Open Storage Architecture Programming Guidelines

Author: Satish Sangapu

Date: 08/30/2009

Table of Contents

Most of our user space code need to run as a service in the background. This is done by converting a process in to daemon. I think we should add a section on daemons how to daemonize a process in linux.

Table of Figures

1. Purpose/Goals

Currently, all the controller firmware developers comprehend the non-pre-emptive multi-tasking model that exists with Data Path 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. Note that at this time, embedded Linux operating system has not been seriously considered for any of the virtual machines.

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.

OSA Development and Build Environment FAM documents the source code management system, build environment, compiler details, and packaging tools/process for the OSA environment. (Document 44682-00)

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, RAS/IPIVM, SOD component, initial Hypervisor related activities) [Maybe this doesn't make sense for this document….]

2. Operating System Concepts

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 operation system uses for its scheduling purposes. Each process is identified by its unique process ID, sometimes called *pid.*

A process on 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 is heavyweight.

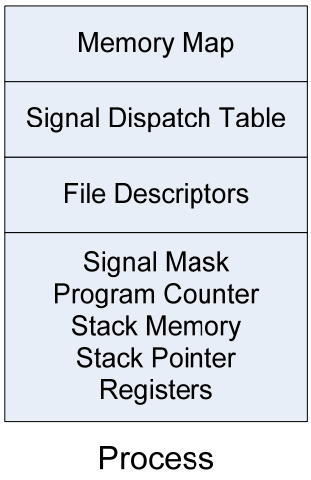

FIGURE 1: ATTRIBUTES OF A PROCESS

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 lightweight than for processes.

