Aspect Architecture Document

OSA Infrastructure – Programming Guidelines

Document: 45016-00

Revision: A.3

Revision Date: 12/01/2009

Author:	Satish Sangapu
Creation Date:	08/30/2009

Change Sheet

Rev.	Date	Section	Change
A.1	08/31/2009	All	Initial Revision – used for internal review
A.2	09/27/2009	1.2, 1.3	Introduced Related Documents and Future Topics sub-sections
		2.4	Clarified how DPL runs its tasks
		3	Additional guidelines for OS abstractions
		4	Added Licensing Guidelines
		5	Split the section into user-space and kernel-space guidelines
A.3	12/04/2009	1	Created the Aspect Overview section for this AAD
		2.1, 2.2	Moved some of the introduction and terminology content into the Aspect Overview
		2.3	Re-hashed the operating system abstraction section
		2.4	Latest enhancements to the Licensing Guidelines section
		2.5.1	Modification to User Space Guidelines
		2.7	Added a new section on Lock Management Guidelines
		2.8	Added a new section on Object Graph Population Guidelines for non-DPL VMs

Table of Contents

List of Figures

List of Tables

1. ASPECT ARCHITECTURE DOCUMENT

1.1 Aspect Introduction

1.1.1 Aspect Description

This document or aspect contains the infrastructure material to develop and productize OSA (Open Storage Architecture) solutions. More specifically, it contains two aspect elements: 1) OSA Development and Build Environment 2) OSA Programming Guidelines.

The OSA environment contains four virtual machines and hypervisor-specific code. This code as well as the operating system images must be maintained, version controlled and built. OSA Development and Build Environment element addresses this by documenting the source code management system, build environment, compiler details, and packaging tools/process for the OSA environment.

OSA Programming Guidelines documents programming principles for developing applications for the OSA environment, more specifically for the Linux virtual machines. It begins by comparing the operating system differences between DPL's VxWorks and Linux, such as scheduling entities and mechanisms. Later, it covers details about operating system abstractions, specific Linux constructs, lock management guidelines, and object population guidelines for non-DPL VMs. OSA Programming Guidelines element complements the existing Coding Standards Document.

1.1.2 Assumptions

1.1.3 Related Documents

Document Number	Name of Document
44682-00	OSA Development and Build Environment FAM (PR 200012473)
http://cfwweb.lsi.com/docgen/docgen_view.php?docid=23	Coding Standards Document

Table 1: Related Documents

1.1.4 Open Issues

1.2 Aspect High Level Requirements

1.2.1 Product Requirements

ClearQuest PR Number	Feature Name
LSIP200012473	USPV1: Tools and Build Infrastructure for OSA Products

Figure 1: Product Requirements

1.2.2 Architectural Requirements

1.3 Terminology

Multi-threading - technique that allows a program or application to execute multiple threads concurrently. Multi-threading can provide many benefits for applications: (1) Good runtime concurrency (2) Parallel programming (multi-core SMP) can be implemented more easily (3) Performance gains and reduced resource consumption (compared to multi-process).

1.4 Future Considerations

Programming for the Windows environment should be considered in the future.

2. OSA PROGRAMMING GUIDELINES

2.1 Introduction

2.1.1 Purpose/Goals

Currently, all the controller firmware developers comprehend the non-pre-emptive multi-tasking model that exists with Data Protection Layer (DPL) in VxWorks. In addition, it is known that the VxWorks kernel is very lightweight and normally does not interfere with embedded RAID programming. Both of these environments will change for the Open Storage Architecture (OSA) Linux virtual machines.

Therefore, this document is meant to aid development in an Open Storage Architecture environment that features one or more Linux virtual machines. It will present high-level differences between the current environment and the new Linux environment. Guidelines such as these must be documented and propagated to the development organization or else, the code-base will eventually lead to incompatible implementations resulting in unstable and un-maintainable system.

The initial sections in the document will give an overview of key operating system concepts, especially ones specific to the Linux environment. It will attempt to compare/contrast Linux, with its pre-emptive task scheduling model with a distinct kernel and user space separation, to that of VxWorks. Finally, the document will conclude with programming guidelines for the CFW organization, catered to the embedded storage environment, to develop on the Linux virtual machines in OSA.

Detailed information about each of the topics can be referenced online or in a textbook; however, note that some of the online resources are stale and in some situations, completely inaccurate.

This document is meant to be refined as the design and architecture matures for the Linux virtual machines, especially during the early phases of development. Therefore, please note items that should be covered as part of this document as that material can be covered in future versions.

2.1.2 Future Considerations

Future topics that *can* be covered as part of this document:

1) Initial high-level component dependencies for Domain0

2) SOD sequence for components in Domain0. (ODP, ODC, DOMI ported over for Linux VMs, RAS/IPIVM, SOD component, initial Hypervisor related activities) [Perhaps this doesn't make sense for this type of document….]

3) There will be many scripts on Domain0 and Service VM. This document may need to be evolved to cover scripting best practices and possibly recommended scripting languages.

4) Evaluate ACE (Adaptive Communication Environment) to understand how it can used in this environment. It could be used to abstract the OS dependencies as well as use tried and true patterns for various functionalities, such as concurrency, IPC, and memory management.

Resolution: Brief information has been documented in section 2.10

5) Research about Linux having thread-safe STL containers.

6) Research if the application code should run as a daemon since there will not be standard in/out usage and will be run in the background until the Linux OS reboots.

7) Extend the kernel guidelines, signals, and IPC sections.

