

# Implementation and Use of Data Structures in Linux

Mohammed Waseem Ashfaque

*Department of Computer Science & IT, College of management and Computer Technology, Aurangabad, India*

Sumegh Tharewal

*Department of Computer Science & IT, Dr. Babasaheb Ambedkar Marathwada University, Aurangabad, India*

Sayyada Sara Banu

*Department of Computer Science & IT Jizan University , Saudi Arabia*

Dr. Abdul Hannan

*Department. of Computer Science AL-Baha University, Albaha, Saudi Arabia*

**ABSTRACT** - This paper describes the *abstract* or *conceptual* software architecture of the Linux kernel. This level of architecture is concerned with the large-scale subsystems within the kernel, but not with particular procedures or variables. One of the purposes of such an abstract architecture is to form a mental model for Linux developers and architects. The model may not reflect the as-built architecture perfectly, but it provides a useful way to think about the overall structure. This model is most useful for entry-level developers, but is also a good way for experienced developers to maintain a consistent and accurate system vocabulary.

The architecture presented here is the result of reverse engineering an existing Linux implementation; the primary sources of information used were the documentation and source code. Unfortunately, no developer interviews were used to extract the live architecture of the system.

**Linux Kernel & System programming.** Some of the information I have taken from books/net and added my perception into it and presenting it in a simple language. Data structures are a fundamental building block for many software systems: computers manipulate information, and this information is often stored in data structures. Implementing linked data structures invariably appears early in an undergraduate Computer Science curriculum. Classically, programmers implement in memory data structures with pointers. Modern programming environments, however, include rich standard.

**Keywords :** Linux, Kernel Data Structure, Memory, Module

## 1. INTRODUCTION

The Linux kernel is composed of five main subsystems that communicate using procedure calls. Four of these five subsystems are discussed at the module interconnection level, and we discuss the architectural style in the sense used by Garlan and Shaw. At all times the relation of particular subsystems to the overall Linux system is considered.

The architecture of the kernel is one of the reasons that Linux has been successfully adopted by many users. In particular, the Linux kernel architecture was designed to support a large number of volunteer developers. Further, the subsystems that are most likely to need enhancements were architected to easily support extensibility. These two qualities are factors in the success of the overall system.

This presentation is somewhat unusual, in that the conceptual architecture is usually formed before the as-built architecture is complete. Since the author of this paper was not involved in either the design or implementation of the Linux system, this paper is the result of reverse engineering the Slackware 2.0.27 kernel source and documentation. A few architectural descriptions were used (in particular, [5] and [8] were quite helpful), but these descriptions were also based on the existing system implementation. By deriving the conceptual architecture from an existing implementation, this paper probably presents some implementation details as conceptual architecture.

In addition, the mechanisms used to derive the information in this paper omitted the best source of information -- the live knowledge of the system architects and developers. For a proper abstraction of the system architecture,

interviews with these individuals would be required. Only in this way can an accurate mental model of the system architecture be described.

Despite these problems, this paper offers a useful conceptualization of the Linux kernel software, although it cannot be taken as an accurate depiction of the system as implemented.

## II. SYSTEM ARCHITECTURE

### 2.1 System Overview

The Linux kernel is useless in isolation; it participates as one part in a larger system that, as a whole, is useful. As such, it makes sense to discuss the kernel in the context of the entire system. Figure 2.1 shows a decomposition of the entire Linux operating system:

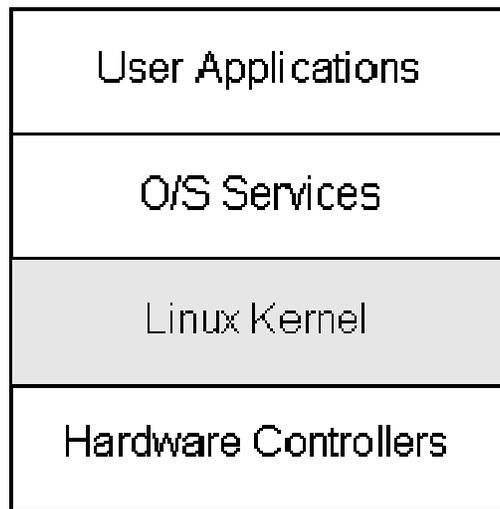


Figure 2.1: Decomposition of Linux System into Major Subsystems

The Linux operating system is composed of four major subsystems:

1. **User Applications** -- the set of applications in use on a particular Linux system will be different depending on what the computer system is used for, but typical examples include a word-processing application and a web-browser.
2. **O/S Services** -- these are services that are typically considered part of the operating system (a windowing system, command shell, etc.); also, the programming interface to the kernel (compiler tool and library) is included in this subsystem.
3. **Linux Kernel** -- this is the main area of interest in this paper; the kernel abstracts and mediates access to the hardware resources, including the CPU.
4. **Hardware Controllers** -- this subsystem is comprised of all the possible physical devices in a Linux installation; for example, the CPU, memory hardware, hard disks, and network hardware are all members of this subsystem[10], [11], [12].

