



RV Educational Institutions[®]
RV College of Engineering[®]

Autonomous Institution
Affiliated to Visvesvaraya
Technological University,
Belagavi

Approved by AICTE,
New Delhi

**DEPARTMENT OF COMPUTER SCIENCE AND
ENGINEERING**

OPERATING SYSTEMS - CS235AI

REPORT

TOPIC: KERNEL IMPLEMENTATION

Submitted by

**RAKSHITHA K
SOMASHEKARA G**

**1RV23CS414
1RV23CS417**

**Computer Science and Engineering
2023-2024**

INDEX

1. Introduction
2. System Architecture
3. Methodology
4. Systems calls used
5. Output/results
6. Conclusion

INTRODUCTION

Kernel development encompasses the intricate process of crafting, refining, and maintaining the fundamental component of an operating system, known as the kernel. Functioning as the core liaison between the hardware and software layers of a computing environment, the kernel plays an indispensable role in orchestrating system operations and facilitating seamless interaction between various system components.

1. Purpose of the Kernel:

The kernel's primary objective revolves around the efficient management of hardware resources within a computing system. It serves as the guardian of vital system components, including the central processing unit (CPU), memory modules, input/output (I/O) devices, and networking interfaces. By regulating resource allocation and utilization, the kernel ensures optimal system performance while mitigating potential conflicts and bottlenecks. Moreover, the kernel provides a foundational framework for executing essential system-level tasks and services, ranging from process management and memory allocation to device control and filesystem operations.

2. Kernel Components:

A kernel comprises several integral components, each fulfilling distinct roles in the overall system operation:

1. **Process Management:** This component governs the creation, scheduling, and termination of processes or threads within the system. By coordinating the execution of multiple concurrent tasks, the kernel ensures efficient resource utilization and responsiveness.
2. **Memory Management:** Responsible for allocating and managing system

memory resources, including physical and virtual memory. The memory management subsystem oversees processes such as memory allocation, deallocation, and virtual memory paging, thereby optimizing memory utilization and ensuring system stability.

3. **Device Drivers:** Kernel-integrated device drivers facilitate communication between the operating system and hardware peripherals. These drivers provide standardized interfaces for interacting with various devices, such as disks, network adapters, and input/output devices, enabling seamless data transfer and device control.
4. **Filesystem Support:** The filesystem component manages the organization, storage, and retrieval of data stored on storage devices, such as hard drives and solid-state drives (SSDs). By implementing filesystem protocols and data structures, the kernel enables efficient file storage, retrieval, and manipulation.
5. **Networking:** Facilitating communication between different nodes within a network, the networking subsystem enables data transmission and reception across interconnected devices. Through protocols and network stack implementations, the kernel facilitates reliable and secure data exchange, supporting diverse networking functionalities such as packet routing, protocol encapsulation, and socket communication.

TYPES OF KERNEL IMPLEMENTATION

1. Custom Configuration (Menuconfig, Xconfig, etc.):

This approach involves customizing an existing kernel by selecting or deselecting various options using configuration interfaces like menuconfig, xconfig, or gconfig. These tools provide user-friendly interfaces to navigate through kernel options, enabling users to enable or disable features, drivers, and subsystems according to their requirements.

2. Manual Configuration (Editing Configuration Files):

Manual configuration involves directly editing the kernel configuration file (usually named `.config`) to specify which features and options to include or exclude in the kernel build. Users can modify configuration options using a text editor, adjusting settings and enabling or disabling features as needed.

3. Writing from Scratch (Custom Kernel Development):

This approach entails creating a kernel entirely from scratch, either by writing the code directly or using a minimalistic kernel framework as a starting point. Custom kernel development offers maximum flexibility but requires deep understanding of kernel architecture and low-level programming concepts.

4. Modular Kernel Configuration:

Rather than compiling all features directly into the kernel image, modular kernel configuration involves compiling some features as loadable kernel modules. This allows for dynamic loading and unloading of functionality, reducing kernel size and enabling better resource utilization.

