#Srinivasan Kannan (alias) Ka.Shrinivaasan (alias) Shrinivas Kannan
#Ph: 9791499106, 9003082186
#Krishna iResearch Open Source Products Profiles:
#http://sourceforge.net/users/ka_shrinivaasan,
#https://github.com/shrinivaasanka,
#https://www.openhub.net/accounts/ka_shrinivaasan
#Personal website(research): https://sites.google.com/site/kuja27/
#emails: ka.shrinivaasan@gmail.com, shrinivas.kannan@gmail.com,
#kashrinivaasan@live.com

VIRGO is an operating system kernel forked off from Linux kernel mainline to add cloud functionalities (system calls, modules etc.,) within kernel itself with machine learning, analytics, debugging, queueing support in the deepest layer of OSI stack i.e AsFer, USBmd, KingCobra together with VIRGO constitute the previous functionalities. Presently there seems to be no cloud implementation with fine-grained cloud primitives (system calls, modules etc.,) included in kernel itself though there are coarse grained clustering and SunRPC implementations available. VIRGO complements other Clustering and application layer cloud OSes like cloudstack, openstack etc., in these aspects - CloudStack and OpenStack can be deployed on a VIRGO Linux Kernel Cloud - OpenStack nova compute, neutron network, cinder/swift storage subsystems can be augmented to have additional drivers that invoke lowlevel VIRGO syscall and kernel module primitives (assuming there are no coincidental replications of functionalities) thereby acting as a foundation to application layer cloud.

## Memory pooling:
---------------
Memory pooling is proposed to be implemented by a new virgo_malloc() system call that transparently allocates a block of virtual memory from memory pooled from virtual memory scattered across individual machines part of the cloud.

## CPU pooling or cloud ability in a system call:
----------------------------------------------
Clone() system call is linux specific and internally it invokes sys_clone(). All fork(),vfork() and clone() system calls internally invoke do_fork(). A new system call virgo_clone() is proposed to create a thread transparently on any of the available machines on the cloud.This creates a thread on a free or least-loaded machine on the cloud and returns the results.

virgo_clone() is a wrapper over clone() that looks up a map of machines-to-loadfactor and get the host with least load and invokes clone() on a function on that gets executed on the host. Usual cloud implementations provide userspace API that have something similar to this - call(function,host). Loadfactor can be calculated through any of the prominent loadbalancing algorithm. Any example userspace code that uses clone() can be replaced with virgo_clone() and all such threads will be running in a cloud transparently.Presently Native POSIX threads library(NPTL) and older LinuxThreads thread libraries internally use clone().

Kernel has support for kernel space sockets with kernel_accept(), kernel_bind(), kernel_connect(), kernel_sendmsg() and kernel_recvmsg() that can be used inside a kernel module. Virgo driver implements virgo_clone() system call that does a kernel_connect() to a remote kernel socket already __sock_create()-d, kernel_bind()-ed and kernel_accept()-ed and does kernel_sendmsg() of the function details and kernel_recvmsg() after function has been executed by clone() in remote machine. After kernel_accept() receives a connection it reads the function and parameter details. Using these kthread_create() is executed in the remote machine and results are written back to the originating machine. This is somewhat similar to SunRPC but adapted and made lightweight to suit virgo_clone() implementation without any external data representation.

## Experimental Prototype

virgo_clone() system call and a kernel module virgocloudexec which implements Sun RPC interface have been implemented.

## VIRGO - loadbalancer to get the host:ip of the least loaded node

### Loadbalancer option 1 - Centralized loadbalancer registry that tracks load:

Virgo_clone() system call needs to lookup a registry or map of host-to-load and get the least loaded host:ip from it. This requires a  load monitoring code to run periodically and update the map. If this registry is located on a single machine then simultaneous virgo_clone() calls from many machines on the cloud could choke the registry. Due to this, loadbalancer registry needs to run on a high-end machine. Alternatively,each machine can have its own view of the load and multiple copies of load-to-host registries can be stored in individual machines. Synchronization of the copies becomes a separate task in itself(Cache coherency). Either way gives a tradeoff between accuracy, latency and efficiency.

Many application level userspace load monitoring tools are available but as virgo_clone() is in kernel space, it needs to be investigated if kernel-to-kernel loadmonitoring can be done without userspace data transport.Most Cloud API explicitly invoke a function on a host. If this functionality is needed, virgo_clone() needs to take host:ip address as extra argument,but it reduces transparent execution.

(Design notes for LB option 1 handwritten by myself are at :http://sourceforge.net/p/virgo-linux/code-0/HEAD/tree/trunk/virgo-docs/MiscellaneousOpenSourceDesignAndAcademicResearchNotes.pdf)

### Loadbalancer option 2 - Linux Psuedorandom number generator based load balancer(experimental) instead of centralized registry that tracks load:

Each virgo_clone() client has a PRG which is queried (/dev/random or /dev/urandom) to get the id of the host to send the next virgo_clone() function to be executed
Expected number of requests per node is derived as:

expected number of requests per node =
summation(each_value_for_the_random_variable_for_number_of_requests *

probability_for_each_value) where random variable ranges from 1 to k where N is number of processors and k is the number of requests to be distributed on N nodes

=expected number of requests per node = (math.pow(N, k+2) - k*math.pow(N,2) + k*math.pow(N,1) - 1) / (math.pow(N, k+3) - 2*math.pow(N,k+2) + math.pow(N,k+1))

This loadbalancer is dependent on efficacy of the PRG and since each request is uniformly, identically, independently distributed use of PRG
would distribute requests evenly. This obviates the need for loadtracking and coherency of the load-to-host table.

(Design notes for LB option 2 handwritten by myself at :http://sourceforge.net/p/virgo-linux/code-0/HEAD/tree/trunk/virgo-docs/MiscellaneousOpenSourceDesignAndAcademicResearchNotes.pdf)

(python script in virgo-python-src/)

****************************************************************************************************
Implemented VIRGO Linux components (as on 7 March 2016)
****************************************************************************************************
1. cpupooling virtualization - VIRGO_clone() system call and VIRGO cpupooling driver by which a remote procedure can be invoked in kernelspace.(port: 10000)
2. memorypooling virtualization - VIRGO_malloc(), VIRGO_get(), VIRGO_set(), VIRGO_free() system calls and VIRGO memorypooling driver by which kernel memory can be allocated in remote node, written to, read and freed - A kernelspace memcache-ing.(port: 30000)
3. filesystem virtualization - VIRGO_open(), VIRGO_read(), VIRGO_write(), VIRGO_close() system calls and VIRGO cloud filesystem driver by which file IO in remote node can be done in kernelspace.(port: 50000)
4. config - VIRGO config driver for configuration symbols export.
5. queueing - VIRGO Queuing driver kernel service for queuing incoming requests, handle them with workqueue and invoke KingCobra service routines in kernelspace.(port: 60000)
6. cloudsync - kernel module for synchronization primitives (Bakery algorithm etc.,) with exported symbols that can be used in other VIRGO cloud modules for critical section lock() and unlock()
7. utils - utility driver that exports miscellaneous kernel functions that can be used across VIRGO Linux kernel
8. EventNet - eventnet kernel driver to vfs_read()/vfs_write() text files for EventNet vertex and edge messages (port: 20000)
9. Kernel_Analytics - kernel module that reads machine-learnt config key-value pairs set in /etc/virgo_kernel_analytics.conf. Any machine learning software can be used to get the key-value pairs for the config. This merges three facets - Machine Learning, Cloud Modules in VIRGO Linux-KingCobra-USBmd , Mainline Linux Kernel
10. Testcases and kern.log testlogs for the above
11. SATURN program analysis wrapper driver.

Thus VIRGO Linux at present implements a minimum cloud OS (with cloud-wide cpu, memory and file system management) over Linux and potentially fills in a gap to integrate both software and hardware into cloud with machine learning and analytics abilities that is absent in application layer cloud implementations. Thus VIRGO cloud is an IoT operating system kernel too that enables any hardware to be remote controlled. Data analytics using AsFer can be done by just invoking requisite code from a kernelspace driver above and creating an updated driver binary (or) by kernel_analytics module which reads the userland machine-learnt config.

*************************************************************************************************************************

VIRGO ToDo and NiceToHave Features (list is quite dynamic and might be rewritten depending on feasibility - longterm with no deadline)
*************************************************************************************************************************

(FEATURE - DONE-minimum separate config file support in client and kernel service )1. More Sophisticated VIRGO config file and read_virgo_config() has to be invoked on syscall clients virgo_clone and virgo_malloc also. Earlier config was being read by kernel module only which would work only on a single machine. A separate config module kernel service has been added for future use while exporting kernel-wide configuration related symbols. VIRGO config files have been split into /etc/virgo_client.conf and /etc/virgo_cloud.conf to delink the cloud client and kernel service config parameters reading and to do away with oft occurring symbol lookup errors and multiple definition errors for num_cloud_nodes and node_ip_addrs_in_cloud - these errors are frequent in 3.15.5 kernel than 3.7.8 kernel. Each VIRGO module and system call now reads the config file independent of others - there is a read_virgo_config_<module>_<client_or_service>() function variant for each driver and system call. Though at present smacks of a replicated code, in future the config reads for each component (system call or module) might vary significantly depending on necessities.  New kernel module config has been added in drivers/virgo. This is for future prospective use as a config export driver that can be looked up by any other VIRGO module for config parameters. include/linux/virgo_config.h has the declarations for all the config variables declared within each of the VIRGO kernel modules.  Config variables in each driver and system call have been named with prefix and suffix to differentiate the module and/or system call it serves.  In geographically distributed cloud virgo_client.conf has to be in client nodes and virgo_cloud.conf has to be in cloud nodes. For VIRGO Queue - KingCobra REQUEST-REPLY peer-to-peer messaging system same node can have virgo_client.conf and virgo_cloud.conf.  Above segregation largely simplifies the build process as each module and system call is independently built without need for a symbol to be exported from other module by pre-loading it.(- from commit comments done few months ago)

(FEATURE - Special case implementation DONE) 2. Object Marshalling and Unmarshalling (Serialization) Features - Feature 4 is a marshalling feature too as Python world PyObjects are serialized into VIRGO linux kernel and unmarshalled back bottom-up with CPython and Boost::Python C++ invocations - CPython and Boost internally take care of serialization.

(FEATURE - DONE) 3. Virgo_malloc(), virgo_set(), virgo_get() and virgo_free() syscalls that virtualize the physical memory across all cloud nodes into a single logical memory behemoth (NUMA visavis UMA). (There are random crashes in copy_to_user and copy_from_user in syscall path for VIRGO memory pooling commands that were investigated but turned out to be mystery). These crashes have either been resolved or occur less in 3.15.5 and 4.1.5 kernels.
Initial Design Handwritten notes committed at: http://sourceforge.net/p/virgo-linux/code-0/210/tree/trunk/virgo-docs/VIRGO_Memory_Pooling_virgomalloc_initial_design_notes.pdf

(FEATURE - DONE) 4. Integrated testing of AsFer-VIRGO Linux Kernel request roundtrip - invocation of VIRGO linux kernel system calls from AsFer Python via C++ or C extensions - Commits for this have been done on 29 January 2016. This unifies userlevel applications and kernelspace modules so that AsFer Python makes VIRGO linux kernel an extension. Following is schematic diagram and More details in commit notes below.

4.1 Schematic Diagram:
```
----------------------
        AsFer Python -----> Boost::Python C++ Extension ------> VIRGO memory 
system calls --------> VIRGO Linux Kernel Memory Drivers
        /\
V
         |
|

---------------------------------------------<-------------------------------------
----------------

        AsFer Python -----> CPython Extensions ------> VIRGO memory system calls
--------> VIRGO Linux Kernel Memory Drivers
         /\
V
         |
|

---------------------------------------------<-------------------------------------
----------------
```

(FEATURE - DONE)5. Multithreading of VIRGO cloudexec kernel module (if not already done by kernel module subsystem internally)

(FEATURE - ONGOING) 6. Sophisticated queuing and persistence of CPU and Memory pooling requests in Kernel Side (by possibly improving already existing kernel workqueues). Either open source implementations like ZeroMQ/ActiveMQ can be used or Queuing implementation has to be written from scratch or both. ActiveMQ supports REST APIs and is JMS implementation.

(FEATURE - DONE-Minimum Functionality) 7. Integration of Asfer(AstroInfer) algorithm codes into VIRGO which would add machine learning capabilities into VIRGO - That is, VIRGO cloud subsystem which is part of a linux kernel installation "learns" and "adapts" to the processes that are executed on VIRGO. This catapults the power of the Kernel and Operating System into an artificially (rather approximately naturally) intelligent computing platform (a software "brain"). For example VIRGO can "learn" about "execution times" of processes and suitably act for future processes. PAC Learning of functions could be theoretical basis for this.  Initial commits for Kernel Analytics Module which reads the /etc/virgo_kernel_analytics.conf config have been done. This config file virgo_kernel_analytics.conf having csv(s) of key-value pairs of analytics variables is set by AsFer or any other Machine Learning code.  With this VIRGO Linux Kernel is endowed with abilities to dynamically evolve than being just a platform for user code. Implications are huge - for example, a config variable "MaxNetworkBandwidth=255" set by the ML miner in userspace based on a Perceptron or Logistic Regression executed on network logs can be read by a kernel module that limits the network traffic to 255Mbps. Thus kernel is no longer a static predictable blob behemoth. With this, VIRGO is an Internet-of-Things kernel that does analytics and based on analytics variable values integrated hardware can be controlled across the cloud through remote kernel module function invocation. This facility has been made dynamic with Boost::Python C++ and CPython extensions that permit flow of objects from machine learnt AsFer kernel analytics variables to VIRGO Linux Kernel memory drivers via VIRGO system calls directly and back - Commits on 29 January 2016 - this should obviate re-reading /etc/virgo_kernel_analytics.conf and is an exemplary implementation which unifies C++/Python into C/Kernel.

----------------------------------------
Example scenario 1 without implementation:
----------------------------------------
- Philips Hue IoT mobile app controlled bulb - http://www2.meethue.com/en-xx/
- kernel_analytics module learns key-value pairs from the AsFer code and exports it VIRGO kernel wide
- A driver function with in bulb embedded device driver can be invoked through VIRGO cpupooling (invoked from remote virgo_clone() system_call)
based on if-else clause of the kernel_analytics variable i.e remote_client invokes virgo_clone() with function argument "lights on" which is routed to another cloud node. The recipient cloud node "learns" from AsFer kernel_analytics that Voltage is low or Battery is low from logs and decides to switch in high beam or low beam.

-----------------------------------------
Example scenario 2 without implementation:
-----------------------------------------
- A swivel security camera driver is remotely invoked via virgo_clone() in the VIRGO cloud.
- The camera driver uses a machine learnt variable exported by kernel_analytics-and-AsFer to pan the camera by how much degrees.
--------------------------------------------------------------------------------------------------------
Example scenario 3 without implementation - probably one of the best applications of NeuronRain IoT OS:
--------------------------------------------------------------------------------------------------------
- Automatic Driverless Automobiles - a VIRGO driver for a vehicle which learns kernel analytics variables (driving directions) set by kernel_analytics driver and AsFer Machine Learning. A naive algorithm for Driverless Car (with some added modifications over A-Star and Motion planning algorithms):

      - AsFer analytics receives obstacle distance data 360+360 degrees (vertical and horizontal) around the vehicle (e.g ultrasound sensors) which is updated in a Spark DataFrame table with high frequency (100 times per second).

      - VIRGO Linux kernel on vehicle has two special drivers for Gear-Clutch-Break-Accelerator-Fuel(GCBAF) and Steering listening on some ports.

      - AsFer analytics with high frequency computes threshold variables for applying break, clutch, gear, velocity, direction, fuel changes which are written to kernel_analytics.conf realtime based on distance data from Spark table.

