# Aspect Architecture Document

# OSA Infrastructure – Programming Guidelines

**Document: 45016-00**

**Revision: A.4**

**Revision Date:  04/04/2010**

|  |  |
|---|---|
| Author: | Kevin Lindgren, 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 |
| A.4 | 04/04/2010 | 2.1 | Modified stated purpose/goals |
| | | 2.2.1 | Added more details about the Linux environment |
| | | 2.3.3 – 2.3.5 | Added pertinent Linux constructs |
| | | 2.4-2.5 | Re-organized and enhanced guidelines about Linux Programming and General OSA |

# Table of Contents

## List of Figures

## List of Tables

# 1. ASPECT ARCHITECTURE DOCUMENT

## 1.1 Aspect Introduction

### 1.1.1 Aspect Description

This aspect contains the infrastructure material to develop and productize OSA (Open Storage Architecture) solutions. More specifically, it contains the following aspect elements: 1) OSA Development and Build Environment 2) OSA Programming Guidelines 3) OSA Persistence.

The OSA environment for Phase1 consists of IO VM, Domain0 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, pertinent Linux constructs, and Linux/OSA guidelines. OSA Programming Guidelines element complements the existing Coding Standards Document. This aspect element is meant to be an evolving document for the Orion program.

The OSA Persistence document cover the Infrastructure for Domain0 Data Persistence which is required for certain application attributes which need to be saved across warm and cold reboots. This does not cover the actual data to be persisted, because that will be described in the corresponding detailed Architecture documents. The data to be persisted from Domain0 will be saved on the IO VM drives in the StableStorage region of the DacStore in the form of specific Domain0 records defined for this purpose. Data to be stored can be classified as Domain0-Specific, which is data specific to the Domain0 on a controller, and global persistent data which is data that applies to both Domain0s and the system in general.

### 1.1.2 Assumptions

Some assumption made for this functionality, include:

- The in-memory copy persisted Domain0-Specific data will be managed by the owning Domain0, and the persisted Domain0 Global data will be managed by the "Designated Domain0" as specified by the Pacemaker package in conjunction with OpenaAIS.
- An active/standby model is maintained to make modifications to global data i.e. only the designated Domain0 will make the changes to global data.
- Locking mechanism across VM's will be accomplished by adhering to strict locking guidelines which are described in the Programming Guidelines Document.

### 1.1.3 Related Documents

| Document Number | Name of Document |
|---|---|
| Document 45016-00 | OSA Programming Guidelines |
| Document 46722-00 | Orion SOD and Overall Architecture |
| 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

#### 1.1.4.1.OSA Persistence Element: Open Issues

- Need to determine if User will be provided Domain0 login access to modify Linux configuration.
- Currently for Phase1, if one VM is offline in a controller, Domain0 reboots all the VMs. Need to look into OSA persistence when a VM being offline, does not result in Domain0 rebooting all the other VM's on the same controller.

## 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 |
| No PR | Support Domain0 persistent data across warm and cold reboots |

**Figure 1: Product Requirements**

### 1.2.2 Architectural Requirements

Architectural requirements for an OSA infrastructure include:

- OSA Development and Build Environment description.
- OSA Programming Guidelines for consistent programming methodology across all VMs
- Provide a method to store Domain0 persistent data, in a location other than the embedded flash since there will be a need to persist data redundantly for applications that involve controller and flash replacement scenarios.

## 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 development organization 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. 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. It will then give a summary of some of the key Linux concepts. Finally, the document will conclude with Linux OSA programming guidelines as well as general OSA guidelines for the CFW organization This evolving document will be further enhanced to denote additional guidelines, especially once development is started on the Linux operating system.

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.

The general guidelines provided in this document is meant to be refined and made more specific as the detailed design and implementation phases begin for the Linux virtual machines. 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) Research about thread-safe STL containers.

## 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 elements for the individual processes.

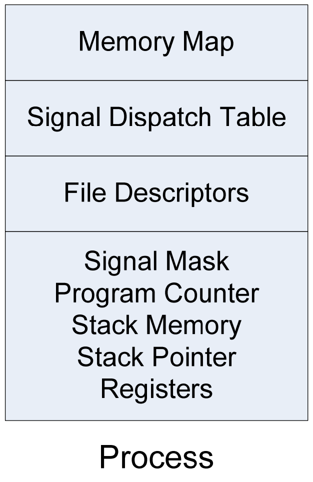

**Figure 2: Attributes of a Process**

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 signmal mask, program counter, stack memory, stack pointer, and registers. All other elements, in particular the virtual memory, are shared with the other threads of the same parent process. Since less data must be saved and restored, context switches between threads are more lightweight than those for processes. For the same reason, thread creation is more efficient than “forking” or spawning a child process.

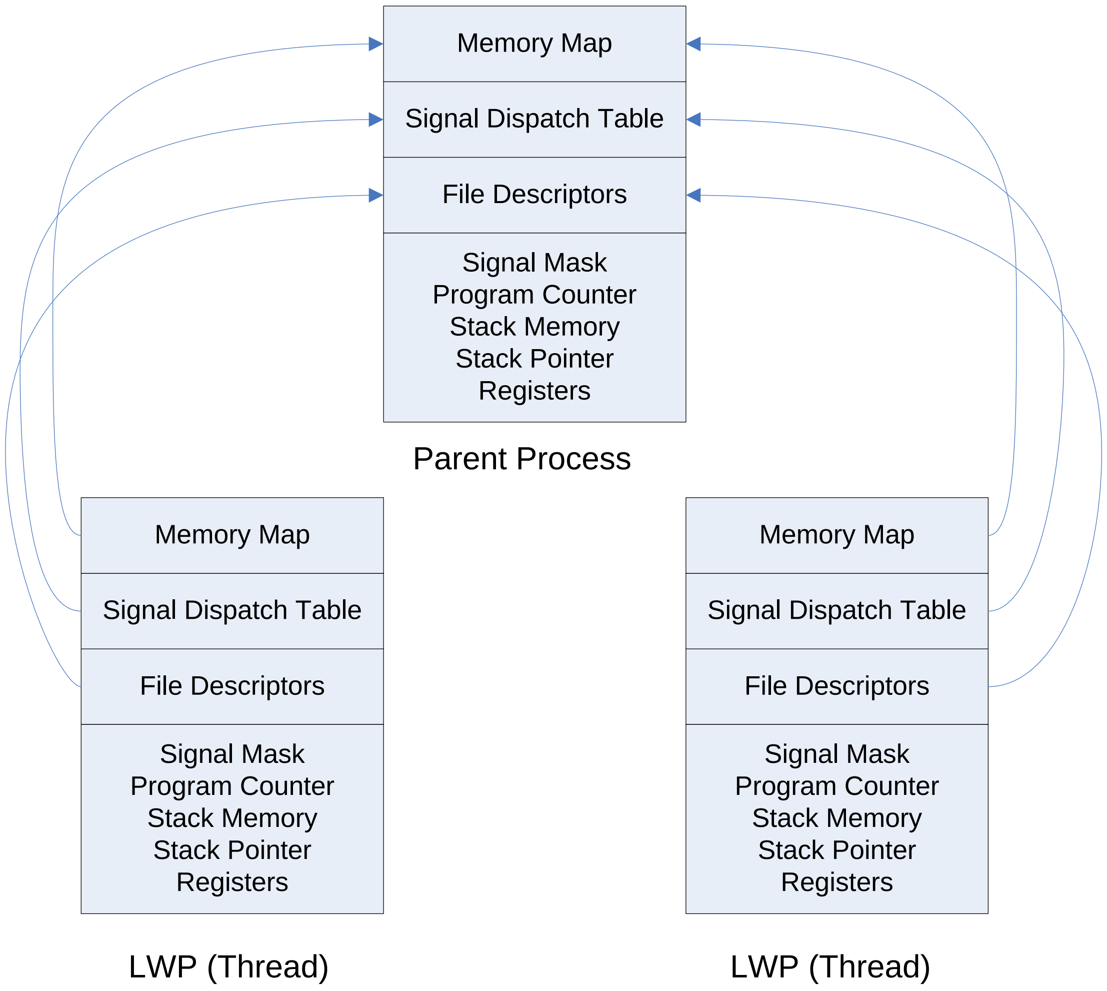

**Figure 3: Process vs Thread (LWP)**

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 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.

#### 2.2.1.1 Processes

The chief advantage of processes is the independent memory spaces, which precludes unrelated units of execution from corrupting each other's memory. Depending on how many threads exist in a process, memory accesses can be simplified. If there is a single thread, no synchronization is really required. The more threads in a process, the more likely memory accesses will need to be protected by mutex locks.

Chief strengths:

- A unique address space is provided per process. This can simplify memory usage.
- Allows for various services each with its own processes to be loaded based on system requirements. This is very flexible.

Chief weaknesses:

- A unique address space is provided per process. Routines can't be directly invoked in other processes nor can data be accessed in other processes.
- Requires inter-process communication, which adds overhead to invoking functionality between components.
- Potentially more expensive to switch execution contexts.
- Radically different than the current DPL model, requiring considerable refactoring to use existing components.

#### 2.2.1.2 Threads

A single process with many threads is relatively close to the model that is used in the DPL code with vxWorks. The chief difference is the multi-tasking model in use. In the DPL code, a non-preemptive multitasking model is used, but Linux threads use preemptive multitasking. The significance of this is the relative predictability of code execution order in the non-preemptive multitasking model used currently in DPL. With preemptive multitasking, the thread can be preempted at any time, potentially in the middle of a memory access of some sort.

Also, reentrancy is a consideration with threads. If a thread is in the middle of a call to routine such as *strtok*, and is preempted by another thread that uses the same routine, the single memory buffer in use by strtok can become corrupted. There are thread safe versions of various routines, but this is something that needs to be considered.