5. Cross-Compilation:

Cross-compilation involves building the kernel on a development machine with a different architecture than the target system. This approach is common in embedded systems development or when targeting specific hardware platforms. Cross-compilation tools facilitate building kernels for various architectures, ensuring compatibility with target devices.

6. Embedded Kernel Development:

Embedded kernel development focuses on creating lightweight kernels optimized for embedded systems, such as IoT devices, embedded boards, or consumer electronics. Embedded kernels are tailored to meet the resource constraints and specific requirements of embedded platforms, prioritizing efficiency and minimalism.

7. Real-Time Kernel Development:

Real-time kernels are designed to meet strict timing requirements for critical applications, such as industrial automation, robotics, and aerospace systems. Real-time kernel development involves optimizing kernel scheduling algorithms, reducing latency, and ensuring predictable response times to meet real-time constraints.

SYSTEM ARCHITECTURE

Menuconfig System Architecture:

Menuconfig operates within the context of the Linux kernel source code, leveraging configuration files such as Kconfig and Makefiles to present a user-friendly interface for customizing kernel options. When a user initiates menuconfig, the tool parses these configuration files to generate a hierarchical menu structure representing various kernel features and dependencies. This structure serves as the backbone of the configuration process, allowing users to navigate through different categories and subcategories to modify settings according to their requirements. As users make selections within the menuconfig interface, the tool updates the kernel configuration file (.config) to reflect the chosen options. Once the configuration process is complete, users can proceed to build the kernel using the updated configuration, resulting in a customized kernel tailored to their specific needs and preferences.

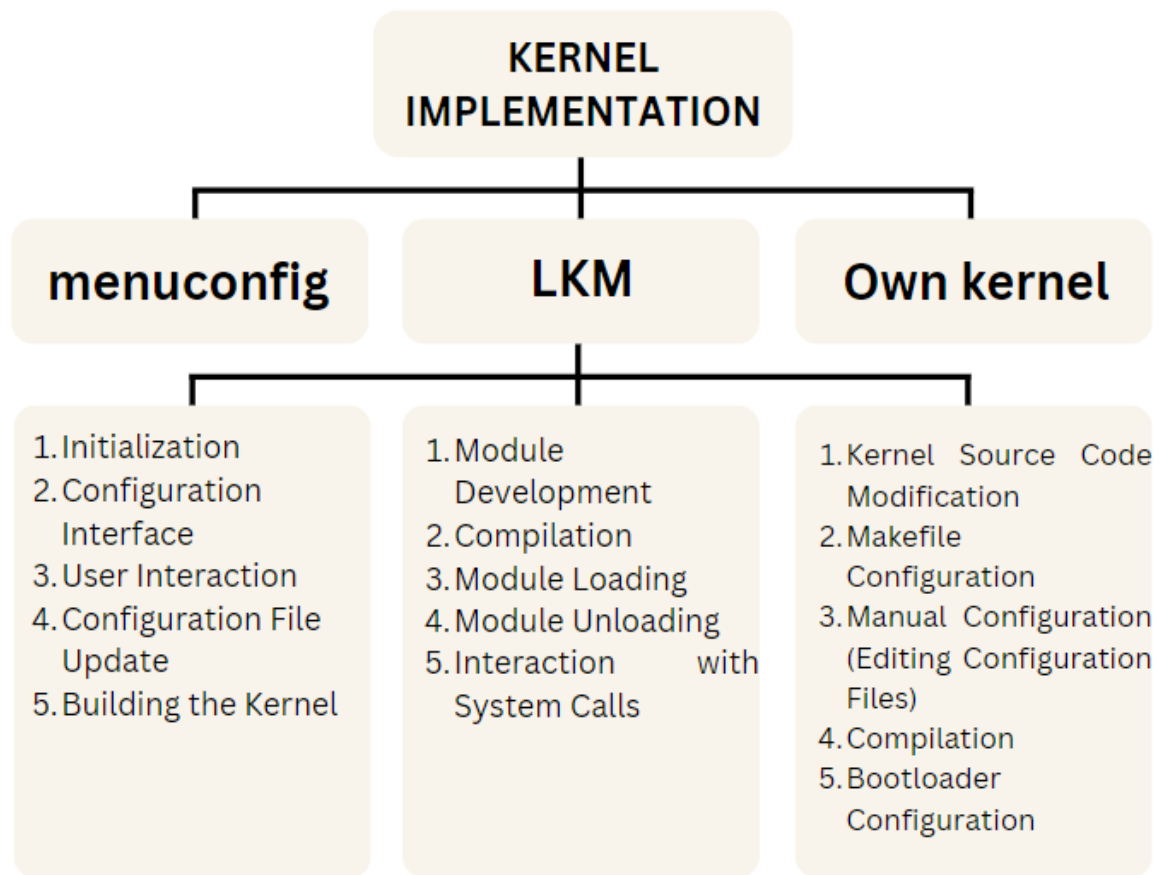
Loadable Kernel Module (LKM) System Architecture:

