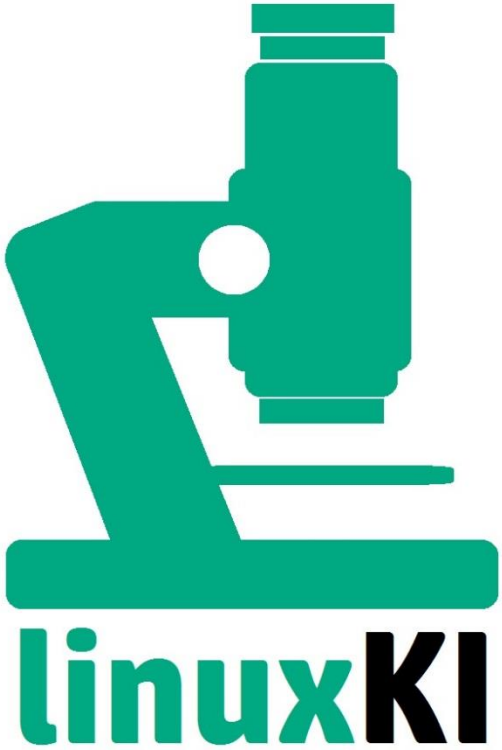
Hewlett Packard Enterprise



MasterClass (v7.2)

Revision 7.2 - November 9th, 2021

Contents

1 Introduction

The LinuxKI Toolset (or LinuxKI for short) is an opensource, application centric, data-driven, advanced mission critical performance troubleshooting tool for Linux. It is designed to identify performance issues beyond the typical performance metrics and results in faster root cause for many performance issues. LinuxKI is a kernel tracing toolkit designed to answer two primary questions about the system:

- If it's running, what's it doing?
- If it's waiting, what's it waiting on?

The LinuxKI Toolset (kiinfo, likit.ko, and supporting scripts) is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License, version 2, as published by the Free Software Foundation.

This toolset is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

WARNING
Be *sure to obtain the customer's permission before running the toolset in production. Any defect in the Linux ftrace code or the Liki trace module could cause a system failure, and customers should be aware of the risks of running the toolset.*

1.1 Why LinuxKI?

Performance analysis has often been a science of symptom-based or metric-driven analysis. For example, you might analyze performance graphs and metrics and try to infer a potential cause based on a spike in CPU or increase in IO service times. While this can be useful, you can often miss a whole class of problems. You can also plug the symptoms into your favorite search engineer to find similar problems and antidotes, or follow some whitepapers or best practices. Or perhaps you can start turning some tuning knobs. The end result is that you are just guessing. Sometimes you get lucky, but often you don't. Customers get frustrated with the "try this, try that" approach, and you end up wasting time and losing credibility with the customer. Certainly, there must be a better way.

What you need is a new approach for demanding mission critical customers and the new style of IT. You need to take out the guess work and reduce the time to resolve critical performance issues with an application-centric, systematic approach to performance troubleshooting and tuning.

1.2 A bit of history...

We (the developers of the LinuxKI) were originally doing performance analysis on HP-UX. HP-UX had a built-in tracing facility called Kernel Instrumentation, or KI for short. So we built a set of tools to collect and analyze the HP-UX KI trace data in new and interesting ways. The tool became popular among HPE performance engineers for helping to resolve complex performance issues very quickly. As Linux became more popular and former HP-UX engineers began to analyze performance issues on Linux, we began hearing the statement – "If we only had a KI trace tool for Linux!"

So an effort was made to "port" the HP-UX KI tracing tool to Linux. While Linux didn't have the same KI tracing facility that HP-UX had, it did have a tracing facility called *ftrace*. The initial prototypes of the LinuxKI used *ftrace* as its primary tracing mechanism, but *ftrace* has some limitations. For example, it didn't do CPU profiling and couldn't collect stack traces when a task went to sleep, and the trace collection wasn't very efficient. So a new tracing mechanism called *LiKI* (for **Li**nux **K**ernel **I**nstrumentation) was developed to replace *ftrace* as the primary tracing mechanism, and on 03/25/2013 LinuxKI was born!

1.3 Supported Linux platforms

In order to use LinuxKI on a specific Linux platform, there are some requirements:

- Supported kernels are based of 2.6.32 through 5.9.16 kernels
- Kernel is configured with CONFIG_FTRACE=y and CONFIG_FTRACE_SYSCALLS=y
- Supported on x86_64 and arm64 systems

Example distributions tested and supported include:

- RHEL
- CentOS
- OEL
- Ubuntu
- Fedora
- Debian
- SLES

Note that RHEL 5 is not supported as it uses a 2.6.18 kernel and the *ftrace* tracing code did not exist. Also, SLES 11 SP1 is not supported as it was compiled with CONFIG_FTRACE=n. Also, be careful to check the Linux kernel versions. For example, the system may be running OEL 6, but the kernel version may be a 2.6.18 kernel. Also be careful regarding some upstream kernels that may not be tested/supported and custom kernels. For example, some custom kernels will not be compiled with CONFIG_FTRACE=y. LinuxKI also will not work with stripped down kernels, like the kernel used for VMware hosts or some embedded systems. And don't try to run it on your Android phone either!

NOTE
While LinuxKI has worked successfully on ARM system, there is currently not enough resource to properly compile, test, and package for ARM-based servers. However, a version 5.3 kiinfo.aarch64 binary is available with the RPM and the sources are available on the LinuxKI GitHub page.

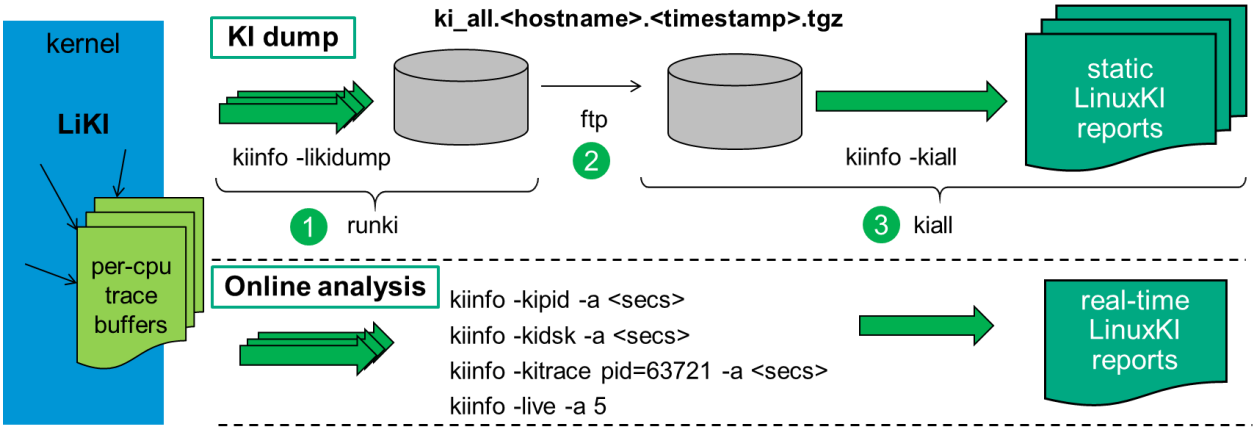
1.4 LinuxKI design

The following files are the basic components of LinuxKI:

runki	Script to capture a LinuxKI dump from a customer system. By default, a 20 second LinuxKI trace dump is collected. The script creates the ki_all.<hostname>.<timestamp>.tgz compressed tar bundle that can be transferred to HPE for analysis.
kiinfo	Binary executable which reads the binary trace data from the kernel ring buffers and also analyzes the binary trace data and generates various reports based on the trace data. It is executed by the ***runki*** script to collect the binary trace data and also used by the ***kiall*** script generate the LinuxKI reports.
likit.ko	The LiKI DLKM tracing module. It generates LiKI kernel trace data which is read by the ***kiinfo*** executable. The ***runki*** script will load the LiKI module automatically. The ***kiinfo*** executable will also load the LiKI module automatically when doing online analysis.
kiall	Script to perform post-processing of LinuxKI dump. Expand the ki_all*.tgz compressed tar bundle and executes ***kiinfo*** to generate a number of default LinuxKI reports (kparse, kipid, kidsk, kirunq, etc...)
kiclean	Script used to clean up the LinuxKI trace data for archiving purposes. This script will preserve the ki_all.*.tgz file, as well as the LinuxKI reports.

There are 2 primary methods to collect and analyze the trace data:

1. Generate LinuxKI dump using the ***runki*** script and analyze later or on another system
2. Stream LiKI data from live system and analyze it real-time with ***kiinfo***



The LinuxKI dump can also be collected using *ftrace* using the *-f* option with the ***runki*** script:

```
$ runki -f
```

However, the online analysis can only be performed using the LiKI tracing mechanism.

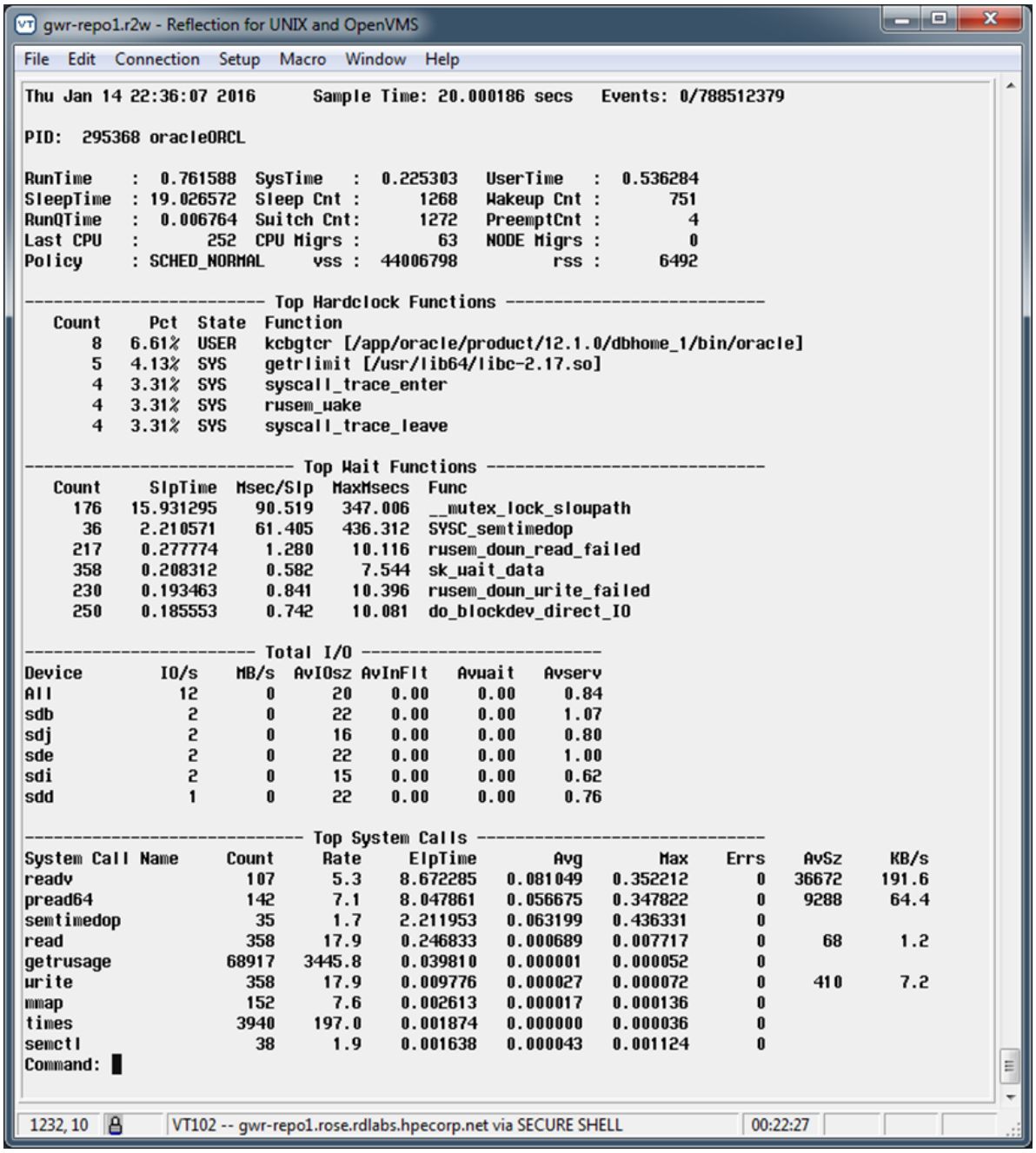
At its simplest form, a 20-second LinuxKI dump using the *LiKI* tracing module can be collected and analyzed as follows:

1. ***runki***
2. FTP to HPE system (optional)
3. ***kiall***

NOTE
The ***runki*** script must be executed when the performance problem exists or during an interested period of execution. Executing the ***runki*** script on an otherwise idle system is not valuable.

1.5 Sample Data Analysis

Below is sample data using the curses-based ***kiinfo*** invocation from a LinuxKI dump collected with the ***runki*** script. In this example, a specific task or PID is being analyzed. The detailed report shows how much time the task spent running on the CPU, waiting for the CPU, or waiting for other reasons. It also shows the CPU function profile, the task wait events, the IO generated by the tasks, and the system calls made by the task. All of the data is derived from the detailed LinuxKI trace records collected by the *LiKI* tracing mechanism.



1.6 ftrace vs. LiKI

LinuxKI is able to collect data using the *ftrace* tracing mechanism or the HPE-provided *LiKI* tracing mechanism. The *LiKI* tracing mechanism is the preferred and default method. The tracing methods are described in more detail below.

1.6.1 LiKI

LiKI is an open-source Dynamically Loadable Kernel Module (DLKM) developed by HPE. The actual filename for the DLKM is likit.ko. When executing the ***runki*** script or the ***kiinfo*** executable, the *LiKI* DLKM is inserted into the kernel and inserts lightweight tracepoints into the kernel similar to *ftrace*. However, the *LiKI* tracing mechanism adds additional key information that *ftrace* does not capture. Below are some of the features provided by LiKI tracing mechanism that is not available to *ftrace*:

- CPU profiling (hardclock trace events), including user and kernel stack traces.
- User and Kernel stack traces on sched_switch and sched_migrate_task events.
- Filtering during trace event collection
- Per-CPU sequence counters to detect log trace records.
- More accurate and efficient timestamps.
- Special system call fields, including:
  - Filenames on open(), stat() system call events.
  - Executable names on exec() and execve() system calls.
  - Inode and device on file related system call entry events.
  - Network IP addresses on network related system call records.
  - io_submit() and io_getevents() IO control blocks
- Async and sync IO counters on block device requests
- TGID logged for each trace event
- And much more...

The *LiKI* tracing mechanism is the preferred and default data collection method.

1.6.2 ftrace

Ftrace was compiled into most Linux kernels starting in 2.6.32. *Ftrace* was the default tracing mechanism in LinuxKI versions prior to 3.0, but it has been replaced as the default tracing mechanism by the *LiKI* tracing mechanism. However, if the *LiKI* DLKM fails to compile or fails to install, the *ftrace* tracing mechanism will be used. The *ftrace* tracing mechanism can also be used by specifying the "-f" option with ***runki***:

```
$ runki -f
```

2 Getting Started

This section focus on downloading and installing LinuxKI.

2.1 LinuxKI Prerequisites

LinuxKI does not have any mandatory pre-requisites or dependencies for installation. The toolset should install and run using the *ftrace* tracing mechanism on any system which supports *ftrace*. However, the *ftrace* tracing mode has some limited capabilities, and the preferred tracing mechanism is the *LiKI* tracing mechanism provided as a dynamically linked executable.

A small set of pre-compiled *LiKI* DLKM modules are provided in the Linux KI Toolset, including some DLKM modules for RHEL and RHEL-compatible kernels. However, in many cases, such as Ubuntu, Debian, and SLES, the *LiKI* DLKM must be created by compiling from the *LiKI* source code. To compile the *LiKI* source code into a DLKM file (likit.ko), the Linux kernel headers must be installed, as well as the gcc compiler and make. Please refer to your specific Linux distribution for information on installing the Linux Kernel headers, gcc, and make. Below is a general guide of the kernel header packages you will need to install in order to compile the *LiKI* source code:

RHEL:
kernel-devel-`uname -r`

UEK:
kernel-uek-devel-`uname -r`

SLES:
kernel-source-`uname -r`
kernel-default-devel-`uname -r`

Debian / Ubuntu:
linux-headers-`uname -r`

Recommendation
As a general rule, it is best to go ahead and install the kernel headers, gcc, and make prior to installing LinuxKI.

NOTE
Some Linux versions also require the *elfutils-libelf-devel* package to be installed as well.

2.2 Downloading LinuxKI

LinuxKI is provided as an RPM package for most distributions, and as a DEB package for Ubuntu and Debian distributions. The RPM/DEB packages include all the scripts and binaries to collect and analyze the LinuxKI data.

You can download the latest version of LinuxKI the following github page:

https://github.com/HewlettPackard/LinuxKI

2.3 Installing LinuxKI

After downloading LinuxKI (linuxki*.rpm / linuxki*.deb), you can use the following instructions to install the toolset. You need superuser or sysadmin privileges to install the rpm or deb packages.

Note
You can also use your favorite package manager such as yum to manage the packages.

2.3.1 Prerequisites

Be sure to install the kernel headers mentioned in the prerequisites. If the installation scripts attempt to build the *LiKI* DLKM from source code and the kernel headers are not present, a message similar to the following will occur as the kernel build directory will not exist.

```
make: *** /lib/modules/3.14.29-1-amd64 /build: No such file or directory.  Stop.
make: *** [build] Error 2
```

If you get the above error, you may still capture LinuxKI trace data using *ftrace* instead of *LiKI*.

2.3.2 Install latest RPM

If LinuxKI is not already installed, the rpm can be installed on most systems as follows:

```
$ rpm --install --nodeps linuxki-7.1-1.noarch.rpm
```

2.3.3 Upgrading LinuxKI to a newer version

If you have previously installed LinuxKI and wish to install a newer version, you can simply perform an upgrade using a later LinuxKI rpm:

```
$ rpm --upgrade –nodeps linuxki-7.1-1.noarch.rpm
```

2.3.4 Verify LinuxKI version installed

After installing or upgrading LinuxKI, verify that it is installed as follows:

```
$ rpm --query linuxki
linuxki-7.1-1.noarch
```

2.3.5 Removing LinuxKI

You can remove the linuxki package as follows:

```
$ rpm --erase linuxki
```

2.3.6 Special instructions for Debian-related distributions

The LinuxKI packages for Debian-related distributions use .deb packages. These packages are installed and managed using the dpkg utility:

```
$ dpkg --status linuxki | grep "Version"
$ dpkg --remove linuxki
$ dpkg --purge linuxki
$ dpkg --install linuxki_7.1-1_all.deb
```

2.3.7 Missing libtinfo.so on some Linux distributions

In some cases the following or similar error will result if the proper libtinfo.so cannot be found:

```
kiinfo: error while loading shared libraries: libtinfo.so.5: cannot open shared object
file: No such file or directory
```

If you encounter this error, be sure to link the missing libtinfo.so file to the corresponding libncurses.so file in either /lib64 or /usr/lib64. For example:

```
$ ln -s /lib64/libncurses.so /lib64/libtinfo.so.5
```

The filenames above may vary depending on the versions of the software installed.

2.3.8 Re-compiling the LiKI DLKM module

The LiKI DLKM module may need to be re-compiled due to one of the following reasons below:

- The Linux kernel has been upgraded to a new version
- The LinuxKI Toolset failed to compile the LiKI DLKM module at installation time, perhaps due to missing pre-requisites.
- The previous LiKI DLKM module was mistakenly removed
- The LiKI source code was manually modified in the /opt/linuxki/src directory

It is not necessary to remove the LinuxKI Toolset and re-install for each case. Instead, the LiKI DLKM can be rebuilt using the following commands:

```
$ rm /opt/linuxki/modules/likit.ko*
$ /opt/linuxki/module_prep
```

2.3.9 Compiling the kiinfo binary

LinuxKI comes with serval pre-compiled kiinfo binaries. When LinuxKI is installed, a kiinfo link will be created to link to the appropriate binary:

```
/opt/linuxki> ll kiinfo*
-rwxr-xr-x. 2 root root 3384904 Sep 11 13:16 kiinfo
-rwxr-xr-x. 1 root root 2546824 Sep  4 13:51 kiinfo.aarch64
-rwxr-xr-x. 1 root root 3352832 Sep  4 13:51 kiinfo.ppc64le
-rwxr-xr-x. 2 root root 3384904 Sep 11 13:16 kiinfo.rhel8
-rwxr-xr-x. 1 root root  889208 Sep  4 13:51 kiinfo.sles15
-rwxr-xr-x. 1 root root  886152 Sep  4 13:51 kiinfo.x86_64
```

However, there may be times when the Linux version you are using doesn't contain the same libraries as the compiled version and it may be necessary to compile the kiinfo binary from source code.

In order to compile kiinfo from source code, you will need the following dependencies:

- gcc
- make
- ncurses-devel
- libpthread

Then simply execute the make(1) command from the /opt/linuxki/src/kiinfo directory and copy the kiinfo binary to /opt/linuxki:

```
$ cd /opt/linuxki/src/kiinfo
$ make
gcc -g -O2 -lc -lrt -Wformat=0 -o kiinfo *.c -D__LIKI_RTMERGE -D_GNU_SOURCE -lpthread
-lncurses
$ cp kiinfo /opt/linuxki
```

3 LinuxKI Dumps

The simplest method to collect LinuxKI trace data is to perform a LinuxKI dump by executing the ***runki*** script.

3.1 runki - collecting trace data

In order to execute the ***runki*** script, you will need the following:

- You must have root permission
- You must be logging into a filesystem with plenty of disk space (recommended 500 MB - 3 GB) and it should be mounted for cached I/O. The sizes of the trace files vary depending on the number of CPUs and the amount of activity on the system.
- If you have sufficient memory, you can use an in-memory filesystem like /dev/shm to collect the data. ***This is the preferred method!***
- If you use a filesystem for collecting data, consider creating a new filesystem just for the LinuxKI data with journaling disabled to avoid spinlock contention in the jbd2 code when writing the LinuxKI data to the filesystem:

```
$ tune2fs -H ^has_journal /dev/<mnt_device>.
```

At its simplest, LinuxKI trace data is collected using the ***runki*** script, which will create a single compressed tar file. There are two modes, depending on whether you want to collect data using *ftrace* or the *LiKI* kernel module (likit.ko):

```
$ /opt/linuxki/runki       # Use kiinfo -likidump to enable LiKI and collect the
                             trace data

$ /opt/linuxki/runki -f    # use kiinfo -kitracedump to enable ftrace and collect
                             the trace data
```

If there is sufficient available memory on the system, you should collect the data in memory using /dev/shm first and then copy the ki_all.*.tgz file to a disk filesystem afterwards. This reduces the risk of missing LinuxKI events during the tracing and does not add to the disk workload during the collection as well.

```
$ cd /dev/shm
$ /opt/linuxki/runki
```

Recommendation
For best results, use /dev/shm to collect the LinuxKI dump if there is sufficient memory.

The ***runki*** script will do all the work. It will make sure the debug filesystem is mounted, insert the *LiKI* DLKM if needed, enable the trace points and dump the trace data to disk. It will also collect supplemental information which can be used to interpret the trace data and help troubleshoot performance problems. The result will be a single compress tar file of the format ki_all.<hostname>.<timestamp>.tgz, where the timestamp will be of the format MMDD_HHMM. The amount of trace data depends on the number of CPU and the rate at which the trace events occur. The

data should be collected in a filesystem with plenty of disk space (200 MB - 5 GB) which is mounted for cached I/O. Note that root privilege will be needed to execute the ***runki*** script. Below is an example using the LiKI tracing:

```
$ /opt/linuxki/runki
=== runki for Linux version 7.1
=== Installing LinuxKI trace points ===
=== Starting: LinuxKI liKI trace dump for 20 seconds ===
===
Command line: /opt/linuxki/kiinfo -likidump
dur=20,events=default,subsys=default,pid=0,tgid=0,cpu=-1,dev=-
1,debug_dir=/sys/kernel/debug,sysignore=ignore_file -ts 0209_1243

kiinfo (7.1)

kiinfo: spooling trace data to disk...
kiinfo: Tracing complete
=== LinuxKI liKI dump complete  ===
=== KI trace complete.  Collecting supplemental files.
=== Running lsof ...
=== Collecting task stack traces from /proc ...
=== Collecting task maps from /proc ...
=== Collecting task numa_maps from /proc ...
=== Collecting binary executable and library symbol table information ...
=== Getting Processor C-state information ...
=== Getting disk information ...
=== Building LVM mapping table ...
=== Collecting volume data ...
=== Gathering misc supporting data ...
=== Copying the syslog files ...
=== Tarring up the results ...
=== Trace completed and archived as ki_all.$HOSTNAME.$tag.tgz"
```

If the *LiKI* DLKM (likit.ko) is not available, the ***runki*** script will use the *ftrace* tracing mechanism. You can also use the "-f" option on the ***runki*** command to initiate an *ftrace* collection. However, the *ftrace* collection has less functionality since it does not log any kernel function names or stack traces on switch records and does not perform any CPU profiling.

WARNING

Be *sure to obtain the customer's permission before running the toolset in production. Any defect in the Linux ftrace code or the LiKI trace module could cause a system failure and customers should be aware of the risks of running the toolset.*

The ***runki*** script has the following syntax and options:

```
runki [-h] [-L] [-M] [-U] [-X] [-a] [-v] [-f] [-p] [-j [-J <path>]] [-n interface]
[-d duration] [-t maxrun] [-P pid] [-G tgid] [-C cpu] [-D dev] [-F] [-I <sysignore>]
[-e event] [-s subsys] [-T <timestamp>} [-c "comment"]

    -h              Help
    -L              Gather minimal data for local analysis...no gzip/tar
    -M              Include Collectl/MeasureWare data collection
    -U              Include userspace profile (perf) data collection
    -X              Include sar data collection
```

```
-a              Execute 'perf annotate' on userspace profile (perf) data
-j              Collect Java stacks
-J <path>       Location of Java jstack command
-n <interface>  Gather tcpdump trace data
-v              Skip vxfs, misc disc data collection
-d <secs>       Duration of KI data collection. Defaults to 20 secs if not
                specified.
-t <secs>       Maximum time for data collection tools to run. Defaults to 120 secs
                if not specified.
-p              Skip Per-PID data (lsof, stacks, numa_maps, maps)
-f              Use ftrace tracing instead of liKI DLKM tracing to dump trace data.
-P <pid>        Filter collection on Task ID or PID (LiKI only)
-G <tgid>       Filter collection on Task Group ID or TGID (LiKI only)
-C <cpu>        Filter collection on cpu (LiKI only)
-D <dev>        Filter collection on device (LiKI only)
-R              Advance CPU stats (Freq, CPI, LLC Hit%) using MSR registers (LiKI
                only)
-e <event>      Identify events to dump. Possible values are:
                default | all | <event>
-s <subsys>     Identify subsys to dump. For example: irq, scsi, block, etc.
-I <sysignore>  File to specify ignored system call (liki only)
-T <timestamp>  Should be of the form MMDD_HHMM
-V "<varargs>"  Specify multiple filters and flags to pass to kiinfo -likidump
-c "comment"    Echo comment into comment.$tag file
```

WARNING
Be careful if collecting data for more than 20 seconds. The longer the duration of the trace (-d option), the larger the resulting trace dump. A 60 second trace dump will be ~3x larger than the default 20 second trace dump. If the trace is large enough, it could exhaust space in the current filesystem, or it could exhaust memory if collecting data in /dev/shm.

3.2 kiall - generating LinuxKI reports (post-processing)

Once the ki_all.*.tgz file is collected, the most useful LinuxKI reports can be generated with the ***kiall*** script, also called post-processing. Note that the post-processing of the LinuxKI dump data can be done on a different system where the data was collected. By default, the ***kiall*** script will explode the tar bundle in the current working directory and execute the ***kiinfo*** tool to generate the various LinuxKI reports:

```
$ /opt/linuxki/kiall
```

The ***kiall*** script can handle multiple ki_all.*.tgz files from multiple hosts as it will search the current working directory for all of the ki_all.*.tgz files. Also, the -r option can be used to create a <hostname>/<timestamp> subdirectory structure which can be used to organize multiple trace collections from different hosts:

```
$ ls ki_all*
ki_all.node1.0828_1429.tgz ki_all.node1.0828_1448.tgz
$ kiall -r
Running KI tools for 0828_1429 for host node1
...
Running KI tools for 0828_1448 for host node1
...
```

After executing ***kiall*** above, the LinuxKI files and generated reports will be stored in a subdirectory of the form <hostname>/<timestamp>. For the example above, it will create 2 subdirectories:

```
node1/0828_1429
node2/0828_1448
```

The -f option for ***kiall*** can be used to search for LinuxKI data in directory tree to be processed. This option is useful for re-processing a number of LinuxKI trace collections without having to change the directories and execute ***kiall*** for each LinuxKI trace collection.

```
/opt/linuxki/kiall -f
```

kiall usage and syntax is listed below:

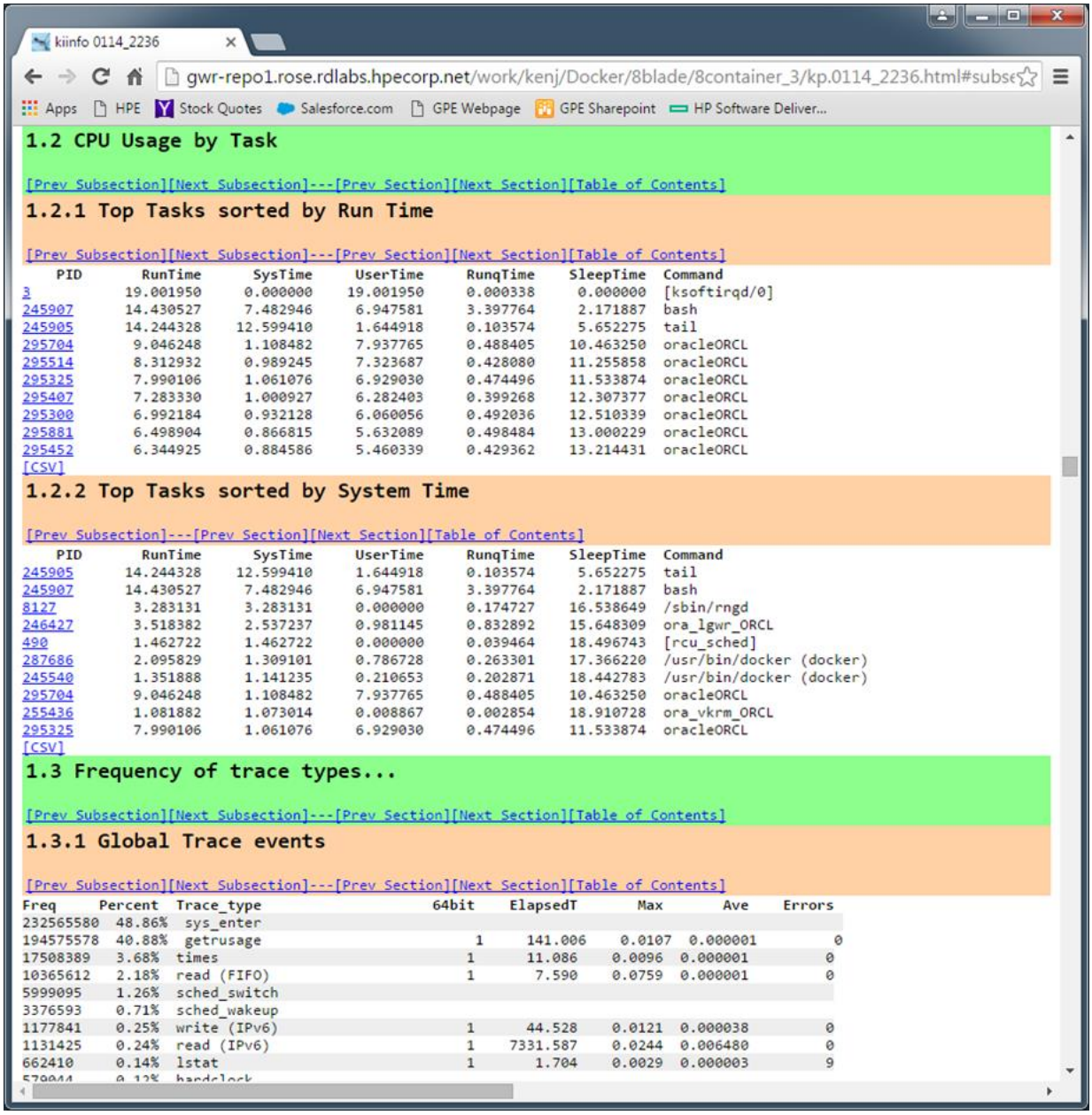
```
kiall [-l] [-m] [-r] [-f] [-c] [-x] [-M] [-B] [-V] [-P [num_threads] [-t timestamp]
    -h                Help
    -l                lite collection
    -m                Skip MW processing
    -r                restore into created subdir - <host>/<MMDD_HHMM>
    -f                Find KI files in pwd and any directories below
    -c                Cluster-wide processing
    -x                Generate kparse text format instead of html format
    -M                Leave C++ function names mangled
    -B                Add Disk Block Frequency stats to Kparse report
    -V                Use Visualization options where possible in reports
    -P [num_threads] Use multithreaded processing when generating Visualization
charts/graphs
    -t <timestamp>    Only runs KI reports bundle with matching timestamp
```

3.3 LinuxKI reports

The ***kiall*** script will generate the following reports automatically:

File	Command	Description	Comments
kparse.*.html	kiinfo -kparse	Kparse Report - System overview of LinuxKI data	
kipid.*.txt	kiinfo -kipid	PID Analysis Report	
kipid.oracle.*.txt	kiinfo -kipid oracle	PID Analysis Report (Oracle-specific)	
kidsk*.txt	kiinfo -kidsk	Disk Analysis Report	
kirunq.*.txt	kiinfo -kirunq	CPU and Runq Analysis Report	
kiprof.*.txt	kiinfo -kiprof	CPU Profile Report	*LiKI* traces only
kiwait.*.txt	kiinfo -kiwait	Wait Event Analysis Report	*LiKI* traces only
kifile.*.txt	kiinfo -kifile	File Activity Report	
kifutex.*.txt	kiinfo -kifutex	FUTEX Activity Report	
kisock.*.txt	kiinfo -kisock	Network Socket Activity Report	
kidock.*.txt	kiinfo -kidock	Docker Activity Report	*new with 5.0*
All of the above	kiinfo -kiall	Generate all of the above reports in a single pass	
ki.<timestamp>	kiinfo -kitrace	Format LinuxKI data as individual trace records	
cp.*.html	kiinfo -clparse	Cluster Overview Report (clparse)	

After executing the ***kiall*** script, you can use your favorite text editor (vi, vim, etc.) to view the *.txt files. If your files are accessible through a web browser, the Kparse file (kp.*.html) will provide a good overview of the data with the hypertext navigation of a browser. For example:



3.4 kiinfo – customizing reports

While the post-processing reports generated by the ***kiall*** script are very helpful, sometimes it is necessary to customize the output of one or more of the reports. You can generate custom reports using the ***kiinfo*** executable, which has a wide variety of subtools and options.

For example, if you want to see the top 20 tasks accessing a specific disk device, you can do the following:

```
$ kiinfo -kidsk dev=0x00800010,npid=20 -ts 0815_2118
```

Or perhaps you want to see all the activity from a specific task between the 4th and 5th second of the LinuxKI dump:

```
$ kiinfo -kipid pid=27653 -ts 0815_2118 -start 4.0 -end 4.0
```

There are many flags and options available to customize the reports. Be sure to review the ***kiinfo*** reference pages in the LinuxKI reports and kiinfo reference pages section.

3.5 Filtering

Sometimes, it is important to collect data over a long period of time. However, the size of the trace data can grow abnormally large over a longer trace period. Prior to LinuxKI version 4.0, all filtering was done within ***kiinfo***, which meant that all the trace records were collected by *LiKI* and then filtered later with ***kiinfo***. LinuxKI has several features to focus the trace collection on specific criteria to reduce the amount of trace data collected.

3.5.1 Filtering with the LiKI DLKM module

LinuxKI has the ability to filter the data as it is collected if the *LiKI* tracing mechanism is used. For example, if filtering on a specific PID, only the traces associated with the specific PID are captured. This greatly reduces the amount of trace data collected when only a subset of tasks needs to be analyzed. There are some traces records (such as block_rq_issue and block_rq_complete records) that are always logged, but the goal is to reduce the number of trace records logged and passed to ***kiinfo*** or written to disk.

Trace records can be filtered on task (PID), task group id (TGID), CPU, or disk device. Below is an example of using filtering during the data collection using the ***runki*** script so that only trace records matching PID 1234 or TGID 6350 will be logged in the LinuxKI binary trace file:

```
$ runki -P 1234 -G 62350
```

Filtering with *LiKI* is also performed when running ***kiinfo*** in live mode. The following will generate a PID Analysis Report (***kiinfo -kipid***) for PID 7356 and PID 16200 over a 15 second interval for 4 passes (total of 60 seconds):

```
$ kiinfo -kipid pid=7356,pid=16200 -a 15 -p 4
```

Note that filtering when running ***kiinfo*** in live mode will filter within the *LiKI* DLKM and within ***kiinfo***. So any extra trace records like block_rq_complete that makes it though the *LiKI* DLKM will be filtered out by ***kiinfo***.

Note that some filters do not apply to all ***kiinfo*** subtools. For example, it doesn't make sense to filter on a disk device when executing ***kiinfo -kiprof***, since ***kiinfo -kiprof*** only collects the hardclock trace records and not the block trace records.

3.5.2 Ignoring selected system calls

In some applications, certain system calls are performed very frequently. Below is part of a Kparse report which shows the most frequent trace records:

```
Freq      Percent  Trace_type          64bit  ElapsedT        Max        Ave  Errors
35927039   46.65%  sys_enter
32869289   42.68%  getrusage               1    38.394     0.0542   0.000001       0
2520180     3.27%  times                   1     2.473     0.0408   0.000001       0
354287      0.46%  sched_switch
248342      0.32%  sched_wakeup
182170      0.24%  write (IPv6)            1    16.080     0.0792   0.000088       0
182133      0.24%  read (IPv6)             1   376.977     0.0830   0.002070       0
```

Note that over 90% of all the traces were used for getrusage() and times() system call entry and exit records. These system calls do not impact the flow of the application (simply takes a little CPU time). These system calls can be ignored by creating a file which contains the system calls that you do not want traced:

```
$ cat scall_ignore
getrusage
time

$ runki -I scall_ignore
```

You can also ignore the system calls if you are using ***kiinfo*** in live mode:

```
$ kiinfo -kipid pid=15216,sysignore=scall_ignore
```

NOTE
The ***sysignore=*** flag can only be used when the trace data is collected. It has no effect when used with an existing trace dump collected using the runki script.

3.5.3 runki -V option

The -V option is used to add multiple filters for the runki command. Now, whatever options you can pass to likidump, you can pass through the runki script. For example:

```
$ runki -V "pid=7648,pid=28745,subsys=irq,subsys=block"
```

Note that many of the runki options can now be done with the -V option. For example, the following runki command:

```
$ runki -P 15216 -I scall_ignore -e all
```

is equivalent to:

```
$ runki -V "pid=15216,events=all,sysignore=scall_ignore"
```

For more information on what you can pass using the -V option, please refer to kiinfo -likidump.

3.6 kiclean - cleaning up after analysis

When you have completed your analysis on LinuxKI data, you can cleanup a subdirectory using the ***kiclean*** script. This script will keep the ki_all.*.tgz file as well as the generated reports, but will remove the LinuxKI binary traces files and other related data (this data is saved in the ki_all.*.tgz file). If the -f option is used, other miscellaneous files created during the analysis are also removed. Below is an example:

```
$ cd node1/0828_1429

$ kiclean -f
Cleaning /work/mcr/node1/0828_1429
kiclean complete

$ ls
ki_all.node1.0828_1429.tgz       kifile.0828_1429.txt      kiprof.0828_1429.txt
kiwait.0828_1429.txt             PIDS                      kidsk.0828_1429.txt
kipid.0828_1429.txt              kirunq.0828_1429.txt      kp.0828_1429.html
```

The -p option will remove the PIDS subdirectory as well as the other files as this can take a lot of additional space. The -r option is useful when cleaning up multiple LinuxKI trace collections that reside somewhere under the current working directory. For example:

```
$ pwd
/work/mcr/cust1
$ ls
1211_1335 1211_1435

$ kiclean -r -f -p
Cleaning /work/mcr/cust1/1211_1335
Cleaning /work/mcr/cust1/1211_1435
```

kiclean usage and syntax is listed below:

```
kiclean [-h] [-r] [-f] [-p] [-v]
            -h          Help
            -r          Recursively traverse subdirs looking for KI data to archive
            -p          Remove PIDS and CIDS subdirectories
            -v          Remove VIS subdirectory & related sh/php/html files
            -f          Force remove misc/tmp files
```

4 Online LinuxKI Analysis

There are two types of online LinuxKI analysis. You can generate the same types of text based reports that you can generate using a LinuxKI dump, or you can use the curses-based user interface introduced with LinuxKI version 4.1 described in Section 5.

Note
The following are requirements needed for performing Online LinuxKI analysis:

- the *LiKI* DLKM tracing module must be available
- you must have root access
- the debug filesystem must already be mounted:

```
$ mount -t debugfs debugfs /sys/kernel/debug
```

Typically, LinuxKI data is collected using the ***runki*** script to log data to LinuxKI binary files over a short period of time (default 20 seconds). However, the ***kiinfo*** tool can enable the *LiKI* tracing and read the LinuxKI binary trace data straight from the *LiKI* ring buffers to perform live analysis. Live tracing with ***kiinfo*** can be performed over longer intervals as the LinuxKI trace data is analyzed in real-time. The detailed LinuxKI trace data is not retained, only the summary reports. Live LinuxKI tracing is initiated using the alarm option ("-a") and the passes option ("-p") for ***kiinfo***. For example, suppose you want to monitor the disks looking for intermittently high IO service times. You could execute the ***runki*** script and hope you get lucky, or you could use ***kiinfo -kidsk*** to generate a report every 30 seconds for a one hour period.

```
$ kiinfo -kidsk -a 30 -p 120 >kidsk.txt
```

When possible, try to use filters to limit the amount of data being analyzed. Using filters will allow ***kiinfo*** to reduce its overhead and CPU usage. For example, if you are concerned about a single process, or a small set of processes, you can setup a filter to limit the data collected to only those processes. For example:

```
$ kiinfo -kipid pid=8272,pid=8274,pid=8276,npid=10,rqhist -a 60 -p 10 >kipid.txt
```

You can also use the Curses-based user interface with live tracing as discussed in the next section. For example:

```
$ kiinfo -live
```

Some tools and options, such as ***kiinfo -kipid coop*** are not allowed to be executed for live analysis due to the amount of data tracked by the tasks. Other tools like kiwait, kiprof, kirunq, kifile, kitrace, and kipid will all work with the live tracing (-a and -p options).

Also, for live LinuxKI tracing to work well, the ***kiinfo*** process does need CPU resources to collect and analyze the LinuxKI data. If the system is under severe CPU pressure, LinuxKI events could easily be missed which can distort the data collected.

5 Curses-based LinuxKI analysis (kiinfo –live)

The curses-based user interface was introduced with LinuxKI version 4.1 with the ***kiinfo -live*** option. For those familiar with running Glance on HP-UX, this will be very familiar. The curses-based UI can be used to analyze data in real-time from a running system, or from a LinuxKI dump collected with the ***runki*** script. To initiate ***kiinfo -live*** on a running system, simply use the following command:

```
$ kiinfo -live
```

The default refresh interval is 5 seconds. You can change the interval time using the **-a** option. The example below will refresh the screen every 10 seconds.

```
$ kiinfo -live -a 10
```

To initiate ***kiinfo -live*** on a LinuxKI dump collected from ***runki***, simply specify the timestamp:

```
$ kiinfo -live -ts 0223_1536
```

Recommendation
For best results when using the curses-based UI, expand your terminal screen. It works best with 160 columns or more and 48 rows or more. ***Kiinfo*** will also dynamically adjust the output based on the size of the terminal screen.

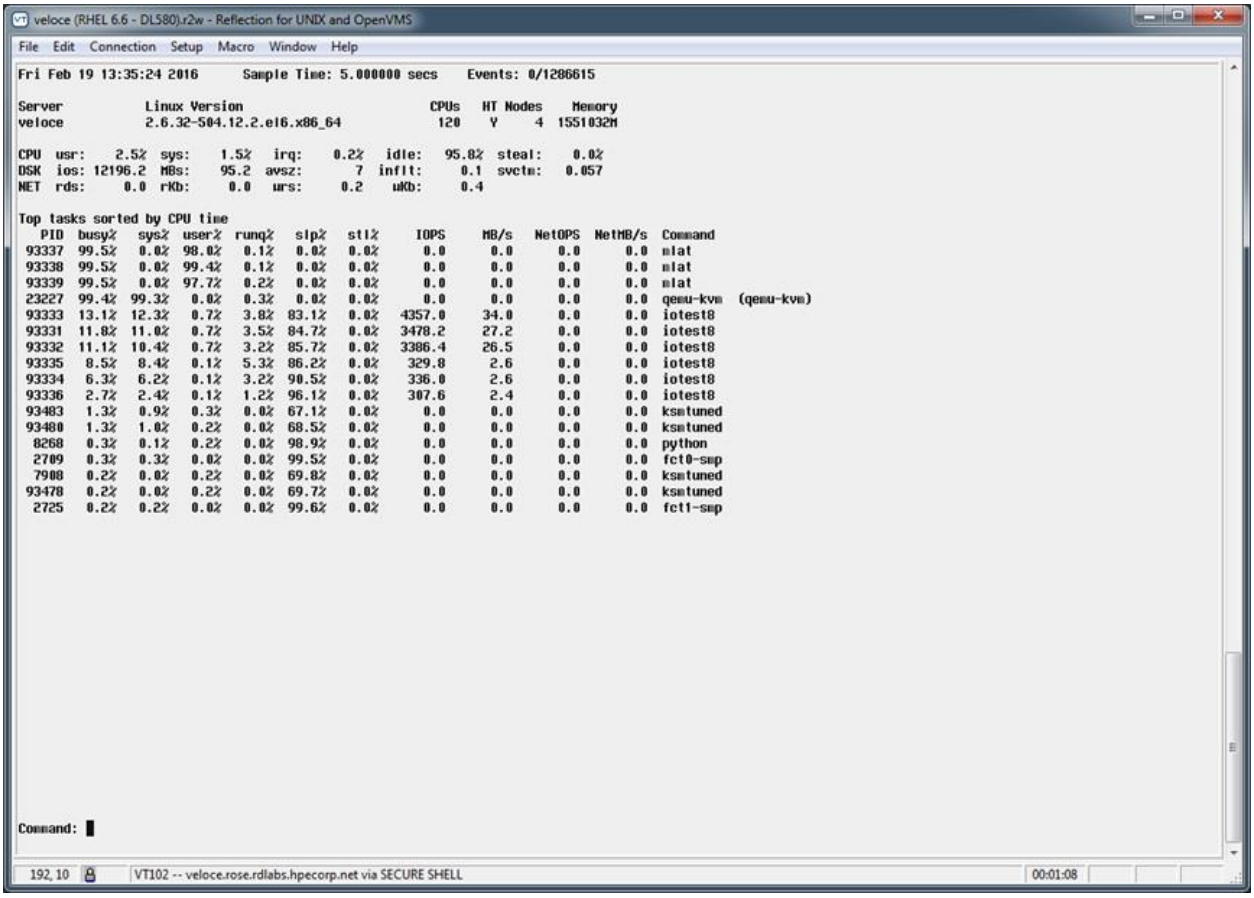
Below is a list of commands that can be used with ***kiinfo -live***:

```
                                Commands Menu

s - Select Task/CPU/Disk  G - Task Main Stats       ? - Help
g - Global Task List      D - Task Disk Stats       r - Refresh
l - Global Node Stats     M - Task Mpath Stats      b - Prev Screen
c - Global CPU Stats      L - Task System Calls     + - Show Syscall Detail
p - Global Prof Stats     W - Task Wait Stats       - - Hide Syscall Detail
h - Global HT CPU Stats   P - Task Profile Stats    > - Next Step
i - Global IRQ Stats      F - Task File Stats       < - Prev Step
d - Global Disk Stats     O - Task Coop Stats       j - Step Time
m - Global Mpath Stats    U - Task Futex Stats      J - Jump to Time
y - Global WWN Stats
z - Global HBA Stats      C - Select CPU Stats
t - Global IO by PID      T - Select Disk Stats
f - Global File Stats     I - Select IRQ Stats
w - Global Wait Stats     K - Select Docker Stats
u - Global Futex Stats    X - Select Futex Stats
n - Global Socket Stats
k - Global Docker Stats
```

The Global Docker Stats, Select Docker Stats, and Select IRQ stats were added in LinuxKI version 5.0. The Global WWN Stats were added in LinuxKI version 5.6.

