys::Virt::CLOSE_REASON_CLIENT
The client application requested the connection be closed

Sys::Virt::CLOSE_REASON_EOF
End-of-file was encountered reading data from the connection

Sys::Virt::CLOSE_REASON_ERROR
An I/O error was encountered reading/writing data from/to the connection

Sys::Virt::CLOSE_REASON_KEEPALIVE
The connection keepalive timer triggered due to lack of response from the server

CPU STATS CONSTANTS

The following constants provide the names of known CPU stats fields

Sys::Virt::NODE_CPU_STATS_IDLE
Time spent idle

Sys::Virt::NODE_CPU_STATS_IOWAIT
Time spent waiting for I/O to complete

Sys::Virt::NODE_CPU_STATS_KERNEL
Time spent executing kernel code

Sys::Virt::NODE_CPU_STATS_USER
Time spent executing user code

Sys::Virt::NODE_CPU_STATS_INTR
Time spent processing interrupts

Sys::Virt::NODE_CPU_STATS_UTILIZATION
Percentage utilization of the CPU.

MEMORY STAS CONSTANTS

The following constants provide the names of known memory stats fields

Sys::Virt::NODE_MEMORY_STATS_BUFFERS
: The amount of memory consumed by I/O buffers

Sys::Virt::NODE_MEMORY_STATS_CACHED
: The amount of memory consumed by disk cache

Sys::Virt::NODE_MEMORY_STATS_FREE
: The amount of free memory

Sys::Virt::NODE_MEMORY_STATS_TOTAL
: The total amount of memory

IP address constants

The following constants are used to interpret IP address types

Sys::Virt::IP_ADDR_TYPE_IPV4
: An IPv4 address type

Sys::Virt::IP_ADDR_TYPE_IPV6
: An IPv6 address type

BUGS

Hopefully none, but the XS code needs to be audited to ensure it is not leaking memory.

AUTHORS

Daniel P. Berrange <berrange@redhat.com>

COPYRIGHT

LICENSE

SEE ALSO

Sys::Virt::Domain, Sys::Virt::Network, Sys::Virt::StoragePool, Sys::Virt::StorageVol, Sys::Virt::Error, `http://libvirt.org`