Chief strengths:

- All threads in a process share the same memory, allowing for various components to directly invoke methods and access memory.
- Relatively faster context switching compared to processes.
- Much like the current IOVM model, allowing components to be reused with relative ease (must adapt for preemptive multitasking, though)
- Thread creation from the kernel perspective is less expensive than creating a process because it doesn't need to allocate an address space from scratch and also it can share various other descriptors like file handlers etc.

Chief Weaknesses:

- All threads in a process share the same memory. This requires careful consideration of memory usage and access.
- Functionality is well defined in the process containing the threads, and can only be changed by changing the code. Not as flexible as multiple processes, but our requirements are well defined and don't vary, so this is weakness is mitigated.

#### 2.2.1.3 Process Example

This diagram illustrates the relationships of a few processes and how they might interact in a Linux operating environment.

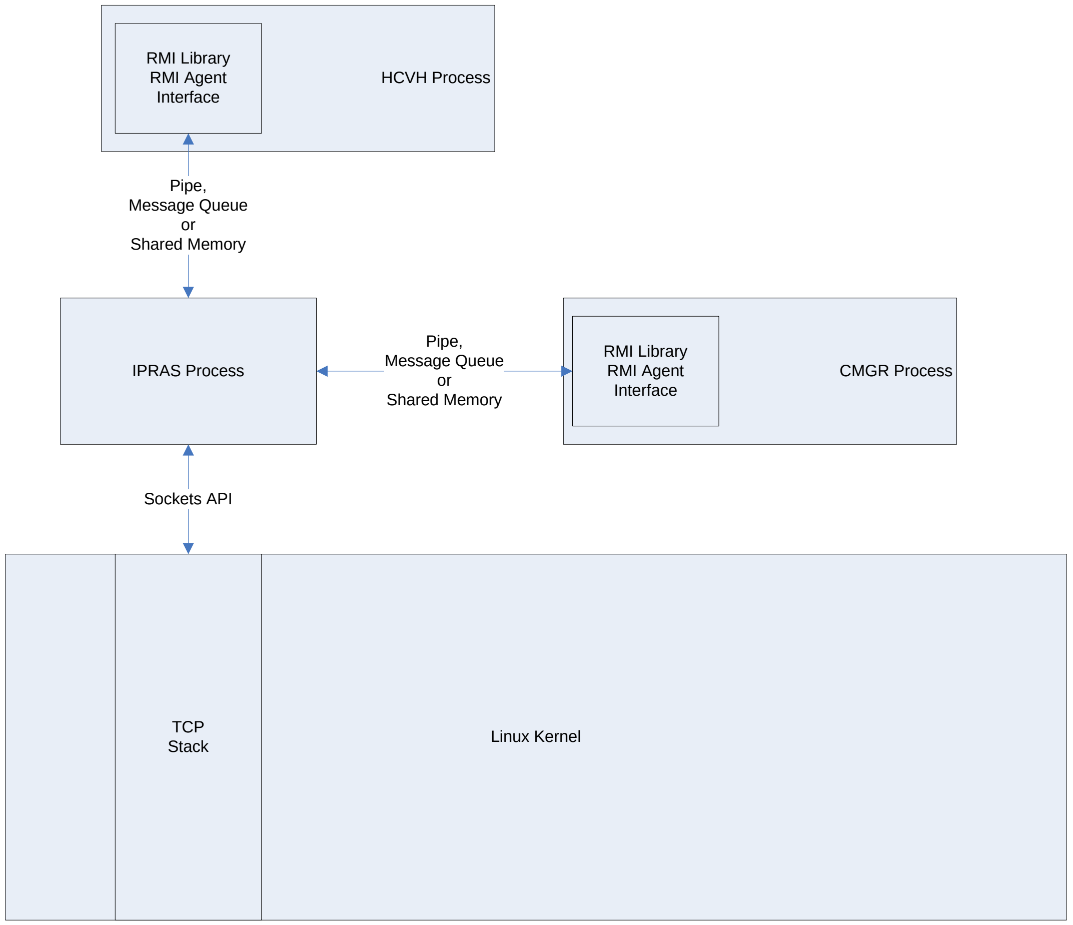

**Figure 4 - LSI Processes in a Linux Environment**

The major refactoring would be the use of inter-process communications. The inter-process communications might be a combination of any of the mentioned methods, but there would be no direct interactions with the IPRAS singleton by either the HCVH process or the CMGR process. It might be possible to provide an IPRAS proxy interface that would look like the existing interface and hide the inter-process communications behind that interface so that client could remain as unchanged as possible.

### 2.2.1.4 Thread Example 1

The first thread example illustrates receiving an IPRAS message. The message processing is performed in the IPRAS TCP Thread, much like processing would occur in a task in vxWorks.

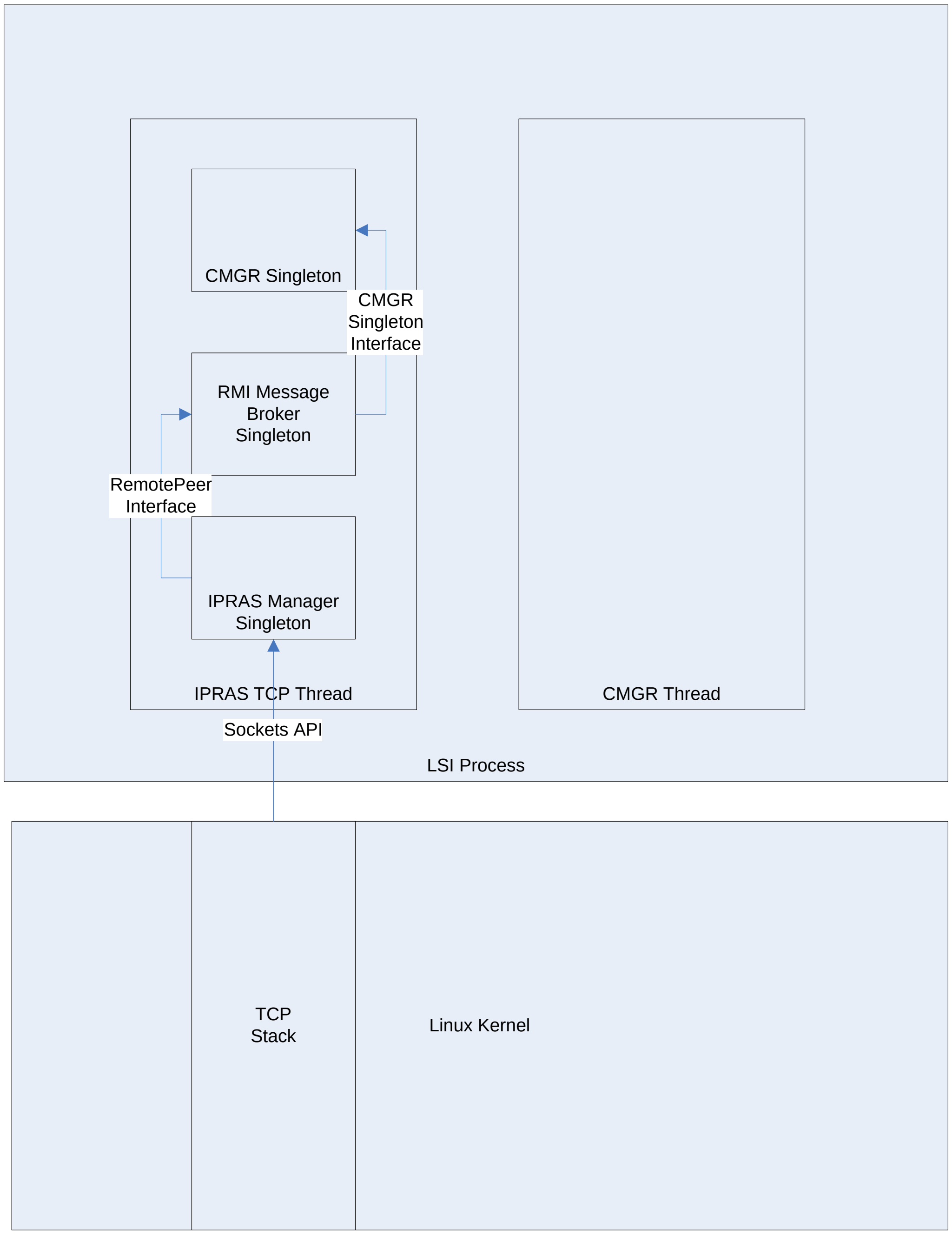

**Figure 5- Linux Thread Example 1**

The data is received by the IPRAS Manager from the TCP socket and a method is invoked on the RMI Message Broker singleton, which in turn invokes the proper method in the CMGR singleton to fulfill the request.

Since the components are all in the same process, they have the same memory space and can interact directly much like they do currently do.  The refactoring in this case would be in the form of adapting to preemptive multitasking.

### 2.2.1.5 Thread Example 2

The next example illustrates the CMGR component sending an IPRAS message. In this case, the processing is performed in the CMGR Thread.