      - These analytics variables are continuously read by GCBAF and Steering drivers which autopilot the vehicle.

      - Above applies to Fly-by-wire aeronautics too with appropriate changes in analytics variables computed.

      - The crucial parameter is the response time in variable computation and table updates which requires a huge computing power unless the vehicle is hooked onto a Spark cloud in motion by wireless which process the table and compute analytic variables.

----------------------------------------------
References for Machine Learning + Linux Kernel
----------------------------------------------
7.1 KernTune - http://repository.uwc.ac.za/xmlui/bitstream/handle/10566/53/Yi_KernTune(2007).pdf?sequence=3
7.2 Self-learning, Predictive Systems - https://icri-ci.technion.ac.il/projects/past-projects/machine-learning-for-architecture-self-learning-predictive-computer-systems/
7.3 Linux Process Scheduling and Machine Learning - http://www.cs.ucr.edu/~kishore/papers/tencon.pdf
7.4 Network Latency and Machine Learning - https://users.soe.ucsc.edu/~slukin/rtt_paper.pdf
7.5 Machine Learning based Meta-Scheduler for Multicore processors - https://books.google.co.in/books?id=1GWcHmCrl0QC&pg=PA528&lpg=PA528&dq=linux+kernel+machine+learning&source=bl&ots=zfJsq_uu5q&sig=mMIUZ-oyJIwZXtYj4HntrQE8NSk&hl=en&sa=X&ved=0CCAQ6AEwATgKahUKEwjs9sqF9vPIAhVBFZQKHbNtA6A

8. A Symmetric Multi Processing subsystem Scheduler that virtualizes all nodes in cloud (probably this would involve improving the loadbalancer into a scheduler with priority queues)

(FEATURE - ONGOING) 9. Virgo is an effort to virtualize the cloud as a single machine - Here cloud is not limited to servers and desktops but also mobile devices that run linux variants like Android, and other Mobile OSes. In the longterm, Virgo may have to be ported or optimized for handheld devices.

(FEATURE - DONE) 10. Memory Pooling Subsystem Driver - Virgo_malloc(), Virgo_set(), Virgo_get() and Virgo_free() system calls and their Kernel Module Implementations. In addition to syscall path, telnet or userspace socket client interface is also provided for both VIRGO CPU pooling(virgo_clone()) and VIRGO Memory Pooling Drivers.

(FEATURE - DONE) 11. Virgo Cloud File System with virgo_cloud_open(), virgo_cloud_read() , virgo_cloud_write() and virgo_cloud_close() commands invoked through telnet path has been implemented that transcends disk storage in all nodes in the cloud. It is also fanciful feature addition that would make VIRGO a complete all-pervading cloud platform. The remote telnet clients send the file path and the buf to be read or data to be written. The Virgo File System kernel driver service creates a unique Virgo File Descriptor for each struct file* opened by filp_open() and is returned to client. Earlier design option to use a hashmap (linux/hashmap.h) looked less attractive as file desciptor is an obvious unique description for open file and also map becomes unscalable. The kernel upcall path has been implemented (paramIsExecutable=0) and may not be necessary in most cases and all above cloudfs commands work in kernelspace using VFS calls.

(FEATURE - DONE) 12. VIRGO Cloud File System commands through syscall paths - virgo_open(),virgo_close(),virgo_read() and virgo_write(). All the syscalls have been implemented with testcases and more bugs fixed. After fullbuild and testing, virgo_open() and virgo_read() work and copy_to_user() is working.

(FEATURE - DONE) 13. VIRGO memory pooling feature is also a distributed key-value store similar to other prominent key-store software like BigTable implementations, Dynamo, memory caching tools etc., but with a difference that VIRGO mempool is implemented as part of Linux Kernel itself thus circumventing userspace latencies. Due to Kernel space VIRGO mempool has an added power to store and retrieve key-value pair in hardware devices directly which otherwise is difficult in userspace implementations.

14. VIRGO memory pooling can be improved with disk persistence for in-memory key-value store using virgo_malloc(),virgo_set(),virgo_get() and virgo_free() calls. Probably this might be just a set of invocations of read and write ops in disk driver or using sysfs. Probably this could be redundant as the VIRGO filesystem primitives have been implemented that write to a remote host's filesystem in kernelspace.

15. (FEATURE-DONE) Socket Debugging, Program Analysis and Verification features for user code that can find bugs statically. Socket skbuff debug utility and SATURN Program Analysis Software has been integrated into NEURONRAIN VIRGO Linux Kernel.

16(FEATURE - DONE-Minimum Functionality). Operating System Logfile analysis using Machine Learning code in AstroInfer for finding patterns of processes execution and learn rules from the log. Kernel_Analytics VIRGO module reads /etc/virgo_kernel_analytics.conf config key-value pairs which are set by AsFer or other Machine Learning Software. At present an Apache Spark usecase that mines Uncomplicated Fire Wall logs in kern.log for most prominent source IP has been implemented in AsFer codebase : http://sourceforge.net/p/asfer/code/704/tree/python-src/SparkKernelLogMapReduceParser.py . This is set as a key-value config in /etc/virgo_kernel_analytics.conf read and exported by kernel_analytics module.

17. Implementations of prototypical Software Transactional Memory and LockFree Datastructures for VIRGO memory pooling.

18. Scalability features for Multicore machines - references: (http://halobates.de/lk09-scalability.pdf, http://pdos.csail.mit.edu/papers/linux:osdi10.pdf)

19. Read-Copy-Update algorithm implementation for VIRGO memory pooling that

supports multiple simultaneous versions of memory for readers - widely used in redesigned Linux Kernel.

20. (FEATURE - SATURN integration - minimum functionality DONE) Program Comprehension features as an add-on described in : https://sites.google.com/site/kuja27/PhDThesisProposal.pdf?attredirects=0. SATURN program analysis has been integrated into VIRGO linux with a stub driver.

21. (FEATURE - DONE) Bakery Algorithm implementation - cloudsync kernel module

22. (FEATURE - ONGOING) Implementation of Distributed Systems primitives for VIRGO cloud viz., Logical Clocks, Termination Detection, Snapshots, Cache Coherency subsystem etc.,(as part of cloudsync driver module). Already a simple timestamp generation feature has been implemented for KingCobra requests with <ipaddress>:<localmachinetimestamp> format

23. (FEATURE - minimum functionality DONE) Enhancements to kmem if it makes sense, because it is better to rely on virgo_malloc() for per machine memory management and wrap it around with a cloudwide VIRGO Unique ID based address lookup implementation of which is already in place.
Kernel Malloc syscall kmalloc() internally works as follows:
      - kmem_cache_t object has pointers to 3 lists
      - These 3 lists are full objects SLAB list, partial objects SLAB list and free objects SLAB list - all are lists of objects of same size
 and cache_cache is the global list of all caches created thus far.
      - Any kmalloc() allocation searches partial objects SLAB list and allocates a memory block with kmem_cache_alloc() from the first SLAB available - returned to caller.
      - Any kfree() returns an object to a free SLAB list
      - Full SLABs are removed from partial SLAB list and appended to full SLAB list
      - SLABs are virtual memory pages created with kmem_cache_create
      - Each SLAB in SLABs list has blocks of similar sized objects (e.g. multiples of two). Closest matching block is returned and fragmentation is minimized (incidentally this is the knapsack and packing optimization LP problem and thus NP-complete).

KERNELSPACE:
VIRGO address translation table already implements a tree registry of vtables each of capacity 3000 that keep track of kmalloc() allocations across all cloud nodes. Implementation of SLAB allocator for kmalloc() creates a kmem_cache(s) of similar sized objects and kmem_cache_alloc() allocates from these caches. kmalloc() already does lot of per-machine optimizations. VIRGO vtable registry tree maintained in VIRGO memory syscall end combined with per-machine kmalloc() cache_cache already look sufficient. Instrumenting kmem_cache_create() with #ifdef SLAB_CLOUD_MALLOC flags to do RPC looks superfluous. Hence marking this action item as done. Any further optimization can be done on top of existing VIRGO address translation table struct - e.g bookkeeping flags, function pointers etc.,.
USERSPACE: sbrk() and brk() are no longer used internally in malloc() library routines. Instead mmap() has replaced it (http://web.eecs.utk.edu/courses/spring2012/cs360/360/notes/Malloc1/lecture.html , http://web.eecs.utk.edu/courses/spring2012/cs360/360/notes/Malloc1/diff.html).

24.(FEATURE - ONGOING) Cleanup the code and remove unnecessary comments.

25.(FEATURE - DONE) Documentation - This design document is also a documentation for commit notes and other build and debugging technical details. Doxygen html cross-reference documentation for AsFer, USBmd, VIRGO, KingCobra and Acadpdrafts has been created along with summed-up design document and committed to GitHub Repository at https://github.com/shrinivaasanka/Krishna_iResearch_DoxygenDocs

26. (FEATURE - DONE) Telnet path to virgo_cloud_malloc,virgo_cloud_set and virgo_cloud_get has been tested and working. This is similar to memcached but

stores key-value in kernelspace (and hence has the ability to write to and retrieve from any device driver memory viz., cards, handheld devices).An optional todo is to write a script or userspace socket client that connects to VIRGO mempool driver for these commands.

27. Augment the Linux kernel workqueue implementation (http://lxr.free-electrons.com/source/kernel/workqueue.c) with disk persistence if feasible and doesn't break other subsystems - this might require additional persistence flags in work_struct and additional #ifdefs in most of the queue functions that write and read from the disk. Related to item 6 above.

28.(FEATURE - DONE) VIRGO queue driver with native userspace queue and kernel workqueue-handler framework that is optionally used for KingCobra and is invoked through VIRGO cpupooling and memorypooling drivers. (Schematic in http://sourceforge.net/p/kcobra/code-svn/HEAD/tree/KingCobraDesignNotes.txt and http://sourceforge.net/p/acadpdrafts/code/ci/master/tree/Krishna_iResearch_opensourceproducts_archdiagram.pdf)

29.(FEATURE - DONE) KERNELSPACE EXECUTION ACROSS CLOUD NODES which geographically distribute userspace and kernelspace execution creating a logical abstraction for a cloudwide virtualized kernel:

```
      Remote Cloud Node Client
      (cpupooling, eventnet, memorypooling, cloudfs, queueing - telnet and
syscalls clients)
            |
            |
 (Userspace)      |
            |-------------------------------------Kernel
Sockets-------------------------------------> Remote Cloud Node Service
                                                     (VIRGO cpupooling,
memorypooling, cloudfs, queue, KingCobra drivers)

      |

      |

      |  (Kernelspace execution)

      |

      V
            <-------------------------------------Kernel
Sockets---------------------------------------------
            |
            |
            |
 (Userspace)      |
```

30. (FEATURE - DONE) VIRGO platform as on 5 May 2014 implements a minimum set of features and kernelsocket commands required for a cloud OS kernel: CPU virtualization(virgo_clone), Memory virtualization(virgo_malloc,virgo_get,virgo_set,virgo_free) and a distributed cloud file system(virgo_open,virgo_close,virgo_read,virgo_write) on the cloud nodes and thus gives a logical view of one unified, distributed linux kernel across all cloud nodes that splits userspace and kernelspace execution across cloud as above.

31. (FEATURE - DONE) VIRGO Queue standalone kernel service has been implemented in addition to paths in schematics above. VIRGO Queue listens on hardcoded port 60000 and enqueues the incoming requests to VIRGO queue which is serviced by KingCobra:

VIRGO Queue client(e.g telnet) ------> VIRGO Queue kernel service ---> Linux Workqueue handler ------> KingCobra

32. (FEATURE - DONE) EventNet kernel module service:
VIRGO eventnet client (telnet) -------> VIRGO EventNet kernel service -----> EventNet graph text files

33. (FEATURE - DONE) Related to point 22 - Reuse EventNet cloudwide logical time infinite graph in AsFer in place of Logical clocks. At present the eventnet driver listens on port 20000 and writes the edges and vertices files in kernel using vfs_read()/vfs_write(). These text files can then be read by the AsFer code to generate DOT files and visualize the graph with graphviz.

34. (FEATURE - OPTIONAL) The kernel modules services listening on ports could return a JSON response when connected instead of plaintext, conforming to REST protocol. Additional options for protocol buffers which are becoming a standard data interchange format.

35. (FEATURE-Minimum Functionality DONE) Pointer Swizzling and Unswizzling of VIRGO addressspace pointers to/from VIRGO Unique ID (VUID). Presently VIRGO memory system calls implement a basic minimal pointer address translation to unique kmem location identifier.

*************************************************************************************************
CODE COMMIT RELATED NOTES
*************************************************************************************************

VIRGO code commits as on 16/05/2013
-----------------------------------
1. VIRGO cloudexec driver with a listener kernel thread service has been implemented and it listens on port 10000 on system startup
through /etc/modules load-on-bootup facility

2. VIRGO cloudexec virgo_clone() system call has been implemented that would kernel_connect() to the VIRGO cloudexec service listening at
port 10000

3. VIRGO cloudexec driver has been split into virgo.h (VIRGO typedefs), virgocloudexecsvc.h(VIRGO cloudexec service that is invoked by
module_init() of VIRGO cloudexec driver) and virgo_cloudexec.c (with module ops definitions)

4. VIRGO does not implement SUN RPC interface anymore and now has its own virgo ops.

5. Lot of Kbuild related commits with commented lines for future use have been done viz., to integrate VIRGO to Kbuild, KBUILD_EXTRA_SYMBOLS for cross-module symbol reference.

VIRGO code commits as on 20/05/2013
----------------------------------
1. test_virgo_clone.c testcase for sys_virgo_clone() system call works and connections are established to VIRGO cloudexec kernel module.

2. Makefile for test_virgo_clone.c and updated buildscript.sh for headers_install for custom-built linux.

VIRGO code commits as on 6/6/2013
--------------------------------
1. Message header related bug fixes

VIRGO code commits as on 25/6/2013
---------------------------------
1.telnet to kernel service was tested and found working
2.GFP_KERNEL changed to GFP_ATOMIC in VIRGO cloudexec kernel service

VIRGO code commits as on 1/7/2013
----------------------------------
1. Instead of printing iovec, printing buffer correctly prints the messages
2. wake_up_process() added and function received from virgo_clone() syscall is executed with kernel_thread and results returned to
virgo_clone() syscall client.

commit as on 03/07/2013
-----------------------
PRG loadbalancer preliminary code implemented. More work to be done

commit as on 10/07/2013
-----------------------
Tested PRG loadbalancer read config code through telnet and virgo_clone. VFS code to read from virgo_cloud.conf commented for testing

commits as on 12/07/2013
------------------------
PRG loadbalancer prototype has been completed and tested with test_virgo_clone and telnet and symbol export errors and PRG errors have been fixed

commits as on 16/07/2013
-----------------------
read_virgo_config() and read_virgo_clone_config()(replica of read_virgo_config()) have been implemented and tested to read the virgo_cloud.conf config parameters(at present the virgo_cloud.conf has comma separated list of ip addresses. Port is hardcoded to 10000 for uniformity across all nodes). Thus minimal cloud functionality with config file  support is in place. Todo things include function pointer lookup in kernel service, more parameters to cloud config file if needed, individual configs for virgo_clone() and virgo kernel service, kernel-to-userspace upcall and execution instead of kernel space, performance tuning etc.,

commits as on 17/07/2013
------------------------
moved read_virgo_config() to VIRGOcloudexec's module_init so that config is read at boot time and exported symbols are set beforehand.
Also commented read_virgo_clone_config() as it is redundant

commits as on 23/07/2013
------------------------

Lack of reflection kind of facilities requires map of function_names to pointers_to_functions to be executed
on cloud has to be lookedup in the map to get pointer to function. This map is not scalable if number of functions are
in millions and size of the map increases linearly. Also having it in memory is both CPU and memory intensive.
Moreover this map has to be synchronized in all nodes for coherency and consistency which is another intensive task.
Thus name to pointer function table is at present not implemented. Suitable way to call a function by name of the function
is yet to be found out and references in this topic are scarce.

