LBNL Node Health Check Documentation

Release 1.4.3-dev

Michael Jennings

November 18, 2015

Contents

Contents:

CHAPTER 1

LBNL Node Health Check (NHC)

[![Join the chat at https://gitter.im/mej/nhc](https://badges.gitter.im/Join%20Chat.svg)](https://gitter.im/mej/nhc?utm_source=badge&utm_medium=badge&utm_campaign=badge&utm_content=badge)

TORQUE, SLURM, and other schedulers/resource managers provide for a periodic "node health check" to be performed on each compute node to verify that the node is working properly. Nodes which are determined to be "unhealthy" can be marked as down or offline so as to prevent jobs from being scheduled or run on them. This helps increase the reliability and throughput of a cluster by reducing preventable job failures due to misconfiguration, hardware failure, etc.

Though many sites have created their own scripts to serve this function, the vast majority are one-off efforts with little attention

- Reliable - To prevent single-threaded script execution from causing hangs, execution of subcommands is kept to an absolute minimum, and a watchdog timer is used to terminate the check if it runs for too long.
- Fast - Implemented almost entirely in native *bash* (2.x or greater). Reducing pipes and subcommands also cuts down on execution delays and related overhead.
- Flexible - Anything which can be described in a shell function can be a check. Modules can also populate cache data and reuse it for multiple checks.
- Extensible - Its modular functional interface makes writing new checks easy. Just drop modules into the scripts directory, then add your checks to the config file!
- Reusable - Written to be ultra-portable and can be used directly from a resource manager or scheduler, run via cron, or even spawned centrally (e.g., via *pdsh*). The configuration file syntax allows for all compute nodes to share a single configuration.

In a typical scenario, the NHC driver script is run periodically on each compute node by the resource manager client daemon (e.g., *pbs_mom*). It loads its configuration file to determine which checks are to be run on the current node (based on its hostname). Each matching check is run, and if a failure is encountered, NHC will exit with an error message describing the problem. It can also be configured to mark nodes offline so that the scheduler will not assign jobs to bad nodes, reducing the risk of system-induced job failures. NHC can also log errors to the syslog (which is often forwarded to the master node). Some resource managers are even able to use NHC as a pre-job validation tool, keeping scheduled jobs from running on a newly-failed node, and/or a post-job cleanup/checkup utility to remove nodes from the scheduler which may have been adversely affected by the just-completed job.

1.1 Getting Started

The following instructions will walk you through downloading and installing LBNL NHC, configuring it for your system, testing the configuration, and implementing it for use with the TORQUE resource manager.

1.1.1 Installation

Pre-built RPM packages for Red Hat Enterprise Linux versions 4, 5, 6, and 7 are made available with each release along with the source tarballs. The latest release, as well as prior releases, can be found [on GitHub](https://github.com/mej/nhc/releases/). Simply download the appropriate RPM for your compute nodes (e.g., [lbnl-nhc-1.4.2-1.el7.noarch.rpm](https://github.com/mej/nhc/releases/download/1.4.2/lbnl-nhc-1.4.2-1.el7.noarch.rpm)) and install it into your compute node VNFS.

The NHC Yum repository is currently unavailable, but we hope to provide one in the very near future!

The [source tarball for the latest release](https://github.com/mej/nhc/archive/1.4.2.tar.gz) is also available via the [NHC Project on GitHub](https://github.com/mej/nhc/). If you prefer to install from source, or aren't using one of the distributions shown above, use the commands shown here:

```
./configure --prefix=/usr --sysconfdir=/etc --libexecdir=/usr/libexec
make test
make install
```

Note: The *make test* step is optional but recommended. This will run NHC's built-in unit test suite to make sure everything is functioning properly!

Note: You can also fork and/or clone the whole NHC project on GitHub; this is recommended if you plan to contribute to NHC development as this makes it very easy to submit your changes upstream using GitHub Pull Requests! Visit the [NHC Project Page](https://github.com/mej/nhc/) to Watch, Star, or Fork the project!

Whether you use RPMs or install from source, the script will be installed as */usr/sbin/nhc*, the configuration file and check scripts in */etc/nhc*, and the helper scripts in */usr/libexec/nhc*. Once you've completed one of the 3 installation methods above on your compute nodes' root filesystem image, you can proceed with the configuration.

1.1.2 Sample Configuration

The default configuration supplied with LBNL NHC is intended to be more of an overview of available checks than a working configuration. It's essentially impossible to create a default configuration that will work out-of-the-box for any host and still do something useful. But there are some basic checks which are likely to apply, with some modifications of boundary values, to most systems. Here's an example *nhc.conf* which shouldn't require too many tweaks to be a solid starting point:

```
# Check that / is mounted read-write.
* || check_fs_mount_rw /


# Check that sshd is running and is owned by root.
* || check_ps_service -u root -S sshd


# Check that there are 2 physical CPUs, 8 actual cores, and 8 virtual cores (i.e., threads)
* || check_hw_cpuinfo 2 8 8


# Check that we have between 1kB and 1TB of physical RAM
* || check_hw_physmem 1k 1TB


# Check that we have between 1B and 1TB of swap
* || check_hw_swap 1b 1TB
```

```
# Check that we have at least some swap free
* || check_hw_swap_free 1


# Check that eth0 is available
* || check_hw_eth eth0
```

Obviously you'll need to adjust the CPU and memory numbers, but this should get you started.

Config File Auto-Generation

Instead of starting with a basic sample configuration and building on it, as of version 1.4.1, the *nhc-genconf* utility is supplied with NHC which uses the same shell code as NHC itself to query various attributes of your system (CPU socket/core/thread counts, RAM size, swap size, etc.) and automatically generate an initial configuration file based on its scan. Simply invoke *nhc-genconf* on each system where NHC will be running. By default, this will create the file */etc/nhc/nhc.conf.auto* which can then be renamed (or used directly via NHC's *-c* option), tweaked, and deployed on your system!

Normally the config file which *nhc-genconf* creates will use the hostname of the node on which it was run at the beginning of each line. This is to allow multiple files to be merged and sorted into a single config that will work across your system. However, you may wish to provide a custom match expression to prefix each line; this may be done via the *-H* option (e.g., *-H host1* or *-H '*'*).

The scan also includes BIOS information obtained via the *dmidecode* command. The default behavior only includes lines from the output which match the regular expression */([Ss]peed|[Vv]ersion)/*, but this behavior may be altered by supplying an alternative match string via the *-b* option (e.g., *-b '*release*'*).

It can be incredibly tedious, especially for large, well-established heterogeneous or multi-generational clusters to gather up all the different types of hardware that exist in your system and write the appropriate NHC config file rules, match expressions, etc. The following commands might come in handy for aggregating the results of *nhc-genconf* across a large group of nodes:

```
wwsh ssh 'n*' "/usr/sbin/nhc-genconf -H '*' -c -" | dshbak -c
# OR
pdsh -a "/usr/sbin/nhc-genconf -H '*' -c -" | dshbak -c
```

1.1.3 Testing

As of version 1.2 (and higher), NHC comes with a built-in set of fairly extensive unit tests. Each of the check functions is tested for proper functionality; even the driver script (*/usr/sbin/nhc* itself) is tested! To run the unit tests, use the *make test* command at the top of the source tree. You should see something like this:

```
# make test
make -C test test
make[1]: Entering directory `/home/mej/svn/lbnl/nhc/test'
Running unit tests for NHC:
nhcmain_init_env...ok 6/6
nhcmain_finalize_env...ok 14/14
nhcmain_check_conffile...ok 1/1
nhcmain_load_scripts...ok 6/6
nhcmain_set_watchdog...ok 1/1
nhcmain_run_checks...ok 2/2
common.nhc...ok 18/18
ww_fs.nhc...ok 61/61
```

```
ww_hw.nhc...ok 65/65
ww_job.nhc...ok 2/2
ww_nv.nhc...ok 4/4
ww_ps.nhc...ok 32/32
All 212 tests passed.
make[1]: Leaving directory `/home/mej/svn/lbnl/nhc/test'
```

If everything works properly, all the unit tests should pass. Any failures represent a problem that should be reported to the [NHC Users' Mailing List](mailto:nhc@lbl.gov)!

Before adding the node health check to your resource manager (RM) configuration, it's usually prudent to do a test run to make sure it's installed/configured/running properly first. To do this, simply run */usr/sbin/nhc* with no parameters. Successful execution will result in no output and an exit code of 0. If this is what you get, you're done testing! Skip to the next section.

If you receive an error, it will look similar to the following:

```
ERROR Health check failed:  Actual CPU core count (2) does not match expected (8).
```

Depending on which check failed, the message will vary. Hopefully it will be clear what the discrepancy is based on the content of the message. Adjust your configuration file to match your system and try again. If you need help, feel free to post to the [NHC Users' Mailing List](mailto:nhc@lbl.gov).

Additional information may be found in */var/log/nhc.log*, the runtime logfile for NHC. A successful run based on the configuration above will look something like this:

```
Node Health Check starting.
Running check:  "check_fs_mount_rw /"
Running check:  "check_ps_daemon sshd root"
Running check:  "check_hw_cpuinfo 2 8 8"
Running check:  "check_hw_physmem 1024 1073741824"
Running check:  "check_hw_swap 1 1073741824"
Running check:  "check_hw_swap_free 1"
Running check:  "check_hw_eth eth0"
Node Health Check completed successfully (1s).
```

A failure will look like this:

```
Node Health Check starting.
Running check:  "check_fs_mount_rw /"
Running check:  "check_ps_daemon sshd root"
Running check:  "check_hw_cpuinfo 2 8 8"
Health check failed:  Actual CPU core count (2) does not match expected (8).
```

We can see from the excerpt here that the *check_hw_cpuinfo* check failed and that the machine we ran on appears to be a dual-socket single-core system (2 cores total). Since our configuration expected a dual-socket quad-core system (8 cores total), this was flagged as a failure. Since we're testing our configuration, this is most likely a mismatch between what we told NHC to expect and what the system actually has, so we need to fix the configuration file. Once we have a working configuration and have gone into production, a failure like this would likely represent a hardware issue.

Once the configuration has been modified, try running */usr/sbin/nhc* again. Continue fixing the discrepancies and re-running the script until it succeeds; then, proceed with the next section.

1.1.4 Implementation

Instructions for putting NHC into production depend entirely on your use case. We can't possibly hope to delineate them all, but we'll cover some of the most common.

TORQUE Integration

NHC can be executed by the *pbs_mom* process at job start, job end, and/or regular intervals (irrespective of whether or not the node is running job(s)). More detailed information on how to configure the *pbs_mom* health check can be found in the [TORQUE Documentation](http://docs.adaptivecomputing.com/torque/help.htm#topics/11-troubleshooting/computeNodeHealthCheck.htm). The configuration used here at LBNL is as follows:

```
$node_check_script /usr/sbin/nhc
$node_check_interval 5,jobstart,jobend
$down_on_error 1
```

This causes *pbs_mom* to launch */usr/sbin/nhc* every 5 "MOM intervals" (45 seconds by default), when starting a job, and when a job completes (or is terminated). Failures will cause the node to be marked as "down."

> **NOTE:** Some concern has been expressed over the possibility for "OS jitter" caused by NHC. NHC was designed to keep jitter to an absolute minimum, and the implementation goes to extreme lengths to reduce and eliminate as many potential causes of jitter as possible. No significant jitter has been experienced so far (and similar checks at similar intervals are used on _extremely_ jitter-sensitive systems); however, increase the interval to *80* instead of *5* for once-hourly checks if you suspect NHC-generated jitter to be an issue for your system. Alternatively, some sites have configured NHC to detect running jobs and simply exit (or run fewer checks); that works too!