2.2 Operating System Concepts

2.2.1 Tasks vs. Processes/Threads

In VxWorks, every unit of execution is referred to as a task. In Linux, a *process* defines the unit of execution which the operating system uses for its scheduling purposes. Each process is identified by its unique process ID, sometimes called *pid.*

A process on a Linux system consists of the following fundamental elements: memory mapping table (page mappings), signal dispatching table, set of file descriptor set (locks, sockets), signal mask and machine context (program counter, stack memory, stack pointer, registers). On every process switch, the kernel saves and restores these ingredients for the individual processes; therefore, context switch between processes is considered to be an heavyweight function.

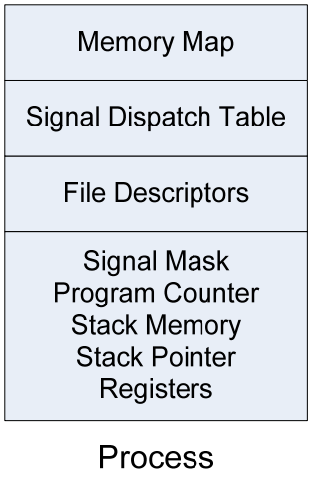

Figure 2: Attributes of a Process

Linux threads are lightweight processes (LWP) and represent a finer-grained unit of execution than processes. Threads share all of the above attributes except the machine context and signal mask. In other words, a thread consists of only a program counter, stack memory, stack pointer, registers and signal mask. All other elements, in particular the virtual memory, are shared with the other threads of the same parent process. Threads require less overhead than "forking" or spawning a new process because the system does not initialize a new system virtual memory space and environment for the process, therefore, the context switch is more lightweight than for processes.

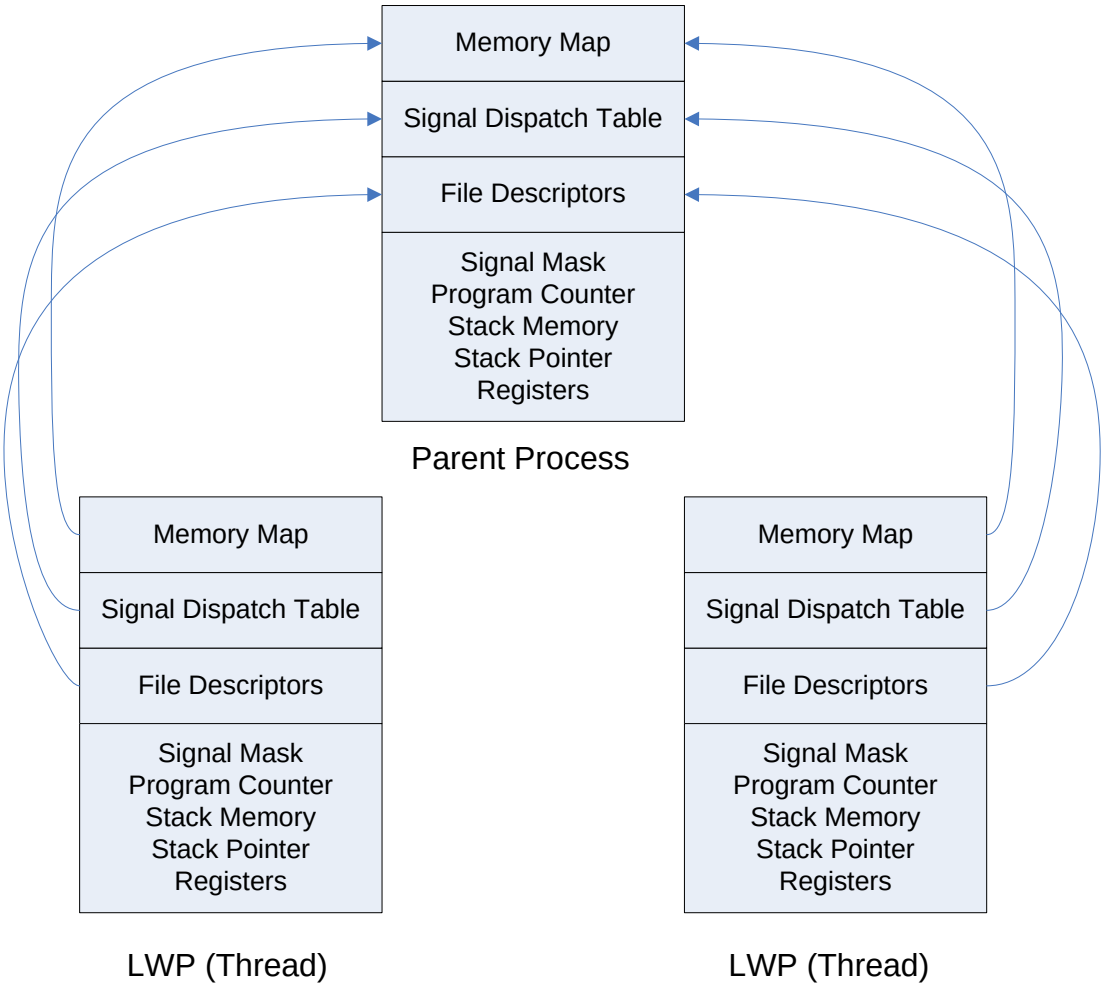

Figure 3: Process vs Thread (LWP)

The concept is that a single process can have a number of threads or LWPs and everything is shared between them except the machine context and signal mask. This way, if a shared resource is modified in one thread, the change is visible in all other threads. The disadvantage is that care must be taken to avoid problems where multiple threads try to access a shared resource at the same time.

The Linux kernel does not distinguish between a process and a thread with regards to the structure that defines it. Both use the *task_struct* definition and both are scheduled globally by the operating system.

2.2.2 Kernel vs. User Space

In an embedded operating system like VxWorks, the kernel is very lightweight and does not interfere excessively with DPL application programming and the context switch between non-kernel tasks and kernel tasks is minimal.