The Global Task List screen ('g') will be displayed when executing ***kiinfo -live***:

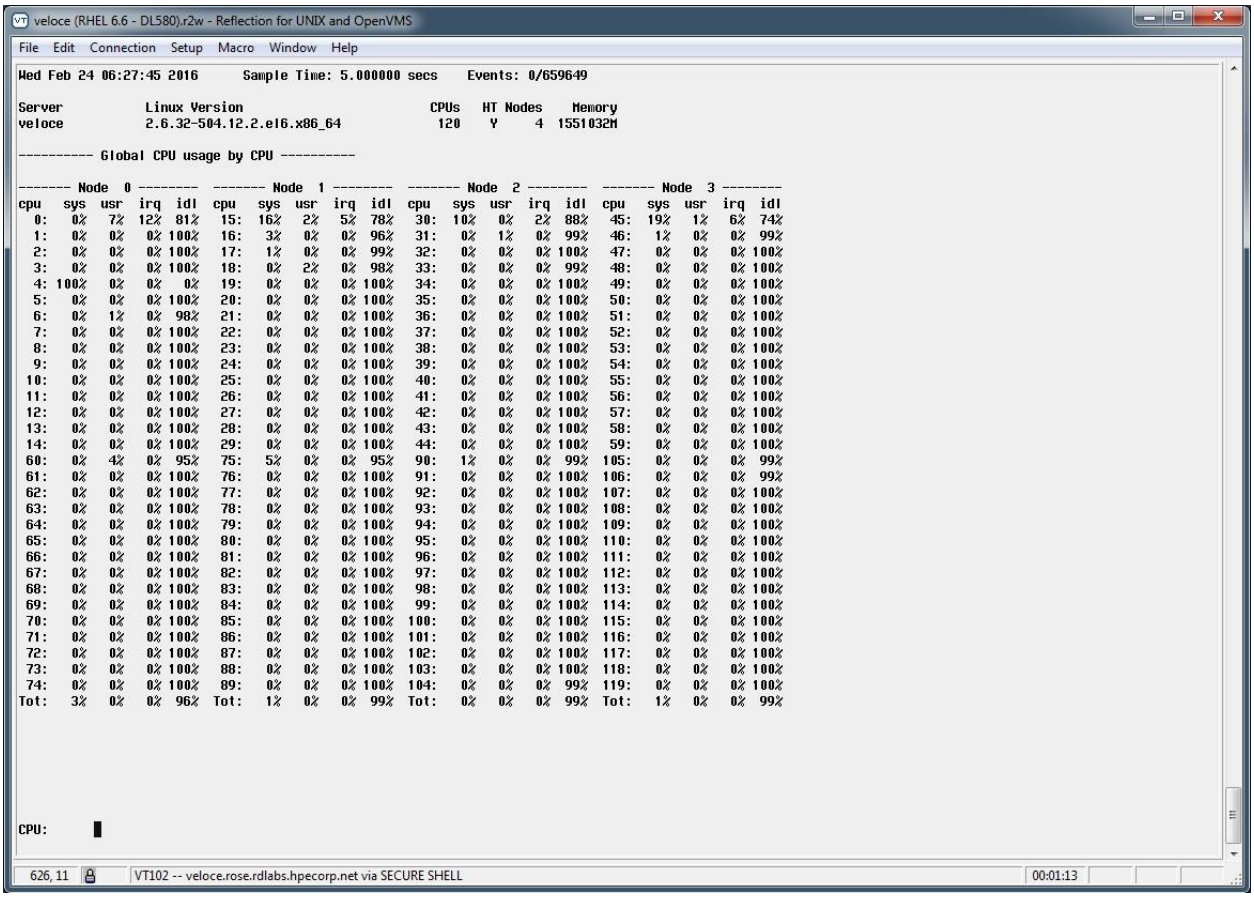


From the command prompt at the bottom of the screen, you can select the specific statistics screen you wish to view. Note that global statistics usually use a lower case letter, while task statistics use an uppercase letter.

5.1.1 Selecting a Task/Disk/CPU/LDOM/Futex/Docker Container

Use the 's' command to select one of the items displayed on the screen. The object may be a task, disk, cpu, ldom, or a futex, depending on the data on the current screen. For example, if you are looking at the global disk statistics and use the Select command ('s'), then you will be prompted for a disk device. If you are looking at global CPU stats and use the Select command ('s'), then you will be prompted for a CPU. If you are on the Global Task List screen and use the Select command ('s'), then you will be prompted for a specific task. If you have already selected a task and then you want to look at one of the task statistic screens, such as the Task Wait Stats, then it will use the task you have already selected.

The example below shows that after using the Select command ('s') on the Global CPU Stats screen. Note the user is prompted for a specific CPU to analyze:



5.1.2 Expanding syscall statistics

Some screens will show a summary of each system call. For example the Task screen when selecting a task will show the system calls (provided the screen size is large enough):

```
--------------------------- Top System Calls ----------------------------
SysCall Name  Count     Rate     ElpTime        Avg         Max     Errs     AvSz      KB/s
lseek           540    108.0    0.954095   0.001767    0.205005        0
write           540    108.0    0.850707   0.001575    0.054176        0     8192     864.0
```

You can get more details on the system call provided there are enough rows on the screen using the Show Syscall Detail command ('+'):

```
---------------------------- Top System Calls -----------------------------
System Call Name     Count     Rate     ElpTime        Avg         Max     Errs     AvSz
KB/s
write                 1970    394.0    2.759057   0.001401    0.039001        0     8192
3152.0
   SLEEP              4171    834.2    2.649811   0.000635
      Sleep Func      2202             2.432773   0.001105   __mutex_lock_slowpath
      Sleep Func      1968             0.157757   0.000080   io_schedule
   RUNQ                                0.000020
   CPU                                 0.117886
lseek                 1970    394.0    2.222922   0.001128    0.034938        0
```

```
SLEEP              1678    335.6    2.173081   0.001295
   Sleep Func      1678             2.159010   0.001287  __mutex_lock_slowpath
CPU                                 0.049840
```

To go back to the summary only, use the Hide Syscall Detail command ('-').

5.1.3 Excluding syscalls

If you have system calls that are called at a high rate that are not critical to the performance path, such as gettid(), getrusage(), gettimeofday(), or ctime(), you can exclude these system calls from being traced using the Exclude System Call command ('e'). You will then be prompted for the system call to exclude. If you want to exclude multiple system calls, use the Exclude System Call command ('e') multiple times. You can also startup ***kiinfo -live*** using the sysignore=<filename> flag and specify a file with all the system calls you wish to ignore. You can see all the system calls that are ignored using the Show Excl Syscalls command ('E'). You can clear all the system calls by using the Exclude System Call Command ('e') and typing "clear". You cannot clear just one or a subset of system calls. You must clear them all and reselect system calls to exclude if needed.

NOTE
You can only ignore system calls if you are doing real-time analysis with kiinfo. You cannot ignore system calls from existing trace dump collected using the runki script.

5.1.4 Using the Previous Screen command

The Prev Screen command ('b') will only go back to the previous screen. Selecting 'b' twice will put you back to the window you started on. For example, if you are on the Global Task List screen ('g') and you select a tasks using the Select command ('s'), if you then use the Prev Screen command ('b'), then you will then go back to the Global Task List screen. If you use the Prev Screen command ('b') again, then you will go back to the Task you previously selected.

5.1.5 Using the Refresh command

The Refresh command ('r') can be very helpful in a couple of different cases. If you resize your window, you can refresh the screen to accept the new window size. Also, if you use the refresh command in the middle of an interval, it will update the screen with statistics from the start of the interval to the time the Refresh command ('r') was used. In other words, you don't have to wait for the end of the interval to get some statistics. Also, if you are running ***kiinfo -live*** on a very large LinuxKI dump from the ***runki*** script, the Refresh command can give you statistics based on the trace records analyzed up to the point where the refresh command was entered. However, remember that the analysis is not complete until all the trace records are analyzed.

5.1.6 Terminal screen size

For best results, use a large terminal screen, such as 48 rows and 160 columns (or larger). Note that you can resize the screen online and hit Refresh command ('r') to format for the new screen size. Note that

the amount of statistics displayed depends on the screen size. For example if the screen size only has 80 columns, then the disk statistics will only show the totals statistics for each disk:

```
------------------------ Total I/O -------------------------
Device         IO/s    MB/s  AvIOsz AvInFlt   Avwait   Avserv
All             217       2       8    0.19     0.00     0.08
sda             217       2       8    0.19     0.00     0.08
```

However, if the screen size is 160 columns or larger, then ***kiinfo*** will also split out the read statistics and the write statistics.

```
------------------------ Total I/O ------------------------- ------------------- Write I/O ------
------------ -------------------- Read I/O -------------------
Device         IO/s    MB/s  AvIOsz AvInFlt   Avwait   Avserv    IO/s    MB/s  AvIOsz AvInFlt
Avwait   Avserv    IO/s    MB/s  AvIOsz AvInFlt   Avwait   Avserv
All             245       2       8    0.30     0.00     0.08     237       2       8    0.30
0.00     0.08       8       0       8    0.00     0.00     0.08
sda             245       2       8    0.30     0.00     0.08     237       2       8    0.30
0.00     0.08       8       0       8    0.00     0.00     0.08
```

Also, the number of rows dictates how much data can be shown on each screen.

5.1.7 Other important information

When you change screens, you are likely to change in the middle of an interval. Some of trace records may not have been enabled at the start of the interval, but were enabled when the screen was changed. So you may need to wait for a full interval to get good statistics. If you have a long interval value, you can use the Refresh command ('r') to get some updated statistics before the end of the interval.

When running ***kiinfo -live*** on a running system, the impact on performance depends on the amount of trace records logged. Some windows, such as the Global Task List will trace every task and will produce more trace records. However, if you select on a task, CPU, or disk, you will only log the trace records needed for the selection. Note that ***kiinfo*** will take more CPU time on larger systems as it must merge the per-cpu trace records. So it will can take a significantly more CPU time on fully loaded Superdome Flex server (1792 logical cores) than on a fully loaded DL380.

Also, for live LinuxKI tracing to work well, the ***kiinfo*** process does need CPU resources to collect and analyze the LinuxKI data. If the system is under severe CPU pressure, LinuxKI events could easily be missed which can distort the data collected.

5.1.8 Step feature

The curses-based interface includes a step feature when analyzing KI dump data collected with the LiKI tracing module. The step feature allows the user to step through the trace in time intervals, either forward or backwards, or to jump to a specific time and begin stepping through. The step interval can be specified either on the kiinfo -live command using the step flag, or can be set while in curses mode

using the ‘j’ command. For example, the following line starts up kiinfo in curses mode and specifies a step time of 100 milliseconds (or 0.100 seconds):

```
$ kiinfo -live step=0.100 -ts 0916_1015
```

Once you are in kiinfo in curses mode, you can move forward one interval using the ‘>’ command, or move backwards one interval using the ‘<’ command. You can also specify a new step interval time using the ‘j’ command, or jump to a specific time in the trace using the ‘J’ command.
The new step feature make it easy to perform a “playback” of the KI dump data and look at different statistics for the same interval.

Note that all of the statistics are cleared at the start of each interval.

6 KI trace events

In order to understand the LinuxKI trace reports, you need to understand where kernel trace events come from and which kernel trace events are captured.

6.1 LinuxKI trace events - Where does it come from?

Embedded in the Linux kernel are various trace points, such as the following trace point in the context_switch() function:

```
context_switch(struct rq *rq, struct task_struct *prev,

               struct task_struct *next)
{
         struct mm_struct *mm, *oldmm;

         prepare_task_switch(rq, prev, next);
         trace_sched_switch(rq, prev, next);
         mm = next->mm;
         oldmm = prev->active_mm;
...
```

When enabled, the tracepoint function is executed and logs data to kernel ring buffers which are exposed by accessing the debugfs filesystem. This tracing mechanism is called *ftrace*. Note that ***kiinfo -kitracedump*** will enable the key *ftrace* tracepoints and dump the trace events from the per-CPU ring buffers.

The *LiKI* DLKM uses a similar mechanism used by *ftrace* but installs its own custom tracing code as opposed to executing the *ftrace* tracing code. It also uses a lighter weight ring buffer implementation. Like *ftrace*, *LiKI* exposes its trace data through ring buffer files in the debugfs. ***Kiinfo -likidump*** will enable the *LiKI* tracepoints (installed by the DLKM) and dump the trace events from the per-CPU ring buffers.

The key tracepoints enabled are:

Event	Subsys	Description	Comments
sched_switch	sched	Switch one task off the CPU and another task on the CPU	Kernel stack trace with *LiKI* only
sched_wakeup	sched	Wakeup a task that was asleep	
sched_wakeup_new	sched	Wakeup a new task that was asleep	
sched_migrate_task	sched	Migrate a task to run on a different CPU	Not enabled by default
sys_enter	syscall	System Call entry	
sys_exit	syscall	System Call exit	
block_rq_insert	block	Insert a device block request on the queue	
block_rq_issue	block	Issue a device block request from the queue	
block_rq_complete	block	Complete a device block request	
block_rq_requeue	block	Requeue a request to a block device	
power_start	power	Change the C-state (power savings)	Not enabled by default
power_end	power	End the C-state	Not enabled by default
power_freq	power	Change the CPU clock frequency	
hardclock	prof	CPU function profiling (every 10 ms)	*LiKI* only
irq_handler_entry	irq	Enter a Hard IRQ handler	enabled by default (version 5.4)
irq_handler_exit	irq	Exit a Hard IRQ handler	enabled by default (version 5.4)
softirq_raise	irq	Raise a Soft IRQ interrupt	enabled by default (version 5.4)
softirq_entry	irq	Enter a Soft IRQ handler	enabled by default (version 5.4)
softirq_exit	irq	Exit a Soft IRQ handler	enabled by default (version 5.4)
scsi_dispatch_cmd_start	scsi	Start a SCSI command request	Not enabled by default
scsi_dispatch_cmd_done	scsi	Complete a SCSI command request	Not enabled by default
workqueue_queue_start	workqueue	queue deferred work item	Not enabled by default
workqueue_execute_start	workqueue	execute deferred work item	Not enabled by default

Not all trace events are collected by default. This is done to avoid the LinuxKI from collecting very large amounts of data. Plus, the fewer tracepoints logged, the less overhead the tool has on the system. However, for some special situations and for debugging and education, additional trace points are provided. The non-default tracepoints can be enabled using the events= and subsys= flags for specific ***kiinfo*** tools, such as ***kiinfo -kitrace***, ***kiinfo -kipid***, etc...

The LinuxKI data has little visibility into user space execution. Other than recording a program counter in the HARDCLOCK trace when the CPU is executing in user space, traces are captured only during kernel execution. In some cases, the sched_switch records will dump the user stack trace along with the kernel stack trace.

6.2 Common LinuxKI trace fields

The *ftrace* tracing events and the *LiKI* tracing events have different formats for each trace records. However, ***kiinfo*** will convert each *ftrace* record into the appropriately formatted *LiKI* trace record, so they are displayed the same. Below is an example LinuxKI trace record with the common fields highlighted in yellow:

```
    0.001067 cpu=20 seqcnt=80 pid=66543 tgid=66543 write [1] ret1=0x2000 syscallbeg=
0.000190 A0=3 A1=0x136d000 A2=8192
```

timestamp	By default, time offset in seconds from the beginning of the trace.
cpu	CPU which logged the trace event.
seqcnt	Per-CPU trace event sequence counter (*LiKI* only). By default, the *seqcnt* field is not printed unless the *seqcnt* flag is passed in ***kiinfo -kitrace***.
pid	PID number of the task that logged the trace event. Note that PID 0 means the CPU is idle, and PID -1 means the CPU was executing an interrupt.
tgid	The Task Group ID (TGID) for the task that logged the trace event (*LiKI* only)
rec_id	Event type or record ID. For system call entry or syscall exit records, the system call is listed.

6.3 syscall_enter / syscall_exit events

Both *ftrace* and *LiKI* log two types of system call records, one record for the system call entry and another record for the system call exit:

```
   15.868768 cpu=76 pid=86949 tgid=86949 open [2] entry *pathname=0x388c556fbe
flags=CLOEXEC mode=0666 filename: /proc/meminfo
   15.868772 cpu=76 pid=86949 tgid=86949 open [2] ret=6 syscallbeg=    0.000004
*pathname=0x388c556fbe flags=CLOEXEC mode=0666
```

The syscall trace records will log the following information in addition to the common fields:

[syscallno]	System Call number (shown in brackets, i.e. [2]).
entry	Designates this event as a syscall_enter event
ret1=	Return value (only for syscall_exit events)
syscallbeg=	Elapsed time in seconds of the system call (only for syscall_exit events). This is the amount of time between the syscall entry and the syscall exit. Note that this time is calculated by kiinfo, and not logged by *ftrace* or *LiKI*. If the syscall entry is not logged (i.e. it occurred before the trace started), then syscallbeg= and the arguments are not logged.
arguments	The arguments are displayed and formatted.

If the *LiKI* tracing mechanism is used then ***kiinfo*** will print additional useful information with the system call entry record, such as the filename to the open() system call or the binary name to the exec() or execve() system call. For example:

```
    1.818945 cpu=4 pid=3604 open [2] entry *pathname=0x16529e0 flags=0x0 mode=0x6
filename: /dev/sda
    1.819406 cpu=4 pid=3604 open ret=5 syscallbeg= 0.000461 *pathname=0x16529e0
flags=0x0 mode=0x6
```

Kiinfo will include the inode and device if the system call is reading or writing to a file and will print the local and remote IP:Port and socket type, if LiKI traces in used. For example:

```
    0.000316 cpu=23 pid=51482 tgid=51482 write [1] ret=0x54 syscallbeg= 0.000019 fd=3
*buf=0x7fb727548200 count=84 family=INET type=TCP L=16.105.211.235:22
R=16.212.40.156:52806
   19.485807 cpu=8 pid=21912 tgid=21888 write [1] ret=0x40000 syscallbeg= 0.000147
fd=56 *buf=0x7f1462bbce80 count=262144 type=REG dev=0x0fd0000c ino=131361199
```

6.4 Scheduler events

There are 3 primary scheduling events - sched_wakeup, sched_switch, and sched_migrate task.

6.4.1 sched_switch

A context switch is when a running tasks stops running on a CPU and another task starts running on the same CPU. A context switch is said to be ***voluntary*** if the task that stops running does so in order to go to **sleep** on an event or resource, such as disk I/O, a kernel lock, or a timeout. A context switch is said to be ***involuntary*** if the task is **forced** off the CPU by higher priority tasks or due to a timeslice. If a tasked is forced off the CPU, then the task remains on the CPU Run Queue. In both cases, a sched_switch event is logged.

6.4.1.1 Voluntary Context Switches

For voluntary context switches, when the switch() call is made, the CPU switches from one task to another and a sched_switch trace record is logged. The calling task (pid=) is the one that is going to sleep. The "next_pid=" represents the task that the CPU will run next. Note that PID 0 means the CPU is idle.

An example sched_switch trace record looks like following:

```
    6.124844 cpu=0 pid=15578 sched_switch syscall=lseek prio=120 state=SSLEEP
next_pid=106 next_prio=120 next_tgid=106 policy=SCHED_NORMAL vss=1015 rss=148
STACKTRACE: __mutex_lock_slowpath+0x13e mutex_lock+0x2b ext4_llseek[ext4]+0x60
vfs_llseek+0x3a sys_lseek+0x66 tracesys+0xd9
```

In the above example, PID 15578 is going to sleep while executing the lseek() system call and is uninterruptable (SSLEEP). The CPU is switching to PID 106, which uses scheduling policy SCHED_NORMAL and whose TGID is 106. The stack trace is one of the most important parts of the switch trace record as this tells you the code path leading up to the switch. In the above example, you

can tell that an lseek() system call was made and the task went to sleep on a mutex that is owned by another task.

Unfortunately, the stacktrace is only provided from the *LiKI* DLKM module. If the trace is generated with *ftrace*, the stacktrace as well as the system call, TGID and policy (in red above) are not logged. A typical switch trace record produced from *ftrace* is as follows:

```
    1.988118 cpu=5 pid=15578 sched_switch prio=120 state=SSLEEP next_pid=15576
next_prio=120
```

The switch record only tells part of the story. When examining a task, you can look at the trace records that surround a system call and identify how much time the task spent in kernel code and how much time the task spent sleeping (switched out). For example, below is a task from an *ftrace* trace sample:

```
    0.014583 cpu=30 pid=66547 lseek [8] entry fd=3 offset=0x5470000 whence=0
    0.014609 cpu=30 pid=66547 sched_switch prio=120 state=SSLEEP next_pid=0
next_prio=120
    0.019606 cpu=10 pid=66543 sched_wakeup target_pid=66547 prio=120 target_cpu=30
success=1
    0.019614 cpu=30 pid=0 sched_switch prio=n/a state=n/a next_pid=66547 next_prio=120
    0.019637 cpu=30 pid=66547 sched_switch prio=120 state=SSLEEP next_pid=0
next_prio=120
    0.019839 cpu=10 pid=66543 sched_wakeup target_pid=66547 prio=120 target_cpu=30
success=1
    0.019847 cpu=30 pid=0 sched_switch prio=n/a state=n/a next_pid=66547 next_prio=120
    0.019853 cpu=30 pid=66547 sched_wakeup target_pid=66544 prio=120 target_cpu=130
success=1
    0.019855 cpu=30 pid=66547 lseek [8] ret1=0x5470000 syscallbeg= 0.005272 fd=3
offset=0x5470000 whence=0
```

In the example above, PID 66547 goes to sleep twice (sched_switch in yellow). It remains asleep until the sched_wakeup is done, and then the process is on the CPU RunQ. It stays on the CPU RunQ until another sched_switch is done to switch to PID 66547 (next_pid=66547). The time between the sched_switch and the sched_wakeup is the time the task spent sleeping. So the task slept 0.019606 - 0.014609 = 0.004997 seconds (4.997 milliseconds) and then slept again for 0.019839 - 0.019637 = 0.000202 seconds (or 0.202 milliseconds).

6.4.1.2 Involuntary Context Switches

The switch() kernel routine is also called when a task is interrupted by a higher priority task or its timeslice is used. Note the state of the calling task is state=RUN, which mean the task is now on the CPU RunQ and is not in a SLEEP state.

A sample sched_switch trace record when a tasks is interrupted is as follows:

```
    9.288779 cpu=4 pid=15662 sched_switch syscall=execve prio=120 state=RUN
next_pid=19 next_prio=0 next_tgid=19 policy=SCHED_FIFO vss=2307 rss=284 STACKTRACE:
wait_for_completion+0x1d sched_exec+0xdc do_execve+0xe0 sys_execve+0x4a
stub_execve+0x6a
```

6.4.1.3 User Stack Traces

If *LiKI* tracing is used, the sched_switch traces will attempt to log both the kernel AND user stack traces. If data is collected with the ***runki*** script, the symbol table for each executable and library is saved and included in the ki_all.*.tgz file. During post-processing, ***kiinfo*** is able to perform the symbol table lookups for the user stack functions if the executable or library is not stripped of symbol table information:

```
    0.000126 cpu=391 pid=15965 tgid=15965 sched_switch syscall=read prio=120
state=SLEEP next_pid=0 next_prio=120 next_tgid=n/a policy=n/a vss=102967640 rss=7970
sk_wait_data+0xd9 tcp_recvmsg+0x33b inet_recvmsg+0x5a sock_aio_read+0x1a1
do_sync_read+0xfa vfs_read+0x181 sys_read+0x51 tracesys+0xd9 | [libpthread-2.12.so]:
__read_nocancel+0x7 [oracle]: nttfprd+0x155 nsbasic_brc+0x195 nsbrecv+0x56
nioqrc+0x1ee opikndf2+0x3cd opitsk+0x32c opiino+0x3b1 opiodr+0x48d opidrv+0x24b
sou2o+0x91 opimai_real+0x9a ssthrdmain+0x19c main+0xec [libc-2.12.so]:
__libc_start_main+0xfd
```

6.4.1.4 StealTime (LiKI only)

The sched_switch traces will log the amount of CPU time stolen by a VM host since the previous sched_switch trace for the specific CPU. This only works if the VM host updates the stealtime on the guest CPU. Currently, a KVM host is known to update the stealtime, but a VMware host does not. Collecting the stealtime allows ***kiinfo*** to calculate the amount of time the VM host steals from the VM guest on a per-cpu and per-task basis:

```
    0.041634 cpu=0 pid=2983 tgid=2983 sched_switch syscall=[-240] prio=120 state=RUN
next_pid=10 next_prio=120 next_tgid=10 policy=SCHED_NORMAL vss=29002 rss=387
stealtime= 0.010003 retint_careful+0x14 | 0x7ff33686a5ab
```

6.4.1.5 Advanced CPU metrics from Model Specific Registers (x86_64 only)

The sched_switch traces can log advanced CPU metrics by reading certain Model Specific Registers (MSR) on x86_64 systems. Note that these statistics are not collected for Virtual Machines. For example:

```
    0.000000 cpu=15 pid=0 tgid=0 sched_switch syscall=idle prio=n/a state=n/a
next_pid=31432 next_prio=120 next_tgid=31432 policy=SCHED_NORMAL vss=0 rss=0
llcref=378 llcmiss=129 instrs=32364 cycles=220585 fixed_clkfreq=193219
actual_clkfreq=220791 smicnt=0
    0.000097 cpu=15 pid=31432 tgid=31432 sched_switch syscall=write prio=120
state=SSLEEP next_pid=0 next_prio=120 next_tgid=n/a policy=n/a vss=1016 rss=153
llcref=941 llcmiss=288 instrs=51486 cycles=500974 fixed_clkfreq=438565
actual_clkfreq=501185 smicnt=0 io_schedule+0x73 __blockdev_direct_IO_newtrunc+0xb7d
__blockdev_direct_IO+0x77 ext4_direct_IO[ext4]+0x119 generic_file_direct_write+0xc2
__generic_file_aio_write+0x3a1 generic_file_aio_write+0x88 ext4_file_write[ext4]+0x58
do_sync_write+0xfa vfs_write+0xb8 sys_write+0x51 system_call_fastpath+0x16 |  [libc-
2.12.so]:__write_nocancel+0x7  [iotest8]:main+0x8ee  [libc-2.12.so]:__libc
_start_main+0xfd
```

Using these fields, the following statistics can be calculated on a per-CPU or per-PID basis:

- LLC_hit% - Last level cache hit rate
- CPI - Cycles per instruction
- Avg_MHz - Average CPU frequency
- SMI_cnt - Number of System management interrupts

To collect the advanced CPU metrics, the "-R" option must be used on the runki command, or the "msr" flag must be added to the specific kiinfo subtool. For example:

```
$ runki -R

$ kiinfo -live msr
$ kiinfo -kipid msr
```

6.4.2 sched_wakeup

The kernel routine wakeup() is the routine that places tasks on the CPU run queue (RunQ). During the wakeup, a sched_wakeup event is written to the trace buffers. The sched_wakeup event logs the PID number of the task that does the wakeup along with the PID number of the task being woken up and the target CPU where the task is scheduled to run. The task being woken up is put on the CPU RunQ. This can give us a lot of information about threads, resources and their cooperative relationships.

A sched_wakeup trace looks like the following:

```
    0.019606 cpu=10 pid=66543 sched_wakeup target_pid=66547 prio=120 target_cpu=30
success=1
```

Note that in some cases, the wakeup succeeds (success=1), but occasionally the wakeup will fail (success=0), perhaps due to the fact that the task is not sleeping. In the example above, PID 66543, running on CPU 10, is waking up PID 66547 which is scheduled to run on CPU 30. So PID 66543 may have released some resource that PID 66547 was waiting for.

Sometimes the PID number is 0 (*ftrace*) or -1 (*LiKI*), which indicates that a kernel interrupt routine is performing the wakeup. This generally occurs when the sleeping thread is waiting for an external event such as disk IO or network traffic. When the data arrives, the kernel, running in the interrupt control context, processes it and wakes up the waiting thread.

There is also the scenario known as the "thundering herd". This occurs when many tasks are sleeping and waiting for the same resource. Depending on how the resource is handled in the kernel, either one task will be awakened when the resource is available or all waiting tasks will be awakened. If there are many sleeping tasks that are all woken up at the same time, you have a thundering herd of tasks contending for the resource. One task will get the resource and continue execution. The other tasks will be put back to sleep. This flurry of scheduling activity can impact system performance if it happens frequently.

Scheduling latencies can be found using the sched_wakeup record and the sched_switch record timestamp differences. The time difference from sched_wakeup to sched_switch when the task starts to run again represents the runq wait time for the task.

We can also determine relationships among threads by looking at the sched_wakeup traces. If you follow the LinuxKI records for the task calling sched_wakeup() you can find (assuming it is not called from the ICS) the syscall associated with the sched_wakeup. This helps to understand the nature of the task cooperation. More information on this can be found in the Cooperating/competing tasks section.

Looking at the same example from a previous page, we can see that PID 66543 is waking up PID 66547:

```
    0.014583 cpu=30 pid=66547 lseek [8] entry fd=3 offset=0x5470000 whence=0
    0.014609 cpu=30 pid=66547 sched_switch prio=120 state=SSLEEP next_pid=0
next_prio=120
    0.019606 cpu=10 pid=66543 sched_wakeup target_pid=66547 prio=120 target_cpu=30
success=1
    0.019614 cpu=30 pid=0 sched_switch prio=n/a state=n/a next_pid=66547 next_prio=120
0.019637 cpu=30 pid=66547 sched_switch prio=120 state=SSLEEP next_pid=0 next_prio=120
    0.019839 cpu=10 pid=66543 sched_wakeup target_pid=66547 prio=120 target_cpu=30
success=1
    0.019847 cpu=30 pid=0 sched_switch prio=n/a state=n/a next_pid=66547 next_prio=120
    0.019853 cpu=30 pid=66547 sched_wakeup target_pid=66544 prio=120 target_cpu=130
success=1
    0.019855 cpu=30 pid=66547 lseek [8] ret1=0x5470000 syscallbeg= 0.005272 fd=3
offset=0x5470000 whence=0
```

In the example above, PID 66543 wakes up PID 66547, which puts PID 66547 on the CPU RunQ, where it remains until the next switch for a total of 0.019614 - 0.019606 = 0.000008 seconds (0.008 milliseconds). PID 66547 then goes to sleep again and once again is woken up by PID 66543. PID 66547 then stays on the CPU Runq for 0.019847 - 0.019839 = 0.000008 seconds (0.008 milliseconds).

6.4.3 sched_migrate_task

The sched_migrate_task trace occurs when a task migrates from one CPU to another. This trace record helps identify which process may have moved the target task and why:

```
    0.092849 cpu=2 pid=-1 sched_migrate_task target_pid=15575 target_prio=120
orig_cpu=14 dest_cpu=2 STACKTRACE: try_to_wake_up+0xfd wake_up_process+0x15
dio_bio_end_io+0x96 bio_endio+0x1d dec_pending[dm_mod]+0x174 clone_endio[dm_mod]+0x9f
bio_endio+0x1d req_bio_endio+0x9b blk_update_request+0x107 end_clone_bio[dm_mod]+0x5c
bio_endio+0x1d req_bio_endio+0x9b blk_update_request+0x107
blk_update_bidi_request+0x27
```

Note if the *LiKI* tracing mechanism is used, we also get the stack trace (in red), which can be very useful in determining why the task was moved from one CPU to another. In the case above, it was the result of an IO completion.

To understand the context of the migration more, consider the trace records surrounding the migration:

```
    0.092787 cpu=14 pid=15575 sched_switch syscall=read prio=120 state=SSLEEP
next_pid=0 next_prio=120 next_tgid=n/a policy=n/a irq_time=    0.000000 softirq_time=
0.000000 vss=1015 rss=148 STACKTRACE: io_schedule+0x73
```

```
__blockdev_direct_IO_newtrunc+0x6de __blockdev_direct_IO+0x5e
ext4_ind_direct_IO[ext4]+0xba ext4_direct_IO[ext4]+0x56 generic_file_aio_read+0x6bb
do_sync_read+0xfa vfs_read+0xb5 sys_read+0x51 tracesys+0xd9
    0.092841 cpu=2 pid=-1 block_rq_complete dev_t=0x00800070 wr=read
flags=SOFTBARRIER|NOMERGE|QUEUED sector=0x1be0ba18 len=4096 async=0 sync=1 qpid=15575
spid=15575 qtm=  0.000013 svtm=  0.000078
    0.092844 cpu=2 pid=-1 block_rq_complete dev_t=0x0fd00003 wr=read
flags=SORTED|ELVPRIV|ALLOCED sector=0x1be0ba18 len=4096 async=0 sync=2 qpid=15575
spid=15575 qtm=  0.000018 svtm=  0.000091
    0.092849 cpu=2 pid=-1 sched_migrate_task target_pid=15575 target_prio=120
orig_cpu=14 dest_cpu=2 STACKTRACE: try_to_wake_up+0xfd wake_up_process+0x15
dio_bio_end_io+0x96 bio_endio+0x1d dec_pending[dm_mod]+0x174 clone_endio[dm_mod]+0x9f
bio_endio+0x1d req_bio_endio+0x9b blk_update_request+0x107 end_clone_bio[dm_mod]+0x5c
bio_endio+0x1d req_bio_endio+0x9b blk_update_request+0x107
blk_update_bidi_request+0x27
    0.092856 cpu=2 pid=-1 sched_wakeup target_pid=15575 prio=120 target_cpu=2
success=1
    0.092871 cpu=2 pid=0 sched_switch syscall=idle prio=n/a state=n/a next_pid=15575
next_prio=120 next_tgid=15575 policy=SCHED_NORMAL irq_time=    0.000000 softirq_time=
0.000000 vss=0 rss=0
    0.092881 cpu=2 pid=15575 read ret=0x1000 syscallbeg=    0.033601 fd=3
*buf=0x207c000 count=4096
```

Based on the above trace records, PID 15575 was running on CPU 14 when it went to sleep waiting for IO to complete. The IO then completes on CPU 2, and while processing the interrupt, PID 15575 is migrated to the same CPU where the interrupt was processed, then the process wakes up and is run on CPU 2.

The sched_migrate_task trace is not enabled by default. You will need to enable all events or enable the sched_migrate_task event or enable the sched subsystem.

6.4.3.1 User Stack Traces

If LiKI tracing is used, the sched_migrate_task traces will attempt to log both the kernel AND user stack traces. If data is collected with the ***runki*** script, the symbol table for each executable and library is saved and included in the ki_all.*.tgz file. During post-processing, ***kiinfo*** is able to perform the symbol table lookups for the user stack functions if the executable or library is not stripped of symbol table information:

```
    0.011734 cpu=14 pid=248821 tgid=248821 sched_migrate_task target_pid=246558
target_prio=120 orig_cpu=14 dest_cpu=0 stkdepth=12  default_wake_function+0x12
pollwake+0x73 __wake_up_common+0x58 __wake_up_sync_key+0x44 pipe_write+0x460
do_sync_write+0x8d vfs_write+0xbd sys_write+0x58 tracesys+0xdd | [libc-2.17.so]:
__GI___libc_write+0x10
```

6.5 hardclock - CPU function profiling

The kernel routine profile_tick() contains a "tick hook" which the *LiKI* DLKM uses to insert a hardclock trace point. The profile_tick() function is called on each CPU every clock tick (1 millisecond). The *LiKI* hardclock function will log a hardclock trace event every 10 clock ticks (10 msec) which includes the state of the processor at the time of the profile tick and will attempt to obtain a kernel stack trace if the

processor was executing in kernel code at the time of the profile tick. The hardclock trace records help answer the question - "If it's running, what's it doing?"

Some sample hardclock trace records include:

```
0.467071 cpu=0 pid=0 hardclock state=IDLE STACKTRACE: intel_idle+0xde

0.046750 cpu=22 pid=66541 hardclock state=SYS STACKTRACE: _spin_lock+0x21
__mutex_lock_slowpath+0x146 mutex_lock+0x2b generic_file_aio_write+0x59
ext4_file_write+0x61 fsnotify_add_notify_event+0x1ae do_sync_write+0xfa fsnotify+0x10b
autoremove_wake_function+0x0 sched_clock+0x9 liki_global_clock+0x16
native_sched_clock+0x13 audit_syscall_entry+0x272 security_file_permission+0x16
vfs_write+0xb8 sys_write+0x51

17.248797 cpu=10 pid=0 hardclock state=INTR STACKTRACE: rq_completed+0x2a
dm_softirq_done+0xdf blk_done_softirq+0x85 update_ts_time_stats+0x72 __do_softirq+0xc1
call_softirq+0x1c do_softirq+0x65 irq_exit+0x85
smp_call_function_single_interrupt+0x35 call_function_single_interrupt+0x13
finish_task_switch+0x4f thread_return+0x4e hrtimer_start_range_ns+0x14 cpu_idle+0xee
start_secondary+0x202

1.457261 cpu=20 pid=66541 hardclock state=USER STACKTRACE: 0x372c0d95d0
```

Note that when the CPU is executing in kernel code, we get the symbolic stack trace (provided the data was collected with the ***runki*** script). The ***runki*** script saves a copy of the /proc/kallsyms file to translate the code addresses into symbolic names. When executing in user space, we only get the program counter or the symbolic address of the program counter if possible.

Note that when a CPU is in power savings mode, a profile tick may not occur. This causes idle hardclock traces to be underreported.

Kiinfo -kiprof can provide a summary of the hardclocks and provide the most common kernel functions that were executing when the profile tick occurred as well as the top kernel stack traces. Sample output is as follows:

```
******** GLOBAL HARDCLOCK REPORT ********
  Count    USER     SYS    INTR    IDLE
  12551    1933    7240      17    3361

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Kernel Functions executed during profile
   Count     Pct  State  Function
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    3329  26.52%  IDLE   intel_idle
    1929  15.37%  USER   UNKNOWN
    1408  11.22%  SYS    finish_task_switch
     763   6.08%  SYS    _spin_unlock_irqrestore
     580   4.62%  SYS    scsi_request_fn
     254   2.02%  SYS    clear_page_c

non-idle GLOBAL HARDCLOCK STACK TRACES (sort by count):

   Count     Pct  Stack trace
============================================================
     885   7.05%  finish_task_switch
```

```
     457   3.64%  scsi_request_fn  scsi_request_fn  sched_clock  __blk_run_queue
elv_insert  __elv_add_request  blk_insert_cloned_request  dm_dispatch_request
dm_request_fn  lock_timer_base  __blk_run_queue  cfq_insert_request  elv_insert
__elv_add_request  __make_request  dm_request
     260   2.07%  _spin_unlock_irqrestore  qla24xx_start_scsi  scsi_done
qla24xx_dif_start_scsi  mempool_alloc_slab  mempool_alloc  scsi_setup_fs_cmnd
sd_prep_fn  scsi_done  qla2xxx_queuecommand  scsi_dispatch_cmd  scsi_request_fn
sched_clock  __blk_run_queue  elv_insert  __elv_add_request
     237   1.89%  _spin_unlock_irqrestore  try_to_wake_up  wake_up_process
__mutex_unlock_slowpath  mutex_unlock  generic_file_aio_write  ext4_file_write
fsnotify_add_notify_event  do_sync_write  fsnotify  autoremove_wake_function
liki_global_clock  native_sched_clock  audit_syscall_entry  security_file_permission
vfs_write
     213   1.70%  clear_page_c  clear_huge_page  do_huge_pmd_anonymous_page
handle_mm_fault  __do_page_fault  native_sched_clock  sched_clock  liki_global_clock
do_page_fault  page_fault
```

6.5.1 *User Stack Traces*

If LiKI tracing is used, the hardclock traces will log both the kernel AND user stack traces. If data is collected with the ***runki*** script, the symbol table for each executable and library is saved and included in the ki_all.*.tgz file. During post-processing, ***kiinfo*** is able to perform the symbol table lookups for the user stack functions if the executable or library is not stripped of symbol table information:

```
    0.041351 cpu=152 pid=16042 tgid=16042 hardclock state=USER  [oracle]:
kews_sqlcol_end+0x447
    0.052339 cpu=152 pid=16042 tgid=16042 hardclock state=USER  [oracle]:
kksxsccompat2_static_sql+0x53
    0.063335 cpu=152 pid=16042 tgid=16042 hardclock state=SYS  tracesys+0x7a | [libc-
2.12.so]: __times+0xa [oracle]: kcbchg1_main+0x104d kcbchg1+0xd5 ktuchg2+0x639
ktbchg2+0xf0 kdtchg+0x476 kdtwrp+0xce3 kdtSimpleInsRow+0x135
qerltcSimpleSingleInsRowCBK+0x3e qerltcSingleRowLoad+0x130 qerltcFetch+0x185
insexe+0x2e6 opiexe+0x15af opipls+0x7d2 opiodr+0x48d rpidrus+0xce skgmstack+0x90
rpiswu2+0x2d3 rpidrv+0x622 psddr0+0x1de psdnal+0x27c pevm_EXECC+0x130 0x7f02e03ee2a9
pevm_NCAL+0x3e pfrinstr_XCAL+0x18e pfrrun_no_tool+0x3c pfrrun+0x483 plsql_run+0x2c4
peicnt+0x11d kkxexe+0x2d6 opiexe+0x4c24 opiodr+0x48d ttcpip+0xa8b opitsk+0x6c6
opiino+0x3b1 opiodr+0x48d opidrv+0x24b sou2o+0x91 opimai_real+0x9a ssthrdmain+0x19c
main+0xec [libc-2.12.so]: __libc_start_main+0xfd
```

Note that if the state=USER, then often the *LiKI* trace function is not able to traverse the entire stack, so only the function executing at the time of the hardclock trace is displayed.

6.6 Block IO traces

IO can be thought of in two ways – **Logical IO** and **Physical IO**.

Logical IO is the IO requested by the applications in the form of read() or write() or similar system call. Sometimes, those requests are satisfied from the buffer/page cache and sometimes the IO must come from the disk. Below is an example of a logical IO:

```
    0.192045 cpu=20 pid=66544 read [0] ret1=0x2000 syscallbeg= 0.007420 A0=3
A1=0x21e3000 A2=8192
```

In the above trace for the read() system call, the IO request was for 8192 bytes from file descriptor 3. See the section on system call traces for more information on system calls.

Note that it is not evident from the system call if this is related to file or disk I/O. The read() request could be from a terminal or a special device file like /dev/null. The only way to tell if physical IO is generated by the system call is to examine the trace records for the task between the syscall entry record and the syscall exit record.

Physical IO occurs when data must be transferred to or from the disk device. LinuxKI captures the block requests to understand the physical IO for a device. There are three events recorded in LinuxKI trace data during a typical physical IO:

block_rq_insert – logged when IO is queued to the block device queue.
block_rq_issue - logged when the IO is sent to the IO layers below the block device.
block_rq_complete - logged when the block device is notified that the physical IO is complete.

Note that the block device may be an actual physical disk block device such as /dev/sda, or it may be a device mapper block device, such as /dev/dm-9. For IO to a device mapper device, you will often see subsequent trace records for the actual physical devices that the device mapper device maps to.

You may also see the following additional record logged:

block_rq_requeue - logged when the IO request has been requeued to the block device and often occurs when the queue_depth of a SCSI device has been exceeded.

6.6.1 block_rq_insert

As stated earlier, the block_rq_insert trace is created when a request is queued to the block device. Note that these traces are recorded at the Linux block device layer.

The block_rq_insert trace contains many interesting fields. Note the fields in red are only available if the *LiKI* trace mechanism is used.

```
    0.000971 cpu=0 pid=15572 block_rq_insert dev_t=0x00800070 wr=write
flags=NOMERGE|SYNC sector=0x1bff0140 len=4096 async=0 sync=0 bytes=4096
```

Below is a description of the various fields:

dev_t	The device file associated with the request. The number is in the MMMmmmmm format, where "MMM" is the major number and "mmmmm" is the minor number.
rw	Designates whether the IO is a read or write
flags	Internal request flags. See `__REQ_*` definitions in /usr/include/linux/blk_types.h. Flags are more accurate if *LiKI* tracing is done. If *ftrace* tracing is done, only a subset of flags are available.
sector	512-byte device sector address
len	Number of bytes
async	Number of asynchronous IOs in progress (issued) for the device (does not included IOs queued to the elevator)
sync	Number of synchronous IOs in progress (issued) for the device (does not include IOs queued to the elevator)
bytes	Number of bytes requested to be transferred. Typically the same as the len.

NOTE
The async= and sync= fields will always be 0 if the multiqueue block queuing mechanisms are used (blk-mq, scsi-mq).

6.6.2 block_rq_issue

The block_rq_issue trace is recorded when the IO is removed from the block device queue and forwarded to the lower IO layers (such as the SCSI and FC drivers). The block_rq_issue trace will follow the block_rq_insert trace:

```
    0.000972 cpu=0 pid=15572 block_rq_issue dev_t=0x00800070 wr=write
flags=SOFTBARRIER|NOMERGE|SYNC sector=0x1bff0140 len=4096 async=0 sync=0 bytes=4096
```

The information here is the same as the block_rq_insert. The difference in the time between the block_rq_insert and block_rq_issue is considered the time the request spent on the block device queue, or queue time (qtm). In the above 2 examples, the queue time less than 1 microsecond, so the request was issued immediately after it was queued.

Note the async/sync counter is incremented right after the block_rq_issue trace is logged. So theses counters do not include the current request being issued.

6.6.3 block_rq_complete

The block_rq_complete record is logged after the IO is complete and the request is returned to the block device for additional completion processing:

```
    0.001147 cpu=0 pid=-1 block_rq_complete dev_t=0x00800070 wr=write
flags=SOFTBARRIER|NOMERGE|QUEUED|SYNC sector=0x1bff0140 len=4096 async=0 sync=1
qpid=15572 spid=15572 qtm=  0.000005 svtm=  0.000173
```

Note that the PID logged by the block_rq_complete records is often -1 (or 0 if *ftrace* is used), which indicates that the CPU was executing on the Interrupt Control Stack (ICS). In the block_rq_complete trace, we get the additional fields:

qpid	PID of the task that did the block_rq_insert
spid	PID of the task that did the block_rq_issue
qtm	Queue time in seconds. This is the difference in time between block_rq_insert and block_rq_issue.
svtm	Service time in seconds.

Note that the qtm and svtm values are calculated by trying to find the earlier block_rq_insert and block_rq_issue records. If they cannot be located, the qtm and svtm are set to 0, as are the qpid and spid. Occasionally, with the *ftrace* tracing, some missed records can erroneously result in very large queue times or service times for some IOs.

Note that the qtm and svtm are measured at the block device level. This is similar to await and svctm metrics seen with sar -d, however, we are getting the time for each individual IO. This makes it easier to identify any IOs that can take a long time to complete.

6.6.4 block_rq_requeue

The block_rq_requeue record is logged when a request has been inserted and issued, but for some reason must be re-queued and then re-issued. This typically occurs when the SCSI request queue is full. If you see a lot of requeue requests, you can attempt to increase the SCSI queue_depth, but this can impact the IO service time. The block_rq_requeue record is similar to the block_rq_complete record but has an optional "errors" field.

6.7 IRQ events

Hardware interrupts are typically handled in 2 phases. The first phase is the "hard" interrupt processing. These "hard" interrupt request queue (IRQ) events are considered low-level interrupt handlers which process the initial interrupt from the hardware. However, since a CPU can only handle one hard interrupt at a time, the low-level interrupt handler will do the minimum amount of work before handing off the processing to the higher level soft interrupt handler. Note that hardware interrupts are disabled while in a "hard" IRQ handler, but they are enabled during a "soft" irq handler.

The IRQ traces can be collected using either the *ftrace* or *LiKI* mechanisms.

6.7.1 *irq_handler_entry / irq_handler_exit*

Logged on entry and exit of a hard interrupt handler. Identifies the IRQ number and the name of the irq handler (typically the driver name). The irqtm= field is new with LinuxKI 5.0 and is the difference between the entry and exit trace events. For example:

```
0.000437 cpu=1 pid=0 irq_handler_entry irq=57 name=qla2xxx
0.000444 cpu=1 pid=0 irq_handler_exit irq=57 ret=handled irqtm=0.000007
```

6.7.2 *softirq_raise*

The soft IRQ is typically initiated on behalf of a hard IRQ to perform the high level processing of the interrupt:

```
0.000444 cpu=0 pid=0 softirq_raise vec=4 action=BLOCK
```

6.7.3 *softirq_entry / softirq_exit*

Logged on entry and exit of a soft interrupt handler. Identifies the vector number and associated name. The irqtm= field is new with LinuxKI 5.0 and is the difference between the entry and exit trace events. For example:

```
0.000445 cpu=0 pid=0 softirq_entry vec=4 action=BLOCK
0.000457 cpu=0 pid=0 softirq_exit vec=4 action=BLOCK irqtm=0.000008
```

6.8 SCSI IO events

SCSI IO events are not enabled by default. You will need to enable all events or enable the specific events below or enable the SCSI subsystem.

6.8.1 *scsi_dispatch_cmd_start*

When the block IO sends a request down to the SCSI device driver, the driver will need to dispatch the SCSI command needed to start the I/O. Both *ftrace* and *LiKI* will log a trace similar to the following when the IO is dispatched:

```
    2.000195 cpu=0 pid=15572 scsi_dispatch_cmd_start path=1:0:2:1 opcode=WRITE_10
cmd_len=10 data_sglen=1, prot_sglen=0 prot_op=0 cmd=[2a 00 1b e2 e2 b0 00 00 08 00]
```

The cmd= field can be decoded to identify key parts of the request. For example, bytes 2-5 for the SCSI_10 command is the Logical Block Address (LBA). So in this case, the LBA = 0x1be2e2b0. Note there are a variety of other SCSI commands that can be performed, so please be sure to do your research in order to decode the cmd= field. For example, the event below is for a SCSI_INQUIRY:

```
    12.928094 cpu=1 pid=15709 scsi_dispatch_cmd_start path=1:0:0:2 opcode=INQUIRY
cmd_len=6 data_sglen=1, prot_sglen=0 prot_op=0 cmd=[12 01 00 00 fe 00]
```

6.8.2 *scsi_dispatch_cmd_done*

When the IO is completed and the request is returned to the SCSI driver, the driver must perform any IO post-processing. Note the IO completion will usually occur on the Interrupt Control Stack (PID -1 with *LiKI*, and PID 0 with *ftrace*).

```
    2.000311 cpu=1 pid=-1 scsi_dispatch_cmd_done path=1:0:2:1 opcode=WRITE_10
cmd_len=10 data_sglen=1, prot_sglen=0 prot_op=0 cmd=[2a 00 1b e2 e2 b0 00 00 08 00]
result=0
```

The only good way to match up the scsi_dispatch_cmd_done with its respective scsi_dispatch_cmd_start, is to view the fields, such as the path, opcode, and the cmd field. These should be identical.

If these trace records are enabled, you will typically see them after the block IO request starts and before the block IO request completes. Note how bytes 2-5 of the WRITE_10 cmd matches the sector number in the block IO request traces.

```
    2.000192 cpu=0 pid=15572 block_rq_insert dev_t=0x00800070 wr=write
flags=NOMERGE|SYNC sector=0x1be2e2b0 len=4096 async=0 sync=0 bytes=4096
    2.000193 cpu=0 pid=15572 block_rq_issue dev_t=0x00800070 wr=write
flags=SOFTBARRIER|NOMERGE|SYNC sector=0x1be2e2b0 len=4096 async=0 sync=0 bytes=4096
    2.000195 cpu=0 pid=15572 scsi_dispatch_cmd_start path=1:0:2:1 opcode=WRITE_10
cmd_len=10 data_sglen=1, prot_sglen=0 prot_op=0 cmd=[2a 00 1b e2 e2 b0 00 00 08 00]
                     :                                    :
    2.000311 cpu=1 pid=-1 scsi_dispatch_cmd_done path=1:0:2:1 opcode=WRITE_10
cmd_len=10 data_sglen=1, prot_sglen=0 prot_op=0 cmd=[2a 00 1b e2 e2 b0 00 00 08 00]
result=0
    2.000318 cpu=0 pid=-1 block_rq_complete dev_t=0x00800070 wr=write
flags=SOFTBARRIER|NOMERGE|QUEUED|SYNC sector=0x1be2e2b0 len=4096 async=0 sync=1
qpid=15572 spid=15572 qtm=  0.000005 svtm=  0.000123
```

6.9 Workqueue events

Workqueues are used to queue deferred work items for subsequent processing by one of the kworker deamon tasks.