In addition, NHC will by default mark the node "offline" (i.e., *pbsnodes -o*) and add a note (viewable with *pbsnodes -ln*) specifying the failure. Once the failure has been corrected and NHC completes successfully, it will remove the note it set and clear the "offline" status from the node. In order for this to work, however, each node must have "operator" access to the TORQUE daemon. Unfortunately, the support for wildcards in *pbs_server* attributes is limited to replacing the host, subdomain, and/or domain portions with asterisks, so for most setups this will likely require omitting the entire hostname section. The following has been tested and is known to work:

```
qmgr -c "set server operators += root@*"
```

This functionality is not strictly required, but it makes determining the reason nodes are marked down significantly easier!

Another possible caveat to this functionality is that it only works if the canonical hostname (as returned by the *hostname* command or the file */proc/sys/kernel/hostname*) of each node matches its identity within TORQUE. If your site uses FQDNs on compute nodes but has them listed in TORQUE using the short versions, you will need to add something like this to the top of your NHC configuration file:

```
* || HOSTNAME="$HOSTNAME_S"
```

This will cause the offline/online helpers to use the shorter hostname when invoking *pbsnodes*. This will NOT, however, change how the hostnames are matched in the NHC configuration, so you'll still need to use FQDN matching there.

It's also important to note here that NHC will only set a note on nodes that don't already have one (and aren't yet offline) or have one set by NHC itself; also, it will only online nodes and clear notes if it sees a note that was set by NHC. It looks for the string "NHC:" in the note to distinguish between notes set by NHC and notes set by operators. If you use this feature, and you need to mark nodes offline manually (e.g., for testing), setting a note when doing so is strongly encouraged. (You can do this via the *-N* option, like this: *pbsnodes -o -N 'Testing stuff' n0000 n0001 n0002*) There was a bug in versions prior to 1.2.1 which would cause it to treat nodes with no notes the same way it treats nodes with NHC-assigned notes. This _should_ be fixed in 1.2.1 and higher, but you never know....

SLURM Integration

Add the following to */etc/slurm.conf* (or */etc/slurm/slurm.conf*, depending on version) on your master node **AND** your compute nodes (because, even though the *HealthCheckProgram* only runs on the nodes, your *slurm.conf* file must be the same across your entire system):

```
HealthCheckProgram=/usr/sbin/nhc
HealthCheckInterval=300
```

This will execute NHC every 5 minutes.

For optimal support of SLURM, NHC version 1.3 or higher is recommended. Prior versions will require manual intervention.

Periodic Execution

The original method for doing this was to employ a simple *crontab* entry, like this one:

```
MAILTO=operators@your.com
*/5 * * * * /usr/sbin/nhc
```

Annoyingly, this would result in an e-mail being sent every 5 minutes if one of the health checks fails. It was for this very reason that the contributed *nhc.cron* script was originally written. However, even though it avoids the former technique's flood of e-mail when a problem arose, it still had no clean way of dealing with multiple contexts and could not be set up to do periodic reminders of issues. Additionally, it would fail to notify if a new problem was detected before or at the same time the old problem was resolved.

Version 1.4.1 introduces a vastly superior option: *nhc-wrapper*. This tool will execute ‘nhc‘^{[1](#footnotes)} and record the results. It then compares the results to the output of the previous run, if present, and will ignore results that are identical to those previously obtained. Old results can be set to expire after a given length of time (and thus re-reported). Results may be echoed to stdout or sent via e-mail. Once an unrecognized command line option or non-option argument is encountered, it and the rest of the command line arguments are passed to the wrapped program intact.

This tool will typically be run via *cron(8)*. It can be used to wrap distinct contexts of NHC in a manner identical to NHC itself (i.e., specified via executable name or command line arg); also, unlike the old *nhc.cron* script, this one does a comparison of the results rather than only distinguishing between the presence/absence of output, and those results can have a finite lifespan.

nhc-wrapper also offers another option for periodic execution: looping (*-L*). When launched from a terminal or *inittab*/*init.d* entry in looping mode, *nhc-wrapper* will execute a loop which runs the wrapped program (e.g., *nhc*) at a time interval you supply. It attempts to be smart about interpreting your intent as well, calculating sleep times after subprogram execution (i.e., the interval is from start time to start time, not end time to start time) and using nice, round execution times when applicable (i.e., based on 00:00 local time instead of whatever random time the wrapper loop happened to be entered). For example, if you ask it to run every 5 minutes, it'll run at :00, :05, :10, :15, etc. If you ask for every 4 hours, it'll run at 00:00, 04:00, 08:00, 12:00, 16:00, and 20:00 exactly–regardless of what time it was when you originally launched *nhc-wrapper*!

This allows the user to run *nhc-wrapper* in a terminal to keep tabs on it while still running checks at predictable times (just like *crond* would). It also has some flags to provide timestamps (*-L t*) and/or ASCII horizontal rulers (*-L r*) between executions; clearing the screen (*-L c*) before each execution (*watch*-style) is also available.

Examples:

To run *nhc* and notify *root* when errors appear, are cleared, or every 12 hours while they persist:

```
/usr/sbin/nhc-wrapper -M root -X 12h
```

Same as above, but run the "nhc-cron" context instead (*nhc -n nhc-cron*):

```
/usr/sbin/nhc-wrapper -M root -X 12h -n nhc-cron
# OR
/usr/sbin/nhc-wrapper -M root -X 12h -A '-n nhc-cron'
```

Same as above, but run *nhc-cron* (symlink to *nhc*) instead:

```
/usr/sbin/nhc-wrapper -M root -X 12h -P nhc-cron
# OR
ln -s nhc-wrapper /usr/sbin/nhc-cron-wrapper
/usr/sbin/nhc-cron-wrapper -M root -X 12h
```

Expire results after 1 week, 1 day, 1 hour, 1 minute, and 1 second:

```
/usr/sbin/nhc-wrapper -M root -X 1w1d1h1m1s
```

Run verbosely, looping every minute with ruler and timestamp:

```
/usr/sbin/nhc-wrapper -L tr1m -V
```

Or for something quieter and more *cron*-like:

```
/usr/sbin/nhc-wrapper -L 1h -M root -X 12h
```

1.2 Configuration

Now that you have a basic working configuration, we'll go more in-depth into how NHC is configured, including command-line invocation, configuration file syntax, modes of operation, how individual checks are matched against a node's hostname, and what checks are already available in the NHC distribution for your immediate use.

Configuration of NHC is generally done in one of 3 ways: passing option flags and/or configuration (i.e., environment) variables on the command line, setting variables and specifying checks in the configuration file (*/etc/nhc/nhc.conf* by default), and/or setting variables in the sysconfig initialization file (*/etc/sysconfig/nhc* by default). The latter works essentially the same as any other sysconfig file (it is directly sourced into NHC's *bash* session using the . operator), so this document does not go into great detail about using it. The following sections discuss the other two mechanisms.

1.2.1 Command-Line Invocation

From version 1.3 onward, NHC supports a subset of command-line options and arguments in addition to the configuration and sysconfig files. A few specific settings have CLI options associated with them as shown in the table below; additionally, any configuration variable which is valid in the configuration or sysconfig file may also be passed on the command line instead.

Options

Command-Line Option | Equivalent Configuration Variable | Purpose |
——————— | ——————————— | ——— |
-D confdir | *CONFDIR=confdir* | Use config directory *confdir* (default: */etc/name*) |
-a | *NHC_CHECK_ALL=1* | Run ALL checks; don't exit on first failure (useful for *cron*-based monitoring) |
-c conffile | *CONFFILE=conffile* | Load config from *conffile* (default: *confdir/name.conf*) |
-d | *DEBUG=1* | Activate debugging output |
-f | *NHC_CHECK_FORKED=1* | Run each check in a separate background process (*EXPERIMENTAL*) |
-h | N/A | Show command line help |
-l logspec | *LOGFILE=logspec* | File name/path or BASH-syntax directive for logging output (- for *STDOUT*) |
-n name | *NAME=name* | Set program name to *name* (default: *nhc*); see -D & -c |
-q | *SILENT=1* | Run quietly |
-t timeout | *TIMEOUT=‘‘timeout* | Use timeout of *timeout* seconds (default: 30) |
-v | *VERBOSE=1* | Run verbosely (i.e., show check progress) |

Note: Due to the use of the *getopts bash* built-in, and the limitations thereof, POSIX-style bundling of options (e.g., *-da*) is NOT supported, and all command-line options MUST PRECEDE any additional variable/value-type arguments!

Variable/Value Arguments

Instead of, or possibly in addition to, the use of command-line options, NHC accepts configuration via variables specified on the command line. Simply pass any number of VARIABLE=value‘_ arguments on the command line, and each variable will be set to its respective value immediately upon NHC startup. This happens before the sysconfig file is loaded, so it can be used to alter such values as ‘$SYSCONFIGDIR (*/etc/sysconfig* by default) which would normally be unmodifiable.

It’s important to note that while command-line configuration directives will override NHC’s built-in defaults for various variables, variables set in the configuration file (see below) will NOT be overridden. The config file takes precedence over the command line, in contrast to most other CLI tools out there (and possibly contrary to user expectation) due to the way *bash* deals with variables and initialization. If you want the command line to take precedence, you’ll need to test the value of the variable in the config file and only alter it if the current value matches NHC’s built-in default.

Example Invocations

Most sites just run *nhc* by itself with no options when launching from a resource manager daemon. However, when running from cron or manually at the command line, numerous other possible scenarios exist for invoking NHC in various ways. Here are some real-world examples.

```
nhc -d
nhc DEBUG=1
```

To run for testing purposes in debug mode with no timeout and with node online/offline disabled:

```
nhc -d -t 0 MARK_OFFLINE=0
```

To force use of SLURM as the resource manager and use a sysconfig path in */opt*:

```
nhc NHC_RM=slurm SYSCONFIGDIR=/opt/etc/sysconfig
```

To run NHC out-of-band (e.g., from cron) with the name *nhc-oob* (which will load its config from */etc/sysconfig/nhc-oob* and */etc/nhc/nhc-oob.conf*):

```
nhc -n nhc-oob
```

Note: As an alternative, you may symlink */usr/sbin/nhc-oob* to *nhc* and run *nhc-oob* instead. This will accomplish the same thing.

1.2.2 Configuration File Syntax

The configuration file is fairly straight-forward. Stored by default in */etc/nhc/nhc.conf*, the file is plain text and recognizes the traditional # introducer for comments. Any line that starts with a # (with or without leading whitespace) is ignored. Blank lines are also ignored.

Examples:

```
# This is a comment.
       # This is also a comment.
# This line and the next one will both be ignored.
```

Configuration lines contain a **target** specifier, the separator string ||, and the **check** command. The target specifies which hosts should execute the check; only nodes whose hostname matches the given target will execute the check on that line. All other nodes will ignore it and proceed to the next check.

A check is simply a shell command. All NHC checks are bash functions defined in the various included files in */etc/nhc/scripts/*.nhc*, but in actuality any valid shell command that properly returns success or failure will work. This documentation and all examples will only reference bash function checks. Each check can take zero or more arguments and is executed exactly as seen in the configuration.

As of version 1.2, configuration variables may also be set in the config file with the same syntax. This makes it easy to alter specific settings, commands, etc. globally or for individual hosts/hostgroups!