In non-embedded, multi-tasking operating systems, such as Windows or UNIX, there is a solid separation between user space and kernel space. In fact, there are two different modes of operation for the CPU(s): user mode, which allows normal user programs to run; kernel mode, which allows special instructions to run that only the kernel can execute such as I/O instructions, processor interrupts, etc...

When a user space program needs to execute a kernel call, it must make a *system call*, which is a library function that starts out by executing a special *trap* (int 0x80) instruction. This allows the hardware to give control to the kernel, which verifies the user mode's permissions and performs the requested task.

It should be obvious from the above text that there is a performance penalty in making a system/kernel call in Linux since it has to switch from running the user space process to executing its own kernel and back again. It is a good idea to keep the number of system calls used in a program to a minimum and get each call to do as much as possible; for example, read/write large amounts of data versus a single character at a time.

Kernel is sometimes referred to as drivers since performance critical code, such as drivers, are written for kernel space. However, the current trend seems to be minimizing driver logic and keeping the intelligence in user space with some communication mechanism between kernel and user spaces.

2.2.3 Linux POSIX Threads (pThreads)

POSIX threads, commonly called pThreads, is a portable, standard threading API for C/C++ that is supported on a number of different operating systems, including many different flavors of Unix, MacOS, and Microsoft Windows.

With the introduction of Linux 2.6 kernel, the Native POSIX Thread Library (NPTL) was included into the Linux distributions so that multi-threaded user-level application code could make use of multiple processors or cores.

POSIX threads are built around a 1:1 threading model, which basically means that there is one and the same entity that describes a thread or a process. In other words, each thread (or process) has its own *task_struct*. The *task_struct* for a thread shares many of the attributes such as memory mapping table (page mappings), signal dispatching table, and set of file descriptor set with the parent process that contains it. What is immensely important here is that each thread is independently schedulable by the Linux scheduler and portable to symmetric multiprocessor environments. This also implies that threads can block independently of each other.

2.2.4 Scheduling

The DPL RAID application is designed on one fundamental assumption – the RAID application tasks are not pre-empted by another RAID task, but the tasks are scheduled based upon their priority level. Under VxWorks, the RAID application is non-pre-emptive, and system tasks are pre-emptive. The non-preemptive behavior is achieved by having the current RAID task run at a priority that is higher than all other RAID tasks. When the current RAID task blocks, one of the lower priority waiting RAID tasks is selected for execution, at which time its priority is raised and the previously running RAID task's priority is lowered. This was designed to capitalize on the performance and embedded aspects of the core RAID engine.

In the Linux environment, the Linux processes are preemptive. In preemptive scheduling, the scheduler lets a thread execute until a blocking situation occurs (usually a function call which would block) or the assigned time slice elapses. Then it detracts control from the thread without a chance for the thread to object. There are several scheduling policies that are applicable with the POSIX library, however, the default preemptive, time-sliced model is sufficient for most non-real-time environments.