The architecture of Loadable Kernel Modules (LKMs) revolves around the dynamic loading and unloading of kernel extensions without requiring a system reboot. LKMs are standalone pieces of code compiled separately from the main kernel source, typically as .ko files. When a user loads an LKM into the kernel using utilities like insmod or modprobe, the module loader inserts the module's code and data structures into the kernel's address space, extending its functionality. LKMs interact with the kernel through system calls, enabling them to provide additional features or device support, or modify existing behavior. This modular architecture allows for flexibility and scalability, as modules can be loaded or unloaded based on system requirements, leading to efficient resource utilization and enhanced system flexibility.

Building from Scratch System Architecture:

Building a kernel from scratch involves a comprehensive understanding of kernel architecture and low-level programming concepts. Developers typically start by writing or modifying the kernel's source code, implementing essential components such as process management, memory allocation, and device drivers. The build process relies on Makefiles and compilation tools to translate the source code into executable kernel images suitable for booting. Configuration options are specified either through manual editing of configuration files or via custom configuration interfaces. Unlike menuconfig, which modifies an existing kernel configuration, building from scratch allows for complete control over every aspect of the kernel, from selecting supported hardware to fine-tuning performance parameters. This approach offers maximum flexibility but requires considerable expertise and effort, making it suitable for specialized use cases or highly customized environments where off-the-shelf kernels are insufficient.

METHODOLOGY



SYSTEM CALLS USED

1. Menuconfig:

- **Read and Write System Calls:** Menuconfig interacts with the filesystem through read and write system calls to access and modify configuration files.
- **Fork System Call:** In some cases, menuconfig may fork processes to handle concurrent user interactions or to execute external commands.

2. Loadable Kernel Module (LKM):

- **init_module:** This system call is used to load a kernel module into the kernel's address space.
- **delete_module:** It is used to unload a kernel module from the kernel's address space.

3. Building from Scratch:

- **Execve System Call:** When compiling the kernel from scratch, the build process involves executing various commands and scripts. The execve system call is used to replace the current process image with a new one specified by the build tool.