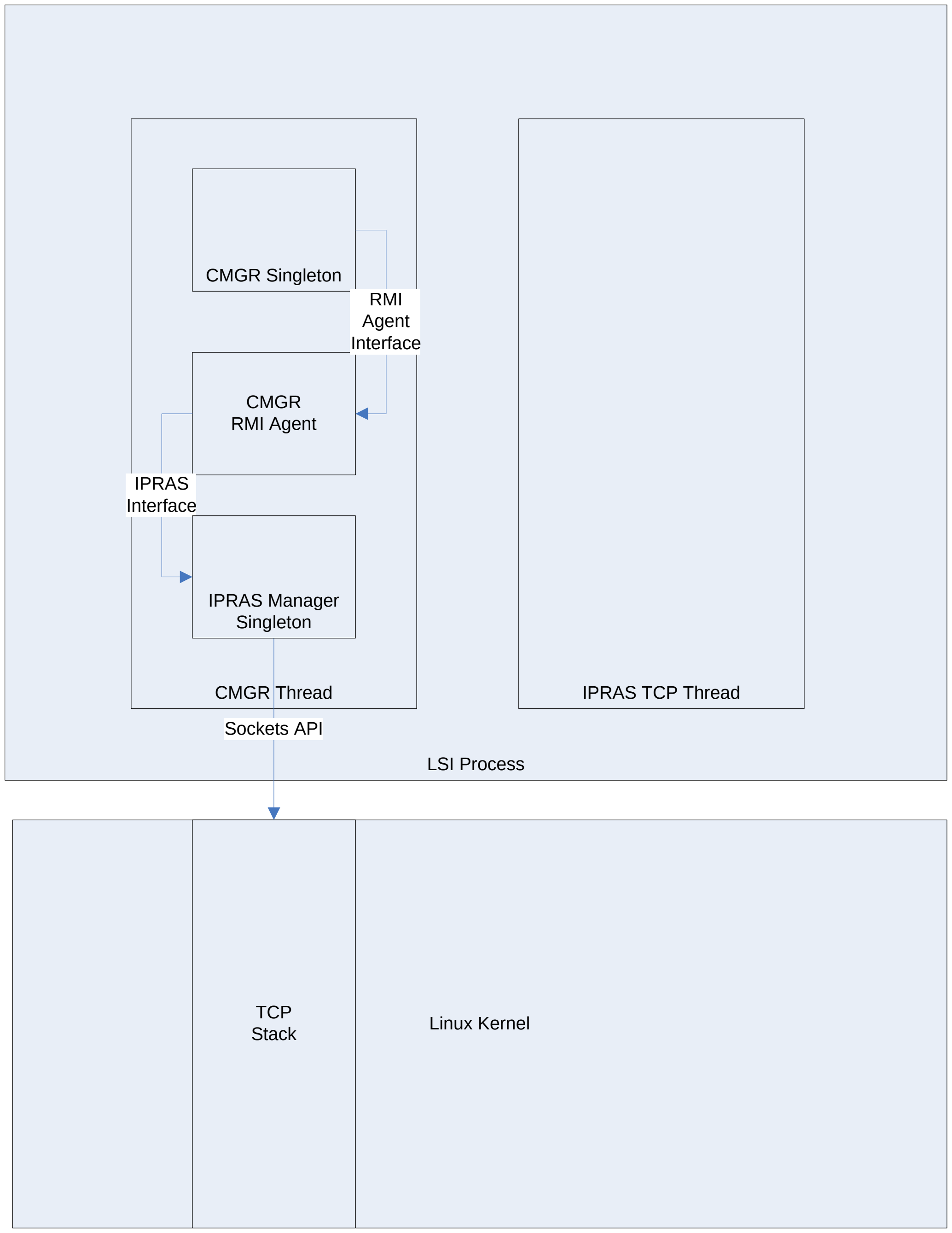

**Figure 6- Linux Thread Example 2**

In this example, CMGR initiates a request to one of its peers by invoking a method on its RMI Agent. The RMI agent calls the IPRAS Manager, and the TCP request is then made, all in the context of the CMGR thread.

### 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 from user-space 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.

High-performance user-space drivers avoid system calls in the performance paths by mapping all kernel memory (data buffers, ring buffers, etc.) and memory-mapped I/O regions into user-space at the start of day. The user-space driver can then interact directly with the associated devices.

(Note: 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. 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.

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.

### 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.

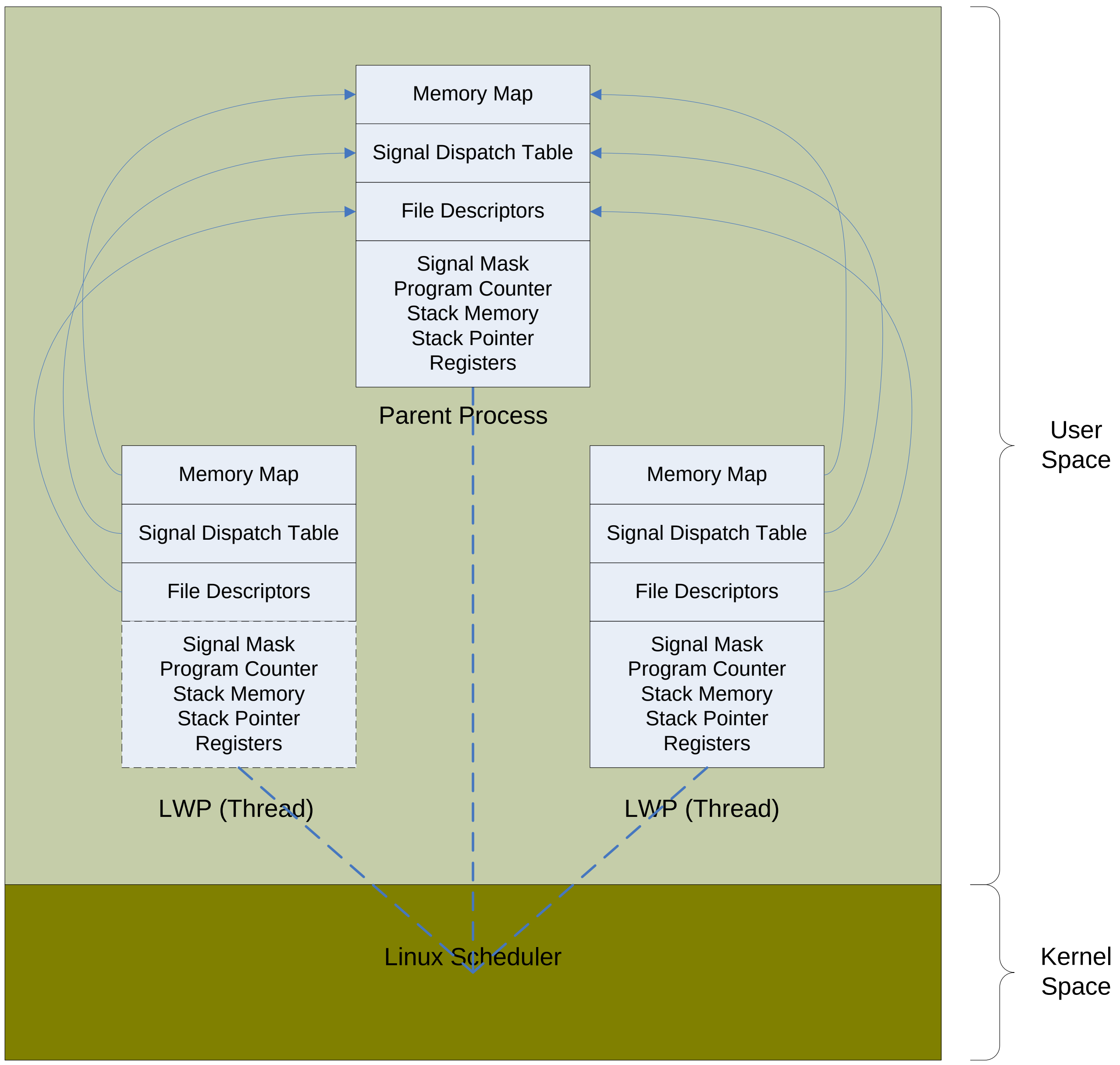

**Figure 7: Threads/Processes controlled by the Linux Scheduler**

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 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.

### 2.3.1 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.

#### 2.3.1.1 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. )

## 2.3.2 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.

### 2.3.2.1 Thread Creation

*pthread_create* creates a new thread and the following prototype:

int pthread_create(pthread_t *thread, const pthread_attr_t, *attr, void *(*start_routine, void*),void *arg);The parameters are as follows:

Pointer to pthread_t. This is used in subsequent calls to thread related routines, and is populated

by the call to pthread_create.

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. This can be NULL if default characteristics are desired.

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). This can be a pointer to

a composite data structure of some sort if multiple data items need to be passed in to the thread.

### 2.3.2.2 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

int pthread_key_create(pthread_key_t *key, void (*destructor, void*));

*pthread_setspecific* is used to change the value associated with the key

int pthread_setspecific(pthread_key_t key, const void *value);

*pthread_getspecific* is used to return the value of the associated key

void *pthread_getspecific(pthread_key_t 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_delete* 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.

int pthread_key_delete(pthread_key_t key);

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.

#### 2.3.2.3 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.

### 2.3.3 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.

#### 2.3.3.1 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.

##### 2.3.3.1.1 pthread_mutex_init

The mutex is initialized with the pthread_mutex_init function. A 0 is returned on successful completion of the function. An error valued is returned on error.

```
int   pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

The mutex argument can be initialized using the PTHREAD_MUTEX_INITIALIZER macro. In addition, depending on the needs of the application, the PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP or the PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP macros can be used. The RECURSIVE version allows the mutex to be locked several times as long as it is the same thread, and the ERRORCHECK version can be used to help find errors while debugging.

##### 2.3.3.1.2 pthread_mutex_lock

The mutex is locked using the pthread_mutex_lock function. A 0 is returned on successful completion of the function. An error valued is returned on error.

int pthread_mutex_lock(pthread_mutex_t *mutex);

The mutex argument is the one created with the call to pthread_mutex_init.

#### 2.3.3.1.3 pthread_mutex_unlock

The mutex is unlocked using the pthread_mutex_unlock function. A 0 is returned on successful completion of the function. An error valued is returned on error.

int pthread_mutex_unlock(pthread_mutex_t *mutex);\

The mutex argument is the one created with the call to pthread_mutex_init.

#### 2.3.3.1.4 pthread_mutex_trylock

The non-blocking version is pthread_mutex_trylock. A 0 is returned on successful completion of the function. An error value is returned on error.

int pthread_mutex_trylock(pthread_mutex_t *mutex);

The mutex argument is the one created with the call to pthread_mutex_init.

#### 2.3.3.1.5 Pthread mutex example

Here is an example of using pthread mutexes.