Workqueue events are not enabled by default. You will need to enable all events or enable the specific events below or enable the workqueue subsystem. The Workqueue events can be collected using either the *ftrace* or *LiKI* mechanisms.

6.9.1 *workqueue_queue_work*

The workqueue_queue_work event is logged when a work item (function) is queued for the kworker tasks to execute. The item is queued to be executed by a kworker tasks for the specified target_cpu. If the target_cpu is -1, then it can execute on any CPU.

```
    0.000052 cpu=2 pid=8619 tgid=8619 workqueue_queue_work func=flush_to_ldisc
target_cpu=-1
```

6.9.2 *workqueue_execute_start*

The workqueue_execute_start event is logged with a work item is removed from the workqueue and the function specified is executed.

```
0.000060 cpu=2 pid=6297 tgid=6297 workqueue_execute_start func=flush_to_ldisc
```

6.10 Power events

As part of the Advanced Configuration and Power Interface (acpi) for the Intel x86_64 platforms, each CPU may have a power state (c-state) from c0 to c3 (or higher), which are defined as follows:

c0 - processor is operating normally.
c1 - processor is not actively executing instructions, but can resume operation instantaneously.
c2 - processor maintains software-visible state, but may take longer to wake up.
c3 - processor does not keep its cache coherent, but maintains other states.

Note the definition of the cstates above may change depending on the hardware specifications.

Both *ftrace* and *LiKI* trace mechanisms will log 3 power events which can be used to identify how the level of power_savings and how the performance may be impacted.

- power_start - logged when the c-state is set to a non-zero value. The c-state is typically set in the idle code so that idle CPUs draw less power than busy CPUs. The higher the c-state, the less power it uses. The power_start trace event is not enabled by default.
- power_end - logged when the c-state is set to zero. This is typically logged when a processor is coming out of the idle loop.
- power_freq - logged when the clock frequency changes on a CPU.

When using ftrace, the power_start / power_end trace events may be replaced by the cpu_power events.

6.10.1 *power_start / power_end*

The power_start event is logged when the c-state is set to a non-zero value. The c-state is typically set in the idle code so that idle CPUs draw less power than busy CPUs. The higher the c-state, the less power it uses.

The power_end event is logged when the c-state is set to c0. This is typically logged when a processor is coming out of the idle loop.

```
0.000079 cpu=11 pid=0 power_start state=3
0.000597 cpu=11 pid=0 power_end
```

In the example above, the c-state is changed to c3. CPU 11 remains idle in the c3 state until the power_end is logged. In the case above, the processor was in the c3 state for 518 microseconds.

When using ftrace, the power_start and power_end trace events may be replaced by the cpu_idle trace event depending on the Linux version. The cpu_idle state=-1 is equivalent to the power_end trace event, which is when the c-state is set to c0.

```
0.000252 cpu=0 pid=0 tgid=0 cpu_idle state=1
0.000307 cpu=0 pid=0 tgid=0 cpu_idle state=-1
```

Note that the power_start and power_end traces are not collected by default. Also, the power_end event may also be missing in some Linux distributions.

6.10.2 power_freq

The power_freq trace record is logged when the cpu clock frequency is changed:

```
0.507872 cpu=11 pid=0 power_freq target_cpu=15 freq=2200000
1.327601 cpu=11 pid=0 power_freq target_cpu=15 freq=2300000
```

6.11 Process creation

Processes are created with the fork/exec or clone system calls. The clone() system call is similar to fork(), but it also allows the child process to share parts of its execution context with the parent process, such as the memory space, file descriptors, and signal handlers. In a sense, the clone() system call is a superset of the fork(), vfork() and pthread_create() system calls on HP-UX.

An example of a parent process calling close is as follows:

```
    8.591883 cpu=121 pid=135636 clone [56] entry *fn()=0x1200011 *child_stack=0x0
flags=0x0 arg3=0x7f1a58a859d0 arg4=0x0 arg5=0x0
    8.592020 cpu=121 pid=135636 sched_wakeup_new target_pid=135637 prio=120
target_cpu=131 success=1
    8.592021 cpu=121 pid=135636 clone [56] ret1=0x211d5 syscallbeg= 0.000138
*fn()=0x1200011 *child_stack=0x0 flags=0x0 arg3=0x7f1a58a859d0 arg4=0x0 arg5=0x0
```

Note the return value (ret1=) is 0x211d5, or PID 135637, which matches the target_pid of the sched_wakeup_new trace record. The child process will start similar to the following:

```
    8.592020 cpu=121 pid=135636 sched_wakeup_new target_pid=135637 prio=120
target_cpu=131 success=1
    8.592123 cpu=131 pid=0 sched_switch prio=n/a state=n/a next_pid=135637
next_prio=120
    8.592147 cpu=131 pid=135637 clone [56] ret1=0x0
          :                  :
    8.592431 cpu=131 pid=135637 execve [59] entry *filename=0x22f1660
*argv[]=0x22f16c0 *envp[]=0x22f0550
    8.592450 cpu=131 pid=135637 execve [59] ret1=0xfffffffffffffffe syscallbeg=
0.000019 *filename=0x22f1660 *argv[]=0x22f16c0 *envp[]=0x22f0550
```

6.12 Tracing non-default events

There are a number of tracing events that are not enabled by default. These trace events are not enabled by default to reduce the overhead of the trace collection as well as to reduce the size of the resulting LinuxKI binary trace file.

Event	Subsys	Description	Comments
sched_migrate_task	sched	Migrate a task to a different CPU	
power_start	power	Change the C-state (power savings)	
power_end	power	End the C-state	
scsi_dispatch_cmd_start	scsi	Start a SCSI command request	
scsi_dispatch_cmd_done	scsi	Complete a SCSI command request	
softirq_raise	irq	Raise a Soft IRQ interrupt	
workqueue_insertion	workqueue	Insert an item onto the workqueue	< 3.0 kernels
workqueue_execution	workqueue	Execute an item on the workqueue	< 3.0 kernels
workqueue_queue_work	workqueue	Insert an item onto the workqueue	>= 3.0 kernels
workqueue_execute_start	workqueue	Insert an item onto the workqueue	>= 3.0 kernels

Note
Beginning with LinuxKI version 5.4, the IRQ trace events (softirq_entry, softirq_exit, irq_handler_entry, irq_handler_exit) are now enabled by default.

6.12.1 Collecting LinuxKI data using non-default tracing events

In most cases, the default traces provide enough detail to solve most problems. However, there may be cases when it's necessary to enable one or more of the non-default tracing events. Note that non-default trace events must be enabled when the data is collected. The ***kiinfo -kitracedump*** and ***kiinfo -likidump*** subtools have been enhanced to allow the events and/or subsys to be traced. The following are examples of using the events and subsys flags to enable non-default tracing events:

```
Enable default events + scsi subsys events
$ kiinfo -kitracedump dur=20,events=default,subsys=scsi

Enable only events needed by kiinfo -kidsk
$ kiinfo -kitracedump dur=20,events=kidsk

Enable all events including non-default events
$ kiinfo -likidump dur=20,events=all

Enables default events and the workqueue_queue_work event
$ kiinfo -likidump dur=20,events=default,events=workqueue_queue_work
```

You can also enable non-default trace events when collecting a LinuxKI dump using the ***runki*** script with the events option "-e" and subsys option "-s":

```
$ runki -e kiprof            # enable only the trace events needed for kiinfo -kiprof
$ runki -e default -s scsi   # enable all default traces + scsi subsys trace events
$ runki -e all               # enable all trace events
```

To specify more than one event or subsys with the runki script, use the -V option.

6.12.2 Analyzing LinuxKI data using non-default tracing events

At the present time, the non-default trace events are only displayed in the formatted ASCII trace file (***kiinfo -kitrace***). The exception to this is the C-state information which uses the power_start and power_end trace events.

6.12.3 Enabling non-default trace event with live LinuxKI analysis

You can also enable non-default trace events when performing live LinuxKI analysis. For example, suppose you want to see if there are any SCSI SYNCHRONIZE_CACHE commands on a live system over a 1 hour period:

```
$ kiinfo -kitrace subsys=scsi -a 60 -p 60 | grep SYNCHRONIZE_CACHE
```

6.13 Putting it all together

The best way to understand how to analyze the formatted LinuxKI records is to look at an example. The following trace records were captured using ***runki*** using the *ftrace* tracing mechanism. With *ftrace*, there are no stacktraces in the sched_switch records. For this example, we are interested in the activity for PID 97700:

```
$ kiinfo -kitrace pid=97700 -ts 0816_1308
              :                              :
    4.462753 cpu=30 pid=97700 write [1] entry fd=3 *buf=0x1f59000 count=2048
    4.462780 cpu=30 pid=97700 block_rq_insert dev_t=0x0fd00003 flags=0x0 wr=read
sector=0xcc6cef len=4096
    4.462786 cpu=30 pid=97700 block_rq_issue dev_t=0x0fd00003 flags=0x0 wr=read
sector=0xcc6cef len=4096
    4.462790 cpu=30 pid=97700 block_rq_insert dev_t=0x04200060 flags=0x0 wr=read
sector=0xcc6cef len=4096
    4.462791 cpu=30 pid=97700 block_rq_issue dev_t=0x04200060 flags=0x0 wr=read
sector=0xcc6cef len=4096
    4.462823 cpu=30 pid=97700 sched_switch prio=120 state=SSLEEP next_pid=0
next_prio=120
    4.470950 cpu=30 pid=0 block_rq_complete dev_t=0x04200060 flags=0x0 wr=read
sector=0xcc6cef len=4096 qpid=97700 spid=97700 qtm= 0.000001 svtm= 0.008159
    4.470951 cpu=30 pid=0 block_rq_complete dev_t=0x0fd00003 flags=0x0 wr=read
sector=0xcc6cef len=4096 qpid=97700 spid=97700 qtm= 0.000006 svtm= 0.008165
    4.470954 cpu=30 pid=0 sched_wakeup target_pid=97700 prio=120 target_cpu=30
success=1
    4.470964 cpu=30 pid=0 sched_switch prio=n/a state=n/a next_pid=97700 next_prio=120
    4.470973 cpu=30 pid=97700 sched_wakeup target_pid=14340 prio=120 target_cpu=53
success=1
    4.470975 cpu=30 pid=97700 write [1] ret1=0x800 syscallbeg= 0.008222 fd=3
*buf=0x1f59000 count=2048
```

Below is an explanation of the events above:

1. write() system call is entered (4.462753). The process is writing to FD 3 for 2048 bytes.
2. A block request is inserted (4.462780) for the block mapper device (major 0xfd = 253) to READ 4096 bytes (4K).
3. The request is then issued (4.462786) from the block mapper device to the lower layers.

4. The request is then inserted (4.462790) to the underlying block device (major 0x42 = 66) to READ 4096 bytes (4K).
5. The request is then issues (4.462791) to the underlying IO layers (SCSI, FC).
6. The task then goes to sleep while it waits for the read to complete (4.462823).
7. The IO is complete (4.470950) and the request is returned to the block device (major 0x42 = 66).
8. Once the lower block device finished the I/O, the request is then returned to the device mapper block device to be completed (4.470951).
9. The sleeping task is then woken up (4.470954) and placed on the CPU RunQ
10. The CPU then switches (4.470964) to the task that was just placed on the RunQ.
11. PID 97700 then wakes up PID 14340, perhaps to flush the page to disk.
12. The write() system call completes. The elapsed time for the write is 0.008222 seconds, or 8.222 milliseconds.

Note that the 2K logical write results in a single 4K physical READ. Why the read? The write operation is performed through the buffer/page cache, and the smallest amount of data it can access is a page (4K). So the write() must first read in the page it is about to modify. Note it must go through a mapper device before it goes to the physical block device. Perhaps the filesystem resides on a LVM volume or on a multipath device. Examining the output of "ll /dev", you can identify the mapper device used, just to remember to convert from hex to decimal. For example, 0x0fd00003 is Major 253, Minor 3.

6.14 Putting it all together (Part 2)

A more advanced analysis can be done when enabling the non-default trace records, such as the SCSI and irq trace records. In this example, the ***runki*** script was executed as follows:

```
$ runki -m -e all
```

The above command will not collect the collectl logs (-m), and it will enable all of the trace events (-e all). The example below is for a simple 4Kb direct IO read:

```
    0.114767 cpu=2 pid=15575 read [0] entry fd=3 *buf=0x207c000 count=4096
    0.114801 cpu=2 pid=15575 sched_switch syscall=read prio=120 state=SSLEEP
next_pid=0 next_prio=120 next_tgid=n/a policy=n/a irq_time=    0.000000 softirq_time=
0.000000 vss=1015 rss=148 STACKTRACE: __mutex_lock_slowpath+0x13e mutex_lock+0x2b
__blockdev_direct_IO_newtrunc+0x1cd __blockdev_direct_IO+0x5e
ext4_ind_direct_IO[ext4]+0xba ext4_direct_IO[ext4]+0x56 generic_file_aio_read+0x6bb
do_sync_read+0xfa vfs_read+0xb5 sys_read+0x51 tracesys+0xd9
    0.204128 cpu=1 pid=15574 sched_wakeup target_pid=15575 prio=120 target_cpu=2
success=1
    0.204136 cpu=2 pid=0 sched_switch syscall=idle prio=n/a state=n/a next_pid=15575
next_prio=120 next_tgid=15575 policy=SCHED_NORMAL irq_time=    0.000000 softirq_time=
0.000000 vss=0 rss=0
    0.204155 cpu=2 pid=15575 block_rq_insert dev_t=0x0fd00003 wr=read
flags=ELVPRIV|ALLOCED sector=0x1c09cab8 len=4096 async=0 sync=0 bytes=4096
    0.204163 cpu=2 pid=15575 block_rq_issue dev_t=0x0fd00003 wr=read
flags=SORTED|ELVPRIV|ALLOCED sector=0x1c09cab8 len=4096 async=0 sync=0 bytes=4096
    0.204168 cpu=2 pid=15575 block_rq_insert dev_t=0x00800070 wr=read flags=NOMERGE
sector=0x1c09cab8 len=4096 async=0 sync=0 bytes=4096
    0.204169 cpu=2 pid=15575 block_rq_issue dev_t=0x00800070 wr=read
flags=SOFTBARRIER|NOMERGE sector=0x1c09cab8 len=4096 async=0 sync=0 bytes=4096
    0.204173 cpu=2 pid=15575 scsi_dispatch_cmd_start path=1:0:2:1 opcode=READ_10
cmd_len=10 data_sglen=1, prot_sglen=0 prot_op=0 cmd=[28 00 1c 09 ca b8 00 00 08 00]
```

```
    0.204182 cpu=2 pid=15575 sched_wakeup target_pid=15572 prio=120 target_cpu=0
success=1
    0.204191 cpu=2 pid=15575 sched_switch syscall=read prio=120 state=SSLEEP
next_pid=0 next_prio=120 next_tgid=n/a policy=n/a irq_time=    0.000000 softirq_time=
0.000000 vss=1015 rss=148 STACKTRACE: io_schedule+0x73
__blockdev_direct_IO_newtrunc+0x6de __blockdev_direct_IO+0x5e
ext4_ind_direct_IO[ext4]+0xba ext4_direct_IO[ext4]+0x56 generic_file_aio_read+0x6bb
do_sync_read+0xfa vfs_read+0xb5 sys_read+0x51 tracesys+0xd9
    0.204265 cpu=1 pid=0 irq_handler_entry irq=57 name=qla2xxx
    0.204268 cpu=1 pid=-1 scsi_dispatch_cmd_done path=1:0:2:1 opcode=READ_10
cmd_len=10 data_sglen=1, prot_sglen=0 prot_op=0 cmd=[28 00 1c 09 ca b8 00 00 08 00]
result=0
    0.204272 cpu=1 pid=0 irq_handler_exit irq=57 ret=handled
    0.204273 cpu=2 pid=0 softirq_raise vec=4 action=BLOCK
    0.204273 cpu=2 pid=0 softirq_entry vec=4 action=BLOCK
    0.204276 cpu=2 pid=-1 block_rq_complete dev_t=0x00800070 wr=read
flags=SOFTBARRIER|NOMERGE|QUEUED sector=0x1c09cab8 len=4096 async=0 sync=1 qpid=15575
spid=15575 qtm=  0.000008 svtm=  0.000104
    0.204277 cpu=2 pid=-1 block_rq_complete dev_t=0x0fd00003 wr=read
flags=SORTED|ELVPRIV|ALLOCED sector=0x1c09cab8 len=4096 async=0 sync=2 qpid=15575
spid=15575 qtm=  0.000015 svtm=  0.000112
    0.204283 cpu=2 pid=-1 sched_wakeup target_pid=15575 prio=120 target_cpu=2
success=1
    0.204287 cpu=2 pid=0 softirq_exit vec=4 action=BLOCK
    0.204298 cpu=2 pid=0 sched_switch syscall=idle prio=n/a state=n/a next_pid=15575
next_prio=120 next_tgid=15575 policy=SCHED_NORMAL irq_time=    0.000000 softirq_time=
0.000000 vss=0 rss=0
    0.204307 cpu=2 pid=15575 read ret=0x1000 syscallbeg=    0.089540 fd=3
*buf=0x207c000 count=4096
```

Below is an explanation of the events above:

1. read() system call is entered (0.114767). The process is reading 4096 bytes from FD 3.
2. the task switches off the CPU while it waits on a mutex lock (0.114801)
3. The task is woken up by PID 15574 after sleeping for 0.204128 - 0.114801 seconds, or 89.327 msecs (0.204128)
4. The task starts to run on CPU 2, which was previously idle (0.204136).
5. The task inserts a read request on the device-mapper block device (major 0xfd or 253, minor 3) for sector 0x1c09cab8 (0.204155)
6. The task removes the request from the device-mapper block device queue and issues the IO to the lower layer (0.204163)
7. The task receives the request from the device-mapper block device and inserts it into the scsi block device queue (major 8, minor 0x70 or 112) (0.204168)
8. The task removes the request from the scsi block device queue and issues the IO to the lower layer (scsi driver) (0.204169)
9. The scsi driver issues a READ_10 scsi command for LBA 0x1c09cab8 for 0x0008 sectors or 4Kb (0.204173)
10. Before the task goes to sleep, it wakes up PID 15572 (0.204182)
11. Task goes to sleep in an uninterruptible state (SSLEEP) sleeping in io_schedule() waiting for the Direct IO read to complete. As a result, CPU 2 becomes idle (next_pid=0) (0.204191)
12. CPU 1 receives an interrupt from IRQ 57 managed by the QLogic FC driver (qla2xxx) (0.204265)
13. As a result of the interrupt, the SCSI driver completes the SCSI command previously issues. The physical IO actual takes 95 microseconds (0.204191)

14. CPU 1 exists the IRQ handler for IRQ 7. The IO is complete, but there is more post-processing for the IO to do (0.204272)
15. As a result of the IO completion, a soft IRQ is raised for the BLOCK device (0.24273)
16. The soft IRQ is entered for the BLOCK device (0.204276)
17. The read request is completed at the SCSI block device layer. As far as the block device goes, the IO took 104 microseconds. This is what gets reported as the IO service time (0.204276).
18. The read request goes back through the device-mapper block device and is completed (204277)
19. While still on the ICS on CPU 2, a wakeup is performed for the task that is waiting on the IO (0.204283)
20. The soft IRQ handler executing on CPU 2 is then exited (0.204287)
21. CPU switches from idle to the task that was just woken up (0.204298)
22. The task wakes up and completes the logical IO request. The read() system call completes after 89.540 milliseconds. Note that most of the time was spent sleeping on the mutex lock. The physical read was very fast. (0.204307)

7 Key LinuxKI statistics

Many of the reports have a variety of key statistics reported. Some statistics are global, while other statistics are specific to a particular task, NUMA node, CPU, disk device, etc. These key statistics and how they are calculated are described in this section.

7.1 RunTime

Run time is the amount CPU consumed by either a task or a CPU.

7.1.1 Task RunTime

For a task, the RunTime is measured from the time the process switches on the CPU, until the time it switches off the CPU. For example...

```
    0.328783 cpu=30 pid=0 sched_switch prio=n/a state=n/a next_pid=49554 next_prio=120
    0.328789 cpu=30 pid=49554 sched_switch prio=120 state=SLEEP next_pid=0
next_prio=120
```

Note in the first sched_switch record, PID 49554 is getting switched on to the CPU. So at that point, the process is running. In the 2nd sched_switch record, PID 49554 is switching off the CPU. The time between the 2 sched_switch records is process RunTime, in this case, only 0.000006 seconds.

The CPU time is either considered SysTime or UserTime depending on whether or not the process was in a system call during the time it was running on the CPU. In the above example, we can't really tell if it was SysTime or UserTime.

Consider the following example:

```
    0.063043 cpu=12 pid=0 sched_switch prio=n/a state=n/a next_pid=47830 next_prio=120
    0.063045 cpu=12 pid=47830 futex [202] entry *uaddr=0x11eaf50 op=0x81 val=0x1
*timeout=0x7f95518b6690 *uaddr2=0x11eaf50 val3=0xbad6
    0.063046 cpu=12 pid=47830 futex [202] ret1=0x0 syscallbeg= 0.000001
*uaddr=0x11eaf50 op=0x81 val=0x1 *timeout=0x7f95518b6690 *uaddr2=0x11eaf50 val3=0xbad6
    0.063054 cpu=12 pid=47830 futex [202] entry *uaddr=0x11eaf7c op=0x189 val=0xb665
*timeout=0x7f95518b6690 *uaddr2=0x11eaf50 val3=0xffffffff
    0.063055 cpu=12 pid=47830 sched_switch prio=120 state=SLEEP next_pid=0
next_prio=120
```

The first record is a sched_switch record indicating that the task started to execute on the CPU. The 2nd record is the futex() sys_enter record, which denotes that the process is moving from UserTime to SysTime. The next record is the futex() sys_exit record. The time in between (provided there is no sched_switch record) is SysTime, which is only 0.000001 seconds. Meanwhile, the time between the futex() sys_exit and the next futex() sys_enter is UserTime, or 0.000008 seconds. Lastly, the time between the futex() sys_enter record and the sched_switch when the process goes to SLEEP is SysTime.

The Pid Analysis Report (***kiinfo -kipid***) will provide a summary of the processed CPU time, and keeps track of the SysTime and UserTime:

```
PID 47830  ./mfwServer
  PPID 1  /sbin/init
  TGID 47788  ./mfwServer
  NLWP: 8
    ********* SCHEDULER ACTIVITY REPORT ********
    RunTime     :  0.078536  SysTime    :  0.022994   UserTime    :  0.055543
    SleepTime   : 19.848711  Sleep Cnt :       4935   Wakeup Cnt  :       179
    RunQTime    :  0.009718  PreemptCnt:        153   Switch Cnt  :      5088
    Last CPU    :         12  CPU Migrs :        109   NODE Migrs  :         1
```

The kipid output also keeps trace of the amount of SysTime which is spent in each system call:

```
    ******** SYSTEM CALL REPORT ********
    System Call Name      Count      Rate      ElpTime        Avg        Max Errs
    futex                  9983     500.5    19.877757   0.001991   0.005365    0
       SLEEP               4935     247.4    19.848711   0.004022
       RUNQ                                   0.007042
       CPU                                    0.022007
    ioctl                   184       9.2     0.001278   0.000007   0.000238    0
       RUNQ                                   0.000292
       CPU                                    0.000986
```

These stats are also kept for each system call on a per-file basis.

7.1.2 CPU RunTime

We can also keep track of CPU RunTime. CPUs are either idle, or they are executing a task. CPUs can be processing interrupts while the CPU is idle or executing a task. When a CPU is running, it may be executing in System mode (in a system call) or executing in User mode.

Again, the switch records can be used to see when a CPU transitions from idle to "running". If a CPU is idle, the sched_switch will show pid=0. For example:

```
    1.831090 cpu=0 pid=0 sched_switch prio=n/a state=n/a next_pid=135623 next_prio=120
    1.831094 cpu=0 pid=135623 nanosleep [35] ret1=0x0 syscallbeg= 0.010105
*req=0x7f7d8a1fbe00 *rem=0x0
    1.831097 cpu=0 pid=135623 read [0] entry fd=316 *buf=0x7f7d7401d000 count=4096
    1.831101 cpu=0 pid=135623 read [0] ret1=0x1000 syscallbeg= 0.000003 fd=316
*buf=0x7f7d7401d000 count=4096
    1.831103 cpu=0 pid=135623 nanosleep [35] entry *req=0x7f7d8a1fbe00 *rem=0x0
    1.831107 cpu=0 pid=135623 sched_switch prio=120 state=SLEEP next_pid=135617
next_prio=120
    1.831109 cpu=0 pid=135617 nanosleep [35] ret1=0x0 syscallbeg= 0.010108
*req=0x7f7da61fbe00 *rem=0x0
    1.831110 cpu=0 pid=135617 read [0] entry fd=304 *buf=0x7f7d900bc000 count=4096
    1.831112 cpu=0 pid=135617 read [0] ret1=0x1000 syscallbeg= 0.000002 fd=304
*buf=0x7f7d900bc000 count=4096
    1.831114 cpu=0 pid=135617 nanosleep [35] entry *req=0x7f7da61fbe00 *rem=0x0
    1.831145 cpu=0 pid=135617 sched_switch prio=120 state=SLEEP next_pid=135634
next_prio=120
    1.831152 cpu=0 pid=135634 nanosleep [35] ret1=0x0 syscallbeg= 0.010057
*req=0x7f7d52bfce00 *rem=0x0
    1.831155 cpu=0 pid=135634 read [0] entry fd=338 *buf=0x7f7d38017000 count=4096
    1.831163 cpu=0 pid=135634 read [0] ret1=0x1000 syscallbeg= 0.000008 fd=338
*buf=0x7f7d38017000 count=4096

    1.831164 cpu=0 pid=135634 nanosleep [35] entry *req=0x7f7d52bfce00 *rem=0x0
```

```
    1.831187 cpu=0 pid=135634 sched_switch prio=120 state=SLEEP next_pid=0
next_prio=120
```

In the above example, the CPU was idle (pid=0) when the sched_switch occurred to switch in PID 135623. The CPU then ran PID 135617 and PID 135634 until it became idle again. Between the first sched_switch and the last, the CPU ran for 0.000097 seconds. Some of the time is SysTime and some of the time is UserTime.

The CPU/RunQ Analysis Report (***kiinfo -kirunq***) provides the CPU RunTime on each CPU:

```
Global CPU Counters
cpu node          Total          sys          user          idle
  0 [ 0] :    19.959429     0.126325     0.015291    19.817813
  1 [ 0] :    19.768416     1.270091     1.435211    17.063114
  2 [ 0] :    19.964742     0.102332     0.032494    19.829915
  3 [ 0] :    19.958890     0.024046     0.003190    19.931654
```

7.2 SleepTime

SleepTime is the elapsed time when a process switches off the CPU in SLEEP mode until the time that another process (or interrupt) wakes up the thread. For example:

```
    0.000502 cpu=70 pid=135431 sched_switch prio=120 state=SSLEEP next_pid=0
next_prio=120
    0.000662 cpu=70 pid=0 sched_wakeup target_pid=135431 prio=120 target_cpu=70
success=1
```

In the above case PID 135431 went to sleep at 0.000502 and was woken up by PID 0 (for *ftrace* traces, this is an interrupt while the CPU was idle) at 0.000662. Thus the process was asleep for 0.000160 seconds. When LinuxKI data is collected via *ftrace*, we cannot tell why the process went to sleep. However, we can identify what system call the process was in (provided it was in a system call), but looking at the surrounding records:

```
    0.000479 cpu=70 pid=135431 write [1] entry fd=3 *buf=0x212d000 count=8192
    0.000502 cpu=70 pid=135431 sched_switch prio=120 state=SSLEEP next_pid=0
next_prio=120
    0.000662 cpu=70 pid=0 sched_wakeup target_pid=135431 prio=120 target_cpu=70
success=1
    0.000675 cpu=70 pid=1040 sched_switch prio=120 state=SLEEP next_pid=135431
next_prio=120
    0.000681 cpu=70 pid=135431 sched_wakeup target_pid=135432 prio=120 target_cpu=50
success=1
    0.000684 cpu=70 pid=135431 write [1] ret1=0x2000 syscallbeg= 0.000205 fd=3
*buf=0x212d000 count=8192
```

Based on the above, we can then attribute the SleepTime of 0.000160 seconds to the write() system call. Note the system call took 0.000205 seconds, so the CPU used by the process in the write() system call was 0.000045 seconds.

If the *LiKI* tracing mechanism is used, then the sched_switch record will log a stack trace, which will provide details of the code path leading up to the switch:

```
    0.000812 cpu=60 pid=135431 sched_switch prio=120 state=SSLEEP next_pid=0
next_prio=120 next_tgid=n/a policy=n/a STACKTRACE: io_schedule+0x73
__blockdev_direct_IO+0x70e ext4_direct_IO+0x11e generic_file_direct_write+0xc2
__generic_file_aio_write+0x345 generic_file_aio_write+0x6f ext4_file_write+0x61
do_sync_write+0xfa vfs_write+0xb8 sys_write+0x51 tracesys+0xd9
    0.000968 cpu=60 pid=-1 sched_wakeup target_pid=135431 prio=120 target_cpu=60
success=1
    0.000977 cpu=60 pid=0 sched_switch prio=n/a state=n/a next_pid=135431
next_prio=120 next_tgid=135431 policy=SCHED_NORMAL
    0.000983 cpu=60 pid=135431 sched_wakeup target_pid=135430 prio=120 target_cpu=30
success=1
```

Based on the stack trace, we can tell the process went to sleep waiting on a write to an ext4 file using direct I/O.

The Pid Analysis Report (***kiinfo -kipid***) will report the sleep time for a process, as well as the sleep time for individual system calls. For example:

```
PID 135431  /home/mcr/iotest8
  PPID 133402  -bash
    ********* SCHEDULER ACTIVITY REPORT ********
    RunTime     :  1.518369  SysTime    :  1.465068   UserTime    :  0.053302
    SleepTime   : 17.847719  Sleep Cnt  :     52251   Wakeup Cnt  :     47033
    RunQTime    :  0.599293  Switch Cnt:      53593   PreemptCnt  :      1342
    Last CPU    :        60  CPU Migrs  :       4564   NODE Migrs  :         1

    ******** SYSTEM CALL REPORT ********
    System Call Name      Count      Rate     ElpTime        Avg        Max Errs
    write                 44679   2237.8    15.764693   0.000353   0.047953    0
       SLEEP              49065   2457.5    13.899644   0.000283
          Sleep Func      44679              8.116796   0.000182   0.028745  io_schedule
          Sleep Func       4385              5.782848   0.001319   0.047529
__mutex_lock_slowpath
       RUNQ                                  0.511035
       CPU                                   1.354047
    lseek                 44680   2237.8     4.095907   0.000092   0.074542    0
       SLEEP               3184    159.5     3.947747   0.001240
          Sleep Func       3184              3.947747   0.001240   0.074469
__mutex_lock_slowpath
       RUNQ                                  0.037199
       CPU                                   0.110961
```

Note the Sleep Func is only available with the *LiKI* tracing mechanism.

The Wait Event Analysis Report (kipid -kiwait) provides global wait event information if the *LiKI* tracing mechanism is used:

```
Kernel Functions calling sleep()
   Count     Pct     SlpTime     Slp%   Msec/Slp   MaxMsecs  Func
  101335  38.52%     84.5643   1.45%      0.835     74.469  __mutex_lock_slowpath
   94817  36.04%     30.0701   0.52%      0.317     64.951  io_schedule
   58421  22.21%   3051.7865  52.34%     52.238   7981.784  worker_thread
    4979   1.89%    172.2950   2.96%     34.604   9992.944  schedule_hrtimeout_range
    1048   0.40%    195.1255   3.35%    186.188  14156.308  cpu_buf_read
```

Note that some wait events are normal. It is not unusual to see a lot of SleepTime in the worker_thread function, as the worker threads commonly sleep while waiting for work to do. So you will need to read some source code and obtain some practice to identify what is normal and what is not.

7.3 RunQTime

RunQTime is technically defined as the time when a task is ready to run, but the task is waiting for the CPU. As far as ***kiinfo*** goes, there are 2 situations that it measures. The first case is when a sched_wakeup is done on a process:

```
    8.866769 cpu=14 pid=46973 sched_wakeup target_pid=49480 prio=120 target_cpu=13
success=1
    8.866780 cpu=13 pid=47155 sched_switch prio=120 state=SLEEP next_pid=49480
next_prio=120
```

Note that the wakeup for PID 49480 occurred at 8.866769, but the switch did not occur to 8.866780. In this case, the process spent 0.000011 seconds on the RunQ waiting for PID 47155 to give up the CPU.

The 2nd case occurs when a process is already running, but must give up the CPU due to a higher priority process or it has used up its timeslice. You'll see that sched_switch state is "state=RUN":

```
    3.594266 cpu=30 pid=49480 sched_switch prio=120 state=RUN next_pid=48120
next_prio=120
    3.594279 cpu=30 pid=48120 sched_switch prio=120 state=SLEEP next_pid=49480
next_prio=120
```

In the case above, PID 49480 is said to be "preempted" by PID 48120. Thus PID 49480 was on the CPU Runq for 0.000013 seconds. This is also called an involuntary or "forced" context switch.

The Pid Analysis Report (***kiinfo -kipid***) can provide details on how much time a process spent on the CPU RunQ. It also summarizes the CPU RunQ time for each system call:

```
PID 135431  /home/mcr/iotest8
  PPID 133402  -bash

    ********* SCHEDULER ACTIVITY REPORT ********
    RunTime    :  1.518369  SysTime   :  1.465068   UserTime   :  0.053302
    SleepTime  : 17.847719  Sleep Cnt :     52251   Wakeup Cnt :     47033
    RunQTime   :  0.599293  Switch Cnt:     53593   PreemptCnt :      1342
    Last CPU   :        60  CPU Migrs :      4564   NODE Migrs :         1

    ******** SYSTEM CALL REPORT ********
    System Call Name     Count     Rate     ElpTime        Avg        Max Errs
    write                44679   2237.8   15.764693   0.000353   0.047953    0
       SLEEP             49065   2457.5   13.899644   0.000283
       RUNQ                                0.511035
       CPU                                 1.354047
    lseek                44680   2237.8    4.095907   0.000092   0.074542    0
       SLEEP              3184    159.5    3.947747   0.001240
       RUNQ                                0.037199
       CPU                                 0.110961
```

Sometimes, a task is woken up and the CPU it is scheduled on is idle. While it doesn't really spend time waiting for another process, it's still considered "RunQTime", although the time is very small. If a thread sleeps frequently, we could see some high RunQ times even on a fairly idle system. To distinguish between time actually waiting on another process and the overhead time of switching the process on an idle CPU, two time statistics are introduced. If the rqhist flag is used, then you will get additional RunQ details, including a histogram of RunQ wait times:

```
******** PID runq latency report ********
RunQTime    :  0.599293  RunQCnt    :     53593   AvRunQTime :  0.000011
RunQPri     :  0.421615  RunQPriCt  :     31475   AvRunQPri  :  0.000013
RunQIdle    :  0.177678  RunQIdleCt:     22118   AvRunQIdle :  0.000008

runq latency in Usecs
cpu    <5     <10    <20    <50    <100   <500   <1000  <2000  <10000 <20000 >20000
60   : 604    18028  26222  1228   83     249    2      0      0      0      0
61   : 1648   548    8      11     1      2      0      0      0      0      0
62   : 812    80     2      2      1      0      0      0      0      0      0
63   : 27     0      1      0      3      0      0      0      0      0      0
140  : 3250   195    49     301    11     11     0      0      0      0      0
141  : 163    27     1      0      0      0      0      0      0      0      0
142  : 19     4      0      0      0      0      0      0      0      0      0

runq latency in Usecs
cpu   Avg.   Max    Total_time  Total_cnt  Migrations  LDOM_migr_in  LDOM_migr_out
60    11     507    537286      46416      42          0             0
61    4      248    9171        2218       2           0             0
62    3      52     3075        897        1           0             0
63    8      57     260         31         1           0             0
140   5      337    21947       3817       42          0             0
141   3      10     674         191        0           0             0
142   3      7      78          23         0           0             0
```

In the above output, the process spent 0.421615 seconds on the RunQ waiting for another process to give up the CPU, and another 0.177678 seconds was just the overhead of switching the process onto an idle CPU.

The CPU/RunQ Analysis Report (***kiinfo -kirunq***) provides a global view of the RunQ times.

```
RUNQ latency histogram (in usecs)
 cpu       <5    <10    <20    <50    <100   <500  <1000  <2000 <10000 <20000 >20000
   0 :      3      4     33      6     22      5     0      0      0      0      0
   1 :    152     34     63     52      6     73     1      1      2      1      0
   2 :      0      0      3      6      0      2     0      0      0      0      0
   3 :      0      0      3      8      0      0     0      0      0      0      0

RUNQ latency statistics (in usecs)
 cpu     Avg       Max  Total_time  Total_cnt  Migrations  NODE_migr_in  NODE_migr_out
   0    42.1       101        3075         73           0             0              0
   1   125.7     15224       48412        385           3             0              0
   2    40.2       122         442         11           0             0              0
   3    23.3        28         256         11           0             0              0
```

In the data above, you can identify the average time a process spent on the RunQ on a per-CPU basis as well as the Maximum and Total RunQ wait times.

7.4 StealTime

In some virtual machine environments, a CPU used by a VM guest may be stolen by the VM host to perform work on the host or for other VM guest. This time is known as "stealtime". You can typically see the steal time with tools like top. Not all VM environments maintain the stealtime reported by the VM guest. Presently, KVM is known to maintain the stealtime, whereas VMware does not. Refer also to the following RedHat documentation:

https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/6.3_Release_Notes/virtualization.html

LinuxKI reports the stealtime in each sched_switch record. The stealtime logged is the amount of stealtime since the previous sched_switch record for the same CPU. For example, consider the following sched_switch records for CPU 0:

```
    0.157024 cpu=0 pid=10 tgid=10 sched_switch syscall=[-1] prio=120 state=SLEEP
next_pid=2983 next_prio=120 next_tgid=2983 policy=SCHED_NORMAL vss=0 rss=0 stealtime=
0.000000 rcu_gp_kthread+0x339 kthread+0xcf ret_from_fork+0x7c
    0.177407 cpu=0 pid=2983 tgid=2983 sched_switch syscall=close prio=120 state=RUN
next_pid=10 next_prio=120 next_tgid=10 policy=SCHED_NORMAL vss=29002 rss=387
stealtime= 0.010011 int_careful+0x14 | [libc-2.17.so]:__close_nocancel+0x7
```

Note that PID 2983 was running on the CPU for 20.383 milliseconds (based on the timestamps on the left). However, out of the 20.383 milliseconds, ~10 milliseconds were stolen by the VM host. So the task only received 50% of the CPU time during the ~20 millisecond period.

Kiinfo has been enhanced to report the stealtime on a per-cpu basis (-kirunq) as well as on a per-task basis (-kipid). Below is an example of a PID Analysis Report (kipid) of a task which is impacted by the steal time:

```
PID 2983  -bash
  PPID 2288  -bash
    ********* SCHEDULER ACTIVITY REPORT ********
    RunTime     :  9.681995  SysTime    :  3.047348   UserTime    :  6.634647
    StealTime   :  4.957096
    SleepTime   :  0.000000  Sleep Cnt  :         0   Wakeup Cnt  :       455
    RunQTime    :  0.336201  Switch Cnt:       458   PreemptCnt  :       458
    Last CPU    :         0  CPU Migrs  :         0   NODE Migrs  :         0
    Policy      : SCHED_NORMAL     vss  :     29002          rss  :       387
    busy     :     96.64%
      sys    :     30.42%
      user   :     66.23%
      steal  :     49.48%
    runQ     :      3.36%
```

Note that on a per-task basis, the StealTime is a portion of the RunTime used by the VM host.

On the per-cpu basis, the stealtime is reported on the CPU/RunQ Analysis Report (kirunq):

```
Global CPU Counters

cpu node    Total Busy         sys         usr        idle    stealbusy   stealidle
  0 [ 0] :     100.00%      33.63%      66.37%       0.00%      50.08%       0.00%
```

```
Total          100.00%      33.63%      66.37%       0.00%      50.08%       0.00%
```

The Kparse Report also includes the above information as well as Section 1.2.3 which reports the top tasks with the most stealtime:

```
PID      RunTime     SysTime    UserTime    RunqTime     StealTime Command
2983    9.681995    3.047348    6.634647    0.336201     4.957096  -bash
27587   0.290104    0.287811    0.002293    8.487972     0.040065  kiinfo
623     0.012034    0.000278    0.011755    0.105667     0.010018  /usr/bin/python (tuned)
11      0.026657    0.026657    0.000000    0.248608     0.010010  [rcuos/0]
```

7.5 IdleTime

IdleTime is the amount of time when a CPU is not executing any process. It can simply be measured using the sched_switch records for pid=0:

```
    0.629087 cpu=20 pid=18647 sched_switch prio=100 state=SLEEP next_pid=0
next_prio=120
    0.875684 cpu=20 pid=0 sched_switch prio=n/a state=n/a next_pid=86 next_prio=0
next_tgid=n/a policy=n/a
```

In the example above, CPU 20 was idle for 0.246597 seconds.

The CPU/RunQ Analysis report (***kiinfo -kirunq***) can give additional details on the IdleTime for each CPU:

```
Global CPU Counters
cpu node          Total          sys         user         idle
  0 [ 0] :    19.965773     0.004152     0.000247    19.961374
  1 [ 0] :    19.965773     0.633233     0.816172    18.516368
  2 [ 0] :    19.965773     0.000421     0.000000    19.965352
  3 [ 0] :    19.965773     0.000427     0.000000    19.965346

cpu node     Total Busy          sys          usr         idle
  0 [ 0] :        0.02%        0.02%        0.00%       99.98%
  1 [ 0] :        7.26%        3.17%        4.09%       92.74%
  2 [ 0] :        0.00%        0.00%        0.00%      100.00%
  3 [ 0] :        0.00%        0.00%        0.00%      100.00%
```

You can also get a histogram of how often the CPU is idle for specific time periods:

```
Idle CPU Time Histogram.
Idle time in Usecs
cpu    <10   <20   <50   <100  <250  <500  <750  <1000 <1250 <1500 <2000 <3000 <5000
<10000 <20000 >20000
 0   : 0     0     1     0     5     4     0     0     0     0     0     0     1
1     1     55
 1   : 0     1     1     0     5     0     0     0     0     0     0     0     0
1     0     18
 2   : 0     0     0     0     0     0     0     0     0     0     0     0     0
0     0     11
 3   : 0     0     0     0     0     0     0     0     0     0     0     0     0
0     1     10
...
Note - Idle time in the <10 Usec bucket is likely just context switch time and not
true IDLE time.
```

So in the case above, CPU 0 is idle for 19.961374 seconds, or 99.98% of the time. For each time the CPU is idle, we record the event in the appropriate time bucket. So in the example above, CPU 0 was idle between 20 and 50 usecs only once and was idle between 100 and 250 usecs five times.

While a CPU is idle, it may enter a power saving state. For example:

```
0.875706 cpu=20 pid=86 sched_switch prio=0 state=SLEEP next_pid=0 next_prio=120
0.875715 cpu=20 pid=0 power_start state=3
1.460681 cpu=20 pid=0 power_end
1.460683 cpu=20 pid=0 power_start state=3
1.627585 cpu=20 pid=0 power_end
1.627593 cpu=20 pid=0 sched_switch prio=n/a state=n/a next_pid=18647 next_prio=100
```

The CPU/RunQ Analysis Report will also provide details on the power-states when the CPUs are idle. Note the c-states are calculated during the ***runki*** script if available, unless the power events are enabled during the trace collection.

```
Processor C-States and Power Events

cpu node   Events  Cstate0 Cstate1 Cstate2 Cstate3 freq_changes   freq_hi  freq_low
0   [ 0]      766    0.03%   0.00%   0.00%  99.96%            0         0         0
1   [ 0]      247    7.26%   0.01%   0.05%  92.68%            0         0         0
2   [ 0]      232    0.00%   0.00%   0.00% 100.00%            0         0         0
3   [ 0]      232    0.00%   0.00%   0.00% 100.00%            0         0         0
```

7.6 Advanced CPU statistics (MSR statistics)

The advanced CPU statistics using the Model Specific Registers (MSR) can be collected in the sched_switch records. Capturing the advanced CPU statistics is not enabled by default, but can be captured using “runki -R” or when using kiinfo with the “msr” flag. For example:

```
$ runki -R

$ kiinfo -live msr
$ kiinfo -kipid msr -a 5
```

The advanced CPU statistics are reported with the CPU statistics and the PID statistics. Below is an example of the advanced CPU statistics from a PID report:

```
PID 31432  /home/mcr/bin/iotest8
  PPID 1  /sbin/init

...

    ******** CPU MSR REPORT *******
        LLC_ref   LLC_hits  LLC_hit%   Instrs   Cycles    CPI    Avg_MHz  SMI_cnt
         16753k     15100k    90.13%     808m    2606m   3.22    2484.77        8
```

7.7 HyperThread statistics

Since ***kiinfo*** can detect when a CPU becomes busy and when a CPU becomes idle, it has the unique ability to get statistics on a HyperThreaded (HT) physical CPU (PCPU) regarding the state of each logical CPU (LCPU). Specifically, we can tell if both LCPUs are idle, or if both LCPUs are busy, or if one of the LCPUs is idle while the other LCPU is busy. This can give an indication of the true idleness of the physical CPU (PCPU) as well as provide some guidance regarding the effectiveness of the HyperThreading:

```
Hyper-threading CPU pair status

     PCPU     double idle   lcpu1 busy   lcpu2 busy  double busy
  [  0  80]:    19.834595     0.004399     0.001062     0.000000
  [  1  81]:    16.895826     1.449170     1.381013     0.000235
  [  2  82]:    19.722444     0.000421     0.000578     0.000000
  [  3  83]:    19.719442     0.000427     0.000494     0.000000

     PCPU     double idle   lcpu1 busy   lcpu2 busy  double busy
  [  0  80]:       100.0%         0.0%         0.0%         0.0%
  [  1  81]:        85.7%         7.3%         7.0%         0.0%
  [  2  82]:       100.0%         0.0%         0.0%         0.0%
  [  3  83]:       100.0%         0.0%         0.0%         0.0%
```

In the example above, note that CPU 1 is 7.3% busy while CPU 81 is idle, and CPU 81 is 7.0 % busy while LCPU 1 is idle. Neither LCPU 1 or LCPU 81 are busy at the same time. The two LCPUs are part of the same Physical CPU (PCPU). While each LCPU is idle 93% of the time, the PCPU is only 85.7% idle. The key point is that HyperThreading makes you believe there is more idle CPU than you really have.

7.8 CPU Migrations

Periodically, a process may migrate from one CPU to another. On a NUMA system, the process may migrate from a CPU on one node to a CPU on another node.

```
    0.369161 cpu=140 pid=135426 sched_switch prio=120 state=SSLEEP next_pid=0
next_prio=120
    0.389303 cpu=70 pid=135428 sched_wakeup target_pid=135426 prio=120 target_cpu=71
success=1
    0.389308 cpu=71 pid=0 sched_switch prio=n/a state=n/a next_pid=135426
next_prio=120
    0.389333 cpu=71 pid=135426 sched_switch prio=120 state=SSLEEP next_pid=0
next_prio=120
```

In the example above, PID 135426 was running on CPU 140 (which is on node 6), but as switched to CPU 71 (which is on node 7). So we have a CPU Migration as well as a Node Migration. The PID Analysis report (***kiinfo -kipid***) will keep track of the number of migrations for each process:

```
PID 135426  /home/mcr/iotest8
  PPID 133402  -bash

    ********* SCHEDULER ACTIVITY REPORT ********
    RunTime    :  0.175019  SysTime   :  0.171910   UserTime   :  0.003109
    SleepTime  : 19.640606  Sleep Cnt :     21868   Wakeup Cnt :      2509
    RunQTime   :  0.128806  Switch Cnt:     21895   PreemptCnt :        27
    Last CPU   :        60  CPU Migrs :      1519   NODE Migrs :         3
```

7.9 System Call Statistics

The elapsed time for a system call must be calculated using the sys_enter record and the sys_exit record. ***Kiinfo -kitrace*** will calculate this automatically.

```
    0.000792 cpu=60 pid=135431 write [1] entry fd=3 *buf=0x212d000 count=8192
    0.000985 cpu=60 pid=135431 write [1] ret1=0x2000 syscallbeg= 0.000193 fd=3
*buf=0x212d000 count=8192
```

In the above example, the system call elapsed time is 0.000193 seconds. Note that the task could be running while in the system call (CPU), or it would be waiting for CPU (RUNQ), or it could be sleeping (SLEEP). You must examine the records in between the sys_enter and sys_exit records to know.

The PID Analysis Report (***kiinfo -kipid***) will provide the system call details for the process along with the elapsed time. ***Kiinfo*** will also break down the system call to see how the time is spent:

```
    ******** SYSTEM CALL REPORT ********
    System Call Name      Count      Rate      ElpTime        Avg        Max Errs
    write                 44679    2237.8    15.764693   0.000353   0.047953    0
       SLEEP              49065    2457.5    13.899644   0.000283
          Sleep Func      44679              8.116796   0.000182   0.028745  io_schedule
          Sleep Func       4385              5.782848   0.001319   0.047529
__mutex_lock_slowpath
       RUNQ                                  0.511035
       CPU                                   1.354047
    lseek                 44680    2237.8     4.095907   0.000092   0.074542    0
       SLEEP               3184     159.5     3.947747   0.001240
          Sleep Func       3184               3.947747   0.001240   0.074469
__mutex_lock_slowpath
       RUNQ                                  0.037199
       CPU                                   0.110961
```

7.10 File Statistics

The file statistics are measured using the system call entry and exit records. In the example below, the write() system call is shown with its arguments:

```
    0.062637 cpu=8 pid=31548 tgid=31548 write [1] entry fd=3 *buf=0x1379000 count=8192
    0.062885 cpu=8 pid=31548 tgid=31548 write [1] ret=0x2000 syscallbeg=    0.000248
fd=3 *buf=0x1379000 count=8192 type=REG dev=0x0fd0000b ino=131334175
```

Note the raw LinuxKI binary data doesn't provide a filename, just a file descriptor (fd=3). If you are using the *LiKI* tracing mechanism, you do get the inode and device value. Also, if you are using *LiKI*, the filename may be shown in the sys_enter trace event for the open() system call KI:

```
    6.042312 cpu=16 pid=13059 tgid=13059 open [2] entry *pathname=0x7fc0968b5d97
flags=0x0 mode=0666 filename: /etc/mtab
    6.042315 cpu=16 pid=13059 tgid=13059 open [2] ret=10 syscallbeg=    0.000003
*pathname=0x7fc0968b5d97 flags=0x0 mode=0666
```

In the above example, the ret1=10 is the file descriptor returned by the open() call. Note that we get a pointer to the pathname, but the actual pathname string is only retrieved if the *LiKI* tracing mechanism is used.

