Challenges in Platform Selection

This article examines the criteria for selecting a software platform, including special requirements, and the various features that the architect needs to consider while making the choice.

To begin, let's cover some fundamentals about operating systems. An operating system is an environment that enables users to execute their programs on the system hardware. Without an operating system, users need to write applications at a machine level, which only the hardware can understand and execute. However, this is extremely difficult, and besides, it's not portable to different hardware sets—thus the need for an environment in which a person can use a high-level language to interact with the system. Besides interfacing between user and hardware, the operating system provides several other important and useful features,

such as multi-tasking, in which many tasks/applications can run on a single system, with the operating system allocating available resources (CPU, main memory, input/output etc) to the applications, based on the priority and criticality of each. In addition, the operating system should provide tools for application development—a Software Development Kit (SDK), and debugging, tracing and monitoring support. In short, the core objectives of any operating system are: user convenience, efficient resource allocation, and support for development and testing of applications.

Different types of kernels

An operating system mainly consists of the kernel and utilities. The core part of the operating system is the kernel, which runs directly on hardware. Its main job is to execute a given program on behalf of the user, and provide the result of execution back to the user in a human-readable format. Utilities include Graphical User Interfaces (GUIs), editors, debuggers, tracers, compilers, Web browsers and so on.

The main subsystems of a kernel are process management, file management, memory management and network management. In our next article, we will go into detail about these subsystems.

The organisation of these subsystems in the kernel leads to classification into monolithic and micro-kernels. A kernel that groups all these subsystems together in a single file is called a monolithic kernel. Most General-Purpose Operating Systems (GPOSs) like Linux, Solaris, UNIX and Windows are based on this type of kernel. GPOSs are time-sharing, and are designed to maximise throughput and provide a fair share of system resources in a multi-user/multi-process environment. Figure 1 depicts the architecture of a monolithic kernel-based operating system.

The default time-sharing system design in GPOSs is not appropriate for multimedia applications or, say, an online ticket reservation system. In such systems, critical tasks need to be assigned suitable priorities, which should persist until the task completes. The kernel should provide a facility to set a priority and scheduling policy for critical tasks—this kind of feature is referred to as 'soft real-time'. This is achievable in GPOSs, so most of them, including Linux, are implemented with soft real-time features.

Operating systems for hard real-time applications

A 'Hard Real Time Fatal' system like a car Anti-lock Braking System (ABS), a flight control system, or a nuclear power plant control system needs to complete the execution of critical tasks within a specific time period. If the system misses the deadline, the results will be catastrophic (huge economic consequences or human lives at risk). The kernel should be predictable, and should have deterministic behaviour.

The monolithic GPOS design is not suitable for hard real-time systems, so we need a special-purpose, differently designed, Real-Time Operating System (RTOS)—mostly based on a micro-kernel. Benjamin Roch's discussion on the pros and cons of the monolithic kernel versus the micro-kernel approach can be found via the *References* section [1]. In this arrangement,