```
#include <pthread.h>
#include <stdio.h>
#include <assert.h>

pthread_mutex_t cntr_mutex = PTHREAD_MUTEX_INITIALIZER;

long protVariable = 0L;

void *myThread( void *arg )
{
  int i, ret;

  for (i = 0 ; i < 10000 ; i++) {

    ret = pthread_mutex_lock( &cntr_mutex );

    assert( ret == 0 );

    protVariable++;

    ret = pthread_mutex_unlock( &cntr_mutex );

    assert( ret == 0 );

  }

  pthread_exit( NULL );
}

#define MAX_THREADS   10
```

```
int main()
{
  int ret, i;
  pthread_t threadIds[MAX_THREADS];

  for (i = 0 ; i < MAX_THREADS ; i++) {
    ret = pthread_create( &threadIds[i], NULL, myThread, NULL );
    if (ret != 0) {
      printf( "Error creating thread %d\n", (int)threadIds[i] );
    }
  }

  for (i = 0 ; i < MAX_THREADS ; i++) {
    ret = pthread_join( threadIds[i], NULL );
    if (ret != 0) {
      printf( "Error joining thread %d\n", (int)threadIds[i] );
    }
  }

  printf( "The protected variable value is %ld\n", protVariable );

  ret = pthread_mutex_destroy( &cntr_mutex );

  if (ret != 0) {
    printf( "Couldn't destroy the mutex\n");
  }

  return 0;
}
```

### 2.3.3.2 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 be 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.

#### 2.3.3.2.1 semget

A semaphore is created using the semget function. Upon successful completion of the function, a non-negative semaphore identifier is returned. A -1 is returned on error, and errno contains error specific information.

int semget ( key_t key, int nsems, int semflg );

The key parameter is the system unique identifier and the semflg parameter controls the creation of the semaphore, or allows the semaphore to be located if the semflg is set to 0. Upon successful return from the call, an integer value is returned that can be used in subsequent calls to other functions in the Semaphore API.

The nsems parameter represents the initial count for the semaphore.

#### 2.3.3.2.2 semop

A semaphore is acquired and released using the semop function. A 0 is returned on successful completion of the routine, and a -1 is returned on error and errno contains error specific information.

```
int semop ( int semid, struct sembuf *sops, size_t nsops );
```

The semid parameter is the value returned from the semget function. The semop function is used to perform atomically an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by semid. The sops argument is a pointer to the array of semaphore-operation structures. The nsops argument is the number of such structures in the array.

Each sembuf structure contains the following members:

```
short   sem_num;   /* semaphore number */
short   sem_op;    /* semaphore operation */
short   sem_flg;   /* operation flags */
```

To acquire a semaphore, a negative value is placed in the sem_op member of the sembuf, and a positive value is placed there to release the semaphore. For now, the value of sem_num is assumed to be 0, as well as the value of sem_flg.

#### 2.3.3.2.3 semctl

You can get and set information on a semaphore as well as remove a semaphore using the semctl function. The nature of the return value is dependent on the request type.

```
int semctl ( int semid, int semnum, int cmd, ... );
```

The semid and semnum parameters are the same as outlined previously. The cmd parameter can optionally be followed by an additional argument depending on the requirements of the request.

A couple of examples are IPC_RMID to remove the semaphore and GETVAL to obtain the value of semval. The balance of these cmd values can be obtained from the semctl man page.

#### 2.3.3.2.4 Semaphore example

Here is an example of creating a semaphore.

```
#include <stdio.h>
#include <sys/sem.h>
#include "common.h"

int main()
{
  int semid;

  /* Create the semaphore with the id MY_SEM_ID */
  semid = semget( MY_SEMARRAY_ID, NUM_SEMAPHORES, 0666 | IPC_CREAT );

  if (semid != -1) {

    printf( "semacreate: Created a semaphore %d\n", semid );

  }

  return 0;
}
```

Here is an example of acquiring a semaphore.

```
#include <stdio.h>
#include <sys/sem.h>
#include <stdlib.h>
#include "common.h"

int main()
{
  int semid, i;
  struct sembuf sb[NUM_SEMAPHORES];

  /* Get the semaphore with the id MY_SEM_ID */
  semid = semget( MY_SEMARRAY_ID, NUM_SEMAPHORES, 0 );

  if (semid >= 0) {

    for (i = 0 ; i < NUM_SEMAPHORES ; i++) {
      sb[i].sem_num = i;
      sb[i].sem_op = -1;
      sb[i].sem_flg = 0;
    }
    printf( "semaacq: Attempting to acquire semaphore %d\n", semid );

    /* Acquire the semaphore */
    if (semop( semid, &sb[0], NUM_SEMAPHORES ) == -1) {

      printf("semaacq: semop failed.\n");
      exit(-1);

    }

    printf( "semaacq: Semaphore acquired %d\n", semid );

  }

  return 0;
}
```

Here is an example of releasing a semaphore.

```
#include <stdio.h>
#include <sys/sem.h>
#include <stdlib.h>
#include "common.h"

int main()
{
  int semid, i;
  struct sembuf sb[NUM_SEMAPHORES];

  for (i = 0 ; i < NUM_SEMAPHORES ; i++) {
    sb[i].sem_num = i;
    sb[i].sem_op = 1;
    sb[i].sem_flg = 0;
  }

  /* Get the semaphore with the id MY_SEM_ID */
  semid = semget( MY_SEMARRAY_ID, NUM_SEMAPHORES, 0 );

  if (semid >= 0) {

    printf( "semarel: Releasing semaphore %d\n", semid );

    /* Release the semaphore */
    if (semop( semid, &sb[0], NUM_SEMAPHORES ) == -1) {

      printf("semarel: semop failed.\n");
      exit(-1);

    }

    printf( "semarel: Semaphore released %d\n", semid );

  }

  return 0;
}
```

### 2.3.3.3 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.

#### 2.3.3.3.1 pthread_cond_init

A condition variable is initialized using the pthread_cond_init function.  A 0 is returned on successful completion of the function.  An error valued is returned on error.

Int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);

The pthread_cond_t pointed to by the cond parameter is initialized with the attributes specified by the attr parameter.  If attr is NULL, default attributes are used.  I don't find much about the attr parameter.

The cond parameter can be initialized with the PTHREAD_COND_INITIALIZER macro rather than calling pthread_cond_init.

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

#### 2.3.3.3.2 pthread_cond_wait

The pthread_cond_wait is used to make a thread wait for a specific condition to occur.  A 0 is returned on successful completion of the function.  An error valued is returned on error.

int pthread_cond_wait(pthread_cond_t *cond,  pthread_mutex_t *mutex);

The cond parameter is the one initialized as previously described and the mutex parameter is a mutex that was initialized as described in the section on mutexs.

The thread can be woken up when the condition has been met by calling the pthread_cond_signal or pthread_cond_broadcast. A 0 is returned on successful completion of the function. An error valued is returned on error.

### 2.3.3.3.3 pthread_condition_signal – pthread_cond_broadcast

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);

The signal version wakes up a single thread, and the broadcast wakes up any tasks that are waiting on the same signal, although they have to do so one at a time since the condition is protected by a mutex.

### 2.3.3.3.4 pthread_condition_destroy

When there is no longer need for the condition variable, the pthread_cond_destroy function is used. A 0 is returned on successful completion of the function. An error valued is returned on error.

int pthread_cond_destroy(pthread_cond_t *);

This function simply sets the values in the pthread_cond_t to uninitialized values. It can then be reinitialized if desired.

### 2.3.3.3.5 Pthread condition variable example

Here is an example of using pthread condition variables. A single producer thread wakes up the multiple consumer threads using condition variables.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

pthread_mutex_t cond_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condition = PTHREAD_COND_INITIALIZER;

int workCount = 0;


void *producerThread( void *arg )
{
  int i, j, ret;
  double result=0.0;

  printf("Producer started\n");

  for ( i = 0 ; i < 30 ; i++ ) {

    ret = pthread_mutex_lock( &cond_mutex );
    if (ret == 0) {
      printf( "Producer: Creating work (%d)\n", workCount );
      workCount++;
      pthread_cond_broadcast( &condition );
      pthread_mutex_unlock( &cond_mutex );
    } else {
      assert( 0 );
    }

    for ( j = 0 ; j < 60000 ; j++ ) {
      result = result + (double)random();
    }

  }

  printf("Producer finished\n");

  pthread_exit( NULL );
}


void *consumerThread( void *arg )
{
  int ret;

  printf( "Consumer %x: Started\n", (unsigned int)pthread_self() );

  pthread_detach( pthread_self() );

  while( 1 ) {

    assert( pthread_mutex_lock( &cond_mutex ) == 0 );
    ret = pthread_cond_wait( &condition, &cond_mutex );
    if (ret == 0) {
      if (workCount) {
        workCount--;
        printf( "Consumer %x: Performed work (%d)\n",
                  (unsigned int)pthread_self(), workCount );
      }
      assert( pthread_mutex_unlock( &cond_mutex ) == 0 );
    }

  }

  printf( "Consumer %x: Finished\n", (unsigned int)pthread_self() );

  pthread_exit( NULL );
}


#define MAX_CONSUMERS  10

int main()
{
  int i;
  pthread_t consumers[MAX_CONSUMERS];
  pthread_t producer;

  /* Spawn the consumer thread */
  for ( i = 0 ; i < MAX_CONSUMERS ; i++ ) {
    pthread_create( &consumers[i], NULL, consumerThread, NULL );
  }

  /* Spawn the single producer thread */
  pthread_create( &producer, NULL, producerThread, NULL );

  pthread_join( producer, NULL );

  while ((workCount > 0));

  /* Cancel and join the consumer threads */
  for ( i = 0 ; i < MAX_CONSUMERS ; i++ ) {
    pthread_cancel( consumers[i] );
  }

  pthread_mutex_destroy( &cond_mutex );
  pthread_cond_destroy( &condition );
  return 0;
}
```

### 2.3.4 Communication between processes (IPC)

In Linux, as previously described, each process has its own memory space and components that reside in separate processes can't directly interact. They must rely on interprocess communications (IPC) techniques. These techniques include pipes, message queues, semaphores, shared memory and sockets.