However, the file's pathname or the open() system call may not be in the LinuxKI trace data itself. The ***kiinfo*** tool may attempt to extract the filenames for a process from the lsof output collected and saved into the ki_all.*.tgz file. This works as long as the file was opened prior to running lsof, and the file descriptor is not closed and re-used for another file open. So the filename may not always be accurate. In some cases, the filename will remain unknown because the file was not opened before lsof was executed or the open() system call was missed.

For many tasks, such as Oracle processes, the names of the files are static. By collecting system call statistics and potentially switch events for each file, ***kiinfo*** can report on the activity of a file for a single process or globally. Note that the PID Analysis report (***kiinfo -kipid***) will show the file activity for a single process:

```
    ******** FILE ACTIVITY REPORT ********
    FD: 3 REG      dev: 0xfd00007 /mcr1/bigfile
    System Call Name     Count      Rate     ElpTime        Avg        Max Errs
    write                44679   2237.8   15.764693   0.000353   0.047953    0
       SLEEP             49065   2457.5   13.899644   0.000283
          Sleep Func     44679             8.116796   0.000182   0.028745  io_schedule
          Sleep Func      4385             5.782848   0.001319   0.047529
__mutex_lock_slowpath
       RUNQ                               0.511035
       CPU                                1.354047
    lseek                44680   2237.8    4.095907   0.000092   0.074542    0
       SLEEP              3184    159.5    3.947747   0.001240
          Sleep Func      3184             3.947747   0.001240   0.074469
__mutex_lock_slowpath
       RUNQ                               0.037199
       CPU                                0.110961
```

Note that the Sleep Func is only available with the *LiKI* tracing mechanism.

The File Activity Report (***kiinfo -kifile***) shows a file's activity across all tasks on the system. There will be a general summary showing read and write activity followed by a more detailed listing for each file:

```
******** GLOBAL FILE ACTIVITY REPORT ********

---  Top Files sorted by System Call Count  ---
Syscalls   ElpTime   Lseeks  Reads  Writes Errs                dev      node     type
Filename
  189582  119.4400   94793    5650   89139    0 0x000000000fd00007        15      REG
/mcr1/bigfile
    8591    0.0099       0    4301       0    0 0x000000000000000a         1      DIR
inotify
     792    0.0020       0     792       0    0 0xffff890ca80e9c80     49894     unix
socket
     351    0.0010       0     176     175    0 0x000000000f800001     31112      CHR
/dev/hpilo/d0ccb1
      64    0.0002       0      32      32    0 0x000000000f800003     31114      CHR
/dev/hpilo/d0ccb3
      36    0.0295       6       0       0    0 0x0000000000000003         1      DIR
/proc
      24    0.0001       0      12      12    0 0x000000000f800005     31116      CHR
/dev/hpilo/d0ccb5
```

```
      20    0.0000       0       0       0     0 0x0000000000a00082     31165      CHR
/dev/watchdog
      16    0.0002       0       0       0     0 0x0000000000009103         2     IPv4
*:161

---  Top Files sorted by System Call Count (Detailed)  ---

device: 0x000000000fd00007 node:      15 fstype:      REG  filename:  /mcr1/bigfile
System Call Name     Count     Rate     ElpTime        Avg        Max Errs    AvSz
KB/s
lseek                94793   4747.8   45.416914   0.000479   0.074548    0
read                  5650    283.0   42.605642   0.007541   0.076053    0
write                89139   4464.6   31.417454   0.000352   0.047953    0

device: 0x000000000000000a node:       1 fstype:      DIR  filename:  inotify
System Call Name     Count     Rate     ElpTime        Avg        Max Errs    AvSz
KB/s
read                  4301    215.4    0.005949   0.000001   0.000335    0
ioctl                 4290    214.9    0.003947   0.000001   0.000283    0
```

7.11 Block IO statistics

As mentioned before under the discussion of the Block IO traces, there are 3 primary trace events:

block_rq_insert – logged when the IO is queued to the block device queue.
block_rq_issue - logged when the IO is sent to the IO layers below the block device.
block_rq_complete – logged when the block device is notified that the physical IO is complete.

Key statistics include:

- queue time (qtm) - The queue time is the time between the insert and issue.
- service time (svtm) - The service time is the time between the issue and complete.
- queue len (qlen) - The number of requests on the queue waiting to be issued.
- average queue (avque)- The number of requests on the disk's queue waiting to be issued.

The queue times and service times are not logged as part of the LinuxKI trace data. So ***kiinfo*** must calculate them based on the timestamps of the records. If the block_rq_insert or block_rq_issue record is not captured for a complete record, then the IO is "tossed" and not accounted for in the statistics. The block_rq_insert and/or block_rq_issue records must be tracked in order to know which PID inserted or issued the request and how long the request sat on the block device queue or how long it took to complete the request. This is not captured on the completion record either.

Note that the times are measured at the block device layer. Which means we get times for the device-mapper devices as well as scsi block devices. However, LVM is a pseudo-block device and doesn't log any trace data.

The Pid Analysis Report (***kiinfo -kipid***) will report on the IO performed by a given process:

```
******** PHYSICAL DEVICE REPORT ********
    device   rw  avque avinflt   io/s   Kb/s  avsz   avwait   avserv
  0x00800020  /dev/sdc
  0x00800020  r   0.00    0.00      0      0     0     0.00     0.00
```

```
      0x00800020  w   0.50    8.61   1323 473637   357     0.00     4.92
      0x00800020  t   0.50    8.61   1323 473637   357     0.00     4.92

    ******** DEVICE-MAPPER REPORT ********
        device   rw  avque avinflt   io/s   Kb/s  avsz   avwait   avserv
0x0fd00003   /dev/mapper/mpathep  ->  /dev/dm-3
      0x0fd00003  r   0.00    0.00      0      0     0     0.00     0.00
      0x0fd00003  w   2.05    8.93   1342 479208   357     0.22     5.03
      0x0fd00003  t   2.05    8.93   1342 479208   357     0.22     5.03
    Totals:
      Physical Writes:
        Cnt   :  13234   Total Kb: 4736470.5    ElpTime:  65.07809
        Rate  : 1323.4   Kb/sec  :  473636.9    AvServ :   0.00492
        Errs:        0   AvgSz   :     357.9    AvWait :   0.00000
        Requeue :         0   MaxQlen :         1
```

Note that the IO totals for the device does not include the device-mapper statistics, since these are not real physical IOs.

The Disk Analysis Report (***kiinfo -kidsk***) provides the same information on a global basis.

```
Physical Device Statistics

      device rw  avque avinflt   io/s   Kb/s  avsz   avwait   avserv
  0x00800080  /dev/sdi  (HW path: 2:0:0:6)    (mpath device: /dev/mapper/mpathef)
  0x00800080  r   0.50    0.00     82   1982    24     0.00     0.52
  0x00800080  w   0.00    0.00      0      0     0     0.00     0.00
  0x00800080  t   0.50    0.00     82   1982    24     0.00     0.52
0x041000b0  /dev/sdab  (HW path: 2:0:1:6)    (mpath device: /dev/mapper/mpathef)
  0x041000b0  r   0.50    0.00     82   1958    23     0.00     0.53
  0x041000b0  w   0.50    0.00      0      2    16     0.00     0.43
  0x041000b0  t   0.50    0.00     82   1960    23     0.00     0.53
0x04200080  /dev/sdao  (HW path: 3:0:0:6)    (mpath device: /dev/mapper/mpathef)
  0x04200080  r   0.50    0.00     82   1994    24     0.00     0.52
  0x04200080  w   0.50    0.00      0      2    16     0.00     0.31
  0x04200080  t   0.50    0.00     82   1996    24     0.00     0.52
0x043000a0  /dev/sdbg  (HW path: 3:0:1:6)    (mpath device: /dev/mapper/mpathef)
  0x043000a0  r   0.50    0.00     82   1970    24     0.01     0.53
  0x043000a0  w   0.50    0.00      0      3    16     0.00     0.35
  0x043000a0  t   0.50    0.00     82   1974    24     0.01     0.53

Mapper Device Statistics

      device rw  avque avinflt   io/s   Kb/s  avsz   avwait   avserv
0x0fd0000a   /dev/mapper/mpathef  ->  /dev/dm-10
  0x0fd0000a  r   0.50    0.19    327   7905    24     0.01     0.56
  0x0fd0000a  w   0.50    0.00      0      6    16     0.02     0.39
  0x0fd0000a  t   0.50    0.19    327   7911    24     0.01     0.56
```

The Disk Analysis report will also show IO statistics by FC HBA path, which can help troubleshoot performance issues with specific paths in case of a SAN or switch issue:

```
Multipath FC HBA Statistics

      HBA    rw  avque avinflt   io/s   Kb/s  avsz   avwait   avserv
2:0:0
       2:0:0  r   0.00    0.00   1359  32713    24     0.00     0.53
```

```
       2:0:0  w   0.00    0.00      0      0     0     0.00     0.00
       2:0:0  t   0.00    0.00   1359  32713    24     0.00     0.53
2:0:1
       2:0:1  r   0.00    0.00   1359  33012    24     0.00     0.53
       2:0:1  w   0.00    0.00      1      5     8     0.00     0.37
       2:0:1  t   0.00    0.00   1359  33017    24     0.00     0.53
3:0:1
       3:0:1  r   0.00    0.00   1359  33009    24     0.00     0.53
       3:0:1  w   0.00    0.00      0      6    16     0.00     0.35
       3:0:1  t   0.00    0.00   1359  33015    24     0.00     0.53
3:0:0
       3:0:0  r   0.00    0.00   1359  32866    24     0.00     0.53
       3:0:0  w   0.00    0.00      0      4    10     0.00     0.28
       3:0:0  t   0.00    0.00   1359  32870    24     0.00     0.53
```

For LinuxKI dumps collected with the ***runki*** script, the HBA statistics are only gathered if multipath is configured.

With LinuxKI 5.6, the Disk Analysis Report will also show I/O statistics based on the Target Path WWN. Along with the FC HBA statistics, the WWN statistics can be very helpful when troubleshooting SAN related issues:

```
Target WWN Statistics

  FC Target WWN      rw  avque avinflt   io/s   KB/s  avsz   avwait   avserv
0x20510002ac020f7e
  0x20510002ac020f7e  r   0.00   15.81    143 188202  1318   118.67    59.98
  0x20510002ac020f7e  w   0.00   16.00      0     26   256    49.47     0.91
  0x20510002ac020f7e  t   0.00   15.81    143 188227  1317   118.62    59.94
0x20520002ac020f7e
  0x20520002ac020f7e  r   0.00   15.90     65  19390   300   172.97    55.61
  0x20520002ac020f7e  w   0.00   16.00      0     13   256   380.32     1.78
  0x20520002ac020f7e  t   0.00   15.90     65  19402   300   173.13    55.56
```

7.12 Power statistics

By default, the power_start and power_end events are disabled. If c-states are enabled on the system, the ***runki*** script will collect the equivalent statistics reported by ***kiinfo -kirunq*** and ***kiinfo -kparse*** using the /sys/devices/system/cpu/<CPU>/cpuidle/state<N>/time files. If the files are not available in sysfs, then power statistics will not be reported unless the power subsys events are enabled. To enable the power events, refer to the Tracing non-default events section.

When power events are enabled, the amount of CPU time spent in each c-state is calculated using the power events. For example:

```
0.000079 cpu=11 pid=0 power_start state=3
0.000597 cpu=11 pid=0 power_end
```

In the example above, CPU 11 spent 518 microseconds in cstate 3. Note that when the power_end is logged, the processor enters cstate 0 (non-idle cstate).

The CPU/Runq Analysis Report (***kiinfo -kirunq***) as well as the Kparse Report (***kiinfo -kparse***) will report the amount of time each CPU spends in each cstate, as well as statistics based on the power_freq records:

```
cpu node   Events  Cstate0 Cstate1 Cstate2  Cstate3 freq_changes   freq_hi  freq_low
0   [ 0]    76961   12.23%   2.48%  64.91%   20.38%             0         0         0
1   [ 0]    49939   12.70%   1.77%  49.13%   36.40%             0         0         0
2   [ 0]    45797    3.70%   5.51%  67.95%   22.85%             0         0         0
3   [ 0]    47537    4.85%   6.62%  48.21%   40.32%             0         0         0
4   [ 0]    12445    4.62%   3.05%   9.96%   82.36%             0         0         0
5   [ 0]    30570    9.25%   3.04%  43.33%   44.39%             0         0         0
6   [ 0]     9600    0.75%   1.11%  11.59%   86.54%             0         0         0
7   [ 0]    10704    5.95%   1.42%  15.05%   77.58%             0         0         0
8   [ 0]     3756    1.59%   0.01%   3.04%   95.36%             0         0         0
9   [ 0]    19293    4.15%   2.09%  21.22%   72.53%             0         0         0
```

CPUs running at lower frequencies and spending more time in the higher cstates (cstates 2 and 3) indicates that the system is in a power savings mode. Power savings mode can help reduce costs by reducing power consumption, but can reduce the throughput by increasing the latency to transition from an idle state to a running state. IO intensive workloads often perform better when the system is in High Performance mode rather than Power Savings more. Consider switching to High Performance mode if maximum performance is desired, or use tuned-adm to use the latency-performance or network-latency profiles.

7.13 IRQ statistics

The IRQ events can be collected using the *LiKI* tracing mechanism or the *ftrace* tracing mechanism. Interrupts can be handled in 2 phases. The "hard" IRQ events are considered low-level interrupt handlers which process the initial interrupt from the hardware. However, since a CPU can only handle one hard interrupt at a time, the low-level interrupt handler will do the minimum amount of work before handing off the processing to the higher level soft interrupt handler. Note that hardware interrupts are disabled while in a "hard" IRQ handler, but they are enabled during a "soft" IRQ handler. The irq trace events are:

- **irq_handler_entry** - logged on entry to a hard interrupt handler (IRQ). Identifies the IRQ and name of the irq_handler (typically the driver name)
- **irq_handler_exit** - logged on exit to a hard interrupt handler. The time between the entry and exit traces is the hard interrupt time.
- **softirq_raise** - logged when a softirq event is generated. Identifies the softirq vector number and its associated action.
- **softirq_entry** - logged when the softirq handler is entered
- **softirq_exit** - logged when the softirq handler is exited. The time between the softirq_entry and softirq_exit is the soft interrupt time.

Below is an example of the IRQ events...

```
0.000437 cpu=1 pid=0 irq_handler_entry irq=57 name=qla2xxx
0.000444 cpu=0 pid=0 softirq_raise vec=4 action=BLOCK
0.000444 cpu=1 pid=0 irq_handler_exit irq=57 ret=handled irqtm=0.000007
0.000445 cpu=0 pid=0 softirq_entry vec=4 action=BLOCK
```

```
    0.000455 cpu=0 pid=0 softirq_raise vec=4 action=BLOCK
    0.000457 cpu=0 pid=0 softirq_exit vec=4 action=BLOCK irqtm=0.000008
```

Note these IRQ records provide more accurate detail about the IRQ processing. Using the events, the tools can identify the following attributes:

- amount of time CPU spent doing hardirq and softirq processing
- amount of time that a process has been interrupted to handle the softirq and hardirqs.
- amount of time attributed to each softirq and hardirq handler. This can be reported globally and per CPU.

The CPU/RunQ Activity Report (***kiinfo -kirunq***) will include additional fields to report the interrupt processing time for each CPU if the irq events are enabled. For example:

```
Global CPU Counters

cpu node          Total          sys         user         idle  hardirq_sys
hardirq_user hardirq_idle  softirq_sys softirq_user softirq_idle
  0 [ 0] :     5.000000     0.429763     0.151368     4.283991     0.000014
0.000008     0.000227     0.001270     0.004863     0.128498
  1 [ 1] :     4.999826     0.352949     0.019630     4.411018     0.000535
0.000040     0.114470     0.002599     0.000179     0.098407
  2 [ 2] :     5.000000     0.001835     0.000752     4.996619     0.000000
0.000000     0.000000     0.000046     0.000056     0.000692
  3 [ 3] :     4.883063     0.001715     0.003807     4.876483     0.000000
0.000000     0.000000     0.000000     0.000022     0.001035
```

The fields represent the amount of interrupt time that occurred when the CPU was interrupted in system code (hardirq_sys / softirq_sys), the amount of time when the CPU was interrupted while executing user code (hardirq_user / softirq_user), and the amount of time when the CPU was interrupted it was idle (hardirq_idle / softirq_idle).

The CPU/RunQ Activity report also provides details on each time of IRQ event and how much time was spent in each event. For example:

```
Hard IRQ Events
===============
IRQ Name                Count      ElpTime
 57 qla2xxx             17554     0.115051
 70 eth0                  180     0.000181
 23 ehci_hcd:usb1          12     0.000067
    Total:              17746     0.115299

 Soft IRQ events
===============
IRQ Name                Count      ElpTime
  4 BLOCK               35109     0.228948
  1 TIMER                2127     0.004671
  7 SCHED                 457     0.004227
  3 NET_RX                180     0.003217
  9 RCU                  2120     0.002584
  8 HRTIMER                40     0.000085
  2 NET_TX                  2     0.000010
  6 TASKLET                 5     0.000009
```

```
    Total:              40040      0.243749
```

The IRQ data is also reported by CPU as well. The Kparse System Analysis report (***kiinfo -kparse***) will report the information above as well.

The PID Analysis Report (***kiinfo -kipid***) has also been updated to show the amount of time a process/task has been interrupted. Note that interrupt time is attributed to the process that was running at the time of the interrupt. Being able to breakout the IRQ time can help show how a process is impacted by the interrupt processing.

```
PID 15572  /home/mcr/bin/iotest8
  PPID 1  /sbin/init

    ********* SCHEDULER ACTIVITY REPORT ********
    RunTime     :  1.478732  SysTime    :  1.400001   UserTime    :  0.073428
    SleepTime   : 17.883474  Sleep Cnt  :     37491   Wakeup Cnt  :     29503
    RunQTime    :  0.602830  Switch Cnt:      37504   PreemptCnt  :        13
    HardIRQ     :  0.000114  HardIRQ-S  :  0.000098    HardIRQ-U  :  0.000016
    SoftIRQ     :  0.005190  SoftIRQ-S  :  0.004943    SoftIRQ-U  :  0.000247
    Last CPU    :         2  CPU Migrs  :       305   NODE Migrs  :         9
    Policy      : SCHED_NORMAL     vss  :      1015          rss  :       149
```

7.14 Stack Traces

Stack traces are only available with the *LiKI* tracing mechanism. The *LiKI* trace code will attempt to create a stack trace using the same tracing mechanism used by the perf tool. Only hardclock, sched_switch and sched_migrate_task records produce a stack trace. The stack trace produced by the switch record is for the task that called switch. Below are examples of stack traces from the switch and hardclock records:

```
    0.064192 cpu=60 pid=135432 hardclock state=SYS STACKTRACE:
ext4_ext_find_extent+0xc9 ext4_ext_find_extent+0x130 ext4_ext_get_blocks+0x12c
dump_trace+0x190 __perf_event_task_sched_out+0x36 start_this_handle+0xe5
ext4_get_blocks+0x7a ext4_get_block_dio_write+0x81 __blockdev_direct_IO+0x872
ext4_direct_IO+0x11e ext4_get_block_dio_write+0x0 ext4_end_io_dio+0x0
generic_file_direct_write+0xc2 __generic_file_aio_write+0x345 try_to_wake_up+0xca
generic_file_aio_write+0x6f

    0.000204 cpu=60 pid=135432 sched_switch prio=120 state=SSLEEP next_pid=0
next_prio=120 next_tgid=n/a policy=n/a STACKTRACE: io_schedule+0x73
__blockdev_direct_IO+0x70e ext4_direct_IO+0x11e generic_file_direct_write+0xc2
__generic_file_aio_write+0x345 generic_file_aio_write+0x6f ext4_file_write+0x61
do_sync_write+0xfa vfs_write+0xb8 sys_write+0x51 tracesys+0xd9
```

The LinuxKI binary data only has the address of the function. The ***kiinfo*** tool uses the /proc/kallsyms file saved in the ki_all.*.tgz file to translate the kernel code addresses into a function name + offset.

Note that due to the way Linux stack unwinding works, some functions that appear in the stack trace are not valid. In the example above, the functions highlighted in blue are likely from a prior function call or just miscellaneous addresses on the stack. The stack traces are thus considered "inexact" stack traces. This is especially true for SLES systems. So be careful and use the Linux source code to verify the code paths.

The switch Sleep Func is the first function+offset listed in the stack trace. This is the function which called switch (in most cases other than SLES). For the hardclock event, the first function listed is the function that was executing when the clock interrupt occurred.

Profiles of the stack traces are provided in the PID Analysis Report (***kiinfo -kipid***), the CPU Profile Report (***kiinfo -kiprof***), and the Wait Event Analysis Report (***kiinfo -kiwait***). The ***kiinfo*** tool will try to build a profiles of stack traces and how often each stack trace was observed. For example, the CPU Profile Report provides details similar to the following:

```
   Count     Pct  Stack trace
============================================================
     764   5.45%  finish_task_switch
     405   2.89%  scsi_request_fn  scsi_request_fn  __blk_run_queue  elv_insert
__elv_add_request  blk_insert_cloned_request  dm_dispatch_request  dm_request_fn
lock_timer_base  __blk_run_queue  cfq_insert_request  elv_insert  __elv_add_request
__make_request  dm_request  generic_make_request
     265   1.89%  _spin_unlock_irqrestore  qla24xx_start_scsi  scsi_done
qla24xx_dif_start_scsi  mempool_alloc_slab  mempool_alloc  scsi_setup_fs_cmnd
sd_prep_fn  scsi_done  qla2xxx_queuecommand  scsi_dispatch_cmd  scsi_request_fn
__blk_run_queue  elv_insert  __elv_add_request  blk_insert_cloned_request
     252   1.80%  _spin_unlock_irqrestore  try_to_wake_up  wake_up_process
__mutex_unlock_slowpath  mutex_unlock  generic_file_aio_write  ext4_file_write
fsnotify_add_notify_event  do_sync_write  fsnotify  autoremove_wake_function
native_sched_clock  audit_syscall_entry  security_file_permission  vfs_write
sys_write
     196   1.40%  clear_page_c  clear_huge_page  do_huge_pmd_anonymous_page
handle_mm_fault  __do_page_fault  native_sched_clock  sched_clock  do_page_fault
page_fault
     170   1.21%  scsi_request_fn  scsi_request_fn  __blk_run_queue  elv_insert
__elv_add_request  blk_insert_cloned_request  dm_dispatch_request  dm_request_fn
cfq_service_tree_add  __blk_run_queue  cfq_insert_request  elv_insert
__elv_add_request  __make_request  dm_request  generic_make_request
```

The stack traces provide additional detail on what a process or CPU was doing when it was running, or what a process was waiting on when it was waiting.

7.14.1 RunQ stack traces

With LinuxKI 5.0, the Pid Analysis Report (kiinfo -kipid) will report the stack traces that were logged on the sched_switch trace events when a tasked switched off the CPU but remained on the CPU RunQ (state=RUN). The stack traces were already reported in the KI ASCII traces:

```
   13.927148 cpu=6 pid=4315 tgid=4315 sched_switch syscall=open prio=120 state=RUN
next_pid=4368 next_prio=98 next_tgid=4368 policy=SCHED_RR vss=418208555 rss=3800
kmem_cache_alloc_trace+0x3c single_open+0x33 proc_single_open+0x1b
do_dentry_open+0x1a7 vfs_open+0x39 do_last+0x1ed path_openat+0xc2 do_filp_open+0x4b
do_sys_open+0xf3 sys_open+0x1e tracesys+0xdd |  [libpthread-
2.17.so]:__open_nocancel+0x7  [oracle]:skgptalive+0xf3  [oracle]:ksopid_alive+0x5d
[oracle]:ksupalv+0x40  ...
```

The Pid Analysis Report will show the stack traces associated with all the sched_switch events with state=RUN. These stack traces can be used to supplement the data from the hardclock records as it

provides details on which code path was executing when the task had to give up the CPU involuntarily. For example:

```
PID 4315  ora_pmon_SCANIA
...
    Process RunQ stack traces (sort by count) - Top 20 stack traces
       count  Stack trace
    ================================================================
          13   kmem_cache_alloc_trace  single_open  proc_single_open  do_dentry_open
vfs_open  do_last  path_openat  do_filp_open  do_sys_open  sys_open  tracesys  |
__open_nocancel  skgptalive  ksopid_alive  ksupalv
          10   mmput  do_task_stat  proc_tgid_stat  proc_single_show  seq_read
vfs_read  sys_read  tracesys  |  __read_nocancel  ksopid_alive  ksupalv
ksuxfd_check_dead  ksuxfd  ksuxfl  ksucln_dpc_main
           3   __fput  ____fput  task_work_run  do_notify_resume  int_signal  |
__close_nocancel  skgptalive  ksopid_alive  ksupalv  ksuxfd_check_dead  ksuxfd  ksuxfl
ksucln_dpc_main  ksucln_dpc  ksucln
           3   kmem_cache_alloc  getname_flags  getname  do_sys_open  sys_open
tracesys  |  __open_nocancel  skgptalive  ksopid_alive  ksupalv  ksuxfd_check_dead
ksuxfd  ksuxfl  ksucln_dpc_main  ksucln_dpc
```

7.14.2 User stack traces

If LiKI tracing is used, LinuxKI will attempt to dump the User stack trace along with the kernel stack trace. For example:

```
    0.000147 cpu=152 pid=16042 tgid=16042 sched_switch syscall=semop prio=120
state=SLEEP next_pid=0 next_prio=120 next_tgid=n/a policy=n/a vss=102967642 rss=7756
sys_semtimedop+0x6a8 sys_semop+0x10 tracesys+0xd9 | [libc-2.12.so]: __GI_semop+0x7
[oracle]: skgpwwait+0xc8 kslges+0x4f2 kslgetl+0x2cc ksqgtlctx+0x50f ktaiam+0x2bf
ktagetp_internal+0x140 ktaadm+0xf0 kksfbc+0x87c opiexe+0x93c opipls+0x7d2 opiodr+0x48d
rpidrus+0xce skgmstack+0x90 rpiswu2+0x2d3 rpidrv+0x622 psddr0+0x1de psdnal+0x27c
pevm_EXECC+0x130 0x7f02e03e8e6d pevm_NCAL+0x3e pfrinstr_XCAL+0x18e pfrrun_no_tool+0x3c
pfrrun+0x483 plsql_run+0x2c4 peicnt+0x11d kkxexe+0x2d6 opiexe+0x4c24 opiodr+0x48d
ttcpip+0xa8b opitsk+0x6c6 opiino+0x3b1 opiodr+0x48d opidrv+0x24b sou2o+0x91
opimai_real+0x9a ssthrdmain+0x19c main+0xec [libc-2.12.so]: __libc_start_main+0xfd
```

In order to get symbolic names, ***kiinfo*** must open the user library or binary and the user library or binary must not be stripped of its symbols. For a LinuxKI dump collected with the ***runki*** script, a copy of the binary and library symbol tables is captured in the ki_all.*.tgz archive file and used with post-processing.

7.14.3 Important notes about stack traces

While stack traces are very helpful, it is one of the most difficult parts of tracing as the various Linux distributions handle the stack tracing differently. Below are some helpful notes about the stack tracing functionality:

- Stack traces are only available with the *LiKI* tracing module
- Stack tracing seems to work best on RHEL and Ubuntu and related distributions (such as CentOS and OEL)

- Some distributions, such as SLES, have VERY inexact stacktraces. As a result, there are a number of stale functions in the kernel stack traces that make understanding the stack traces very difficult. Use the stacks.<ts> file to help supplement some frequent stack traces.
- In the case of stripped libraries or binaries, only the hex code address is displayed instead of the function name.
- For hardclock traces, if the trace event occurs in user space, only the interrupted function is dumped. However, if the trace event occurs in system space, it may be possible to get the entire user stack trace.

7.15 Cooperating / competing tasks

Sometimes, tasks will cooperate or compete with another task. One example of cooperating tasks is when one task may read from a FIFO, while another task writes to the FIFO. The reader task may wait for a message to be written to the FIFO before it progresses. An example of competing processes may be when one task holds a mutex or semaphore that another task needs.

Below is an example of an lseek() call which is going to sleep a couple of times waiting on a kernel mutex:

```
    0.003821 cpu=60 pid=135426 lseek [8] entry fd=3 offset=0x1b4fc000 whence=0
    0.003823 cpu=60 pid=135426 sched_switch prio=120 state=SSLEEP next_pid=0
next_prio=120 next_tgid=n/a policy=n/a STACKTRACE: __mutex_lock_slowpath+0x13e
mutex_lock+0x2b ext4_llseek+0x60 vfs_llseek+0x3a sys_lseek+0x66 tracesys+0xd9
    0.012042 cpu=30 pid=135430 sched_wakeup target_pid=135426 prio=120 target_cpu=61
success=1
    0.012049 cpu=61 pid=0 sched_switch prio=n/a state=n/a next_pid=135426
next_prio=120 next_tgid=n/a policy=n/a
    0.012058 cpu=61 pid=135426 sched_switch prio=120 state=SSLEEP next_pid=0
next_prio=120 next_tgid=n/a policy=n/a STACKTRACE: __mutex_lock_slowpath+0x13e
mutex_lock+0x2b ext4_llseek+0x60 vfs_llseek+0x3a sys_lseek+0x66 tracesys+0xd9
    0.012244 cpu=60 pid=135432 sched_wakeup target_pid=135426 prio=120 target_cpu=61
success=1
    0.012247 cpu=61 pid=0 sched_switch prio=n/a state=n/a next_pid=135426
next_prio=120 next_tgid=n/a policy=n/a
    0.012249 cpu=61 pid=135426 sched_wakeup target_pid=135431 prio=120 target_cpu=62
success=1
    0.012250 cpu=61 pid=135426 lseek [8] ret1=0x1b4fc000 syscallbeg= 0.008428 fd=3
offset=0x1b4fc000 whence=0
```

In the above example, PID 135426 goes to sleep on a mutex twice. Once time it is woken up once by PID 135430 and it is woken up once by PID 135432. We can then examine the traces for PID 135430 and see what system call the process was executing in when it performed the wakeup of PID 135426:

```
    0.002959 cpu=30 pid=135430 lseek [8] entry fd=3 offset=0x3db9a000 whence=0
    0.002961 cpu=30 pid=135430 sched_switch prio=120 state=SSLEEP next_pid=0
next_prio=120 next_tgid=n/a policy=n/a STACKTRACE: __mutex_lock_slowpath+0x13e
mutex_lock+0x2b ext4_llseek+0x60 vfs_llseek+0x3a sys_lseek+0x66 tracesys+0xd9
    0.004169 cpu=60 pid=135432 sched_wakeup target_pid=135430 prio=120 target_cpu=30
success=1
...
    0.011837 cpu=30 pid=135430 sched_switch prio=120 state=SSLEEP next_pid=0
next_prio=120 next_tgid=n/a policy=n/a STACKTRACE: __mutex_lock_slowpath+0x13e
mutex_lock+0x2b ext4_llseek+0x60 vfs_llseek+0x3a sys_lseek+0x66 tracesys+0xd9
```

```
    0.012029 cpu=60 pid=135432 sched_wakeup target_pid=135430 prio=120 target_cpu=30
success=1
    0.012035 cpu=30 pid=0 sched_switch prio=n/a state=n/a next_pid=135430
next_prio=120 next_tgid=n/a policy=n/a
    0.012042 cpu=30 pid=135430 sched_wakeup target_pid=135426 prio=120 target_cpu=61
success=1
    0.012044 cpu=30 pid=135430 lseek [8] ret1=0x3db9a000 syscallbeg= 0.009085 fd=3
offset=0x3db9a000 whence=0
```

Note that the last thing that PID 135430 did before it exited the lseek() system call was to wakeup PID 135426. In this case, there are actually several processes competing for the mutex.

By using the sched_wakeup records along with the system call and switch records, ***kiinfo*** can help identify the tasks woken up by a particular task, as well as the tasks that woke up the target task. The PID Analysis Report (***kiinfo -kipid***) reports similar to the following:

```
PID 135426 /home/mcr/iotest8
    PPID 133402 -bash
...
    Tasks woken up by this task (Top 10)
           PID    Count   SlpPcnt    Slptime     Command
        135431      630    13.32%    2.377797  /home/mcr/iotest8
        135432      591    12.69%    2.263783  /home/mcr/iotest8
        135429      435    10.86%    2.118725  /home/mcr/iotest8
        135430      417    17.32%    3.421943  /home/mcr/iotest8
        135428      397    11.51%    2.251814  /home/mcr/iotest8

    Tasks that have woken up this task(Top 10)
           PID    Count   SlpPcnt    Slptime     Command
        135432     9839    20.63%    4.052633  /home/mcr/iotest8
        135431     9314    18.37%    3.608687  /home/mcr/iotest8
            -1     1372    17.19%    3.376019   ICS
        135430      490    15.46%    3.036041  /home/mcr/iotest8
        135429      427    13.99%    2.746939  /home/mcr/iotest8
        135428      426    14.36%    2.820286  /home/mcr/iotest8
```

The wakeups from the ICS (PID -1 with *LiKI* tracing mechanism and PID 0 with *ftrace* tracing) is likely due to IO completions or timeouts. So you can see that PID 135426 cooperates/competes with 5 other iotest tasks.

You can get additional information on cooperating/competing processes by using the ***coop*** flag when generating the PID Analysis Report (***kiinfo -kipid coop***):

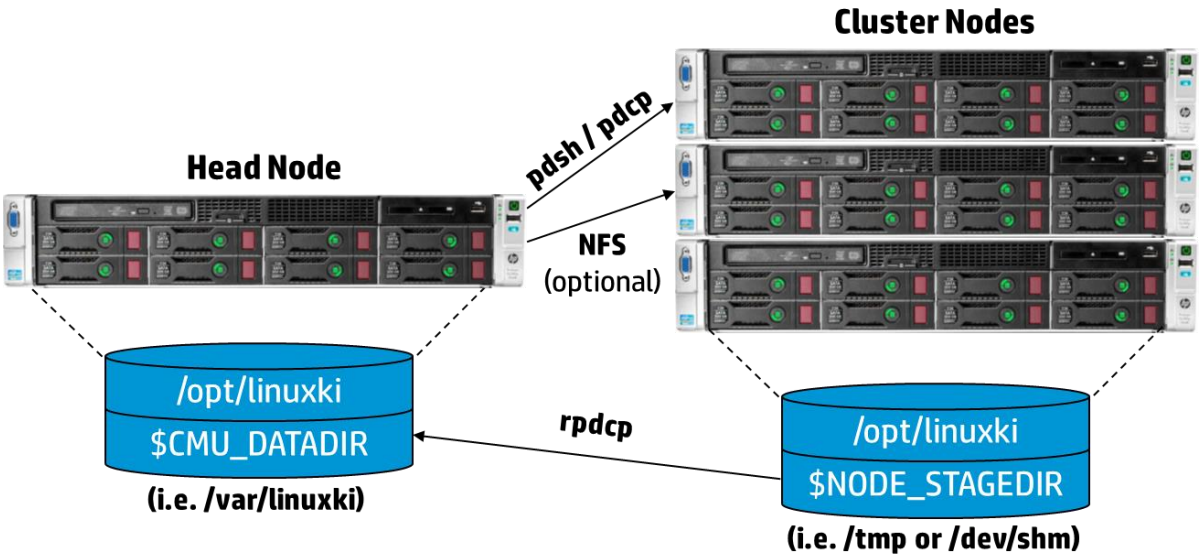
```
  Tasks woken up by this task (Top 10)
      PID  Count  SlpPcnt   Slptime   Command   WakerScall+arg0  SleeperScall+arg0
Sleep function
   135429  14755   24.55%   4.787027  /home/mcr/iotest8
            9411   12.84%   2.503767            write(fd=3)      lseek(fd=3)
__mutex_lock_slowpath+0x13e
             159    2.78%   0.542321            lseek(fd=3)      lseek(fd=3)
__mutex_lock_slowpath+0x13e
            5080    7.34%   1.431968            write(fd=3)      read(fd=3)
__mutex_lock_slowpath+0x13e
             105    1.58%   0.308971            lseek(fd=3)      read(fd=3)
__mutex_lock_slowpath+0x13e
```

In the above example, PID 135426 woke up PID 135429 14755 times. Of the 14755 wakeups, 9411 occurred when the task was in the write() system call and the target task (Sleeper) was in the lseek() system call.

As you can see, this can provide valuable information on how tasks interact with each other.

8 Using LinuxKI on a Linux cluster

LinuxKI can be configured to collect data from a cluster of Linux systems either manually or using HPE's Cluster Management Utility (CMU). The following diagram shows a general overview of the LinuxKI cluster collection:



Note that LinuxKI will be installed on a cluster head node. The head node may or may not collect its own LinuxKI trace data. It then uses pdsh/pdcp to start a LinuxKI dump on each of the cluster nodes (and optionally on the head node). Once it's complete, all of the ki_all.*.tgz files are made available to the head node which can process the data or can copy the data to another system for post-processing.

This section documents the steps necessary to configure the LinuxKI to collect data across a cluster and then analyze it.

8.1 LinuxKI cluster prerequisites

Before installing the LinuxKI in a cluster environment, there are several steps that need to be performed first:

1. Install LinuxKI on a designated head node
2. Install and configure SSH
3. Install and configure PDSH/PDCP
4. Install and configure NFS

Only after confirming each of the above steps should you proceed to Install and Configure LinuxKI for a Cluster environment.

8.1.1 Install LinuxKI on a designated head node

LinuxKI must be installed on a designated "head" node using the normal LinuxKI installation instructions. The cluster LinuxKI installation, data collection, and removal will all be initiated from the head node.

8.1.2 Install and configure SSH

Next, you will need to be sure you can login as root via SSH to each of the member nodes in the cluster without specifying a password. This is done by generating a private key on the head node and propagating that out to each of the cluster nodes.

1. First, log into the head node (hnode) as root and generate a pair of authentication keys. Do not enter a passphrase:

```
root@hnode:~> ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/a/.ssh/id_rsa):
Created directory '/home/a/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/a/.ssh/id_rsa.
Your public key has been saved in /home/a/.ssh/id_rsa.pub.
The key fingerprint is:
3e:4f:05:79:3a:9f:96:7c:3b:ad:e9:58:37:bc:37:e4 root@hnode
```

2. Use SSH to create the ~/.ssh directory as root on each cluster node (cnode). The directory may already exist, which is fine:

```
root@hnode:~> ssh root@cnode1 mkdir -p .ssh
root@cnode1's password:
```

3. Append root's new public key to each cnode's .ssh/authorized_keys file.

```
root@hnode~> cat .ssh/id_rsa.pub | ssh root@cnode 'cat >>.ssh/authorized_keys'
```

You should verify that you are able to login as root to member node without specifying a password before moving on...

```
$ ssh cnode1 uname -a
Linux cnode1 2.6.32-431.17.1.el6.x86_64 #1 SMP Fri Apr 11 17:27:00 EDT 2014 x86_64
x86_64 x86_64 GNU/Linux
```

Note
Depending on the version of SSH, you may also need to do the following:

- Put the public key in .ssh/authorized_keys2
- Change the permissions of .ssh to 700
- Change the permissions of .ssh/authorized_keys2 to 640

8.1.3 Install and configure PDSH/PDCP

You will need to install the following packages - pdsh and pdsh-rcmd-ssh. For our testing, we used version 2.26. You may find this on your subscription service, if not, you can find the RPMS on many different sites, such as http://www.rpmfind.net/linux/rpm2html/search.php. Note the packages must be installed on the head node and each of the cluster's member nodes you plan to use. If you are using a Debian Linux distribution (such as Debian, Ubuntu), you will need to find the appropriate .deb package.

After installing pdsh and pdsh-rcmd-ssh, set up a file which contains the name of each member node that you want to execute LinuxKI on (excluding the head node). For example, this could be the /etc/pdsh/machines file:

```
$ cat /etc/pdsh/machines
cnode1
cnode2
cnode3
```

Now, set the WCOLL environment variable to point to that file:

```
export WCOLL=/etc/pdsh/machines
```

You can use a different file than /etc/pdsh/machines, but it should match the WCOLL variable specified in the /opt/linuxki/config file. Next, verify that pdsh is able to execute successfully:

```
$ pdsh -R ssh uname -a
cnode1: Linux gwrdl980 2.6.32-431.17.1.el6.x86_64 #1 SMP Fri Apr 11 17:27:00 EDT 2014
x86_64 x86_64 x86_64 GNU/Linux
cnode2: Linux veloce.rose.hp.com 2.6.32-504.8.1.el6.x86_64 #1 SMP Fri Dec 19 12:09:25
EST 2014 x86_64 x86_64 x86_64 GNU/Linux
cnode3: Linux gwrlx1.rose.hp.com 2.6.32-131.0.15.el6.x86_64 #1 SMP Tue May 10 15:42:40
EDT 2011 x86_64 x86_64 x86_64 GNU/Linux
```

Also, verify the location of pdsh and pdcp. This needs to be the same for each node in the cluster. The location should match the EXPORTS variable in the /opt/linuxki/config file.

```
$ which pdsh pdcp
/usr/bin/pdsh
/usr/bin/pdcp
/usr/bin/rpdcp
```

8.1.4 Install and configure NFS

Using NFS is not required, but it is recommended for homogeneous clusters. For most systems, NFS is already installed. If not, be sure to install it. Be sure NFS port 2049 is able to accept connections. The following line should be in the /etc/sysconfig/iptables file:

```
-A INPUT -m state --state NEW -m tcp -p tcp --dport 2049 -j ACCEPT
```

If you have to modify the iptables file, be sure to restart the iptables service:

```
$ service iptables restart
```

From the head node, make sure that the NFS services have been started:

```
$ service nfs start
Starting NFS services:                                     [  OK  ]
Starting NFS quotas:                                       [  OK  ]
Starting NFS mountd:                                       [  OK  ]
Starting NFS daemon:                                       [  OK  ]
Starting RPC idmapd:                                       [  OK  ]
```

For each member node, you will need to configure one or 2 entries in the /etc/exports file. You can use a different file than /etc/exports, but you will need to specify the file name in the /opt/linuxki/config file. You can mount /opt/linuxki using NFS, which means that each member node will use the same binaries and *LiKI* DLKM module to collect the data. This means that it should be a homogeneous cluster (same Linux version). Optionally, you can also mount the target directory where the ki_all.*.tgz file will be placed after the LinuxKI cluster collection. The target directory should match the CMU_DATADIR variable specified in the /opt/linuxki/config file. Below is an example of the /etc/exports contents:

```
/var/linuxki          cnode1(rw) cnode2(rw)
/opt/linuxki          cnode1(rw) cnode3(rw)
```

In the example above, note that NFS will be used for /opt/linuxki and /var/linuxki on cnode1. This means that LinuxKI does not have to be installed on cnode1. The LinuxKI data will be copied across NFS to /var/linuxki when the data collection is done. For cnode3, it will also use /opt/linuxki on the head node, but the data will be logged locally and the head node will pull the data to it using rpdcp. For cnode2, it must have LinuxKI installed locally to collect data, but will transfer the data across NFS to the head node when the data collection is complete. In this example, cnode1 and cnode3 should be running the same version of Linux, but cnode2 could be running an entirely difference version of Linux.

After updating the /etc/exports file, be sure to export the files using exportfs -a or exportfs -r so the exported filesystems can be seen by the member nodes.

8.2 Installing LinuxKI on a Linux cluster

To install and configure LinuxKI in a cluster environment, you will need to follow the following steps:

1. Edit /opt/linuxki/config
2. Execute /opt/linuxki/cluster/cluster_install script

8.2.1 Edit /opt/linuxki/config

Next, edit the following variables in the /opt/linuxki/config file:

- CMU_DATADIR - the directory on the head node to receive the data from member nodes, such as /var/linuxki. This should be a fairly large space to handle the incoming data from all the member nodes. If you are using NFS, this filesystem should have been exported.

- NODE_STAGEDIR - the directory on member nodes used for temporary work space during the actual data collection - default is /tmp. If there is sufficient memory on each node, this could even be /dev/shm.
- PDSH, PDCP, RPDCP - location of the pdsh, pdcp, and rpdcp binaries (this should be the same location on all nodes)
- WCOLL - location of the text file on the head node containing a list, one per line, of cluster member host names/IPs and does not include the head node. This file should have been set up in the previous page.
- EXPORTS - location of the NFS exports file on head node, normally /etc/exports

8.2.2 Execute /opt/linuxki/cluster/cluster_install

The cluster_install script will set up the member nodes including NFS mounts for the $CMU_DATADIR and /opt/linuxki (if necessary):

```
$ /opt/linuxki/cluster/cluster_install
cnode1: --- Linux KI Toolset node installation starting...
cnode2: --- Linux KI Toolset node installation starting...
cnode2: --- Linux KI Toolset installed locally
cnode2: --- Linux KI Toolset node installation complete!
cnode3: --- Linux KI Toolset node installation starting...
cnode3: --- Linux KI Toolset node installation complete!
cnode1: --- Linux KI Toolset node installation complete!
```

In the example above, note that cnode2 is not using the NFS to mount the /opt/linuxki directory, but is using its local copy of LinuxKI.

8.3 Collecting LinuxKI data on multiple cluster nodes

If everything is set-up properly, then to collect data on each of the nodes specified in the WCOLL pdsh file, simply execute the cluster_collect script.

```
$ /opt/linuxki/cluster/cluster_collect

=== Linux KI Toolset data collection 0218_0735 starting ===
cnode1: --- Linux KI Toolset data collection running...
cnode2: --- Linux KI Toolset data collection running...
cnode3: --- Linux KI Toolset data collection running...
cnode2: --- Linux KI Toolset data collection complete!
cnode3: --- Linux KI Toolset data collection complete!
cnode1: --- Linux KI Toolset data collection complete!
/var/linuxki:
-rw-r--r-- 1 nfsnobody nfsnobody 261224055 Feb 18 07:37
/var/linuxki/ki_all.cnode1.0218_0735.tgz
-rw-r--r-- 1 nfsnobody nfsnobody 991944 Feb 18 07:36
/var/linuxki/ki_all.cnode2.0218_0735.tgz
-rw-r--r-- 1 nfsnobody nfsnobody 24377455 Feb 18 07:37
/var/linuxki/ki_all.cnode3.0218_0735.tgz
=== Linux KI Toolset data collection 0218_0735 finished ===
```

Remember, if you want to change the ***runki*** options on the member nodes, you can modify the /opt/linuxki/config file on the head node. For example, to collect data for 10 seconds instead of 20 and skip collecting perf and MW data, you can modify the RUNKI_OPTS as follows:

```
export RUNKI_OPTS="-m -u -d 10"
```

8.4 Cluster-wide LinuxKI processing

LinuxKI has the functionality to assist in analyzing LinuxKI data from multiple Linux nodes in a cluster, such as a Hadoop or Vertica cluster. In order for the Cluster-wide LinuxKI reporting to work, the LinuxKI data must be collected on each node in the cluster using the ***runki*** or cluster_collect scripts and copied to a system where the analysis will be done.

The ***kiinfo -clparse*** option and supporting ***kiall*** script will create a consolidated LinuxKI report of activity from multiple nodes in a Linux cluster - the Cluster Overview Report. The HTML output can also link the server names to the per-server Kparse reports and the PID numbers to the appropriate kipid report in the per-server PIDS directory if the -cltree option is used.

Typically, the generation of the Cluster Overview Report will be managed through the ***kiall*** script. The -c argument of the ***kiall*** script will cause a subdirectory of the form <timestamp>/<hostname> to be created for each ki_all.*.tgz file found in the current working directory. Note this directory structure is different than one created with ***kiall*** -r, which uses the form <hostname>/<timestamp>. The ***kiall*** script will proceed to generate the LinuxKI reports for each server. After that, the ***kiall*** script will execute ***kiinfo -clparse*** to generate the Cluster Overview Report. For example:

```
$ cd /var/linuxki
$ ls
ki_all.cnode1.0218_0735.tgz  ki_all.cnode2.0218_0735.tgz  ki_all.cnode3.0218_0735.tgz

$ kiall -c
Processing files in: /var/linuxki/0218_0735/cnode1
Merging KI binary files.  Please wait...
ki.bin files merged by kiinfo -likimerge
/opt/linuxki/kiinfo -kitrace sysargs,sysenter -ts 0218_0735
/opt/linuxki/kiinfo -kiall csv -html -ts 0218_0735
Processing files in: /var/linuxki/0218_0735/cnode2
Merging KI binary files.  Please wait...
ki.bin files merged by kiinfo -likimerge
/opt/linuxki/kiinfo -kitrace sysargs,sysenter -ts 0218_0735
/opt/linuxki/kiinfo -kiall csv -html -ts 0218_0735
Processing files in: /var/linuxki/0218_0735/cnode3
Merging KI binary files.  Please wait...
ki.bin files merged by kiinfo -likimerge
/opt/linuxki/kiinfo -kitrace sysargs,sysenter -ts 0218_0735
/opt/linuxki/kiinfo -kiall csv,oracle -html -ts 0218_0735
/opt/linuxki/kiinfo -clparse csv,cltree,top=20 -html -ts 0218_0735
Number of Servers to analyze: 3
Processing KI files in ./cnode3
Processing KI files in ./cnode2
Processing KI files in ./cnode1
kiall complete
```

If a Cluster Overview Report (cl.<timestamp>.html) is needed for LinuxKI data already processed (for example, the Cluster Overview Report needs to be regenerated), then the ***kiall*** script can be executed again to generate the report by specifying the -t <timestamp> option. This causes the ***kiall*** script to search the directory tree looking for LinuxKI data with the same timestamp. For example:

```
$ pwd
/var/linuxki
$ ls
0218_0735
$ ls 0218_0735
cp.0218_0735.html cnode1  cnode2  cnode3  kiall.0218_0735.csv

$ kiall -c -t 0218_0735]
/opt/linuxki/kiinfo -clparse csv,cltree,top=20 -html -ts 0218_0735
Number of Servers to analyze: 5
Processing KI files in ./0218_0735/cnode3
Processing KI files in ./0218_0735/cnode2
Processing KI files in ./0218_0735/conde1
kiall complete
```

8.5 Adding and removing nodes in a cluster

If you wish to add or remove nodes in the cluster, go through the following steps:

- Remove the LinuxKI Cluster installation
- Add or remove the nodes from the WCOLL file (i.e. /etc/pdsh/machines)
- Check the Pre-requisites for each node being added. You must be able to use pdsh to get to any new member nodes.
- If you are using NFS, be sure to modify the /etc/exports and execute "exportfs -a" or "exportfs -r"
- Re-execute the /opt/linuxki/cluster/cluster_install script

8.6 Removing LinuxKI from a cluster

To remove the LinuxKI cluster configuration, first make sure that NFS mount points used by LinuxKI are not active. Then simply login to the head node and execute the /opt/linuxki/cluster/cluster_remove script:

```
$ /opt/linuxki/cluster/cluster_remove
cnode3: --- Linux KI Toolset node uninstall starting...
cnode1: --- Linux KI Toolset node uninstall starting...
cnode2: --- Linux KI Toolset node uninstall starting...
cnode1: --- Linux KI Toolset node uninstall complete!
cnode3: --- Linux KI Toolset node uninstall complete!
cnode2: --- Linux KI Toolset node uninstall complete!
```

Note that the LinuxKI will still be installed on the head node, as well as any of the member nodes. But the NFS mounts from the member nodes will be removed.