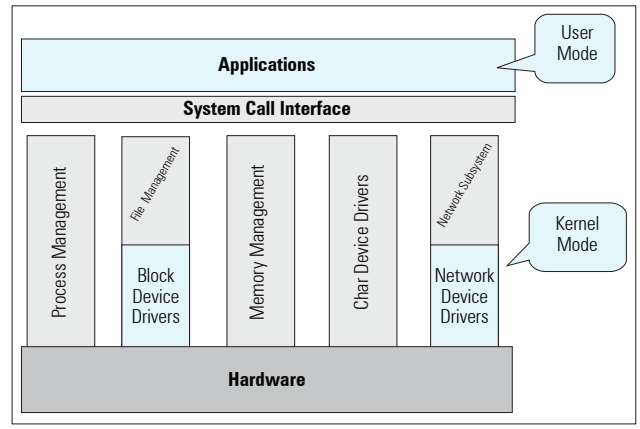

Figure 1: Architecture of a monolithic kernel-based operating system

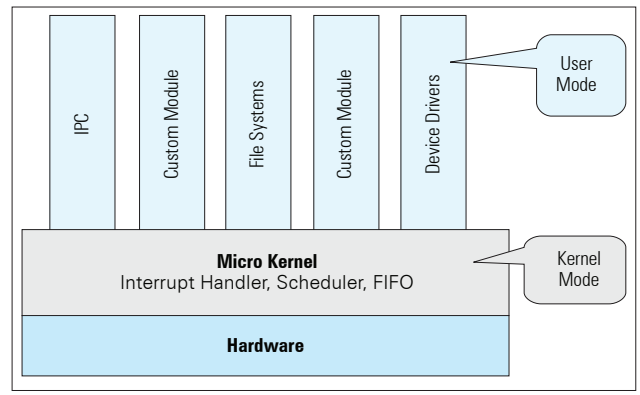

Figure 2: Architecture of a micro kernel

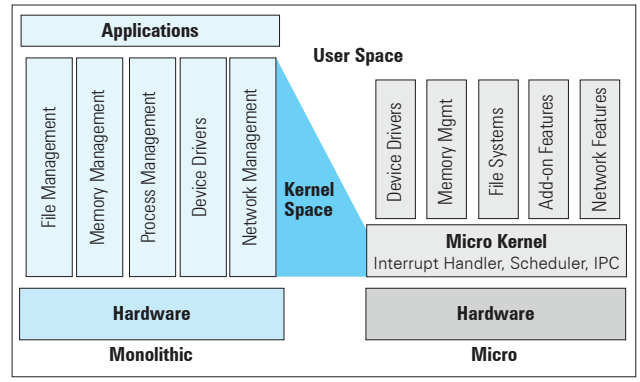

Figure 3: Comparison between monolithic and micro-kernel-based OSs

the kernel houses the essential functions to access hardware like process scheduling, interrupt handling and inter process communications. The other kernel subsystems like memory, file and network management are dynamically loaded as and when required in the user space. The demarcation between what is in the kernel and what is in user-space will differ depending on the design of the product under development, so there is no absolute specification on what constitutes the micro-kernel. Figure 2 depicts the architecture of a typical micro-kernel. Figure 3 is a comparison between monolithic and micro-kernel-based operating systems, showing the differences in what runs in kernel-space and user-space, in each type of kernel.

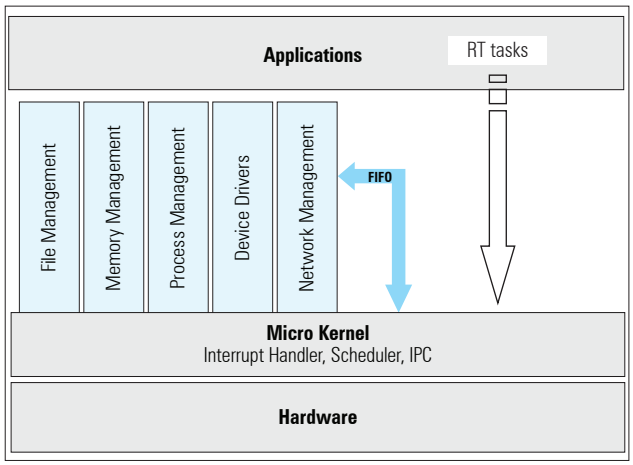

Figure 4: Structure of a hybrid kernel-based operating system

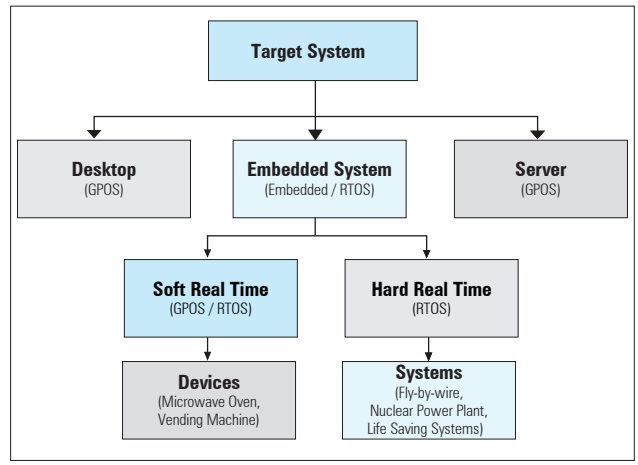

Figure 5: Block diagram of different target systems

Since a micro-kernel's size is lower compared to monolithic kernels, and the kernel is designed for a dedicated system, we can achieve predictability (or worst-case response time) for time-critical tasks. Examples of micro-kernel-based operating systems are Nucleus, eCos, QNX and WinCE. In contrast to our above discussion, we have the VxWorks RTOS, whose kernel is monolithic.

Hybrid kernels

Due to the increasing complexity of embedded devices, at times we need to execute both hard real-time applications and general-purpose applications on a single system. In such a case, both GPOSs and RTOSs are unsuitable: with a GPOS, we cannot get predictable behaviour for the real-time applications, and with an RTOS, the general-purpose applications will considerably degrade system performance. To cater for such a contrasting requirement, we need a hybrid kernel, which offers both real-time and general-purpose features. Real-time applications execute directly in the real-time kernel, and when there are no real-time tasks running, then non-real time tasks are allowed to run. When a real-time task wants to access a certain feature that is not available in the micro-kernel, it exploits a feature from the monolithic kernel via a special communication mechanism using FIFO. Real-Time Application Interface (RTAI), Real-Time Linux (RT Linux) and Windows NT [2] are examples of hybrid kernel-based operating systems. Figure 4 depicts the structure of a hybrid kernel-based operating system.