OUTPUT

1. menuconfig

```
vboxuser@pro2: ~/linux-5.15.25
.config - Linux/x86 5.15.25 Kernel Configuration
> General setup

Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
submenus --->).  Highlighted letters are hotkeys.  Pressing <Y>
includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
exit, <?> for Help, </> for Search.  Legend: [*] built-in [ ]

^(-)
[ ] Local version - append to kernel release
[ ] Automatically append version information to the version string
[ ] Build ID Salt
Kernel compression mode (LZ4) --->
[ ] Default init path
((none)) Default hostname
[*] Support for paging of anonymous memory (swap)
[*] System V IPC
[*] POSIX Message Queues
[*] General notification queue
v(+)
```

```
vboxuser@pro2: ~/linux-5.15.25
.config - Linux/x86 5.15.25 Kernel Configuration
> General setup

CONFIG_KERNEL_LZ4:
LZ4
LZ4 is an LZ77-type compressor with a fixed, byte-oriented encoding.
A preliminary version of LZ4 de/compression tool is available at
<https://code.google.com/p/lz4/>.

Its compression ratio is worse than LZ0. The size of the kernel
is about 8% bigger than LZ0. But the decompression speed is
faster than LZ0.

Symbol: KERNEL_LZ4 [=y]
Type : bool
Defined at init/Kconfig:298
Prompt: LZ4
Depends on: <choice> && HAVE_KERNEL_LZ4 [=y]
Location:

<=select> < Exit > < Help > < Save > < Load >
```

```
vboxuser@pro2: ~
vboxuser@pro2:~$ vmstat
procs -----memory----- --swap-- -----io----- -system-- -----cpu-----
 r b swpd free buff cache si so bi bo in cs us sy id wa st
 3 0 0 2624864 90544 2232336 0 0 241 280 360 332 10 2 88 1
 0

vboxuser@pro2:~$ free
              total            used             free           shared  buff/cache   available
Mem:          5737700          789796          2624864           43204          2323040          4626472
Swap:          2134012              0           2134012

vboxuser@pro2:~$
```

```
vboxuser@pro2: ~
top - 11:40:30 up 45 min, 1 user, load average: 0.03, 0.35, 0.40
Tasks: 180 total, 1 running, 177 sleeping, 2 stopped, 0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  5603.2 total,  2598.3 free,   736.8 used,  2268.1 buff/cache
MiB Swap:  2084.0 total,  2084.0 free,    0.0 used,  4552.5 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM    TIME+  COMMAND
 1417 vboxuser  20   0  4275464 372172 139092 S   0.7   6.5   1:44.10 gnome-s+
 1955 vboxuser  20   0  555552  54460  41220 S   0.3   0.9   0:15.34 gnome-t+
 29943 vboxuser  20   0  13080  4096  3328 R   0.3   0.1   0:00.06 top
    1 root      20   0  167912  13176  8312 S   0.0   0.2   0:01.65 systemd
    2 root      20   0      0      0      0 S   0.0   0.0   0:00.00 kthreadd
    3 root      0 -20      0      0      0 I   0.0   0.0   0:00.00 rcu_gp
    4 root      0 -20      0      0      0 I   0.0   0.0   0:00.00 rcu_par+
    5 root      0 -20      0      0      0 I   0.0   0.0   0:00.00 slub_fl+
    6 root      0 -20      0      0      0 I   0.0   0.0   0:00.00 netns
    8 root      0 -20      0      0      0 I   0.0   0.0   0:00.00 kworker+
   11 root      0 -20      0      0      0 I   0.0   0.0   0:00.00 mm_perc+
   12 root      20   0      0      0      0 I   0.0   0.0   0:00.00 rcu_tas+
   13 root      20   0      0      0      0 I   0.0   0.0   0:00.00 rcu_tas+
   14 root      20   0      0      0      0 I   0.0   0.0   0:00.00 rcu_tas+
   15 root      20   0      0      0      0 S   0.0   0.0   0:00.51 ksoftir+
   16 root      20   0      0      0      0 I   0.0   0.0   0:01.61 rcu_pre+
   17 root      rt   0      0      0      0 S   0.0   0.0   0:00.03 migrati+
```