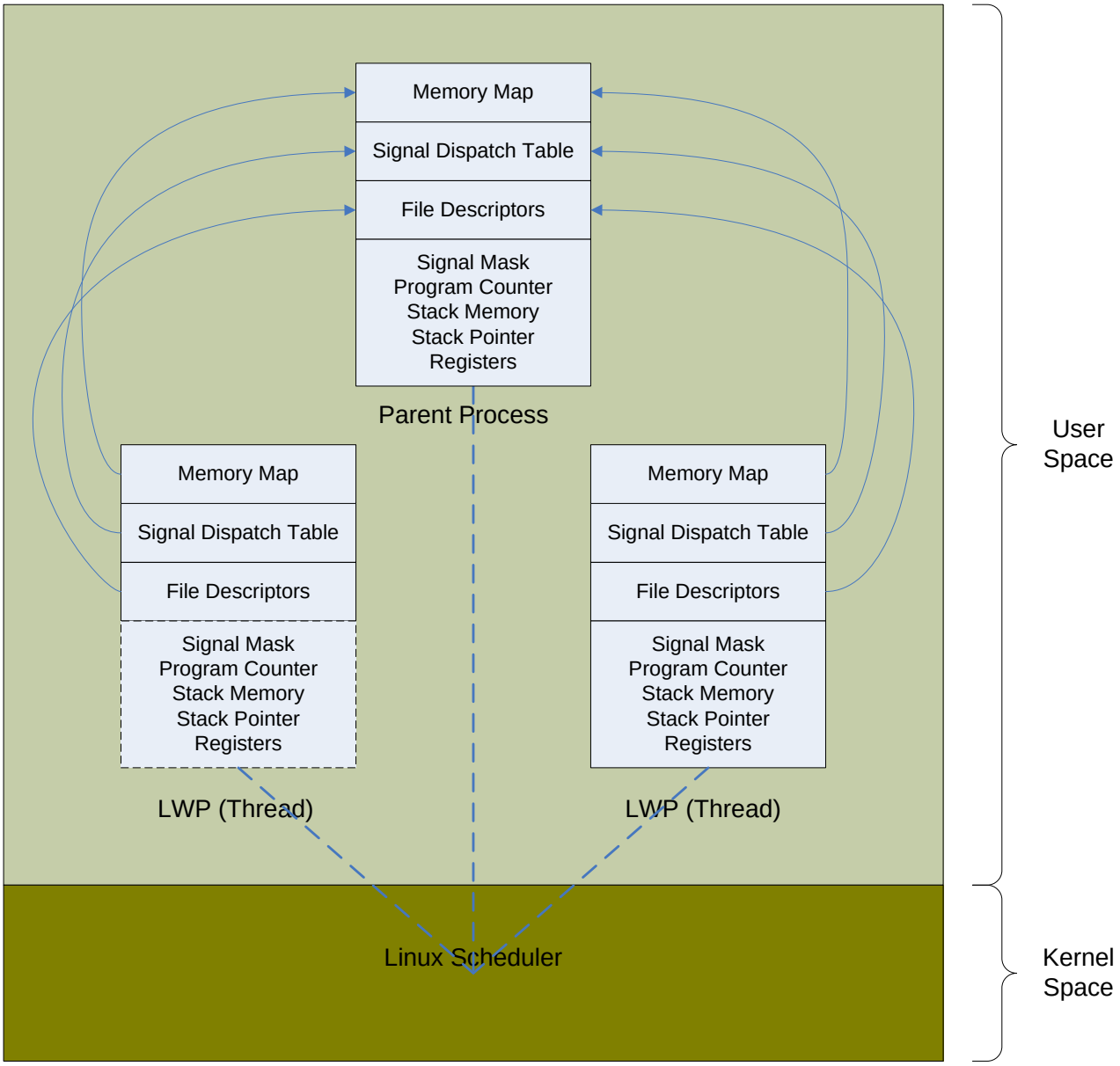

Figure 4: Threads/Processes controlled by the Linux Scheduler

Even though the above diagram depicts a process and threads in user-space, they can also reside in kernel space. This main purpose of the diagram is to illustrate that the Linux kernel scheduler is responsible for all scheduling, both processes and threads (LWPs).

2.3 Operating System Abstraction

In DPL, there are Virtual Kernel Interface (VKI) and Virtual Network Interface (VNI) layers that are used as abstraction layers between the operating system and the RAID application.

This section discusses the need to introduce that type of abstraction for the other guest Linux virtual machines that make up the OSA solution.

There are several benefits of a VKI layer:

1) It provides a common API across different operating systems. This eases portability since the operating system interfaces are abstracted via a single component versus all throughout the application. It should be noted that this also reduces porting effort between different versions of the same OS.
2) It can be used to prevent certain operating system calls or usage of certain parameters. This allows control over what the developer can use. For example, there could be a system/kernel call where one of the options of the parameters should not be allowed in the embedded environment.
3) It allows the ability to add/extend tracing, debugging, and functionality beyond what the OS provides. For example, in DPL, VNI wraps the network related OS calls and in some cases wraps our own system functions for storing and retrieving our network configuration.
4) It allows straightforward identification of system/kernel calls; makes it easier to locate where in the code system/kernel calls are being made.

Given the above benefits, the VKI layer that currently exists for VxWorks should be used for Linux as well. In situations where it is VxWorks-centric, it should be made more general to accommodate both Linux and VxWorks operating systems. Despite this abstraction, developers should know the underlying mechanisms of the system/kernel calls and it should be documented in the VKI specification.

The following key requirements must exist for VKI:

1) Well documented.
2) Lightweight – no performance overhead for commonly-used calls.
3) Supports both Linux user space and kernel space (which implies no c++)
4) Naming convention such that benefit #4 above is achieved.

There are some existing libraries, such as ACE (http://www.cs.wustl.edu/~schmidt/ACE.html), that provide an operating system abstraction but they are implemented in C++, which invalidates requirement #3 above. Section 2.4 discusses licensing requirements and since OSA is determined to be an embedded solution, the VKI module can remain closed, even though it is part of the kernel space.

VKI/VNI interfaces can be categorized based upon the underlying components they interact with. In brief, VKI/VNI can be categorized as follows:

- Interfaces using direct VxWorks/Linux system/kernel calls
- Interfaces using kernel module to get kernel services
- Interfaces that combine multiple system/kernel calls as a single call to ease development

The VKI layer will also be used to wrap the POSIX thread API. This is a contentious subject area as the POSIX thread API is already a standard. However, code developed by CFW will only allow a subset of the features of POSIX; in fact, some of the options of the Linux POSIX library are non-portable outside of Linux. For example, the mutex portion of the POSIX library in Linux provides multiple types of mutexes (fast, recursive, and error-checking), however, only the fast mutex is portable and the other two are not. In addition, combining multiple POSIX functions in a single call to ease developer usage should be considered.

Note that the use of the VKI/VNI layer will be mandated for code developed by the CFW organization, for both VxWorks and Linux virtual machines. It will not be enforced for library packages or organizations outside of the CFW team such as the DML team(s). Furthermore, it should not be used for GPL packages that will be further enhance by LSI since the modified changes will need to be made open, resulting in the VKI module to be open-sourced.

Application code should not include OS header files but rather include VKI header files that include OS header files. Moreover, there will be a common definition file that will identify primitive types – they will be defined with the number of bits in the definition name, such as INT32, CHAR8, UINT32, ULONG64 – this will ease portability.

Currently all the VNI calls are defined in the vniWrap.h header file in DPL. These calls can be split in to two sections: VNI RPC calls and VNI non-RPC calls.

The VNI implementation for Linux should implement all the necessary calls and provide the exact same functionality that is currently available on VxWorks implementation. For two non-RPC macros *VNI_HOST_GET_BY_ADDR(addr, name)* and *VNI_HOST_GET_BY_NAME(name)* the return values in Linux are different from that in VxWorks. The return values should be made compatible to that in VxWorks.

2.4 Licensing Guidelines

Non-Embedded Linux Solutions

(Described for completeness/comparison purposes but does not apply to OSA.)

Kernel resident modules are released as open source under the GPL. User-space applications may remain closed. However, including other GPL libraries or SW modules in a Linux application may trigger the GPL requirement. This excludes .h files, run-time libraries and other libraries that are distributed with Linux. However, libraries that are not distributed as part of Linux and distributed as open-source components work differently. LGPL (L stands for Lesser or Library) was created for open source libraries and removes the viral clause in GPL. You can link LGPL libraries into a closed application. However, many open source libraries are available and distributed under GPL and they are viral. A common mistake is to assume that an open source package is a library covered under LGPL.