We can choose an appropriate operating system once we identify our target system as one of the following:

- Desktop or server
- Embedded system
- Embedded system with soft real-time requirements
- Embedded system with hard real-time requirements

Figure 5 is a block diagram with examples of different target systems classified into GPOSs, soft real-time and hard real-time targets.

In the embedded world of consumer electronics (CE) or telecommunications, we should follow standards that specify the necessary features of the system, such as Consumer Electronic Linux Forum (CELF) specifications, or Carrier Grade Linux (CGL) Specifications [3,4].

An architect's ideal design goal

The ideal design goal of the architect is to create a system or solution with acceptable levels of reliability and performance for the given product and customer. Architects should also check if they need to provide a reasonable Time To Market.

Reliability

Reliability is the continuous assurance of the integrity and consistency of the system and all its transactions. It is defined as the probability that a device will perform its required function under stated conditions for a specific period of time. The ability to predict this with some degree of confidence depends on the correct definition of a number of parameters. Good reliability can only be achieved by sound engineering design and working towards perfection in all aspects of the product life cycle. Product reliability is seen as a proof of the robustness of the hardware, platform and application design.

Software reliability is inversely proportional to the number of defects per KLOC (Kilo Lines Of Code) executed on the system, and the reliability of the product is proportional to the MTBF (Mean Time Between Failures).

Performance

Performance means different things in different system scenarios. Performance is a measure of the effectiveness of your application in terms of response time, transaction throughput, and/or resource usage. It involves trade-offs between cost and benefits.

Time To Market

Time To Market (TTM) is the time from when a product or system is conceived, to the time that it is available for sale. Use of Commercial Off-The-Shelf (COTS) products is encouraged, to achieve better time to market. TTM is crucial

for products that are developed from scratch, and for new systems. Leaders have the luxury of a good TTM, whereas followers are always looking for more time. In other words, for a 'blue sky innovation' product, or a 'new paradigm' product, TTM is not a consideration.

Challenges in platform choice

Before choosing an operating system for a given requirement, architects should explore all possible details about the operating system. We now delve into the many questions that the architect should ask, under various areas.

Kernel features

- What are the processor architectures the kernel needs to support (such as multi-core/multi-processor support)?
- What is the type of kernel required (monolithic, micro or hybrid kernel)?
- What is the footprint (size) of the kernel?
- Can I configure the kernel according to my requirements, and reduce the size of the kernel?
- Will the kernel support dynamic kernel module loading?
- How do I efficiently allocate resources for my kernel modules?
  - Resources can be I/O port addresses, Interrupt Request Lines (IRQs), memory space allocation, etc.
- Will the kernel support sharing IRQs for many devices?
- What kernel mechanisms exist to reduce interrupt latency?
- How can I write a very efficient Interrupt Service Routine (ISR) for a given scenario, deferring work in the ISR to minimise interrupt disable time?
- How do I include custom features in the kernel, apart from the standard set of services?
- How can I protect privileged kernel features from accidental or intentional access by a normal user?
- How do I share common functions between different kernel modules?
- What are the different ways of transferring data between kernel-space and user-space?
- How do I synchronise a critical section that may be shared by a process context or interrupt context?
- What are the available kernel locking mechanisms? (Fine-grained locking mechanisms are needed to protect a shared single integer variable, or a hardware bit.)
- Is there any lock-avoidance mechanism available, to protect a critical section from a race condition?
- How do I handle errors in a kernel module?
- How do I identify a deadlock situation in a kernel module?
- How do I do event-driven programming in a kernel module?
- What devices does the kernel support (available device drivers)?
- How do I identify unwanted code in the kernel source?
- How do I forcefully unmount my devices?
- What are the kernel crash scenarios?
- How do I protect critical data in RAM from kernel crashes?
- How do I reboot the system quickly, bypassing BIOS/ firmware?
- What kernel debugging and tracing tools are available?

File management

- What are the file interfaces available?
- How can we modify the existing file system size without affecting the stored data?
- What are the file systems the kernel supports?
- What are the embedded file systems the kernel will support?
- What is the maximum size limit for each file system?
- What are the available pseudo file systems and their usage?
- What file-locking synchronisation mechanisms does the kernel provide?
- What limitations does the OS have in areas such as file size, number of open files within a process, back up facility, etc?