Example:

```
    * || SOMEVAR="value"
    * || check_something
*.foo || another_check 1 2 3
```

1.2.3 Match Strings

As noted in the last section, the first item on each line of the NHC configuration file specifies the **target** for the check which will follow. When NHC runs on a particular host, it reads and parses each line of the configuration file, comparing the hostname of the host (taken from the *$HOSTNAME* variable) with the specified target expression; if the target matches, the check will be saved for later execution. Lines whose targets don't match the current host are ignored completely. The target is expressed in the form of a **match string** – an NHC expression that allows for exact string matches or a variety of dynamic comparison methods. Match strings are a very important concept and are used throughout NHC, not just for check targets, but as parameters to individual checks as well, so it's important that users fully understand how they work.

There are multiple forms of **match string** supported by NHC. The default style is a **glob**, also known as a **wildcard**. bash will determine if the hostname of the node (specifically, the contents of */proc/sys/kernel/hostname*) matches the supplied glob expression (e.g., *n*.viz*) and execute only those checks which have matching target expressions. If the hostname does not match the glob, the corresponding check is ignored.

The second method for specifying host matches is via **regular expression**. Regex targets must be surrounded by slashes to identify them as regular expressions. The internal regex matching engine of bash is used to compare the hostname to the given regular expression. For example, given a target of */^n00[0-5][0-9].cc2$/*, the corresponding check would execute on *n0017.cc2* but not on *n0017.cc1* or *n0083.cc2*.

The third form of match string (supported in NHC versions 1.2.2 and later) is **node range expressions** similar to those used by *pdsh*, Warewulf, and other open source HPC tools. (_Please note that not all expressions supported by other tools will work in NHC due to limitations in *bash*._) The match expression is placed in curly braces and specifies one or more comma-separated node name ranges, and the corresponding check will only execute on nodes which fall into at least one of the specified ranges. Note that only one range expression is supported per range, and commas within ranges are not supported. So, for example, the target *{n00[00-99].phys,n000[0-4].bio}* would cause its check to execute on *n0030.phys*, *n0099.phys*, and *n0001.bio*, but not on *n0100.phys* nor *n0005.bio*. Expressions such as *{n[0-3]0[00-49].r[00-29]}* and *{n00[00-29,54,87].sci}* are not supported (though the latter may be written instead as *{n00[00-29].sci,n0054.sci,n0087.sci}*).

Match strings of any form (glob/wildcard, regular expression, node range, or external) can be negated. This simply means that a match string which would otherwise have matched will instead fail to match, and vice versa (i.e., the boolean result of the match is inverted). To negate any match string, simply prefix it (before the initial type character,

if any) with an exclamation mark (*!*). For example, to run a check on all but the I/O nodes, you could use the expression: *!io**

Examples:

```
                *  || valid_check1
              !ln* || valid_check2
       /n000[0-9]/ || valid_check3
    !/\.(gpu|htc)/ || valid_check4
      {n00[20-39]} || valid_check5
!{n03,n05,n0[7-9]} || valid_check6
   {n00[10-21,23]} || this_target_is_invalid
```

Throughout the rest of the documentation, we will refer to this concept as a **match string** (or abbreviated **mstr**). Anywhere a match string is expected, either a glob, a regular expression surrounded by slashes, or node range expression in braces, possibly with a leading *!* to negate it, may be specified.

1.2.4 Supported Variables

As mentioned above, version 1.2 and higher support setting/changing shell variables within the configuration file. Many aspects of NHC's behavior can be modified through the use of shell variables, including a number of the commands in the various checks and helper scripts NHC employs.

There are, however, some variables which can only be specified in */etc/sysconfig/nhc*, the global initial settings file for NHC. This is typically for obvious reasons (e.g., you can't change the path to the config file from within the config file!).

The table below provides a list of the configuration variables which may be used to modify NHC's behavior; those which won't work in a config file (only sysconfig or command line) are marked with an asterisk ("*"):

Variable Name | Default Value | Purpose |
——————— | ——————— | ——————— |
* CONFDIR | */etc/nhc* | Directory for NHC configuration data |
* CONFFILE | *$CONFDIR/$NAME.conf* | Path to NHC config file |
DEBUG | *0* | Set to *1* to activate debugging output |
* DETACHED_MODE | *0* | Set to *1* to activate [Detached Mode](#detached-mode) |
* DETACHED_MODE_FAIL_NODATA | *0* | Set to *1* to cause [Detached Mode](#detached-mode) to fail if no prior check result exists |
DF_CMD | *df* | Command used by *check_fs_free*, *check_fs_size*, and *check_fs_used* |
DF_FLAGS | *-Tka* | Flags to pass to *$DF_CMD* for space checks. **_NOTE:_ Adding the '-l' flag is _strongly_ recommended if only checking local filesystems.** |
DFI_CMD | *df* | Command used by *check_fs_inodes*, *check_fs_ifree*, and *check_fs_iused* |
DFI_FLAGS | *-Tia* | Flags to pass to *$DFI_CMD*. **_NOTE:_ Adding the '-l' flag is _strongly_ recommended if only checking local filesystems.** |
* FORCE_SETSID | *1* | Re-execute NHC as a session leader if it isn't already one at startup |
* HELPERDIR | */usr/libexec/nhc* | Directory for NHC helper scripts |
* HOSTNAME | Set from */proc/sys/kernel/hostname* | Canonical name of current node |
* HOSTNAME_S | *$HOSTNAME* truncated at first . | Short name (no domain or subdomain) of current node |
IGNORE_EMPTY_NOTE | *0* | Set to *1* to treat empty notes like NHC-assigned notes (<1.2.1 behavior) |
* INCDIR | *$CONFDIR/scripts* | Directory for NHC check scripts |
JOBFILE_PATH | TORQUE/PBS: *$PBS_SERVER_HOME/mom_priv/jobs*
 SLURM: *$SLURM_SERVER_HOME* | Directory on compute nodes where job records are kept |

* LOGFILE | >>*/var/log/nhc.log* | File name/path or BASH-syntax directive for logging output (- for *STDOUT*) |
LSF_BADMIN | *badmin* | Command to use for LSF's *badmin* (may include path) |
LSF_BHOSTS | *bhosts* | Command to use for LSF's *bhosts* (may include path) |
LSF_OFFLINE_ARGS | *hclose -C* | Arguments to LSF's *badmin* to offline node |
LSF_ONLINE_ARGS | *hopen* | Arguments to LSF's *badmin* to online node |
MARK_OFFLINE | *1* | Set to *0* to disable marking nodes offline on check failure |
MAX_SYS_UID | *99* | UIDs <= this number are exempt from rogue process checks |
MCELOG | *mcelog* | Command to use to check for MCE log errors |
MCELOG_ARGS | *–client* | Parameters passed to *$MCELOG* command |
MCELOG_MAX_CORRECTED_RATE | *9* | Maximum number of **corrected** MCEs allowed before *check_hw_mcelog()* returns failure |
MCELOG_MAX_UNCORRECTED_RATE | *0* | Maximum number of **uncorrected** MCEs allowed before *check_hw_mcelog()* returns failure |
MDIAG_CMD | *mdiag* | Command to use to invoke Moab's *mdiag* command (may include path) |
* NAME | *nhc* | Used to populate default paths/filenames for configuration |
NHC_AUTH_USERS | *root nobody* | Users authorized to have arbitrary processes running on compute nodes |
NHC_CHECK_ALL | *0* | Forces all checks to be non-fatal. Displays each failure message, reports total number of failed checks, and returns that number. |
NHC_CHECK_FORKED | *0* | Forces each check to be executed in a separate forked subprocess. NHC attempts to detect directives which set environment variables to avoid forking those. Enhances resiliency if checks hang. |
NHC_RM | Auto-detected | Resource manager with which to interact (*pbs*, *slurm*, *sge*, or *lsf*) |
NVIDIA_HEALTHMON | *nvidia-healthmon* | Command used by *check_nv_healthmon* to check nVidia GPU status |
NVIDIA_HEALTHMON_ARGS | *-e -v* | Arguments to *$NVIDIA_HEALTHMON* command |
OFFLINE_NODE | *$HELPERDIR/node-mark-offline* | Helper script used to mark nodes offline |
ONLINE_NODE | *$HELPERDIR/node-mark-online* | Helper script used to mark nodes online |
PASSWD_DATA_SRC | */etc/passwd* | Colon-delimited file in standard passwd format from which to load user account data |
PATH | */sbin:/usr/sbin:/bin:/usr/bin* | If a path is not specified for a particular command, this variable defines the directory search order.
PBSNODES | *pbsnodes* | Command used by above helper scripts to mark nodes online/offline |
PBSNODES_LIST_ARGS | *-n -l all* | Arguments to *$PBSNODES* to list nodes and their status notes |
PBSNODES_OFFLINE_ARGS | *-o -N* | Arguments to *$PBSNODES* to mark node offline with note |
PBSNODES_ONLINE_ARGS | *-c -N* | Arguments to *$PBSNODES* to mark node online with note |
PBS_SERVER_HOME | */var/spool/torque* | Directory for TORQUE files |
RESULTFILE | */var/run/nhc/$NAME.status* | Used in [Detached Mode](#detached-mode) to store result of checks for subsequent handling
RM_DAEMON_MATCH | TORQUE/PBS: */bpbs_momb/*
 SLURM: */bslurmdb/*
 SGE/UGE: */bsge_execdb/* | [Match string](#match-strings) used by *check_ps_userproc_lineage* to make sure all user processes were spawned by the RM daemon |
SILENT | *0* | Set to *1* to disable logging via *$LOGFILE* |
SLURM_SCONTROL | *scontrol* | Command to use for SLURM's *scontrol* (may include path) |
SLURM_SC_OFFLINE_ARGS | *update State=DRAIN* | Arguments to pass to SLURM's *scontrol* to offline a node |
SLURM_SC_ONLINE_ARGS | *update State=IDLE* | Arguments to pass to SLURM's *scontrol* to online a node |
SLURM_SERVER_HOME | */var/spool/slurmd* | Location of SLURM data files (see also: *$JOBFILE_PATH*) |
SLURM_SINFO | *sinfo* | Command to use for SLURM's *sinfo* (may include path) |
STAT_CMD | */usr/bin/stat* | Command to use to *stat()* files |
STAT_FMT_ARGS | *-c* | Parameter to introduce format string to *stat* command |
* TIMEOUT | *30* | Watchdog timer (in seconds) |

VERBOSE | *0* | Set to *1* to display each check line before it's executed |

Example usage:

```
     * || export PATH="$PATH:/opt/torque/bin:/opt/torque/sbin"
n*.rh6 || MAX_SYS_UID=499
n*.deb || MAX_SYS_UID=999
*.test || DEBUG=1
     * || export MARK_OFFLINE=0
     * || NVIDIA_HEALTHMON="/global/software/rhel-6.x86_64/modules/nvidia/tdk/3.304.3/nvidia-healthmo
```

1.2.5 Detached Mode

Version 1.2 and higher support a feature called "detached mode." When this feature is activated on the command line or in */etc/sysconfig/nhc* (by setting *DETACHED_MODE=1*), the *nhc* process will immediately fork itself. The foreground (parent) process will immediately return success. The child process will run all the checks and record the results in *$RESULTFILE* (default: */var/run/nhc.status*). The next time *nhc* is executed, just before forking off the child process (which will again run the checks in the background), it will load the results from *$RESULTFILE* from the last execution. Once the child process has been spawned, it will then return the previous results to its caller.

The advantage of detached mode is that any hangs or long-running commands which occur in the checks will not cause the resource manager daemon (e.g., *pbs_mom*) to block. Sites that use home-grown health check scripts often use a similar technique for this very reason – it's non-blocking.