If parameterIsExecutable is set to 1 the data received from virgo_clone() is not a function but name of executable
This executable is then run on usermode using call_usermodehelper() which internally takes care of queueing the workstruct

and executes the binary as child of keventd and reaps silently. Thus workqueue component of kernel is indirectly made use of.
This is sometimes more flexible alternative that executes a binary itself on cloud and
is preferable to clone()ing a function on cloud. Virgo_clone() syscall client or telnet needs to send the message with name of binary.

If parameterIsExecutable is set to 0 then data received from virgo_clone() is name of a function and is executed in else clause
using dlsym() lookup and pthread_create() in user space. This unifies both call_usermodehelper() and creating a userspace thread
with a fixed binary which is same for any function. The dlsym lookup requires mangled function names which need to be sent by
virgo_clone or telnet. This is far more efficient than a function pointer table.

call_usermodehelper() Kernel upcall to usermode to exec a fixed binary that would inturn execute the cloneFunction in userspace
by spawning a pthread. cloneFunction is name of the function and not binary. This clone function will be dlsym()ed
and a pthread will be created by the fixed binary. Name of the fixed binary is hardcoded herein as
"virgo_kernelupcall_plugin". This fixed binary takes clone function as argument. For testing libvirgo.so has been created from
virgo_cloud_test.c and separate build script to build the cloud function binaries has been added.

 - Ka.Shrinivaasan (alias) Shrinivas Kannan (alias) Srinivasan Kannan
   (https://sites.google.com/site/kuja27)

commits as on 24/07/2013
------------------------

test_virgo_clone unit test case updated with mangled function name to be sent to remote cloud node. Tested with test_virgo_clone
end-to-end and all features are working. But sometimes kernel_connect hangs randomly (this was observed only today and looks similar
to blocking vs non-blocking problem. Origin unknown).

- Ka.Shrinivaasan (alias) Shrinivas Kannan (alias) Srinivasan Kannan
  (https://sites.google.com/site/kuja27)

commits as on 29/07/2013
------------------------

Added kernel mode execution in the clone_func and created a sample kernel_thread for a cloud function. Some File IO logging added to upcall
binaries and parameterIsExecutable has been moved to virgo.h

commits as on 30/07/2013
------------------------
New usecase virgo_cloud_test_kernelspace.ko kernel module has been added. This exports a function virgo_cloud_test_kernelspace() and is
accessed by virgo_cloudexec kernel service to spawn a kernel thread that is executed in kernel addresspace. This Kernel mode execution
on cloud adds a unique ability to VIRGO cloud platform to seamlessly integrate hardware devices on to cloud and transparently send commands
to them from a remote cloud node through virgo_clone().

Thus above feature adds power to VIRGO cloud to make it act as a single "logical device driver" though devices are in geographically in a remote server.

commits as on 01/08/2013 and 02/08/2013
---------------------------------------
Added Bash shell commandline with -c option for call_usermodehelper upcall

clauses to pass in remote virgo_clone command message as
arguments to it. Also tried output redirection but it works some times that too with a fatal kernel panic.

Ideal solutions are :
1. either to do a copy_from_user() for message buffer from user address space (or)
2. somehow rebuild the kernel with fd_install() pointing stdout to a VFS file* struct. In older kernels like 2.6.x, there is an fd_install code
with in kmod.c (___call_usermodehelper()) which has been redesigned in kernel 3.x versions and fd_install has been removed in kmod.c .
3. Create a Netlink socket listener in userspace and send message up from kernel Netlink socket.

All the above are quite intensive and time consuming to implement.Moreover doing FileIO in usermode helper is strongly discouraged in kernel docs

Since Objective of VIRGO is to virtualize the cloud as single execution "machine", doing an upcall (which would run with root abilities) is
redundant often and kernel mode execution is sufficient. Kernel mode execution with intermodule function invocation can literally take over
the entire board in remote machine (since it can access PCI bus, RAM and all other device cards)

As a longterm design goal, VIRGO can be implemented as a separate protocol itself and sk_buff packet payload from remote machine
can be parsed by kernel service and kernel_thread can be created for the message.

commits as on 05/08/2013:
-------------------------
Major commits done for kernel upcall usermode output logging with fd_install redirection to a VFS file. With this it has become easy for user space to communicate runtime data to Kernel space. Also a new strip_control_M() function has been added to strip \r\n or " ".

11 August 2013:
---------------
Open Source Design and Academic Research Notes uploaded to
http://sourceforge.net/projects/acadpdrafts/files/MiscellaneousOpenSourceDesignAndAcademicResearchNotes_2013-08-11.pdf/download

commits as on 23 August 2013
----------------------------
New Multithreading Feature added for VIRGO Kernel Service - action item 5 in ToDo list above (virgo_cloudexec driver module). All dependent headers changed for kernel threadlocalizing global data.

commits as on 1 September 2013
------------------------------
GNU Copyright license and Product Owner Profile (for identity of license issuer) have been committed. Also Virgo Memory Pooling - virgo_malloc() related initial design notes (handwritten scanned) have been
committed(http://sourceforge.net/p/virgo-linux/code-0/HEAD/tree/trunk/virgo-docs/VIRGO_Memory_Pooling_virgomalloc_initial_design_notes.pdf)

commits as on 14 September 2013
-------------------------------
Updated virgo malloc design handwritten nodes on kmalloc() and malloc() usage in kernelspace and userspace execution mode of virgo_cloudexec service
(http://sourceforge.net/p/virgo-linux/code-0/HEAD/tree/trunk/virgo-docs/VIRGO_Memory_Pooling_virgomalloc_design_notes_2_14September2013.pdf). As described in handwritten notes, virgo_malloc() and related system calls might be

needed when a large scale allocation of kernel memory is needed when in kernel space execution mode and large scale userspace memory when in user modes (function and executable modes). Thus a cloud memory pool both in user and kernel space is possible.

---------------------------------------
VIRGO virtual addressing
---------------------------------------
VIRGO virtual address is defined with the following datatype:

```
struct virgo_address
{
      int node_id;
      void* addr;
};
```

VIRGO address translation table is defined with following datatype:

```
struct virgo_addr_transtable
{
      int node_id;
      void* addr;
};
```

------------------------------------------------
VIRGO memory pooling prototypical implementation
------------------------------------------------
VIRGO memory pooling implementation as per the design notes committed as above is to be implemented as a prototype under separate directory under drivers/virgo/memorypooling and $LINUX_SRC_ROOT/virgo_malloc. But the underlying code is more or less similar to drivers/virgo/cpupooling and $LINUX_SRC_ROOT/virgo_clone.

virgo_malloc() and related syscalls and virgo mempool driver connect to and listen on port different from cpupooling driver. Though all these code can be within cpupooling itself, mempooling is implemented as separate driver and co-exists with cpupooling on bootup (/etc/modules). This enables clear demarcation of functionalities for CPU and Memory virtualization.

Commits as on 17 September 2013
-------------------------------
Initial untested prototype code - virgo_malloc and virgo mempool driver - for VIRGO Memory Pooling has been committed - copied and modified from virgo_clone client and kernel driver service.

Commits as on 19 September 2013
-------------------------------
3.7.8 Kernel full build done and compilation errors in VIRGO malloc and mempool driver code and more functions code added

Commits as on 23 September 2013
-------------------------------
Updated virgo_malloc.c with two functions, int_to_str() and addr_to_str(), using kmalloc() with full kernel re-build.
(Rather a re-re-build because some source file updates in previous build got deleted somehow mysteriously. This could be related to Cybercrime issues mentioned in https://sourceforge.net/p/usb-md/code-0/HEAD/tree/USBmd_notes.txt )

Commits as on 24 September 2013
-------------------------------
Updated syscall*.tbl files, staging.sh, Makefiles for virgo_malloc(),virgo_set(),virgo_get() and virgo_free() memory pooling syscalls. New testcase test_virgo_malloc for virgo_malloc(), virgo_set(), virgo_get(), virgo_free() has been added to repository. This testcase might have to be

```
updated if return type and args to virgo_malloc+ syscalls are to be changed.

Commits as on 25 September 2013
-------------------------------
All build related errors fixed after kernel rebuild some changes made to
function names to reflect their
names specific to memory pooling. Updated /etc/modules also has been committed
to repository.

Commits as on 26 September 2013
-------------------------------
Circular dependency error in standalone build of cpu pooling and memory pooling
drivers fixed and
datatypes and declarations for CPU pooling and Memory Pooling drivers have been
segregated into respective header files (virgo.h and
virgo_mempool.h with corresponding service header files) to avoid any dependency
error.

Commits as on 27 September 2013
-------------------------------
Major commits for Memory Pooling Driver listen port change and parsing VIRGO
memory pooling commands have been done.

Commits as on 30 September 2013
-------------------------------
New parser functions added for parameter parsing and initial testing on
virgo_malloc() works with telnet client with logs in test_logs/

Commits as on 1 October 2013
-----------------------------
Removed strcpy in virgo_malloc as ongoing bugfix for buffer truncation in
syscall path.

Commits as on 7 October 2013
----------------------------
Fixed the buffer truncation error from virgo_malloc syscall to mempool driver
service which was caused by
sizeof() for a char*. BUF_SIZE is now used for size in both syscall client and
mempool kernel service.

Commits as on 9 October 2013 and 10 October 2013
------------------------------------------------
Mempool driver kernelspace virgo mempool ops have been rewritten due to lack of
facilities to return a
value from kernel thread function. Since mempool service already spawns a
kthread, this seems to be sufficient. Also the iov.iov_len in virgo_malloc has
been changed from BUF_SIZE to strlen(buf) since BUF_SIZE
causes the kernel socket to block as it waits for more data to be sent.

Commits as on 11 October 2013
-----------------------------
sscanf format error for virgo_cloud_malloc() return pointer address and
sock_release() null pointer exception has been rectified.
Added str_to_addr() utility function.

Commits as on 14 October 2013 and 15 October 2013
-------------------------------------------------
Updated todo list.

Rewritten virgo_cloud_malloc() syscall with:
- mutexed virgo_cloud_malloc() loop
- redefined virgo address translation table in virgo_mempool.h
- str_to_addr(): removed (void**) cast due to null sscanf though it should have
worked
```

Commits as on 18 October 2013
------------------------------
Continued debugging of null sscanf - added str_to_addr2() which uses simple_strtoll() kernel function
for scanning pointer as long long from string and casting it to void*. Also more %p qualifiers where
added in str_to_addr() for debugging.

Based on latest test_virgo_malloc run, simple_strtoll() correctly parses the address string into a long long base 16 and then is reinterpret_cast to void*. Logs in test/

Commits as on 21 October 2013
-----------------------------
Kern.log for testing after vtranstable addr fix with simple_strtoll() added to repository and still the other %p qualifiers do not work and only simple_strtoll() parses the address correctly.

Commits as on 24 October 2013
-----------------------------
Lot of bugfixes made to virgo_malloc.c for scanning address into VIRGO transtable and size computation. Testcase test_virgo_malloc.c has also been modified to do reinterpret cast of long long into (struct virgo_address*) and corresponding test logs have been added to repository under virgo_malloc/test.

Though the above sys_virgo_malloc() works, the return value is a kernel pointer if the virgo_malloc executes in the Kernel mode which is more likely than User mode (call_usermodehelper which is circuitous). Moreover copy_from_user() or copy_to_user() may not be directly useful here as this is an address allocation routine. The long long reinterpret cast obfuscates the virgo_address(User or Kernel) as a large integer which is a unique id for the allocated memory on cloud. Initial testing of sys_virgo_set() causes a Kernel Panic as usual probably due to direct access of struct virgo_address*. Alternatives are to use only long long for allocation unique-id everywhere or do copy_to_user() or copy_from_user() of the address on a user supplied buffer. Also vtranstable can be made into a bucketed hash table that maps each alloc_id to a chained virgo malloc chunks than the present sequential addressing which is more similar to open addressing.

Commits as on 25 October 2013
----------------------------
virgo_malloc.c has been rewritten by adding a userspace __user pointer to virgo_get() and virgo_set() syscalls which are internally copied with copy_from_user() and copy_to_user() kernel function to get and set userspace from kernelspace.Header file syscalls.h has been updated with changed syscalls prototypes.Two functions have been added to map a VIRGO address to a unique virgo identifier and viceversa for abstracting hardware addresses from userspace as mentioned in previous commit notes. VIRGO cloud mempool kernelspace driver has been updated to use virgo_mempool_args* instead of void* and VIRGO cloudexec mempool driverhas been updated accordingly during intermodule invocation.The virgo_malloc syscall client has been updated to modified signatures and return types for all mempool alloc,get,set,free syscalls.

Commits as on 29 October 2013
-----------------------------
Miscellaneous ongoing bugfixes for virgo_set() syscall error in copy_from_user().

Commits as on 2 November 2013
-----------------------------
Due to an issue which corrupts the kernel memory, presently telnet path to VIRGO mempool driver has been
tested after commits on 31 October 2013 and 1 November 2013 and is working but

again there is an issue in kstrtoul() that returns the wrong address in
virgo_cloud_mempool_kernelspace.ko that gives the address for
data to set.

Commits as on 6 November 2013
-----------------------------
New parser function virgo_parse_integer() has been added to
virgo_cloud_mempool_kernelspace driver module which is carried over from
lib/kstrtox.c and modified locally to add an if clause to discard quotes and
unquotes. With this the telnet path commands for virgo_malloc()
and virgo_set() are working. Today's kern.log has been added to repository in
test_logs/.

Commits as on 7 November 2013
------------------------------
In addition to virgo_malloc and virgo_set, virgo_get is also working through
telnet path after today's commit for "virgodata:" prefix in
virgo_cloud_mempool_kernelspace.ko. This prefix is needed to differentiate data
and address so that toAddressString() can be invoked to sprintf() the address in
virgo_cloudexec_mempool.ko. Also mempool command parser has been updated to
strcmp() virgo_cloud_get command also.

Commits as on 11 November 2013
------------------------------
More testing done on telnet path for virgo_malloc, virgo_set and virgo_get
commands which work correctly. But there seem to be unrelated
kmem_cache_trace_alloc panics that follow each successful virgo command
execution. kern.log for this has been added to repository.

Commits as on 22 November 2013
------------------------------
More testing done on telnet path for virgo_malloc,virgo_set and virgo_set after
commenting kernel socket shutdown code in the VIRGO cloudexec
mempool sendto code. Kernel panics do not occur after commenting kernel socket
shutdown.

Commits as on 2 December 2013
-----------------------------
Lots of testing were done on telnet path and syscall path connection to VIRGO
mempool driver and screenshots for working telnet path (virgo_malloc, virgo_set
and virgo_get) have been committed to repository. Intriguingly, the syscall path
is suddenly witnessing series of broken pipe erros, blocking errors etc., which
are mostly Heisenbugs.

Commits as on 5 December 2013
------------------------------
More testing on system call path done for virgo_malloc(), virgo_set() and
virgo_get() system calls with test_virgo_malloc.c. All three syscalls work in
syscall path after lot of bugfixes. Kern.log that has logs for allocating memory
in remote cloud node with virgo_malloc, sets data "test_virgo_malloc_data" with
virgo_set and retrieves data with virgo_get.

VIRGO version 12.0 tagged.

Commits as on 12 March 2014
---------------------------
Initial VIRGO queueing driver implemented that flips between two internal
queues: 1) a native queue implemented locally and 2) wrapper around linux
kernel's workqueue facility 3) push_request() modified to pass on the request
data to the workqueue handler using container_of on a wrapper
structure virgo_workqueue_request.