This decomposition follows Garlan and Shaw's *Layered* style discussed in [[1]]; each subsystem layer can only communicate with the subsystem layers that are immediately adjacent to it. In addition, the dependencies between subsystems are from the top down: layers pictured near the top depend on lower layers, but subsystems nearer the bottom do not depend on higher layers.

Since the primary interest of this paper is the Linux kernel, we will completely ignore the User Applications subsystem, and only consider the Hardware and O/S Services subsystems to the extent that they interface with the Linux kernel subsystem.

### 2.2 Purpose of the Kernel

The Linux kernel presents a virtual machine interface to user processes. Processes are written without needing any knowledge of what physical hardware is installed on a computer -- the Linux kernel abstracts all hardware into a consistent virtual interface. In addition, Linux supports multi-tasking in a manner that is transparent to user processes: each process can act as though it is the only process on the computer, with exclusive use of main memory and other hardware resources. The kernel actually runs several processes concurrently, and is responsible for mediating access to hardware resources so that each process has fair access while inter-process security is maintained. 10], [11], [12].

### 2.3 Overview of the Kernel Structure

The Linux kernel is composed of five main subsystems:

1. The **Process Scheduler** (SCHED) is responsible for controlling process access to the CPU. The scheduler enforces a policy that ensures that processes will have fair access to the CPU, while ensuring that necessary hardware actions are performed by the kernel on time.
2. The **Memory Manager** (MM) permits multiple process to securely share the machine's main memory system. In addition, the memory manager supports virtual memory that allows Linux to support processes that use more memory than is available in the system. Unused memory is swapped out to persistent storage using the file system then swapped back in when it is needed.
3. The **Virtual File System** (VFS) abstracts the details of the variety of hardware devices by presenting a common file interface to all devices. In addition, the VFS supports several file system formats that are compatible with other operating systems.
4. The **Network Interface** (NET) provides access to several networking standards and a variety of network hardware.
5. The **Inter-Process Communication** (IPC) subsystem supports several mechanisms for process-to-process communication on a single Linux system.

Figure 2.2 shows a high-level decomposition of the Linux kernel, where lines are drawn from dependent subsystems to the subsystems they depend on: [10], [11], [12].

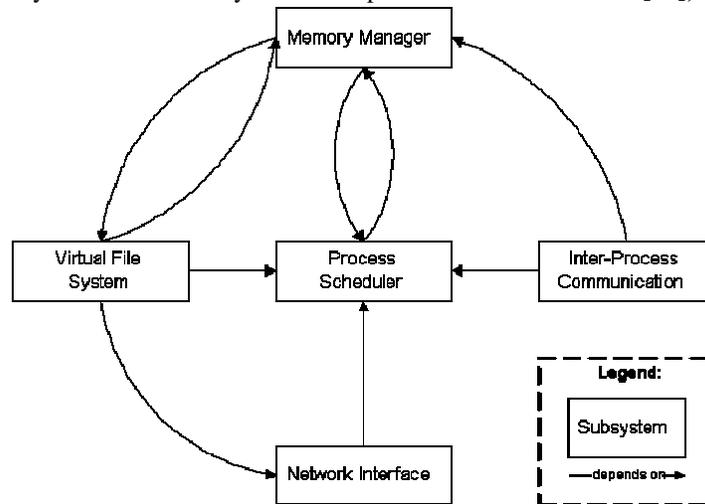


Figure 2.2: Kernel Subsystem Overview

This diagram emphasizes that the most central subsystem is the process scheduler: all other subsystems depend on the process scheduler since all subsystems need to suspend and resume processes. Usually a subsystem will suspend a process that is waiting for a hardware operation to complete, and resume the process when the operation is finished. For example, when a process attempts to send a message across the network, the network interface may need to suspend the process until the hardware has completed sending the message successfully. After the message has been sent (or the hardware returns a failure), the network interface then resumes the process with a return code indicating the success or failure of the operation. The other subsystems (memory manager, virtual file system, and inter-process communication) all depend on the process scheduler for similar reasons.