Signals are undesirable 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.

#### 2.3.4.1 Pipes

Pipes provide a FIFO method of providing one way communications between two processes. They basically connect STDOUT with STDIN for the two processes. Pipes can be either named or anonymous. Anonymous pipes are used between a parent and child process and the pipe descriptor is passed in as part of the child process creation. Named pipes are part of the filesystem and can be opened by any process. Being part of the file system, they can be manipulated like any file descriptor.

##### 2.3.4.1.1 pipe

Anonymous pipes are creatingcreated using the pipe function. A 0 is returned on successful completion of the routine, and a -1 is returned on error and errno contains error specific information.

int pipe( int fds[2] );

Upon successful return from the call, the fds array contains two active file descriptors that can be used to access the pipe for reading and writing.

##### 2.3.4.1.2 mkfifo

Named pipe are created using the mkfifo function. A 0 is returned on successful completion of the routine, and a -1 is returned on error and errno contains error specific information.

int mkfifo( const char *pathname, mode_t mode );

This creates a named pipe located at pathname. Pathname can then be used to open the pipe using the fopen function.

##### 2.3.4.1.3 fopen

Pathname can then be used to open the pipe using the fopen function. Upon successful completion of the function, a pointer to the object controlling the file stream is returned. A NULL pointer is returned in the case of a failure, and errno contains error specific information.

FILE* fopen( const char* pathname, const char* mode );

To use a pipe, one process must open the pipe for writing. The pipe cannot be opened for reading until a process had opened it for writing first. The pipe has visibility to the entire process, so any threads in the process could access the pipe.

Using pipes is like writing and reading a file. Here is an example of a provider and a consumer utilizing a pipe for PPC.

#### 2.3.4.1.4 Pipe example

Here is a simple example of a pair of processes using a pipe.

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <wait.h>

#define MAX_LINE      80

int main()
{
  int thePipe[2], ret;
  char buf[MAX_LINE+1];
  const char *testbuf={"a test string."};

  if ( pipe( thePipe ) == 0 ) {

    if (fork() == 0) {

      ret = read( thePipe[0], buf, MAX_LINE );
      buf[ret] = 0;
      printf( "Child read %s\n", buf );

    } else {

      ret = write( thePipe[1], testbuf, strlen(testbuf) );
      ret = wait( NULL );

    }

  }
  return 0;
}
```

### 2.3.4.2 Message Queues

Message queues provide another method of PPC. Messages can be sent and retrieved from the message queue, much like with the pipe. The data in the pipe is somewhat arbitrary, though but the messages in a message queue are actual discrete packets of data that include the type of the message specified by a long word followed by the message data itself. Message queues also allow multiple processes to read and write to a single message queue. The message queue is located using a system wide unique identifier that needs to be known to all of the processes that are going to use it.

Below are some attributes of message queues:

- Limited to local events only (same virtual machine and cannot use to communicate between kernel and user space)
- Number of messages needs to be pre-defined
- Uses system calls and copy to deliver messages (performance penalty)

#### 2.3.4.2.1 msgget

A message queue is created using the msgget function. Upon successful completion of the function, a non-negative message queue identifier is returned. A -1 is returned on failure and errno contains error specific information.

```
int msgget ( key_t key, int msgflg );
```

The key parameter is the system unique identifier and the msgflg parameter controls the creation of the message queue, or allows the message queue to be located if the msgflg is set to 0. Upon successful return from the call, an integer value is returned that can be used in subsequent calls to other functions in the message queue API.

#### 2.3.4.2.2 msgctl

The message queue is created with a default size of 16KB, but that, among other things can be modified using the msgctl function. Information about the message queue can also be retrieved using this function. A 0 is returned on successful completion of the routine, and a -1 is returned on error and errno contains error specific information.

```
int msgctl ( int msqid, int cmd, struct msqid_ds *buf );
```

The msqid is the value returned from the msgget call and cmd is one of the following defined values.

```
IPC_ALLOC    /* entry currently allocated */
IPC_CREAT    /* create entry if key doesn't exist */
IPC_EXCL     /* fail if key exists */
IPC_NOWAIT   /* error if request must wait */
IPC_PRIVATE  /* private key */
```

| | |
|---|---|
| IPC_RMID | /* remove identifier */ |
| IPC_SET | /* set options */ |
| IPC_STAT | /* get options */ |
| IPC_O_RMID | /* remove identifier */ |
| IPC_O_SET | /* set options */ |
| IPC_O_STAT | /* get options */ |

The buf parameter is a structure of type msqid_ds, which is a complex structure used to configure and retrieve information about the message queue, and is beyond the scope of this introductory discussion.

## 2.3.4.2.3 msgsnd

Messages are sent using the msgsnd function. A 0 is returned on successful completion of the routine, and a -1 is returned on error and errno contains error specific information.

```
int msgsnd ( int msqid, const void* message, size_t size , int msgflg );
```

The msqid is the value returned from the msgget call, message is a pointer to the memory containing the message, size is the length of the message in bytes and msgflg can be used to control the function in various ways.

## 2.3.4.2.4 msgrcv

Messages are retrieved from the message queue using the msggetmsgrcv function. Upon successful completion of the routine, the actual number of byte placed in the buffer are returned, and a -1 is returned on error and errno contains error specific information.

```
ssize_t msgrcv ( int msqid, void *msgp, size_t msgsz, long int msgtyp, int msgflg );
```

The msqid is the value returned from the msgget call, message is a pointer to the memory for the message to be retrieved into, size is the size of the buffer in bytes and msgflg can be used to control the function in various ways.

## 2.3.4.2.5 Message queue example

Here is an example of sending a message.

```
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>
#include "common.h"

int main()
{
  MY_TYPE_T myObject;
  int qid, ret;

  /* Get the queue ID for the existing queue */
  qid = msgget( MY_MQ_ID, 0 );

  if (qid > 0) {

    /* Create our message with a message queue type of 1 */
    myObject.type = 1L;
    myObject.fval = 128.256;
    myObject.uival = 512;
    strncpy( myObject.strval, "This is a test.\n", MAX_LINE );

    /* Send the message to the queue defined by the queue ID */
    ret = msgsnd( qid, (struct msgbuf *)&myObject,
                   sizeof(MY_TYPE_T), 0 );

    if (ret != -1) {

      printf( "Message successfully sent to queue %d\n", qid );

    }

  }

  return 0;
}
```

Here is an example of retrieving a message.

```
#include <sys/msg.h>
#include <stdio.h>
#include "common.h"

int main()
{
  MY_TYPE_T myObject;
  int qid, ret;

  qid = msgget( MY_MQ_ID, 0 );

  if (qid >= 0) {
    ret = msgrcv( qid, (struct msgbuf *)&myObject,
                   sizeof(MY_TYPE_T), 1, 0 );

    if (ret != -1) {

      printf( "Message Type: %ld\n", myObject.type );
      printf( "Float Value:  %f\n", myObject.fval );
      printf( "Uint Value:   %d\n", myObject.uival );
      printf( "String Value: %s\n", myObject.strval );

    }

  }

  return 0;
}
```

## 2.3.4.3 Shared Memory

Since Linux processes each have their own private memory space, it would be useful to be able to share memory between processes for some applications. The Shared Memory API allows for this kind of access to memory that can be accessed by multiple processes. The Shared Memory API provides only the access to the memory, and any algorithms that are required to control access to the memory or synchronize updates must be implemented by the applications using the shared memory. The shared memory is located using a system wide unique identifier that needs to be known to all of the processes that are going to use it.

**Below are some attributes of shared memory and semaphores**

- Limited to local events only
- Number of messages needs to be pre-defined based on shared memory size
- Can be extended to kernel mode clients if needed but not a very standard practice
- Most efficient in terms of performance but lot of synchronization

## 2.3.4.3.1 shmget

A shared memory segment is created using the shmget function. Upon successful completion of the function, a non-negative shared memory identifier is returned. A -1 is returned on failure and errno contains error specific information.

int shmget ( key_t key, size_t size, int shmflg );

The key parameter is the system unique identifier and the shmflg parameter controls the creation of the shared memory segment, or allows the shared memory to be located if the shmflg is set to 0. Upon successful return from the call, an integer value is returned that can be used in subsequent calls to other functions in the Shared Memory API.

The size parameter specifies the desired size of the memory segment, and must be a multiple of the page size in use in the system, which is generally 4KB, but can be determined with the getpagesize function.

If the size and shmflg parameters are 0, the shmget function is used to locate the shared memory segment.

##### 2.3.4.3.2 shmat

To use the shared memory segment, it must be attached and mapped to the local process memory space.  This is accomplished with the shmat functions.  Upon successful completion of the routine, a pointer to the start of the attached shared memory data segment is returned.  A -1 is returned on failure and errno contains error specific information.

```
void *shmat ( int shmid, const void *shmaddr, int shmflg );
```

The shmid parameter is the value returned from the shmget function.  If the shmaddr argument is 0, the memory segment is attached to the first available suitable address in the memory space of the process.  The shmaddr parameter can also be used to locate the memory location where the shared segment will be attached.

##### 2.3.4.3.3 shmdt

The shared memory segment can also be detached using the shmdt function.  A 0 is returned on successful completion of the routine, and a -1 is returned on error and errno contains error specific information.

```
int shmdt( const void *addr );
```

The addr parameter specifies the pointer to the location where the shared memory was mapped with the call to shmat.

Information about the shared memory segment can be obtained, and some parameters set using the shmctl function.