Commits as on 20 March 2014

---------------------------
- VIRGO queue with additional boolean flags for its use as KingCobra queue
- KingCobra kernel space driver that is invoked by the VIRGO workqueue handler

Commits as on 30 March 2014
----------------------------
- VIRGO mempool driver has been augmented with use_as_kingcobra_service flags in CPU pooling and Memory pooling drivers

Commits as on 6 April 2014
--------------------------
- VIRGO mempool driver recvfrom() function's if clause for KingCobra has been updated for REQUEST header formatting mentioned in KingCobra design notes

Commits as on 7 April 2014
--------------------------
- generate_logical_timestamp() function has been implemented in VIRGO mempool driver that generates timestamps based on 3 boolean flags. At present machine_timestamp is generated and prepended to the request to be pushed to VIRGO queue driver and then serviced by KingCobra.

Commits as on 25 April 2014
---------------------------
- client ip address in VIRGO mempool recvfrom KingCobra if clause is converted to host byte order from network byte order with ntohl()

Commits as on 5 May 2014
-------------------------
- Telnet path commands for VIRGO cloud file system - virgo_cloud_open(), virgo_cloud_read(), virgo_cloud_write(), virgo_cloud_close() has been implemented and test logs have been added to repository (drivers/virgo/cloudfs/ and cloudfs/testlogs) and kernel upcall path for paramIsExecutable=0

Commits as on 7 May 2014
-------------------------
- Bugfixes to tokenization in kernel upcall plugin with strsep() for args passed on to the userspace

Commits as on 8 May 2014
------------------------
- Bugfixes to virgo_cloud_fs.c for kernel upcall (parameterIsExecutable=0) and with these the kernel to userspace upcall and writing to a file in userspace (virgofstest.txt) works. Logs and screenshots for this are added to repository in test_logs/

Commits as on 6 June 2014
-------------------------
- VIRGO File System Calls Path implementation has been committed. Lots of Linux Full Build compilation errors fixed and new integer parsing functionality added (similar to driver modules).  For the timebeing all syscalls invoke loadbalancer. This may be further optimized with a sticky flag to remember the first invocation which might be usually virgo_open syscall to get the VFS descriptor that is used in subsequent syscalls.

Commits as on 3 July 2014
--------------------------
- More testing and bugfixes for VIRGO File System syscalls have been done. virgo_write() causes kernel panic.

7 July 2014 - virgo_write() kernel panic notes:
---------------------------------------------
warning within http://lxr.free-electrons.com/source/arch/x86/kernel/smp.c#L121:

```
static void native_smp_send_reschedule(int cpu)
```

```
{
        if (unlikely(cpu_is_offline(cpu))) {
                WARN_ON(1);
                return;
        }
        apic->send_IPI_mask(cpumask_of(cpu), RESCHEDULE_VECTOR);
}
```

This is probably a fixed kernel bug in <3.7.8 but recurring in 3.7.8:
- http://lkml.iu.edu/hypermail/linux/kernel/1205.3/00653.html
- http://www.kernelhub.org/?p=3&msg=74473&body_id=72338
- http://lists.openwall.net/linux-kernel/2012/09/07/22
- https://bugzilla.kernel.org/show_bug.cgi?id=54331
- https://bbs.archlinux.org/viewtopic.php?id=156276

Commits as on 29 July 2014
---------------------------
All VIRGO drivers(cloudfs, queuing, cpupooling and memorypooling) have been built on 3.15.5 kernel with some Makefile changes for ccflags and paths

--------------------------------------------------------------------------------------
Commits as on 17 August 2014
--------------------------------------------------------------------------------------
(FEATURE - DONE) VIRGO Kernel Modules and System Calls major rewrite for 3.15.5 kernel - 17 August 2014
--------------------------------------------------------------------------------------
1. VIRGO config files have been split into /etc/virgo_client.conf and /etc/virgo_cloud.conf to delink the cloud client and kernel service config parameters reading and to do away with oft occurring symbol lookup errors and multiple definition errors for num_cloud_nodes and node_ip_addrs_in_cloud - these errors are frequent in 3.15.5 kernel than 3.7.8 kernel.

2. Each VIRGO module and system call now reads the config file independent of others - there is a read_virgo_config_<module>_<client_or_service>() function variant for each driver and system call. Though at present smacks of a replicated code, in future the config reads for each component (system call or module) might vary significantly depending on necessities.

3. New kernel module config has been added in drivers/virgo. This is for future prospective use as a config export driver that can be looked up by any other VIRGO module for config parameters.

4. include/linux/virgo_config.h has the declarations for all the config variables declared within each of the VIRGO kernel modules.

5. Config variables in each driver and system call have been named with prefix and suffix to differentiate the module and/or system call it serves.

6. In geographically distributed cloud virgo_client.conf has to be in client nodes and virgo_cloud.conf has to be in cloud nodes. For VIRGO Queue - KingCobra REQUEST-REPLY peer-to-peer messaging system same node can have virgo_client.conf and virgo_cloud.conf.

7. Above segregation largely simplifies the build process as each module and system call is independently built without need for a symbol to be exported from other module by pre-loading it.

8. VIRGO File system driver and system calls have been tested with above changes and the virgo_open(),virgo_read() and virgo_write() calls work with much less

```
crashes and freeze problems compared to 3.7.8 (some crashes in VIRGO FS syscalls
in 3.7.8 where already reported kernel bugs which seem to have been fixed in
3.15.5). Today's kern.log test logs have been committed to repository.

---------------------------------------------
Committed as on 23 August 2014
---------------------------------------------
Commenting use_as_kingcobra_service if clauses temporarily as disabling also
doesnot work and only commenting the block
works for VIRGO syscall path. Quite weird as to how this relates to the problem.
As this is a heisenbug further testing is
difficult and sufficient testing has been done with logs committed to
repository. Probably a runtime symbol lookup for kingcobra
causes the freeze.
For forwarding messages to KingCobra and VIRGO queues, cpupooling driver is
sufficient which also has the use_as_kingcobra_service clause.

---------------------------------------------
Committed as on 23 August 2014 and 24 August 2014
---------------------------------------------
As cpupooling driver has the same crash problem with kernel_accept() when
KingCobra has benn enabled, KingCobra clauses have been commented in both
cpupooling and memorypooling drivers. Instead queueing driver has been updated
with a kernel service infrastructure to accept connections at port 60000. With
this following paths are available for KingCobra requests:

      VIRGO cpupooling or memorypooling ====> VIRGO Queue =====> KingCobra

                              (or)
      VIRGO Queue kernel service ===========================> KingCobra

-----------------------------------------------
Committed as on 26 August 2014
-----------------------------------------------
- all kmallocs have been made into GFP_ATOMIC instead of GFP_KERNEL
- moved some kingcobra related header code before kernel_recvmsg()
- some header file changes for set_fs()

This code has been tested with modified code for KingCobra and the standalone
kernel service that accepts requests from telnet directly at port 60000, pushes
to virgo_queue
and is handled to invoke KingCobra servicerequest kernelspace function, works
(the kernel_recvmsg() crash was most probably due to Read-Only filesystem -errno
printed is -30)

---------------------------------------------------------------
VIRGO version 14.9.9 has been release tagged on 9 September 2014
---------------------------------------------------------------

------------------------------------------------------
Committed as on 26 November 2014
------------------------------------------------------
New kernel module cloudsync has been added to repository under drivers/virgo
that can be used for synchronization(lock() and unlock()) necessities in VIRGO
cloud. Presently Bakery Algorithm has been implemented.

------------------------------------------------------
Committed as on 27 November 2014
------------------------------------------------------
virgo_bakery.h bakery_lock() has been modified to take 2 parameters - thread_id
and number of for loops (1 or 2)

------------------------------------------------------
Committed as on 2 December 2014
```

--------------------------------------------------------
VIRGO bakery algorithm implementation has been rewritten with some bugfixes.
Sometimes there are soft lockup errors due to looping in kernel time durations
for which are kernel build configurable.

--------------------------------------------------------------------
Committed as on 17 December 2014
--------------------------------------------------------------------
Initial code commits for VIRGO EventNet kernel module service:
--------------------------------------------------------------
1.EventNet Kernel Service listens on port 20000

2.It receives eventnet log messages from VIRGO cloud nodes and writes the log
messages
after parsing into two text files /var/log/eventnet/EventNetEdges.txt and
/var/log/eventnet/EventNetVertices.txt by VFS calls

3.These text files can then be processed by the EventNet implementations in
AsFer (python pygraph and
C++ boost::graph based)

4.Two new directories virgo/utils and virgo/eventnet have been added.

5.virgo/eventnet has the new VIRGO EventNet kernel module service implementation
that listens on
port 20000.

6.virgo/utils is the new generic utilities driver that has a
virgo_eventnet_log()
exported function which connects to EventNet kernel service and sends the vertex
and edge eventnet
log messages which are parsed by kernel service and written to the two text
files above.

7.EventNet log messages have two formats:
   - Edge message - "eventnet_edgemsg#<id>#<from_event>#<to_event>"
   - Vertex message - "eventnet_vertextmsg#<id>-<partakers csv>-<partaker
conversations csv>"

8.The utilities driver Module.symvers have to be copied to any driver which are
then merged with the symbol files of the corresponding driver. Target clean has
to be commented while
building the unified Module.symvers because it erases symvers carried over
earlier.

9.virgo/utils driver can be populated with all necessary utility exported
functions that might be needed
in other VIRGO drivers.

10.Calls to virgo_eventnet_log() have to be #ifdef guarded as this is quite
network intensive.

------------------------------------------------------------------
Commits as on 18 December 2014
------------------------------------------------------------------
Miscellaneous bugfixes,logs and screenshot

- virgo_cloudexec_eventnet.c - eventnet messages parser errors and eventnet_func
bugs fixed
- virgo_cloud_eventnet_kernelspace.c - filp_open() args updated due to
vfs_write() kernel panics. The vertexmessage vfs_write is done after looping
through the vertice textfile and appending the conversation to the existing
vertex.Some more code has to be added.
- VIRGO EventNet build script updated for copying Module.symvers from utils

driver for merging with eventnet Module.symvers during Kbuild
- Other build generated sources and kernel objects
- new testlogs directory with screenshot for edgemsg sent to EventNet kernel service and kern.log with previous history for vfs_write() panics due to permissions and the logs for working filp_open() fixed version
- vertex message update

-----------------------------------------------------------------
Commits as on 2,3,4 January 2015
-----------------------------------------------------------------
- fixes for virgo eventnet vertex and edge message text file vfs_write() errors
- kern.logs and screenshots

-----------------------------------------------------------------
VIRGO version 15.1.8 release tagged on 8 January 2015
-----------------------------------------------------------------

--------------------------------------------------------------------------------------
Commits as on 3 March 2015 - Initial commits for Kernel Analytics Module which reads the /etc/virgo_kernel_analytics.conf config (and) VIRGO memorypooling Key-Value Store Architecture Diagram
---------------------------------------------------------------------------------------
- Architecture of Key-Value Store in memorypooling (virgo_malloc,virgo_get,virgo_set,virgo_free) has been
uploaded as a diagram at http://sourceforge.net/p/virgo-linux/code-0/HEAD/tree/trunk/virgo-docs/VIRGOLinuxKernel_KeyValueStore_and_Modules_Interaction.jpg

- new kernel_analytics driver for AsFer <=> VIRGO+USBmd+KingCobra interface has been added.
- virgo_kernel_analytics.conf having csv(s) of key-value pairs of analytics variables is set by AsFer or any other Machine Learning code.  With this VIRGO Linux Kernel is endowed with abilities to dynamically evolve than being just a platform for user code. Implications are huge - for example, a config variable "MaxNetworkBandwidth=255" set by the ML miner in userspace based on a Perceptron or Logistic Regression executed on network logs can be read by a kernel module that limits the network traffic to 255Mbps. Thus kernel dynamically changes behaviour.
- kernel_analytics Driver build script has been added

-------------------------------------------------------------------------
Commits as on 6 March 2015
-------------------------------------------------------------------------
- code has been added in VIRGO config module to import EXPORTed kernel_analytics config key-pair array
set by Apache Spark (mined from Uncomplicated Fire Wall logs) and manually and write to kern.log.

-------------------------------------------------------------------------
NeuronRain version 15.6.15 release tagged
-------------------------------------------------------------------------