Furthermore, if LSI develops a kernel module, that kernel module is subject to GPL, however, LSI owns the IP from that module. Since LSI owns the IP, we can use the same kernel source in our closed application without contaminating the closed application. In other words, the kernel module can be distributed with multiple sets of licenses, one with GPL for the kernel-only module; and one as closed with kernel/application together. If LSI uses someone else's kernel module that is "not part of Linux", then we cannot use it in our closed application without contaminating our closed application unless there is a “private contract” to own the IP for that kernel module.

Embedded Linux Solutions

Embedded Linux solutions have less restrictive GPL requirements than non-embedded Linux solutions. In other words, loadable kernel modules (LKM) can remain closed. OnStor, DPM, and Orion (or OSA solutions) fall into the category of Embedded Linux Solutions.

Given this direction, loadable kernel modules can be closed for Orion, however, modifications to existing kernel modules should be open sourced.

Example: The coupling driver for Orion uses SCSI protocol based on DCP (Dual-Core Protocol), which is open-sourced and has been given to us under two licenses: both GPL and a closed proprietary license between Intel and LSI. Given these two options, it can be used as follows:

1. We can use the Coupling driver in RAIDCore under the closed license from Intel to avoid infecting RAIDCore with GPL.
2. We can use the coupling driver under GPL on the Linux side in order to more fully comply with the Linux GPL.
3. If our coupling driver modifications do not expose proprietary IP, we should include those changes in the GPL open source in order to more fully comply with GPL.

Since close source in Linux is a 'gray area', anything we can open source without exposing proprietary IP helps build our credibility with the Linux community and reduces the risk of complaints. It shows we are trying to be good members of the Linux community by giving something back.

All licensing issues within CFW must be routed through the Firmware Architecture group as they are responsible for setting the guidelines and direction for the CFW organization. The FW Architecture group will perform a periodic audit with Open-Source Software Review Board (OSSRB), especially after the initial OSA design is matured. This activity must be performed pro-actively, as matters not caught early in the development process can potentially result in severe consequences.

General guidelines are documented on the OSSRB Wiki at http://ictwiki.ks.lsil.com/index.php/Open_Source_Leadership_Team (See the section titled: "Guidelines for Common Licenses")

2.5 Linux Programming Guidelines

Almost all of the code written by CFW for the Linux OSes will reside in user-space. There will be some kernel modules such as the SCSI coupling driver but most of the code will exist in user space. Here are few of the advantages and disadvantages with user-space code:

Advantages	Disadvantages
Debugging is easier than with kernel modules. In other words, a crash in the module will not result in rebooting the OS	High latency/context switch when communicating with system calls
Open source issues are manageable	

Figure 5: Advantages/Disadvantages of User-Space code

2.5.1 User Space Guidelines

The code should make use of POSIX threads with VKI abstractions – this will enable the code to be SMP-compliant and that option can be used, if necessary. In addition, all kernel/system calls should be abstracted via the VKI/VNI layer.

The initial direction for the user-space program for the Linux VMs is to use utilize (POSIX) threads versus creating a process per functionality. Multiple processes can belong to a single program but given the heavyweight attributes associated with multiple processes, this should only be used for non-performance critical code. Note that this direction will most likely evolve over time as more design and functionality matures for the Linux VMs.

The disadvantage of having just one process with multiple tasks is that all tasks within that process will use the same memory space, which could result in memory corruption for several/unrelated tasks due to errant code. Due to this rational, it is advisable to partition the functions between processes/threads appropriately.

The guideline for threads will follow functional separation for the most part. In other words, functionality with a distinct purpose is created as a thread. It may optionally create multiple threads. For example, the health-check service will be created as a thread; the IP communication layer (RAS/IPIVM) for the VMs will be created as multiple threads. All threads within a process share the same address space so only related threads should be added as part of the same process. Do note that since the address space is shared between the threads, care must be taken to protect critical sections. As mentioned above, if properly coded, this implementation can implicitly take advantage of SMPs.

The scheduling policy will be a time-sliced, preemptive environment. The priorities of the threads of the process will not change from the default. Once again, this can be examined as the OSA project progresses.