However, a word of caution: if a detached-mode *nhc* encounters a failure, it won't get acted upon until the **next execution**. So let's say you have NHC configured to only on job start and job end. Let's further suppose that the */tmp* filesystem encounters an error and gets remounted read-only at some point after the completion of the last job and that you have *check_fs_mount_rw /tmp* in your *nhc.conf*. In normal mode, when a new job tries to start, *nhc* will detect the read-only mount on job start and will take the node out of service before the job is allowed to begin executing on the node. In detached mode, however, since *nhc* has not been run in the meantime, and the previous run was successful, *nhc* will return success and allow the job to start _before_ the error condition is noticed!

For this reason, when using detached mode, periodic checks are HIGHLY recommended. This will not completely prevent the above scenario, but it will drastically reduce the odds of it occurring. Users of detached mode, as with any similar method of delayed reporting, must be aware of and accept this caveat in exchange for the benefits of the more-fully-non-blocking behavior.

1.2.6 Built-in Checks

In the documentation below, parameters surrounded by square brackets ([like this]) are **optional**. All others are **required**.

The LBNL Node Health Check distribution supplies the following checks:

check_cmd_output

check_cmd_output [-t timeout] [-r retval] [-m match [...]] { -e 'command [arg1 [...]]' | command [arg1 [...]] }

check_cmd_output executes a *command* and compares each line of its output against any *mstr's ([match strings](#match-strings)) passed in. If any positive match **is not** found in the command output, or if any negative match **is** found, the check fails. The check also fails if the exit status of 'command* does not match *retval* (if supplied) or if the *command* fails to complete within *timeout* seconds (default 5). Options to this check are as follows:

Check Option | Purpose |
———————- | ———— |

-e command | Execute *command* and gather its output. The *command* is split on word boundaries, much like */bin/sh -c '...'* does. |

-m mstr | If negated, no line of the output may match the specified *mstr* expression. Otherwise, at least one line must match. This option may be used multiple times as needed. |

-r retval | Exit status (a.k.a. return code or return value) of *command* must equal *retval* or the check will fail. |

-t secs | Command will timeout if not completed within *secs* seconds (default is 5). |

Note: If the *command* is passed using *-e*, the *command* string is split on word boundaries to create the *argv[]* array for the command. If passed on the end of the check line, DO NOT quote the command. Each parameter must be distinct. Only use quotes to group multiple words into a single argument. For example, passing *command* as *"service bind restart"* will work if used with *-e* but will fail if passed at the end of the check line (use without quotes instead)!

**Example** (Verify that the *rpcbind* service is alive): *check_cmd_output -t 1 -r 0 -m '/is running/' /sbin/service rpcbind status*

check_cmd_status

check_cmd_status [-t timeout] -r retval command [arg1 [...]]

check_cmd_status executes a *command* and redirects its output to */dev/null*. The check fails if the exit status of *command* exit status does not match *retval* or if the *command* fails to complete within *timeout* seconds (default 5). Options to this check are as follows:

Check Option | Purpose |
———————- | ———— |

-r retval | Exit status (a.k.a. return code or return value) of *command* must equal *retval* or the check will fail. |

-t secs | Command will timeout if not completed within *secs* seconds (default is 5). |

**Example** (Make sure SELinux is disabled): *check_cmd_status -t 1 -r 1 selinuxenabled*

check_dmi_data_match

check_dmi_data_match [-h handle] [-t type] [-n | '!'] string

check_dmi_data_match uses parsed, structured data taken from the output of the *dmidecode* command to allow the administrator to make very specific assertions regarding the contents of the DMI (a.k.a. SMBIOS) data. Matches can be made against any output or against specific types (classifications of data) or even handles (identifiers of data blocks, typically sequential). Output is restructured such that sections which are indented underneath a section header have the text of the section header prepended to the output line along with a colon and intervening space. So, for example, the string "<tab><tab>ISA is supported" which appears underneath the "Characteristics:" header, which in turn is underneath the "BIOS Information" header/type, would be parsed by *check_dmi_data_match* as "BIOS Information: Characteristics: ISA is supported"

See the *dmidecode* man page for more details.

Warning: Although *string* is technically a [match string](#match-strings), and supports negation in its own right, you probably don't want to use negated [match strings](#match-strings) here. Passing the *-n* or *!* parameters to the check means, "check all relevant DMI data and pass the check only if no matching line is found." Using a negated [match string](#match-strings) here would mean, "The check passes as soon as _ANY_ non-matching line is found" – almost certainly not the desired behavior! A subtle but important distinction!

**Example** (check for BIOS version): *check_dmi_data_match "BIOS Information: Version: 1.0.37"*

check_dmi_raw_data_match

check_dmi_raw_data_match match_string [...]

check_dmi_raw_data_match is basically like a *grep* on the raw output of the *dmidecode* command. If you don't need to match specific strings in specific sections but just want to match a particular string anywhere in the raw output, you can use this check instead of *check_dmi_data_match* (above) to avoid the additional overhead of parsing the output into handles, types, and expanded strings.

**Example** (check for firmware version in raw output; could really match any version): *check_dmi_raw_data_match "Version: 1.24.4175.33"*

check_file_contents

check_file_contents file mstr [...]

check_file_contents looks at the specified file and allows one or more (possibly negated) *mstr* [match strings](#match-strings) (glob, regexp, etc.) to be applied to the contents of the file. The check fails unless ALL specified expressions successfully match the file content, but the order in which they appear in the file need not match the order specified on the check line. No post-processing is done on the file, but take care to quote any shell metacharacters in your match expressions properly. Also remember that matching against the contents of large files will slow down NHC and potentially cause a timeout. Reading of the file stops when all match expressions have been successfully found in the file.

The file is only read once per invocation of *check_file_contents*, so if you need to match several expressions in the same file, passing them all to the same check is advisable.

Note: This check handles negated [match strings](#match-strings) internally so that they "do the right thing:" ensure that no matching lines exist in the entire file.

_**Example** (verify setting of $pbsserver in pbs_mom config)_: *check_file_contents /var/spool/torque/mom_priv/config '/^$pbsserver master$/'*

check_file_stat

check_file_stat [-D num] [-G name] [-M mode] [-N secs] [-O secs] [-T num] [-U name] [-d num] [-g gid] [-m mode] [-n secs] [-o secs] [-t num] [-u uid] filename(s)

check_file_stat allows the user to assert specific properties on one or more files, directories, and/or other filesystem objects based on metadata returned by the Linux/Unix *stat* command. Each option specifies a test which is applied to each of the _filename(s)_ in order. The check fails if any of the comparisons does not match. Options to this check are as follows:

Check Option | Purpose |

——————- | ——— |
-D num | Specifies that the device ID for _filename(s)_ should be *num* (decimal or hex) |
-G ‘‘name | Specifies that _filename(s)_ should be owned by group *name* |
-M mode | Specifies that the permissions for _filename(s)_ should include at LEAST the bits set in *mode* |
-N ‘‘secs | Specifies that the *ctime* (i.e., inode change time) of _filename(s)_ should be newer than *secs* seconds ago |
-O ‘‘secs | Specifies that the *ctime* (i.e., inode change time) of _filename(s)_ should be older than *secs* seconds ago |
-T num | Specifies that the minor device number for _filename(s)_ be *num* |
-U ‘‘name | Specifies that filename(s) should be owned by user *name* |
-d num | Specifies that the device ID for _filename(s)_ should be *num* (decimal or hex) |
-g gid | Specifies that _filename(s)_ should be owned by group id *gid* |
-m mode | Specifies that the permissions for _filename(s)_ should include at LEAST the bits set in *mode* |
-n ‘‘secs | Specifies that the *mtime* (i.e., modification time) of _filename(s)_ should be newer than *secs* seconds ago |
-o ‘‘secs | Specifies that the *mtime* (i.e., modification time) of _filename(s)_ should be older than *secs* seconds ago |
-t num | Specifies that the major device number for _filename(s)_ be *num* |
-u uid | Specifies that _filename(s)_ should be owned by uid *uid* |

**Example** (Assert correct uid, gid, owner, group, & major/minor device numbers for */dev/null*): *check_file_stat -u 0 -g 0 -U root -G root -t 1 -T 3 /dev/null*

check_file_test

check_file_test [-a] [-b] [-c] [-d] [-e] [-f] [-g] [-h] [-k] [-p] [-r] [-s] [-t] [-u] [-w] [-x] [-O] [-G] [-L] [-S] [-N] filename(s)

check_file_test allows the user to assert very simple attributes on one or more files, directories, and/or other filesystem objects based on tests which can be performed via the shell's built-in *test* command. Each option specifies a test which is applied to each of the _filename(s)_ in order. NHC internally evaluates the shell expression *test <option filename>* for each option given for each *filename* specified. (In other words, passing 2 options and 3 filenames will evaluate 6 *test* expressions in total.) The check fails if any of the *test* command evaluations returns false. For efficiency, this check should be used in preference to *check_file_stat* whenever possible as it does not require calling out to the *stat* command. Options to this check are as follows:

Check Option | **Purpose** |
——————- | ——— |
-a | Evaluates to true if the *filename* being tested exists (same as *-e*). |
-b | Evaluates to true if the *filename* being tested exists and is block special. |
-c | Evaluates to true if the *filename* being tested exists and is character special. |
-d | Evaluates to true if the *filename* being tested exists and is a directory. |
-e | Evaluates to true if the *filename* being tested exists. |
-f | Evaluates to true if the *filename* being tested exists and is a regular file. |
-g | Evaluates to true if the *filename* being tested exists and is setgid. |
-h | Evaluates to true if the *filename* being tested exists and is a symbolic link. |
-k | Evaluates to true if the *filename* being tested exists and has its sticky bit set. |
-p | Evaluates to true if the *filename* being tested exists and is a named pipe. |
-r | Evaluates to true if the *filename* being tested exists and is readable. |
-s | Evaluates to true if the *filename* being tested exists and is not empty. |
-t | Evaluates to true if the *filename* being tested is a numeric file descriptor which references a valid tty. |

-u | Evaluates to true if the *filename* being tested exists and is setuid. |
-w | Evaluates to true if the *filename* being tested exists and is writable. |
-x | Evaluates to true if the *filename* being tested exists and is executable. |
-O | Evaluates to true if the *filename* being tested exists and is owned by NHC's EUID. |
-G | Evaluates to true if the *filename* being tested exists and is owned by NHC's EGID. |
-L | Evaluates to true if the *filename* being tested exists and is a symbolic link (same as *-h*). |
-S | Evaluates to true if the *filename* being tested exists and is a socket. |
-N | Evaluates to true if the *filename* being tested exists and has been modified since it was last read. |

**Example** (Assert correct ownerships and permissions on */dev/null* similar to above, assuming NHC runs as root): *check_file_test -O -G -c -r -w /dev/null*

check_fs_inodes

check_fs_inodes mountpoint [min] [max]

Ensures that the specified *mountpoint* has at least *_min_* but no more than *_max_* total inodes. Either may be blank.

> **WARNING:** Use of this check requires execution of the */usr/bin/df* command which may HANG in cases of NFS failure! If you use this check, consider also using [Detached Mode](#detached-mode)!

**Example** (make sure */tmp* has at least 1000 inodes): *check_fs_inodes /tmp 1k*

check_fs_ifree

check_fs_ifree mountpoint min

Ensures that the specified *mountpoint* has at least *_min_* free inodes.

> **WARNING:** Use of this check requires execution of the */usr/bin/df* command which may HANG in cases of NFS failure! If you use this check, consider also using [Detached Mode](#detached-mode)!

**Example** (make sure */local* has at least 100 inodes free): *check_fs_ifree /local 100*

check_fs_iused

check_fs_iused mountpoint max

Ensures that the specified *mountpoint* has no more than *_max_* used inodes.