2. Loadable kernel module

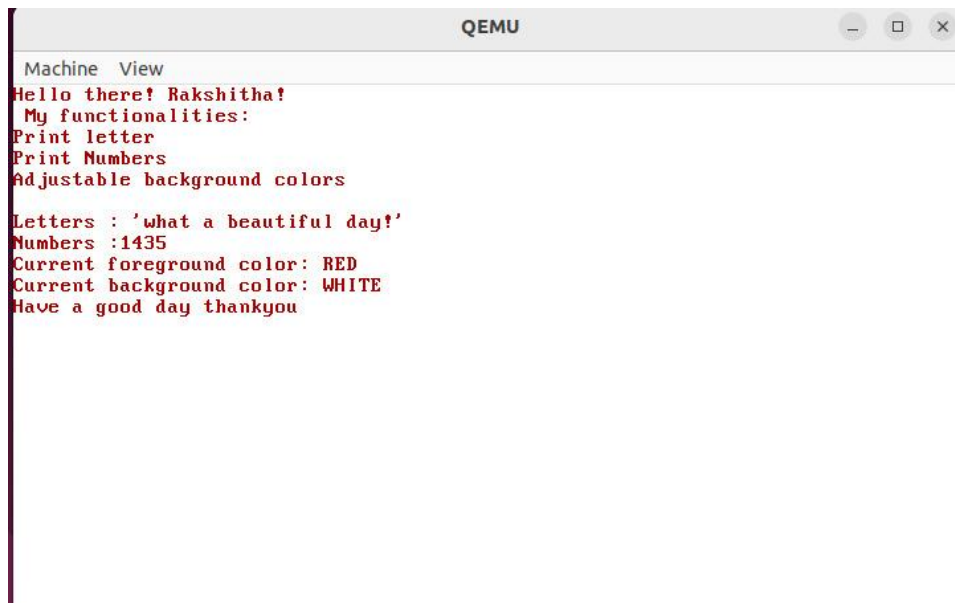
```

1289 1571 1861 37 5507 62 dma self
1290 1573 19 3893 5514 63 driver slabinfo
1291 1578 1919 39 5523 64 dynamic_debug softirqs
13 1593 195 3912 5581 657 execdomains stat
1306 16 1952 4 5582 67 fb swaps
1309 1606 2 40 5631 673 filesystems sys
1319 1620 20 4035 5669 68 fs sysrq-trigger
1323 1622 2000 4059 5673 69 interrupts sysvipc
1326 1635 2004 4075 57 693 iomem thread-self
1331 1636 2028 41 571 7 timer_list
1333 1638 21 42 5717 702 irq tty
1336 1639 22 4234 5718 711 jiffies uptime
1345 1640 224 425 572 712 kallsyms version
1385 1642 225 43 5728 730 kcore version_signature
14 1649 2272 435 574 738 keys vmallocinfo
1407 1650 23 437 575 760 key-users vmstat
1422 1654 245 44 576 77 kmsg zoneinfo
1441 1660 25 45 578 79 kpagecgroup
1449 1661 26 4512 585 804 kpagecount
1452 1663 266 46 586 83 kpageflags
15 1666 27 4640 589 982 loadavg
1515 1671 2780 4645 59 995 locks
1520 1674 28 47 590 mdstat
rakshitha@ubuntuclass: ~/linux-6.7.7/proc_jiffies$

```

[illegible]

3. Building your own kernel



The image shows a QEMU window titled "QEMU" with a menu bar containing "Machine" and "View". The main display area shows the output of a custom kernel boot. The text is as follows:

```
Machine  View
Hello there! Rakshitha!
My functionalities:
Print letter
Print Numbers
Adjustable background colors

Letters : 'what a beautiful day!'
Numbers :1435
Current foreground color: RED
Current background color: WHITE
Have a good day thankyou
```

CONCLUSION

In wrapping up this project, it's clear that each methodology offers unique ways to tweak and enhance Linux kernel environments. Menuconfig provides an easy-to-use interface, perfect for beginners like myself, to customize kernel features without delving too deeply into the intricacies of kernel development. Loadable kernel modules offer a flexible solution for adding or removing functionalities on-the-fly, which is handy when experimenting with new features or supporting different hardware configurations. Lastly, building the kernel from scratch might seem daunting at first, but it's a rewarding experience for those looking to gain a deeper understanding of how the kernel works and customize it to their exact needs. Overall, these methodologies open up a world of possibilities for students and aspiring developers, allowing us to optimize Linux kernel setups for various tasks while learning along the way.