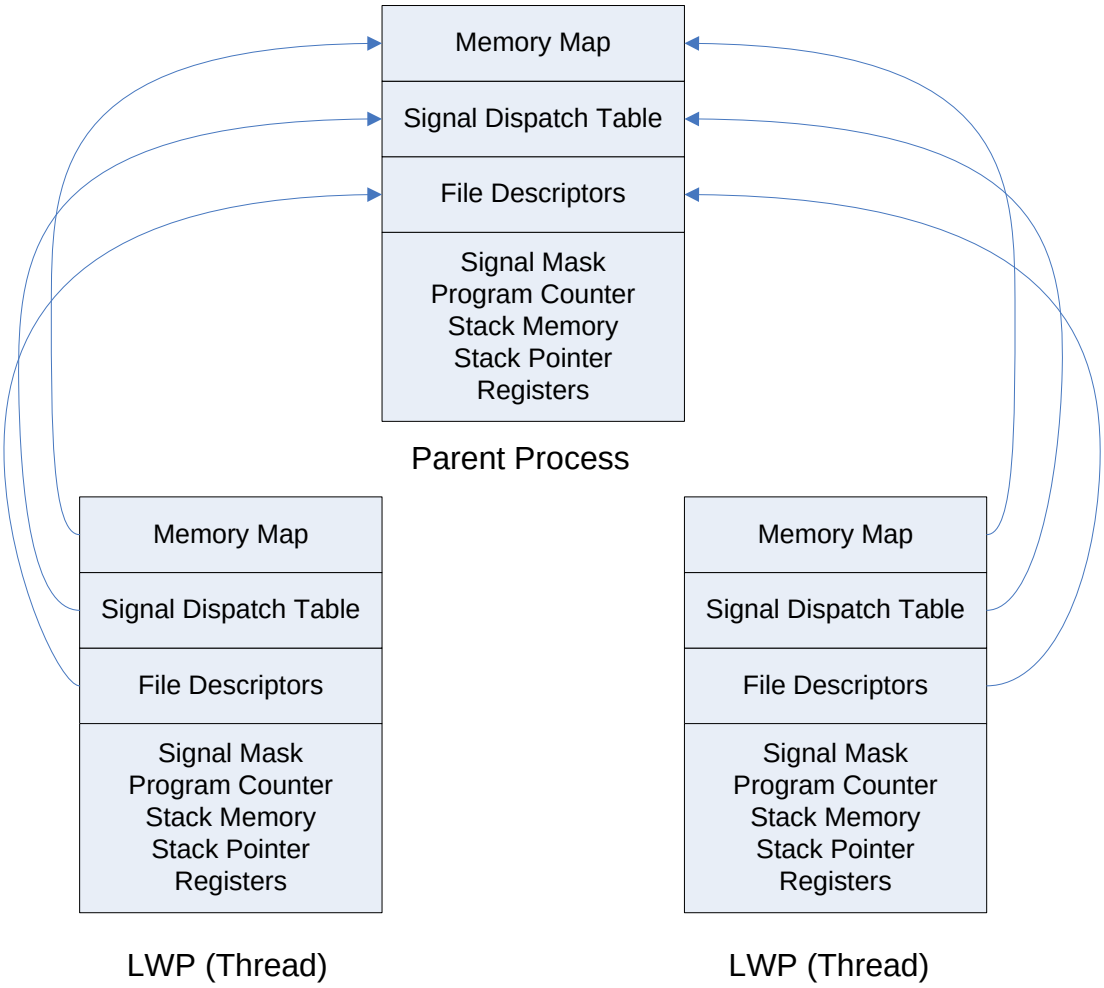

FIGURE 2: PROCESS VS THREAD (LWP)

The concept is that a single process can have a number of threads or LWPs and everything ~~that~~ 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.

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).

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 a 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 as normally, performance critical code, such as drivers, are written in kernel mode. 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. 

2.3. 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 (locks, sockets) 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.

2.4. Scheduling

The DPL RAID application is designed on one fundamental assumption – the main RAID application task is not pre-empted by another task within the application, 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 increasing the RAID application task's priority level to more than that of the highest priority task's (of RAID application), but less than that of the system tasks. This way when a RAID application task under VxWorks is scheduled to run, it will not be preempted by another RAID application task until it relinquishes the CPU through a VxWorks system call. 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. This is usually realized by interrupting the thread through a hardware interrupt signal (for kernel-space threads) or a software interrupt signal (for user-space threads), like `SIGALRM` or `SIGVTALRM`.

There are several scheduling policies that are applicable with the POSIX library, however, the default preemptive, time-sliced model is sufficient for most non-realtime environments.

