# Kernel Concepts

Presented by Hayden Tremethick.

**In the beginning** the computer was without multitasking, and darkness was upon the face of the multi-users. The first computers were single task, single user '*bare metal'* machines. The executing processes had full control over the memory, the hardware and the tele-type user interface.

The requirement for multi-user, and multi-tasking necessitated the invention of the O**perating System** (**OS**). An OS is a collection of programs that manage a computers resources and provide services common to all application processes. At the heart of an OS is the Kernel. The Kernel is the first process loaded at boot time, and it remains in continuous use for the duration of the session.

In its simplest form a Kernel provides the following services
- a 'scheduler' to allocate CPU (Central Processor Unit) resources
- a 'memory manager'
- IPC (Inter-Process Communication) control.

The Kernel may also provide
- *physical device drivers* (PDD) abstraction layers for peripheral devices (hardware drivers),
- *logical device drivers* (LDD) abstraction layers for,
  - disk filing systems called a Virtual File System (VFS),
  - Protocol Stacks such as Sockets (TCP/IP stack).

**The Scheduler** is a kernel module that queues all current process in order of priority for execution.

**The Memory Manager** creates Virtual Memory Space (VMS) addressing which is decoupled from physical memory addressing and segregates the VMS into protected ***Kernelspace*** and non-protected ***Userspace***, confining each process to a permitted area of VMS.

**IPC** techniques facilitate the flexible, efficient message passing (in various formats, including complex data structures) between processes. IPC and VMS together, provide the basis for ***process isolation***.

**An abstraction layer** provides a common interface layer for one or more underlying physical or logical devices. The side presented to the calling process is a general concept 'system call', which hides the details (abstracts) of the operations on underlying devices. Abstraction layer techniques reduce the duplication of code and enhance security. (Refer to *the Restaurant Anology*)

**Protocol Stack;** because coms protocol modules typically communicate with other coms protocol modules above and/or below (e.g. TCP/IP) they are imagined as layers in a stack. **A socket** is a representation of a *bidirectional* connection between exactly two pieces of communicating software, usually on two separate networked computers. However sockets can also be used to communicate locally (*interprocess*) using 'localhost' on a single computer e.g. *Cups* (http://localhost:631/ or 127.0.0.1:631). Socket addressing is based on a combination of IP address and Port number. Each socket is mapped by the OS to a coms-process.

**Kernels are written in a fast language** such as 'C' to ensure speedy operation.

**Kernelspace** memory is protected memory space reserved for running the Kernel.

**Userspace** is the memory space where all user mode applications work, parts of this memory can be 'swapped out' as required. Each user process normally runs in its own memory space and cannot access the memory space of other processes. This is the basis of 'memory protection' and a building block in 'privilege separation'. Depending on the privileges granted to a process it may be able to ask the kernel to map part of another process's memory space as part of its own space, e.g. debuggers.

**Userland** refers to the programs and libraries that run in user space and interface with the kernel to perform input/output and file system manipulation etc. Userland software effectively provides a 'wraper' around the userspace to provide communication between the userspace programs and the kernel.

A **system call** is request for service message that starts as a request from a userspace process to a userland library API (Application Program Interface). Refering to the *'restaurant analogy*' below, this is the equivalent to placing a menu request with the waiter. The userland library processes the API request and places a system call with the Kernel. Linux has over 300 different calls available to libraries, however most system calls can be grouped into the following five categories:
**Process Control**. eg load, execute, start and terminate processes.
**File management**. eg open, close, create and delete files.
**Device Management.** e.g. request & release devices, read from or write to devices.
**Information Maintenance.** e.g. get/set time, date; get/set file or device attributes etc.
**Communication.** e.g. create, delete a coms connection send, receive messages.

A Linux **Virtual File System (VFS)** is a logical abstraction layer for file systems. The VFS enables userland applications to access any mounted filesystem without regard to the actual of file-system on the physical device. VFS makes it relatively easy to add support for new file system types.