> **WARNING:** Use of this check requires execution of the */usr/bin/df* command which may HANG in cases of NFS failure! If you use this check, consider also using [Detached Mode](#detached-mode)!

**Example** (make sure */tmp* has no more than 1 million used inodes): *check_fs_iused /tmp 1M*

check_fs_mount

check_fs_mount [-0] [-r] [-t fstype] [-s source] [-o options] [-O remount_options] [-e missing_action] [-E found_action] {-f\-F} mountpoint [...]

-OR- (_deprecated_)

check_fs_mount mountpoint [source] [fstype] [options]

check_fs_mount examines the list of mounted filesystems on the local machine to verify that the specified entry is present. *mountpoint* specifies the directory on the node where the filesystem should be mounted. _source_ is a [match string](#match-strings) which is compared against the device, whatever that may be (e.g., _server_:/_path_ for NFS or */dev/sda1* for local). _fstype_ is a [match string](#match-strings) for the filesystem type (e.g., *nfs*, *ext4*, *tmpfs*). _options_ is a [match string](#match-strings) for the mount options. Any number (zero or more) of these 3 items (i.e., sources, types, and/or options) may be specified; additionally, one or more mountpoints may be specified. Use *-f* for normal filesystems and *-F* for auto-mounted filesystems (to trigger them to be mounted prior to performing the check).

Unless the *-0* (non-fatal) option is given, this check will fail if any of the specified filesystems is not found or does not match the type(s)/source(s)/option(s) specified. The *-r* (remount) option will cause NHC to attempt to re-mount missing filesystem(s) by issuing the system command *mount -o <remount_options> <filesystem>‘* in the background as root. This is “best effort,” so success or failure of the mount attempt is not taken into account. If specified, _missing_action_ is executed if a filesystem is not found. Also, if specified, _found_action_ is executed for each filesystem which **is** found and correctly mounted.

**Example** (check for NFS hard-mounted */home* from *bluearc1:/global/home* and mount if missing): *check_fs_mount -r -s bluearc1:/global/home -t nfs -o *hard* -f /home*

check_fs_mount_ro

check_fs_mount_ro [-0] [-r] [-t fstype] [-s source] [-o options] [-O remount_options] [-e missing_action] [-E found_action] -f mountpoint

-OR- (_deprecated_)

check_fs_mount_ro mountpoint [source] [fstype]

Checks that a particular filesystem is mounted read-only. Shortcut for *check_fs_mount -o ‘/(^|,)ro($|,)/’ ‘_...‘_*

check_fs_mount_rw

check_fs_mount_rw [-0] [-r] [-t fstype] [-s source] [-o options] [-O remount_options] [-e missing_action] [-E found_action] -f mountpoint

-OR- (_deprecated_)

check_fs_mount_rw mountpoint [source] [fstype]

Checks that a particular filesystem is mounted read-write. Shortcut for *check_fs_mount -o ‘/(^|,)rw($|,)/’ ‘_...‘_*

check_fs_free

check_fs_free mountpoint minfree

(Version 1.2+) Checks that a particular filesystem has at least _minfree_ space available. The value for _minfree_ may be specified either as a percentage or a numerical value with an optional suffix (*k* or *kB* for kilobytes, the default; *M* or *MB* for megabytes; *G* or *GB* for gigabytes; etc., all case insensitive).

> **WARNING:** Use of this check requires execution of the */usr/bin/df* command which may HANG in cases of NFS failure! If you use this check, consider also using [Detached Mode](#detached-mode)!

**Example**: *check_fs_free /tmp 128MB*

check_fs_size

check_fs_size mountpoint [minsize] [maxsize]

(Version 1.2+) Checks that the total size of a particular filesystem is between _minsize_ and _maxsize_ (inclusive). Either may be blank; to check for a specific size, pass the same value for both parameters. The value(s) for _minsize_ and/or _maxsize_ are specified as positive integers with an optional suffix (*k* or *kB* for kilobytes, the default; *M* or *MB* for megabytes; *G* or *GB* for gigabytes; etc., all case insensitive).

> **WARNING:** Use of this check requires execution of the */usr/bin/df* command which may HANG in cases of NFS failure! If you use this check, consider also using [Detached Mode](#detached-mode)!

**Example**: *check_fs_size /tmp 512m 4g*

check_fs_used

check_fs_used mountpoint maxused

(Version 1.2+) Checks that a particular filesystem has less than _maxused_ space consumed. The value for _maxused_ may be specified either as a percentage or a numerical value with an optional suffix (*k* or *kB* for kilobytes, the default; *M* or *MB* for megabytes; *G* or *GB* for gigabytes; etc., all case insensitive).

> **WARNING:** Use of this check requires execution of the */usr/bin/df* command which may HANG in cases of NFS failure! If you use this check, consider also using [Detached Mode](#detached-mode)!

**Example**: *check_fs_used / 98%*

check_hw_cpuinfo

check_hw_cpuinfo [sockets] [cores] [threads]

check_hw_cpuinfo compares the properties of the OS-detected CPU(s) to the specified values to ensure that the correct number of physical sockets, execution cores, and "threads" (or "virtual cores") are present and functioning on the system. For a single-core, non-hyperthreading-enabled processor, all 3 parameters would be identical. Multicore CPUs will have more _cores_ than _sockets_, and CPUs with [Intel HyperThreading Technology (HT)](https://en.wikipedia.org/wiki/Hyper-threading) turned on will have more _threads_ than _cores_. Since HPC workloads often suffer when HT is active, this check is a handy way to make sure that doesn't happen.

**Example** (dual-socket 4-core Intel Nehalem with HT turned off): *check_hw_cpuinfo 2 8 8*

check_hw_eth

check_hw_eth device

check_hw_eth verifies that a particular Ethernet device is available. Note that it cannot check for IP configuration at this time.

**Example**: *check_hw_eth eth0*

check_hw_gm

check_hw_gm device

check_hw_gm verifies that the specified Myrinet device is available. This check will fail if the Myrinet kernel drivers are not loaded but does not distinguish between missing drivers and a missing interface.

**Example**: *check_hw_gm myri0*

check_hw_ib

check_hw_ib rate [device]

check_hw_ib determines whether or not an active IB link is present with the specified data rate (in Gb/sec). Version 1.3 and later support the *device* parameter for specifying the name of the IB device. Version 1.4.1 and later also verify that the kernel drivers and userspace libraries are the same OFED version.

**Example** (QDR Infiniband): *check_hw_ib 40*

check_hw_mcelog

check_hw_mcelog

check_hw_mcelog queries the running *mcelog* daemon, if present. If the daemon is not running or has detected no errors, the check passes. If errors are present, the check fails and sends the output to the log file and syslog.

The default behavior is to run *mcelog –client* but is configurable via the *$MCELOG* and *$MCELOG_ARGS* variables.

(Version 1.4.1 and higher) *check_hw_mcelog* will now also check the correctable and uncorrectable error counts in the past 24 hours and compare them to the settings *$MCELOG_MAX_CORRECTED_RATE* and *$MCELOG_MAX_UNCORRECTED_RATE*, respectively; if either actual count exceeds the value specified in the threshold, the check will fail. Set either or both variables to the empty string to obtain the old behavior.

check_hw_mem

check_hw_mem min_kb max_kb

check_hw_mem compares the total system memory (RAM + swap) with the minimum and maximum values provided (in kB). If the total memory is less than _min_kb_ or more than _max_kb_ kilobytes, the check fails. To require an exact amount of memory, use the same value for both parameters.

**Example** (exactly 26 GB system memory required): *check_hw_mem 27262976 27262976*

check_hw_mem_free

check_hw_mem_free min_kb

check_hw_mem_free adds the free physical RAM to the free swap (see below for details) and compares that to the minimum provided (in kB). If the total free memory is less than _min_kb_ kilobytes, the check fails.

**Example** (require at least 640 kB free): *check_hw_mem_free 640*

check_hw_physmem

check_hw_physmem min_kb max_kb

check_hw_physmem compares the amount of physical memory (RAM) present in the system with the minimum and maximum values provided (in kB). If the physical memory is less than _min_kb_ or more than _max_kb_ kilobytes, the check fails. To require an exact amount of RAM, use the same value for both parameters.

**Example** (at least 12 GB RAM/node, no more than 48 GB): *check_hw_physmem 12582912 50331648*

check_hw_physmem_free

check_hw_physmem_free min_kb

check_hw_physmem_free compares the free physical RAM to the minimum provided (in kB). If less than _min_kb_ kilobytes of physical RAM are free, the check fails. For purposes of this calculation, kernel buffers and cache are considered to be free memory.

**Example** (require at least 1 kB free): *check_hw_physmem_free 1*

check_hw_swap

check_hw_swap min_kb max_kb

check_hw_swap compares the total system virtual memory (swap) size with the minimum and maximum values provided (in kB). If the total swap size is less than _min_kb_ or more than _max_kb_ kilobytes, the check fails. To require an exact amount of memory, use the same value for both parameters.

**Example** (at most 2 GB swap): *check_hw_swap 0 2097152*

check_hw_swap_free

check_hw_swap_free min_kb

check_hw_swap_free compares the amount of free virtual memory to the minimum provided (in kB). If the total free swap is less than _min_kb_ kilobytes, the check fails.

**Example** (require at least 1 GB free): *check_hw_swap_free 1048576*

check_moab_sched

check_moab_sched [-t timeout] [-a alert_match] [-m [!]mstr] [-v version_match]

check_moab_sched executes *mdiag -S -v* and examines its output, similarly to *check_cmd_output*. In addition to the arbitrary positive/negative *mstr* [match strings](#match-strings), it also accepts an alert_match‘_ for flagging specific Moab alerts and a _‘version_match‘_ for making sure the expected version is running. The check will fail based on any of these matches, or if ‘mdiag does not return within the specified timeout.

**Example** (ensure we’re running Moab 7.2.3 and it’s not paused): *check_moab_sched -t 45 -m ‘!/PAUSED/’ -v 7.2.3*

check_moab_rm

check_moab_rm [-t timeout] [-m [!]mstr]

check_moab_rm executes *mdiag -R -v* and examines its output, similarly to *check_cmd_output*. In addition to the arbitrary positive/negative *mstr* [match strings](#match-strings), it also checks for any RMs which are not in the *Active* state (and fails if any are inactive). The check will also fail if *mdiag* does not return within the specified timeout.

**Example** (basic Moab RM sanity check): *check_moab_rm -t 45*

check_moab_torque

check_moab_torque [-t timeout] [-m [!]mstr]

check_moab_torque executes *qmgr -c 'print server'* and examines its output, similarly to *check_cmd_output*. In addition to the arbitrary positive/negative *mstr* [match strings](#match-strings), it also checks to make sure that the *scheduling* parameter is set to *True* (and fails if it isn't). The check will also fail if *qmgr* does not return within the specified timeout.