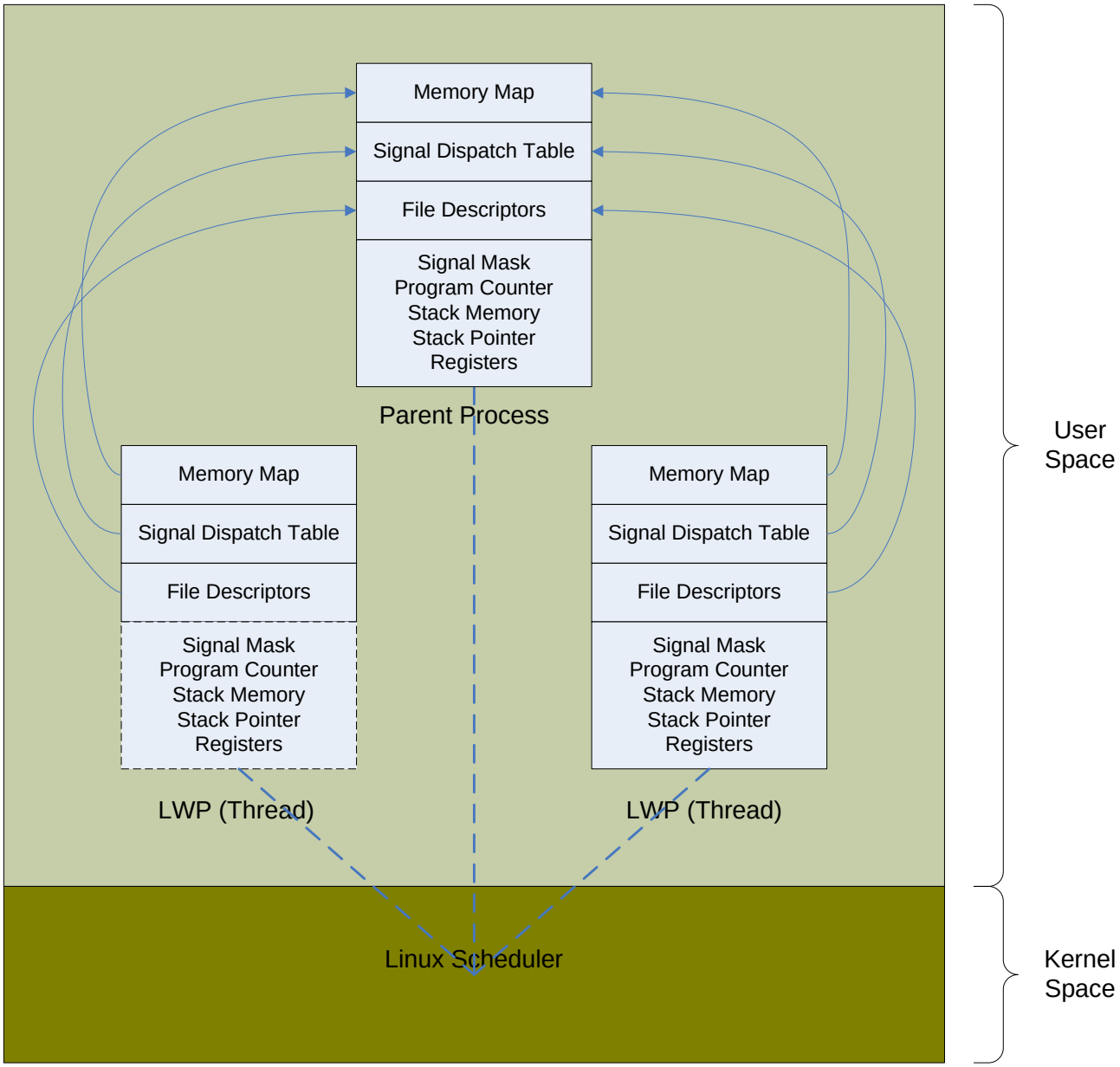

FIGURE 3: THREADS/PROCESSES CONTROLLED BY THE LINUX SCHEDULER

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).

3. Operating System Abstraction

DPL, there is a Virtual Kernel Interface (VKI) and Virtual Network Interface (VNI) layer that is used as an abstraction layer between the operating system and the RAID application. A similar abstraction layer VKI/VNI will exist for the Linux VMs. This will exist in the user space since most of the application code in Linux that is written for OSA will exist in user space plus there will be no licensing issues with that direction.

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 Linux system calls
- Interfaces using kernel module to get kernel services
- Interfaces that are implemented in user application itself and does not depend upon lower layers
- Interfaces that combine multiple system calls as a single call to ease development

The advantage of a VKI/VNI layer is to ease portability such that the operating system interface is called from one location rather than from all the users of that OS service. Also, it prevents the developer from misusing or abusing the system calls. For example, there could be a system call where one of the options of the parameters is not allowed in this specific environment – in that case, the VKI layer can prevent that from happening.

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 ease of the VKI/VNI layer will be mandated for code developed by the CFW organization. It will not be enforced for library packages or organizations outside of the CFW team such as the DML applications.

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.

4. Licensing Guidelines

Ignore this section for now – I have the material but I would like Ken Gibson to approve my summary

5. Linux Programming Guidelines for CFW

Almost all of the code written by CFW for the Linux OSes will reside in user-space. There will be some kernel modules, especially for the SCSI coupling driver that will reside in kernel space 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 4: ADVANTAGES/DISADVANTAGS OF USER-SPACE CODE

The code should make use of POSIX threads with VKI abstractions – this will enable the code to be SMP-complaint 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 (POSIX) threads versus creating multiple processes. Granted, this direction will most likely change/evolve over time as more design and functionality matures for the Linux VMs. Multiple processes can belong to a single program but given the heavyweight attributes associated with multiple processes, this will only be introduced when deemed necessary.

The scheduling policy will be 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.

The guidelines for the threads will follow functional separation for the most part. In other words, functionality with a distinct purpose is created as a task. 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 tasks should be added as part of the same process. In this situation, all tasks are part of the ESG CFW OSA functionality. Do note that since the address space is shared between the tasks, care must be taken to protect critical sections. As mentioned above, if properly coded, this implementation can automatically take advantage of SMPs..

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 debugability. In addition, kernel modules must be released to the open-source community under General Public License (GPL).