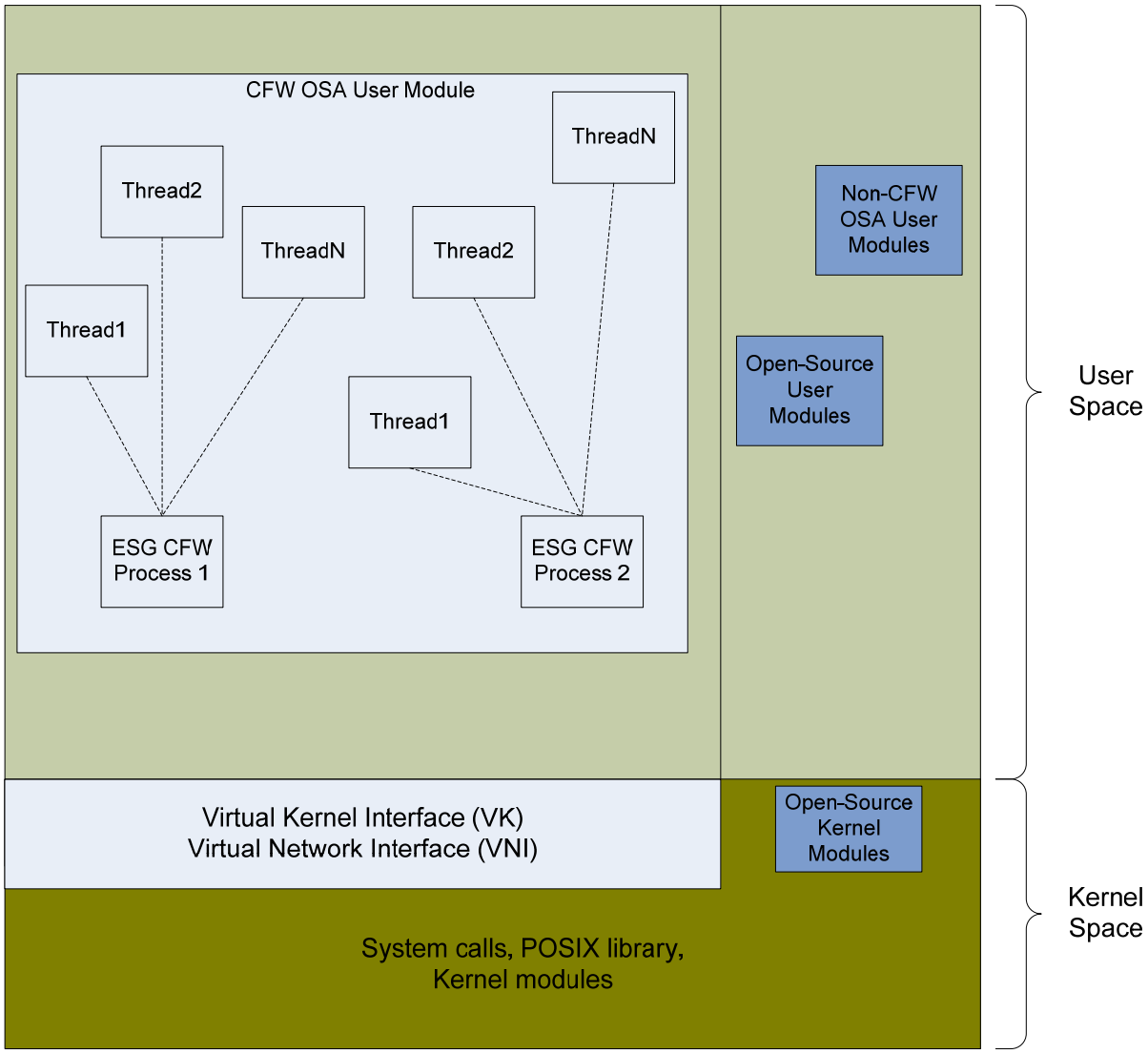

Figure 6: CFW OSA Environment for the Linux Virtual Machines

2.5.2 Kernel Mode Guidelines

In general, kernel threads should only be used for highly optimized functionality that needs direct control of hardware devices. Kernel development is much more difficult than user space work because of the lack of memory protection and debug-ability. In addition, kernel modules must be released to the open-source community under General Public License (GPL).

Kernel development is debug-able using kdb or kgdb.

If there is code that needs to support multiple operating systems, then the KERNEL_VERSION preprocessor macro should be used for compilation purposes. Gears variation cannot be used for kernel modules as that will end up being GPL as well.

The mechanism to add a new IOCTL API when there is a user to kernel interface is deprecated and should not be used for new development. Instead, netlink sockets should be considered for fast user to kernel communication.

Guidelines for kernel modules will be extended as the document evolves.

2.6 Linux Concepts

This section lists useful constructs in Linux that would be applicable for OSA CFW - it is not exhaustive by any means. As with the rest of the document, this portion will be updated as the OSA program progresses.

Note that even though the interfaces are being referenced directly in the below sub-sections, they must be called with a VKI/VNI interface in the CFW code. It is done in that manner here since the VKI/VNI layer is not present in the Linux VMs. To some extent, these sub-sections should be used for that VKI/VNI layer.

2.6.1 System Calls

In Linux, a system call is the fundamental interface between an user process and the Linux kernel. It allows user processes to request services from the operating system. There are blocking and non-blocking system calls. System calls are not invoked directly; instead C library wrapper functions perform the steps required (such as trapping to kernel mode) in order to invoke the system call. From the developer standpoint, it looks the same as invoking a normal library function.

More information about system functions can be gathered by looking at their header files, which reside in /usr/include and /usr/include/sys. Also, they could be found by invoking the manual pages on system calls (*man* syscalls).

Private system files that should not be directly included from the user programs are located in /usr/include/bits, /usr/include/asm, and /usr/include/linux. They are included from the other “main” system files that reside /usr/include and /usr/include/sys.

2.6.1.1 Error Codes

Majority of system calls return zero if the operation succeeds or a nonzero value if the operation fails. Most system calls use a special variable named *errno* to store additional information in case of failure.

Error values are integers; possible values are given by preprocessor macros, by convention named in all capitals and starting with “E”—for example, EACCES and EINVAL. Always use these macros to refer to errno values rather than integer values. Include the <errno.h> header if you use errno values. GNU/Linux provides a convenient function, strerror, that returns a character string description of an errno error code, suitable for use in error messages. Include <string.h> if you use strerror (VKI_STRERROR).

One possible error code that you should be on the watch for, especially with I/O functions, is EINTR. Some functions, such as read, select, and sleep, can take significant time to execute. These are considered *blocking* functions because program execution is blocked until the call is completed. However, if the program receives a signal while blocked in one of these calls, the call will return without completing the operation.
In this case, errno is set to EINTR. Usually, you'll want to retry the system call in this case.

(Note: Need to investigate if VKI should abstract these values or not. Currently in VxWorks/DPL, the errno returned by the OS is not interpreted by the application code.)

2.6.2 POSIX Threads

Linux implements the POSIX standard thread API (known as *pthreads).* All thread functions and data types are declared in the header file <pthread.h>. The pthread functions are not included in the standard C library; instead, they are part of *libpthread* so "-lpthread" should be used at the command line to link the library - note that this will be handled by the makefile(s). The -lpthread says to link with the system pthread library.