**Example** (basic TORQUE configuration/responsiveness sanity check): *check_moab_torque -t 45*

check_net_ping

check_net_ping [-I interface] [-W timeout] [-c count] [-i interval] [-s packetsize] [-t ttl] [-w deadline] target(s)

(Version 1.4.2+) *check_net_ping* provides an NHC-based wrapper around the standard Linux/UNIX *ping* command. The most common command-line options for *ping* are supported, and any number of hostnames and/or IP addresses may be specified as *targets*. All options specified on the *check_net_ping* command line are passed directly to *ping -q -n* for each *target* specified. The following options are supported:

Check Option | Purpose |
———————- | ———— |
-I interface | *interface* is either an address or an interface name from which to send packets |
-W timeout | Wait *timeout* seconds for a response |
-c count | Stop after sending *count* packets |
-i interval | Wait *interval* seconds before sending each packet |
-s packetsize | Specifies that packets with *packetsize* bytes of data be sent |
-t ttl | Set IP Time To Live in each packet to *ttl* |
-w deadline | *ping* will exit after *deadline* seconds regardless of how many packets were sent/received |

**Example** (check network connectivity to master, io, and xfer nodes): *check_net_ping -W 3 -i 0.25 -c 5 master io000 xfer*

check_net_socket

check_net_socket [-0] [-a] [-!] [-n <name>] [-p <proto>] [-s <state>] [-l <locaddr>[:<locport>]] [-r <rmtaddr>[:<rmtport>]] [-t <type>] [-u <user>] [-d <daemon>] [-e <action>] [-E <found_action>]

(Version 1.4.1+) *check_net_socket* executes either the command *$NETSTAT_CMD $NETSTAT_ARGS* (default: *netstat -Tanpee -A inet,inet6,unix*) or (if *$NETSTAT_CMD* is not in *$PATH*) the command *$SS_CMD $SS_ARGS* (default: *ss -anpee -A inet,unix*). The output of the command is parsed for socket information. Then each socket is compared with the match criteria passed in to the check: protocol *proto*, state *state*, local and/or remote address(es) *locaddr*/*rmtaddr* with optional ports *locport*/*rmtport*, type *type*, owner *user*, and/or process name daemon‘_. If a matching socket is found, _‘found_action‘_ is executed, and the check returns successfully. If no match is found, _‘action‘_ is executed, and the check fails. Reverse the success/failure logic by specifying ‘-! (i.e., if NHC finds one or more matching sockets, the check will fail).

The *name* parameter may be used to label the type of socket being sought (e.g., *-n 'SSH daemon TCP listening socket'*). If *-0* is specified, the check is non-fatal (i.e., missing matches will be noted but will not terminate NHC. Use *-a* to locate all matching sockets (mainly for debugging).

**Example** (search for HTTP daemon IPv4 listening socket and restart if missing): *check_net_socket -n "HTTP daemon" -p tcp -s LISTEN -l '0.0.0.0:80' -d httpd -e 'service httpd start'*

check_nv_healthmon

check_nv_healthmon

(Version 1.2+) *check_nv_healthmon* runs the command *$NVIDIA_HEALTHMON* (default: *nvidia-healthmon*) with the arguments specified in *$NVIDIA_HEALTHMON_ARGS* (default: *-e -v*) to check for problems with any nVidia Tesla GPU devices on the system. If any errors are found, the entire (human-readable) output of the command is logged, and the check fails. **NOTE:** Version 3.304 or higher of the nVidia Tesla Deployment Kit (TDK) is required! See <http://developer.nvidia.com/cuda/tesla-deployment-kit> for details and downloads.

**Example**: *check_nv_healthmon*

check_ps_blacklist

(_deprecated_) *check_ps_blacklist command [[!]owner] [args]*

(Version 1.2+) *check_ps_blacklist* looks for a running process matching _command_ (or, if _args_ is specified, _command+args_). If _owner_ is specified, the process must be owned by _owner_; if the optional *!* is also specified, the process must NOT be owned by _owner_. If any matching process is found, the check fails. (This is the opposite of *check_ps_daemon*.)

Note: This check (as well as its complementary check, *check_ps_daemon*) has largely been replaced with *check_ps_service*. The latter should be used instead whenever possible.

**Example** (prohibit sshd NOT owned by root): *check_ps_blacklist sshd !root*

check_ps_cpu

check_ps_cpu [-0] [-a] [-f] [-K] [-k] [-l] [-s] [-u [!]user] [-m [!]mstr] [-e action] threshold

(Version 1.4+) *check_ps_cpu* is a resource consumption check. It flags any/all matching processes whose current percentage of CPU utilization meets or exceeds the specified _threshold_. The % suffix on the _threshold_ is optional but fully supported. Options to this check are as follows:

Check Option | Purpose |
———————- | ——— |
-0 | Non-fatal. Failure of this check will be ignored. |
-a | Find, report, and act on all matching processes. Default behavior is to fail check after first matching process. |
-e action | Execute */bin/bash -c <action>* if matching process is found. |
-f | Full match. Match against entire command line, not just first word. |
-K | Kill **parent** of matching process (or processes, if used with *-a*) with SIGKILL. (NOTE: Does NOT imply *-k*) |
-k | Kill matching process (or processes, if used with *-a*) with SIGKILL. |
-l | Log matching process (or processes, if used with *-a*) to NHC log (*$LOGFILE*). |
-m mstr | Look only at processes matching _mstr_ (NHC [match string](#match-strings), possibly negated). Default is to check all processes. |
-r value | Renice matching process (or processes, if used with *-a*) by the specified *value* (may be positive or negative). |
-s | Log matching process (or processes, if used with *-a*) to the syslog. |

-u [!] 'user | User match. Matches only processes owned by _user_ (or, if negated, NOT owned by _user_). |

**Example** (look for non-root-owned process consuming 99% CPU or more; renice it to the max): *check_ps_cpu -u !root -r 20 99%*

check_ps_daemon

(_deprecated_) *check_ps_daemon command [owner] [args]*

check_ps_daemon looks for a running process matching _command_ (or, if _args_ is specified, _command+args_). If _owner_ is specified, the process must be owned by _owner_. If no matching process is found, the check fails.

Note: This check (as well as its complementary check, *check_ps_blacklist*) has largely been replaced with *check_ps_service*. The latter should be used instead whenever possible.

**Example** (look for a root-owned sshd): *check_ps_daemon sshd root*

check_ps_kswapd

check_ps_kswapd cpu_time discrepancy [action [actions...]]

check_ps_kswapd compares the accumulated CPU time (in seconds) between *kswapd* kernel threads to make sure there's no imbalance among different NUMA nodes (which could be an early symptom of failure). Threads may not exceed _cpu_time_ seconds nor differ by more than a factor of _discrepancy_. Unlike most checks, *check_ps_kswapd* need not be fatal. Zero or more _actions_ may be specified from the following allowed actions: *ignore* (do nothing), *log* (write error to log file and continue), *syslog* (write error to syslog and continue), or *die* (fail the check as normal). The default is "*die*" if no _action_ is specified.

**Example** (max 500 CPU hours, 100x discrepancy limit, only log and syslog on error): *check_ps_kswapd 1800000 100 log syslog*

check_ps_loadavg

check_ps_loadavg limit_1m limit_5m limit_15m

check_ps_loadavg looks at the 1-minute, 5-minute, and 15-minute load averages reported by the kernel and compares them to the parameters limit_1m'_, _'limit_5m'_, and _'limit_15m'_, respectively. If any limit has been exceeded, the check fails. Limits which are empty (i.e., ''') or not supplied are ignored (i.e., assumed to be infinite) and will never fail.

**Example** (ensure the 5-minute load average stays below 30): *check_ps_loadavg '' 30*

check_ps_mem

check_ps_mem [-0] [-a] [-f] [-K] [-k] [-l] [-s] [-u [!]user] [-m [!]mstr] [-e action] threshold

(Version 1.4+) *check_ps_mem* is a resource consumption check. It flags any/all matching processes whose total memory consumption (including both physical and virtual memory) meets or exceeds the specified _threshold_. The _threshold_ is interpreted as kilobytes (1024 bytes) or can use NHC's standard byte-suffix syntax (e.g., *32GB*). Percentages are not supported for this check at this time. Options to this check are as follows:

Check Option | Purpose |

———————- | ——— |
-0 | Non-fatal. Failure of this check will be ignored. |
-a | Find, report, and act on all matching processes. Default behavior is to fail check after first matching process. |
-e action | Execute */bin/bash -c <action>* if matching process is found. |
-f | Full match. Match against entire command line, not just first word. |
-K | Kill **parent** of matching process (or processes, if used with *-a*) with SIGKILL. (NOTE: Does NOT imply *-k*) |
-k | Kill matching process (or processes, if used with *-a*) with SIGKILL. |
-l | Log matching process (or processes, if used with *-a*) to NHC log (*$LOGFILE*). |
-m mstr | Look only at processes matching _mstr_ (NHC [match string](#match-strings), possibly negated). Default is to check all processes. |
-r value | Renice matching process (or processes, if used with *-a*) by the specified *value* (may be positive or negative). |
-s | Log matching process (or processes, if used with *-a*) to the syslog. |
-u [!]_‘user‘_ | User match. Matches only processes owned by _user_ (or, if negated, NOT owned by _user_). |

**Example** (look for process owned by *baduser* consuming 32GB or more of memory; log, syslog, and kill it): *check_ps_mem -u baduser -l -s -k 32G*

check_ps_physmem

check_ps_physmem [-0] [-a] [-f] [-K] [-k] [-l] [-s] [-u [!]user] [-m [!]mstr] [-e action] threshold

(Version 1.4+) *check_ps_physmem* is a resource consumption check. It flags any/all matching processes whose physical memory consumption (i.e., resident RAM only) meets or exceeds the specified _threshold_. The _threshold_ is interpreted as a percentage if followed by a %, or as a number of kilobytes (1024 bytes) if numeric only, or can use NHC's standard byte-suffix syntax (e.g., *32GB*). Options to this check are as follows:

Check Option | **Purpose** |
———————- | ——— |
-0 | Non-fatal. Failure of this check will be ignored. |
-a | Find, report, and act on all matching processes. Default behavior is to fail check after first matching process. |
-e action | Execute */bin/bash -c <action>* if matching process is found. |
-f | Full match. Match against entire command line, not just first word. |
-K | Kill **parent** of matching process (or processes, if used with *-a*) with SIGKILL. (NOTE: Does NOT imply *-k*) |
-k | Kill matching process (or processes, if used with *-a*) with SIGKILL. |
-l | Log matching process (or processes, if used with *-a*) to NHC log (*$LOGFILE*). |
-m mstr | Look only at processes matching _match_ (NHC [match string](#match-strings), possibly negated). Default is to check all processes. |
-r value | Renice matching process (or processes, if used with *-a*) by the specified *value* (may be positive or negative). |
-s | Log matching process (or processes, if used with *-a*) to the syslog. |
-u [!]_‘user‘_ | User match. Matches only processes owned by _user_ (or, if negated, NOT owned by _user_). |

**Example** (look for all non-root-owned processes consuming more than 20% of system RAM; syslog and kill them all, but continue running): *check_ps_physmem -0 -a -u !root -s -k 20%*

check_ps_service

check_ps_service [-0] [-f] [-S|-r|-c|-s|-k] [-u user] [-d daemon | -m mstr] [-e action | -E action] service

(Version 1.4+) *check_ps_service* is similar to *check_ps_daemon* except it has the ability to start, restart, or cycle services which aren't running but should be, and to stop or kill services which shouldn't be running but are. Options to this check are as follows:

Check Option | Purpose |
———————- | ———— |
-0 | Non-fatal. Failure of this check will be ignored. |
-S | Start service. Service *service* will be started if not found running. Equivalent to *-e '/sbin/service '_'service'_* start'' |
-c | Cycle service. Service *service* will be cycled if not found running. Equivalent to *-e '/sbin/service '_'service'_* stop ; sleep 2 ; /sbin/service _'*service*'_ start'' |
-d daemon | Match running process by *daemon* instead of *service*. Equivalent to *-m '*daemon'* |
-e action | Execute */bin/bash -c <action>* if process IS NOT found running. |
-E action | Execute */bin/bash -c <action>* if process IS found running. |
-f | Full match. Match against entire command line, not just first word. |
-k | Kill service. Service *service* will be killed (and check will fail) if found running. Similar to *pkill -9 <service>* |
-m mstr | Use *match* to search the process list for the service. Default is *service* |
-r | Restart service. Service *service* will be restarted if not found running. Equivalent to *-e '/sbin/service '_'service'_* restart'' |
-s | Stop service. Service *service* will be stopped (and check will fail) if found running. Performs */sbin/service <service>* stop' |
-u [!]_'user'_ | User match. Matches only processes owned by _user_ (or, if negated, NOT owned by _user_). |