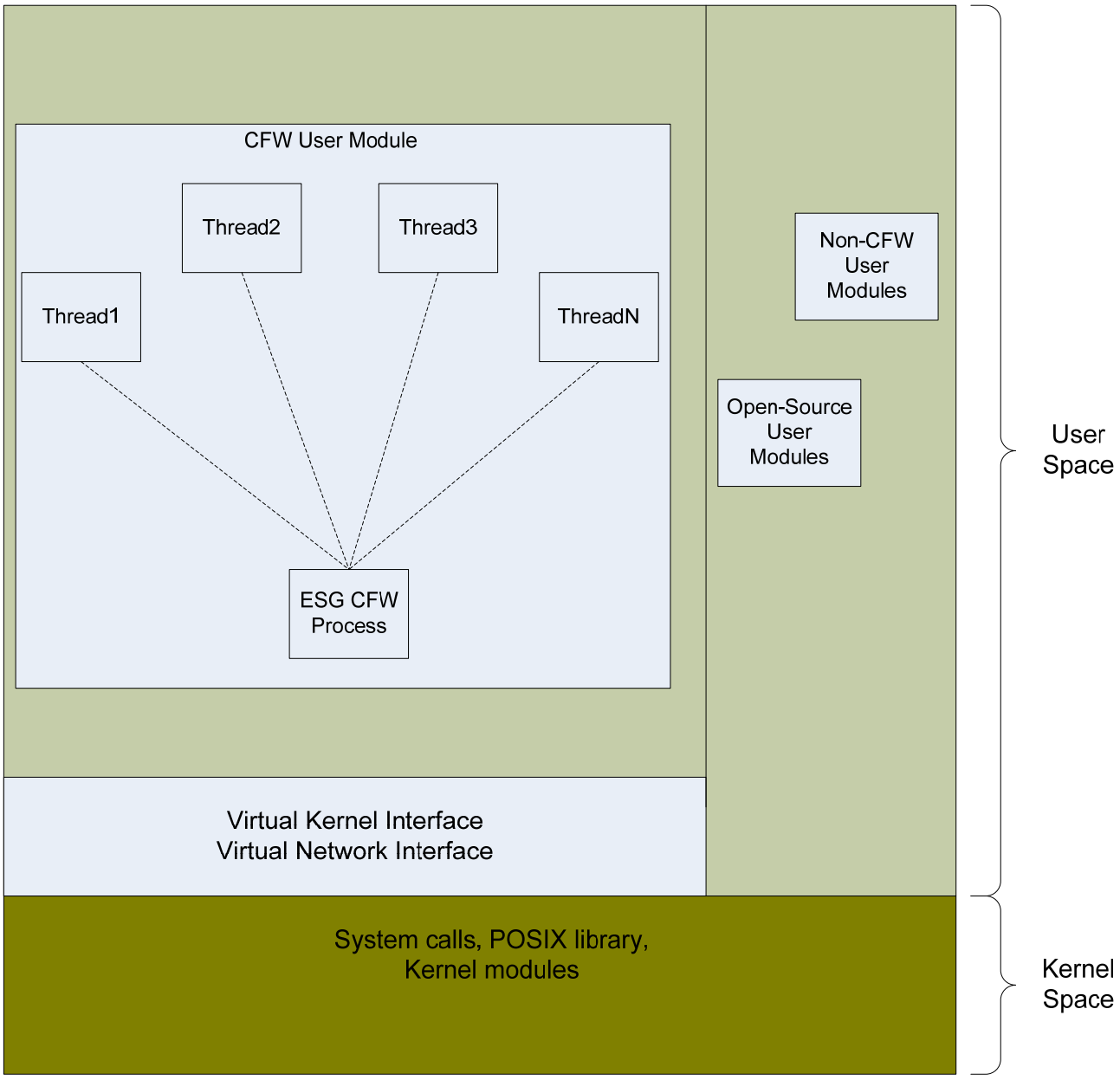

FIGURE 5: CFW OSA ENVIRONMENT FOR THE LINUX VIRTUAL MACHINES

Guidelines for kernel modules are not mentioned in the document. However, that can be added as the document evolves.

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.

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 (asynchronous) and non-blocking (synchronous) 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, could be found by invoking the manual 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.

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.

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.)

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.

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 (NWP) is created.

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

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_exist*), 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.

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.

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.

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.

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 type of overall locks will be introduced in the Linux VMs, depending on the requirement, as the OSA program progresses.

6.3.3. Condition 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.

6.4. Signals

Signals are UNIX mechanisms to interrupt a running process and to communicate with that process. More on this will be provided in a later iteration.

6.5. Communication between processes (IPC)

This section will be covered in later iterations of the document as currently, the design will be based on a single process with multiple threads.

6.6. Debugging in Linux

GNU Debuggers (GDB) is the debugger used by most Linux programmers. You can use GBD 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.

7. Start of Day on Domain0

cesses 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 functionally – 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.

8. References / Resources

Here are many good resources on POSIX threads:

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

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

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/.