The other dependencies are somewhat less obvious, but equally important:

- 1) The process-scheduler subsystem uses the memory manager to adjust the hardware memory map for a specific process when that process is resumed.
- 2) The inter-process communication subsystem depends on the memory manager to support a shared-memory communication mechanism. This mechanism allows two processes to access an area of common memory in addition to their usual private memory.
- 3) The virtual file system uses the network interface to support a network file system NFS and also uses the memory manager to provide a ram disk device.
- 4) The memory manager uses the virtual file system to support swapping; this is the only reason that the memory manager depends on the process scheduler. When a process accesses memory that is currently swapped out, the

memory manager makes a request to the file system to fetch the memory from persistent storage, and suspends the process.

In addition to the dependencies that are shown explicitly, all subsystems in the kernel rely on some common resources that are not shown in any subsystem. These include procedures that all kernel subsystems use to allocate and free memory for the kernel's use, procedures to print warning or error messages, and system debugging routines. These resources will not be referred to explicitly since they are assumed ubiquitously available and used within the kernel layer of Figure 2.1.

The architectural style at this level resembles the *Data Abstraction* style discussed by Garlan and Shaw in [[1]]. Each of the depicted subsystems contains state information that is accessed using a procedural interface, and the subsystems are each responsible for maintaining the integrity of their managed resources.

#### 2.4 Supporting Multiple Developers

The Linux system was developed by a large number of volunteers (the current CREDITS file lists 196 developers that have worked on the Linux system). The large number of developers and the fact that they are volunteers has an impact on how the system should be architected. With such a large number of geographically dispersed developers, a tightly coupled system would be quite difficult to develop -- developers would be constantly treading on each others code. For this reason, the Linux system was architected to have the subsystems that were anticipated to need the most modification -- the file systems, hardware interfaces, and network system -- designed to be highly modular. For example, an implementation of Linux can be expected to support many hardware devices which each have distinct interfaces; a naive architecture would put the implementation of all hardware devices into one subsystem. An approach that better supports multiple developers is to separate the code for each hardware device into a *device driver* that is a distinct module in the file system. Analyzing the credits file gives most of the developers who have worked on the Linux kernel, and the areas that they appeared to have implemented. A few developers modified many parts of the kernel; for clarity, these developers were not included. For example, Linus Torvalds was the original implementor of most of the kernel subsystems, although subsequent development was done by others. This diagram can't be considered accurate because developer signatures were not maintained consistently during the development of the kernel, but it gives a general idea of what systems developers spent most of their effort implementing.

This diagram confirms the large-scale structure of the kernel as outlined earlier. It is interesting to note that very few developers worked on more than one system; where this did occur, it occurred mainly where there is a subsystem dependency. The organization supports the well-known rule of thumb stated by Melvin Conway (see [4]) that system organization often reflects developer organization. Most of the developers worked on hardware device drivers, logical file system modules, network device drivers, and network protocol modules. It's not surprising that these four areas of the kernel have been architected to support extensibility the most.

#### 2.5 System Data Structures

##### 2.5.1 Task List

The process scheduler maintains a block of data for each process that is active. These blocks of data are stored in a linked list called the task list; the process scheduler always maintains a current pointer that indicates the current process that is active.

##### 2.5.2 Memory Map

The memory manager stores a mapping of virtual to physical addresses on a per-process basis, and also stores additional information on how to fetch and replace particular pages. This information is stored in a memory-map data structure that is stored in the process scheduler's task list.

##### 2.5.3 I-nodes

The Virtual File System uses index-nodes (i-nodes) to represent files on a logical file system. The i-node data structure stores the mapping of file block numbers to physical device addresses. I-node data structures can be shared across processes if two processes have the same file open. This sharing is accomplished by both task data blocks pointing to the same i-node. [16][18]

##### 2.5.4 Data Connection

All of the data structures are rooted at the task list of the process scheduler. Each process on the system has a data structure containing a pointer to its memory mapping information, and also pointers to the i-nodes representing all of the opened files. Finally, the task data structure also contains pointers to data structures representing all of the opened network connections associated with each task.

### III. DATA STRUCTURE IMPLEMENTATIONS

#### Subsystem Architectures

##### 3.1 Process Scheduler Architecture

3.1.1 Goals

The process scheduler is the most important subsystem in the Linux kernel. Its purpose is to control access to the computer's CPU(s). This includes not only access by user processes, but also access for other kernel subsystems.