There is thread ID of type pthread_t.

2.6.2.1 Thread Creation

pthread_create creates a new thread and takes the following 4 parameters:

Pointer to pthread_t

Pointer to thread attribute object. Attributes such as scheduling policy, scope, stack address, and stack size can be specified. The VKI layer will prevent some of the options allowed for the attribute object.

Pointer to thread function "void* (*) (void*)" – this is the starting function that is called when the thread (LWP) is created.

Thread argument value of void* (this is passed into the above thread function)

2.6.2.2 Thread Specific Data (TSD)

Programs often need memory for a given thread that should not be shared with other threads. Since threads share one memory space of the parent process, POSIX provides a way to possess memory that is private to individual threads. This memory is called "thread specific data" (TSD) and there is a key associated with such memory. The keys are common to all threads, but the value associated with a given key can be different in each thread.

pthread_key_create creates a key with a passed-in destroy function that is called on cleanup

pthread_key_setspecific is used to change the value associated with the key

pthread__getspecific is used to return the value of the associated key

When the thread is destroyed (when thread terminates or via *pthread_exit*), the destroy function is called with the value of the key as the argument.

pthread_key_thread de-allocates a TSD key but the destroy function is not called. This does not seem like a good interface to be provided by VKI.

It might be worthwhile to present an interface where the thread specific data is created at the time the thread is created. Instead of calling the above interfaces separately from the create routine and individually as specified above, the VKI layer can provide one interface that not only creates a thread but also sets up the thread specific data. This will not only allow the developer to understand the memory model at the time of the thread creation but also prevent misuse of the steps required to obtain/destroy thread specific data.

2.6.2.3 Thread Deletion

Threads can be terminated explicitly by calling *pthread_exit* or by letting the function return.

All cleanup handlers that have been set for the calling thread with *pthread_cleanup_push* are executed.

These interfaces need to be analyzed for VKI inclusion.

2.6.3 Synchronization

Synchronization is the methods for ensuring that multiple threads do not accidently modify the data that another thread is using. This can be an issue for functionality that is multi-threaded as threads of the same parent process have the same memory space. The race condition of multiple threads trying to access same data could result in either stale access or a crash (segmentation fault).

To eliminate race conditions, operations must be atomic. Atomic operation is indivisible and uninterruptible; not pause or be interrupted until it completes and no other operation can take place meanwhile.

Critical section is that chunk of code and data that must be allowed to complete atomically with no interruption. They should normally be short as possible since concurrency of the program is affected if they are not. All data structures that can be accessed by multiple threads must be protected or locked.

2.6.3.1 Mutual Exclusion

The mutual exclusion lock is the simplest synchronization mechanism. It provides a way to lock the critical section such that only one thread can access it at any given time. POSIX provides *pthread_mutex_init, pthread_mutex_lock,* and *pthread_mutex_unlock* for this purpose. There is also a non-blocking mutex test called *pthread_mutex_trylock*. This will test to see if the mutex is locked; if it is not locked, then it will lock, else it will return immediately with error code EBUSY. This is a good mechanism to use if the program wants to perform some other task when it's locked.

2.6.3.2 Semaphores

Semaphores can also be used to protect critical sections. Mutexes are a restrictive form of semaphores called binary semaphore, where the section can either be locked or unlocked. Semaphores are perfect for a situation there is a need to count protected information. s*em_wait* and *sem_post* are defined in semaphore.h and are used for this purpose.

There are some clear distinctions between mutexes and semaphores that are worth noting:

1) With mutexes, thread that owns it is responsible for freeing it, however, with semaphores, any thread can free it.
2) Semaphores are system-wide and remain in the form of files on the filesystem, unless otherwise cleaned up. Mutex are process-wide and get cleaned up automatically when a process exits.
3) Mutex are lighter when compared to semaphores. What this means is that a program with semaphore usage has a higher memory footprint when compared to a program having Mutex.
4) The nature of semaphores makes it possible to use them in synchronizing related and unrelated process, as well as between threads. Mutex can be used only in synchronizing between threads and at most between related processes.

There are several locks in DPL such as MasterTransactionLock that can used globally across the application. These types of overall locks will be introduced in the Linux VMs, depending on the requirement, as the OSA program progresses.

2.6.3.3 Conditional Variables

A condition variable is a variable of type *pthread_cond_t* and is used with the appropriate functions for waiting and continuation. The condition variable mechanism allows threads to suspend execution and relinquish the processor until some condition is true. A condition variable must always be associated with a mutex to avoid a race condition.Any mutex can be used, there is no explicit link between the mutex and the condition variable. *pthread_cond_init, pthread_cond_wait, pthread_cond_signal,* and *pthread_cond_broadcast* are all used for this purpose.

2.6.4 Communication between processes (IPC)

Signals are obnoxious in that their behavior depends on the operating system. For example in some cases, the signal handler is responsible for re-registering itself when it is invoked. This allows a small window where another signal could kill the application.

Due to the above reason, signals should be avoided for the purpose of IPC; sockets or FIFOs should be used.

2.6.5 Debugging in Linux

GNU Debuggers (GDB) is the debugger used by most Linux programmers. You can use GDB to set through your code, set breakpoints, and examine the values of local variables.

To use GDB, compile the code with debugging information enabled. The –g switch on the compiler allows that to occur. The compiler generates extra information in the object files and executables, which is useful when the gdb is actually executed.

GDB is a command line oriented debugger and there are user friendly shells available to interface with it.

2.7 Lock Management Guidelines

There will be a need for Linux virtual machines to synchronize code such that multiple threads or peer virtual machines are not executing critical sections simultaneously.