```
int shmctl ( int shmid, int cmd, struct shmid_ds *buf );
```

The shmid is the value returned from the shmget call and cmd is one of the following defined values.

```
IPC_ALLOC     /* entry currently allocated */
IPC_CREAT     /* create entry if key doesn't exist */
IPC_EXCL      /* fail if key exists */
IPC_NOWAIT    /* error if request must wait */
IPC_PRIVATE   /* private key */
IPC_RMID      /* remove identifier */
IPC_SET       /* set options */
IPC_STAT      /* get options */
```

| | |
|---|---|
| IPC_O_RMID | /* remove identifier */ |
| IPC_O_SET | /* set options */ |
| IPC_O_STAT | /* get options */ |

The buf parameter is a structure of type shmid_ds, which is a complex structure used to configure and retrieve information about the message queue, and is beyond the scope of this introductory discussion.

### 2.3.4.3.4 Shared memory example

Here is an example of creating a shared memory segment.

```
#include <stdio.h>
#include <sys/shm.h>
#include "common.h"

int main()
{
  int shmid;

  /* Create the shared memory segment using MY_SHM_ID */
  shmid = shmget( MY_SHM_ID, 4096, 0666 | IPC_CREAT );

  if ( shmid >= 0 ) {

    printf( "Created a shared memory segment %d\n", shmid );

  }

  return 0;
}
```

Here is an example of attaching to a shared memory segment.

```
#include <stdio.h>
#include <sys/shm.h>
#include <errno.h>
#include "common.h"

int main()
{
  int shmid, ret;
  void *mem;

  /* Get the shared memory segment using MY_SHM_ID */
  shmid = shmget( MY_SHM_ID, 0, 0 );

  if ( shmid >= 0 ) {

    mem = shmat( shmid, (const void *)0, 0 );

    if ( (int)mem != -1 ) {
      printf( "Shared memory was attached in our address space at %p\n", mem );

      ret = shmdt( mem );

      if (ret == 0) {
        printf("Successfully detached memory\n");
      } else {
        printf("Memory detached Failed (%d)\n", errno);
      }
    } else {
      printf( "shmat failed (%d)\n", errno );
    }
  } else {
    printf( "Shared memory segment not found.\n" );
  }
  return 0;
}
```

Here is an example of writing to a shared memory segment.

```
#include <stdio.h>
#include <sys/shm.h>
```

```
#include <string.h>
#include "common.h"

int main()
{
  int shmid, ret;
  void *mem;

  /* Get the shared memory segment using MY_SHM_ID */
  shmid = shmget( MY_SHM_ID, 0, 0 );

  mem = shmat( shmid, (const void *)0, 0 );

  strcpy( (char *)mem, "This is a test string.\n" );

  ret = shmdt( mem );

  return 0;
}
```

Here is an example of reading from a shared memory segment.

```
#include <stdio.h>
#include <sys/shm.h>
#include <string.h>
#include "common.h"

int main()
{
  int shmid, ret;
  void *mem;

  /* Get the shared memory segment using MY_SHM_ID */
  shmid = shmget( MY_SHM_ID, 0, 0 );

  mem = shmat( shmid, (const void *)0, 0 );

  printf( "%s", (char *)mem );

  ret = shmdt( mem );