**Example** (look for a root-owned sshd and start if missing): *check_ps_service -u root -S sshd*

check_ps_time

check_ps_time [-0] [-a] [-f] [-K] [-k] [-l] [-s] [-u [!]user] [-m [!]mstr] [-e action] threshold

(Version 1.4+) *check_ps_time* is a resource consumption check. It flags any/all matching processes whose total utilization of CPU time meets or exceeds the specified _threshold_. The _threshold_ is a quantity of minutes suffixed by an *M* and/or a quantity of seconds suffixed by an *S*. A number with no suffix is interpreted as seconds. Options to this check are as follows:

Check Option | Purpose |
———————- | ———— |
-0 | Non-fatal. Failure of this check will be ignored. |
-a | Find, report, and act on all matching processes. Default behavior is to fail check after first matching process. |
-e action | Execute */bin/bash -c <action>* if matching process is found. |
-f | Full match. Match against entire command line, not just first word. |
-K | Kill **parent** of matching process (or processes, if used with *-a*) with SIGKILL. (NOTE: Does NOT imply *-k*) |
-k | Kill matching process (or processes, if used with *-a*) with SIGKILL. |
-l | Log matching process (or processes, if used with *-a*) to NHC log (*$LOGFILE*). |

-m mstr | Look only at processes matching _match_ (NHC [match string](#match-strings), possibly negated). Default is to check all processes. |

-r value | Renice matching process (or processes, if used with *-a*) by the specified *value* (may be positive or negative). |

-s | Log matching process (or processes, if used with *-a*) to the syslog. |

-u [!]_'user'_ | User match. Matches only processes owned by _user_ (or, if negated, NOT owned by _user_). |

**Example** (look for *runawayd* daemon process consuming more than a day of CPU time; restart service and continue running): *check_ps_time -0 -m '/runawayd/' -e '/sbin/service runawayd restart' 3600m*

check_ps_unauth_users

check_ps_unauth_users [action [actions...]]

check_ps_unauth_users examines all processes running on the system to determine if the owner of each process is authorized to be on the system. Authorized users are anyone with a UID below, by default, 100 (including root) and any users currently running jobs on the node. All other processes are unauthorized. If an unauthorized user process is found, the specified action(s) are taken. The following actions are valid: *kill* (terminate the process), *ignore* (do nothing), *log* (write error to log file and continue), *syslog* (write error to syslog and continue), or *die* (fail the check as normal). The default is "*die*" if no _action_ is specified.

**Example** (log, syslog, and kill rogue user processes): *check_ps_unauth_users log syslog kill*

check_ps_userproc_lineage

check_ps_userproc_lineage [action [actions...]]

check_ps_userproc_lineage examines all processes running on the system to check for any processes not owned by an "authorized user" (see previous check) which are not children (directly or indirectly) of the Resource Manager daemon. Refer to the *$RM_DAEMON_MATCH* [configuration variable](#supported-variables) for how NHC determines the RM daemon process. If such a rogue process is found, the specified action(s) are taken. The following actions are valid: *kill* (terminate the process), *ignore* (do nothing), *log* (write error to log file and continue), *syslog* (write error to syslog and continue), or *die* (fail the check as normal). The default is "*die*" if no *action* is specified.

**Example** (mark the node bad on rogue user processes): *check_ps_userproc_lineage die*

1.3 Customization

Once you've fully configured NHC to run the built-in checks you need for your nodes, you're probably at the point where you've thought of something else you wish it could do but currently can't. NHC's design makes it very easy to create additional checks for your site and have NHC load and use them at runtime. This section will detail how to create new checks, where to place them, and what NHC will do with them.

While technically a "check" can be anything the *nhc* driver script can execute, for consistency and extensibility purposes (as well as usefulness to others), we prefer and recommend that checks be shell functions defined in a distinct, namespaced *.nhc* file. The instructions contained in this section will assume that this is the model you wish to use.

Note: If you do choose to write your own checks, and you feel they might be useful to the NHC community, we encourage you to share them. You can either e-mail them to the [NHC Developers' Mailing List](mailto:nhc-devel@lbl.gov) or submit a [Pull Request](https://github.com/mej/nhc/compare/master...mej:master) on [GitHub](https://github.com/). GitHub PRs are definitely preferred, but if you choose to use e-mail instead, please

provide either individual file attachments or a unified diff (i.e., *diff -Nurp*) against the NHC tarball/git tree if at all possible (though any usable format will likely be accepted).

1.3.1 Writing Checks

The first decision to be made is what to name your check file. As mentioned above, check files live (by default; see the *$INCDIR* and *$CONFDIR* [configuration variables](#supported-variables)) in */etc/nhc/scripts/* and are named *something.nhc*‘^{[2](#footnotes)}. *A file containing utility and general-purpose functions called ‘common.nhc* can be found here. All other files placed here by the upstream package follow the naming convention siteid_class‘_.nhc‘ (e.g., the NHC project’s file containing hardware checks is named *lbnl_hw.nhc*). Your siteid‘_ can be anything you’d like (other than ‘lbnl, obviously) but should be recognizable. The *class* should refer to the subsystem or conceptual group of things you’ll be monitoring.

For purposes of this example, we’ll pretend we’re from John Sheridan University, using site abbreviation “*jsu*,” and we want to write checks for our “*stuff*.”

Your */etc/nhc/scripts/jsu_stuff.nhc* file should start with a header which provides a summary of what will be checked, the name and e-mail of the author, possibly the date or other useful information, and any copyright or license restrictions you are placing on the file^{[3](#footnotes)}. It should look something like this:

```
# NHC -- John Sheridan University's Checks for Stuff
#
# Your Name <you@your.com>
# Date
#
# Copyright and/or license information if different from upstream
#
```

Next, initialize any variables you will use to sane defaults. This does two things: it provides anyone reading your code a single place to look for “global” variables, and it makes sure you have something to test for later if you need to check the existence of cache data. Make sure your variables are properly namespaced; that is, they should start with a prefix corresponding to your site, the system you’re checking, etc.

```
# Initialize variables
declare STUFF_NORMAL_VARIABLE=""
declare -a STUFF_ARRAY_VARIABLE=( )
declare -A STUFF_HASH_VARIABLE=( )
```

If your check may run more than once and does anything that’s resource-intensive (running subprocesses, file I/O, etc.), you should (in most cases, unless it would cause malfunctions to occur) perform the intensive tasks only once and store the information in one or more shell variables for later use. These should be the variables you just initialized in the section above. They can be arrays or scalars.

```
# Function to populate data structures with data
function nhc_stuff_gather_data() {
    # Gather and cache data for later use.
    STUFF_NORMAL_VARIABLE="value"
    STUFF_ARRAY_VARIABLE=( "value" )
    STUFF_HASH_VARIABLE=( [key]="value" )
}
```

Next, you need to write your check function(s). These should be named <code>check_class_purpose</code> where *class* is the same as used previously (“*stuff*” for this example), and *purpose* gives a descriptive name to the check to convey what it checks. Our example will use the obvious-but-potentially-vague “works” as its purpose, but the name you choose will undoubtedly be more clever.

If you have created a data-gathering function as shown above and populated one or more cache variables, the first thing your check should do is see if the cache has been populated already. If not, run your data-gathering function before proceeding with the check.

As for how you write the check...well, that's entirely up to you. It will depend on what you need to check and the available options for doing so. (However, consult the next section for some tips and bashisms to make your checks more efficient.) The example here is clearly a useless and contrived one but should nevertheless be illustrative of the general concept:

```
# Check to make sure stuff is functioning properly
function check_stuff_works() {
    # Load cache if empty
    if [[ ${#STUFF_ARRAY_VARIABLE[*]} -eq 0 ]]; then
        nhc_stuff_gather_data
    fi

    # Use cached data
    if [[ "${STUFF_ARRAY_VARIABLE[0]}" = "" ]]; then
        die "Stuff is not working"
    fi
    return 0
}
```

If other check functions are needed for a particular subsystem, write those similarly. If you're using a cache, each check should look for (and call the *gather* function if necessary) the cache variables before doing the actual checking as shown above.

Once you have all the checks you need, you can add them to the configuration file on your node(s), like so:

```
*  || check_stuff_works
```

Next time NHC runs, it will automatically pick up your new check(s)!

1.3.2 Tips and Best Practices for Checks

Several of the philosophies and underlying principles which governed the design and implementation of the LBNL Node Health Check project were mentioned above in the [Introduction](#lbnl-node-health-check-nhc). Certain code constructs were used to fulfill these principles which are not typical for the average run-of-the-mill shell script, largely because things which must be highly performant tend not to be written as shell scripts. Why? Two reasons: (1) It doesn't have a lot of the fancier, more complex features of the dedicated (i.e., non-shell) scripting languages; and (2) Many script authors don't know of many of the features bash _does_ offer because they're used so infrequently. It can be somewhat of a vicious cycle/feedback loop when nobody bothers to learn something specifically because no one else is using it.

So why was bash chosen for this project? Simple: it's everywhere. If you're running Linux, it's almost guaranteed to be there^{[4](#footnotes)}. The same cannot be said of any other scripting or non-compiled language (not even PERL or Python). And forcing everyone to write their checks in C or another compiled language would raise the barrier to entry and reduce the number of sites for which NHC could be useful. Since half the point is getting more places using a common tool (or at least a common framework), that would defeat the purpose. Thus, bash made the most sense.

The important question, then, becomes how to make bash scripts more efficient. And the solution is clear: do as much as possible with native bash constructs instead of shelling out to subcommands like *sed*, *awk*, *grep*, and the other common UNIX swashbucklers. The more one investigates the features bash provides, the more one finds how many of its long-held features tend to go unused and just how much one truly _is_ able to do without the need to fork-and-exec. In this section, several aspects of common shell script constructs (plus 1 or 2 not-so-common ones) will be reviewed along with ways to improve efficiency and avoid subcommands whenever possible.

Arrays

Arrays are an important tool in any sufficiently-capable scripting language. Bash has had support for arrays for quite some time; recent versions even add associative array support (i.e., string-based indexing, akin to hashes in PERL). To maintain compatibility, associative arrays are not currently used in NHC, but traditional arrays are used quite heavily. Though a complete tutorial on arrays in bash is beyond the scope of this document, a brief "cheat sheet" is probably a good idea. So here you go:

Syntax | Purpose |
——— | ———- |
declare -a AVAR | Declare the shell variable *$AVAR* to be an array (not strictly required, but good form). |
AVAR=(...) | Assign elements of array *$AVAR* based on the word expansion of the contents of the parentheses. ... is one or more words of the form *['subscript*]='value' or an expression which expands to such. Only the 'value'(s) are required. |
${AVAR['subscript]}'* | Evaluates to the *subscript'th element of the array '$AVAR. Array indexes in bash start from 0, just like in C or PERL. *subscript* must evaluate to an integer >= 0. |
${#AVAR[*]}* | Evaluates to the number of elements in the array *$AVAR. |
${AVAR[*]}* | Evaluates to all the values in the *$AVAR array as a single word (like *$**). Use only where keeping values separate doesn't matter. |
"${AVAR[@]}"* | Evaluates to all values in the *$AVAR array, each as a separate word. This keeps values distinct (just like *$@* vs. *$**). |
<code>"${AVAR[@]:offset:length}"</code> | Evaluates to the values of *$AVAR* as above, starting at element <code>${AVAR[offset]}</code> and including at most *length* elements. *length* may be omitted, and *offset* may be negative. |

A more detailed examination of bash arrays can be found [here](http://tldp.org/LDP/abs/html/arrays.html).