3.1.2 Modules

The scheduler is divided into four main modules:

1. The scheduling policy module is responsible for judging which process will have access to the CPU; the policy is designed so that processes will have fair access to the CPU.
2. Architecture-specific modules are designed with a common interface to abstract the details of any particular computer architecture. These modules are responsible for communicating with a CPU to suspend and resume a process. These operations involve knowing what registers and state information need to be preserved for each process and executing the assembly code to effect a suspend or resume operation.
3. The architecture-independent module communicates with the policy module to determine which process will execute next, then calls the architecture-specific module to resume the appropriate process. In addition, this module calls the memory manager to ensure that the memory hardware is restored properly for the resumed process.

The system call interface module permits user processes access to only those resources that are explicitly exported by the kernel. This limits the dependency of user processes on the kernel to a well-defined interface that rarely changes, despite changes in the implementation of other kernel modules.

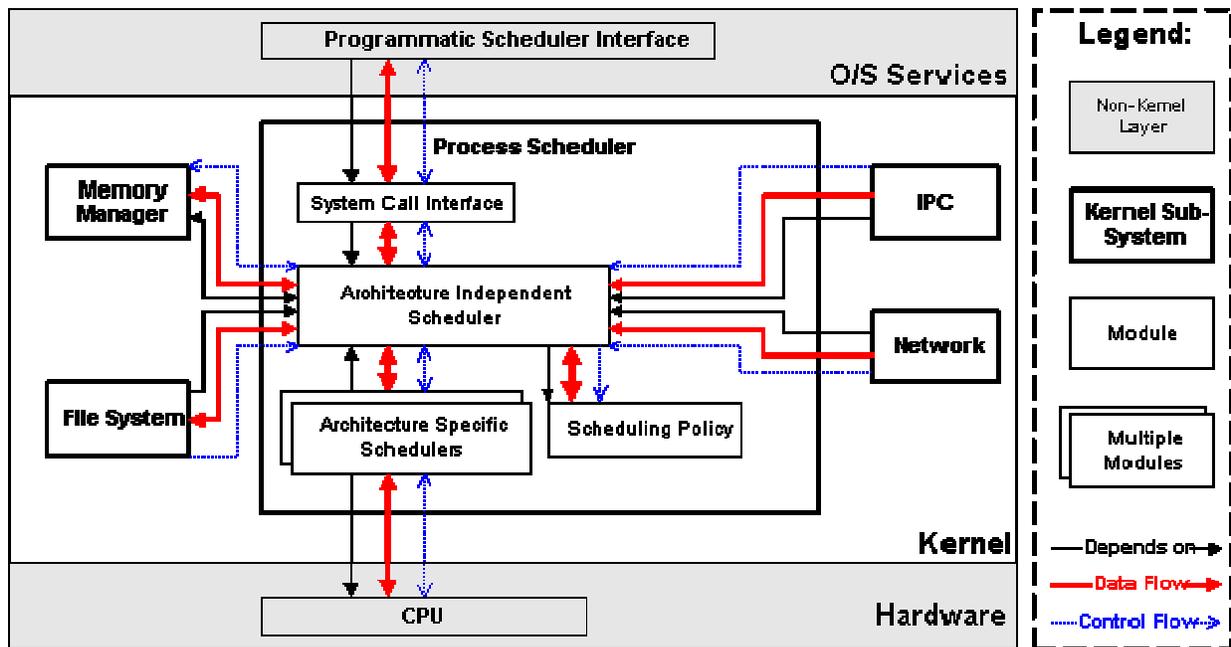


Figure 3.1: Process Scheduler Subsystem in Context

3.1.3 Data Representation

The scheduler maintains a data structure, the task list, with one entry for each active process. This data structure contains enough information to suspend and resume the processes, but also contains additional accounting and state information. This data structure is publicly available throughout the kernel layer

3.1.4 Dependencies, Data Flow, and Control Flow

The process scheduler calls the memory manager subsystem as mentioned earlier; because of this, the process scheduler subsystem depends on the memory manager subsystem. In addition, all of the other kernel subsystems depend on the process scheduler to suspend and resume processes while waiting for hardware requests to complete. These dependencies are expressed through function calls and access to the shared task list data structure. All kernel

subsystems read and write the data structure representing the current task, leading to bi-directional data flow throughout the system.