8.7 Integration with Cluster Management Utility (CMU)

LinuxKI can easily be integrated with HP Insight Customer Management Utility (CMU). The CMU integration simplifies the cluster node installation and collection process by using the CMU menus and allowing the user to easily select which cluster nodes to install LinuxKI and collect data.

8.7.1 Prerequisites

Most the prerequisites for using LinuxKI on a cluster are already satisfied in a cluster managed with CPU:

- No password SSH/pdsh access (null passphrase) for root to each cluster node
- CMU Monitoring Client installed on each cluster node
- NFS access to head node from each cluster node (optional)

8.7.2 Install LinuxKI on the CMU Head Node

To integrate LinuxKI with CMU, simply install LinuxKI on the head node as you would normally install the toolset. LinuxKI will detect that it is being installed on the CMU head node and will automatically do the following:

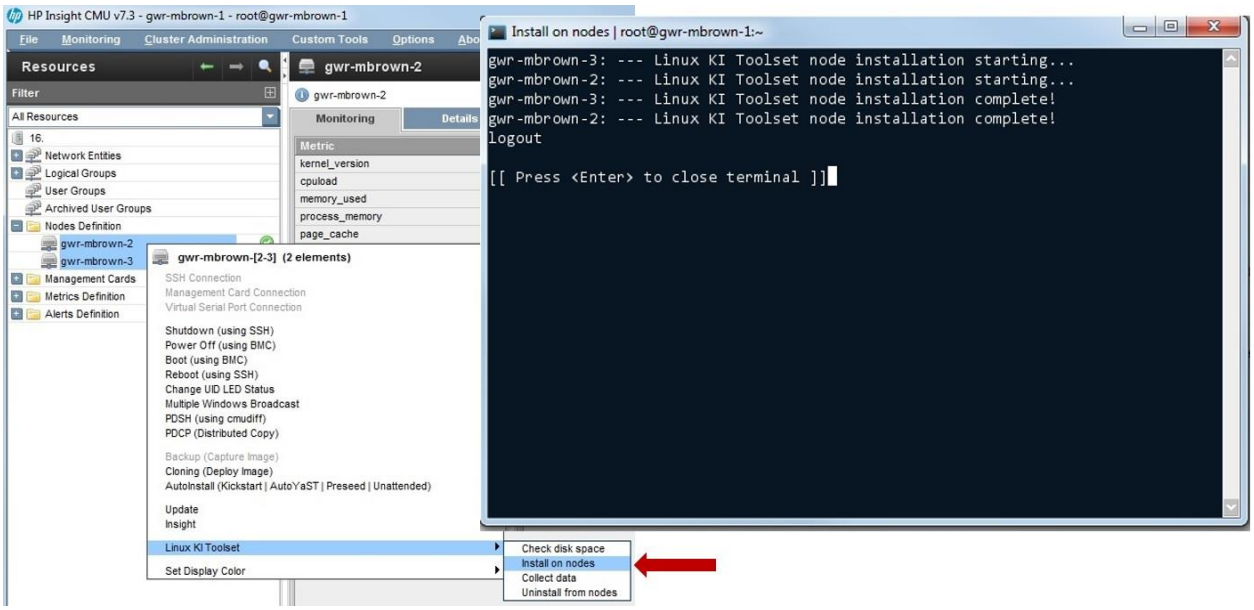
- Adds menu items to /opt/cmu/etc/cmu_custom_menu
- Adds /opt/linuxki to /etc/exports

8.7.3 Verify/modify the /opt/linuxki/config configuration file

After installation, you will need to verify/modify the /opt/linuxki/config configuration file. For example, if you want to collect the LinuxKI data into a different directory than /var/linuxki, you can modify the config file as needed.

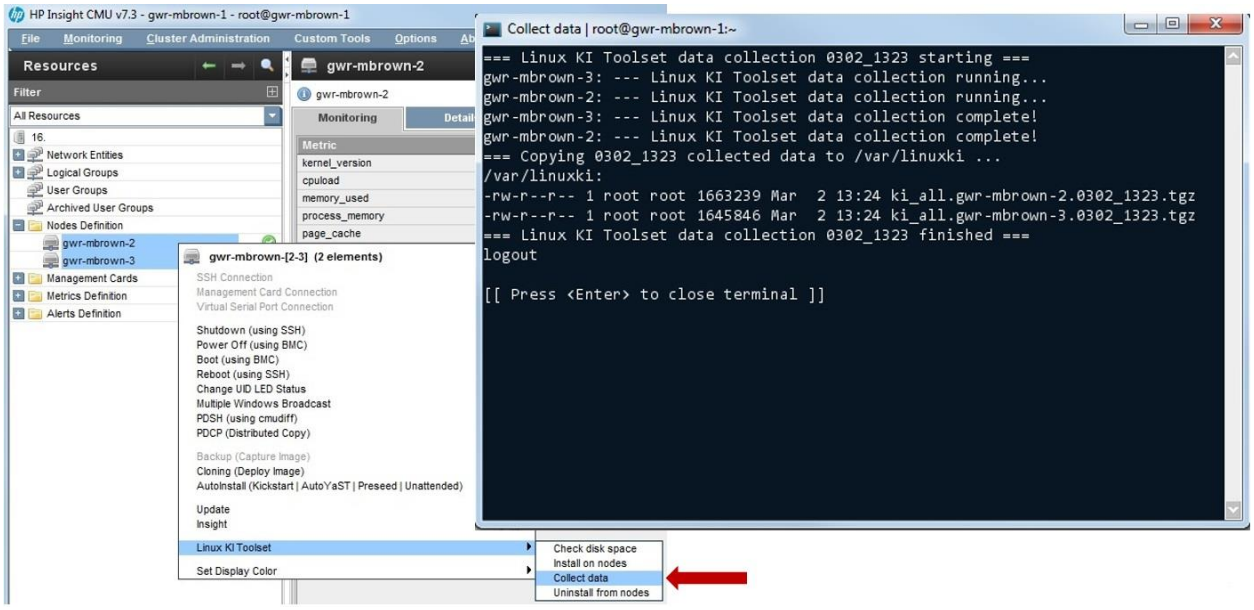
8.7.4 Install LinuxKI on cluster nodes

Installing LinuxKI on the cluster nodes is done via CMU. Select the nodes where you intend to collect LinuxKI data under the Nodes Definition list from the Resources menu. Then right click for the pull down menu and select "Linux KI Toolset --> Install on nodes". CMU will then install LinuxKI on the selected nodes. There is no need to edit the /etc/pdsh/machines file. When LinuxKI is installed on a cluster node, it will attempt to mount /opt/linuxki from the head node via NFS. If NFS is not available on the cluster node, then the cluster node will use its local copy of /opt/linuxki if it was previously installed. This method allows the LinuxKI data collections to occur on heterogeneous or homogeneous clusters.



8.7.5 LinuxKI Cluster collection

Starting a LinuxKI Cluster collection with CMU is similar to installing the toolset. Simply select the nodes where you previously installed LinuxKI from the previous set and select "Linux KI Toolset --> Collect Data". The LinuxKI data will be collected from each node and stored on the head node in the CMU_DATADIR specified in the /opt/linuxki/config file.



9 Visualization Charts and Graphs

LinuxKI has the ability to produce enhanced Visualization charts and graphs. The intent is give users a visual aid to understand and search for anomalies in behavior. While LinuxKI data often encompasses only 20 seconds of data, it can generate an enormous quantity of trace records containing rich, detailed data. Often, even a 20 second snapshot can have many variations and anomalies that can be lost in the averages. The Visualization charts and graphs give the LinuxKI user another method for viewing the data, and finer granularity can also bring out interesting patterns when displayed graphically.

The Charts and graphs are dynamic, interactive and utilize the D3 (Data Driven Documents) JavaScript libraries. Many of the charts are linked to others. For example, the PIDs highlighted in the clparse and kparse reports are linked to the per-task visualizations, and the IO scatter graph chart is linked to charts for the PIDs that issued the IO, etc.

Since many of the charts are highly interactive, descriptions here may not fully describe all features, and the reader is encouraged to use/explore the charts for themselves.

The rich set of raw data available through LinuxKI offer many more visualization possibilities than those presented in these charts.

9.1 Visualization prerequisites

The html charts use Scalable Vector Graphics (SVG) which most modern browsers support. The Google Chrome browser appears to have better performance, but Firefox, Safari, and IE9+ work as well.

The html charts load the data (JSON or CSV files) as well as JS libraries via http/https requests ***so the LinuxKI data and visualization charts must be accessed and hosted via web-server.***

The dynamic chart creations (timeline drill-downs) use PHP scripts to create the requested reports/charts, so ***the PHP rpm needs to be installed and configured for use in the web-server***.

The web-user account (e.g. apache) ***must have write permission to the directories used for dynamic content creation*** via .php scripts. This is typically the ./tl_temp directory and its sub-directories.

9.2 Generating visualization data

The ***kiall*** script includes a ' -V ' option which will create the visualization files:

```
$ kiall -V
```

Generating visualization data can take time. With LinuxKI 6.0, the -P options was added to allow for parallel processing during the visualization processing.

```
$ kiall -V -P [num_threads]
```

By default, if the -P option is specified but the number of threads is omitted, the visualization processing will use as many threads as there are logical cores.

If processing a cluster set of LinuxKI data collections:

```
$ kiall -c -V
```

When processing a set of LinuxKI collections using the above ' -c ' cluster option, the following directory structure is created:

```
$ ./<timestamp>/<hostnames>/
```

The ./<timestamp>/clparse.<timestamp>.html report will contain hyperlinks to the individual host kparse reports as well as PIDs within the collections. This is a good starting point to explore the data, and it contains links to the visualization charts for the processes and systems included in the cluster data.

At this same directory level (./<timestamp>/) you will find a cluster_timeline.html report which includes a fine grained visualization of activity for the cluster data. This report is also a good starting point to begin the cluster analysis. A more detailed description of the timeline charts is covered in the next section.

For a stand-alone (single) LinuxKI data set:

```
kiall -r -V    (data will be placed in  ./<hostname>/<timestamp>/  subdir)
kiall -V       (data will be placed in your current working directory,  ./  )
```

For the stand-alone data collections, the **kparse.<timestamp>.html** report and the **timeline.html** report are appropriate starting points for the analysis. These report are also generated at these same directory levels for each server when doing cluster processing.

9.3 Timeline Graphs

The typical LinuxKI reports contain data which is a summary of the entire trace duration (typically 20 seconds by default). Within the trace data however, there can be distinctly different behaviors which only occur for small (sub-second) periods. The LinuxKI data is time-stamped to the nanosecond precision, and reported at the microsecond timeframe so it is possible to report on periods much finer-grained than the ~20 second trace total timeframe. You can use the timeline charts to drill down on subsections of the trace and generate reports only for the timespan desired. The various metrics used to encode color and height can help identify the more interesting time ranges.

There are two timeline charts created to help identify and automate the creation of reports on smaller sub-sections of the trace duration. The cluster-wide chart is cluster_timeline.html and is found at ./<timestamp>/cluster_timeline.html in the directory structure created when using ***kiall -c -V*** to format the cluster LinuxKI data collection. The individual server chart is timeline.html and is found in the main directory that contains the raw LinuxKI binary data and standard LinuxKI text and html report files. By default, the timeline charts summarize the LinuxKI data in 100ms intervals.

A broad range of server-wide metrics is created and available in CSV format in the files:

Chart Name	Type	Data Source
cluster_timeline.html	TimeLine	cluster_timeline.csv
timeline.html	TimeLine	server_timeline.csv

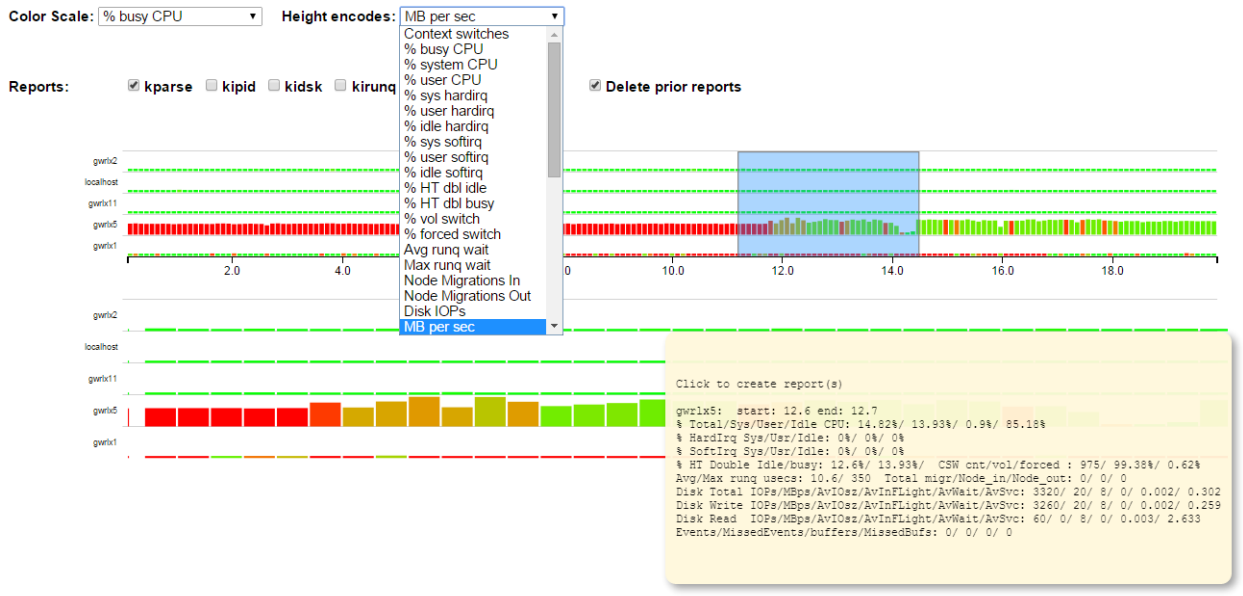
These metrics can be used to encode the color and the height of the interval 'bars' displayed for each of the 100ms intervals via pull-down menus at the top of the chart. The timeline charts contain two areas used to view the activity. The upper section displays the activity for the entire trace duration, and the lower section displays a larger, scalable subsection of the trace timeline. The interval bars do not appear in the lower section until a timespan is selected in the upper section. To select a subsection of the trace, left-click and drag in the upper section and a shaded window will appear to highlight the selected time range. The data from this selection window is displayed in the lower section. Once a subsection of the trace timeline is selected, you can generate various LinuxKI reports on this filtered time range using the check-boxes across the top of the chart.

Other features of the timeline charts include pop-up summaries of the metrics when hovering the mouse cursor over the lower section bars.

By default, the "Delete prior reports" box is checked and all content created from previous drill-downs is deleted when a new drill-down is performed. To save previous reports, uncheck this box.

The cluster-wide timeline chart is shown below. The single-server timeline chart looks similar but with only one host line shown.

Cluster Activity Timeline



9.4 Pid Analysis Report with Visualization

The PID Analysis visualizations have several components. The partition diagram titled **Task Time Chart** is meant to help visually understand where a task spends its time - either running, waiting on a runq, or blocked. The other visualization is a **Task Wait Dependency** chart which is meant to help display the interaction between tasks, how much time one task spends waiting for another, as well as task CPU utilization. These charts contain hyperlinks to the other PIDs referenced in the chart allowing you to easily traverse the visualizations for a group of related tasks/PIDs. The **PID Activity Timeline** shows a timeline based activity profile for various statistics. Using the PID Activity Timeline, you can also generate a **Task Scheduling Timeline** for a specific interval to drill down on how the task spent its time during the interval.

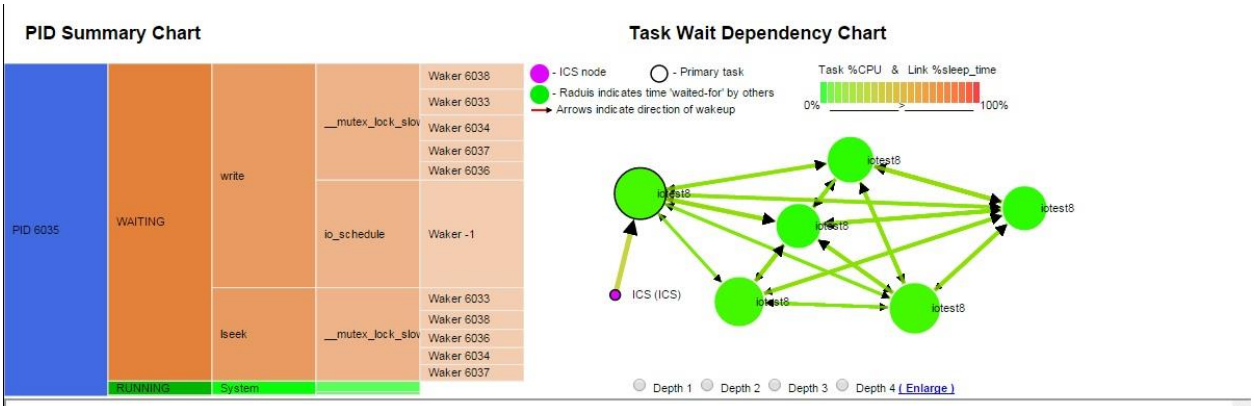
The **Task Time Chart** depicts a hierarchy of runtime, blocking time, and runq wait time, showing what syscalls and blocking functions in the kernel are contributing to each of the main time categories. **Clicking on a particular section will expand and zoom in** to show that portion of the hierarchy in more detail. **Hovering the mouse over a section** will display detail similar to that found in the kipid text reports. At the bottom of the visualization page we include the full text report for the PID as well. The blocking kernel functions have varying degrees of accuracy based on the kernel distribution used, with RHEL6+ and Ubuntu being the most accurate. Where possible we display the tasks responsible for waking from a particular blocking path. **If you click on the partition for the wakers task ID**, you will be taken to the kipid report (with visualizations) for that particular task.

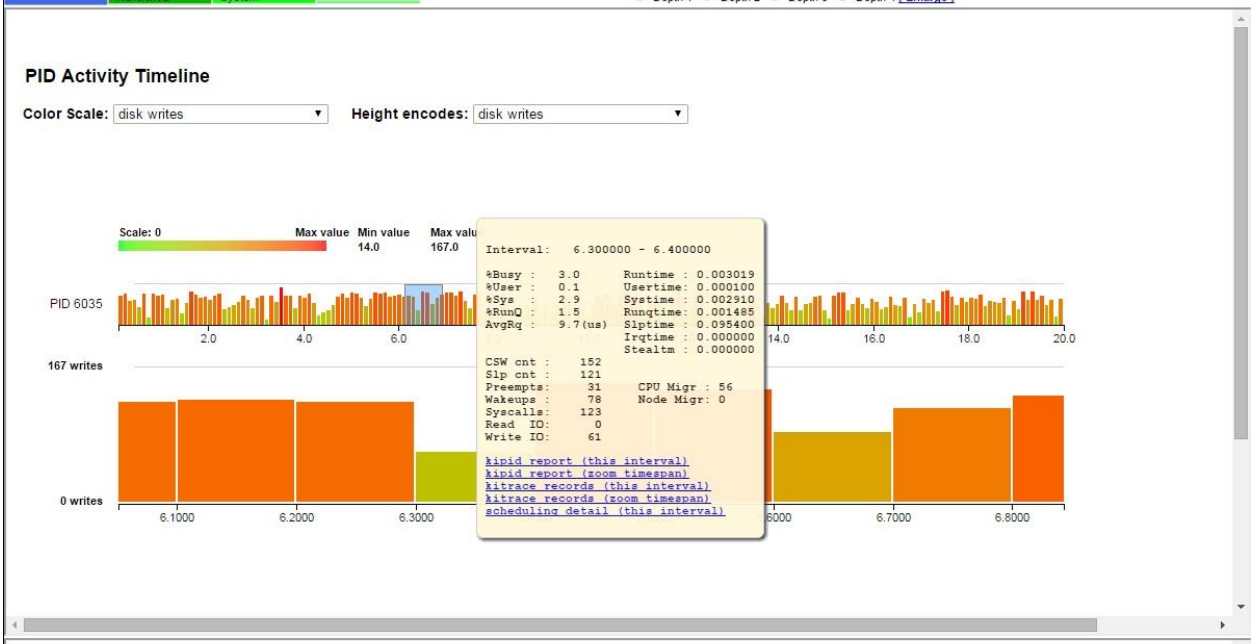
The **Task Wait Dependency** chart depicts tasks, represented by circles. The color scale (blue to red) of the circle encodes the CPU utilization of the task. The diameter of the circle encodes the cumulative amount of wait time that all other tasks have waited for this task to wake them up. Hence a large red circle represents a task which itself uses a lot of CPU, and has many other tasks waiting a significant amount of time for it. A large blue-ish circle represents a task which itself blocks much of the time, but has many others waiting for him to wake them. The arrowed links between tasks represent the wakeup mechanisms. The direction of the arrow indicates who is being woken. The width of the link scales to amount of sleep time in seconds for the woken task, while the color scales to the % of the sleepers total sleep time due to this waker. Hence, thick red arrows indicate the waker is responsible for the majority of the sleep time associated with the woken task. These encoded sizes and colors are meant to highlight the most interesting tasks to investigate further within a larger group of inter-related tasks. The chain of related tasks begins with those tasks that have directly woken or been woken by the primary task (the one whose circle perimeter is highlighted in black). We can expend the depth of sleeper/wakers to then include all tasks who sleep/wake the secondary tasks, etc. **To control the depth of these relationships displayed, select the 'Depth' radio button on the chart.** By default the ***kiall*** script will display a depth of 3, for any tasks that are responsible for 1% or more of a sleepers blocking time. These two values, the depth and the % sleeptime, are options passed to the ***kiinfo*** tool when post-processing the LinuxKI collections (see vis, vdepth, vpct options of ***kiinfo*** tool). **In order to follow a chain of interacting tasks, you can click on the circle/task** to see the kipid report (with visualizations) for the task chosen. All **circles/tasks can be dragged and pinned to a location via mouse left-click** in order to better organize the layout. The ICS is represented as a magenta colored circle at all times.

The **PID Activity Timeline** is allows you to select on different metrics, such as CPU, syscall count, or disk reads or writes. The subsequent timeline can help identify periods of interesting activity during the life of the Task (PID). You can select a subset of time to use to drill down further. Then, by right clicking on

the interval in question, you can generate various reports for the specific interval. Note if you right click on a specific interval, one of the options is "scheduling detail (this interval)". If you click on the "scheduling detail", then you will get the **Task Scheduling Timeline** for the specific interval.

When using the timeline charts to drill down into a particular timeframe, the PID Analysis Report (***kiinfo -kipid***) and visualizations it generates are restricted to the timeframe of the drill-down, allowing for a finer grained view of activity.





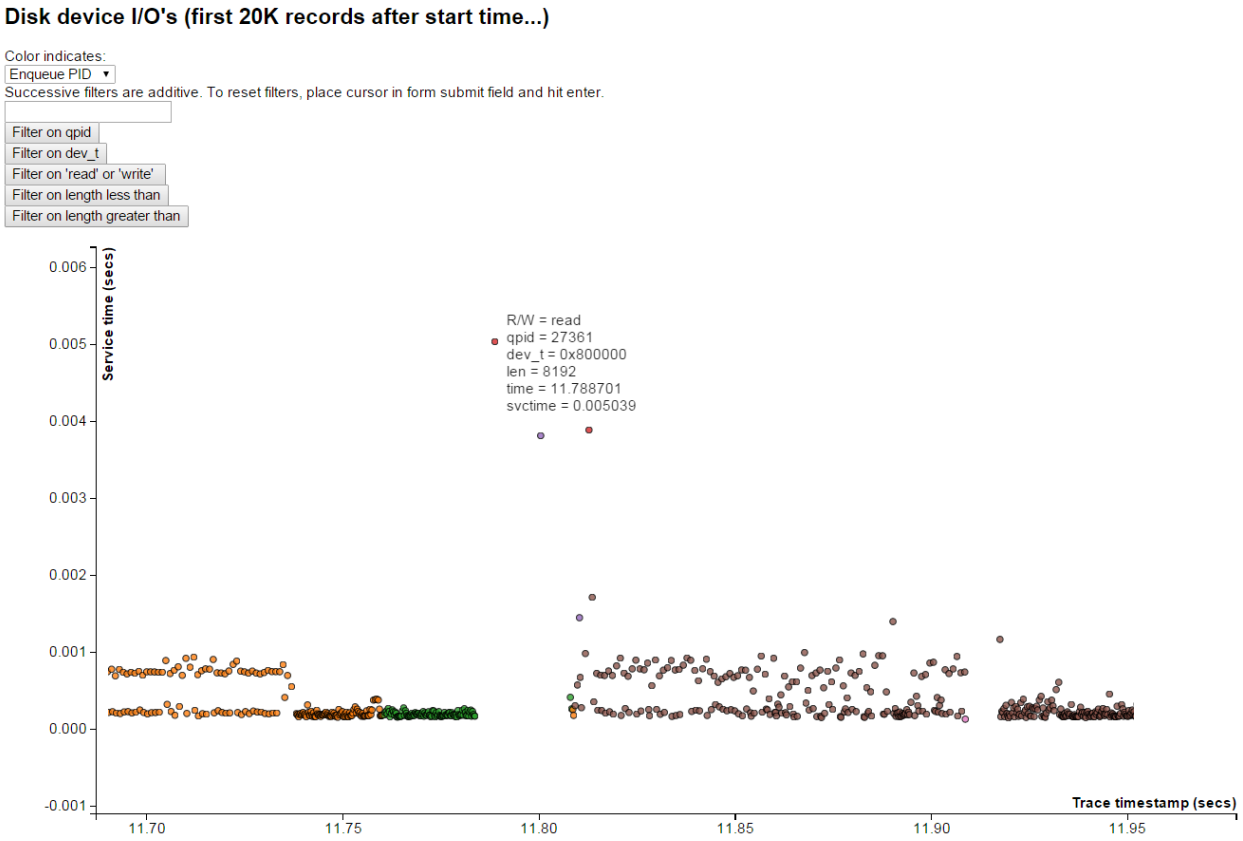

```
PID 6035  /home/mcr/bin/iotest8
  PPID 1  /sbin/init

    ********* SCHEDULER ACTIVITY REPORT ********
    RunTime     :  0.840574  SysTime    :  0.810054   UserTime    :  0.028887
    SleepTime   : 18.820487  Sleep Cnt :     35369   Wakeup Cnt :     24142
    RunQTime    :  0.338531  Switch Cnt:     40129   PreemptCnt :      4760
    HardIRQ     :  0.000044  HardIRQ-S :  0.000043    HardIRQ-U :  0.000001
    SoftIRQ     :  0.001588  SoftIRQ-S :  0.001513    SoftIRQ-U :  0.000075
    Last CPU    :         0  CPU Migrs :     11429   NODE Migrs :        16
    Policy      : SCHED_NORMAL     vss :      1016          rss :       152
```

9.5 Scatter Graphs

There are two types of scatter graphs created, **kidsk_scatter.html** and **futex_scatter.html**. As the names imply, one displays Disk IOs in an interactive scatter graph and the other displays futex system calls being made.

The kidsk scatter graph display a colored dot of varying diameter for every IO observed completing. The menus allow you to change the meaning of the color encoding between read/write, disk device, and enqueuing PID. The diameter encodes the IO size. The sizes are scaled across a range of the smallest IO observed to the largest IO in the data collection. If you mouse over the IO dot, a popup displays the metrics associated with the IO. If you left-click on the IO, you will be taken to the kipid report for the task that enqueued the IO.



You can use a mouse scroll wheel to expand (zoom in) or contract (zoom out) the timescale. You can use the mouse left-click (when not on an IO dot) to drag/pan the timescale left or right. The vertical axis does not zoom or scale, only the time axis.

You can also use the blank submit field to specify a filter by enqueue PID, dev_t, read, write, and IO length. When specifying filters, each successive filter is additive. To reset the display, place the cursor in the submit box and just hit enter.

The chart only displays the first 20K IO's for the selected timeframe since most browsers start to perform poorly with more than 20-30K elements to display. If you would like to increase this amount,

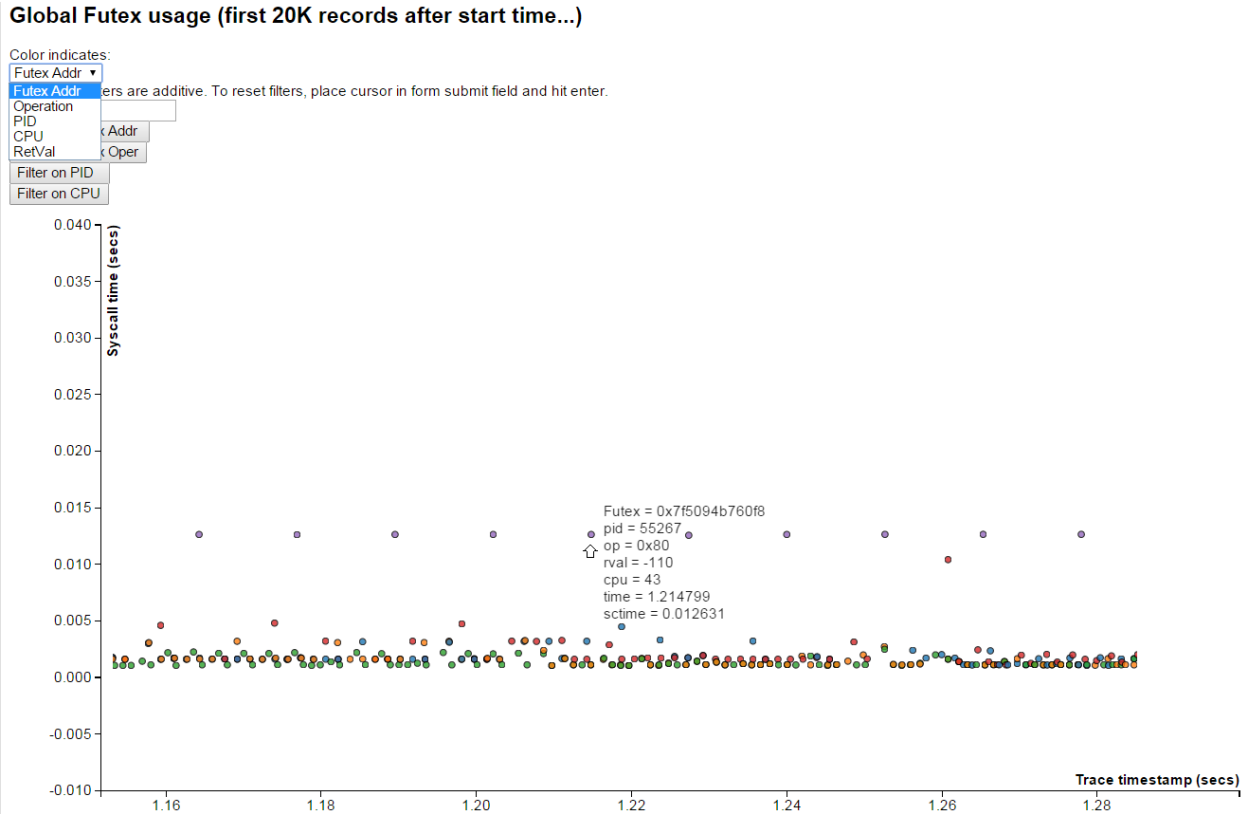
you can alter the **/opt/linuxki/experimental/vis/kidsk_2_csv.sh** shell script that creates the kidsk_global.csv file used by the kidsk_scatter.html chart. To do so, change the argument passed to the 'head' command in the following line of the **kidsk_2_csv.sh** script:

```
$ sed 's/ /,/g' kidsk_global.raw2 | head -20000  >> kidsk_global.csv
```

There is a similar 20K element limit for the futex-scatter.html chart, and this too can be altered by changing the same 'head' command argument used in the shell script, futex_2_csv.sh.

```
sed 's/ /,/g' futex_global.raw3 | head -20000 >> futex_global.csv
```

The futex chart has similar control features for displaying and filtering the futex related data.



9.6 CSV Charts (parallel coordinate charts

There are many CSV files generated for which there is no dedicated html viewing chart. A general purpose CSV html chart has been provided to help visualize the relationships among the row/columns of data.

The columns of data are shown as a scale if numeric or a list if non numeric. The values for a particular row are linked by a colored line specific to that row. The placement of the columns can be changed by

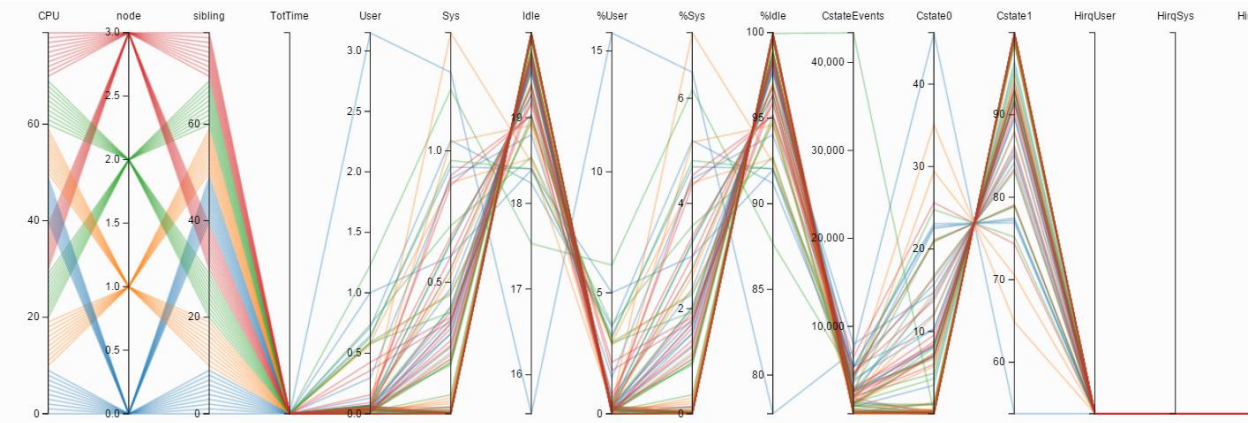
dragging them into the desired configuration. You can also select a range of data to highlight (others not in the range will be shaded out) by clicking and dragging down a columns scale range. Any row that has data within the selected range remains highlighted. The spreadsheet table at the bottom is automatically filtered to only show rows with rows matching the filtered column. You can filter on multiple columns. If you select a row from the spreadsheet data in the table at bottom, the corresponding colored line for this row element is highlighted and all others are shaded out. To deselect any highlighted range, simply click below the range box on the particular column and the selection will be cleared.

When you use the ***kiall*** script to process a complete LinuxKI collection it creates CSV files for various categories of metrics. This is not part of the visualization package and has been in place for some time. When the -V option is used with the ***kiall*** script, the following html files are linked into the directory to view the various CSV files:

- kisdk.html (for kidsk.csv)
- kifile.html (for kifile.csv)
- kipid_io.html (uses IO stats from kipid.csv which is an extracted subset of kipid.csv)
- kipid_sched.html (uses scheduling stats from kipid.csv which is an extracted subset of kipid.csv)
- kirunq.html (for kirunq.csv)
- kiwait.html (for kiwait.csv)

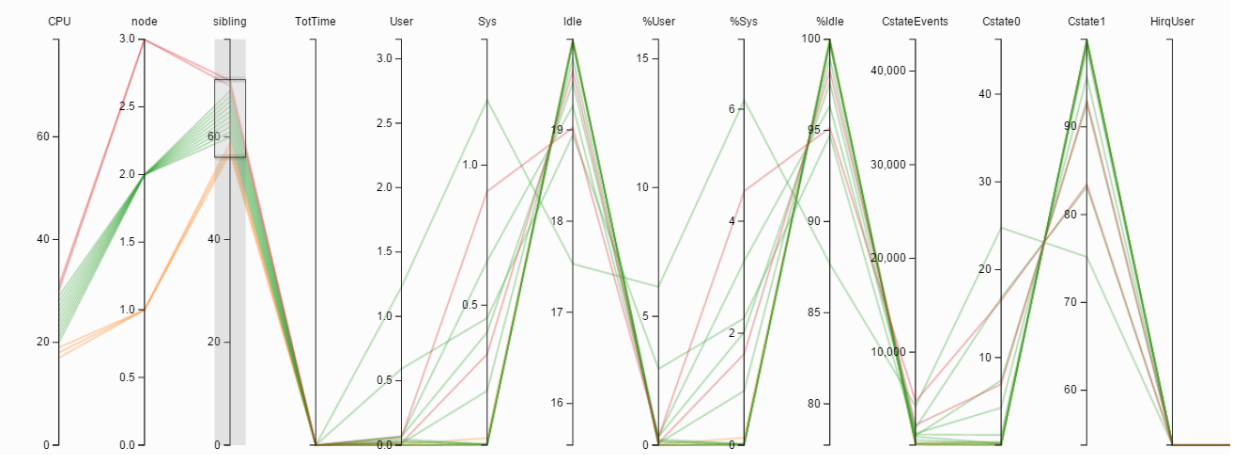
All of these html files are basically the same except for the name of the CSV file they read in. See **/opt/linuxki/experimental/vis/d3csv.sh** and **/opt/linuxki/experimental/vis/kipid_awk_csv.sh** for details.

The three variations of the kirunq.html examples shown below are 1) the complete chart, 2) a range selected from the "siblings" column, and 3) a specific CPU highlighted in the spreadsheet table below the chart.



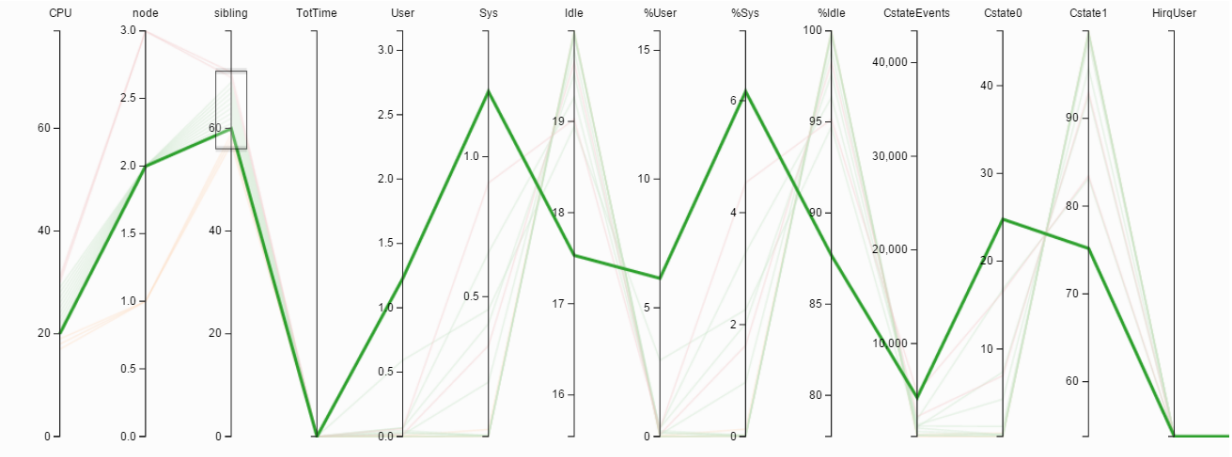
Showing all 80 rows

CPU	node	sibling	TotTime	User	Sys	Idle	HirqUser	HirqSys	HirqIdle	SirqUser	SirqSys	SirqIdle	%User	%Sys	%Idle
0	0	40	19.994984	0.716285	1.040387	18.238311	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	3.58	5.20	91.21
1	0	41	19.994984	0.128894	0.478835	19.387254	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.64	2.39	96.96
2	0	42	19.994984	0.073088	0.652131	19.269765	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.37	3.26	96.37
3	0	43	19.994984	0.066353	0.552633	19.375998	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.33	2.76	96.90
4	0	44	19.994984	0.069214	0.361744	19.564026	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.35	1.81	97.84
5	0	45	19.994984	0.647225	0.939138	18.408620	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	3.24	4.70	92.07



Showing all 15 rows

CPU	node	sibling	TotTime	User	Sys	Idle	HirqUser	HirqSys	HirqIdle	SirqUser	SirqSys	SirqIdle	%User	%Sys
17	1	57	19.994984	0.000000	0.000664	19.994319	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00	0.00
18	1	58	19.994984	0.000000	0.003013	19.991971	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00	0.02
19	1	59	19.994984	0.006298	0.025319	19.963366	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.03	0.13
20	2	60	19.994984	1.228975	1.233343	17.532665	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	6.15	6.17
21	2	61	19.994984	0.068141	0.657787	19.269056	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.34	3.29
22	2	62	19.994984	0.592790	0.454374	18.947820	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	2.96	2.27



Showing all 15 rows

CPU	node	sibling	TotTime	User	Sys	Idle	HirqUser	HirqSys	HirqIdle	SirqUser	SirqSys	SirqIdle	%User	%Sys
17	1	57	19.994984	0.000000	0.000664	19.994319	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00	0.00
18	1	58	19.994984	0.000000	0.003013	19.991971	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00	0.02
19	1	59	19.994984	0.006298	0.025319	19.963366	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.03	0.13
20	2	60	19.994984	1.228975	1.233343	17.532665	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	6.15	6.17
21	2	61	19.994984	0.068141	0.657787	19.269056	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.34	3.29
22	2	62	19.994984	0.592790	0.454374	18.947820	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	2.96	2.27

10 LinuxKI Docker container for browser access

LinuxKI version 5.5 introduced a new feature where it can start a docker container to open the default browser on the Linux server where LinuxKI is installed to access the Kparse HTML file as well as Visualization Charts and Graphs generated with kiall -V.

10.1 LinuxKI docker scripts for browser access to HTML files and visualization

The Docker container exposes ports 80 and 443 but the SSL certificate is self-signed so the user must add an exception for the certificate if HTTPS is used. The `kivis-start` script starts a container with the exposed ports on random ports to prevent conflicts. It looks up the port mapped to the exposed port 80 and opens the default browser using that port so the user doesn't need to worry about which port is actually being used. The random ports which the exposed ports are mapped to can be discovered using the standard docker commands (e.g., docker ps, docker inspect, docker port) but the kivis-start script outputs the port mapping.

The Docker container image and container which are managed by the `kivis-*` scripts are called **`$USER-linuxki`**. This makes it easy for a user to use the scripts without having to specify the container image name or container name but it also prevents more than one container from being used at a time for each user. The `kivis-start` script will automatically stop any existing container before starting a new one. Containers are automatically removed when they are stopped.

It is also possible to create the Docker container image and create and destroy containers directly from the provided Dockerfile using Docker commands, for those who are familiar with them. The `kivis-*` commands are simple wrappers around the Docker commands for those who aren't familiar with Docker or who just want a simple and short command to view the HTML reports.

LinuxKI includes 3 scripts in the creating and managing the docker container - `kivis-build`, `kivis-start`, and `kivis-stop`.

10.1.1 kivis-build

The `kivis-build` script builds the Docker container linuxki which is used by the LinuxKI visualization tools `kivis-start` and `kivis-stop`. Must be run by a user with sudo privileges or the user must be added to the docker group.

10.1.2 kivis-start

The `kivis-start` script runs the Docker container $USER-linuxki (where $USER is replaced with the current username) and shows the hosted files in the default web browser. The container starts the Apache web server (with PHP extensions) which hosts the files in the current directory. After the container is started, the default browser is opened to show either the kp.*.html page (if there is only one in the current directory or subdirectories) or the files in the current directory. If the -V option is used when kiall is run, then the visualizations will be shown as well.

When finished viewing the reports, run kivis-stop to stop the container.

If the $USER- linuxki container is already running, it will be stopped and a new container will be started.

10.1.3 kivis-stop

The `kivis-stop` script stops the Docker container **linuxki**, which is used by the LinuxKI visualization tools. The container is only stopped if it is running so it is not an error to run this command when the container is not running.

10.2 Example Usage

To use the LinuxKI Docker container for browser access, you will need to have previously LinuxKI reports using the kiall script:

```
$ kiall [-r] [-V]
```

The -V opton will enable the Visualization Charts and Graphs. If the -V option is omitted, the Kparse HTML file can still be viewed in the docker container.

To start the container which will open up the browser to view the files:

```
$ kivis-start
```

When finished viewing the reports, the docker container can be stopped:

```
$ kivis-stop
```

10.3 Special instructions for LinuxKI containers for browser access

This documentation assume the user has basic knowledge of docker containers, including installation and management of dockers. However, a few helpful notes are provided.

10.3.1 Linux browser access

You must be able to start your default Linux browser from your session. So you will need access to a Linux graphical desktop. Running `kivis-start` from your PUTTY window will not work.

10.3.2 non-root users

If it is necessary for non-root users to execute kivis-* scripts, then the user should be added to the docker group in /etc/group.

```
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
```

10.3.3 proxy environment variables

kivis-build supports proxies for HTTP and HTTPS when building the image. The yum commands used when building the container will use the following proxy environment variables:

http_proxy, https_proxy
: If set, the http_proxy and https_proxy variables should contain the URL of the proxy for the HTTP and HTTPS connection respectively. For example: http://proxy.fqdn:8080

no_proxy

If set, this variable should contain a comma-separated list of domain extensions the proxy should not be used for. For example: localhost,127.0.0.1,.dev.net

10.3.4 selinux

Access to the web pages hosted by the container will be blocked by SELinux unless SELinux is in permissive mode or disabled. The sestatus command can be used to check the status of selinux:

```
$ sestatus
SELinux status:                 enabled
```

10.3.5 Test your docker installation

Test that your docker installation is working by running the Docker image hello-world:

```
$ docker run hello-world
```

If you are unable to run the hello-world container, be sure docker is installed, the service is started, and the appropriate proxies are set.

10.3.6 For more information

For more information on Docker, please refer to the following URLs:

https://docs.docker.com/get-started
https://docs.docker.com/config/daemon/systemd/
https://docs.docker.com/install/linux/linux-postinstall/

11 LinuxKI reports and kiinfo reference pages

This section documents the ***kiinfo*** executable with the various subtools, options, and flags used to generate the LinuxKI reports.

11.1 kiinfo collection and analysis program

Kiinfo is one of the primary components of the Linux KI Toolset. It is used to enable and dump the trace data using either *ftrace* or the *LiKI* tracing mechanisms and it is also used to analyze the trace data from either a LinuxKI dump collected with the ***runki*** script or from a running system.

Kiinfo is executed by both the ***runki*** script to collect a LinuxKI dump as well as from the ***kiall*** script to generate the LinuxKI reports as part of the post-processing. Most of the time, using the ***runki*** and ***kiall*** scripts is sufficient. However, there are times when a more customized collection or a customized report is needed. See also kiinfo – customizing reports section. This section documents the various subtools, options, and flags for ***kiinfo***.

Kiinfo works on LinuxKI dump files using the `-ts <timestamp>` option, as well as live systems using the -a <secs> and -p <passes> options. ***Kiinfo*** also requires one of the subtools listed in the Options below, such as -kitrace. See the Examples below for additional details.

For the overall syntax of ***kiinfo***, you can always execute ***kiinfo -help***.

Usage:

```
kiinfo [options ...]

Default: coredir="." if "INDEX" file present else
kernel="/stand/vmunix" core="/dev/kmem"
```

Options:

```
-h | -help
-dump | -kitracedump                # Enable Ftrace and dump debugfs per-cpu ring
                                      buffer data to disk
-likid | -likidump                  # Enable LiKI and dump debugfs per-cpu ring buffer
                                      data to disk. Assumes likit.ko DLKM module is
                                      loaded
-likim | -likimerge                 # Merge ki.bin.<cpu>.<timetamp> files into a single
                                      ki.bin file (LiKI only)
-live | -kilive                     # curses-based user interface
-kparse | -kp [flag, flag, ...]     # first pass analysis tool
-kitrace [flag, flag,...]           # KI ASCII format tool
-kipid [flag, flag, ...]            # KI PID Analysis Report
-kiprof [flag, flag, ...]           # KI CPU Profiling Report
-kidsk [flag, flag, ...]            # KI Disk Analysis Report
-kirunq [flag, flag, ...]           # KI CPU/RUNQ Analysis Report
-kiwait [flag, flag, ...]           # KI Wait Event Analysis Report
-kifile [flag, flag, ...]           # KI File Activity Report
-kifutex [flag, flag, ...]          # KI Futex Activity Report
-kisock [flag, flag, ...]           # KI Network Socket Activity Report
-kidock [flag, flag, ...]           # KI Docker Activity Report
-kiall [flag, flag, ...]            # Generate all KI reports in single pass (except
                                      ASCII trace)
-clparse [flag, flag, ...]          # KI Cluster Report.  Reports on several KI trace
                                      collections from difference nodes in a cluster.
                                      Each KI trace collection must have the same
                                      timestamp.
-h | -help [detail]                 # Help Information/Syntax
-html                               # Output data in HTML format (works best with
                                      -kparse option)
-ts | -timestamp <timestamp>        # Timestamp from ki.all.<timestamp>.tgz file
                                      created by runki script
-start | -starttime <time_in_secs>  # Report only data after the start time
-end | -endtime <time_in_secs>      # Report only data before the end time
-a | -alarm <sec>                   # Length of interval for live analysis using LiKI
                                      tracing mechanism
-p | -passes [count]                # Number of intervals/passes for live analysis
                                      (default 1)
```

Examples:

- Collect LinuxKI trace data using *ftrace* for 30 seconds and provide a "tag" using the current timestamp:

  ```
  $ kiinfo -kitracedump dur=30 -ts `date +%m%d_%H%M`
  ```

- Get kparse html output from files collected by ***runki***:

```
$ kiinfo -kparse -html -ts 0901_1548  >kp.html
```

- Create the Disk Analysis report and also show top 10 tasks performing disk I/O

```
$ kiinfo -kidsk npid=10 -ts 0901_1548
```

- Analyze disk with major=0x68 and minor=0x0 (0x68,0) on live system and show top 5 processes accessing the device. Show results every 5 seconds for 6 passes.

```
$ kiinfo -kidsk dev=0x06800000,npid=5 -a 5 -p 6
```

- Trace a live process for 2 seconds

```
$ kiinfo -kitrace pid=5836 -a 2 >pid.5836
```

- Generate a Cluster Overview Report for several nodes in a cluster with the same timestamp:

```
$ kiinfo -clparse csv,cltree,top=20 -html -ts 0422_1517
```

- Get help information on the -kipid option:

```
$ kiinfo -kipid help
```

- Start curses-based user interface with 5 second updates

```
$ kiinfo -live -a 5
```

11.2 Ftrace dump (kiinfo -kitracedump)

kiinfo -kitracedump is used to collect the kernel trace records using *ftrace*. The -kitracedump subtool will enable the key *ftrace* events, then it will read the debugfs per-cpu ring buffers and dump the trace records to the related ki.bin .<cpu>.<timestamp> files. ***Kiinfo -kitracedump*** is executed by the ***runki*** script to collect the kernel trace data along with other helpful files used for post-processing by other options such as ***kiinfo -kitrace*** and ***kiinfo -kparse***.