-------------------------------------------------------------------------
Portability to linux kernel 4.0.5
-------------------------------------------------------------------------
The VIRGO kernel module drivers are based on kernel 3.15.5. With kernel 4.0.5 kernel which is the latest following
compilation and LD errors occur - this is on cloudfs VIRGO File System driver :
- msghdr has to be user_msghdr for iov and iov_len as there is a segregation of msghdr
- modules_install throws an error in scripts/Makefile.modinst while overwriting already installed module

```
-------------------------------------------------------------------------
Commits as on 9 July 2015
-------------------------------------------------------------------------
VIRGO cpupooling driver has been ported to linux kernel 4.0.5 with msghdr
changes as mentioned previously
with kern.log for VIRGO cpupooling driver invoked in parameterIsExecutable=2
(kernel module invocation)
added in testlogs

-------------------------------------------------------------------------
Commits as on 10,11 July 2015
-------------------------------------------------------------------------
VIRGO Kernel Modules:
- memorypooling
- cloudfs
- utils
- config
- kernel_analytics
- cloudsync
- eventnet
- queuing
along with cpupooling have been ported to Linux Kernel 4.0.5 - Makefile and
header files have been
updated wherever required.

-------------------------------------------------------------------------
Commits as on 20,21,22 July 2015
-------------------------------------------------------------------------
Due to SourceForge Storage Disaster(http://sourceforge.net/blog/sourceforge-
infrastructure-and-service-restoration/),
the github replica of VIRGO is urgently updated with some important changes for
msg_iter,iovec
etc., in 4.0.5 kernel port specifically for KingCobra and VIRGO Queueing. These
have to be committed to SourceForge Krishna_iResearch
repository at http://sourceforge.net/users/ka_shrinivaasan once SourceForge
repos are restored.
Time to move on to the manufacturing hub? GitHub ;-)
-------------------------------
VIRGO Queueing Kernel Module Linux Kernel 4.0.5 port:
------------------------------------------------------
- msg_iter is used instead of user_msghdr
- kvec changed to iovec
- Miscellaneous BUF_SIZE related changes
- kern.logs for these have been added to testlogs
- Module.symvers has been recreated with KingCobra Module.symvers from 4.0.5
KingCobra build
- clean target commented in build script as it wipes out Module.symvers
- updated .ko and .mod.c
-------------------------------
KingCobra Module Linux Kernel 4.0.5 port
------------------------------------------------------
- vfs_write() has a problem in 4.0.5
- the filp_open() args and flags which were working in 3.15.5 cause a
kernel panic implicitly and nothing was written to logs
- It took a very long time to figure out the reason to be vfs_write and
filp_open
- O_CREAT, O_RDWR and O_LARGEFILE cause the panic and only O_APPEND is working,
but
does not do vfs_write(). All other VIRGO Queue + KingCobra functionalities work
viz.,
enqueueing, workqueue handler invocation, dequeueing, invoking kingcobra
kernelspace service
request function from VIRGO queue handler, timestamp, timestamp and IP parser,
reply_to_publisher etc.,
- As mentioned in Greg Kroah Hartman's "Driving me nuts", persistence in Kernel
space is
a bad idea but still seems to be a necessary stuff - yet only vfs calls are used
which have to be safe
- Thus KingCobra has to be in-memory only in 4.0.5 if vfs_write() doesn't work
- Intriguingly cloudfs filesystems primitives - virgo_cloud_open,
virgo_cloud_read, virgo_cloud_write etc.,
work perfectly and append to a file.
- kern.logs for these have been added to testlogs
- Module.symvers has been recreated for 4.0.5
- updated .ko and .mod.c

----------------------------------------------------------------
Due to SourceForge outage and for a future code diversification
NeuronRain codebases (AsFer, USBmd, VIRGO, KingCobra)
in http://sourceforge.net/u/userid-769929/profile/ have been
replicated in GitHub also - https://github.com/shrinivaasanka
excluding some huge logs due to Large File Errors in GitHub.
----------------------------------------------------------------

---------------------------------------------------------------------
Commits as on 30 July 2015
---------------------------------------------------------------------
VIRGO system calls have been ported to Linux Kernel 4.0.5 with commented gcc
option -Wimplicit-function-declaration,
msghdr and iovec changes similar to drivers mentioned in previous commit notes
above. But Kernel 4.1.3 has some Makefile and build issues.
The NeuronRain codebases in SourceForge and GitHub would henceforth be mostly
and always out-of-sync and not guaranteed to be replicas - might get diversified
into different research and development directions (e.g one codebase might be
more focussed on IoT while the other towards enterprise bigdata analytics
integration with kernel and training which is yet to be designed- depend on lot
of constraints)

---------------------------------------------------------------------
Commits as on 2,3 August 2015
---------------------------------------------------------------------
- new .config file added which is created from menuconfig
- drivers/Kconfig has been updated with 4.0.5 drivers/Kconfig for trace event
linker errors
Linux Kernel 4.0.5 - KConfig is drivers/ has been updated to resolve RAS driver
trace event linker error. RAS was not included in KConfig earlier.
- link-vmlinux.sh has been replaced with 4.0.5 kernel version

---------------------------------------------------------------------
Commits as on 12 August 2015
---------------------------------------------------------------------
VIRGO Linux Kernel 4.1.5 port - related code changes - some important notes:
---------------------------------------------------------------------
- Linux Kernel 4.0.5 build suddenly had a serious root shell drop error in
initramfs which was not resolved by:
      - adding rootdelay in grub
      - disabling uuid for block devices in grub config
        - mounting in read/write mode in recovery mode
        - no /dev/mapper related errors
        - repeated exits in root shell
      - delay before mount of root device in initrd scripts
- mysteriously there were some firmware microcode bundle executions in
ieucodetool
- Above showed a serious grub corruption or /boot MBR bug or 4.0.5 VIRGO kernel
build problem
- Linux 4.0.x kernels are EOL-ed
- Hence VIRGO is ported to 4.1.5 kernel released few days ago
- Only minimum files have been changed as in commit log for Makefiles and
syscall table and headers and a build script has been added
for 4.1.5:
    Changed paths:
    A buildscript_4.1.5.sh
    M linux-kernel-extensions/Makefile
    M linux-kernel-extensions/arch/x86/syscalls/Makefile
    M linux-kernel-extensions/arch/x86/syscalls/syscall_32.tbl
    M linux-kernel-extensions/drivers/Makefile
    M linux-kernel-extensions/include/linux/syscalls.h

- Above minimum changes were enough to build an overlay-ed Linux Kernel with
VIRGO codebase

----------------------------------------------------------------------
Commits as on 14,15,16 August 2015
----------------------------------------------------------------------
Executed the minimum end-end telnet path primitives in Linux kernel 4.1.5 VIRGO
code:
- cpu virtualization
- memory virtualization
- filesystem virtualization (updated filp_open flags)
and committed logs and screenshots for the above.

----------------------------------------------------------------------
Commits as on 17 August 2015
----------------------------------------------------------------------
VIRGO queue driver:
- Rebuilt Module.symvers
- kern.log for telnet request to VIRGO Queue + KingCobra queueing system in
kernelspace

----------------------------------------------------------------------
Commits as on 25,26 September 2015
----------------------------------------------------------------------
VIRGO Linux Kernel 4.1.5 - memory system calls:
----------------------------------------------
- updated testcases and added logs for syscalls invoked
separately(malloc,set,get,free)
- The often observed unpredictable heisen kernel panics occur with 4.1.5 kernel
too. The logs are 2.3G and
only grepped output is committed to repository.
- virgo_malloc.c has been updated with kstrdup() to copy the buf to iov.iov_base
which was earlier
crashing in copy_from_iter() within tcp code. This problem did not happen in
3.15.5 kernel.
- But virgo_clone syscall code works without any changes to iov_base as above
which does a strcpy()
 which is an internal memcpy() though. So what causes this crash in memory
system calls alone
is a mystery.
- new insmod script has been added to load the VIRGO memory modules as necessary
instead of at boot time.
- test_virgo_malloc.c and its Makefile has been updated.

VIRGO Linux Kernel 4.1.5 - filesystem calls- testcases and logs:
------------------------------------------------------------
  - added insmod script for VIRGO filesystem drivers
  - test_virgo_filesystem.c has been updated for syscall numbers in 4.1.5 VIRGO
kernel
  - virgo_fs.c syscalls code has been updated for iov.iov_base kstrdup() -
without this there are kernel panics in copy_from_iter(). kern.log
testlogs have been added, but there are heisen kernel panics. The virgo syscalls
are executed but not written to kern.log due to these crashes.
```

Thus execution logs are missing for VIRGO filesystem syscalls.

----------------------------------------------------------------------
Commits as on 28,29 September 2015
----------------------------------------------------------------------

VIRGO Linux Kernel 4.1.5 filesystem syscalls:
--------------------------------------------
- Rewrote iov_base code with a separate iovbuf set to iov_base and strcpy()-ing the syscall command to iov_base similar to VIRGO
memory syscalls
- Pleasantly the same iovbuf code that crashes in memory syscalls works for VIRGO FS without crash.Thus both virgo_clone and virgo_filesystem
syscalls work without issues in 4.1.5 and virgo_malloc() works erratically in 4.1.5 which remains as issue.
- kern.log for VIRGO FS syscalls and virgofstest text file written by virgo_write() have been added to repository

VIRGO Linux 4.1.5 kernel memory syscalls:
------------------------------------
- rewrote the iov_base buffer code for all VIRGO memory syscalls by allocating separate iovbuf and copying the message to it - this just replicates the virgo_clone syscall behaviour which works without any crashes mysteriously.
- did extensive repetitive tests that were frequented by numerous kernel panics and crashes
- The stability of syscalls code with 3.15.5 kernel appears to be completely absent in 4.1.5
- The telnet path works relatively better though
- Difference between virgo_clone and virgo_malloc syscalls despite having same kernel sockets code looks like a non-trivial bug and a kernel stability issue.
- kernel OOPS traces are quite erratic.
- Makefile path in testcase has been updated

----------------------------------------------------------------------
Commits as on 4 October 2015
----------------------------------------------------------------------
VIRGO Linux Kernel 4.1.5 - Memory System Calls:
-----------------------------------------------
- replaced copy_to_user() with a memcpy()
- updated the testcase with an example VUID hardcoded.
- str_to_addr2() is done on iov_base instead of buf which was causing NULL parsing
- kern.log with above resolutions and multiple VIRGO memory syscalls tests - malloc,get,set
- With above VIRGO malloc and set syscalls work relatively causing less number of random kernel panics
- return values of memory calls set to 0
- in virgo_get() syscall, memcpy() of iov_base is done to data_out userspace pointer
- kern.log with working logs for syscalls - virgo_malloc(), virgo_set(), virgo_get() but still there are random kernel panics
- Abridged kern.log for VIRGO Memory System Calls with 4.1.5 Kernel - shows example logs for virgo_malloc(), virgo_set() and virgo_get()

----------------------------------------------------------------
Commits as on 14 October 2015
----------------------------------------------------------------
VIRGO Queue Workqueue handler usermode clause has been updated with 4.1.5 kernel paths and kingcobra in user mode is enabled for invoking KingCobra Cloud Perfect Forwarding.

---------------------------------------------------------------
Commits as on 15 October 2015

----------------------------------------------------------------
- Updated VIRGO Queue kernel binaries and build generated sources
- virgo_queue.h has been modified for call_usermodehelper() - set_ds() and fd_install() have been uncommented for output redirection. Output redirection works but there are "alloc_fd: slot 1 not NULL!" errors at random (kern.log in kingcobra testlogs) which seems to be a new feature in 4.1.5 kernel. This did not happen in 3.7.8-3.15.5 kernels

------------------------------------------------------------------
Commits as on 3 November 2015
------------------------------------------------------------------
- kern.log for VIRGO kernel_analytics+config drivers which export the analytics variables from /etc/virgo_kernel_analytics.conf kernel-wide and print them in config driver has been added to config/testlogs

------------------------------------------------------------------
Commits as on 10 January 2016
------------------------------------------------------------------
NeuronRain VIRGO enterprise version 2016.1.10 released.

---------------------------------------------------------------------------------
NeuronRain - AsFer commits for VIRGO - C++ and C Python extensions
- Commits as on 29 January 2016
---------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------
(FEATURE - DONE)  Python-C++-VIRGOKernel and Python-C-VIRGOKernel boost::python and cpython implementations:
------------------------------------------------------------------------------------------------------------
- It is a known idiom that Linux Kernel and C++ are not compatible.
- In this commit an important feature to invoke VIRGO Linux Kernel from userspace python libraries via two alternatives have been added.
- In one alternative, C++ boost::python extensions have been added to encapsulate access to VIRGO memory system calls - virgo_malloc(), virgo_set(), virgo_get(), virgo_free(). Initial testing reveals that C++ and Kernel are not too incompatible and all the VIRGO memory system calls work well though initially there were some errors because of config issues.
- In the other alternative, C Python extensions have been added that replicate boost::python extensions above in C - C Python with Linux kernel
works exceedingly well compared to boost::python.
- This functionality is required when there is a need to set kernel analytics configuration variables learnt by AsFer Machine Learning Code
dynamically without re-reading /etc/virgo_kernel_analytics.conf.
- This completes a major integration step of NeuronRain suite - request travel roundtrip to-and-fro top level machine-learning C++/python
code and rock-bottom C linux kernel - bull tamed ;-).
- This kind of python access to device drivers is available for Graphics Drivers already on linux (GPIO - for accessing device states)
- logs for both C++ and C paths have been added in cpp_boost_python_extensions/ and cpython_extensions.
- top level python scripts to access VIRGO kernel system calls have been added in both directories:

        CPython - python cpython_extensions/asferpythonextensions.py
        C++ Boost::Python - python
cpp_boost_python_extensions/asferpythonextensions.py
- .so, .o files with build commandlines(asferpythonextensions.build.out) for "python setup.py build" have been added
in build lib and temp directories.
- main implementations for C++ and C are in
cpp_boost_python_extensions/asferpythonextensions.cpp and
cpython_extensions/asferpythonextensions.c

```
--------------------------------------------------------------------------------
---------
Commits as on 12 February 2016
--------------------------------------------------------------------------------
---------
Commits for Telnet/System Call Interface to VIRGO CPUPooling -> VIRGO Queue ->
KingCobra
--------------------------------------------------------------------------------
---------
*) This was commented earlier for the past few years due to a serious kernel
panic in previous kernel versions - <= 3.15.5
*) In 4.1.5 a deadlock between VIRGO CPUPooling and VIRGO queue driver init was
causing following error in "use_as_kingcobra_service" clause :
        - "gave up waiting for virgo_queue init, unknown symbol push_request()"
*) To address this a new boolean flag to selectively enable and disable VIRGO
Queue kernel service mode "virgo_queue_reactor_service_mode" has been added.
*) With this flag VIRGO Queue is both a kernel service driver and a standalone
exporter of function symbols - push_request/pop_request
*) Incoming request data from telnet/virgo_clone() system call into cpupooling
kernel service reactor pattern (virgo cpupooling listener loop) is treated as
generic string and handed over to VIRGO queue and KingCobra which publishes it.
*) This resolves a long standing deadlock above between VIRGO cpupooling
"use_as_kingcobra_service" clause and VIRGO queue init.
*) This makes virgo_clone() systemcall/telnet both synchronous and asynchronous
- requests from telnet client/virgo_clone() system call can be either
synchronous RPC functions executed on a remote cloud node in kernelspace (or) an
asynchronous invocation through "use_as_kingcobra_service" clause path to VIRGO
Queue driver which enqueues the data in kernel workqueue and subsequently popped
by KingCobra.
*) Above saves an additional code implementation for virgo_queue syscall paths -
virgo_clone() handles, based on config selected, incoming data passed to it
either as a remote procedure call or as a data that is pushed to VIRGO
Queue/KingCobra pub-sub kernelspace.
--------------------------------------------------------------------------------
---------
Prerequisites:
--------------
- insmod kingcobra_main_kernelspace.ko
- insmod virgo_queue.ko compiled with flag virgo_queue_reactor_service_mode=1
      (when virgo_queue_reactor_service_mode=0, listens on port 60000 for direct
telnet requests)
- insmod virgo_cloud_test_kernelspace.ko
- insmod virgo_cloudexec.ko (listens on port 10000)

--------------------------------------------------------------------------------
---------
Schematic Diagram
--------------------------------------------------------------------------------
---------
VIRGO clone system call/telnet client ---> VIRGO cpupooling(compiled with
use_as_kingcobra_service=1) ------> VIRGO Queue kernel service (compiled with
virgo_queue_reactor_service_mode=1) ---> Linux Workqueue handler ------>
KingCobra

--------------------------------------------------------------------------------
---------
Commits as on 15 February 2016 - Kernel Analytics - VIRGO Linux Kernelwide
imports
--------------------------------------------------------------------------------
---------
- Imported Kernel Analytics variables into CloudFS kernel module - printed in
driver init()
- Module.symvers from kernel_analytics has been merged with CloudFS
```

Module.symvers
- Logs for above has been added in cloudfs/test_logs/
- Makefile updated with correct fs path
- Copyleft notices updated

--------------------------------------------------------------------------------------------
Commits as on 15 February 2016 - Kernel Analytics - VIRGO Linux Kernelwide imports
--------------------------------------------------------------------------------------------
- Kernel Analytics driver exported variables have been imported in CPU virtualization driver
- Module.symvers from kernel_analytics has been merged with Module.symvers in cpupooling
- kern.log for this import added to cpupooling/virgocloudexec/test_logs/

--------------------------------------------------------------------------------------------
Commits as on 15 February 2016 - Kernel Analytics - VIRGO Linux Kernelwide imports
--------------------------------------------------------------------------------------------
- Imported kernel analytics variables into memory virtualization driver init() , exported from kernel_analytics driver
- build shell script updated
- logs added to test_logs/
- Module.symvers from kernel_analytics has been merged with memory driver Module.symvers
- Makefile updated

--------------------------------------------------------------------------------------------
Commits as on 15 February 2016 - Kernel Analytics - VIRGO Linux Kernelwide imports
--------------------------------------------------------------------------------------------
- Imported kernel analytics variables into VIRGO Queueing Driver
- logs for this added in test_logs/
- Makefile updated
- Module.symvers from kernel_analytics has been merged with Queueing driver's Module.symvers
- .ko, .o and build generated sources