In addition to the data and control flow within the kernel layer, the O/S services layer provides an interface for user processes to register for timer notification. This corresponds to the implicit execution architectural style described in[[1]]. This leads to a flow of control from the scheduler to the user processes. The usual case of resuming a dormant process is not considered a flow of control in the normal sense because the user process cannot detect this operation. Finally, the scheduler communicates with the CPU to suspend and resume processes; this leads to a data flow, and a flow of control. The CPU is responsible for interrupting the currently executing process and allowing the kernel to schedule another process.

### *3.2 Memory Manager Architecture*

#### *3.2.1 Goals*

The memory manager subsystem is responsible for controlling process access to the hardware memory resources. This is accomplished through a hardware memory-management system that provides a mapping between process memory references and the machine's physical memory. The memory manager subsystem maintains this mapping on a per process basis, so that two processes can access the same virtual memory address and actually use different physical memory locations. In addition, the memory manager subsystem supports swapping; it moves unused memory pages to persistent storage to allow the computer to support more virtual memory than there is physical memory.

#### *3.2.2 Modules*

The memory manager subsystem is composed of three modules:

1. The architecture specific module presents a virtual interface to the memory management hardware
2. The architecture independent manager performs all of the per-process mapping and virtual memory swapping. This module is responsible for determining which memory pages will be evicted when there is a page fault -- there is no separate policy module since it is not expected that this policy will need to change.
3. A system call interface is provided to provide restricted access to user processes. This interface allows user processes to allocate and free storage, and also to perform memory mapped file I/O.

#### *3.2.3 Data Representation*

The memory manager stores a per-process mapping of physical addresses to virtual addresses. This mapping is stored as a reference in the process scheduler's task list data structure. In addition to this mapping, additional details in the data block tell the memory manager how to fetch and store pages. For example, executable code can use the executable image as a backing store, but dynamically allocated data must be backed to the system paging file. Finally, the memory manager stores permissions and accounting information in this data structure to ensure system security.

#### *3.2.4 Data Flow, Control Flow, and Dependencies*

The memory manager controls the memory hardware, and receives a notification from the hardware when a page fault occurs -- this means that there is bi-directional data and control flow between the memory manager modules and the memory manager hardware. Also, the memory manager uses the file system to support swapping and memory mapped I/O. This requirement means that the memory manager needs to make procedure calls to the file system to store and fetch memory pages from persistent storage. Because the file system requests cannot be completed immediately, the memory manager needs to suspend a process until the memory is swapped back in; this requirement causes the memory manager to make procedure calls into the process scheduler. Also, since the memory mapping for each process is stored in the process scheduler's data structures, there is a bi-directional data flow between the memory manager and the process scheduler. User processes can set up new memory mappings within the process address space, and can register themselves for notification of page faults within the newly mapped areas. This introduces a control flow from the memory manager, through the system call interface module, to the user processes. There is no data flow from user processes in the traditional sense, but user processes can retrieve some information from the memory manager using select system calls in the system call interface module.

### *3.3 Virtual File System Architecture [15],[17]*

#### *3.4 Network Interface Architecture*

##### *3.4.1 Goals*

The network subsystem allows Linux systems to connect to other systems over a network. There are a number of possible hardware devices that are supported, and a number of network protocols that can be used. The network subsystem abstracts both of these implementation details so that user processes and other kernel subsystems can access the network without necessarily knowing what physical devices or protocol is being used.

### 3.4.2 Modules

1. Network device drivers communicate with the hardware devices. There is one device driver module for each possible hardware device.
2. The device independent interface module provides a consistent view of all of the hardware devices so that higher levels in the subsystem don't need specific knowledge of the hardware in use.
3. The network protocol modules are responsible for implementing each of the possible network transport protocols.
4. The protocol independent interface module provides an interface that is independent of hardware devices and network protocol. This is the interface module that is used by other kernel subsystems to access the network without having a dependency on particular protocols or hardware.

Finally, the system calls interface module restricts the exported routines that user processes can access. [13],[14],[15].