Default *ftrace* events enabled are:

- Scheduler records - sched_swtch, sched_wakeup, sched_wakeup_new
- Block IO records - block_rq_insert, block_rq_issue, block_rq_complete, block_rq_requeue
- Power frequency events - power_freq
- System calls - sys_enter, sys_exit

Note that ***kiinfo -kitracedump*** will enable the trace events and then pause for 10 seconds to allow the CPUs to sync up with regards to their time. This helps avoid negative timestamps which can invalidate

some of the data. After 10 seconds, it will print a maker into the *ftrace* ring buffer. Tools such as ***kiinfo -kitrace***, ***kiinfo -kipid***, etc., will ignore all trace records leading up to the marker.

Usage:

```
kiinfo <-kitracedump | -dump> [flags,flags,...] [options ...]
```

Options:

see kiinfo options

Flags:

```
help
debug_dir=<filename>       # debugfs mount point.  By default, uses /sys/kernel/debug.
dur=<seconds>              # number of seconds to collect data.  Default value is 20.
                             This does not include the 10 seconds that kitracedump
                             uses to sync the CPU clocks.
events=[                   #
       default |           # Only trace default events
       all |               # trace all valid ftrace events
       <kitool> |          # Only the events needed for a specific subtool is used.
       <event> ]           # Trace a specific event
subsys=<subsys>            # Enable tracing only for certain subsystems.  Valid
                             subsystems include: power, scsi, block, sched, syscalls,
                             irq.
```

Examples:

- Typical invocation performed by ***runki*** to collect data with ***kiinfo -kitracedump***:

```
$ runki -f
```

- Dump all *ftrace* events for 5 seconds using the default debugfs mount point:

```
$ mount -t debugfs debugfs /sys/kernel/debug
$ kiinfo -dump events=all,dur=5
```

- Note that the subsys and events could be combined. For example, you can trace the same events used by kidsk and also add the scsi related events as well:

```
$ kiinfo -dump events=kidsk,subsys=scsi,dur=20
```

11.3 LiKI Dump (kiinfo -likidump)

kiinfo -likidump is used to collect the kernel trace records using the *LiKI* tracing mechanism. The -likidump option will enable the key *LiKI* trace records, then it will read the debugfs per-cpu ring buffers and dump the trace records to related ki.bin .<cpu>.<timestamp> files. ***Kiinfo -likidump*** is executed by the ***runki*** script to collect the kernel trace data along with other helpful files used for post-processing by other options such as ***kiinfo -kitrace*** and ***kiinfo -kparse***.

Trace events enabled by default are:

- Scheduler records - sched_swtch, sched_wakeup, sched_wakeup_new
- Block IO records - block_rq_insert, block_rq_issue, block_rq_complete, block_rq_requeue
- System calls - sys_enter, sys_exit
- Power frequency events - power_freq
- CPU profiling events - hardclock

In order to use ***kiinfo -likidump***, the debugfs must be mounted and the likit.ko DLKM module must be loaded.

Usage:

```
kiinfo <-likidump | -likid> [flags,flags,...] [options ...]
```

Options:

see kiinfo options

Flags:

```
help
debug_dir=<filename>       # debugfs mount point.  By default, uses /sys/kernel/debug.
pid=<pid>                  # Filter LiKI trace data on pid
tgid=<tgid>                # Filter LiKI trace data on task group
```

```
dev=<dev>                   # Filter LiKI trace data on device
cpu=<cpu>                   # Filter LiKI trace data on CPU
dur=<seconds>               # Number of seconds to collect data.  Default value is 20.
msr                         # Collect adanced CPU statistics such as LLC Hit%,
                              CPI, average CPU frequency, and SMI count
sysignore=<filename>        # Do not trace system calls listed in the <ignore_file>
                              This can reduce trace data by eliminating frequently
                              called system calls, such as getrusage(),gettimeofday(),
                              time(), etc...
events=[                    # Trace events to be traced
       default |            # Only trace default events
       all |                # trace all valid ftrace events
       <kitool> |           # Only the events needed for a specific subtool is used.
       <event> ]            # Trace a specific event
subsys=<subsys>             # Enable tracing only for certain subsystems.  Valid
                              subsystems include: power, scsi, block, sched, syscalls,
                              irq.
```

Examples:

- Typical invocation performed by ***runki*** to collect data using ***kiinfo -likidump***:

```
$ runki
```

- Dump all *LiKI* trace events for 5 seconds using the default debugfs mount point:

```
$ mount -t debugfs debugfs /sys/kernel/debug
$ insmod /opt/linuxki/likit.ko
$ kiinfo -likidump events=all,dur=5
$ rmmod likit.ko
```

- Note that the subsys and events could be combined. For example, you can trace the same events used by kidsk and also add the scsi related events as well:

```
$ kiinfo -likid events=kidsk,subsys=scsi,dur=20
```

- Collect all event data (including non-default events) for specific tasks and ignore the gettimeofday() and getrusage() system calls:

```
$ cat ignore_syscalls
getrusage
gettimeofday

$ kiinfo -likid events=all,pid=1234,pid=62503,sysignore=ignore_syscalls
```

11.4 LiKI Dump Binary File Merge (kiinfo -likimerge)

kiinfo -likim*erge** is used to merge *LiKI* per-CPU binary files into a single LinuxKI binary file, which can then be post-processed by tools such as ***kiinfo -kipid, ***kiinfo -kparse***, etc. Merging the data results in fewer LinuxKI binary files in the working directory and subsequent tools should run faster as the LinuxKI data is already merged. ***Kiinfo -likimerge*** is run on behalf of the ***kiall*** script.

kiinfo -likimerge does not work with the LinuxKI binary files generated using the *ftrace* mechanism.

Usage:

```
kiinfo <-likimerge | -likim> -ts <timestamp> [options]
```

Options:

```
-ts <timestamp>              # The timestamp is required
```

Flags:

```
None
```

Examples:

- Typical invocation performed by ***kiall***:

  ```
  $ kiall
  ```

- Merge the ki.bin.<CPU>.0927_0857 files.

  ```
  $ kiinfo -likimerge -ts 0927_0857
  ```

11.5 Kiinfo curses-based user interface (kiinfo -live)

The ***kiinfo -live*** option uses a curses-based user interface (similar to Glance on HP-UX) to explore the performance data. The LinuxKI data can be analyzed from a running system using the -a <secs> option to update every <secs> seconds, or LinuxKI data can be analyzed from a LinuxKI dump collected from the ***runki*** script using the -ts <timestamp> option. Once ***kiinfo*** is started, the question mark "?" can be entered to see the available commands. Note that global statistics are typically shown with lower case commands, while per-task statistics are typically shown with uppercase commands.

For additional information, please see the Curses-based LinuxKI analysis (kiinfo –live) section.

Usage:

```
kiinfo -live [options ...]
```

Options:

```
See kiinfo options
```

Flags:

```
msr                      # Collect advanced CPU statistics such as LLC Hit%,
                           CPI, average CPU frequency, and SMI count
mangle                   # Leave C++ function names mangled
step=<step_time>         # Floating point time is seconds of the step interval
```

```
sysignore=<filename>      # Do not trace system calls listed in the <ignore_file>
                            This can reduce trace data by eliminating frequently
                            called system calls, such as getrusage(),gettimeofday(),
                            time(), etc...
edus=<filename>           # Specify output of "db2pd -edus" to get DB2 thread names
help                      # print help syntax
```

Examples:

- Analyze data on a running system updating the terminal screen every 5 seconds (default):

  ```
  $ kiinfo -live
  ```

- Analyze data from a LinuxKI dump collected using the ***runki*** script:

  ```
  $ kiinfo -live -ts 0223_1536
  ```

11.6 Kparse System Overview Report (kiinfo -kparse)

The ***kiinfo -kparse*** option produces the Kparse System Overview Report in either text or html format with a variety of information from the LinuxKI binary data created by the ***runki*** script using either *LiKI* trace data or *ftrace* data. The kparse option will attempt to identify various patterns and provides a warning if a condition _may_ be suspected of causing some performance issues and warrants additional investigation.

Usage:

```
kiinfo -kparse [flag,flag,...] [options ...]
```

Options:

```
See kiinfo options
```

Flags:

```
help                # Provided help information for kparse flags
kptree              # When used with kiinfo -kipid pidtree, links PID numbers to
                      kipid output in the PIDS directory
nooracle            # Skip oracle section in kparse output
nofutex             # Do not collect futex statistics
mangle              # Leave C++ function names mangled
blkfrq              # Add Disk Block Frequency stats to Kparse report
lite                # Reduce memory footprint of kiinfo for large collections
                      Some per-thread sections are omitted
edus=<filename>     # Specify output of "db2pd -edus" to get DB2 thread names
jstack=<filename>   # Jstack output file to use (default jstack.<timestamp>)
events=[                    # Trace events to be traced
       default |            # Only trace default events
       all |                # trace all valid ftrace events
       <kitool> |           # Only the events needed for a specific subtool is used
       <event>              # Trace a specific event
subsys=<subsys>     # Enable tracing only for certain subsystems
                      Valid subsystems include: power, scsi, block, sched, syscalls,
```

```
                    irq
```

Examples:

- Running ***kiinfo -kparse*** on LinuxKI data from the ***runki*** script and link PID numbers to the kipid output in the PIDS subdirectory

```
$ kiinfo -kparse kptree -html -ts 1215_0744 >kp.1215_0744.html
```

11.7 KI ASCII trace (kiinfo -kitrace)

kiinfo -kitrace reads kernel trace data from a LinuxKI dump or from a running system (*LiKI* only) and prints each trace record in ASCII format.

Usage:

```
kiinfo -kitrace [flag,flag,...] [options ...]
```

Options:

```
See kiinfo options
```

Flags:

```
help
pid=<pid>                 # Filter on pid
tgid=<tgid>               # Filter on task group ID
cpu=<cpu>                 # Filter on CPU
dev=<dev>                 # Filter on device
nosysenter                # Do not print system call entry record
nosysargs                 # Do not format system call arguments
nomapper                  # Do not print trace records for mapper devices (multipath,
                            lvm)
nomarker                  # Print all trace records, even those before and after the
                            ftrace markers
mangle                    # Leave C++ function names mangled
msr                       # Collect advanced CPU statistics such as LLC Hit%,
                            CPI, average CPU frequency, and SMI count
printcmd                  # Print command with trace record (ftrace only)
seqcnt                    # Print seqcnt in trace records (LiKI on only)
abstime                   # Print absolute time (seconds since boot) for each record
fmttime                   # Print formatted time for each records
                           (i.e.  Wed Feb  5 16:40:15.529100)
epochtime                 # Print time in seconds since the epoch (Jan 1, 1970)

objfile=<filename>        # For runki data collections, use objfile binary to perform
                            symbol table lookups.
sysignore=<filename>      # Do not trace system calls listed in the <ignore_file>
                            This can reduce trace data by eliminating frequently
                            called system calls, such as getrusage(),gettimeofday(),
                            time(), etc...
edus=<filename>           # Specify output of "db2pd -edus" to get DB2 thread names
```

```
events=[                    # Trace events to be traced
       default |            # Only trace default events
       all |                # trace all valid ftrace events
       <kitool> |           # Only the events needed for a specific subtool is used
       <event>              # Trace a specific event
subsys=<subsys>             # Enable tracing only for certain subsystems.  Valid
                              subsystems include: power, scsi, block, sched, syscalls,
                              irq.
csv                         # Create CSV report ki.*.csv
```

Examples:

- Format data from a LinuxKI dump collected from the ***runki*** script printing the system call entry records and formatting the arguments:

  ```
  $ kiinfo -kitrace -ts 1215_0744
  ```

- Format data for a single process from a LinuxKI dump collected from the ***runki*** script:

  ```
  $ kiinfo -kitrace pid=5250 -ts 1215_0744
  ```

- Collect LinuxKI data from a live process for 5 seconds, tracing only the block subsystem trace records:

  ```
  $ kiinfo -kitrace pid=22904,subsys=block -a 5 -p 1
  ```

- Collect LinuxKI data for all events (including non-default events) for all tasks in TGID 67304 and ignore getrusage() and gettimeofday() system calls:

  ```
  $ cat syscall_ignore
  gettimeofday
  getrusage

  $ kiinfo -kitrace tgid=67304,events=all,sysignore=scall_ignore -a 60 -p 5
  ```

Sample Output:

```
Command line: /opt/linuxki/kiinfo -kitrace -ts 0612_1157

kiinfo (4.3)

Linux sut90.atlpss.hp.net 3.10.0-123.el7.x86_64 #1 SMP Mon May 5 11:16:57 EDT 2014
x86_64 x86_64 x86_64 GNU/Linux

KI Binary Version 5
Fri Jun 12 08:57:16 2015
    0.000001 cpu=1 pid=122248 tgid=122248 rt_sigaction [13] entry sig=SIGALRM
*act=0x7fff5f11e490 *oact=0x7fff5f11e530 sigsetsize=8
    0.000001 cpu=1 pid=122248 tgid=122248 rt_sigaction [13] ret=0x0 syscallbeg=
0.000000 sig=SIGALRM *act=0x7fff5f11e490 *oact=0x7fff5f11e530 sigsetsize=8
    0.000001 cpu=1 pid=122248 tgid=122248 alarm [37] entry seconds=50
    0.000002 cpu=1 pid=122248 tgid=122248 alarm [37] ret=0x0 syscallbeg=    0.000000
seconds=50
```

```
    0.000008 cpu=1 pid=122248 tgid=122248 select [23] entry nfds=9
*readfds=0x7fff5f11e6d0 *writefds=0x0 *exceptfds=0x0 *timeout=0x7fff5f11e660
timeout=50.000000 readfds=0x0001 writefds=0x0000 exceptfds=0x0000
    0.000009 cpu=1 pid=122248 tgid=122248 select [23] ret=0x1 syscallbeg=    0.000001
nfds=9 *readfds=0x7fff5f11e6d0 *writefds=0x0 *exceptfds=0x0 *timeout=0x7fff5f11e660
    0.000009 cpu=1 pid=122248 tgid=122248 recvfrom [45] entry sockfd=8
*buf=0x7f79bf1f6060 len=212 flags=0x0 *src_addr=0x0 *addrlen=0x0
```

11.8 PID Analysis Report (kiinfo -kipid)

kiinfo -kipid is used to generate the **PID Analysis Report** on selected or all tasks on the system. By default, ***kiinfo -kipid*** will report on all tasks. You can filter on one or more PIDs as well (***kiinfo -kipid*** pid=100,pid=200,pid=300).

Kiinfo -kipid works on a kernel trace dump collected by ***runki*** script when the -ts <timestamp> option is passed to ***kiinfo***. ***Kiinfo -ki*pid** also works on running systems (*LiKI* only) when using the -a <secs> and -p <passes> options.

The PID Analysis Report is made up of several sections or reports for each PID. Below is a summary of data reported:

********** SCHEDULER ACTIVITY REPORT ********

Reports general scheduling activities, such as time spent sleeping, running, or on the run queue. Also reports context witches and CPU/NODE migrations.

********** CPU MSR REPORT **********

Reports the advanced CPU statistics, such as the Last Level Cache Hit rate, Cycles per Instruction (CPI), average CPU frequency, and the System Management Interrupt count.

******** PID RUNQ LATENCY REPORT ********

Print when the rqhist flag is used. Reports how much of the RunQ time is due to waiting on another process/task (RunQPri) and how much of the time is just due to the transition from the idle task (PID 0) to the target task. Also adds a runq latency histogram on a per-CPU basis as well as other per-CPU information, such as the average RunQ latency per CPU in Usecs (microseconds).

******** COOPERATING/COMPETING TASKS REPORT ********

Identified which tasks have woken up the specific task and how long the specific task was asleep before waking up. Also identifies tasks that have been woken up by the specific task and how long they have been sleeping before they were woken up. The coop flag reports additional detail about thread wait states, blocking syscalls, which threads are involved in waking the sleeper, etc. When viewing the data generated with the coop flag, be sure to use a wide screen (180-200 columns).

******** SLEEP REPORT ********

Provides list of functions where the task goes to sleep, the count and amount of time sleeping in each function. Also shows the kernel stacktraces for the sleep events. The Sleep Report is only printed if the *LiKI* tracing mechanism is used as *ftrace* does not log the sleep function.

In LinuxKI 5.0, the RunQ stack traces were included in this section. While the threads are not technically "sleeping", they are waiting for CPU in order to run.

********** CPU ACTIVITY REPORT **********

Examines the HARDCLOCK traces to identify where the threads are taking CPU time. Includes stack traces. (*LiKI* only)

********** SYSTEM CALL REPORT **********

Report lists each system call and how much time was spent in each system call. For each system call, the amount of time spent in sleeping, running, and on the CPU runq is also printed. If the trace data is from *LiKI*, then the SLEEP time is broken out into kernel functions where sleep is called.

********** FUTEX REPORT **********

Reports on activity on the top Fast User-space muTEXes (futex) for the process. By default, the top 10 futexes are listed.

********** FILE ACTIVITY REPORT **********

Reports system call activity for each file. Additional information for each system call is listed, similar to the SYSTEM CALL REPORT.

********** PHYSICAL DEVICE REPORT **********

Reports physical IO statistics for each physical device

********** DEVICE-MAPPER REPORT **********

Reports IO statistics for each multipath device-mapper device.

Usage:

```
kiinfo -kipid [flag,flag,...] [options ...]
```

Options:

```
see kiinfo options
```

Flags:

```
help
pid=<pid>                   # Filter on pid
tgid=<tgid>                 # Filter on task group
nsym=<nsym>                 # Reports <nsym> kernel functions for switch and hardclock
                              records (LiKI only)
nfutex=<nfutex>             # Limit the number of futexes listed for each task
                              (default 10)
npid=<npid>                 # Top <npid> tasks woken up or tasks that did wakeups in
                              Schedule Activity Report (default 10)
coop                        # Include additionl details on cooperating/competing
mangle                      # Leave C++ function names mangled
msr                         # Collect adanced CPU statistics such as LLC Hit%,
                              CPI, average CPU frequency, and SMI count
oracle                      # Report on Oracle processes
pidtree                     # Create PIDS subdirectory and kipid output for each
                              process. Used with kiinfo -kparse kptree
rqhist                      # Include RunQ Histogram
nofutex                     # Do not collect futex statistics
kitrace                     # Include formatted ASCII trace records prior to the
                              standard report
nosysenter                  # Do not print syscall entry records when using kitrace
                              flag
abstime                     # Print absolute time (seconds since boot) for each record
                              when using kitrace flag
fmttime                     # Print formatted time for each record when using kitrace
                              flag (i.e.  Wed Feb  5 16:40:15.529100)
epochtime                   # Print time in seconds since the epoch (Jan 1, 1970)
sysignore=<filename>        # Do not trace system calls listed in the <ignore_file>
                              This can reduce trace data by eliminating frequently
                              called system calls, such as getrusage(),gettimeofday(),
                              time(), etc...
objfile=<filename>          # Uses objfile to perform USER hardclock lookup. pid= or
                              oracle option must be used.
edus=<filename>             # Specify output of "db2pd -edus" to get DB2 thread names
jstack=<filename>           # Jstack output file to use (default jstack.<timestamp>)
events=[                    # Trace events to be traced
       default |            # Only trace default events
       all |                # trace all valid ftrace events
       <kitool> |           # Only the events needed for a specific subtool is used.
       <event> ]            # Trace a specific event
subsys=<subsys>             # Enable tracing only for certain subsystems.  Valid
                              subsystems include: power, scsi, block, sched, syscalls,
                              irq.
report=<[-]asxhfopnm>       # Customize sections of the PID Analysis Report to show
         - - Omit Reports
         a - Scheduler Activity Report
         s - System Call Report
         x - Futex Report
         h - CPU Activity Report
         f - File Activity Report
         o - Network/Socket Activity Report
         p - Physical Volume Report
         m - Memory Report
csv                         # Create CSV report kipid.*.csv
```

Examples:

- Reading from a kitrace binary file using the -ts option

Note that when the -ts option is used, ***kiinfo -kipid*** will attempt to open the lsof.{timestamp} file to resolve filenames in the File Activity Report, and open the ps-eLf.{timestamp} file to resolve process names. This is the recommended option for running on a ki.bin.* file. The command below also creates the PIDS subdirectory (pidtree flag) for use with ***kiinfo -kparse kptree***.

```
$ kiinfo -kipid npid=10,pidtree -ts 0901_1548 >kipid.0901_1548.txt
```

1. Create a PID report for a specific process with runq histograms (rqhist flag). Show up to 20 sleep functions or profile functions and 20 stacktraces for each (nsym=20 flag). Also include additional details about the cooperating/competing processes (coop flag)

```
$ kiinfo -kipid pid=26649,rqhist,nsym=20,coop -ts 0814_2040
```

- Create a PID report for a specific online process for 30 seconds:

```
$ kiinfo -kipid pid=5683,nsym=10,npid=5 -a 30 -p 1
```

Sample Output:

Note that the output in red is only available with the *LiKI* tracing mechanism is used:

```
Command line: /opt/linuxki/kiinfo -kipid pid=15572,nsym=5,npid=5 -ts 0816_0839

kiinfo (2.0)

KI Binary Version 3
Linux gwr-repo1.rose.hp.com 2.6.32-358.2.1.el6.x86_64 #1 SMP Wed Feb 20 12:17:37 EST
2013 x86_64 x86_64 x86_64 GNU/Linux


PID 15572  /home/mcr/bin/iotest8
  PPID 1  /sbin/init

    ********* SCHEDULER ACTIVITY REPORT ********
    RunTime    :  1.478732  SysTime    :  1.400001   UserTime    :  0.073428
    SleepTime  : 17.883474  Sleep Cnt :     37491   Wakeup Cnt :     29503
    RunQTime   :  0.602830  Switch Cnt:     37504   PreemptCnt :        13
    HardIRQ    :  0.000114  HardIRQ-S :  0.000098    HardIRQ-U :  0.000016
    SoftIRQ    :  0.005190  SoftIRQ-S :  0.004943    SoftIRQ-U :  0.000247
    Last CPU   :         2  CPU Migrs :       305   NODE Migrs :         9
    Policy     : SCHED_NORMAL     vss :      1015          rss :       149

    busy   :      7.41%
      sys  :      7.01%
      user :      0.37%
      irq  :      0.03%
    runQ   :      3.02%
    sleep  :     89.60%

    ******** COOPERATING/COMPETING TASKS REPORT ********

    Tasks woken up by this task (Top 5)
           PID    Count   SlpPcnt     Slptime  Command
         15573     5736     -nan%    0.000000  /home/mcr/bin/iotest8
         15574     5049     -nan%    0.000000  /home/mcr/bin/iotest8
```

```
         15577     4989     -nan%    0.000000  /home/mcr/bin/iotest8
         15575     4855     -nan%    0.000000  /home/mcr/bin/iotest8
         15578     4422     -nan%    0.000000  /home/mcr/bin/iotest8

    Tasks that have woken up this task(Top 5)
           PID    Count   SlpPcnt     Slptime  Command
            -1    29197    28.74%    5.139739   ICS
         15574     4444    11.40%    2.038824  /home/mcr/bin/iotest8
         15573     3576    11.92%    2.131962  /home/mcr/bin/iotest8
         15575       99    18.46%    3.301462  /home/mcr/bin/iotest8
         15576       78    13.04%    2.331323  /home/mcr/bin/iotest8

    ******** SLEEP REPORT ********

    Kernel Functions calling sleep() - Top 5 Functions
       Count     Pct    SlpTime    Slp% TotalTime%   Msec/Slp   MaxMsecs  Func
        8294  22.12%    12.7437  71.26%     63.83%      1.536    198.951
__mutex_lock_slowpath
       29196  77.87%     5.1397  28.74%     25.74%      0.176     30.659  io_schedule

    Process Sleep stack traces (sort by % of total wait time) - Top 5 stack traces
       count    wpct      avg   Stack trace
                  %     msecs
    ================================================================
        5415  47.77     1.578   __mutex_lock_slowpath  mutex_lock
generic_file_aio_write  ext4_file_write  do_sync_write  vfs_write  sys_write  tracesys
       29196  28.74     0.176   io_schedule  __blockdev_direct_IO_newtrunc
__blockdev_direct_IO  ext4_ind_direct_IO  ext4_direct_IO  generic_file_direct_write
__generic_file_aio_write  generic_file_aio_write  ext4_file_write  do_sync_write
vfs_write  sys_write  tracesys
        2879  23.49     1.459   __mutex_lock_slowpath  mutex_lock  ext4_llseek
vfs_llseek  sys_lseek  tracesys

    ******** CPU ACTIVITY REPORT ********
    The percentages below reflect the percentage
    of the Thread's total RunTime spent in either
    User code or System code
    RunTime:   1.4787

      Count    USER     SYS    INTR
        465       8     457       0
              1.72%  98.28%   0.00%

    HARDCLOCK entries
       Count     Pct  State  Function
          76  16.34%  SYS    scsi_request_fn
          58  12.47%  SYS    _spin_unlock_irqrestore
          51  10.97%  SYS    finish_task_switch
          25   5.38%  SYS    blk_queue_bio
          14   3.01%  SYS    mutex_spin_on_owner

       Count     Pct  HARDCLOCK Stack trace
       ============================================================
          64  13.76%  scsi_request_fn  scsi_request_fn  native_sched_clock
__blk_run_queue  elv_insert  __elv_add_request  blk_insert_cloned_request
dm_dispatch_request  dm_request_fn  __generic_unplug_device  generic_unplug_device
dm_unplug_all  blk_unplug  dm_table_unplug_all  trace_nowake_buffer_unlock_commit
dm_unplug_all
          36   7.74%  _spin_unlock_irqrestore  try_to_wake_up  wake_up_process
__mutex_unlock_slowpath  mutex_unlock  generic_file_aio_write  ext4_file_write
do_sync_write  autoremove_wake_function  native_sched_clock  sched_clock
```

```
trace_nowake_buffer_unlock_commit  security_file_permission  vfs_write  sys_write
tracesys
          16   3.44%  _spin_unlock_irqrestore  qla24xx_start_scsi  scsi_done
qla2xxx_queuecommand  scsi_dispatch_cmd  scsi_request_fn  native_sched_clock
__blk_run_queue  elv_insert  __elv_add_request  blk_insert_cloned_request
dm_dispatch_request  dm_request_fn  __generic_unplug_device  generic_unplug_device
dm_unplug_all
          13   2.80%  blk_queue_bio  blk_queue_bio  trace_nowake_buffer_unlock_commit
dm_request  generic_make_request  submit_bio  dio_bio_submit
__blockdev_direct_IO_newtrunc  __find_get_block  __getblk  __blockdev_direct_IO
ext4_get_block  find_get_pages  ext4_ind_direct_IO  ext4_get_block  ext4_direct_IO
          12   2.58%  scsi_request_fn  scsi_request_fn  native_sched_clock
__blk_run_queue  elv_insert  __elv_add_request  blk_insert_cloned_request
dm_dispatch_request  dm_request_fn  __blk_run_queue  cfq_insert_request  elv_insert
__elv_add_request  blk_queue_bio  trace_nowake_buffer_unlock_commit  dm_request

    ******** SYSTEM CALL REPORT ********
    System Call Name     Count     Rate     ElpTime        Avg        Max    Errs
AvSz     KB/s
    write                29196   1460.3   15.609013   0.000535   0.214018       0
4096   5841.3
       SLEEP             34611   1731.2   13.682159   0.000395
          Sleep Func     10830            17.084840   0.001578   0.198951
__mutex_lock_slowpath
          Sleep Func     58392            10.279479   0.000176   0.030659  io_schedule
       RUNQ                                 0.581584
       CPU                                  1.345270
    lseek                29196   1460.3    4.282247   0.000147   0.157017       0
       SLEEP              2880    144.1    4.201314   0.001459
          Sleep Func      5758             8.402629   0.001459   0.147119
__mutex_lock_slowpath
       RUNQ                                 0.021228
       CPU                                  0.059760

    ******** FILE ACTIVITY REPORT ********
    FD: 3 REG      dev: 0xfd00006 /work/mcr/bigfile
    System Call Name     Count     Rate     ElpTime        Avg
    write                29196   1460.3   15.609013   0.000535   0.214018       0
4096   5841.3
       SLEEP             34611   1731.2   13.682159   0.000395
          Sleep Func      5415             8.542420   0.001578   0.198951
__mutex_lock_slowpath
          Sleep Func     29196             5.139739   0.000176   0.030659  io_schedule
       RUNQ                                 0.581584
       CPU                                  1.345270
    lseek                29196   1460.3    4.282247   0.000147   0.157017       0
   SLEEP              2880    144.1    4.201314   0.001459
          Sleep Func      2879             4.201314   0.001459   0.147119
__mutex_lock_slowpath
       RUNQ                                 0.021228
       CPU                                  0.059760

    ******** PHYSICAL DEVICE REPORT ********
          device   rw  avque avinflt   io/s   Kb/s  avsz  avwait  avserv
      0x00800070  /dev/sdh
        0x00800070  r   0.00    0.00      0      0     0    0.00    0.00
        0x00800070  w   0.50    0.05    731   2923     4    0.00    0.19
        0x00800070  t   0.50    0. 0  14610      0      0     0.0    30.7
      0x008000a0  /dev/sdk
        0x008000a0  r   0.00    0.00      0      0     0    0.00    0.00
        0x008000a0  w   0.50    0.07    729   2918     4    0.00    0.18
```

```
      0x008000a0  t   0.50    0.07    729   2918     4    0.00    0.18

******** DEVICE-MAPPER REPORT ********
        device   rw  avque avinflt   io/s   Kb/s  avsz  avwait  avser
    0x0fd00003   /dev/mapper/mpathb  ->  /dev/dm-3
      0x0fd00003  r   0.00    0.00      0      0     0    0.00    0.00
      0x0fd00003  w   1.40    0.13   1460   5841     4    0.00    0.19
      0x0fd00003  t   1.40    0.13   1460   5841     4    0.00    0.19

Totals:
  Physical Writes:
    Cnt   :  29193   Total Kb:  116772.0    ElpTime:   5.34646
    Rate  : 1460.2   Kb/sec  :    5840.7    AvServ :   0.00018
    Errs:       0   AvgSz   :       4.0    AvWait :   0.00000
    Requeue :   0   MaxQlen :         1
```

11.9 CPU Profiling Report (kiinfo -kiprof)

Kiinfo -kiprof is used to generate the **CPU Profiling Report** using the hardclock trace events from the *LiKI* tracing. By profiling the kernel every 10 milliseconds, frequently accessed kernel functions and traces can be identified to understand how the CPUs and tasks are spending their time.

Note
CPUs will not log a hardclock trace event if the CPU is in a deep c-state wait. The hardclock trace vent will only be logged as a CPU completes the MWAIT CPU idle-wait instruction, which may be longer than 10 milliseconds.

kiprof will report the following sections:

********** GLOBAL HARDCLOCK REPORT **********

Reports total number of HARDCLOCK trace records by type (USER, SYS, INTR, IDLE), followed by the kernel function executed during the profile and their respective stack traces.

********** PERCPU HARDCLOCK REPORT **********

Similar to the Global Hardclock report, but broken out for each CPU.

********** PER-PROCESS HARDCLOCK REPORT **********

Reports counts of kernel functions broken out for each process (PID). Top <npid> processes listed (default 20)

Usage:

```
kiinfo -kiprof [flag,flag,...] [options ...]
```

Options:

```
see kiinfo options
```

Flags:

```
help
pid=<pid>          # Filter on pid
tgid=<tgid>        # Filter on task group ID
cpu=<cpu>          # Filter on CPU
npid=<npid>        # Report Top Pids using most CPU
nsym=<nsym>        # Report only top nsym kernel symbols
kitrace            # Include formatted ASCII trace records prior to the standard
                     report
abstime            # Print absolute time (seconds since boot) for each record when
                     using kitrace flag
fmttime            # Print formatted time for each record when using kitrace flag
                     (i.e.  Wed Feb  5 16:40:15.529100)
epochtime          # Print time in seconds since the epoch (Jan 1, 1970)
objfile=<objfile>  # Uses objfile to perform USER hardclock lookup. pid= filter must
                     be used.
edus=<filename>    # Specify output of "db2pd -edus" to get DB2 thread names
jstack=<filename>  # Jstack output file to use (default jstack.<timestamp>)
```

Examples:

- Reading from a kitrace binary file, display top 50 functions and stack traces and top 20 PIDs:

  ```
  $ kiinfo -kiprof nsym=50,npid=20 -ts 0725_0733
  ```

- Reading from a live system for a specific PID and report every 10 seconds, for 3 passes:

  ```
  $ kiinfo -kiprof pid=26649 -a 10 -p 3
  ```

Sample output

```
Command line: kiinfo -kiprof nsym=5,npid=4 -ts 0816_0839

kiinfo (2.0)

KI Binary Version 3
Linux gwr-repo1.rose.hp.com 2.6.32-358.2.1.el6.x86_64 #1 SMP Wed Feb 20 12:17:37 EST
2013 x86_64 x86_64 x86_64 GNU/Linux

NOTE: idle hardclock traces are not always logged.


******** GLOBAL HARDCLOCK REPORT ********
  Count    USER     SYS    INTR    IDLE
   4804     669    2936      38    1161

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Kernel Functions executed during profile
   Count     Pct  State  Function
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    1155  24.04%  IDLE   native_safe_halt
     882  18.36%  SYS    finish_task_switch
```

```
     667  13.88%  USER   UNKNOWN
     276   5.75%  SYS    copy_user_generic_string
     226   4.70%  SYS    scsi_request_fn

non-idle GLOBAL HARDCLOCK STACK TRACES (sort by count):

   Count     Pct  Stack trace
============================================================
     545  11.34%  finish_task_switch
     164   3.41%  scsi_request_fn  scsi_request_fn  native_sched_clock
__blk_run_queue  elv_insert  __elv_add_request  blk_insert_cloned_request
dm_dispatch_request  dm_request_fn  __generic_unplug_device  generic_unplug_device
dm_unplug_all  blk_unplug  dm_table_unplug_all  trace_nowake_buffer_unlock_commit
dm_unplug_all
     161   3.35%  copy_user_generic_string  cpu_buf_read  security_file_permission
vfs_read  sys_read  tracesys
      97   2.02%  copy_user_generic_string  __copy_from_user_inatomic
iov_iter_copy_from_user_atomic  generic_file_buffered_write  ext4_dirty_inode
__generic_file_aio_write  generic_file_aio_write  ext4_file_write  do_sync_write
autoremove_wake_function  native_sched_clock  sched_clock
trace_nowake_buffer_unlock_commit  security_file_permission  vfs_write  sys_write
      89   1.85%  _spin_unlock_irqrestore  try_to_wake_up  wake_up_process
__mutex_unlock_slowpath  mutex_unlock  generic_file_aio_write  ext4_file_write
do_sync_write  autoremove_wake_function  native_sched_clock  sched_clock
trace_nowake_buffer_unlock_commit  security_file_permission  vfs_write  sys_write
tracesys

******** PERCPU HARDCLOCK REPORT ********
  CPU   Count    USER     SYS    INTR    IDLE
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    0     649      72     433       5     139
    1     380      46     201      20     113
    2    1101      18     984      10      89
    3     285      22     156       1     106
    4     250      39     132       0      79
    5     622     258     250       2     112
    6     378      19     278       0      81
    7     249      72      86       0      91
    8      82      19      26       0      37
    9     189      20      82       0      87
   10     281      14     206       0      61
   11      94      26      26       0      42
   12      51      15      12       0      24
   13      79      12      16       0      51
   14      73       8      37       0      28
   15      41       9      11       0      21

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Kernel Functions for CPU  0
Sample count is 649/4804 -- Percent for this CPU is   13.51
   Count     Pct  State  Function
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
     138  21.26%  IDLE   native_safe_halt
     115  17.72%  SYS    finish_task_switch
      72  11.09%  USER   UNKNOWN
      51   7.86%  SYS    scsi_request_fn
      47   7.24%  SYS    _spin_unlock_irqrestore

non-idle CPU 0  HARDCLOCK STACK TRACES (sort by count):

   Count     Pct  Stack trace
```

```
============================================================
      70  10.79%  finish_task_switch
      47   7.24%  scsi_request_fn  scsi_request_fn  native_sched_clock
__blk_run_queue  elv_insert  __elv_add_request  blk_insert_cloned_request
dm_dispatch_request  dm_request_fn  __generic_unplug_device  generic_unplug_device
dm_unplug_all  blk_unplug  dm_table_unplug_all  trace_nowake_buffer_unlock_commit
dm_unplug_all
      29   4.47%  _spin_unlock_irqrestore  try_to_wake_up  wake_up_process
__mutex_unlock_slowpath  mutex_unlock  generic_file_aio_write  ext4_file_write
do_sync_write  autoremove_wake_function  native_sched_clock  sched_clock
trace_nowake_buffer_unlock_commit  security_file_permission  vfs_write  sys_write
tracesys
      12   1.85%  finish_task_switch  thread_return  trace_nowake_buffer_unlock_commit
prepare_to_wait  cfq_kick_queue  worker_thread  autoremove_wake_function
worker_thread  kthread  child_rip  kthread  child_rip
      11   1.69%  _spin_unlock_irqrestore  qla24xx_start_scsi  scsi_done
qla2xxx_queuecommand  scsi_dispatch_cmd  scsi_request_fn  native_sched_clock
__blk_run_queue  elv_insert  __elv_add_request  blk_insert_cloned_request
dm_dispatch_request  dm_request_fn  __generic_unplug_device  generic_unplug_device
dm_unplug_all

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Kernel Functions for CPU  1
Sample count is 380/4804 -- Percent for this CPU is    7.91
   Count     Pct  State  Function
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
     113  29.74%  IDLE   native_safe_halt
      62  16.32%  SYS    finish_task_switch
      46  12.11%  USER   UNKNOWN
      26   6.84%  SYS    _spin_unlock_irqrestore
      14   3.68%  SYS    scsi_request_fn

non-idle CPU 1  HARDCLOCK STACK TRACES (sort by count):

   Count     Pct  Stack trace
============================================================
      38  10.00%  finish_task_switch
      14   3.68%  scsi_request_fn  scsi_request_fn  native_sched_clock
__blk_run_queue  elv_insert  __elv_add_request  blk_insert_cloned_request
dm_dispatch_request  dm_request_fn  __generic_unplug_device  generic_unplug_device
dm_unplug_all  blk_unplug  dm_table_unplug_all  trace_nowake_buffer_unlock_commit
dm_unplug_all
       9   2.37%  _spin_unlock_irqrestore  try_to_wake_up  wake_up_process
__mutex_unlock_slowpath  mutex_unlock  generic_file_aio_write  ext4_file_write
do_sync_write  autoremove_wake_function  native_sched_clock  sched_clock
trace_nowake_buffer_unlock_commit  security_file_permission  vfs_write  sys_write
tracesys
       7   1.84%  copy_user_generic_string  cpu_buf_read  security_file_permission
vfs_read  sys_read  tracesys
       5   1.32%  _spin_unlock_irqrestore  qla24xx_intr_handler  handle_IRQ_event
handle_fasteoi_irq  handle_irq  do_IRQ  ret_from_intr  native_safe_halt  default_idle
c1e_idle  cpu_idle  start_secondary

                    :                                          :

******** PER-PROCESS HARDCLOCK REPORT ********

Pid: 0       Sys/Count:     552/2936    ( 18.80%)  Command: (null)
------------------------------------------------------------------
  Count    USER     SYS    INTR
   1736       0     552      23
```

```
------------------------------------------------------------
   Count    %Pid  State  Function
    1155  66.53%  IDLE   native_safe_halt
     545  31.39%  SYS    finish_task_switch
      12   0.69%  INTR   _spin_unlock_irqrestore
       6   0.35%  SYS    thread_return
       4   0.23%  IDLE   cpu_idle
       3   0.17%  INTR   handle_IRQ_event
       2   0.12%  IDLE   tick_nohz_stop_sched_tick
       2   0.12%  INTR   rb_reserve_next_event
       1   0.06%  INTR   scsi_decide_disposition
       1   0.06%  INTR   __do_softirq

Pid: 15572   Sys/Count:     457/2936    ( 15.57%)  Command: /home/mcr/bin/iotest8
------------------------------------------------------------
  Count    USER     SYS    INTR
    465       8     457       0
------------------------------------------------------------
   Count    %Pid  State  Function
      76  16.34%  SYS    scsi_request_fn
      58  12.47%  SYS    _spin_unlock_irqrestore
      51  10.97%  SYS    finish_task_switch
      25   5.38%  SYS    blk_queue_bio
      14   3.01%  SYS    mutex_spin_on_owner
      13   2.80%  SYS    trace_clock_local
      13   2.80%  SYS    __rb_reserve_next
       9   1.94%  SYS    rb_reserve_next_event
       9   1.94%  SYS    __lookup
       8   1.72%  SYS    ring_buffer_lock_reserve

Pid: 15574   Sys/Count:     419/2936    ( 14.27%)  Command: /home/mcr/bin/iotest8
------------------------------------------------------------
  Count    USER     SYS    INTR
    423       4     419       0
------------------------------------------------------------
   Count    %Pid  State  Function
      64  15.13%  SYS    scsi_request_fn
      52  12.29%  SYS    _spin_unlock_irqrestore
      40   9.46%  SYS    finish_task_switch
      21   4.96%  SYS    blk_queue_bio
      12   2.84%  SYS    mutex_spin_on_owner
       7   1.65%  SYS    rb_end_commit
       7   1.65%  SYS    __blockdev_direct_IO_newtrunc
       7   1.65%  SYS    qla2xxx_queuecommand
       6   1.42%  SYS    trace_clock_local
       6   1.42%  SYS    ring_buffer_unlock_commit
```

11.10 Disk Analysis Report (kiinfo -kidsk)

kiinfo -kidsk generates the **Disk Analysis Report** which reports IO related statistics. There are 6 main sections reported:

- List of IOs with service times greater a given number of milliseconds (default 1000).
- Per-disk IO statistics including IO service times, average IO size, and throughput. Statistics are reported for read IO, write IO, and total IO. SCSI devices are listed first, followed by mapper block devices.
- Multipath FC HBA statistics.

- IO service time histogram.
- Report on Top <npid> PIDs doing multipath IO. Only done if npid= flag is used
- Report on Top <npid> PIDs doing physical IO. Only done if npid= flag is used.

Keep in mind that all of the traces are measured at the block device level (block_rq_insert, block_rq_issue, and block_rq_complete).

Usage:

```
kiinfo -kidsk [flags,flags,...] [options ...]
```

Options:

```
see kiinfo options
```

Flags:

```
help
dev=<dev>           # Filter on device
pid=<pid>           # Filter on pid
tgid=<tgid>         # Filter on task group
npid=<npid>         # Report on top <npid> PIDs doig I/O
percpu              # Provide per-cpu stats where IOs are completed
nodev               # Omit the per-device statistics
nomapper            # Omit block device mapper statistics.
mpath_detail        # Provide per-cpu stats for each device
detail=<0|1>        #   detail=0  - Do not show individual IOs
                        detail=1  - Print IO on single line (default)
kitrace             # Include formatted ASCII trace records prior to the standard
                      report
abstime             # Print absolute time (seconds since boot) for each record when
                      using kitrace flag
fmttime             # Print formatted time for each record when using kitrace flag
                      (i.e.  Wed Feb  5 16:40:15.529100)
epochtime           # Print time in seconds since the epoch (Jan 1, 1970)
bkfname=<filename>  # Used to change histogram values detail=<detail>
                      default shows IOs that exceed 1000 msec
edus=<filename)     # Specify output of "db2pd -edus" to get DB2 thread names
jstack=<filename>   # Jstack output file to use (default jstack.<timestamp>)
csv                 # Create CSV report kidsk.*.csv
```

Examples:

- Create Disk Activity Report with Top 5 PIDs doing physical IO:

  ```
  $ kiinfo -kidsk npid=5 -ts 0712_0838
  ```

- Get statistics on a physical device and display top 10 PIDs accessing the device:

  ```
  $ kiinfo -kidsk dev=0x042000c0,npid=10 -ts 0901_1743
  ```

  You can check the major and minor numbers of the device to create the device number:

```
$ ll /dev/sdas
brw-rw---- 1 root disk 66, 192 Aug  4 10:51 /dev/sdas
```

Note the major number is 66 or 0x42. The minor number is 192 or 0xc0. The device number is the combination of the major and minor numbers in hexadecimal. In this case - 0x042000c0.