--------------------------------------------------------------------------------
Commits as on 16,17 February 2016
--------------------------------------------------------------------------------
(FEATURE-DONE) Socket Buffer Debug Utility Function - uses linux skbuff facility
--------------------------------------------------------------------------------
- In this commit a multipurpose socket buffer debug utility function has been added in utils driver and exported kernelwide.
- It takes a socket as function argument does the following:
      - dereference the socket buffer head of skbuff per-socket transmit data queue
      - allocate skbuff with alloc_skb()
      - reserve head room with skb_reserve()
      - get a pointer to data payload with skb_put()
      - memcpy() an example const char* to skbuff data
      - Iterate through the linked list of skbuff queue in socket and print headroom and data pointers
      - This can be used as a packet sniffer anywhere within VIRGO linux network stack
- Any skb_*() functions can be plugged-in here as deemed necessary.
- kern.log(s) which print the socket internal skbuff data have been added to a

```
new testlogs/ directory
- .cmd files generated by kbuild

--------------------------------------------------------------------------------
(FEATURE-DONE) Commits as on 24 February 2016
--------------------------------------------------------------------------------
skbuff debug function in utils/ driver:
(*) Added an if clause to check NULLity of skbuff headroom before doing
skb_alloc()
(*) kern.log for this commit has been added testlogs/
(*) Rebuilt kernel objects and sources

--------------------------------------------------------------------------------
Commits as on 29 February 2016
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-----
(FEATURE-DONE) Software Analytics - SATURN Program Analysis added to VIRGO Linux
kernel drivers
--------------------------------------------------------------------------------
-----
- SATURN (saturn.stanford.edu) Program Analysis and Verification software has
been
integrated into VIRGO Kernel as a Verification+SoftwareAnalytics subsystem
- A sample driver that can invoke an exported function has been added in drivers
- saturn_program_analysis
- Detailed document for an example null pointer analysis usecase has been
created in virgo-docs/VIRGO_SATURN_Program_Analysis_Integration.txt
- linux-kernel-
extensions/drivers/virgo/saturn_program_analysis/saturn_program_analysis_trees/e
rror.txt is the error report from SATURN
- SATURN generated preproc and trees are in linux-kernel-
extensions/drivers/virgo/saturn_program_analysis/preproc and
linux-kernel-
extensions/drivers/virgo/saturn_program_analysis/saturn_program_analysis_trees/

--------------------------------------------------------------------------------
Commits as on 10 March 2016
--------------------------------------------------------------------------------
- SATURN analysis databases (.db) for locking, memory and CFG analysis.
- DOT and PNG files for locking, memory and CFG analysis.
- new folder saturn_calypso_files/ has been added in saturn_program_analysis/
with new .clp files virgosaturncfg.clp and virgosaturnmemory.clp
- SATURN alias analysis .db files

--------------------------------------------------------------------------------
-----------------------------------------------------------
(FEATURE-DONE) NEURONRAIN - ASFER Commits for VIRGO - CloudFS systems calls
integrated into Boost::Python C++ and Python CAPI invocations
--------------------------------------------------------------------------------
-----------------------------------------------------------
--------------------------------------------------------------------------------
------------------------------------------
AsFer Commits as on 30 May 2016
--------------------------------------------------------------------------------
------------------------------------------
VIRGO CloudFS system calls have been added (invoked by unique number from
syscall_32.tbl) for C++ Boost::Python interface to
VIRGO Linux System Calls. Switch clause with a boolean flag has been introduced
to select either VIRGO memory or filesystem calls.
kern.log and CloudFS textfile Logs for VIRGO memory and filesystem invocations
from AsFer python have been committed to testlogs/

--------------------------------------------------------------------------------
```

------------------------------------------
AsFer Commits as on 31 May 2016
--------------------------------------------------------------------------------
------------------------------------------
Python CAPI interface to NEURONRAIN VIRGO Linux System Calls has been updated to include File System open, read, write primitives also.
Rebuilt extension binaries, kern.logs and example appended text file have been committed to testlogs/. This is exactly similar to
commits done for Boost::Python C++ interface. Switch clause has been added to select memory or filesystem VIRGO syscalls.

Srinivasan Kannan (alias) Ka.Shrinivaasan (alias) Shrinivas Kannan
http://sites.google.com/site/kuja27

USBmd driver is an experimental modified version of already existing USB driver in linux.

Purpose of this modified version is for doing more sophisticated debugging of USB endpoints and devices and as
USB packet sniffer. Technical Necessity for this was created due to prolonged data theft, id spoofing and cybercrime that has been happening

in author's personal electronic devices for years that resulted in a Cybercrime Police Complaint also few years ago.

There were also such incidents while developing open source code (some code commits have description of these mysterious occurrences). There is no comprehensive USB debugger available on linux to sift bad traffic though there are strong evidences of such cybercrime and datatheft through other sources. Author is inclined to believe that such recurring events of datatheft that defies all logic can have no other intent but to cause malafide theft or loss of private data and an act of defamation among other things.

This is also done as a technical learning exercise to analyze USB Hosts, packets and USB's interaction,if any, with wireless devices including
mobiles, wireless LANs(radiotap) etc.,

In the longterm USBmd might have to be integrated into VIRGO. As VIRGO would would have the synergy of AstroInfer machine learning
codebase for "learning" from datasets, this USBmd driver can have the added ability of analyzing large USB traffic (as a dataset)
using some decision making algorithms and evolve as an anti-cybercrime, anti-plagiarism and anti-theft tool to single out
"malevolent" traffic that would save individuals and organisations from the travails of tampering and loss of sensitive confidential data.

The pattern mining of numeric dataset designed for AstroInfer can apply here also since USB bitstream can be analyzed using algorithms for
numerical dataset mining. Also Discrete Fourier Transform used for analyzing data for frequencies (periodicities if any) can be used for
USB data , for example USB wireless traffic.

======================================================
new UMB driver bind - 27 Feb 2014 (for Bus id 7)
======================================================
Following example commandlines install umb.ko module, unbind the existing option driver from bus-device id and bind the umb.ko to that bus id:

```
sudo insmod umb.ko
echo -n "7-1:1.0" > /sys/bus/usb/drivers/option/unbind
echo -n "7-1:1.0" > /sys/bus/usb/drivers/umb/bind
```

======================================================
Commits as on 29 July 2014
======================================================
Driver has been ported and built on 3.15.5 kernel. Also a driver build script has been committed.

------------------------------------------------------
USBmd version 14.9.9 has been release tagged on 9 September 2014
------------------------------------------------------
------------------------------------------------------
USBmd version 15.1.8 has been release tagged on 8 January 2015
------------------------------------------------------

http://sourceforge.net/p/usb-md/code-0/HEAD/tree/Adding%20new%20vendor%20and
%20product%20IDs%20to%20an%20existing%20USB%20driver%20on%20Linux.html has steps to add new vendor-id.

-------------------------------------------------------------------------------
USB debug messages from "cat /sys/kernel/debug/usb/devices" for UMB bound above:
-------------------------------------------------------------------------------

```
T:  Bus=07 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#= 12 Spd=12   MxCh= 0
D:  Ver= 1.10 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs=  1
P:  Vendor=12d1 ProdID=140b Rev= 0.00
```

```
S:  Manufacturer=HUAÿWEI TECHNOLOGIES
S:  Product=HUAWEI Mobile
S:  SerialNumber=ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
C:* #Ifs= 4 Cfg#= 1 Atr=a0 MxPwr=500mA
I:* If#= 0 Alt= 0 #EPs= 3 Cls=ff(vend.) Sub=ff Prot=ff Driver=umb
E:  Ad=81(I) Atr=03(Int.) MxPS=  16 Ivl=128ms
E:  Ad=82(I) Atr=02(Bulk) MxPS=  64 Ivl=0ms
E:  Ad=02(O) Atr=02(Bulk) MxPS=  64 Ivl=0ms
I:* If#= 1 Alt= 0 #EPs= 2 Cls=ff(vend.) Sub=ff Prot=ff Driver=option
E:  Ad=84(I) Atr=02(Bulk) MxPS=  64 Ivl=0ms
E:  Ad=04(O) Atr=02(Bulk) MxPS=  64 Ivl=0ms
I:* If#= 2 Alt= 0 #EPs= 2 Cls=ff(vend.) Sub=ff Prot=ff Driver=option
E:  Ad=86(I) Atr=02(Bulk) MxPS=  64 Ivl=0ms
E:  Ad=06(O) Atr=02(Bulk) MxPS=  64 Ivl=0ms
I:* If#= 3 Alt= 0 #EPs= 2 Cls=08(stor.) Sub=06 Prot=50 Driver=usb-storage
E:  Ad=87(I) Atr=02(Bulk) MxPS=  64 Ivl=0ms
E:  Ad=08(O) Atr=02(Bulk) MxPS=  64 Ivl=0ms
```

```
---------------------------------------------------------------------
usbmon, libpcap tcpdump and wireshark (or vusb-analyzer) debugging
---------------------------------------------------------------------
*mount -t debugfs none_debugs /sys/kernel/debug
*modprobe usbmon
*ls /sys/kernel/debug/usb/usbmon/

0s  0u      1s  1t      1u  2s      2t  2u      3s  3t      3u  4s      4t  4u
      5s  5t      5u  6s      6t  6u      7s  7t      7u  8s      8t  8u

*cat /sys/kernel/debug/usb/usbmon/8t > usbmon.mon (any of the above usbmon debug
logs)
*vusb-analyzer usbmon.mon

ef728540 3811287714 S Ci:001:00 s a3 00 0000 0001 0004 4 <
ef728540 3811287743 C Ci:001:00 0 4 = 00010000
ef728540 3811287752 S Ci:001:00 s a3 00 0000 0002 0004 4 <
ef728540 3811287763 C Ci:001:00 0 4 = 00010000
f50f6540 3811287770 S Ii:001:01 -115 2 <
f50f6540 3811287853 C Ii:001:01 -2 0
f5390540 3814543695 S Ci:001:00 s a3 00 0000 0001 0004 4 <
f5390540 3814543715 C Ci:001:00 0 4 = 00010000
f5390540 3814543756 S Ci:001:00 s a3 00 0000 0002 0004 4 <
f5390540 3814543767 C Ci:001:00 0 4 = 00010000
f50f6540 3814543805 S Ii:001:01 -115 2 <

*modprobe usbmon
*ls /dev/usbmon[1-8]
*tcpdump -i usbmon1 -w usbmon.pcap
tcpdump: listening on usbmon1, link-type USB_LINUX_MMAPPED (USB with padded
Linux header), capture size 65535 bytes
^C86 packets captured
86 packets received by filter

*wireshark usbmon.pcap (loads on wireshark)

----------------------------------------------------------
Dynamic Debug - dev_dbg() and dev_vdbg()
----------------------------------------------------------

USB Debugging References:
-------------------------
- Texas Instruments -
http://elinux.org/images/1/17/USB_Debugging_and_Profiling_Techniques.pdf

---------------------------------------------------------
```

```
NeuronRain version 15.6.15 release tagged
---------------------------------------------------------

---------------------------------------------------------
Commits as on 11 July 2015
---------------------------------------------------------
usbmd kernel module has been ported to Linux Kernel 4.0.5

---------------------------------------------------------
Commits as on 26 November 2015
---------------------------------------------------------
- Updated USB-md driver with a lookup of VIRGO kernel_analytics config variable
exported by kernel_analytics module in umb_read() as default.
- New header file umb.h has been added that externs the VIRGO kernel_analytics
config array variables
- Module.symvers has been imported from VIRGO kernel_analytics and clean target
has been commented in build script after initial build as make clean removes
Module.symvers.
- kern.log with umb_read() and umb_write() have been added with following
commandlines:
      - cat /dev/umb0 - invokes umb_read() but there are kernel panics sometimes
      - cat <file> > /dev/umb0 - invokes umb_write()
  where umb0 is usb-md device name registered with /sys/bus/usb as below:
      - insmod umb.ko
      - echo -n "7-1:1.0" > /sys/bus/usb/drivers/option/unbind
      - echo -n "7-1:1.0" > /sys/bus/usb/drivers/umb/bind
- Updated build generated sources and object files have been added

----------------------------------------------------------
Commits as on 27 November 2015
----------------------------------------------------------
New folder usb_wwan_modified has been added that contains the USB serial, option
and wireless USB modem WWAN drivers from kernel mainline
instrumented with lot of printk()s so that log messages are written to kern.log.
Though dev_dbg dynamic debugging can be used by writing to
/sys/kernel/debug/<...>/dynamic_debug
printk()s are sufficient for now. This traces through the USB connect and data
transfer code:
        - probe
        - buffer is copied from userspace to kernelspace
        - URB is allocated in kernel
        - buffer is memcopied to URB
        - usb send/receive bulk pipe calls
        - usb_fill_bulk_urb
Almost all buffers like in and out buffers in URBs, portdata, interfacedata,
serial_data, serial_port_data are printed to kern.log. This log is
analyzable by AsFer machine learning code for USB debugging similar to usbmon
logs.

These are initial commits only and usb-serial.c, usb_wwan.c, option.c and
serial.h might be significantly altered going forward.

----------------------------------------------------------
Commits as on 30 November 2015
----------------------------------------------------------
Added usb.h from kernel mainline, instrumented with printk() to print
transfer_buffer in usb_fill_[control/bulk/interrupt]_urb() functions. kern.log
for this has been added in usb_wwan_modified/testlogs.

----------------------------------------------------------
Commits as on 1 December 2015
----------------------------------------------------------
- new kernel function print_buffer() has been added in usb.h that prints
contents of char buffer in hex
```

- Above print_buffer() is invoked to print transfer_buffer in usb_wwan.c, usb-serial.c, option.c
- kern.log with print_buffer() output has been added - This dumps similar to wireshark, usbmon and other usb analyzers.

----------------------------------------------------------
Commits as on 2 December 2015
----------------------------------------------------------
- changed print_buffer() printk() to print a delimiter in each byte for AsFer Machine Learning code processing
- add a parser script for kern.log to print print_buffer() lines
- parsed kern.log with print_buffer() lines has been added
- Added an Apache Spark MapReduce python script to compute byte frequency in parsed print_buffer() kern.log

------------------------------------------------------------------
(ONGOING) NeuronRain USBmd Debug and Malafide Traffic Analytics
------------------------------------------------------------------
As mentioned in commit notes above, USB incoming and outgoing data transfer_buffer are dumped byte-by-byte. Given this data various analytics can be performed most of which are already implemented in AsFer codebase:
- frequency of bytes
- most frequent sequence of bytes
- bayesian and decision tree inference
- deep learning
- perceptrons
- streaming algorithms for USB data stream
and so on.

------------------------------------------------------------------
Commits as on 3 December 2015
------------------------------------------------------------------
- Apache Spark script for analyzing the USBWWAN byte stream logs has been updated with byte counts map-reduce functions from print_buffer() logs and temp DataFrame Table creation with SparkSQL.
- logs for the script have been added in usb_wwan_modified/python-src/testlogs/Spark_USBWWANLogMapReduceParser.out.3December2015
- kern.log parser shellscript has been updated

-------------------------------------------------------------------
AsFer commits for USBmd as on 4 December 2015
-------------------------------------------------------------------
All the Streaming_<>.py Streaming Algorithm implementations in AsFer/python-src/ have been updated with:
- hashlib ripemd160 hash MD algorithm for hash functions and return hexdigest()
- USBWWAN byte stream data from USBmd print_buffer() logs in usb-md/usb_wwan_modified/testlogs/ has been added as a Data Storage and Data Source
- logs for the above have been added to asfer/python-src/testlogs/
- Streaming Abstract Generator has been updated with USB stream data iterable and parametrized for data source and storage
- Some corrections to the asfer/python-src/Streaming_<> scripts

-------------------------------------------------------------------
Commits as on 7 December 2015
-------------------------------------------------------------------
- added Spark Mapreduce and DataFrame log for USBWWAN byte stream
- added a parsed kern.log with only bytes from USBWWAN stream
- Added dict() and sort() for query results and printed cardinality of the stream data set which is the size of the dict.
An example log has been added which prints the cardinality as ~250. In contrast, LogLog and HyperLogLog counter estimations
approximate the cardinality to 140 and 110 respectively

--------------------------------------------------------------------------------------
AsFer commits for USBmd as on 11 December 2015 - USBWWAN stream data backend in MongoDB
--------------------------------------------------------------------------------------
 Dependency Injection code commits for MongoDB backend - With this MongoDB is also a storage backend for AsFer algorithms similar to MySQL:
- Abstract_DBBackend.py has been updated for both MySQL and MongoDB injections
- MongoDB configuration and backend connect/query code has been added. Backend is either populated by Robomongo or pymongo reading from the
Streaming Abstract Generator iterable framework.
- With this AsFer supports both SQL(MySQL) and NoSQL(file,hive,hbase,cassandra backends in Streaming Abstract Generator).
- log with a simple NoSQL table with StreamingData.txt and USBWWAN data has been added to testlogs/.
- MongoDB configuration has a database(asfer-database) and a collection(asfer-collection).
- MongoDB_DBBackend @provides pymongo.collection.Collection which is @inject-ed to Abstract_DBBackend

--------------------------------------------------------------------------------------
Commits as on 10 January 2016
--------------------------------------------------------------------------------------
NeuronRain USBmd enterprise version 2016.1.10 released.