Timers

What are the different timers available in the kernel (see the list below), and what is our choice, based on application and product requirements?

- Monotonic timers
- Periodic timers
- High-resolution timers
- Real-time timers
- POSIX timers
- Hardware timers

Memory management

Our choice of a memory-management policy depends on speed and space requirements, and on whether the application is running on a server, an embedded system or a desktop. Some of the memory management and protection mechanisms available include:

- Virtual memory support
- Kernel space protection
- User-space protection from other processes
- Shared memory support
- Disk traffic time reduction (buffer cache, sync( )).
- Minimise fragmentation (thrashing, page allocation, internal fragmentation, etc)

We also need to know:

- How to extend swap size dynamically
- How to avoid swapping, and lock the process space in physical memory

Process/thread management

We need to understand the following process and thread subsystem parameters before we start to use the system:

- Usage of processes and threads
- Timer interrupt configuration
- Choice of scheduling policy, and the method to set it
- Available priority range (dynamic versus static)
- Pre-emptive versus non-pre-emptive, and whether it's configurable
- Support for real-time process and threads
- Support for signals

- The total number of available signals, and default action
- Signals useful for our purpose
- Priority available for signals
- Presence of real-time signals

- Inter-process communications
  - Communication between related/unrelated processes
  - Inter-thread communication
  - Shared memory set-up and usage
  - Synchronising critical sections
    - Thread synchronisation
    - Priority inheritance support
    - Priority-based waiter queue
    - Deadlock detection tool
  - Communication between different systems
  - Sender-receiver synchronisation if one of them is faster (blocking calls)
  - Error-checking in sender/receiver: if the receiver or sender fails, and an exception or buffer overflow occurs, how does one prevent the failure from spreading? (single point of failure)
  - How to handle a graceful exit or shut-down—unwinding operations before exit
  - Scenarios in which two IPCs will be working together (shared memory/message queue—semaphore)
  - Need for event-driven programming (message notification through signal or creation of a new process or thread, wait queue, completion)
  - Shared re-entrant code between processes (shared read-only portion; for example, text portion of the *vi* editor, shared libraries)
  - Reducing kernel overhead in IPC mechanisms
  - Deriving a real-time response from IPC mechanisms (priority setting, time-out, light-weight IPC, priority inheritance support and priority waiter queue)
  - Self-correction systems (periodically cleaning unused PIDs, file descriptors, IPC ids, *malloc* pointers via *cron* jobs)

Network management

- We should be able to access/communicate with other systems, when the platform is running in a distributed environment, using *telnet* (*teletype network*) or *ssh* (secure shell), or through a Web server.
- Does the platform have support for remote file access, remote debugging and remote updating?
- Does the platform support protocols like TCP and SNMP?
- Does the platform have tools support for monitoring, regulating and blocking traffic on the network?
- Does the platform have enough tools to monitor memory and performance, such as Purify and Quantify? These tools are of great help during integration and systems testing. Architects must validate important test cases during system and integration testing.

Does the platform support enough tests to simulate security threats that will be seen in the actual network? Is the platform robust enough to ward off security threats?

Performance tuning

- What are the code optimisation techniques and tools available?
- How could we reduce context switch latency (pinning to a processor)?
- How do we fine-tune the timer interrupt period for the desktop, server and real-time systems?
- Do we have a provision for a tick-less kernel?
- What performance analysis and monitoring tools are available?
- How to override system limitations?
- What levels of debugging support are available:
  - user level
  - kernel level
  - monitoring tools
  - tracing tools
  - kernel crash dump analysis

Support for standards and forum specifications

The operating system of choice should have compliance for standards/forum specifications, such as:

- Institute of Electrical and Electronics Engineers (IEEE) [5]
- The Open Group [6]
- Portable Operating System Interface [for Unix] (POSIX) [7]
- Linux Standard Base (LSB) specifications [8]
- Embedded Linux Consortium (ELC) platform specifications [9]

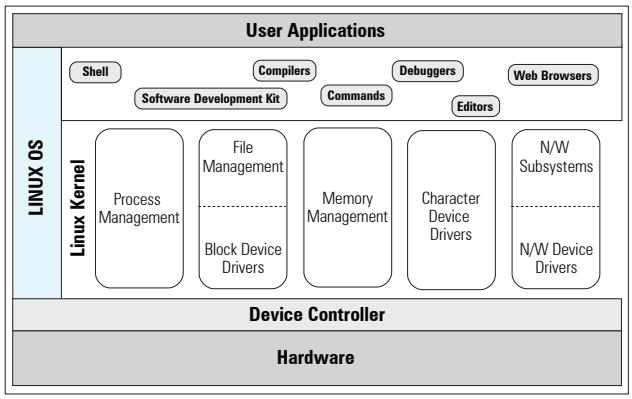

Figure 6: Schematic diagram of Linux architecture

- Carrier Grade Linux (CGL) specifications [4]
- Consumer Electronics Linux Forum (CELF) specifications [3]