- Changing the msec times in the Physical I/O Histogram array

```
$ echo "1 2 4 8 16 32 64 100" >buckets
$ cat buckets
1 2 4 8 16 32 64 100

$ kiinfo -kidsk bkfname=buckets -ts 0901_1753
          :                    :
Physical I/O Histogram
     msecs <1    <2    <4    <8     <16    <32    <64    <100   >100
    Write: 0     0     4     14     5      0      0      0      0
    Read:  0     0     0      2     0      0      0      0      0
```

- Collect disk statistics every 30 seconds for 60 minutes on a live system to looks for IOs that take more than 1000 msecs. Don't print device or device mapper statistics or the top pids:

```
$ kiinfo -kidsk nodev,nomapper,npid=0 -a 30 -p 120 >kidsk.txt
```

Sample Output:

```
Command line: /opt/linuxki/kiinfo -kidsk npid=20,csv -ts 0409_1620

kiinfo (5.3i)

Linux saphana001 4.4.74-92.38-default #1 SMP Tue Sep 12 19:43:46 UTC 2017 (545c055)
x86_64 x86_64 x86_64 GNU/Linux


Physical Device Statistics

      device rw  avque avinflt   io/s   KB/s  avsz   avwait   avserv    tot    seq
rnd  reque  flush maxwait maxserv
0x00800020  /dev/sdc  (HW path: 0:0:0:52)   (mpath device: /dev/mapper/TW1_data)
  0x00800020  r   0.50    0.80     16 269567 16337     0.00    79.72    330     14
317      0      0     0.0   131.3
  0x00800020  w   0.50    1.00      0     13   256     0.00     0.93      1      0
1      0      0     0.0     0.9
  0x00800020  t   0.50    0.80     17 269579 16288     0.00    79.48    331     14
318      0      0     0.0   131.3
0x00800080  /dev/sdi  (HW path: 0:0:1:52)   (mpath device: /dev/mapper/TW1_data)
  0x00800080  r   0.50    0.80     16 268287 16309     0.00    77.70    329     10
320      0      0     0.0   126.7
  0x00800080  w   0.50    1.00      0     14    92     0.00     0.99      3      0
3      0      0     0.0     1.5
  0x00800080  t   0.50    0.80     17 268300 16162     0.00    77.01    332     10
323      0      0     0.0   126.7
0x008000e0  /dev/sdo  (HW path: 0:0:2:52)   (mpath device: /dev/mapper/TW1_data)
  0x008000e0  r   0.50    0.75     16 266187 16380     0.00    75.65    325     12
314      0      0     0.0   125.0
```

```
  0x008000e0  w   0.50    1.00      0     14   136     0.00     0.80      2      0
2      0      0     0.0     1.0
  0x008000e0  t   0.50    0.75     16 266201 16281     0.00    75.19    327     12
316      0      0     0.0   125.0
0x04100040  /dev/sdu  (HW path: 0:0:3:52)   (mpath device: /dev/mapper/TW1_data)
  0x04100040  r   0.50    0.76     16 269208 16365     0.00    78.66    329     13
317      0      0     0.0   122.9
  0x04100040  w   0.50    0.75      0     13    67     0.00     0.67      4      0
4      0      0     0.0     0.9
  0x04100040  t   0.50    0.76     17 269222 16169     0.00    77.72    333     13
321      0      0     0.0   122.9
0x041000a0  /dev/sdaa  (HW path: 0:0:4:52)   (mpath device: /dev/mapper/TW1_data)
  0x041000a0  r   0.50    0.77     16 266904 16324     0.00    80.41    327     12
316      0      0     0.0   127.4
  0x041000a0  w   0.50    1.00      0     13    88     0.00     0.72      3      0
3      0      0     0.0     1.0
  0x041000a0  t   0.50    0.77     16 266917 16176     0.00    79.69    330     12
319      0      0     0.0   127.4
0x04200000  /dev/sdag  (HW path: 0:0:5:52)   (mpath device: /dev/mapper/TW1_data)
  0x04200000  r   0.50    0.75     16 265880 16361     0.00    78.95    325     14
312      0      0     0.0   126.9
  0x04200000  w   0.50    1.00      0     13   130     0.00     1.18      2      0
2      0      0     0.0     1.3
  0x04200000  t   0.50    0.75     16 265893 16262     0.00    78.47    327     14
314      0      0     0.0   126.9
0x04200060  /dev/sdam  (HW path: 1:0:0:52)   (mpath device: /dev/mapper/TW1_data)
  0x04200060  r   0.50    0.76     16 263781 16384     0.00    83.78    322     10
314      0      0     0.0   178.3
  0x04200060  w   0.50    0.67      0     13    88     0.00     0.61      3      0
3      0      0     0.0     1.0
  0x04200060  t   0.50    0.76     16 263794 16233     0.00    83.01    325     10
317      0      0     0.0   178.3
0x042000c0  /dev/sdas  (HW path: 1:0:1:52)   (mpath device: /dev/mapper/TW1_data)
  0x042000c0  r   0.50    0.79     16 257228 16384     0.00    84.83    314      6
310      0      0     0.0   241.7
  0x042000c0  w   0.50    0.50      0      0     4     0.00     1.98      2      0
2      0      0     0.0     3.4
  0x042000c0  t   0.50    0.79     16 257228 16280     0.00    84.31    316      6
312      0      0     0.0   241.7
0x04300020  /dev/sday  (HW path: 1:0:2:52)   (mpath device: /dev/mapper/TW1_data)
  0x04300020  r   0.50    0.73     16 260504 16384     0.00    85.29    318      7
312      0      0     0.0   190.0
  0x04300020  w   0.50    1.00      0     38   256     0.00     1.01      3      0
3      0      0     0.0     1.1
  0x04300020  t   0.50    0.73     16 260543 16233     0.00    84.51    321      7
315      0      0     0.0   190.0
0x04300080  /dev/sdbe  (HW path: 1:0:3:52)   (mpath device: /dev/mapper/TW1_data)
  0x04300080  r   0.50    0.73     16 263781 16333     0.00    84.72    323     11
313      0      0     0.0   185.8
  0x04300080  w   0.50    0.80      0     39   155     0.00     1.46      5      0
5      0      0     0.0     3.3
  0x04300080  t   0.50    0.73     16 263820 16086     0.00    83.45    328     11
318      0      0     0.0   185.8
0x043000e0  /dev/sdbk  (HW path: 1:0:4:52)   (mpath device: /dev/mapper/TW1_data)
  0x043000e0  r   0.50    0.73     17 278117 16311     0.00    78.17    341      9
334      0      0     0.0   178.2
  0x043000e0  w   0.50    0.50      0     39   193     0.00     1.51      4      0
4      0      0     0.0     2.7
  0x043000e0  t   0.50    0.73     17 278156 16125     0.00    77.28    345      9
338      0      0     0.0   178.2
0x04400040  /dev/sdbq  (HW path: 1:0:5:52)   (mpath device: /dev/mapper/TW1_data)
```

```
  0x04400040  r   0.50    0.76     16 262962 16384     0.00    85.18    321     14
308      0      0     0.0   190.6
  0x04400040  w   0.00    0.00      0      0     0     0.00     0.00      0      0
0      0      0     0.0     0.0
  0x04400040  t   0.50    0.76     16 262962 16384     0.00    85.18    321     14
308      0      0     0.0   190.6

Mapper Device Statistics

      device rw  avque avinflt   io/s   KB/s  avsz   avwait   avserv    tot    seq
rnd  reque  flush maxwait maxserv
0x0fe00002   /dev/mapper/TW1_data  ->  /dev/dm-2  (HW path: unknown)
  0x0fe00002  r   0.55   15.22    195 3192407 16354     0.00    81.11   3904     17
3902      0      0     0.0   241.8
  0x0fe00002  w   0.50   15.25      2    209   130     0.00     1.11     32      0
32      0      0     0.0     3.5
  0x0fe00002  t   0.55   15.22    197 3192616 16222     0.00    80.46   3936     17
3934      0      0     0.0   241.8

Multipath FC HBA Statistics

      HBA    rw  avque avinflt   io/s   KB/s  avsz   avwait   avserv    tot    seq
rnd  reque  flush maxwait maxserv
0:0:0
       0:0:0  r   0.00    0.80     16 269567 16337     0.00    79.72    330     14
317      0      0     0.0   131.3
       0:0:0  w   0.00    0.14      0     15    44     0.00     0.70      7      0
7      0      0     0.0     1.1
       0:0:0  t   0.00    0.78     17 269582 15999     0.00    78.08    337     14
324      0      0     0.0   131.3
0:0:1
       0:0:1  r   0.00    0.80     16 268287 16309     0.00    77.70    329     10
320      0      0     0.0   126.7
       0:0:1  w   0.00    0.33      0     18    40     0.00     0.89      9      0
9      0      0     0.0     1.6
       0:0:1  t   0.00    0.79     17 268305 15876     0.00    75.66    338     10
329      0      0     0.0   126.7
0:0:2
       0:0:2  r   0.00    0.75     16 266187 16380     0.00    75.65    325     12
314      0      0     0.0   125.0
       0:0:2  w   0.00    0.25      0     20    49     0.00     0.71      8      0
8      0      0     0.0     1.1
       0:0:2  t   0.00    0.74     17 266207 15988     0.00    73.85    333     12
322      0      0     0.0   125.0
0:0:3
       0:0:3  r   0.00    0.76     16 269208 16365     0.00    78.66    329     13
317      0      0     0.0   122.9
       0:0:3  w   0.00    0.30      0     17    33     0.00     0.74     10      0
10      0      0     0.0     1.3
       0:0:3  t   0.00    0.75     17 269225 15883     0.00    76.36    339     13
327      0      0     0.0   122.9
0:0:4
       0:0:4  r   0.00    0.77     16 266904 16324     0.00    80.41    327     12
316      0      0     0.0   127.4
       0:0:4  w   0.00    0.33      0     15    33     0.00     0.62      9      0
9      0      0     0.0     1.0
       0:0:4  t   0.00    0.76     17 266919 15888     0.00    78.28    336     12
325      0      0     0.0   127.4
0:0:5
       0:0:5  r   0.00    0.75     16 265880 16361     0.00    78.95    325     14
312      0      0     0.0   126.9
```

```
       0:0:5  w   0.00    0.29      0     17    47     0.00     0.72      7      0
7      0      0     0.0     1.3
       0:0:5  t   0.00    0.74     17 265897 16017     0.00    77.30    332     14
319      0      0     0.0   126.9
1:0:0
       1:0:0  r   0.00    0.76     16 263781 16384     0.00    83.78    322     10
314      0      0     0.0   178.3
       1:0:0  w   0.00    0.25      0     16    40     0.00     0.57      8      0
8      0      0     0.0     1.0
       1:0:0  t   0.00    0.75     16 263797 15987     0.00    81.76    330     10
322      0      0     0.0   178.3
1:0:1
       1:0:1  r   0.00    0.79     16 257228 16384     0.00    84.83    314      6
310      0      0     0.0   241.7
       1:0:1  w   0.00    0.14      0      3     9     0.00     1.16      7      0
7      0      0     0.0     3.4
       1:0:1  t   0.00    0.77     16 257231 16026     0.00    83.01    321      6
317      0      0     0.0   241.7
1:0:2
       1:0:2  r   0.00    0.73     16 260504 16384     0.00    85.29    318      7
312      0      0     0.0   190.0
       1:0:2  w   0.00    0.33      0     42    93     0.00     0.63      9      0
9      0      0     0.0     1.1
       1:0:2  t   0.00    0.72     16 260546 15935     0.00    82.96    327      7
321      0      0     0.0   190.0
1:0:3
       1:0:3  r   0.00    0.73     16 263781 16333     0.00    84.72    323     11
313      0      0     0.0   185.8
       1:0:3  w   0.00    0.36      1     43    77     0.00     1.93     11      0
11      0      0     0.0    11.6
       1:0:3  t   0.00    0.72     17 263824 15797     0.00    81.99    334     11
324      0      0     0.0   185.8
1:0:4
       1:0:4  r   0.00    0.73     17 278117 16311     0.00    78.17    341      9
334      0      0     0.0   178.2
       1:0:4  w   0.00    0.20      0     42    83     0.00     0.94     10      0
10      0      0     0.0     2.7
       1:0:4  t   0.00    0.72     18 278159 15849     0.00    75.97    351      9
344      0      0     0.0   178.2
1:0:5
       1:0:5  r   0.00    0.76     16 262962 16384     0.00    85.18    321     14
308      0      0     0.0   190.6
       1:0:5  w   0.00    0.00      0      2     8     0.00     1.48      5      0
5      0      0     0.0     3.2
       1:0:5  t   0.00    0.75     16 262964 16132     0.00    83.90    326     14
313      0      0     0.0   190.6

Physical I/O Histogram
     msecs <5     <10    <20    <50    <100   <200   <300   <500   <1000  >1000
    Read:  0      0      1      163    3011   727    2      0      0      0
    Write: 99     0      1      0      0      0      0      0      0      0

Top 20 Tasks sorted by Multipath I/O

     Cnt      r/s      w/s    KB/sec    Avserv      PID  Process
================================================================================
    3906      194        1 3172121.2    80.582    25660  hdbindexserver (SubmitThread-
DA)
      39        0        2      30.4     0.780    25395  /opt/quest/sbin/.vasd
      15        1        0   12287.9    79.868    26734  hdbindexserver (PoolThread)
```

```
      12        0        1       2.4     0.593    25658  hdbindexserver (SubmitThread-
LO)
       6        0        0       2.6     0.485    23619  [kworker/u194:6]
       5        0        0       1.6     2.976     4971  [xfsaild/dm-11]
       5        0        0    4096.0    64.154    26042  hdbindexserver (JobWrk0070)
       5        0        0    4096.0    83.468    27064  hdbindexserver (JobWrk0197)
       3        0        0       1.2     0.616    25525  hdbnameserver (PoolThread)
       2        0        0       0.4     0.568    25385  hdbnameserver (PoolThread)
       1        0        0       0.8     0.652     8926  [kworker/6:2H]
       1        0        0       0.4     1.268     4963  [xfsaild/dm-10]
       1        0        0      12.8     1.155    25639  hdbxsengine (SubmitThread-DA)
       1        0        0       0.8     0.355     3399  [kworker/30:1H]
       1        0        0       0.8     0.624    69034  [kworker/31:1H]
       1        0        0       0.8     0.694    28424  [kworker/17:1H]

Top 20 Tasks sorted by physical I/O

     Cnt      r/s      w/s    KB/sec    Avserv      PID  Process
================================================================================
    3936      195        2 3192615.9    80.393     1493  [kdmwork-254:2]
      39        0        2      30.4     0.766     1404  [kdmwork-254:0]
      14        0        1       3.6     0.626     1658  [kdmwork-254:5]
      11        0        1       3.6     1.633     1890  [kdmwork-254:6]
       4        0        0       2.6     0.389     1952  [kdmwork-254:7]
```

Description of fields:

```
rw          values are read (r), write(w), reads+writes(t).
avque       Average number of requests queued to the block device
avinflt     Average number of requests that have been issued from the block device and
            are "inflight" to the lower layers
io/s        IOs per second
Kb/s        Kilobytes per second
avsz        Average size of the IOs
avwait      Average time in milliseconds a request spent on the block device queue
            before it is issued to the lower layers
avserv      Average time in milliseconds after a block request has been issued to the
            lower layers until it is completed by the block device.
ios         Total IOs during kidsk pass
seq         Sequential IOs as seen by the disk. Can be misleading if path switching
            software is used (multipath)
rnd         Random IOs
            NOTE: seq + rnd = ios
reque       Number of IOs that had to be requeued
flush       Number of Barrier writes to the block device
maxwait     Max time in msecs that an IO spent on the device block queue
maxserv     Max time in msecs that an IO request took to complete
```

11.11 CPU/RUNQ Analysis report (kiinfo -kirunq)

kiinfo -kirunq produces a **CPU/RUNQ Analysis Report** based on the switch and wakeup tracepoints. It also reports the top 10 tasks (default) by runq wait time, and optionally further reports the task level detail. The CPU/RUNQ report will the Global CPU statistics, per-Node statistics, Hyperthread CPU-pair statistics, System-wide Runq statistics (per-CPU and per-Node), as well as Top PIDs (tasks) on the Runq, as well as taking CPU. The CPU/RUNQ Report will also include both soft and hard IRQ statistics.

Usage:

```
kiinfo -kirunq [flag,flag,...] [options ...]
```

Options:

```
See kiinfo options
```

Flags:

```
help
npid=<ntid>                # Report Top PIDs using CPU, System CPU, and waiting on
                             the CPU RunQ
rqpid=<rqtid>              # Pid to monitor for runq delay detailed reporting
                             (requires rqdetail be set)
rqwait=<usecs>             # Runq wait in usecs that triggers delay warning one-liner
msr                        # Collect advanced CPU statistics such as LLC Hit%,
                             CPI, average CPU frequency, and SMI count
kitrace                    # Include formatted ASCII trace records prior to the
                             standard report
abstime                    # Print absolute time (seconds since boot) for each record
                             when using kitrace flag
fmttime                    # Print formatted time for each record when using kitrace
                             flag (i.e.  Wed Feb  5 16:40:15.529100)
epochtime                  # Print time in seconds since the epoch (Jan 1, 1970)
edus=<filename>            # Specify output of "db2pd -edus" to get DB2 thread names
jstack=<filename>          # Jstack output file to use (default jstack.<timestamp>)
events=[                   # Trace events to be traced
       default |           # Only trace default events
       all |               # trace all valid ftrace events
       <kitool> |          # Only the events needed for a specific subtool is used.
       <event> ]             # Trace a specific event
subsys=<subsys>            # Enable tracing only for certain subsystems.  Valid
                             subsystems include: power, scsi, block, sched, syscalls,
                             irq.
csv                        # Create CSV report kirunq.*.csv
```

Examples:

- Reading from a kitrace binary file:

  ```
  $ kiinfo -kirunq -ts 1211_0900
  ```

- Running on a live system, listing the top 10 threads spending time on the RunQ, as well reporting on the IRQ times and events

  ```
  $ kiinfo -kirunq npid=10,subsys=sched,subsys=irq -a 10 -p 1
  ```

- One-liner warning of runq delays for any task using rqwait flag.

  Example using flags rqwait=50000 (Seeking one-liner warnings for any task with a delay above 50ms)

  ```
  Command line: kiinfo -kirunq rqwait=50000 -ts 0712_0838
  ```

```
kiinfo (2.0)

Linux aps41-120 2.6.32-131.21.1.el6.x86_64 #1 SMP Fri Nov 11 11:50:54 EST 2011
x86_64 x86_64 x86_64 GNU/Linux

    1.369216 cpu=20 pid=18718 sched_switch RUNQ Delay!  112.102ms next_pid=83
prev_state=PREEMPTED prio=0
    3.369241 cpu=20 pid=83 sched_switch RUNQ Delay!  65.210ms next_pid=85
prev_state=SLEEPING prio=120
    5.955957 cpu=0 pid=152087 sched_switch RUNQ Delay!  1051.963ms next_pid=4
prev_state=SLEEPING prio=120
    7.972499 cpu=0 pid=152087 sched_switch RUNQ Delay!  96.072ms next_pid=4
prev_state=SLEEPING prio=120
   16.000576 cpu=0 pid=152087 sched_switch RUNQ Delay!  51.215ms next_pid=4
prev_state=SLEEPING prio=120
   18.016705 cpu=0 pid=152087 sched_switch RUNQ Delay! 421.002ms next_pid=4
prev_state=SLEEPING prio=120
```

- Hyperthreaded CPU reporting

  The hyperthreaded CPU reporting has two distinct parts. First, there is a CPU utilization report for each physical CPUs. The report shows the amount of time (and percentage of time) that both logical CPUs (LCPU) of a physical CPU (PCPU) were idle, busy, or one LCPU was idle while the other was busy. Below is an example:

```
Hyper-threading CPU pair status

     PCPU     double idle   lcpu1 busy   lcpu2 busy  double busy
  [  0  40]:     0.252331     9.537897     3.982534     6.180018
  [  1  41]:     0.041383     7.284849     9.471828     3.129207
  [  2  42]:     0.037760     7.870877     8.058199     3.960363
  [  3  43]:     0.022695     7.298441     5.639476     6.979489

     PCPU     double idle   lcpu1 busy   lcpu2 busy  double busy
  [  0  40]:         1.3%        47.8%        20.0%        31.0%
  [  1  41]:         0.2%        36.6%        47.5%        15.7%
  [  2  42]:         0.2%        39.5%        40.4%        19.9%
  [  3  43]:         0.1%        36.6%        28.3%        35.0%
```

  The above report gives a better picture of how much idle time (and thus capacity) is truly on the system.

- Double-Busy Double-Idle histograms.

  The next section provides two histograms showing how often a PCPU is double idle when there is one or more PCPUs that are double busy. The histograms reflects the greatest amount of time a PCPU was double-idle when the designated PCPU went double-busy. There is a system-wide histogram where any PCPU on the system could be double-idle, and a locality-wide histogram where only PCPUs on the same LDOM as the designated PCPU are analyzed. The reports are similar to the following:

```
System-Wide Double-Busy Double-Idle CPU Time Histogram.
Idle time in Usecs
```

```
     PCPU        <10    <20    <50    <100   <250   <500   <750  <1000  <1250
<1500  <2000  <3000  <5000 <10000 <20000 >20000
  [  0  40]:     544       3       8       9      17       8       3       2       3
3       0       6       2       2       2       0
  [  1  41]:     250       6       4       6       9       2       2       2       4
2       2       2       0       0       1       0
  [  2  42]:     215       4       4       6      12       5       1       0       0
1       2       2       4       1       4       2
  [  3  43]:     421       6       5      16      21       4       0       2       1
5       3       0       2       0       3       0

Locality-Wide Double-Busy Double-Idle CPU Time Histogram.
Idle time in Usecs
     PCPU  NODE     <10    <20    <50   <100   <250   <500   <750  <1000  <1250
<1500  <2000  <3000  <5000 <10000 <20000 >20000
  [  0  40][ 0]:    581       4       2       6      13       4       1       0       0
1       0       0       0       0       0       0
  [  1  41][ 0]:    277       3       3       2       6       0       0       1       0
0       0       0       0       0       0       0
  [  2  42][ 0]:    238       0       4       5       6       3       1       0       0
0       1       1       3       1       0       0
  [  3  43][ 0]:    460       3       2       6      13       1       0       2       0
0       2       0       0       0       0       0
```

11.12 Wait Event Analysis Report (kiinfo -kiwait)

kiinfo -kiwait generates the Wait Event Report based on the switch tracepoints when using the *LiKI* tracing mechanism. The Wait Event Report will identify the top kernel functions where tasks go to sleep and their corresponding stack traces. It also prints the top tasks calling switch to go to sleep and statistics about each function where the task goes to sleep.

kiwait will report a section on the following:

********** GLOBAL SCHEDULER ACTIVITY REPORT ********

Reports global information such as CPU migrations, Voluntary and Forced Context Switches, etc.

******** GLOBAL SWITCH REPORT ********

Displays top kernel functions calling sleep and top kernel stack traces leading up to the switch events

********** PER-PID SCHEDULER ACTIVITY REPORTS ********

Reports Top PIDs calling switch() and their Scheduling Activity

Usage:

```
kiinfo -kiwait [flag,flag,...] [options ...]
```

Options:

```
See kiinfo options
```

Flags:

```
help
npid=<npid>       # List Top Tasks calling SWTCH frequently
nsym=<nsym>       # List Top <nsym> kernel functions calling SWTCH
kitrace           # Include formatted ASCII trace records prior to the standard report
abstime           # Print absolute time (seconds since boot) for each record when
                    using kitrace flag
fmttime           # Print formatted time for each record when using kitrace flag
                    (i.e.  Wed Feb  5 16:40:15.529100)
epochtime         # Print time in seconds since the epoch (Jan 1, 1970)

edus=<filename>   # Specify output of "db2pd -edus" to get DB2 thread names
jstack=<filename> # Jstack output file to use (default jstack.<timestamp>)
csv               # Create CSV report kiwait.*.csv
```

Examples:

- Create a Wait Event Report for the top 10 kernel functions and display top 5 tasks calling switch() from a LinuxKI dump:

  ```
  $ kiinfo -kiwait npid=5,nsym=10 -ts 1215_0744
  ```

- Create a Wait Event Report on a live system, creating a report every 5 seconds for 4 passes:

  ```
  $ kiinfo -kiwait npid=10,nsym=20 -a 5 -p 4
  ```

Sample Output:

```
Command line: /opt/linuxki/kiinfo -kiwait nsym=10,npid=5,csv -ts 0414_1119

kiinfo (4.3)

Linux gwr-repo1.rose.rdlabs.hpecorp.net 2.6.32-504.el6.x86_64 #1 SMP Tue Sep 16
01:56:35 EDT 2014 x86_64 x86_64 x86_64 GNU/Linux


********* GLOBAL SCHEDULER ACTIVITY REPORT ********
    CPUs:       16  Seconds :    20.00  Migrations:        0
Switches:   218285  Forced  :     3446   Voluntary:   214839
 TotTime: 320.0026  RunTime :   4.3445    CPU Util:    1.36%

********* GLOBAL SWITCH REPORT ********

Kernel Functions calling sleep() - Top 10 Functions
   Count     Pct    SlpTime    Slp%   Msec/Slp   MaxMsecs  Func
   81609  37.99%    95.7483   3.48%      1.173    100.477 __mutex_lock_slowpath
   65023  30.27%    21.1248   0.77%      0.325     95.061 io_schedule
   63881  29.73%   630.8682  22.95%      9.876   2162.916 worker_thread
    2663   1.24%   292.1936  10.63%    109.723  10010.000 poll_schedule_timeout
     966   0.45%   213.8960   7.78%    221.424   8840.067 cpu_buf_read
     167   0.08%   127.7838   4.65%    765.172  10279.996 ksoftirqd
      80   0.04%   143.2621   5.21%   1790.776  20000.137 do_nanosleep
```

```
      65   0.03%    16.3917   0.60%    252.181   6999.403 kjournald2
      63   0.03%     0.0468   0.00%      0.743      8.191 jbd2_log_wait_commit
      46   0.02%    62.4485   2.27%   1357.577   9985.955 futex_wait_queue_me

Globals switch stack traces (sort by count):
   count   wpct       avg   Stack trace
              %     msecs
============================================================
   63881  22.95     9.876   worker_thread  kthread  child_rip
   52267   0.67     0.352   io_schedule  __blockdev_direct_IO_newtrunc
__blockdev_direct_IO  ext4_ind_direct_IO  ext4_direct_IO  generic_file_direct_write
__generic_file_aio_write  generic_file_aio_write  ext4_file_write  do_sync_write
vfs_write  sys_write  tracesys
   34961   1.51     1.184   __mutex_lock_slowpath  mutex_lock
generic_file_llseek_size  ext4_llseek  vfs_llseek  sys_lseek  tracesys
   24186   0.96     1.096   __mutex_lock_slowpath  mutex_lock  generic_file_aio_write
ext4_file_write  do_sync_write  vfs_write  sys_write  tracesys
   22451   1.01     1.241   __mutex_lock_slowpath  mutex_lock
__blockdev_direct_IO_newtrunc  __blockdev_direct_IO  ext4_ind_direct_IO
ext4_direct_IO  generic_file_aio_read  do_sync_read  vfs_read  sys_read  tracesys
   12534   0.10     0.214   io_schedule  __blockdev_direct_IO_newtrunc
__blockdev_direct_IO  ext4_ind_direct_IO  ext4_direct_IO  generic_file_aio_read
do_sync_read  vfs_read  sys_read  tracesys
    1440   4.12    78.691   poll_schedule_timeout  do_select  core_sys_select
sys_select  tracesys
    1223   6.51   146.262   poll_schedule_timeout  do_sys_poll  sys_poll  tracesys
     966   7.78   221.424   cpu_buf_read  vfs_read  sys_read  tracesys
     167   4.65   765.172   ksoftirqd  kthread  child_rip
     152   0.00     0.244   io_schedule  sync_buffer  __wait_on_bit
out_of_line_wait_on_bit  __wait_on_buffer  jbd2_journal_commit_transaction  kjournald2
kthread  child_rip
      80   5.21  1790.776   do_nanosleep  hrtimer_nanosleep  sys_nanosleep  tracesys

********* PER-PID SCHEDULER ACTIVITY REPORTS ********
-----------------------------------------------------
PID 202  [kblockd/0]
RunTime    :  0.044596  SysTime   :  0.044449   UserTime   :  0.000000
SleepTime  : 19.856610  Sleep Cnt :     33302   Wakeup Cnt :         0
RunQTime   :  0.098759  PreemptCnt:         0   Switch Cnt :     33302
Last CPU   :         0  CPU Migrs :         0   NODE Migrs :         0
schedpolicy: SCHED_NORMAL
Kernel Functions calling sleep() - Top 50 Functions
   Count     Pct    SlpTime    Slp% TotalTime%   Msec/Slp   MaxMsecs  Func
   33301 100.00%    19.8564 100.00%     99.28%      0.596     94.478  worker_thread
-----------------------------------------------------
PID 210  [kblockd/8]
RunTime    :  0.039336  SysTime   :  0.039118   UserTime   :  0.000000
SleepTime  : 19.893738  Sleep Cnt :     30075   Wakeup Cnt :         0
RunQTime   :  0.066857  PreemptCnt:         0   Switch Cnt :     30075
Last CPU   :         8  CPU Migrs :         0   NODE Migrs :         0
schedpolicy: SCHED_NORMAL
Kernel Functions calling sleep() - Top 50 Functions
   Count     Pct    SlpTime    Slp% TotalTime%   Msec/Slp   MaxMsecs  Func
   30074 100.00%    19.8937 100.00%     99.47%      0.661   2162.916  worker_thread
-----------------------------------------------------
PID 31550  /home/mcr/bin/iotest8
RunTime    :  0.531838  SysTime   :  0.511715   UserTime   :  0.019013
SleepTime  : 19.250368  Sleep Cnt :     29493   Wakeup Cnt :     19662
RunQTime   :  0.217716  PreemptCnt:      1093   Switch Cnt :     30586
Last CPU   :         8  CPU Migrs :      5556   NODE Migrs :        35
schedpolicy: SCHED_NORMAL
```

```
Kernel Functions calling sleep() - Top 50 Functions
   Count     Pct    SlpTime    Slp% TotalTime%   Msec/Slp   MaxMsecs  Func
   17511  59.37%     7.9291  41.19%     39.65%      0.453     87.119  io_schedule
   11981  40.62%    11.3206  58.81%     56.60%      0.945     98.893
__mutex_lock_slowpath
                   :                                    :
```

11.13 File Activity Report (kiinfo -kifile)

kiinfo -kifile generates the File Activity Report. The kifile option will use the System Call records as well as the switch/wakeup to identify file activity times. The kifile report will show the Global File Activity Report followed by additional details on the system calls made to the top accessed files.

Usage:

```
kiinfo -kifile [flag,flag,...]
```

Options:

```
See kiinfo options
```

Flags:

```
help
nfile=<nfile>         # Limit reports sections to Top <nfile> files
kitrace               # Include formatted ASCII trace records prior to the standard
                        report
nosysenter            # Do not print syscall entry records when using kitrace flag
abstime               # Print absolute time (seconds since boot) for each record when
                        using kitrace flag
fmttime               # Print formatted time for each record when using kitrace flag
                        (i.e.  Wed Feb  5 16:40:15.529100)
epochtime             # Print time in seconds since the epoch (Jan 1, 1970)
sysignore=<filename> # Do not trace system calls listed in the <ignore_file>
                        This can reduce trace data by eliminating frequently
                        called system calls, such as getrusage(),gettimeofday(),
                        time(), etc...
edus=<filename>       # Specify output of "db2pd -edus" to get DB2 thread names
jstack=<filename>     # Jstack output file to use (default jstack.<timestamp>)
csv                   # Create CSV report kifile.*.csv
```

Examples:

- Create a File Activity Report and report on top 4 files:

  ```
  $ kiinfo -kifile nfile=4,csv -ts 0816_0839
  ```

- Create a File Activity Report for the top 5 active online files over a 15 second period

  ```
  $ kiinfo -kifile nfile=5 l -a 15 -p 1
  ```

Sample Output:

```
Command line: kiinfo -kifile nfile=5 -ts 0414_1119

kiinfo (7.1)

Linux gwr-repo1.rose.rdlabs.hpecorp.net 2.6.32-504.el6.x86_64 #1 SMP Tue Sep 16
01:56:35 EDT 2014 x86_64 x86_64 x86_64 GNU/Linux

KI Binary Version 7
Thu Apr 14 11:19:44 2016

******** GLOBAL FILE ACTIVITY REPORT ********

---  Top Files sorted by System Call Count  ---
Syscalls   ElpTime  Lseeks   Reads  Writes    Errs         dev/fdatap       node
type  Filename
  129604  119.8623   64803   12534   52267       0 0x000000000fd0000b  131334175
REG  /work/mcr/bigfile
    3988    0.0078       0       0     997    2392 0xffff88101e507400      33397
unix  socket
    2220    0.0053       0       0     599     802 0xffff88101e507740      33398
unix  socket
      73    0.0472      12       0      53       0 0x000000000fd00007     262970
REG  /var/opt/perf/datafiles/logdev
      24    0.0001       0      16       8       0 0xffff88201b789480      66603
unix  /var/run/libvirt/libvirt-sock-ro

---  Top Files sorted by System Call Count (Detailed)  ---

device: 0x0fd0000b node: 131334175 fstype:      REG  filename:  /work/mcr/bigfile
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
write                52267   2613.3   46.725737   0.000894   0.101502       0    8192
20906.6
   SLEEP             76456   3822.8   44.887961   0.000587
      Sleep Func     24186            26.511726   0.001096   0.098893
__mutex_lock_slowpath
      Sleep Func     52267            18.376235   0.000352   0.087119  io_schedule
   RUNQ                                 0.523166
   CPU                                  1.131697
lseek                64803   3240.1   41.905358   0.000647   0.116171       0
   SLEEP             34962   1748.1   41.382893   0.001184
      Sleep Func     34961            41.382893   0.001184   0.100477
__mutex_lock_slowpath
   RUNQ                                 0.197993
   CPU                                  0.183160
read                 12534    626.7   31.231226   0.002492   0.100681       0    8192
5013.6
   SLEEP             34987   1749.3   30.539475   0.000873
      Sleep Func     22451            27.851434   0.001241   0.099161
__mutex_lock_slowpath
      Sleep Func     12534             2.688042   0.000214   0.095061  io_schedule
   RUNQ                                 0.200071
   CPU                                  0.446576

device: 0xffff88101e507400 node:   33397 fstype:     unix  filename:  socket
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
writev                 997     49.8    0.004971   0.000005   0.000173       0      25
1.2
   RUNQ                                 0.001406
```

```
   CPU                                  0.000190
recvfrom              2991    149.5    0.002870   0.000001   0.000012    2392       6
0.9
   RUNQ                                 0.000044
   CPU                                  0.000050

device: 0xffff88101e507740 node:   33398 fstype:     unix  filename:  socket
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
recvmsg               1621     81.0    0.003159   0.000002   0.000029     802      15
1.2
   RUNQ                                 0.000009
   CPU                                  0.000018
writev                 599     29.9    0.002100   0.000004   0.000021       0      32
0.9

device: 0x0fd00007 node:  262970 fstype:      REG  filename:
/var/opt/perf/datafiles/logdev
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
write                  53      2.6    0.047199   0.000891   0.001232       0     112
0.3
   SLEEP              107      5.3    0.044917   0.000420
      Sleep Func       53             0.032507   0.000613   0.000891
jbd2_log_wait_commit
      Sleep Func       54             0.012411   0.000230   0.000268  io_schedule
   RUNQ                                 0.000611
   CPU                                  0.001521
fcntl                   8      0.4    0.000008   0.000001   0.000001       0
lseek                  12      0.6    0.000003   0.000000   0.000001       0

device: 0xffff88201b789480 node:   66603 fstype:     unix  filename:
/var/run/libvirt/libvirt-sock-ro
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
write                   8      0.4    0.000039   0.000005   0.000010       0      28
0.0
read                   16      0.8    0.000024   0.000001   0.000003       0      14
0.0

---  Top Files sorted by Errors  ---
Syscalls   ElpTime  Lseeks   Reads  Writes    Errs         dev/fdatap       node
type  Filename
    3988    0.0078       0       0     997    2392 0xffff88101e507400      33397
unix  socket
    2220    0.0053       0       0     599     802 0xffff88101e507740      33398
unix  socket
      19    0.0000       0       0       5      11 0xffff882018c2ca80      35406
unix  socket
      20    0.0000       0       0       4      11 0xffff881000316800      68971
unix  socket
      13    0.0000       0       0       2       9 0xffff881026813480      33277
unix  socket

---  Top Files sorted by Elapsed Time  ---
Syscalls   ElpTime  Lseeks   Reads  Writes    Errs         dev/fdatap       node
type  Filename
  129604  119.8623   64803   12534   52267       0 0x000000000fd0000b  131334175
REG  /work/mcr/bigfile
       7   20.1588       0       2       5       0 0x0000000000a000e5      15611
CHR  /dev/fuse
```

```
       2    4.9912       0       2       0       0 0x000000000000000a          1
DIR  inotify
      73    0.0472      12       0      53       0 0x000000000fd00007     262970
REG  /var/opt/perf/datafiles/logdev
    3988    0.0078       0       0     997    2392 0xffff88101e507400      33397
unix  socket
```

Note that the Sleep Functions (in red) are only available if the *LiKI* tracing mechanism is used.

11.14 Network Socket Activity Report (kinfo -kisock)

kiinfo -kisock generates the **Network Socket Activity Report**. The kisock option will use the system call events as well as the switch/wakeup events identify network socket activity times. The report will show the top network data flows based on the IP address, the top Local and remote IP and IP:Port activity, and the most active sockets on the system. Note that this report is based on system call activity on the network sockets. If there is network communication outside of a system call (perhaps from a system daemon), the activity will not be traced and reported.

Usage:

```
kiinfo -kisock [flag,flag,...]
```

Options:

```
See kiinfo options
```

Flags:

```
help
nsock=<nsock>     # Limit reports sections to Top <nsock> sockets
kitrace           # Include formatted ASCII trace records prior to the standard report
nosysenter        # Do not print syscall entry records when using kitrace flag
abstime           # Print absolute time (seconds since boot) for each record when using
                    kitrace flag
fmttime           # Print formatted time for each record when using kitrace flag
                    (i.e.  Wed Feb  5 16:40:15.529100)
epochtime         # Print time in seconds since the epoch (Jan 1, 1970)

sysignore=<filename>  # Do not trace system calls listed in the <ignore_file>
                        This can reduce trace data by eliminating frequently
                        called system calls, such as getrusage(),gettimeofday(),
                        time(), etc...
csv               # Create CSV report kifile.*.csv
```

Examples:

- Create a File Report and reporting on top 5 sockets:

```
$ kiinfo -kisock nsock=5 -ts 0204_0643
```

```
Command line: kiinfo -kisock nsock=5 -ts 0204_0643

kiinfo (7.1)
```

```
******** NETWORK SOCKET ACTIVITY REPORT ********

Top IP->IP dataflows sorted by System Call Count
================================================
Syscalls      Rd/s      RdKb/s      Wr/s      WrKb/s  Connection
   40457    1011.4         9.8    1011.4      1011.4  L=15.3.104.152 R=15.43.209.141
   35897     897.4         8.7     897.4       897.4  L=15.3.104.152 R=15.43.213.42
      67       2.1         0.0       1.1         0.0  L=127.0.0.1 R=127.0.0.1
       1       0.0         0.0       0.0         0.0  L=15.3.104.152 R=16.98.16.170

Local IP Statistics
===================================================
L=15.3.104.152   Syscalls: 76355
System Call Name  Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
recvfrom             38176   1908.8   24.561136   0.000643   0.007056       0      10
18.6
sendto               38178   1908.9    0.379765   0.000010   0.000531       0    1024
1908.9
write                    1      0.0    0.000011   0.000011   0.000011       0      84
0.0

L=127.0.0.1   Syscalls: 67
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
sendmsg                 20      1.0    0.000251   0.000013   0.000022       0      41
0.0
read                    40      2.0    0.000131   0.000003   0.000024       0      20
0.0
write                    2      0.1    0.000110   0.000055   0.000100       0     242
0.0
recvmsg                  3      0.1    0.000024   0.000008   0.000016       1     161
0.0
getpeername              2      0.1    0.000002   0.000001   0.000001       0

Local IP:Port Statistics
===================================================
L=15.3.104.152:5506   Syscalls: 76354
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
recvfrom             38176   1908.8   24.561136   0.000643   0.007056       0      10
18.6
sendto               38178   1908.9    0.379765   0.000010   0.000531       0    1024
1908.9

L=127.0.0.1:26512   Syscalls: 42
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
read                    40      2.0    0.000131   0.000003   0.000024       0      20
0.0
write                    2      0.1    0.000110   0.000055   0.000100       0     242
0.0

L=127.0.0.1:23331   Syscalls: 25
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
sendmsg                 20      1.0    0.000251   0.000013   0.000022       0      41
0.0
```

```
recvmsg                   3      0.1    0.000024   0.000008   0.000016       1     161
0.0
getpeername               2      0.1    0.000002   0.000001   0.000001       0

L=15.3.104.152:5632   Syscalls: 1
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
write                     1      0.0    0.000011   0.000011   0.000011       0      84
0.0

Remote IP Statistics
================================================
R=15.43.209.141   Syscalls: 40457
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
recvfrom              20228   1011.4   12.272651   0.000607   0.001607       0      10
9.9
sendto                20229   1011.4    0.199215   0.000010   0.000531       0    1024
1011.4

R=15.43.213.42   Syscalls: 35897
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
recvfrom              17948    897.4   12.288485   0.000685   0.007056       0      10
8.8
sendto                17949    897.4    0.180550   0.000010   0.000495       0    1024
897.4

R=127.0.0.1   Syscalls: 67
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
sendmsg                  20      1.0    0.000251   0.000013   0.000022       0      41
0.0
read                     40      2.0    0.000131   0.000003   0.000024       0      20
0.0
write                     2      0.1    0.000110   0.000055   0.000100       0     242
0.0
recvmsg                   3      0.1    0.000024   0.000008   0.000016       1     161
0.0
getpeername               2      0.1    0.000002   0.000001   0.000001       0

R=16.98.16.170   Syscalls: 1
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
write                     1      0.0    0.000011   0.000011   0.000011       0      84
0.0

Remote IP:Port Statistics
================================================
R=15.43.209.141:9120   Syscalls: 40457
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
recvfrom              20228   1011.4   12.272651   0.000607   0.001607       0      10
9.9
sendto                20229   1011.4    0.199215   0.000010   0.000531       0    1024
1011.4

R=15.43.213.42:47523   Syscalls: 35897
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
```

```
recvfrom             17948    897.4   12.288485   0.000685   0.007056       0      10
8.8
sendto               17949    897.4    0.180550   0.000010   0.000495       0    1024
897.4

R=127.0.0.1:23331   Syscalls: 42
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
read                    40      2.0    0.000131   0.000003   0.000024       0      20
0.0
write                    2      0.1    0.000110   0.000055   0.000100       0     242
0.0

R=127.0.0.1:26512   Syscalls: 25
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
sendmsg                 20      1.0    0.000251   0.000013   0.000022       0      41
0.0
recvmsg                  3      0.1    0.000024   0.000008   0.000016       1     161
0.0
getpeername              2      0.1    0.000002   0.000001   0.000001       0

R=16.98.16.170:12228   Syscalls: 1
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
write                    1      0.0    0.000011   0.000011   0.000011       0      84
0.0

Top Sockets sorted by System Call Count
Syscalls      Rd/s      RdKb/s      Wr/s      WrKb/s  Connection
================================================
   40457    1011.4         9.8    1011.4      1011.4  L=15.3.104.152:5506
R=15.43.209.141:9120
   35897     897.4         8.7     897.4       897.4  L=15.3.104.152:5506
R=15.43.213.42:47523
      42       2.0         0.0       0.1         0.0  L=127.0.0.1:26512
R=127.0.0.1:23331
      25       0.1         0.0       1.0         0.0  L=127.0.0.1:23331
R=127.0.0.1:26512
       1       0.0         0.0       0.0         0.0  L=15.3.104.152:5632
R=16.98.16.170:12228

Top Sockets sorted by System Call Count (Detailed)
================================================
L=15.3.104.152:5506 -> R=15.43.209.141:9120   Syscalls: 40457
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
recvfrom             20228   1011.4   12.272651   0.000607   0.001607       0      10
9.9
   SLEEP             20225   1011.2   11.621089   0.000575
      Sleep Func     20224            11.621089   0.000575   0.001579  sk_wait_data
   RUNQ                               0.108565
   CPU                                0.323236
sendto               20229   1011.4    0.199215   0.000010   0.000531       0    1024
1011.4
   RUNQ                               0.002471
   CPU                                0.000016

L=15.3.104.152:5506 -> R=15.43.213.42:47523   Syscalls: 35897
System Call Name     Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
```

```
recvfrom              17948    897.4   12.288485   0.000685   0.007056       0      10
8.8
   SLEEP              17949    897.4   11.686403   0.000651
      Sleep Func      17948            11.686403   0.000651   0.007006  sk_wait_data
   RUNQ                                 0.098323
   CPU                                  0.293920
sendto                17949    897.4    0.180550   0.000010   0.000495       0    1024
897.4

L=127.0.0.1:26512 -> R=127.0.0.1:23331   Syscalls: 42
System Call Name      Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
read                     40      2.0    0.000131   0.000003   0.000024       0      20
0.0
write                     2      0.1    0.000110   0.000055   0.000100       0     242
0.0

L=127.0.0.1:23331 -> R=127.0.0.1:26512   Syscalls: 25
System Call Name      Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
sendmsg                  20      1.0    0.000251   0.000013   0.000022       0      41
0.0
recvmsg                   3      0.1    0.000024   0.000008   0.000016       1     161
0.0
getpeername               2      0.1    0.000002   0.000001   0.000001       0

L=15.3.104.152:5632 -> R=16.98.16.170:12228   Syscalls: 1
System Call Name      Count     Rate     ElpTime        Avg        Max    Errs    AvSz
KB/s
write                     1      0.0    0.000011   0.000011   0.000011       0      84
0.0
```

Note that the Sleep Functions are only available if the *LiKI* tracing mechanism is used.

11.15 Futex Activity Report (kiinfo -kifutex)

kiinfo -kifutex generates the Futex Activity Report. Futex stands for Fast Userspace muTEX. The kifutex option will use the system call and sched_wakeup trace records to identify the Futex activity times. The kifutex report will show the Global Futex Report sorted by both Elapsed Time and Count.

Usage:

```
kiinfo -kifutex [flag,flag,...]
```

Options:

```
See kiinfo options
```

Flags:

```
help
uaddr=<futex addr>        # Filter on a specific futex address
nfutex=<nfutex>           # Limit output only list top nfutex futexes
npid                      # Limit per-futex details to list npids for each futex
kitrace                   # Include formatted ASCII trace records prior to the
                            standard report
nosysenter                # Do not print syscall entry records when using kitrace flag
```

```
abstime                    # Print absolute time (seconds since boot) for each record
                             when using kitrace flag
fmttime                    # Print formatted time for each record when using kitrace
                             flag (i.e.  Wed Feb  5 16:40:15.529100)
epochtime                  # Print time in seconds since the epoch (Jan 1, 1970)
sysignore=<filename>       # Do not trace system calls listed in the <ignore_file>
                             This can reduce trace data by eliminating frequently
                             called system calls, such as getrusage(),gettimeofday(),
                             time(), etc...
edus=<filename>            # Specify output of "db2pd -edus" to get DB2 thread names
jstack=<filename>          # Jstack output file to use (default jstack.<timestamp>)
```

Examples:

- Create a Futex Report and reporting on top 5 futex and listing 10 PIDs for each futex:

```
$ kiinfo -kifutex nfutex=5,npid=10 -ts 1115_1758
```

- Create a Futex Report for a specific futex address on a live system

```
$ kiinfo -kifile uaddr=0x7f0b8b6a1ca0,npid=4 -a 15 -p 1
```

Sample Output:

```
Command line: /opt/linuxki/kiinfo -kifutex nfutex=2,npid=5 -ts 1115_1758

kiinfo (7.1)

KI Binary Version 3
Linux hphanar41 3.0.93-futex-0.8-default #2 SMP Fri Nov 1 11:39:57 MDT 2013 x86_64
x86_64 x86_64 GNU/Linux

******** GLOBAL FUTEX REPORT ********
Top Futex Addrs & top PIDs by elapsed time
Total Futex count = 3922 (Top 2 listed)

Futex 0x7fbd0c94afe8     - Total PID count = 241
Operation                        Count  EAGAIN  ETIMEDOUT  AvRetVal     ElpTime
Avg         Max   Max_waker
  ALL                            42274       3      42212      0.00    4753.604
0.112447    8.158833   -1       (ICS)
  FUTEX_WAIT_PRIVATE             42253       3      42212      0.00    4753.592
0.112503    8.158833   -1       (ICS)
    PID= 28704                     176       0        175      0.00      27.850
0.158238    8.158833   -1       (ICS)
    PID= 28748                     177       0        177      0.00      19.922
0.112553    0.173100   -1       (ICS)
    PID= 28690                     178       0        178      0.00      19.921
0.111918    0.144126   -1       (ICS)
    PID= 28800                     177       0        177      0.00      19.904
0.112450    0.153787   -1       (ICS)
    PID= 28679                     176       0        176      0.00      19.901
0.113073    0.170339   -1       (ICS)
    Total PID count = 240 (Top 5 listed)
  FUTEX_WAKE_PRIVATE                21       0          0      1.81       0.012
0.000561    0.004013   0        (NA)
```

```
    PID= 25771                        21       0          0      1.81       0.012
0.000561    0.004013   0        (NA)
    Total PID count = 1 (Top 1 listed)

Futex 0x7eff09b85fe8     - Total PID count = 241
Operation                          Count  EAGAIN  ETIMEDOUT  AvRetVal     ElpTime
Avg         Max   Max_waker
  ALL                              42244       0      42222      0.00    4746.187
0.112352    4.403012   -1       (ICS)
  FUTEX_WAIT_PRIVATE               42235       0      42222      0.00    4746.182
0.112376    4.403012   -1       (ICS)
    PID= 161377                      177       0        177      0.00      19.922
0.112552    0.180507   -1       (ICS)
    PID= 161201                      176       0        176      0.00      19.910
0.113124    0.166394   -1       (ICS)
    PID= 161329                      170       0        170      0.00      19.905
0.117086    0.444780   -1       (ICS)
    PID= 161328                      178       0        178      0.00      19.900
0.111797    0.173277   -1       (ICS)
    PID= 161340                      178       0        178      0.00      19.893
0.111761    0.145822   -1       (ICS)
    Total PID count = 240 (Top 5 listed)
  FUTEX_WAKE_PRIVATE                   9       0          0      1.44       0.005
0.000599    0.001188   0        (NA)
    PID= 158536                        9       0          0      1.44       0.005
0.000599    0.001188   0        (NA)
    Total PID count = 1 (Top 1 listed)

Top Futex Addrs & top PIDs by count
Total Futex count = 3922 (Top 2 listed)

...
```

11.16 Docker Activity Report (kiinfo -kidock)

kiinfo -kidock generates the Docker Activity Report.

Usage:

```
kiinfo -kidock [flag,flag,...]
```

Options:

```
See kiinfo options
```

Flags:

```
help
npid=<npid>              # Limit per-docker container details to list npids for each
                           docker container
```

Examples:

- Create a Docker Activity Report and reporting the top 10 PIDs for each docker container:

  ```
  $ kiinfo -kidock npid=10 -ts 1115_1758
  ```

Sample Output:

```
Command line: /opt/linuxki/kiinfo -kidock npid=10 -ts 0202_0236

kiinfo (5.0)

Linux npar1.atc-hp.com 3.10.0-229.el7.x86_64 #1 SMP Thu Jan 29 18:37:38 EST 2015
x86_64 x86_64 x86_64 GNU/Linux

CPU Statistics

Container                busy          sys         user         runq
hammerdb2         1279.093769    82.100031  1180.303488    58.403992
hammerdb1          721.319254    27.197146   667.737309    30.576445
hammerdb3          417.054572    58.112565   353.545529    16.751530
hammerdb4          311.057596    55.096920   251.968391    12.914742
system             127.943019    72.027050    14.934374     9.728378

IO Statistics
                  --------------------  Total  --------------------
Container           IO/s    MB/s  AvIOsz AvInFlt   Avwait   Avserv
hammerdb2           1192     271     232    2.10     0.02     2.54
hammerdb1            624     135     222    0.46     0.01     1.44
hammerdb3            447      90     207    1.49     0.02     2.17
hammerdb4            380      65     176    1.31     0.02     2.30
system                73       1      20    4.21     0.00     0.47

-----------------------------------------------------
Container: hammerdb4

Top Tasks sorted by RunTime
PID       RunTime    SysTime   UserTime   RunqTime   SleepTime  Command
179996   1.996742   0.361413   1.610582   0.062075   17.595848  oracleORCL <hammerdb4>
180311   1.957356   0.309340   1.623354   0.128142   17.914383  oracleORCL <hammerdb4>
180321   1.949744   0.341260   1.584101   0.088157   17.955171  oracleORCL <hammerdb4>
179773   1.934043   0.320889   1.588971   0.100435   17.965523  oracleORCL <hammerdb4>
179868   1.929099   0.329468   1.575442   0.092636   17.978401  oracleORCL <hammerdb4>

Top Tasks sorted by SysTime
PID       RunTime    SysTime   UserTime   RunqTime   SleepTime  Command
179965   1.895105   0.379913   1.490881   0.070800   17.912243  oracleORCL <hammerdb4>
180047   1.893537   0.362085   1.507738   0.109118   17.987523  oracleORCL <hammerdb4>
179996   1.996742   0.361413   1.610582   0.062075   17.595848  oracleORCL <hammerdb4>
179797   1.861143   0.356734   1.481493   0.068001   18.070745  oracleORCL <hammerdb4>
180474   1.926227   1.353623   1.547788   0.043023   17.896793  oracleORCL
<hammerdb4>

Top Tasks sorted by RunQTime
PID       RunTime    SysTime   UserTime   RunqTime   SleepTime  Command
180311   1.957356   0.309340   1.623354   0.128142   17.914383  oracleORCL <hammerdb4>
179705   1.877085   0.315917   1.538102   0.114456   18.008665  oracleORCL <hammerdb4>
180453   1.669901   0.256388   1.391567   0.110270   18.100988  oracleORCL <hammerdb4>
180371   1.836491   0.321230   1.492139   0.109970   17.934713  oracleORCL <hammerdb4>
180047   1.893537   0.362085   1.507738   0.109118   17.987523  oracleORCL <hammerdb4>

Top Tasks sorted by Memory Usage
     vss      rss      PID Command
44009596    13534    83230 ora_mmon_ORCL <hammerdb4>
44008391     8542    83270 ora_cjq0_ORCL <hammerdb4>
44007314     6676   179933 oracleORCL <hammerdb4>
44007310     6672   180504 oracleORCL <hammerdb4>
```

```
44007311     6660   179747 oracleORCL <hammerdb4>

Top Tasks sorted by physical IO
     Cnt      r/s      w/s    KB/sec    Avserv      PID  Process
    1632        0       82   30806.0     4.032    83212  ora_lgwr_ORCL <hammerdb4>
      89        4        0     102.0     1.109   180353  oracleORCL <hammerdb4>
      71        4        0      92.4     0.616   180057  oracleORCL <hammerdb4>
      65        3        0      78.0     0.945   179868  oracleORCL <hammerdb4>
      61        2        1      48.8     0.356    83216  ora_ckpt_ORCL <hammerdb4>
                     :                                :
```

11.17 Kiall Report Generation (kiinfo -kiall)

The ***kiinfo -kiall*** option generates all the LinuxKI reports (kparse, kipid, kidsk, kirunq, kiprof, kiwait, kifutex, kisock, and kifile) in a single pass of the LinuxKI data. When executed from the ***kiall*** script, this option results in a performance improvement of nearly 5x compared to generating each LinuxKI report separately. Note that this option does not generate the LinuxKI ASCII file (***kiinfo -kitrace).***

Usage:

```
kiinfo -kiall [flag,flag,...] [options ...]
```

Options:

```
See kiinfo options
```

Flags:

```
help                # Help
oracle              # Includes Oracle kipid output
nofutex             # Do not collect futex statistics
mangle              # Leave C++ function names mangled
blkfrq              # Add Disk Block Frequency stats to Kparse report
edus=<filename>     # Specify output of "db2pd -edus" to get DB2 thread names
jstack=<filename>   # Jstack output file to use (default jstack.<timestamp>)
csv                 # Generates CSV outputs for kipid, kidsk, kirunq, kifile, and
                      kiwait
```

Examples:

- Running ***kiinfo -kiall*** on LinuxKI data from the ***runki*** script to generate all the LinuxKI reports and CSV files in a single pass:

  ```
  $ kiinfo -kiall oracle,csv -html -ts 1422_0623
  ```

  After executing, the following reports (and appropriate CSV files) will be generated:

  ```
  kp.0422_0623.html
  kidsk.0422_0623.txt
  kifile.0422_0623.txt
  kifutex.0422_0623.txt
  kipid.0422_0623.txt
  ```

```
kipid.oracle.0422_0623.txt
kiprof.0422_0623.txt
kirunq.0422_0623.txt
kiwait.0422_0623.txt
kisock.0422_0623.txt
kidsk.0422_0623.csv
kifile.0422_0623.csv
kipid.0422_0623.csv
kirunq.0422_0623.csv
kiwait.0422_0623.csv
```

11.18 Cluster Parse Report (kiinfo -clparse)

The ***kiinfo -clparse*** option produces a report in either text or html format with a variety of information from the LinuxKI binary data created by the ***runki*** script from multiple nodes in a cluster. The clparse option will attempt to identify various patterns and provides warning if a condition _may_ be suspected of causing some performance issues and warrants additional investigation.