----------------------------------------------------------------------------------------
KingCobra - A Research Software for Distributed Request Service on Cloud with Arbiters

--------------------------------------------------------------------------------------------------
Copyright (C):
Srinivasan Kannan (alias) Ka.Shrinivaasan (alias) Shrinivas Kannan
Independent Open Source Developer, Researcher and Consultant
Ph: 9789346927, 9003082186, 9791165980
Open Source Products Profile(Krishna iResearch):
http://sourceforge.net/users/ka_shrinivaasan,
https://www.openhub.net/accounts/ka_shrinivaasan
Personal website(research): https://sites.google.com/site/kuja27/
emails: ka.shrinivaasan@gmail.com, shrinivas.kannan@gmail.com,
kashrinivaasan@live.com
--------------------------------------------------------------------------------------------------
*****************************************************************************************/

Theoretical Interludes, Design Notes and ToDo (long term with no deadline)

--------------------------------------------------------------------------

[This is a major research oriented extension of iCloud(AsFer+USBmd+VIRGO) and COBRA project (done by the author in his BE (1995-1999) along with few other esteemed classmates.
http://sourceforge.net/projects/acadpdrafts/files/Excerpts_Of_PSG_BE_FinalProjec
t_COBRA_done_in_1999.pdf/download.)]

[KingCobra though a misnomer is expanded as ClOud With ARBiters MimicKING containing the anagram]

1. (THEORY) There is a cloud of nodes which execute a set of services from randomly created clients.

2. (THEORY) This cloud could be on iCloud (AsFer+USBmd+VIRGO) platform or any other opensource cloud platforms like Hadoop Cluster.

3. (THEORY) The Clients are publishers of Service requests which are of many types -  miscellaneous types of Service that could be dynamically added through other kernel modules and invoked through a switch-case or embedded in function itself. Identified by unique id(s) for different types of services (for example Problem reports, Suggestions etc.,)

4. (THEORY) The Services on the Cloud are Subscribers to these requests of specific type. Thus this is the conventional publisher-subscriber model.

5. (THEORY) The requests flow through cloud using a workqueue (which could be a lowlevel Linux workqueue or VIRGO queue or some other queuing middleware software like ActiveMQ). The publishers enqueue and Subscribers dequeue the requests.

6. (THEORY) The difference is that the Cloud has nodes that "deceive" or "corrupt".

7. (THEORY) Service requests - are published by the clients in the need of a service which could be defined by markup file. These requests are scheduled and routed by the middleware to competent authority which services it (with or without timeframe) and replies to the client.

8. (THEORY) Problem reports - are published by clients which are "dissatisfied" by the quality of service by the cloud. These are analyzed by "arbiters" in the cloud which find the faulting node(s) and take action. This allows manual intervention but minimizes it.

9. (THEORY) Suggestions - are enhancement requests sent by clients and require manual intervention.

10. (THEORY) Cloud nodes have a Quality of Service metric calculated by a model.

11. (THEORY) The cloud has a reporting structure of nodes - either as a graph or tree. The graph is dynamically reorganized by weighting the Quality of Service of each node.

12. (THEORY) The difficult part of the above is using Arbiters to find "faulty" nodes based on problem reports from clients.

13. (THEORY) Brewer's CAP conjecture proved by [GilbertLynch] as a theorem (still debated) states that only 2 of the 3 (Consistency of data, Avaliability of data and Partition tolerance when some nodes or messages are lost) can be guaranteed and not all 3 are simultaneously achievable.

14. (THEORY) CAP theorem does not seem to apply to the above faulty scenario with corrupt nodes under Consistency or Availablity or Partition Tolerance. This is because a corrupt node can have any 2 of the 3 - it can give consistent data,

is available with success response or can make the cloud work with missing data in partition tolerance but yet can "corrupt" the cloud. Probably this needs to be defined as a new attribute called Integrity.

15. (THEORY) As "corruption" is more conspicuous with monetary element, if above services are "charged" with a logical currency (e.g. bitcoin), then corruption in cloud is defineable approximately as (but not limited to)- "Undue favour or harm meted out to a client not commensurate with the charge for the service (or) unreasonable extra logical currency demanded to execute the service of same quality (or) deliberate obstruction of justice to a client with malevolent and unholy collusion with other cloud nodes with feigned CAP".

16. (THEORY) Identifying criminal nodes as in (15) above seems to be beyond the ambit of CAP. Thus CAP with Integrity further places a theoretical limit on "pure" cloud. If Integrity is viewed as a Byzantine problem with faulty or corrupt processes in a distributed system, and if resilience factor is rf (expected number of faulty nodes), then most algorithms can ensure a "working" cloud only if resilience is ~30% or less (3*rf+1) of the total number of cloud nodes. Probably this could apply to Integrity also that places a limit of 30% on "corrupt nodes" for the Cloud to work with sanity. Translating this to a Governance problem, a corruption-free administration is achievable with a maximum limit of 30% "corrupt" elements.

17. (THEORY-ONGOING) Analytics on the Problem reports sent to the cloud queue give a pattern of corrupt nodes. Intrinsic Merit ranking with Citation graph maxflow considering cloud as a flow network where a node positively or negatively cites or "opines" about a node, as mentioned in http://arxiv.org/abs/1006.4458 (author's Master's thesis) and http://www.nist.gov/tac/publications/2010/participant.papers/CMI_IIT.proceedings .pdf(published by the author during PhD) give a p2p ranking of cloud nodes that can be used for analysis though may not be reliable fully. AsFer has bigdata analytics functionality that fits well to this point to analyse the problem reports with machine learning algorithms and set the key-value pairs that are read by VIRGO kernel_analytics module and exported kernelwide. The persisted REQUEST_REPLY.queue with the logged request-reply IPs and timestamps can be mined with AsFer bigdata capability (e.g. Spark)

18. (THEORY) Policing the cloud nodes with arbiters - This seems to be limited by CAP theorem and Integrity as above. Also this is reducible to perfect inference problem in http://sourceforge.net/p/asfer/code/HEAD/tree/AstroInferDesign.txt and drafts in https://sites.google.com/site/kuja27/

19. (THEORY) Brooks-Iyengar algorithm for sensors in all cloud nodes is an improved Byazantine Fault Tolerant algorithm.

20. (THEORY) BitCoin is a Byzantine Fault Tolerant protocol.

21. (THEORY) Byzantine Fault Tolerance in Clouds is described in http://www.computer.org/csdl/proceedings/cloud/2011/4460/00/4460a444-abs.html, http://www.eurecom.fr/~vukolic/ByzantineEmpire.pdf which is more on Cloud of Clouds - Intercloud with cloud nodes that have malicious or corrupt software. Most of the key-value(get/set) implementations do not have byzantine nodes (for example CAP without Byzantine nodes in Amazon Dynamo: http://www.eurecom.fr/~michiard/teaching/slides/clouds/cap-dynamo.pdf)

22. (THEORY) Related to point 18 - The problem of fact finding or fault finding using a cloud police has the same limitation as the "perfect inference" described in http://sourceforge.net/p/asfer/code/HEAD/tree/AstroInferDesign.txt. "Money trail" involving the suspect node in point 28 is important to conclude something about the corruption. In real world tracking money trail is a daunting task. In cloud that abides by CAP, missing messages on trail can prevent reaching a conclusion thereby creating benefit-of-doubt. Also fixing exact value for a transaction that involves MAC currency message is undecidable - a

normative economics problem can never be solved by exact theoretical computer science.

23. (THEORY) Reference article on cloud BFT for Byzantine, Corrupt brokers - Byzantine Fault-Tolerant Publish/Subscribe: A Cloud Computing Infrastructure (www.ux.uis.no/~meling/papers/2012-bftps-srdsw.pdf)

---------------------------------------------------------------
24. KingCobra messaging request-response design - options
---------------------------------------------------------------

24a. Implementing a message subscription model in kernelspace where clients publish the message that is queued-in to subscribers' queue (Topic like implementation - use of ActiveMQ C implementation if available).

24b. (DONE-minimal implementation) At present a minimum kernelspace messaging system that queues remote request and handles through workqueue handler is in place. This responds to the client once the Kingcobra servicerequest function finishes processing the request(reply_to_publisher() in KingCobra driver). Unlike the usual messaging server, in which client publishes messages of a particular type that are listened to by interested clients, one option is to continue the status-quo of KingCobra as a peer-to-peer messaging system. Thus every VIRGO node is both a kernelspace messaging client and server that can both publish and listen. Every message in the cloud can have a universally unique id assigned by a timestamp server - https://tools.ietf.org/html/rfc4122 (similar to bitcoin protocol) so that each message floating in the cloud is unique across the cloud (or) no two messages on the VIRGO cloud are same. The recipient node executing kingcobra_servicerequest_kernelspace() parses the unique-id (example naive unique-id is <ip-address:port>#localtimestampofmachine which is a simplified version of RFC4122) from the incoming remote request and responds to the remote client through kernel socket connection  that gets queued-in the remote client and handled similar to incoming remote request. To differentiate request and response-for-request response messages are padded with a string "REPLY:<unique-id-of-message>" and requests are padded with "REQUEST:<unique-id-of-message>". This is more or less similar to TCP flow-control with SEQ numbers but state-less like UDP. Simple analogy is post-office protocol with reference numbers for each mail and its reply. Thus there are chronologically two queues: (1) queue at the remote VIRGO cloud service node for request (2) queue at the remote client for response to the request sent in (1). Thus any cloudnode can have two types of messages - REQUEST and REPLY. Following schematic diagram has been implemented so far.

--------------------------------------------------------------------------------------------------
24c.(DONE) KingCobra - VIRGO queue - VIRGO cpupooling , mempooling and queue service drivers interaction schematic diagram:
--------------------------------------------------------------------------------------------------

```
      KingCobraClient ==========>=<REQUEST:id>===================> VIRGO
cpupooling service =====> VIRGO Queue ============> KingCobraService
               ||
                          ||
             ||
                   ||
            <================= VIRGO Queue <====== VIRGO cpupooling service
====<REPLY:id>=================================== V

      KingCobraClient ==========>=<REQUEST:id>===================> VIRGO
mempooling service =====> VIRGO Queue ============> KingCobraService
               ||
                          ||
             ||
                  ||
            <================== VIRGO Queue <====== VIRGO mempooling service
====<REPLY:id>==================================== V

      KingCobraClient ==========>=<REQUEST:id>===================> VIRGO Queue
service ====================================> KingCobraService
               ||
                           ||
             ||
                  ||
            <================== VIRGO Queue service
============================<REPLY:id>==================================== V
```

24d. (ONGOING) kingcobra_servicerequest_kernelspace() distinguishes the "REQUEST" and "REPLY" and optionally persists them to corresponding on-disk filesystem.  Thus a disk persistence for the queued messages can either be implemented in 1) VIRGO queue driver 2)  workqueue.c (kernel itself needs a rewrite (or) 3) KingCobra driver. Option (2) is difficult in the sense that it could impact the kernel as-a-whole whereas 1) and 3) are modularized. At present Option 3 persistence within KingCobra driver has been implemented.

24e.Above option 24b implements a simple p2p queue messaging in kernel. To get a Topic-like behaviour in VIRGO queue might be difficult as queue_work() kernel function has to be repeatedly invoked for the same work_struct on multiple queues which are subscribers of that message in AMQ protocol. Moreover creating a queue at runtime on need basis looks difficult in kernel which is usually done through some CLI or GUI interface in ActiveMQ and other messaging servers.

25. (ONGOING) For the timestamp service, EventNet described in http://sourceforge.net/p/asfer/code/HEAD/tree/AstroInferDesign.txt is a likely implementation choice. AsFer already has a primitive text files based EventNet graph implementation in place. Periodic topological sort (quite expensive) of EventNet gives logical ordering and thus a logical timestamp of the cloud events.

26. (THEORY - ONGOING Implementation) MESSAGE-AS-CURRENCY PROTOCOL: If each message payload is also construed as a currency carrier, each message id can be mapped to a unique currency or a coin with fixed denomination. This is similar to each currency note having a serial number as unique id. Uniqueness is guaranteed since there can be only one message (or coin) with that id on the cloud. This simulates a scenaio - "Sender of the message pays the Receiver with a coin having the unique id and Receiver acknowledges receipt". This is an alternative to BitCoin protocol. Double spending is also prohibited since at any point in time the message or "coin" with unique id can be sent by only one node in the cloud. Unique Cloudwide Timestamp server mimicks the functionality of "Mint". There is a difference here between conventional send-receive of messages - Once a message is sent to remote cloud node, no copy of it should exist anywhere in the cloud. That is, every MAC currency message is a cloudwide singleton. In pseudocode this is expressible as:

```
      m1=MAC_alloc(denomination)
      m2=m1 (---- this is disallowed)
```

Linux kernel allocation functions - kmalloc() - have a krefs functionality for reference counting within kernel. Refcount for MAC message can never exceed 1 across cloud for above singleton functionality - this has to be a clause everywhere for any unique MAC id. This requires a cloudwide krefs rather. Buyer decrements cloudwide kref and Seller increments it. In C++ this is done by std::move() and often required in "Perfect Forwarding" - http://thbecker.net/articles/rvalue_references/section_07.html - within single addressspace. By overloading operator=() with Type&& rvalue reference, the necessary networking code can be invoked that does the move which might include serialization. But unfortunately C++ and Linux kernel are not compatible. The Currency object has to be language neutral and thus Google Protocol Buffers which have C,C++,Java, Python .proto files compilers support might be useful but yet the move semantics in Kernel/C is non-trivial that requires cloudwide transactional kernel memory as mentioned in (31) below.  A C++ standalone userspace client-server cloud object move implementation based on std::move() over network of Protocol Buffer Currency Objects has been added to AsFer repository at - http://sourceforge.net/p/asfer/code/HEAD/tree/cpp-src/cloud_move which can optionally be upcall-ed to userspace from VIRGO and KingCobra drivers. This C++ implementation is invoked in userspace with call_usermodehelper() from VIRGO Queue Messaging via the kernel workqueue handler.

26.1 Schematic Diagram for Cloud Perfect Forwarding with AsFer+VIRGOQueue+KingCobraUserspace:

```
--------------------------------------------------------------------------------
-------------

    Telnet or other client ===========> VIRGO Queue Service Listener ======>
VIRGO Workqueue Handler
                                                           |
                                                           |  [KernelSpace]
                                                           V
      AsFer Cloud Perfect Forwarding Client <===== KingCobra Userspace shell
script(call_usermodehelper)
                  |                                        |
                  |                                        |
  Virtual Currency      |                                        |  [UserSpace]
                    V                                      V
        AsFer Cloud Perfect Forwarding Server
<=================================/
```

References:

----------

26.2 An example distributed transactional memory implementation in cloud - http://infinispan.org/tutorials/simple/tx/ and http://www.cloudtm.eu/ - these are in userspace cloud (C++ and Java) and may not have cloud move functionality - move has to be simulated in a transaction: replicate data, delete in one endpoint and create in other endpoint

27. (THEORY) SIMULATING A VIRTUAL ECONOMY with above MAC protocol (Message-as-currency): If each message sent is considered as "money" element and cloud nodes and clients are the consumers and producers of "electronic money", the timestamp "Mint" becomes a virtual Federal Reserve or Central Bank that controls the "electronic money" circulation in the cloud. Infact any REPLY messages could be mapped to a Service a client derives by s(p)ending the REQUEST "money message". Thus value(REQUEST) should equal value(REPLY) where value() is a function that measures the value of a money denomination and the value of goods and/or services for that money. For example Rs.10000 or $10000 has no meaning if it doesn't translate into a value (analogy: erstwhile Gold Standard).When the value() function gets skewed phenomena like Inflation arise. Thus above model could also have a notion of value() and "electronic money inflation". Thus any "message money" with a unique id assigned by the cloud unique id(or logical timestamp) server can exist at most in only one node in the cloud. Money trail can be implemented by prefixing a header to the incoming message money in each cloud node that receives the money which traces the "path" taken. Cloud has to implement some Byzantine Fault Tolerant protocol. The value() function to some extent can measure the "deceit" as above. When a Buyer and Seller's value() functions are at loggerheads then that is starting point of "cloud corruption" at either side and might be an undecidable problem.