  return 0;
}
```

#### 2.3.4.4 Semaphores

Semaphores aren't so much process to process communications as they are a means to provide synchronization between processes. A semaphore can be acquired and released. There are two types of semaphores, binary and counting. The binary semaphore can be acquired one time so it can be used to restrict access to a single resource to a single user at any given time, such as a shared memory segment. The counting semaphore is similar in nature, but starts with a given count, and each time the semaphore is acquired, the count is decremented. Once the count reaches zero, the semaphore can no longer be acquired. Each release of the semaphore increments that count, which allows the next acquire request to succeed. This allows control over a set of resources.

A semaphore is created using the semget function. A semaphore is acquired and released using the semop function. You can get and set information on a semaphore as well as remove a semaphore using the semctl function.

#### 2.3.4.5 Sockets

Another method for process to process communications is the use of sockets. The Sockets API provides a means to perform point to point communications across any network, from local to the internet. It can be used to communicate between processes on the same host or virtual machine or to communicate with a remote host or virtual machine. Sockets provides a connection based paradigm of operation, and generally a client connects to a server and makes requests for data or services to be delivered or performed by the server. Using the Sockets API is somewhat more involved than the other PPC methods described here, but it also provides the widest range of applications.

Here are some attributes of Sockets

- Most scalable can be expanded to distributed architecture across controllers and VMs if needed
- Can be extended to even kernel space if needed
- Event driven, numbers of events need not be pre-defined
- Uses system calls and copies (Performance penalty)

### 2.3.5 Linux Libraries

Libraries in Linux can come in three forms; static, shared and dynamic libraries. Each has its own set of strengths and weaknesses.

#### 2.3.5.1 Static Libraries

Static libraries are linked during compile time into the image of the executable. Since that is the case, the library is not required to exist on the system on which the executable will run. If the library is used in multiple executables, each image will contain that library, so the library is represented multiple times both on disk and in memory at the time of execution. This guarantees that the library will be the correct one at runtime, and issues of library incompatibility are avoided. Also, no dynamic linking is required at runtime, which may hasten the load process.

#### 2.3.5.2 Shared Libraries

Shared libraries are loaded at runtime from an image separate from the image of the executable that is using the library. If the library has already been loaded, the same image will be used by the current executable, and only dynamic linking will be required. Each instance of an executable that uses the library will share the code contained in the library, but the data associated with the library will be unique per process. This means that both disk space and memory usage are minimized since only one copy of the library need reside on disk and in memory. This requires runtime linking which may cause loading and starting the executable to take long than if it were using a static library. It also means that incompatible libraries can be accessed, causing unexpected runtime behavior of the executable.

#### 2.3.5.3 Dynamic Libraries

Whereas shared libraries are loaded at the start of runtime, dynamic libraries are shared libraries that are only loaded when their use is required. They can also be unloaded when they are no longer needed. This can optimize memory use if it seems likely that the usefulness of the library will be limited during the runtime of the executable. Rather than the library being automatically loaded and linked at startup, the application itself determines when the library is needed and uses the DL API to load the library and link local pointers to functions of interest in the dynamically linked library. The dynamic library shares the same memory and disk advantages of the shared library.

#### 2.3.5.4 Library Conclusion

It seems unlikely that we would benefit from the use of dynamic libraries. Most likely the libraries will be required for the ongoing functionality of the array. Depending on the size of the library and the number of unique executables that are going to use the library, shared libraries may be of some benefit. We are not going to allow installing modules at anywhere near the granularity of a library, so we would have no issues with incompatible libraries. The startup penalty (if there is one) would have to be analyzed to make sure that wouldn't unnecessarily slow our SOD processing, which we certainly want to keep as short as possible.

### 2.3.6 Singletons

A design pattern that is in widespread use in the LSI DPL code is the singleton pattern. The singleton has some drawbacks when used in the context of a multiple process Linux system. Since each process has its own memory space, the scope of a singleton is basically confined to that memory space. If a component in one process wants to access a singleton in another process, it cannot do so directly. This is a relatively limiting factor for the singleton design pattern in a multi-process environment, unless you define a singleton as the one and only instance of an object in any given process/memory space. It would be possible to group all of the clients of the singleton and the singleton itself into a single process, and in that way the singleton could be used just as it is today. Any clients outside of that process would need to use some kind of singleton proxy to access the singleton in another process. This would entail creating a library that provided an interface that looks like the interface of the singleton itself, but would use PPC methods to interact with the actual singleton in another process. This would be conceptually possible to do.

The best approach is to group all of the clients of the singleton as well as the singleton into a single process so that the well known usage of the singleton can be utilized.

(Note: The singleton template will be copied over from DPL code-base to be used in Domain0 and modified as appropriate.)

## 2.4 Linux OSA 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 or FPGA 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<br>Also, debugging tools are in general better developed and supported in user space. | High latency/context switch when communicating with system calls |
| Open source issues are manageable | |

**Figure 8: Advantages/Disadvantages of User-Space code**

### 2.4.1 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 2.5.1 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.

Are there requirements for 64-bit compatibility? Probably most important is to note that wherever pointers are coerced to integers and vice-versa (this shouldn't be common, but it does happen), "long" should be used instead of "int", as sizeof(long) is the same as sizeof(void*) in Linux. See http://en.wikipedia.org/wiki/64-bit#64-bit_data_models.

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.