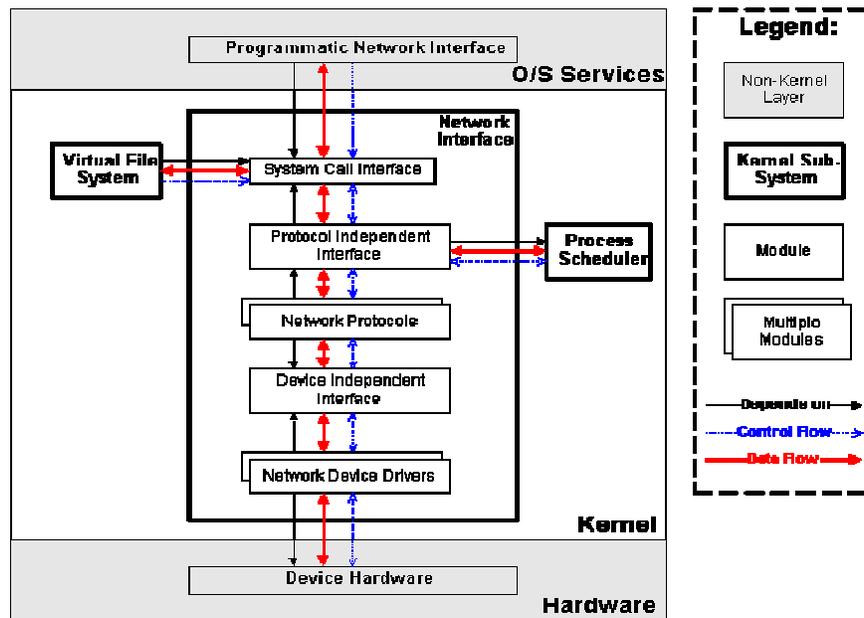


Figure 3.4: Network Interface Subsystem in Context

### 3.4.3 Data Representation

Each network object is represented as a socket. Sockets are associated with processes in the same way that i-nodes are associated; sockets can be shared amongst processes by having both of the task data structures pointing to the same socket data structure.

### 3.4.4 Data Flow, Control Flow, and Dependencies

The network subsystem uses the process scheduler to suspend and resume processes while waiting for hardware requests to complete (leading to a subsystem dependency and control and data flow). In addition, the network subsystem supplies the virtual file system with the implementation of a logical file system (NFS) leading to the virtual file system depending on the network interface and having data and control flow with it.

### 3.5 Inter-Process Communication Architecture

The architecture of the inter-process communication subsystem is omitted for brevity since it is not as interesting as the other subsystems.

Data structure uses [16][18]

LIST\_HEAD\_INIT - Initialize the task list node embedded in the task\_struct.  $\rightarrow$  container\_of(ptr, type, member) - Returns the parent structure containing a list\_head (circular doubly linked list node).  $\rightarrow$  1. Linux Kernel Data Structure Task list - It's a circular doubly linked list. Each entry contains a structure (task\_struct). Each structure contains detailed information about corresponding process and a task list node. The node contains previous and next pointer only). Macros to Initialize / manipulate Task list:

list\_for\_each\_entry\_safe\_continue (pos, n, head, member) - safe against entry removal while continuing after current point pos: pointer type to use as a loop cursor. n: pointer type to store next pointer. head: the head for your list (struct list\_head \*head) member: the name of the list\_struct (struct list\_head \*list) → list\_for\_each\_entry\_reverse\_safe(pos, n, head, member) safe against entry removal while traversing the list backward → list\_for\_each\_entry\_reverse(pos, head, member) – Traverse given list backward → list\_for\_each\_entry\_safe(pos, n, head, member) – safe against entry removal while traversing the list → list\_for\_each\_entry(pos, head, member) – Traverse given list → 2. Linux Kernel Data Structure Macros to Initialize / manipulate Task list :

list\_del\_init(struct list\_head \*entry) – delete a node from a linked list and reinitialize the given list head → list\_del(struct list\_head \*entry) – delete a node from the list → list\_add\_tail( struct list\_head \*new, struct list\_head \*head) - add the new node to the list immediately before the head node → list\_add(struct list\_head \*new, struct list\_head \*head) - add a node to given list → 3. Linux Kernel Data Structure Functions to Initialize/manipulate Task list :

list\_splice\_init(struct list\_head \*list, struct list\_head \*head) – Join two lists and initialize first list → list\_splice(struct list\_head \*list, struct list\_head \*head) – Join two lists → list\_empty(struct list\_head \*head) – verify if list is empty → list\_move\_tail list\_move(struct list\_head \*list, struct list\_head \*head) ) – Remove the node from the list and add the node add the node before head node in the given list. → list\_move(struct list\_head \*list, struct list\_head \*head) – Remove the node from the list and add the node after head node in the given list. → 4. Linux Kernel Data Structure Functions to Initialize/manipulate Task list :