When the -clparse option is used, ***kiinfo*** will search all of the subdirectories looking for any LinuxKI data with the same timestamp.

Usage:

```
kiinfo -clparse [flag,flag,...] [options ...]
```

Options:

```
See kiinfo options
```

Flags:

```
help                # Provide help information for clparse flags.
top=<num>           # Only list the top <num> items in each section.  Default is 20.
nooracle            # Do not print the Oracle section of the clparse report.
mangle              # Leave C++ function names mangled
cltree              # Links server names to appropriate kparse HTML report for
                      respective server and links PID numbers to the appropriate
                      PID Reports.  Assumes use of kiinfo -kparse kptree and kiinfo
                      -kipid pidtree was previously performed on each server's KI
                      data.
csv                 # Generates CSV output file with statistics for each server.
```

Examples:

- Running ***kiinfo -clparse*** on LinuxKI data from the ***runki*** script and link servers and PIDs to the respective server overview report or specific PID reports:

  ```
  $ kiinfo -clparse csv,cltree -html -ts 1215_0744 >cp.1215_0744.html
  ```

Typically, the generation of the Cluster Overview Report will be managed through the ***kiall*** script. The -c argument of the ***kiall*** script will cause a subdirectory of the form <timestamp>/<hostname> to be created for each ki_all.*.tgz file found in the current working directory. The ***kiall*** script will proceed to generate the LinuxKI reports for each server. After that, the Cluster Overview Report will be generated. For example:

```
$ ls
ki_all.gwrlx1.0225_1129.tgz       ki_all.gwrlx2.0225_1129.tgz
ki_all.localhost.0225_1129.tgz    ki_all.gwrlx11.0225_1129.tgz
ki_all.gwrlx5.0225_1129.tgz

$ kiall -c
Processing files in: /work/mcr/cluster/0225_1129/gwrlx1
Merging KI binary files.  Please wait...
```

```
ki.bin files merged by kiinfo -likimerge
/opt/linuxki/kiinfo -kitrace -ts 0225_1129
/opt/linuxki/kiinfo -kiall csv -html -ts 0225_1129...
...
/opt/linuxki/kiinfo -clparse csv,cltree,top=20 -html -ts 0225_1129
Number of Servers to analyze: 5
Processing KI files in ./gwrlx5
Processing KI files in ./gwrlx11
Processing KI files in ./gwrlx2
Processing KI files in ./localhost
Processing KI files in ./gwrlx1
kiall complete
```

12 Using LinuxKI to read Windows ETL trace files

Typically, the Windows ETL traces can be analyzed using Windows Performance Analyzer (WPA) or PerfView, which provide some very detailed information about the trace data. But WPA and PerfView can be slow at times, especially for large ETL trace files, and it can be cumbersome to traverse and drilldown on data as the data must be processed frequently.

LinuxKI is not designed to replace existing Windows tools, but to supplement the analysis by bringing the advantages of LinuxKI to Windows ETL trace analysis, including text-based reports, a system overview HTML report, an ASCII-format trace files, and the curses-based user interface. LinuxKI does for Windows what it does for Linux, and helps answer the questions:

- When it's running, what's it doing?
- When it's waiting, what's it waiting on?

12.1 Prerequisites

In order to collect Windows ETL trace files, the Windows Assessment and Deployment Kit (Windows ADK) needs to be installed. The Windows ADK includes the Windows Performance Toolkit and XPERF.

The Windows ADK can be downloaded from the following site:

https://docs.microsoft.com/en-us/windows-hardware/get-started/adk-install

12.2 Setup

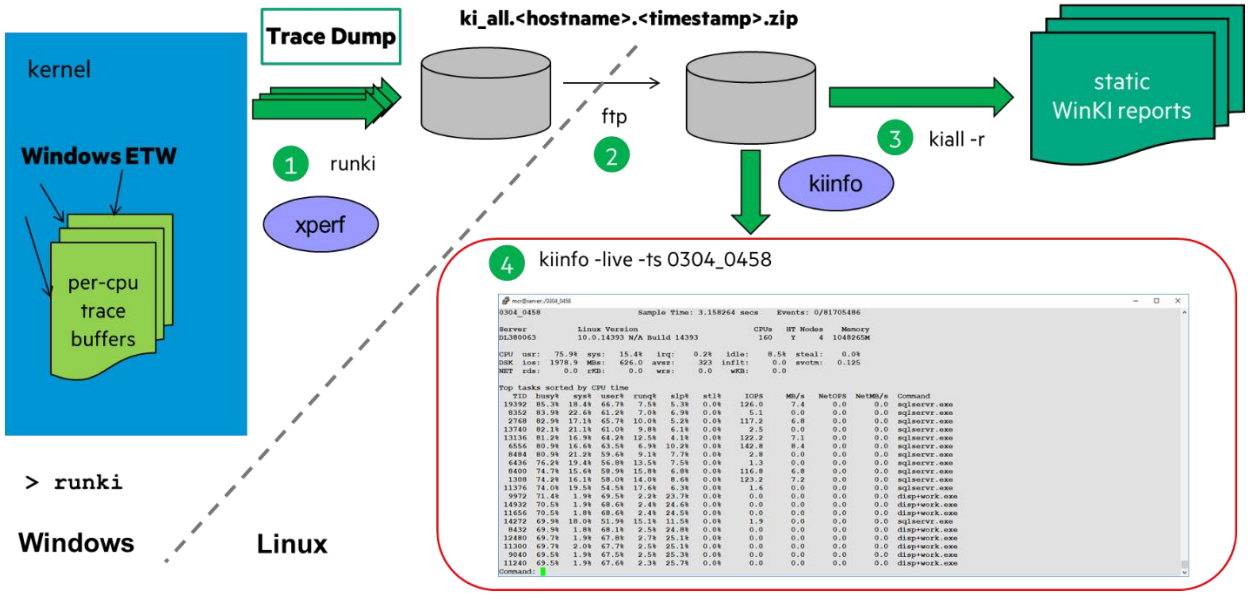
To collect the Windows ETL trace files, you will need to copy the runki.cmd and GetSQLServerInfo.ps1 files from /opt/linuxki/winki from the Linux server with LinuxKI installed or from the following github page to the Windows server to be analyzed:

https://github.com/HewlettPackard/LinuxKI/tree/master/winki)

For best results, put the runki.cmd and GetSQLServerInfo.ps1 scripts into a filesystem with fast storage (SSD preferred if available) and plenty of disk space available (3GB or more).

12.3 Collecting Windows ETL trace data and Analyzing with LinuxKI

The following diagram provides an overview for collecting Windows ETL trace data on a Windows server and then analyzing the data using the LinuxKI Toolset.



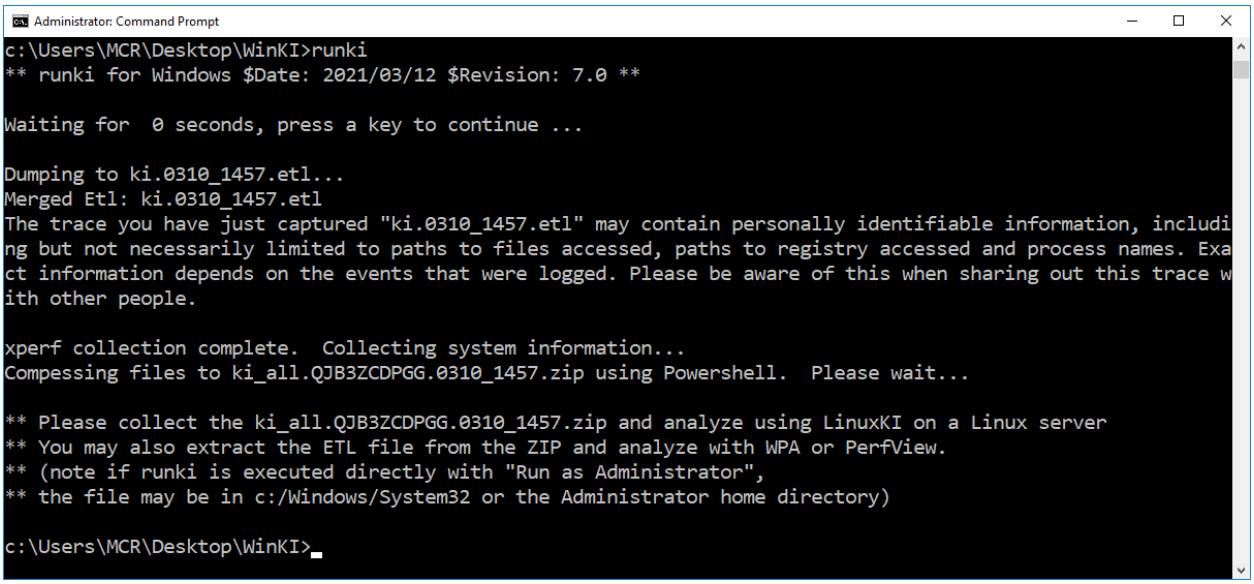
The steps are described in detail below.

1. Collect the Windows ETL trace data using the runki script from LinuxKI.

    a. Open a “cmd” or Powershell window, but be sure to “Run as administrator”.
    b. To get SQL thread names using the GetSQLServerInfo.ps1 script, the user will also need SQLServer permissions.
    c. Use “cd” to change the working directory to the directory where the runki.cmd file was saved.
    d. When you have an interesting workload that needs to be analyzed, execute the runki.cmd script. The runki script will use the Xperf from the Windows Assessment and Deployment Kit (ADK) to collect the trace data and will also collect some supplemental files. You can specify the duration in seconds as the 1st argument (default 20 seconds). You can also press the ‘Enter’ key to stop Xperf at any time.

    ```
    > runki
    ```

    Below is an example of collecting the Windows ETL trace file using the runki script:



2. Move the ki_all.<hostname>.<timestamp>.zip file to a Linux server where LinuxKI is installed for processing and analysis

3. Use the kiall script to perform post-processing and generate the various reports, just as you would for a LinuxKI trace dump.

   ```
   $ kiall -r
   ```

   The "-r" option can be used to create a subdirectory structure using the hostname and timestamp and is recommended the first time you process the KI trace data. It should be omitted if the KI data is already processed and placed into the <hostname>/<timestamp> subdirectory.

4. Optionally, the curses-based user interface can be used to view the statistics generated from the Windows ETL trace data.

   ```
   $ kiinfo -live -ts 0310_1457
   ```

12.4 LinuxKI reports from Windows ETL traces

With LinuxKI version 7.1, many features and reports available for Linux are available with the Windows ETL traces.

File	Command	Description
kparse.*.html	kiinfo -kparse	Kparse Report - System overview of Windows ETL data
kipid.*.txt	kiinfo -kipid	PID Analysis Report
kidsk*.txt	kiinfo -kidsk	Disk Analysis Report
kirunq.*.txt	kiinfo -kirunq	CPU and Runq Analysis Report
kiprof.*.txt	kiinfo -kiprof	CPU Profile Report
kiwait.*.txt	kiinfo -kiwait	Wait Event Analysis Report
kifile.*.txt	kiinfo -kifile	File Activity Report
kisock.*.txt	kiinfo -kisock	Network Activity Report

All of the above	kiinfo -kiall	Generate all of the above reports in a single pass
ki.<timestamp>	kiinfo -kitrace	Format Windows ETL trace data as individual records

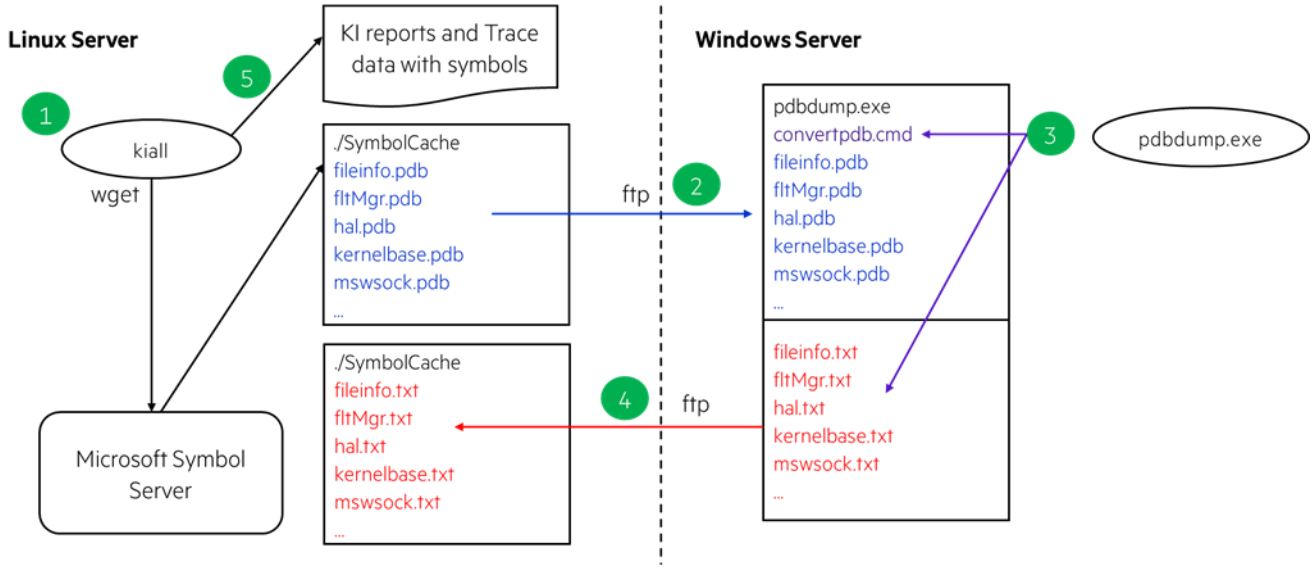
After executing the kiall script, you can use your favorite text editor (vi, vim, etc.) to view the *.txt files. If your files are accessible through a web browser, the Kparse file (kp.*.html) will provide a good overview of the data with the hypertext navigation of a browser.

Unfortunately, the mapping of the function addresses to symbols was not possible on Linux and will be discussed in the next section.

12.5 Obtaining Windows symbols from the Windows Symbol server

When you first execute kiall on the kiall.<hostname>.<timestamp>.zip file, the function addresses are not decoded into function names. However, several Windows symbol table files (PDB files) are automatically downloaded from the Microsoft Symbol Server, provided the Symbol Server is accessible to the Linux server. Be sure to check your http_proxy environment variable if applicable. The following instructions can be used to convert the PDB files to simple text files that LinuxKI can understand.

The following diagram summarizes the steps for performing the symbol table lookups:



A detailed explanation of the steps are provided below:

1. Execute kiall to perform the initial post-processing. The initial kiall will call wget to transfer the PDB file to the Linux server into the SymbolCache directory. Be sure to set the http_proxy variable if applicable as the Linux server will need to connect with the Microsoft Symbol Server.

2. Copy the following files to any Windows server:

```
/opt/linuxki/winki/pdbdump.exe
/opt/linuxki/winki/convertpdb.cmd
```

```
./SymbolCache/*.pdb
```

3. Run the convertpdb.cmd script, which will use pdbdump.exe to convert each PDB file into a simple text file.

4. Copy the *.txt files from the Windows server back to the Linux server into the SymbolCache directory.

5. Execute kiall again to obtain the function names from the SymbolCache directory.

12.6 Sample Data Analysis

After you have executed the kiall script, analyzing the Windows ETL trace data is done just like it is with Linux trace data. There are many similarities, but you will notice some differences as well as the operating systems and trace data are very different.

Below is sample data using the curses-based kiinfo invocation from a Windows ETL trace dump collected with the runki script on Windows. In this example, a specific thread (TID 4508) is being analyzed. The detailed report shows how much time the thread spent running on the CPU, waiting for the CPU, or waiting for other reasons. It also shows the CPU function profile, the task wait events, the IO generated by the tasks (if any), and the system calls made by the thread. All of the data is derived from the detailed Windows ETL trace records collected by the runki script using Xperf. In summary, it's very much like analyzing Linux trace data.

gse-rose-st068

```
1015_2045                    Sample Time: 21.027439 secs   Events: 0/16776575

PID:    4508 CASSI.exe

RunTime     :  2.398512  SysTime    :  0.565999   UserTime    :  1.818851
SleepTime   : 17.974354  Sleep Cnt  :     24275   Wakeup Cnt  :       795
RunQTime    :  0.426383  Switch Cnt:      24275   PreemptCnt  :         0
Last CPU    :         3  CPU Migrs  :       4372   NODE Migrs  :         0
Policy      : SCHED_NORMAL

------------------------- Top Hardclock Functions --------------------------
   Count     Pct  State  Function
     341  16.91%  USER   UNKNOWN [msodbcsql17.dll]
     212  10.51%  USER   UNKNOWN [LE370.dll]
     160   7.93%  USER   UNKNOWN [cblrtsm.dll]
     104   5.16%  SYS    RtlpWalkFrameChain [ntoskrnl.exe]
      87   4.31%  USER   UNKNOWN [odbcrw32.dll]

---------------------------- Top Wait Functions ----------------------------
   Count     SlpTime  Msec/Slp  MaxMsecs  Func
   23949   17.556581     0.733     6.618  KeWaitForSingleObject

----------------------------- Top System Calls -----------------------------
System Call Name                Count     Rate     ElpTime        Avg        Max    Errs    AvSz     KB/s
NtWaitForSingleObject           24478   1164.1   18.410537   0.000752   0.515920       0
   SLEEP                        24238   1152.7   17.678341   0.000729
      Sleep Func                23914            17.521185   0.000733  KeWaitForSingleObject
   RUNQ                                          0.425832
   CPU                                           0.082841
NtDeviceIoControlFile           54825   2607.3    0.444028   0.000008   0.004110       0
NtReadFile                        121      5.8    0.028911   0.000239   0.023906       0
   SLEEP                            9      0.4    0.026479   0.002942
      Sleep Func                    9             0.026479   0.002942  KeWaitForSingleObject
   RUNQ                                          0.000127
   CPU                                           0.002070
NtOpenProcess                     201      9.6    0.013954   0.000069   0.000168       0
NtFreeVirtualMemory                49      2.3    0.011411   0.000233   0.000690       0
NtQueryAttributesFile              22      1.0    0.010955   0.000498   0.003015       0
   SLEEP                            6      0.3    0.001431   0.000239
      Sleep Func                    6             0.001431   0.000239  KeWaitForSingleObject
   RUNQ                                          0.000106
   CPU                                           0.009418
NtWriteFile                       296     14.1    0.007321   0.000025   0.000957       0
   SLEEP                            6      0.3    0.001961   0.000327
      Sleep Func                    5             0.001769   0.000354  KeWaitForSingleObject
   RUNQ                                          0.000075
   CPU                                           0.005052
NtClose                           409     19.5    0.007261   0.000018   0.002608       0
NtCreateFile                        8      0.4    0.003999   0.000500   0.000826       0
NtReleaseSemaphore                201      9.6    0.002679   0.000013   0.000036       0
NtAlpcSendWaitReceivePort          55      2.6    0.001307   0.000024   0.000048       0
Command:
```

Note that the symbols for user binaries and libraries are not available, so the function will show UNKNOWN, however, the name of the binary or library is provided.

12.7 Special note on analyzing Windows ETL traces with LinuxKI

Analyzing Windows ETL traces with LinuxKI is still in its infancy. Some features are planned but not yet implemented, such as visualization charts and graphs, and cluster processing.

Also note that the tracing overhead is higher on Windows than on Linux. So for the 20 seconds or so when the trace data is collected with runki/Xperf, there may be some minor impacts on performance and the trace data will show elevated System Time.

Version History

7.2 - 11/9/21

- Added .NET PDB (clr.pdb) to list of PDBs collected during Windows trace analysis
- Fix issue with nested system calls in Windows traces which mess up the system call stats. Now, outer nested system calls times will be 0.
- Enabled FileIo FileRundown trace events and increased the file object hash size to improve filename lookups.
- Fixed issue in runki script where only first FC interface is listed in the fc_linkspeed file
- Collect per-NUMA node meminfo data into the numa_meminfo file.
- Print warning when sysignore file is used with kiinfo when doing trace dump analysis. The sysignore file only ignores the system calls during the actual tracing of the events.
- Updated link on Windows TCP Timeout issue

7.1 - 7/8/2021

- Added IO Controllers section in Kparse report (for Linux)
- Added Memory DIMM section in Kparse report (for Linux)
- Kparse Warning for KVM host page faults (for Linux)
- Kparse Warning for Smart Array Controllers (for Linux)
- Kparse Warning for Oracle column statistics (for Linux)
- Kparse Warning for Memory imbalance across NUMA nodes (for Linux)
- Kparse Warning for Low Memory on a NUMA node (for Linux)
- Kparse Warning for Windows TCP timeouts (for Windows)
- Kparse Warning for SQL auto statistics (for Windows)
- Added File Activity Report (kifile.*.txt (for Windows)
- Added Socket Activity Report - kisock.*.txt (for Windows)
- Added SQL thread names to KI reports (for Windows)
- Decode thread wait_reason and mode (for Windows)
- Modify liki Makefile to support Oracle Linux (OEL) to get better stack traces
- CSV files added for Windows data (kipid.*.csv, kidsk.*.csv, kirunq.*.csv, kiwait.*.csv, kifile.*.csv, kisock.*.csv)
- Fixed LiKI to get IP addresses after version 4.17.0 due to changes in getname() function
- Fixed kiinfo coredump when doing clparse (bug in print_socket_lip())
- fixed coredump when using runki -I <sysignore>, with kiinfo -likidump due to missing initialization of the syscall_arg_list variable (introduced in 7.0)
- Fixed fc_linkspeed output in runki script as it was not appending to the file
- fixed bug with kiall -f if you have a file name with valid bash option prefix in it, such as "--badfile"
- add additional PDBs for symbol table lookup (for Windows)
- fixed command name of threads when doing kiinfo -live on live system due to incorrect use of cmdstr variable (for Linux)
- Do not hash filename for FileIo Rundown events as this could be excessive (for Windows)
- Fixed several visualization issues

7.0 - 3/12/2021

- Add support for Linux kernels through 5.9.16

- Add support for collecting and analyzing Windows ETL trace files
- Fixed coredump when analyzing KI data from ftrace from Linux kernels >5.0
- Fixed select() fds_bytes on syscall exit records
- Fixed file open modes to print RDONLY, WRONLY, or RDWR on open() trace records
- Fixed coredump in print_hardclock_rec()/print_stacktrace() due to uninitialized variable in parse_kallsyms()
- Fixed various formatting and alignment issues

6.2 - 10/16/2020

- Add support for Linux kernels through 5.8.14
- Enabled LinuxKI to work on Power servers running Linux
- Enabled LinuxKI to work with ArchLinux
- Fixed LiKI Makefile to work for CentOS 8.2
- Fixed Bus error coredump from kiinfo -likimerge (via kiall) on ftrace collections
- Improved formatting for arguments for the following system calls: mprotect, munmap, mbind, openat
- Removed all trace events (mostly hidden events) that relied on kprobes except for hardclock events on older Linux versions.
- For multi-threaded processes in a docker container, threads will inherit docker information from primary thread (TGID)
- Print docker name on kiinfo -live docker screens of columns > 180 characters
- Enhance runki script to collect FC Link Speeds, “docker info” output, and cpupower output
- Add read access to supplemental files in kiall script
- Fixed cstate times for cstate 0 (CSTATE_BUSY)
- Several formatting and alignment fixes in the reports
- Removed page_fault_user/page_fault_kernel hidden trace events to avoid a crash on SLES 15 when using ftrace tracing (this is not a bug in LinuxKI, but in Linux).

6.1 - 04/19/2020

- Add support for Linux kernels through 5.6.4
- Added support for RHEL 8.2
- Improved performance of kiinfo by removing the block frequency stats from kparse reports as the new default behavior
- Added the ***blkfrq*** flag for kiinfo -kiall and kiinfo -kparse and the -B option to the kiall script to include block frequency statistics in the kparse reports.
- The ***scdetail*** flag has been removed on the kiinfo reports as all reports will show the system call details
- Fixed Docker stats to include the threads of a multi-threaded process
- Fixed kiinfo coredump due to very long docker names
- Fixed kiinfo to include the full pathnames for insmod/rmmod
- Fixed kparse HTML file to work with text-based browsers such as Lynx.

6.0 - 11/25/2019

- Improved speed of visualization processing (kiall -V) by adding parallel processing (runki -V -P [num_threads]

- Fixed LiKI compiler errors for RHEL 8.1
- Fixed SID Table Overflow with kiinfo -kipid oracle processing.
- Modified kipid Per-pid I/O stats format for total I/O.
- Fixed AIO Write metrics in kipid output
- Fixed potential crash dump due to changes in perf_callchain_entry structure. (only observed on Power systems)
- Fixed permissions issues and removed unused files.

5.10 - 09/20/2019

- Add support for Linux kernels through 5.2.8
- Fixed System Crash due to race with hardclock timer being re-armed after LiKI module is unloaded when running kiinfo -live
- Add Avg IRQ Time to kiinfo -live IRQ screens
- Fixed irqtm or SoftIRQ if hardirq occurs in middle of softirq
- Increased Global Futex hash size to speed up kiinfo processing if there are many Futexes
- Fixed kiinfo futex reporting as some futexes were not printed
- Fixed Oracle SID printing
- Modified FD type for sendfile from hex to decimal
- Include sendfile64() system calls in file statistics
- print DockerID instead of Docker name due to very long docker names
- Improve Docker reporting in kparse output
- Fixed kiinfo -live coredumps if empty DockerID or Dev is entered
- Modified LiKI Makefile to include OpenSuse
- Add bash auto-completion for kiinfo
- Automatically update LinuxKI version in Dockerfile for each new release

5.9 - 03/26/2019

- Added per-pid device-mapper statistics to kipid
- Decode result= field in scsi_dispatch_cmd_done records if non-zero
- Fixed block_rq_complete records as spid, qpid, qtm, and stm fields were 0 starting with version 5.8
- Fixed Barrier Write detection
- Fixed numa node assignments when numa=off, up to 96 cores per NUMA node supported
- Fixed likimerge loop if trace binary is truncated prematurely causing a liki trace record size to be 0
- Changes to support RHEL 8
    - Removed in_flight reference in likit.c in 4.18 kernels and later as the field has been deprecated in the kernel
    - Fixed liki makefile so it supports RHEL 8
    - Fixed potential system crash after unloading likit.ko due to race in timer shutdown code. Seen when testing RHEL 8.

5.8 - 01/22/2019

- Enhance Kparse output to detect the following new Warnings:
    - Excessive CPU time in pcc_cpufreq driver

    - Lock contention using large Oracle SGA and huge pages on large multi-core systems
    - High SYS CPU time by processes reading /proc/stat such as ora_diag and ora_lmhb
    - Excessive poll() calls by Oracle
    - Side-Channel Attack (Spectre/Meltdown) mitigation can negatively impact performance
- Detect Sub-Numa Clustering (SNC) and use SNC nodes rather than physical notes when reporting
- Fix incorrect Network and File stats caused by frequent open() and close() using the same FD

5.7 - 11/20/2018

- Fixed defect where IRQ stats were not counted if CPU is IDLE. Bug was introduced in version 5.6.

5.6 - 11/07/2018

- Added support for Linux kernels through 4.19.0
- Increased MAXCPUS to 2048 and MAXLDOMS from 16 to 128
- Add filename for access() system calls
- Add Target Path WWN statistics to help troubleshoot SAN issues for kidsk, kparse and kiinfo live
- Handle splice/vmsplice system calls for tracking logical IOs
- Fix SleepTime for processes that died during the trace
- Fix runki script so trace-cmd output goes to /dev/null instead of /dev=null
- Fix sock_types to print the correct socket type on socket-related system calls

5.5 - 08/13/2018

- Added support for Linux kernels through 4.17.6
- Add docker container to support viewing HTML reports with visualizations - kivis-build, kivis-start, kivis-stop commands.
- Include kiinfo source code in RPM package
- Add average interrupt time to kirunq and Kparse output
- Fixed demangled c++ function names with kiinfo -live stack traces
- Fixed warnings with newer compilers
- Fixed system call detail statistics in kifile.*.txt output
- Fixed kipid.*.csv file wen using Advanced CPU Metrics (runki -R)
- Modified to avoid compile warning for kiinfo on later compiler versions

5.4 - 05/10/2018

- Added support for Linux kernels through 4.16.3.
- IRQ trace events now enabled by default.
- Added new workqueue events as non-default events
- Adding Kparse warning for the following conditions
    - How add_random impacts block device performance
    - Network-latency tuned profile may increase System CPU usage and decrease overall performance

    - High kworker CPU usage when using software RAID (md driver) with barrier writes
- Enabled Advanced CPU statistics on Skylake processors.
- Disabled Advanced CPU statistics for Virtual Machines as it was unreliable for some VMs.
- Added LinuxKI manpages. See man linuxki(7).
- Added /etc/profile.d/linuxki.sh to add /opt/linuxki so the PATH variable.
- Added support to demangle C++ function names (and also an option to leave them mangled if desired).
- Added Top Tasks by Multipath device to kidsk/kparse/clparse output to help identify tasks generating IO when kworkers initiate the IO at the SCSI layer
- For runki script, added -p option to skip per-PID data (lsof, stacks, numa_maps_maps) to avoid long delays if system has thousands of tasks on the system.
- Improved error reporting if online analysis is done without root access or if debugfs is not mounted.
- Added code to clear /sys/module/kgdboc/parameters/kgdboc to avoid crash as it is incompatible with LinuxKI. Typically, customer systems do not have kgdboc set, but some internal lab systems do.
- Fixes
    - Change madvise/mmap/mmap2 length argument formatting from decimal to hex
    - Fixed kiinfo coredump when parsing cpuinfo output due to missing "@" character when looking for the GHz speed
    - for kiinfo -live, fixed Global CPU usage (usr/sys% is sometimes off) on main global screen
    - Fixed the multipath parsing to understand lines that start with "|-|-"
    - Removed PID stats from CPU window when running kiinfo -live on dumps as this stat is not available with trace dumps.

5.3 - 08/24/2017

- Added support for Linux kernels through 4.12.8
- Fixed block_rq flags for 2.6.39 kernels
- Fixed barrier write detection in kparse and kidsk output

5.2 - 06/20/2017

- Fix "mpsched-S_data" syntax error when executing the runki script.
- Fix IPv4 IP addresses (no R= address) in kipid File Statistics
- Changed kparse warnings to point to HTML files on github
- Added warning for RHEL 7.3 multipath bug
- change maplookup() logic so it is called if symlookup() fails

5.1 - 06/07/2017

- LinuxKI introduced as an open source tool
- Fix for looping kiinfo thread when using kiinfo -live on an ftrace dump and one or more CPUs have zero records logged

- Ability to customer symbol lookups using <PID>.map files which includes the starting address, length, and symbol names
- Fix for system crash when collecting advanced CPU statistics (runki -R) on a KVM guest.
- Add /sbin:/usr/sbin to PATH variable in runki script.

5.0 - 04/24/2017

- Docker enhancements
  - kiinfo -kidock command to create the Docker Report file (kidock.<timestamp>.txt).
  - kiall script now produces new Docker Report file (kidock.<timestamp.txt) to report activity for each docker container
  - kiinfo -live has been enhanced with a new Global Docker Screen (using the 'k' key) to show Docker usage by container based on CPU usage and I/O throughput.
  - Kiinfo -live has been enhanced with a new Select Docker Screen (using the 'K' key) to drill down on a specific docker container, showing the top tasks in the docker container using CPU and performing I/O.
  - Docker Container name listed outside of process/task name (i.e. <container_name>)
  - Added new Docker section to Kparse report if docker containers are present.
- IRQ enhancements
  - Added irqtm to IRQ exit traces
  - Improved IRQ stats when HardIRQ interrupts SoftIRQ
  - Added Select IRQ Stats (I) screen on kiinfo -live
- Added stack traces on sched_switch in TSRUN state for kiinfo -kipid output
- Fixed User Stack traces when analyzing traces from Linux kernels > 3.0, which was caused by Linux kernel changes.
- Improved/Fixed network IP addresses
- Added stack tracing back to INTR hardclock trace events
- Added Top Tasks by Hardclock for Select CPU window when running kiinfo -live on a KI dump
- Fixed formatting of ioflags on block_rq_* trace events due to many changes in the Linux kernel header file
- Fixed HTML links due to sharepoint migration
- Fixed File Detail reports for multi-threaded tasks
- Modified module_prep flags to avoid "likit: disagrees about version of symbol module_layout" message during installation on most Linux servers.
- Removed bytes= field from block_rq_* trace events as it's not needed.

4.6 - 01/11/2017

- Password is no longer required for post-processing/analysis with any system
- Support Linux Kernels through 4.8.15
- Include pread64/pwrite64 in file statistics in File Activity Report
- Include dev_t/ino for pread64/pwrite64 system calls
- Fixed coredump in print_slp_info() due to unknown kernel symbol
- Fix LiKI Makefile to support Fedora
- runki now reports numastats for the duration of the traces (not just since bootup).

4.4 - 09/26/2016

- New “step” feature to allow the user to move through a KI dump in time intervals when using kiinfo in curses mode
- New advanced CPU metrics such as LLC Hit%, CPI, Avg CPU Freq, and SMI counts. Enabled using the “-R” flag on the runki command or using the “msr” flag on certain kiinfo commands, such as kiinfo -live, kiinfo -kipid, kiinfo -kitrace, and kiinfo -kirunq.
- Fixed mapper stats due to failure to open the devices file
- Fixed VIS socket wheel data

4.3 - 07/18/2016

- Support Linux kernels through 4.4
- Support ARM64 processors
- Support for IPv6 IP addresses
- Added edus flag for kiinfo -live to get DB2 thread names
- Added warning for Power Savings vs. Performance in kparse report
- Trim pidp->cmd to last pathname if > 32 characters
- Fixed per-syscall RunQ times in kiinfo -kipid
- Fixed total number of CPUs in kiinfo -kiall
- Fixed filenames/sockets for multi-threaded tasks when parsing lsof output
- Fixed block_rq traces when filtering data
- Fixed incorrect headings in clparse CSV
- New -V option for runki filters and options
- Fixed bus error when analyzing data from ftrace

4.2 - 04/05/2016

- Fixed coredump when block_rq_requeue/abort records were present
- Fixed missing softirq names in kirunq output
- Fixed interval_start initialization for VIS when time filtering is used
- Fixed pollfd revents fields and improved print display
- Fixed missing HT CPUs in live mode
- Fixed coredump caused by double=free on rsock/lsock during dup2 trace records
- Fixed power savings report in clparse output
- Added Global HBA Stats (z) in kiinfo -live
- Added kparse warning for unaligned XFS Direct I/O
- Added kparse warning for XFS Direct IO on files with cached pages
- Added Last Pid in Global File Report in kiinfo -live
- Added sleep pct in kipid output. Fixed bug in percentages

4.1 - 02/29/2016

- New curses-based user interface - kiinfo -live
- New Visualization charts

    - New PID timeline
    - New scheduling timeline
    - If Java stacks (jstack) is collected, include Java thread names in KI reports
- reduce memory usage when kiinfo is collecting/analyzing live data
- Fixed memory leaks when kiinfo is collecting/analyzing live data
- Add epochtime flag to print KI trace record timestamps in seconds since 1970
- Fixed coredump on long kiinfo command line, for example, if there are lots of filters
- Fixed coredump in parse_mpath if only 3 arguments are present on the mpath line
- Fixed RUNQ_HISTOGRAM flag setting with kipid rqhist
- Fixed incorrect number of CPUs for LDOMs in kirunq report is generated with kiinfo -kiall
- Many other bug fixes!
- Included *LiKI* 4.3 DLKM
    - bug fix with system call filtering
    - fix build for Debian, CentOS, RHEL7
    - fix stack tracing for SLES 12
    - fix to re-enable kernel probes
    - added socket type to syscall traces to identify UDP and IPv6 traffic
    - fixed TGID field on several trace records
    - fixed crash when filtered PID is in an interrupt
    - fixed dev in filesystem syscalls to use rdev is non-null, otherwise, use dev
- runki enhancements
    - fixed repeated functions in stack trace
    - runki modified so it does NOT collect data, perf data, and sar data by default
    - new runki options -M -U -X to collect, perf data, and sar data, respectively
    - collect ethtool output for non-Ethernet devices
    - Include -o blkdevname and dmsetup table in dmsetup output
    - increase file hard limit to support 576 logical cores (GriffinHawk)

4.0 - 11/31/2015

- Includes *LiKI* 4.2 DLKM
    - Allows filtering on PID, TGID, DEV, or CPU when collecting the trace records which can greatly reduce the amount of data collected if only certain tasks, devices, or CPUs need to be traced
    - Expand filtering to follow fork()/clone() system calls
    - Allows filtering out some system calls (so frequent system calls like gettimeofday can be ignored)
    - Include inode and dev for filesystem system calls
    - Improved IP:Port reporting for network system calls it works with IPv4 addresses over IPv6
    - Network socket calls now include the socket_type
    - Include "stealtime" on sched_switch records to report amount of CPU stolen by the KVM host is using a KVM guest
- Improved filtering using *LiKI* DLKM as well as the post-processing and live tracing with kiinfo. Includes filtering on Task Group ID (TGID), PID, Device, or CPU

- New "sysignore=<filename>" flag so user can provide a file which contains a list of system calls to ignore.
- Include "stealtime" for CPUs (kiinfo -kirunq) and tasks (kiinfo -kirunq)
- Added new kparse section for top tasks with the most steal time
- Include kparse warning if stealtime exceeds 10% for any CPU
- Added kparse warning for semaphore lock contention issue when semaphore tunables are misconfigured
- New kitrace flag when running kiinfo on live system so formatted traces records are printed prior to the summary report (i.e. kiinfo -kipid pid=1234,kitrace -a 5)
- Change kiinfo -kitrace defaults so it prints syscall entry records and formats the arguments (obsolete sysenter and sysargs options). Add nosysenter flag to prevent printing syscall entry records and the nosysargs flag to perform generic argument processing.
- Formatted val3 argument for futex() system call when called with FUTEX_WAKE_OP operation
- Added argument return values for poll/ppoll/select/pselect system call exit record
- Removed unreliable stack traces for IDLE and INTR hardclock traces
- Removed misleading Incomplete AIO warnings from kparse output
- Fixed issue where coop and sleep stats were not reset after each pass when using live tracing (kiinfo with -a <secs> option)
- Fixed VM detection on KVM guests
- Fixed bug where password was needed for a runki collection even if system analyzed was an HP system
- Fixed missing TGID from some *LiKI* trace records
- Fixed coredump when lsof contained very long filenames
- Fixed coredump for Linux 4.2.0 collections using *ftrace* due to long *ftrace* event name
- Fixed coredump is run on system with >512 logical cores (i.e. GriffinHawk)
- Fixed missing requeue counts for per-disk statistics (kiinfo -kipid output)
- Improvements to the runki script shipped
    - modified to take advantage of *LiKI* filtering using the -G <tgid> -P <pid> -C <cpu> and -D <device> options
    - modified to take advantage of *LiKI* system call ignore feature using the -I <ignore_file> option
    - collects tuned and clocksource data
    - improved to add cstate names to cstate.$tag file

3.2 - 08/17/2015

- HBA statistics are double counted when used with kiinfo -kiall
- Fix coredump from VM guest with more than 16 cores
- Added links to .txt files and VIS files in kparse output
- Fix up disk CSV file to print HW path and HBA FC path and mapper name for each device
- Add AIO statistics to system calls and files using io_submit
- Formatted syscall arguments in several syscalls (mprotect, mmap, ipc, lseek, kill, sigaction, etc...)
- Fixed Port numbers printed in KI ASCII as they were in Big Indian and needed to be converted)
- Fixed USER library HARDCLOCKS showing up as UNKNOWN

3.1 - 05/28/2015

- Removed fingerprint check causing "Program kiinfo tried to access /dev/mem between 7990b000->7990b010"
- fixed fmttime on kiinfo when using filtering such as kiinfo -kitrace fmttime,pid=<PID>
- fixed coredump when printing stack trace from *LiKI* v1 traces due to bug in conv_sched_switch()
- fixed ENOMEM errors in map_elf()
- Fixed hardirq total if missed buffer occurs
- Fixed sched stats when doing time filtering (-start option)
- Fixed symlookup when objfile spans more than one memory region
- Added xnode_servers CSV file and re-worked kisock CSV file
- Added binary search to speedup symlookup64()
- Added additional vis code
- Added oracle flag to kiall when visualization is used
- Changed cstate display and pct to omit cstate 0
- Added kparse warning if kernel.numa_balancing=1 and system is spending time migrating pages
- Added kparse warning if numa=off is set at boot time

3.0 - 02/28/2014

- New Network Activity Report (kiinfo -kisock)
- Include Network statistics in Kparse (kiinfo -kparse) and Cluster (kiinfo -clparse) reports
- Include IO statistics by FC HBA when multipathing is used in Disk Activity Report (kiinfo -kidsk) and Kparse.
- Group SCSI disks (/dev/sd*) by multipath lun in Disk Activity Report (kiinfo -kidsk)
- Print Local and Remote IP:Port on IPv4-related system calls in KI ASCII traces (kiinfo -kitrace)
- Print io_submit() and io_getevents() IO control blocks (iocb) in KI ASCII traces (kiinfo -kitrace)
- New fmttime option on kiinfo -kitrace to format the record timestamp providing date/time of each trace record
- Eliminate printing of getrusage() and time() system calls from KI ASCII traces (kiinfo -kitrace) to reduce the size of the output
- Group new Oracle Log Writer slaves (ora_lg*) with the master Log Writer process (ora_lgwr) in Oracle PID Analysis Report (kiinfo -kipid oracle) and Kparse.
- Add Cluster-wide Server Summary at the beginning of the Cluster Report (kiinfo -clparse)
- New processing to allow for faster time filtering using -start and -end time-filtering options
- User stack traces provided in KI ASCII trace (kiinfo -kitrace) and PID Analysis Report (kiinfo -kipid)
- symbolic lookup for user binary and library functions on off-line data collections collected with the runki script
- DB2 thread name lookup using edus= option or specifying an edus.<timestamp> file with customer supplied output of "db2pd -edus" command
- New monthly password for post-processing of data from non-HP hardware or VMs.'

2.4 - 09/04/2014

- Fixed missing barrier and requeue counts in kparse report
- Fixed missing Oracle processes from kipid output when using kiinfo -kiall
- Fixed cstate reporting cstate3 when cstates are disabled and runki -e all is done
- Fixed detection of mapper device major number

- Fixed cpu/ldom migration count
- Fixed problem sysargs flag
- In ki_clone(), inherit the cmd name from parent thread
- Fixed sysargs for clone() system call
- Fixed kirunq to print delay properly (label said ms, but was printing seconds) when rqdelay was used
- Fixed kirunq/kipid to skip RunQ latencies if thread is in an unknown state
- Improved buffer miss handling in kiinfo when data is collected using liki
- Changed gettid return value to decimal
- Changed add_command to replace name if it is different to fix thread names when prctl(PR_SET_NAME) is called
- Fixed issue where there is no data in KI trace ASCII file when data is collected using *ftrace*
- Fixed incorrect CPU count when data is from VM guest
- Fixed coredump in kiinfo with *ftrace* on RHEL7 due to change in the syscall_enter/format file
- Modified runki to include more ethtool data and sar -A data (if available)

2.3 - 04/23/2014

- Added changes to support new lsof output with potential TID field
- updated kiinfo to print file type when printing syscall info if available
- Changed hardclock processing to update the schedp->sched_stat.state if needed. This allows more correct accounting for tasks that do not make system calls (either all USER time or all SYS time).
- parse dmsetup file to correctly identify the device mapper major number
- Fixed bad avque from kipid report
- Fixed Logical IO reporting in kifile report
- Added missing thread names in kipid CSV report and some sections in kparse
- Added Hadoop process names in {} in reports
- Added -kiall option to generate all the reports in a single pass
- Added starttime/endtime filtering
- Added CSV and file links to the new File Links Section of the Kparse report
- Initial KI Cluster Report (kiinfo -clparse) to create a kparse-like output for a set of KI trace files from multiple nodes in a cluster

2.2 - 01/23/2014

- Added symlookup and objfile= option for USER PC symbols in kipid and kitrace output
- Added report= option for kipid
- Added thread names for offline kifutex, kiprof, and kidsk reports
- Added warning for SAP processes calling semget() and getting ENOENT
- propagate file name when doing an fcntl(F_DUPFD) or dup() system call
- fix CPU and PID filter for kiprof
- Fixed futex uaddr filtering for kifutex
- Fixed bug for dev= filtering for kidsk
- fixed coredump in futex_update_wakeup_stats() due to bad call to find_hash()

2.1 - 12/09/2013

- Added Futex Reports in Kparse output and per-PID futex details in kipid output
- Added kifutex report (including live tracing)
- For multithreaded tasks, fixed bug where top files were not reported in the global reports
- sort per-pid FD by Elapsed time, rather than by ftype
- In kiinfo -kitrace, always set svtm=0.0 for barrier writes (sector == 0)
- Fixed C-state reporting (runki script)
- Better reporting for block device parameters (runki script)
- Get stack traces for tasks in a multithreaded process (runki script)
- Fixed kiall so that it can execute on multiple ki_all files with the same timestamp (kiall script)

2.0 - 11/08/2013

- Support for the LinuxKI has been expanded to support more Linux distributions - RHEL 6/7, SLES 11 SP2/SP3, Ubuntu 12.04/12.10, CentOS 6, OEL 6 (*ftrace* only)
- The kiall/kiclean scripts are now included in the linuxki RPM patch. So now there is now a single linuxki RPM for each supported distribution, eliminating the linuxki-hponly RPMs
- The password for the KI post-processing has been eliminated if the data is generated from physical HP server. See also Password Processing
- Support for live KI tracing and analysis using the *LiKI* tracing mechanism. See also Running kiinfo on a live system
- Added signal handling so the runki script and kiinfo -kitracedump/likidump can properly cleanup after a Control-C or process abort is performed
- Improved scaling when collecting data using the *LiKI* DLKM tracing module
- Improved stack unwinding on SLES 11 SP2/SP3
- The runki script now collects cstate times for each CPU eliminating the need for LinuxKI to collect power_start and power_end traces, reducing the overall overhead of LinuxKI by reducing the number of trace events collected.
- New non-default trace events added:
    - scsi_dispatch_cmd_start / scsi_dispatch_cmd_done
    - irq_handler_entry / irq_handler_exit
    - softirq_entry / softirq_exit / softirq_raise
    - sched_migrate_task
    - listen_overflow (*LiKI* only)
- Addition of key arguments for some system calls (*LiKI* only):
    - open/creat/lstat/stat/unlink - filename
    - exec/execve - executable name
    - select - fd_set bitmasks (readfds, writefds, exceptfds)
    - pselect - fd_set bitmasks and timeout value
    - poll - fds bitmask
    - ppoll - fds bitask and timeout value
- Enhancement to trace fields in *LiKI* DLKM tracing modules to improve reporting
    - Flag in system call to identify if system call is 32-bit or 64-bit to provide more reliable system call names
    - Addition of async_in_flight and sync_inflight counters on block events

    - More accurate IO service times on block_rq_complete traces
    - Virtual Set Size (vss) and Resident Set Size (rss) values for a patches in sched_switch records
    - Corrected target CPU reporting in power_freq trace records
- New csv option available for most of the KI analysis reports
- kinfo -kparse, kiinfo -kirunq, and kiinfo -kipid have been enhanced to include IRQ statistics if irq events are enabled
- kiinfo -kidsk has been enhanced to show average inflight IO (avinflt) when a new IO is issued
- kiinfo -kifile has new scdetail option to show additional detail about a system call for top files
- Fixed bug in parsing kallsyms file which lead to increased runtimes for kiinfo post-processing

1.2 - 06/18/2013

- Fixed c/r in the "task woken up" section of the PIDS/<PID> output (changed printf to pid_printf)
- Do not include system calls with EBADF in the FD section
- Fixed c/r in print_fd_info() of the PIDS/<PID> output
- Fixed kiinfo to work with Ubuntu and RHEL 7 *ftrace* data
- Fixed kipid.oracle output
- Added "flush" field to report barrier IOs in kidsk/kipid/kparse reports
- Added BARRIER write warning
- Added logical IO status such as AvSz and KB/sec to the system calls
- Fixed Trace Events for top 10 processes
- Allow syscall return values to be either hex or decimal depending on the syscall_arglist*[] array values
- Removed SystemT (total_systime) from the trace type info
- Added Oracle Kparse Section

1.1 - 03/25/2013

- Includes *LiKI* version v0.8_rc4 to improve scaling
- Fixed Last CPU / Migrations counters run on non-likimerge binaries
- added per_cpu and mpath_detail options to kiinfo -kidsk (1.0c)
- Fixed SlpTime in "Threads woken up by this task" section, due to uninitialized coop_delta variable
- Fixes to accommodate running LinuxKI with *ftrace* on OEL 6. Note that *LiKI* does not run on OEL 6.x at this release

1.0 - 01/28/2013

- Initial Release