Several examples of array-based techniques will appear in the following sections, so make sure you have a solid grasp on the basic usage of array syntax before continuing.

File I/O

When using the command prompt, most of us reach for things like *cat* or *less* when we need to view the contents of a file; thus, our inclination tends to be to reach for the same tools when writing shell scripts. *cat*, however, is not a bash built-in, so a fork-and-exec is required to spawn */bin/cat* just so it can read a file and return the contents. This overhead is negligible for interactive shell usage, and may be a non-issue for many shell-scripting scenarios, but for efficiency-critical scenarios like NHC, we can and should do better!

File input and output (either truncate or append) are both natively supported by bash using the (mostly) well-known [Redirection Operators](https://www.gnu.org/software/bash/manual/html_node/Redirections.html). Rather than reading data from files into variables (arrays or scalars) using command substitution (i.e., the `` and *$()* operators), use redirection operators to pull the contents of the file into the variable. One technique for doing this is to redirect to the *read* built-in. So instead of this:

```
MOTD=`cat /etc/motd`
```

use:

```
read MOTD < /etc/motd
```

bash also allows an even simpler form for using this technique:

```
MOTD=$(< /etc/motd)
```

It looks similar to command substitution but uses I/O redirection in place of an actual command. It **does**, however, still do a *fork()* and *pipe()* to do the file I/O. On Linux, this is done via *clone()* which is fairly lightweight but still not quite as efficient as the *read* command shown above (which is a *bash* built-in).

The same syntax can be used to populate array variables with multiple fields' worth of data:

```
UPTIME=( $(< /proc/uptime) )
```

This will store the system uptime (in seconds) in the variable *${UPTIME[0]}* and the idle time in *${UPTIME[1]}*. Declare *$UPTIME* as an array in advance using *declare -a* or *local -a* to make this clearer, and (as always!) make sure to add comments! To avoid the *fork()* (see above), use *read* instead:

```
read -a UPTIME < /proc/uptime
```

Though not as easy to spot, other subcommands may also be able to be eliminated using this technique. For example, the Linux kernel makes the full hostname of the system available in a file in the */proc* filesystem. Knowing this, the *hostname* command substitution may be eliminated by utilizing the contents of this file:

```
read HOSTNAME < /proc/sys/kernel/hostname
```

As an aside... Knowing these tricks may also be helpful in other situations. If you're trying to repair a system in which the root filesystem has become partially corrupted, and the *cat* command no longer works, this can provide you a way to view the contents of system files directly in your shell!

Line Parsing and Loops

While certainly not as capable as [PERL](http://www.perl.org/) at text processing, the shell does offer some seldom-used features to facilitate the processing of line-oriented input. By default, the shell splits things up based on whitespace (i.e., space characters, tabs, and newlines) to distinguish each "word" from the next. This is why quoting must be used to join arguments which contain spaces to allow them to be treated as single parameters. As with many aspects of the shell, however, this behavior can be customized, allowing for different delimiter characters to be applied to input (typically file I/O). Since character-delimited files are commonplace in UNIX, this idiom is quite frequently useful when shell scripting.

One easily-recognized example would be */etc/passwd*. It is both line-oriented and colon-delimited. Parsing its contents is often useful for shell scripts, but most which need this data tend to use *awk* or *cut* to pull the appropriate fields. Direct splitting and parsing of this file can be done in native bash without the use of subcommands:

```
IFS=':'
while read -a LINE ; do
    THIS_UID=${LINE[2]}
    UIDS[${#UIDS[*]}]=$THIS_UID
    PWDATA_USER[$THIS_UID]="${LINE[0]}"
    PWDATA_GID[$THIS_UID]=${LINE[3]}
    PWDATA_GECOS[$THIS_UID]="${LINE[4]}"
    PWDATA_HOME[$THIS_UID]="${LINE[5]}"
    PWDATA_SHELL[$THIS_UID]="${LINE[6]}"
done < /etc/passwd
IFS=$' \t\n'
```

The above code reads a line at a time from */etc/passwd* into the *$LINE* array. Because the bash Input Field Separator variable, *$IFS*, has been set to a colon ('*:*') instead of whitespace, each field of the *passwd* file will go into a separate element of the *$LINE* array. The values in *$LINE* are then used to populate 5 parallel arrays with the userid, GID, GECOS field, home directory, and shell for each user (indexed by UID). It also keeps an array of all the UIDs it has seen. Everything here is done in the same bash process which is executing the script, so it is quite efficient. The *$IFS* variable is reset to its proper value after the loop completes.

Sometimes, however, the elimination of a subprocess is impractical or impossible. A similar approach may still be used to keep the parsing of the command's output as efficient as possible. For example, a bash-native implementation of the *netstat -nap* command would be impossible (or at least a very close approximation thereof), so we could use the following method to populate our cache data from its output:

```
IFS=$'\n'
LINES=( $(netstat -nap) )

IDX=0
for ((i=0; i<${#LINES[*]}; i++)); do
    IFS=$' \t\n'
    LINE=( ${LINES[$i]} )
    if [[ "${LINE[0]}" != "tcp" && "${LINE[0]}" != "udp" ]]; then
        continue
    fi
    NET_PROTO[$IDX]=${LINE[0]}
    NET_RECVQ[$IDX]=${LINE[1]}
    NET_SENDQ[$IDX]=${LINE[2]}
    NET_LOCADDR[$IDX]=${LINE[3]}
    NET_REMADDR[$IDX]=${LINE[4]}
    if [[ "${NET_PROTO[$IDX]}" == "tcp" ]]; then
        NET_STATE[$IDX]=${LINE[5]}
        NET_PROC[$IDX]=${LINE[6]}
    else
        NET_STATE[$IDX]=""
        NET_PROC[$IDX]=${LINE[5]}
    fi
    if [[ "${NET_PROC[$IDX]}" == */* ]]; then
        IFS='/'
        LINE=( ${NET_PROC[$IDX]} )
        NET_PROCPID[$IDX]=${LINE[0]}
        NET_PROCNAME[$IDX]=${LINE[1]}
    else
        NET_PROCPID[$IDX]='???'
        NET_PROCNAME[$IDX]="unknown"
    fi
    ((IDX++))
done
IFS=$' \t\n'
```

By resetting *$IFS* to contain only a newline character, we can easily split the command results into individual lines. We place these results into the *$LINES* array. Each line is then split on the traditional whitespace characters and placed into the *$LINE* (with no '*S*' on the end) array. We're tracking only TCP and UDP sockets here, so everything else (including column headers) gets thrown away. We store each field in our cache arrays, and we even further split one of the fields which uses '*/*' as a separator. After our loop is complete, we reset *$IFS*, and we now have a fully-populated set of cache variables containing all our TCP- and UDP-based sockets, all with only 1 fork-and-exec required!

Text Transformations

Bash got a regular expression matching operator in version 3, but it still lacks regex-based transforms. However, with a minimum of extra effort, glob-based transforms can often provide the necessary functionality.

The following basic variable transformations are available:

Syntax | Purpose |
——— | ——- |

<code>${VAR:offset}</code> | Evaluates to the substring of *$VAR* starting at *offset* and continuing until the end of the string. If *offset* is negative, it is interpreted relative to the end of *$VAR*. |

<code>${VAR:offset:length}</code> | Same as above, but the result will contain at most *length* characters from *$VAR*. |

${#VAR}* | Gives the length, in characters, of the value assigned to *$VAR. |

<code>${VAR#pattern}</code> | Removes the shortest string matching *pattern* from the beginning of *$VAR*. |

<code>${VAR##pattern}</code> | Same as above, but the longest string matching *pattern* is removed. |

<code>${VAR%pattern}</code> | Removes the shortest string matching *pattern* from the end of *$VAR*. |

<code>${VAR%%pattern}</code> | Same as above, but the longest string matching *pattern* is removed. |

<code>${VAR/pattern/replacement}</code> | The first string matching *pattern* in *$VAR* is replaced with *replacement*. *replacement* and the last / may be omitted to simply remove the matching string. Patterns starting with # or % must match beginning or end (respectively) of *$VAR*. |

<code>${VAR//pattern/replacement}</code> | Same as above, but ALL strings matching *pattern* are replaced/removed. |

So here are some ways the above constructs can be used to do common operations on strings/files:

Traditional Method | Native *bash* method |
——————— | ——————— |
*sed 's/^ *//'* | *while [["$LINE" != "${LINE## }"]]; do LINE="${LINE## }" ; done* |
*sed 's/ *$//'* | *while [["$LINE" != "${LINE%% }"]]; do LINE="${LINE%% }" ; done* |
echo ${LIST[*]}* | *fgrep string* | *[["${LIST[*]//string}" != "${LIST[*]}"]] |
tail -1 | *${LINES[*]:-1}* |
cat file | *tr 'r' ''* | *LINES=("${LINES[@]//$'r'}")* |

There are infinitely more, of course, but these should get you thinking along the right lines!

Matching

Matching input data against potential or expected patterns is common to all programming, and NHC is no exception. As previously mentioned, however, bash 2.x did not have regular expression matching capability. To abstract this out, NHC's *common.nhc* file (loaded automatically by *nhc* when it runs) provides the *mcheck_regexp()*, *mcheck_range()*, and *mcheck_glob()* functions which return 0 (i.e., bash's "true" or "success" value) if the first argument matches the pattern provided as the second argument. To allow for a single matching interface to support all styles of matching, the *mcheck()* function is also provided. If the pattern is surrounded by slashes (e.g., */pattern/*), *mcheck()* will attempt a regular expression match; if the pattern is surrounded by braces (e.g., *{pattern}*), a range match is attempted; otherwise, it attempts a glob match. (For older bash versions which lack the regex matching operator, *egrep* is used instead...which unfortunately will mean additional subshells.) The *mcheck()* function is used to implement the pattern matching of the first field in *nhc.conf* as well as all other occurrences of [match strings (a.k.a. ‘mstr‘s)](#match-strings) used as check parameters throughout the configuration file.

For consistency with NHC's built-in checks, it is recommended that user-supplied checks which require matching functionality do so by simply calling <code>mcheck string expression</code> and evaluating the return value. If true (i.e., 0), *string* did match the */regex/*, *{range}*, external match expression, or glob supplied as *expression*. If false, the match failed.

See the earlier section on [Match Strings](#match-strings) for details.

1.4 Footnotes

[1]: Actually, nhc-wrapper will strip "-wrapper" off the end of its name and execute whatever remains, or you can specify a subprogram directly using the -P option on the nhc-wrapper command line. It was intentionally written to be somewhat generic in its operation so as to be potentially useful in wrapping other utilities.

[2]: Previously, _any_ file in that directory got loaded __regardless__ of extension. **This is no longer the case**, so use of the *.nhc* extension is now **required**. This change was made to avoid loading **.nhc.rpmnew* files, for example.

[3]: If you don't specify otherwise, all checks made available publicly or directly to the NHC development team at [LBNL](http://www.lbl.gov/) are copyrighted by the author and licensed as specified in the [LBNL-BSD license](https://github.com/mej/nhc/LICENSE) used by NHC.

[4]: Well, okay... If you're running enough of Linux that it can function as a compute node. Bootstrap images and other embedded/super-minimal cases aren't really applicable to NHC anyway.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search