unsigned int kfifo\_in(struct kfifo \*fifo, const void \*from, unsigned int len) – add data of size “len” from memory pointed to by “from pointer” to queue → static inline bool kfifo\_initialized(struct kfifo \*fifo) – verify if queue is initialized → void kfifo\_init(struct kfifo \*fifo, void \*buffer, unsigned int size)- creates a queue and initializes with “size” bytes from the memory pointed by buffer → int kfifo\_alloc(struct kfifo \*fifo, unsigned int size, gfp\_t gfp\_mask) – creates a queue and initializes with “size” bytes → 5. Linux Kernel Data Structure Kfifo - Generic Queue used by Linux kernel Functions to Initialize/manipulate Queue :

static inline void kfifo\_reset\_out(struct kfifo \*fifo) – does not remove entire contents of queue → static inline void kfifo\_reset(struct kfifo \*fifo) – remove entire contents of queue → unsigned int kfifo\_out\_peek(struct kfifo \*fifo, void \*to, unsigned int len, unsigned int offset) – peek data within the queue without removing it → unsigned int kfifo\_out(struct kfifo \*fifo, void \*to, unsigned int len)– remove from queue data of size “len” and copy to memory pointed by “to pointer” from queue → 6. Linux Kernel Data Structure Functions to Initialize/manipulate Queue :

void kfifo\_free(struct kfifo \*fifo) – free memory allocated to queue using kfifo\_alloc() → unsigned int kfifo\_avail(struct kfifo \*fifo) – returns no of bytes available in the queue → static inline unsigned int kfifo\_len(struct kfifo \*fifo) – returns queue size → int kfifo\_is\_full (struct kfifo \*fifo) - verify if queue is full → int kfifo\_is\_empty(struct kfifo \*fifo) – verify if queue is empty → 7. Linux Kernel Data Structure Functions to Initialize/manipulate Queue :

Linux Kernel Data Structure IDR – It is the map implementation of Linux . Struct idr is used for mapping user-space UIDs to their associated kernel data structure Functions to Initialize/manipulate IDR: void idr\_init(struct idr \*idp)- initialize Map (idr) Allocating a new UID (create a new map entry) 2 steps: 1. int idr\_pre\_get(struct idr \*idp, gfp\_t gfp\_mask) Resizes the backing tree 2. int idr\_get\_new(struct idr \*idp, void \*ptr, int \*id) Uses the idr pointed at by idp to allocate a new UID and associate it with the pointer ptr.

Linux Kernel Data Structure Functions to Initialize/manipulate Queue: void \*idr\_find(struct idr \*idp, int id)– look up a UID void idr\_remove(struct idr \*idp, int id) Remove a UID from an idr void idr\_remove\_all(struct idr \*idp) – remove all entries from idr void idr\_destroy(struct idr \*idp) - Destroy entire idr.

Linux Kernel Data Structure Rbtree (red-black tree) – It is the Linux’s implementation of (semi) Balanced Binary search tree. It is useful for inserting and searching efficiently. prio\_tree (PST) - It is the Linux’s implementation of radix priority search tree which is a mix of heap and radix trees. It is useful for storing intervals. e.g. consider a vma as a closed interval of file pages [offset\_begin, offset\_end], and store all vmas that map a file in a PST.

#### IV. CONCLUSIONS

The Linux kernel is one layer in the architecture of the entire Linux system. The kernel is conceptually composed of five major subsystems: the process scheduler, the memory manager, the virtual file system, the network interface, and the inter-process communication interface. These subsystems interact with each other using function calls and shared data structures.

At the highest level, the architectural style of the Linux kernel is close to Garlan and Shaw's *Data Abstraction* style ([Garlan1994]); the kernel is composed of subsystems that maintain internal representation consistency by using a specific procedural interface. As each of the subsystems is elaborated, we see an architectural style that is similar to the *layered* style presented by Garlan and Shaw. Each of the subsystems is composed of modules that communicate only with adjacent layers.

The conceptual architecture of the Linux kernel has proved its success; essential factors for this success were the provision for the organization of developers, and the provision for system extensibility. The Linux kernel architecture was required to support a large number of independent volunteer developers. This requirement suggested that the system portions that require the most development -- the hardware device drivers and the file and network protocols -- be implemented in an extensible fashion. The Linux architect chose to make these systems be extensible using a data abstraction technique: each hardware device driver is implemented as a separate module that supports a common interface. In this way, a single developer can add a new device driver, with minimal interaction required with other developers of the Linux kernel. The success of the kernel implementation by a large number of volunteer developers proves the correctness of this strategy.