### 2.4.2 User Space Guidelines

The code should make use of POSIX threads with VKI abstractions – this will enable the code to be SMP-compliant Furthermore, all user-space code must be written in C++ and there will be a need for scripts  as well (see below for scripting guidelines).

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 also be utilized but given the  context switch overhead and the appropriate IPC mechanisms involved, this should only be used when appropriate. Note that this direction will most likely evolve over time as more design and functionality matures for the Linux VMs. (Example: CFW portion of Domain0 will likely contain a handful of processes – probably around 4-5 total.)

The disadvantage of having just one process with multiple threads is that all threads 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.

The default time-sliced, preemptive scheduling policy will be utilized and this direction can be re-examined once the OSA project progresses.

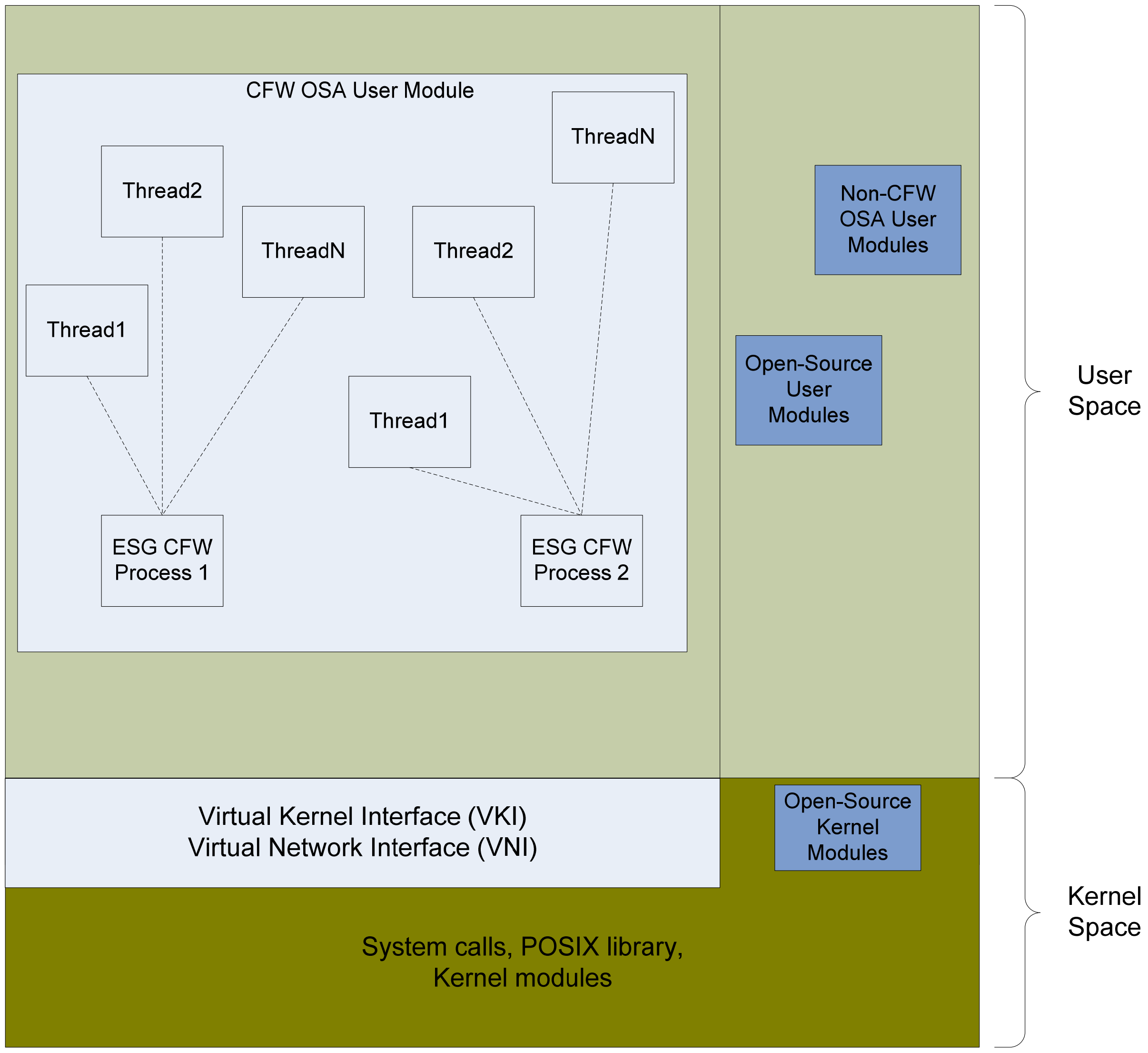

**Figure 9: CFW OSA Environment for the Linux Virtual Machines**

### 2.4.2.1 Domain0 Process – Component distribution

There are various ways to organize components into logical groupings in processes. In general, it makes good sense to group components together that have a high level of affinity and interact with each other on a regular basis. It is possible to create one large process that contains all of the required LSI components, and this provides the most straight forward method for the components to interact. Even in this case, multiple processes are necessary to meet our requirements. The components would be grouped as indicated in this diagram.

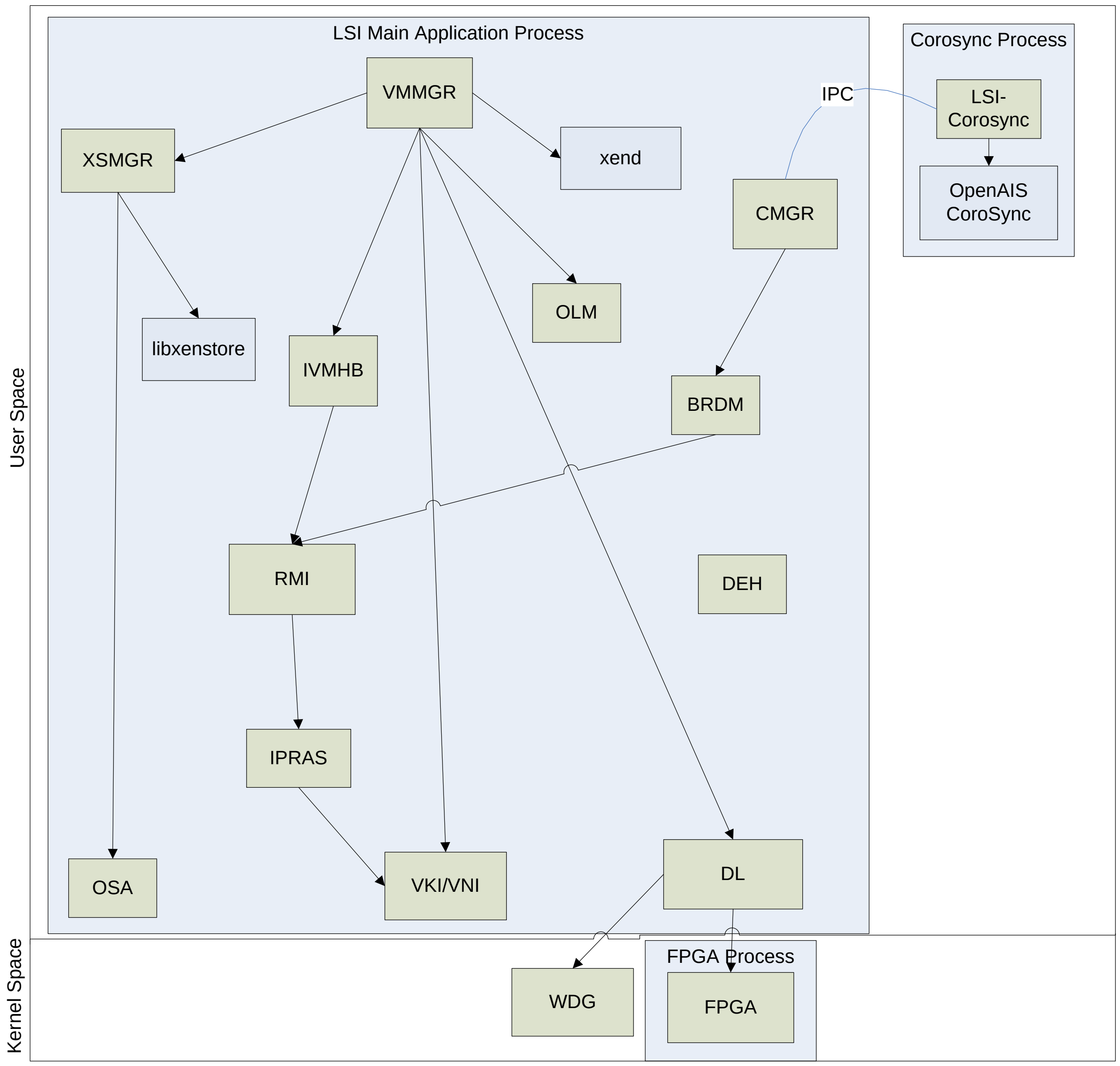

**Figure 10 - Single Monolithic LSI process**

Each component could have one or more threads, or the component might be called in the context of the thread of one of the other components, as most likely would be the case with RMI. The IPRAS component would have one or more threads to manage incoming TCP/IP traffic, and CMGR and VMGR would likely have their own threads as well. This model would require the least in the way of providing “proxy” methods for the components to interact with each other, as singletons could be directly accessed just as they are today. Additional work would have to be done to keep memory accesses single threaded.

Although it is not explicitly called out, with multiple components being initialized we would need something like a SOD component to instantiate and initialize all of the components in the proper sequence.

Another approach would be to subdivide the components into multiple processes, each containing its own IPRAS singleton. This would simplify interactions between RMI and IPRAS, and the scope of the singleton would be the process rather than system wide. This would required each IPRAS to have a unique socket to listen for requests, and the remote IPRAS or RMI would have to know the proper mapping, This would subdivide the components in potentially more closely aligned groupings and minimize the potential for inter-component memory corruption, That organization would look something like this,

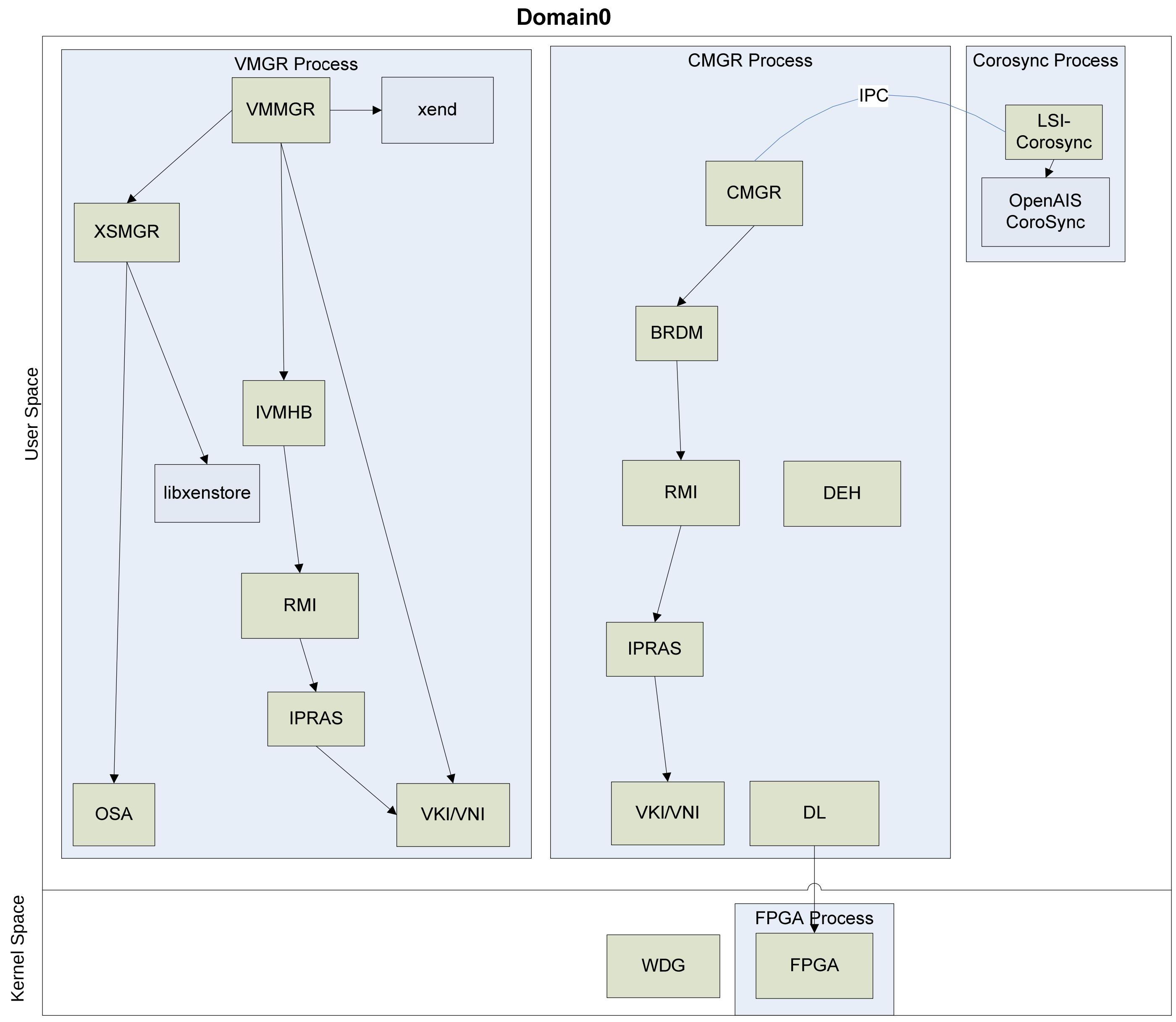

**Figure 11 - Multiple Process - Multiple IPRAS**

Another approach is to contain IPRAS as an actual system wide singleton in its own processes that could be utilized by components in any of the other processes. This would simplify the job of remote IPRAS and RMI as the same listening socket would be used in all virtual machines. This would require considerable modification to the interaction between RMI and IPRAS, as inter process communication would have to be used to both send requests and to receive status and data from the remote virtual machines. That organization would look something like this.

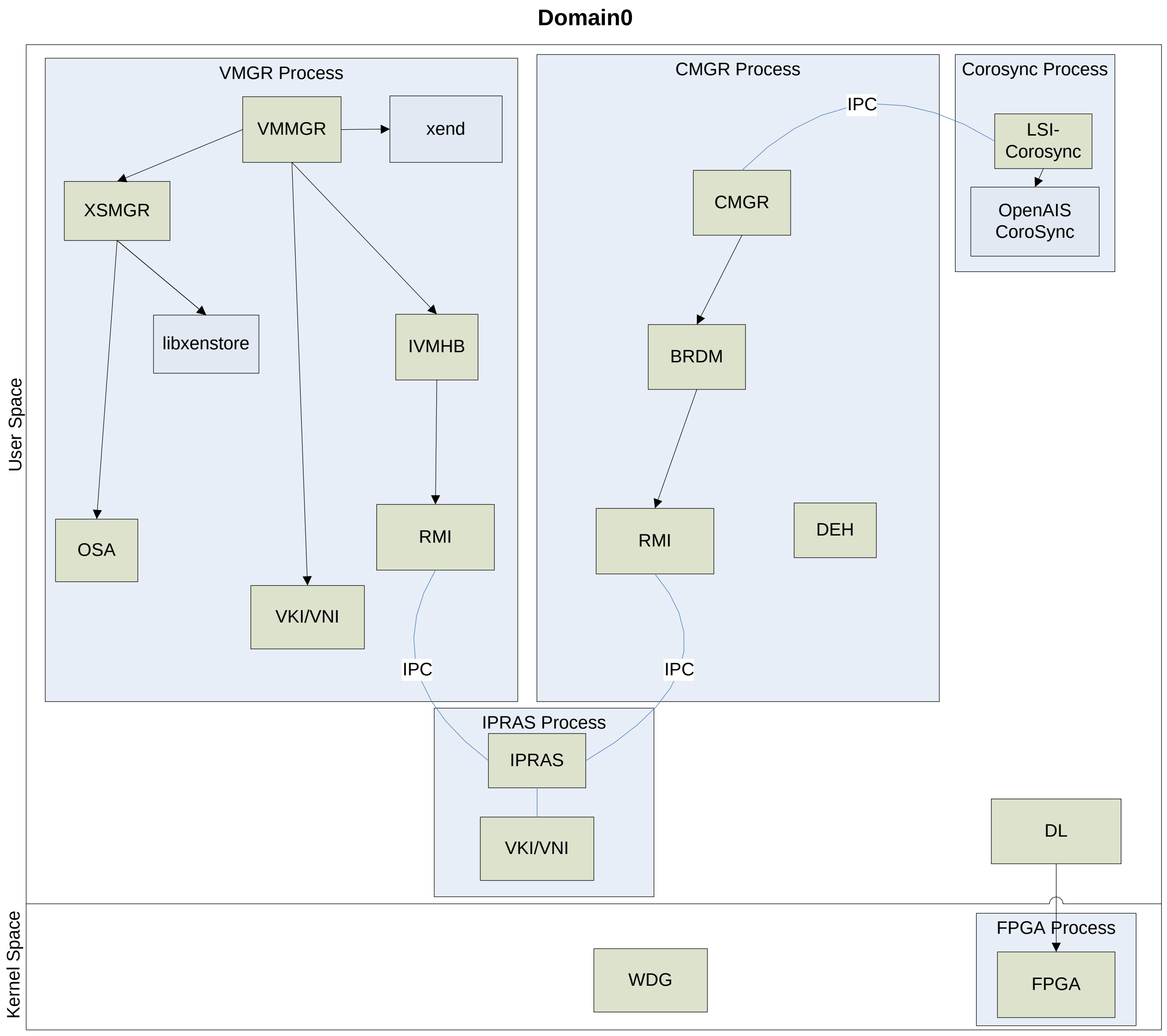

**Figure 12 - Multiple Processes - Single IPRAS**

These examples move from most similar to our current component interactions to most dissimilar to our current component interactions.  Even then, in the monolithic process example, there will have to be considerable refactoring to adapt to the Linux operating environment.

## 2.4.3 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.

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.

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

### 2.4.4 Scripting Guidelines

Here are the guidelines on scripting:

1) Bash shell scripts should be used for OS related scripts, such as startup scripts

2) Xen uses Python extensively and xend commands are implemented in Python. Therefore, any Xen related scripting should use Python.

3) Perl scripts are currently used to auto-generate DOMI/RMI agents at compile-time and this should stay as such. One of the benefits of using Perl is that it is readily usable across OSes with no dependency on Xen/Linux. Since Perl features OS independence and is already used at compile-time, it is leading candidate for non-OS, non-Xen related scripting.

## 2.5 General OSA Guidelines

### 2.5.1 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. Are the MPT drivers an example of this? If so, it might be good to cite them as such.
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”)

## 2.5.2 Hierarchical Locking

With the introduction of DRM (Domain0 Record Manager), we are starting to see cases where there is locking across VMs.

For example:

Suppose, there is a change in Domain0 that needs to update its in-memory structure as well as update the DBM database, we are doing the following:

Domain0: Acquire “Lock_Domain0”

IOVM: Acquire “MasterTransaction”

What this is implying is that “Lock_Domain0” must be acquired BEFORE “MasterTransaction” in ALL cases. What we all know is that if this ordering changes, you will readily end up in a deadlock situation.

For example, if something started in IOVM that took a MTxn and we had to talk to Domain0 for it, which in-turn had to talk to IOVM for persistence, we would have something like below:

IOVM: Acquire “MasterTransaction”

Domain0: Acquire “Lock_Domain0”

IOVM: Acquire “MasterTransaction”

This is a deadlock waiting to happen *unless* MasterTransaction is acquired in the same “task context”. That is not guaranteed *at all* b/c IPRAS has no control as to which task actually wakes up to honor socket communication. BTW, it’s pretty easy to come up with an example that results in deadlock with acquiring lock once on each VM.

For example, use-case1 is doing:

IOVM: Acquire “MasterTransaction”

Domain0: Acquire “Lock_Domain0”

and at the same time, use-case2 is doing:

Domain0: Acquire “Lock_Domain0”

IOVM: Acquire “MasterTransaction”

Both are able to execute the 1st step and get go further – this is a classic example from our OS classes.

Therefore, we need to define rules of locking between VMs, which I dubbed hierarchical locking:

Simple rules

Domain0’s transaction lock MUST BE acquired before acquiring IOVM’s transaction lock

IOVM cannot communicate with Domain0 while holding a transaction lock

We can get more complex than the above 2 rules IFF IOVM requires to talk to Domain0 in the context of a transaction.

## 2.6 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.

There are profiling and coverage tools, such as oprofile and gcov. Both can be used in user space and kernel space.

## 2.7 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 Bill 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/.