28. (THEORY) TRADING WITH ABOVE KINGCOBRA MAC protocol - somewhat oversimplified:

```
                        --------------------
                        |Unique MAC id MINT|
                  -------------------
                        ||
      -----money trail------------------
      |
      V
      ....
      Buyer ======= sends MAC message (REQUEST id) =======> Seller (stores the
MAC in local cash reserve and prepends money trail)
      ||                                       ||
      <============ sends the goods and services (REPLY id) ===
```

In the above schematic, money with unique id in cloud reaches a buyer after many buyer-seller transitions called "money trail". The MAC currency is prefixed by each node to create a chain. Buyer then sends a request to the seller through MAC virtual currency and seller replies with goods and services. Seller prepends the money trail chain. When a transaction occurs the whole cloud need not be notified about it except only buyer and seller. MAC Mint could create a bulk of money denominations and circulate them in cloud economy.

29. (THEORY) VALUE FOR ELECTRONIC MONEY: How is above MAC money earned - This again requires linking value to money (as money is not a value by itself and only a pointer to valuable item or resource). Thus any buyer can "earn" MAC money by something similar to a barter.

30. (THEORY) FIXING VALUE FOR MAC MONEY: To delineate corruption as discussed in 27 above with value() disparity between MAC money and REPLY goods and services, an arbiter node in cloud has to "judge" the value of MAC sent by buyer and goods and services from seller and proclaim if corruption exists. Thus value() function itself has to be some kind of machine learning algorithm. This is related to or same as points 12 to 23 above. For example, while buying an item for few million bucks, value() has to take as input the description of the item and calculate the value "ideally" which is difficult. Because there are no perfect references to evaluate and only a weighted average of available market price range has to be taken as a reference which is error-prone. value() function can be recursively defined as("Reductionism"):

```
      -------------------------------------------------------------------------
---------------------------
      |value(i) = summation(value(ingredients of i)) + cost(integrating the
ingredients to create item i) |
      -------------------------------------------------------------------------
---------------------------
```

Obviously the above recursion combinatorially explodes into exponential number of nodes in the recursion tree. Ideally recursion has to go deep upto quarks and leptons that makeup the standard model. If for practical purposes, recursion depth is restricted to t then size of value() tree is O(m^t) where m is average number of ingredients per component. Hence any algorithm computing the value() recursion has to be exponential in time. Computation of value() in the leaf nodes of the recursion is most crucial as they percolate bottom-up. If leaf nodes of all possible items are same (like quarks and leptons making up universe) then such atomic ingredient has to have "same" value for all items. Only the integration cost varies in the levels of the tree.For infinite case, value() function is conjectured to be undecidable - probably invoking some halting problem reduction .But above value() function could be Fixed Parameter Tractable in parameter recursion depth - t but yet could only be an approximation. A Turing machine computing value() function exactly might loop forever and thus Recursively Enumerable and not Recursive.

31. (THEORY) Buyer-Seller and MAC electronic money transaction schematic:
-------------------------------------------------------------------------

```
      Buyer        A-------<id><refcnt:0>---------------------> Seller
<id><refcnt:1> (increments refcnt)
      (<id><refcnt:1>  |
       <id><refcnt:0>  |
      after decrement  |
      refcnt           |
      )---------------->
```
Above has to be transactional (i.e atomic across cloud nodes)

32. (THEORY) MAC protocol reaper
--------------------------------
Reaper thread in each cloud node harvests the zero refcounted allocations and invokes destructors on them. Same  MAC id cannot have kref count of 1 or above in more than one cloud node due to the transaction mentioned previously.

33. (THEORY) Cloud Policing With Arbiters - Revisited:
------------------------------------------------------
When a suspect node is analyzed when a complaint problem is filed on it, (1) it is of foremost importance on how flawless is the arbiter who investigates on that and is there a perfect way to choose a perfect arbiter. In the absence of the previous credibility of entire cloud judiciary is blown to smithereens and falls apart. (2) Assuming a perfect arbiter which is questionable, next thing is to analyze the credibility of the node who sulked. This is nothing but the Citation problem in http://arxiv.org/abs/1106.4102 and http://www.nist.gov/tac/publications/2010/participant.papers/CMI_IIT.proceedings.pdf where a node can positively or negatively cite another node a "Societal Norm" which can be faulted and citations/opinions could be concocted with malafide intent(perjury). This is rather a generalization of PageRank algorithm with negative citations. Thus Perfect Cloud Arbitration could be an unsolvable problem. (3) Even if both arbiter and complainant are perfect which is again questionable, there are still loopholes - lack of evidences or implicated witnesses might portray a negative impression of a positive node. Thus there are 3 tiers of weakenings in cloud arbitration and there could be more. P(Good) series in https://sites.google.com/site/kuja27/ precisely addresses this problem.

34. (THEORY) MAC Money Flow as MaxFlow problem:
--------------------------------------------
Transactions happening in a cloud are edges between the nodes involved (buyer and seller). Thus it creates a huge directed graph. Flow of money in this graph can be modelled as Flow network. Minimum Cut of this graph shows crucial nodes in the graph which play vital role in cloud economy removal of which paralyzes the cloud (could be Central bank and other financial institutions). This graph has bidirectional edges where one direction is for money and the opposite direction is for Goods and Services. In the Flow network sum of flows is zero. But in the Money Flow Network each node is having a cash reserve ratio (CRR) due to commercial transactions which is confidential and privy to that node only and thus sum of flows can not be zero. Hub nodes in the Money Flow graph which can be obtained by getting k-core or D-core of the graph by some graph peeling algorithms are crucial nodes to the economy that contribute to Money circulation.

35. (THEORY) Cycles and components in above MAC Money Flow Graph:
---------------------------------------------------------------
Above graph of money transactions could be cyclic which implies a supply chain. Strongly connected components of this graph are most related nodes that are in same industry.

36. (THEORY) STOCK TRADING:
---------------------------
One of the component in above MAC Money Flow Graph of cloud could be a virtual Stock Exchange. Based on the financial and securities transactions of constituent organizations in the graph, index of the exchange varies.

37. (THEORY) Analysis of Poverty and Alleviation through above money flow graph:
--------------------------------------------------------------------------------
Weights of the edges of money flow graph are the denominations of the transaction. Thus high value edges and low value edges divide the Graph

logically into Rich and Poor strata(Bourgeoisie and Proletariat subgraphs). Equitable graph is the one which does not have too much of value difference between Rich and Poor sets of edges - a utopian to achieve. Mathematically, it is an optimization LP problem that seeks to minimize sum(RichEdges)-sum(PoorEdges) or Sum(RichVertices) - sum(PoorVertices) - without harming either - to be precise. This requires money flow to be programmed to find a feasible solution to this LP subject to constraints like work-pay parity etc.,(there could be more variables and constraints to this LP) . Due to CRR above Vertices also can be Rich and Poor in addition to Rich Edges and Poor Edges.

38.(THEORY) Demand and Supply and Value() function:
------------------------------------------
Alternative to the recursive definition of value() function above can be done through Demand and Supply - more the demand and less the supply, price increases and vice-versa. This is quite subjective compared to absolute recursive definition above. To simulate demand and supply, the weights of the money edges (-> direction) in the bidirectional graph change and fluctuate dynamically over time for unchanging weights of the Goods and Services edges ( <- direction) between any pair of Buyer-Seller vertices. This makes Money and G&S Flow graph a Dynamic Graph with edge weight update primitive.

39.(THEORY) Hidden or Colored Money:
-------------------------------------
In an ideal Cloud with only MAC currencies, colored money can co-exist if (not limited to) some money trails are missing, due to "cloud corruption", systemic failure, hardware and network issues etc.,. Probably this is the direct consequence of CAP theorem and can be conjectured to be undecidable. Hidden money is to some extent dependent on quantity of net flow (if non-zero) and how much of this net flow is contributed by Rich vertices and Edges.

-----------------------------------------------
Commits as on 1 March 2014
-----------------------------------------------
Example java Publisher and Listeners that use ActiveMQ as the messaging middleware have been committed to repository for an ActiveMQ queue instance created for KingCobra. For multiple clients this might have to be a Topic rather than Queue instance. Request types above and a workflow framework can be added on this. This will be a JMS compliant implementation which might slow down compared to a linux workqueue or queue implementation being done in VIRGO.
----------------------------------------------
Commits as on 17 March 2014
---------------------------------------------
KingCobra userspace library and kernelspace driver module have been implemented that are invoked 1) either in usermode by call_usermodehelper()
2) or through intermodule invocation through exported symbols in KingCobra kernel module, by the workqueue handler in VIRGO workqueue implementation.

---------------------------------------------
Commits as on 22 March 2014
---------------------------------------------
Minimalistic Kernelspace messaging server framework with kernel workqueue,handler and remote cloud client has been completed - For this VIRGO clone cpupooling driver has been added a clause based on a boolean flag, to direct incoming request from remote client to VIRGO linux workqueue which is popped by workqueue handler that invokes a servicerequest function on the KingCobra kernel module. (Build notes: To remove any build or symbol errors, Module.symvers from VIRGO queue has to be copied to VIRGO cloudexec and built to get a unified VIRGO cloudexec Module.symvers that has exported symbol definitions for push_request()). End-to-end test with telnet path client sending a request to VIRGO cloudexec service, that gets queued in kernel workqueue, handled by workqueue handler that finally invokes KingCobra service request function has been done and the kern.log has been added to repository at drivers/virgo/queuing/test_logs/

----------------------------------------------
Commits as on 29 March 2014
----------------------------------------------
Initial commits for KingCobra Request Response done by adding 2 new functions parse_ip_address() and reply_to_publisher() in kingcobra_servicerequest_kernelspace()

--------------------------------------------
Commits as on 30 March 2014
--------------------------------------------
Both VIRGO cpupooling and mempooling drivers have been modified with use_as_kingcobra_service boolean flag for sending incoming remote cloud node requests to VIRGO queue which is serviced by workqueue handler and KingCobra service as per the above schematic diagram and replied to.

--------------------------------------------
Commits as on 6 April 2014
--------------------------------------------
Fixes for REQUEST and REPLY headers for KingCobra has been made in virgo_cloudexec_mempool recvfrom() if clause and in request parser in KingCobra with strsep(). This has been implemented only in VIRGO mempool codepath and not in VIRGO clone.

-------------------------------------------
Commits as on 7 April 2014
-------------------------------------------
New function parse_timestamp() has been added to retrieve the timestamp set by the VIRGO mempool driver before pushing the request to VIRGO queue driver

------------------------------------------
Commits as on 29 April 2014
------------------------------------------
Intial commits for disk persistence of KingCobra request-reply queue messages have been done with addition of new boolean flag kingcobra_disk_persistence. VFS calls are used to open and write to the queue.

------------------------------------------
Commits as on 26 August 2014
------------------------------------------
KingCobra driver has been ported to 3.15.5 kernel and bugs related to a kernel_recvmsg() crash, timestamp  parsing etc., have been fixed. The random crashes were most likely due to incorrect parameters to filp_open() of disk persistence file and filesystem being mounted as read-only.

---------------------------------------------------
Version 14.9.9 release tagged on 9 September 2014
--------------------------------------------------
---------------------------------------------------
Version 15.1.8 release tagged on 8 January 2015
--------------------------------------------------

-------------------------------------------
Commits as on 17 August 2015
-------------------------------------------
KingCobra + VIRGO Queuing port of Linux Kernel 4.1.5 :
 - changed the REQUEST_REPLY.queue disk persisted queue path to /var/log/kingcobra/REQUEST_REPLY.queue
 - kernel built sources, object files
 - kern.log with logs for telnet request sent to VIRGO queue driver, queued in kernel work queue and handler invocation for the
KingCobra service request kernel function for the popped request; disk persisted /var/log/kingcobra/REQUEST_REPLY.queue

--------------------------------------------

Commits as on 14 October 2015
---------------------------------------------
AsFer Cloud Perfect Forwarding binaries are invoked through
call_usermodehelper() in VIRGO queue. KingCobra commands has been updated with a clause for cloud perfect forwarding.

---------------------------------------------
Commits as on 15 October 2015
---------------------------------------------
- Updated KingCobra module binaries and build generated sources
- kingcobra_usermode_log.txt with "not found" error from output redirection (kingcobra_commands.c). This error is due to need for absolute path. But there are "alloc_fd: slot 1 not NULL!" after fd_install() is uncommented in virgo_queue.h call_usermodehelper() code. The kern.log with these errors has been added to testlogs
- kingcobra_commands.c has been changed to invoke absolute path executable. With uncommenting of fd_install and set_ds code in virgo_queue the return code of call_usermodehelper() is 0 indicating successful invocation

---------------------------------------------------------------------------
Commits as on 10 January 2016
---------------------------------------------------------------------------
NeuronRain KingCobra enterprise version 2016.1.10 released.

------------------------------------------------------------------------------------------------
NEURONRAIN VIRGO Commits for virgo_clone()/telnet -> VIRGO cpupooling -> VIRGO Queue -> KingCobra
- as on 12 February 2016
------------------------------------------------------------------------------------------------
VIRGO commit:
https://github.com/shrinivaasanka/virgo-linux-github-code/commit/72d9cfc90855719542cdb62ce40b798cc7431b3d

Commit comments:
-----------------------------------------------------------------------------------------
Commits for Telnet/System Call Interface to VIRGO CPUPooling -> VIRGO Queue -> KingCobra
-----------------------------------------------------------------------------------------
*) This was commented earlier for the past few years due to a serious kernel panic in previous kernel versions - <= 3.15.5
*) In 4.1.5 a deadlock between VIRGO CPUPooling and VIRGO queue driver init was causing following error in "use_as_kingcobra_service" clause :
  - "gave up waiting for virgo_queue init, unknown symbol push_request()"
*) To address this a new boolean flag to selectively enable and disable VIRGO Queue kernel service mode "virgo_queue_reactor_service_mode" has been added.
*) With this flag VIRGO Queue is both a kernel service driver and a standalone exporter of function symbols - push_request/pop_request
*) Incoming request data from telnet/virgo_clone() system call into cpupooling kernel service reactor pattern (virgo cpupooling listener loop) is treated as generic string and handed over to VIRGO queue and KingCobra which publishes it.
*) This resolves a long standing deadlock above between VIRGO cpupooling "use_as_kingcobra_service" clause and VIRGO queue init.
*) This makes virgo_clone() systemcall/telnet both synchronous and asynchronous - requests from telnet client/virgo_clone() system call can be
either synchronous RPC functions executed on a remote cloud node in kernelspace (or) an asynchronous invocation through "use_as_kingcobra_service"
 clause path to VIRGO Queue driver which enqueues the data in kernel workqueue and subsequently popped by KingCobra.
*) Above saves an additional code implementation for virgo_queue syscall paths - virgo_clone() handles, based on config selected, incoming

data passed to it either as a remote procedure call or as a data that is pushed to VIRGO Queue/KingCobra pub-sub kernelspace
*) Kernel Logs and REQUEST_REPLY.queue for above commits have been added to kingcobra c-src/testlogs/