Another important extension to the Linux kernel is the addition of more supported hardware platforms. The architecture of the system supports this extensibility by separating all hardware-specific code into distinct modules within each subsystem. In this way, a small group of developers can effect a port of the Linux kernel to a new hardware architecture by re-implementing only the machine-specific portions of the kernel. In future this work may be extended using database and Linux.

## REFERENCES

- [1] David Garlan and Mary Shaw, *An Introduction to Software Architecture*, Advances in Software Engineering and Knowledge Engineering, Volume I, World Scientific Publishing Company, 1993.
- [2] Robert T. Monroe, Andrew Kompanek, Ralph Melton, and David Garlan, *Architectural Styles, Design Patterns, and Objects*, IEEE Software, January 1997, pp 43-52.
- [3] *Slackware Linux Unleashed*, by Timothy Parker, et al, Sams Publishing, 201 West 103rd Street, Indianapolis.
- [4] *The New Hackers Dictionary*, Second Edition, compiled by Eric S. Raymond. The MIT Press, Cambridge Massachusetts, 1993.
- [5] *The Linux Kernel*, by David A. Rusling, draft, version 0.1-13(19), <ftp://sunsite.unc.edu/pub/Linux/docs/linux-doc-project/linux-kernel/> or <http://www.linuxhq.com/guides/TLK/index.html>.
- [6] Soni, D.; Nord, R. L.; Hofmeister, C., *Software Architecture in Industrial Applications*, IEEE ICSE 1995, pp. 196-210.
- [7] *Modern Operating Systems*, by Andrew S. Tanenbaum, Prentice Hall, 1992.
- [8] *Linux System Administrators' Guide 0.6*, by Lars Wirzenius, <http://www.iki.fi/liw/linux/sag/r> [http://www.linuxhq.com/LDP/LDP/sag/in ex.html](http://www.linuxhq.com/LDP/LDP/sag/in%20ex.html).
- [9] Dewayne E. Perry and Alexander L. Wolf, *Foundations for the Study of Software Architecture*, ACM SIGSOFT Software Engineering Notes, 17:4, October 1992 pp 40-52.
- [10] Hua Zhong and Jason Nieh. **CRAK: Linux Checkpoint / Restart As a Kernel Module**. Technical Report CUCS- 014-01. Department of Computer Science, Columbia University, November 2002
- [11] Yee-Ting Li and Doug Leith. **Bictcp implementation in Linux kernels**. Technical report, Hamilton Institute, February 2004
- [12] Dario Faggioli, Fabio Checconi, Michael Trimarchi, and Claudio Scordino. **An EDF scheduling class for the Linux kernel**. In Proceedings of the Eleventh Real- Time Linux Workshop, Dresden, Germany, September 2009.
- [13] M. Beck, H. B'ohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. **The Linux File System**. In *Linux Kernel Internals, Second Edition*, pages 152–73. Addison-Wesley, 1998.
- [14] J. Heidemann and G. Popek. Performance of cache coherence in stackable filing. *Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, CO). Association for Computing Machinery SIGOPS, 3–6 December 1995.
- [15] D. S. H. Rosenthal. Requirements for a “**VFS Interface/ Stacking**” Vnode/. Unix International document SD-01-02-N014. UNIX International, 1992.
- [16] D. S. H. Rosenthal. Evolving the **Vnode Interface**. *USENIX Conference Proceedings* (Anaheim, CA), pages 107–18. USENIX, Summer 1990.
- [17] SMCC. **lofs – loopback virtual file system**. SunOS Reference Manual, Section 7. Sun Microsystems, Incorporated, 20 March 1992.
- [18] E. Zadok, I. Badulescu, and A. Shender. **Cryptfs: A Stackable Vnode Level Encryption File System**. Technical Report CUCS-021-98. Computer Science Department, Columbia University, 28 July 1998. Available <http://www.cs.columbia.edu/library/>.
- [19] ARCANGELI, A., CAO, M., MCKENNEY, P. E., AND SARMA, D. Using read-copy update techniques for System **VIPC in the Linux 2.5 kernel**. In Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track) (June 2003), USENIX Association, pp. 297–310.
- [20] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An analysis of Linux scalability to many cores. In 9th USENIX Symposium on **Operating System Design and Implementation** (Vancouver, BC, Canada, October 2010), USENIX, pp. 1–16.