Licensing issues—proprietary versus FOSS

The debate over the choice of a proprietary or open source OS is as old as computing science itself, but there are some standard parameters on which this choice should be based:

- Is time to market an important parameter for your product?
- Does your team have adequate FOSS skills?
- Are you selling your product in the volume market or niche market?
- Do you feel licence fees are more important compared to development costs?
- Do you need a custom GUI, tool-set, etc.?
- Is it fine to share your product code with others?
- Is it necessary to maintain your Intellectual Property (IP)?
- Do you want scalability for your product?
- Do you have some serious hard real-time requirements in your product, and hence want to minimise risk?

Vendor support and marketing trends

When do you need vendor support? We have to check for the presence of the following skills in the team:

- Are there adequate open source software skills in your team?
- Is there a business or technical need to use third-party software?
- Is there a need for porting from proprietary to FOSS?
- What middleware would you like to use?
- Are there real-time requirements in your solution or system?
- Are there development tools, such as a platform SDK and debugging tools for monitoring, live updating, efficient power management, security enhancement, remote debugging, etc?
- Does the OS have a minimal kernel footprint?
- What hardware support do you need?

Marketing trends

Let's understand some crucial marketing trends pertaining to the choice of platform:

- What is the market share of the platform to be selected, among the host of companies providing the platform?
- What is the roadmap or future of the OS? This will help us better understand system features, and also compatibility with long-term plans for the product.
- What kind of support does the platform have for different processor architectures, third-party products and tools?

Once we obtain details on all these aspects for the given operating system, we can decide whether the OS will satisfy our project's requirements, or whether we need to look for alternatives.

If the given operating system is Linux, then we should get a clear understanding of how Linux addresses the above aspects. The schematic diagram for Linux architecture is given in Figure 6. The split view of the Linux kernel is discussed in detail and is available at [10].

In the next article, we will explore the details about what kind of support Linux provides in the ten areas we have discussed here, beginning with file management. END

References

1. Monolithic kernel versus micro-kernel, by Benjamin Roch: *http://www.vmars.tuwien.ac.at/courses/akti12/journal/04ss/article_04ss_Roch.pdf*
2. Hybrid kernel: *http://en.wikipedia.org/wiki/Hybrid_kernel*
3. Consumer Electronic Linux Forum home page: *http://celinuxforum.org/*
4. Carrier Grade Linux home page at: *http://www.linuxfoundation.org/collaborate/workgroups/cgl*
5. Institute of Electrical and Electronics Engineers (IEEE) home page at: *http://standards.ieee.org/*
6. The Open Group: *http://www.opengroup.org/onlinepubs/009695399/*
7. Portable Operating System Interface (POSIX): *http://posixcertified.ieee.org/*
8. Linux Standard Base (LSB) specifications: *http://refspecs.freestandards.org/lsb.shtml*
9. Embedded Linux Consortium (ELC) platform specifications: *http://www.linuxfoundation.org/en/ELC*
10. Split view of Linux kernel, Figure 1-1, page 6, available at: *http://lwn.net/images/pdf/LDD3/ch01.pdf*

Gururajan Narasimhan Erode and Dr B. Thangaraju

Gururajan Narasimhan Erode currently works with Wipro Technologies as a lead consultant in the Wireless and Embedded Domain. He has 16 years of core technology experience after a Masters degree in Engineering, and has worked with various multinational companies in the USA, Germany and India. Prior to joining Wipro, he worked with Symbian Software India Ltd, Mascon Global Ltd and Lucent Technologies India. Gururajan's specialisation is in wireless communication, algorithms and data structures, and Linux systems programming. He can be reached at gururajan.erode@wipro.com.

Dr B. Thangaraju received his Ph.D in Physics and worked as a Research Associate in the Indian Institute of Science from 1996 to 2001. From 2001, he has been working at Wipro Technologies as a senior consultant, and his core expertise is the Linux kernel with knowledge of embedded and real-time Linux. He has published more than 35 papers on Linux in renowned international and national journals, and has presented more than 20 technical papers at national and international conferences. He can be reached at balat.raju@wipro.com.