There will be a notion of component or functionality-specific lock, lock for a given controller (LOCK_CTLR), lock for both controllers (LOCK_CTLR on both ctlrs), and a lock used for persistence (LOCK_PERSISTENCE). All these locks, besides the component-specific lock will be introduced in a module called LOCK in the Linux virtual machines, such as Domain0 and Service VM. The locks should be provided in an object-oriented manner, similar to *txn::Transaction* and *txn::MasterTransaction* classes from DPL. Note that heartbeat mechanisms are documented in the *Hypervisor AAD – Virtual Machine Management* document (44341-00).

To aid the above requirements, the following guidelines should be followed:

Persistent Changes Required?	Inter-controller concurrency?	Prevent all configuration changes on that controller?	Guideline	Notes
YES	Implicitly yes	Implicitly yes	LOCK_CTLR on both controllers, followed by	If persistence or inter-controller concurrency

			LOCK_PERSISTENCE, which is acquired internally by persistence mgr	is required, then this requires a heartbeat mechanism
NO	YES	Implicitly yes	LOCK_CTLR on both controller	Requires heartbeat mechanism
NO	NO	NO	LOCK_CTLR on that controller	
NO	NO	NO	Perhaps this needs a component specific lock to prevent other changes in that component.	These finer grained locks should be provided by the component itself.

Figure 7: Lock Management Guidelines for Linux Virtual Machines

As mentioned generally above, guidelines will be further modified as OSA solutions are matured with respect to their design and implementation.

2.8 Object Graph Population Guidelines for non-DPL VMs

There will be situations where non-DPL VMs will be responsible for managing data that is populated in SYMbolAPI's ObjectBundle. In those situations, non-IOVMs should minimize its involvement with SYMbolAPI constructs as that is the responsibility of IOVM/DPL.

The following guidelines should be followed:

1) During SOD, components in IOVM are responsible for querying (pulling) information from the non-IOVMs in order to populate the data in the ObjectGraph.

2) If the managed data changes in non-IOVM due to some SYMbolAPI initiated command, then the non-IOVM should communicate this information back to IOVM in the return data structure, instead of having another IP message to report the modified data.

3) In general, SYMbolAPI constructs/dependencies should be managed by IOVM before invoking non-IOVM about the operation. For example, *setEthernetInterfaceProperties_1* can be called on one controller to be executed on the other controller. In this situation, it is the responsibility of IOVM to peer that request to the alternate's controller's IOVM and then to non-IOVM on that controller. In essence, the peering should not occur in the non-IOVM since that is being done due to SYMbolAPI constructs.

4) If the data is being managed by only one controller's non-IOVM (of the two), then the responsibility falls on IOVM to peer that data to the alternate controller's IOVM for object graph population purposes. The information to let IOVM know about alternate controller communication should be passed as part of the data exchange.

5) If the data is being managed by non-IOVM on both controllers, then it is the responsibility of the non-IOVM to notify IOVM on both controllers, thereby eliminating the necessity for IOVM to do this exchange between controllers.

6) When non-IOVMs inform IOVM about the modified data, IOVM should accept the data in an asynchronous fashion. In other words, IOVM should not execute the population of the object graph in the context of the notification, unless the modified data is being exchanged within the context of a return from an invoked SYMbolAPI operation from IOVM. This guideline exists so that there is no inadvertent cascading lock acquisition condition. Besides, the completion of the command or event on non-IOVM is separate and distinct from the object graph population operation in IOVM, therefore, they should be de-coupled and

treated independently.

As mentioned generally above, guidelines are subject to modifications as OSA solutions are matured with respect to their design and implementation.

2.9 Start of Day on Domain0

Processes are created via startup scripts kicked off by the init process. This is done with /etc/rc* scripts and newer systems use upstart, which is event based (More on this information in later revisions).
Since threads will be defined by functionality – here are a list of threads that are applicable on Domain0 so far:

- Health-check monitoring thread
- Virtual watchdog monitoring thread (design has yet to be matured)
- IPIVM threads (for IP communication between VMs)
- General threads needed for processing requests from other VMs
  - Similar to non-RW threads
  - Waits on semaphore or queue and executes if a request arrives

Note that these threads will be created as start-of-day processes in Domain0.

2.10 Tools / Utilities / Libraries

The following (incomplete) list of tools/utilities/libraries should be considered for OSA Linux virtual machines:

Boost libraries should be considered for general, infrastructure-related libraries; it is primarily used for C++ code. See http://en.wikipedia.org/wiki/Boost_C%2B%2B_Libraries.

ACE (Adapative Communication Environment - http://www.cs.wustl.edu/~schmidt/ACE.html) is a C++ framework for various infrastructure-related libraries. It is stable and easily portable to another OS. ACE provides a C++ wrapper over typically a C OS API and there are abstractions for IPC (messages, queues), threads, synchronization/concurrency, timers, memory management and networking. One downside might be that documentation is not excellent, when compared to some of the well-established APIs, such as MSDN or Java.

Linux has a host of memory debugging tools. In user space, there is dmalloc (http://dmalloc.com). In kernel space, there is the standard kmalloc_debug facilities provide by the kernel; there are also more powerful tools such as kmemcheck (http://lxr.linux.no/linux+*/Documentation/kmemcheck.txt) and kmemleak (http://lxr.linux.no/linux+*/Documentation/kmemleak.txt).

2.11 References / Resources

Here are some good resources on POSIX threads:

http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

https://computing.llnl.gov/tutorials/pthreads/

Multithreaded Programming with PThreads by Bil Lewis, Daniel J. Berg

If the development environment or Linux servers are not ready, VMware software could be used on any Windows desktop platform to run Linux. It could be used for education, experiments as well as coming up to speed to develop for the OSA program.

VMPlayer can be downloaded from http://www.vmware.com/products/player/. After which, Linux images (such as CentOS) can be downloaded from http://www.thoughtpolice.co.uk/vmware/.