**CPU modes** (selectable by machine code commands) place restrictions on the type and scope of operations being run by the CPU. Restricted modes typically restrict access to certain areas of memory and restrict or prevent input/output operations. This enables kernel space processes to run with unrestricted privileges while restricting the privileges of userspace processes.

CPUs that support 'mode' operation will offer at least two modes, an unrestricted *kernel mode* and a restricted *user mode*. Some CPU architectures support multiple *user modes* with a hierarchy of privileges. This allows a *hypervisor* in kernel mode to run multiple operating systems s beneath it

System calls and IPC require switching modes which takes time and slows system performance so some OS designs allow time-critical software (e.g. device drivers) to run with full privileges.

**Kernel Types**
- Micro-kernel,
- Monolithic-kernel
- Hybrid Kernel.

**Microkernels** have minimal 'built in' functions, scheduling, memory management, and IPC. All other OS functions are assigned to userland e.g. drivers for hardware but controlled by the kernel.

The Microkernel has the advantage that device drivers and window managers are in the user space and can be coded in languages not used by the kernel and also easily replaced without modifying the kernel. Another advantage (in theory) is if a driver 'crashes' it can be reloaded and restarted without disruption to the kernel.. Disadvantages include a very heavy IPC overhead. 'Mach' and 'MINIX' are example of microkernels. Microkernels were a *very hot* item in the late 1980s & early 1990s.

The **Monolithic-kernel** has all the microkrnel functionality plus hardware drivers, virtual file system (VFS) and Protocol Stacks built in. This reduces inter-process communication requirements and enhances system security but requires careful system design and reduces flexibility available to the designer. Unix and Linux are examples of monolithic kernels.

**Hybrid-kernels** . attempt to get the best of both worlds, by using a mix of micro-kernel and

monolithic kernel techniques;. for example Windows has a few basic drivers built into the kernel with all additional drivers in userland. Unfortunately some hybrid-kernel designs seem to incorporate the worst of both worlds. Windows and Mac claim their kernels are hybrid.

**Monolithic vs Hybrid (Linux vs Windows).** Windows has the advantage of a degree of hardware flexibility. When a new piece of hardware comes on the market, it's usable immediately upon installing the OEM driver. The converse to this is, if you upgrade your OS version, some peripheral devices may not work unless the OEM has released a driver compatible with the new OS. However once a Linux driver is in the Kernel your peripheral will work until it is worn out.

Windows needs to load into userland many drivers (along with associated 'crudware') and associated processes just in case they may be required. Linux can dynamically load kernel drivers when they are required. Linux has more than 6,800 drivers and still runs lean and mean

System calls in hybrid kernels hit OS performance because they require a complex flow of IPC "housekeeping" messages along with an associated potential of introducing security holes. This security problem is compounded by allowing some device drivers to run with elevated privileges to improve system speed.

Linux, with it's Monolithic structure, needs a lot less "housekeeping" messaging because system tasks are internal to the kernel.

Just for fun consider the classic animal hybrid, the mule, it is the offspring of a donkey(micro) over a mare(monolithic). A mule may have some good characteristics but you can't change the fact that its ugly, it's sterile (no long term future) and you will never win a race riding a mule; there you have the reasons why 'big iron' and supercomputers don't use Windows or Mac OSs.

The last word goes to Linus.
"As to the whole "hybrid kernel" thing - it's just marketing." Linus Torvalds 09/05/2006

**The Restaurant Anology -**, APIs, libraries, system calls, abstraction, IPC in the real world. When you (the user process) chose from a Menu and place an order with the Waiter (Menu&Waiter=API&Library), the Waiter processes the order details places one (or more) *system calls* on the kitchen; the kitchen 'abstracts' the the detail of preparing the meal and *returns the call* to the Waiter who in turn returns with your meal. The Waiter and the Kitchen (abstraction layer) will have IPC to ensure the details of your order are satisfied.

**Kernel Names**
vmlinuz
C:\Windows\system32\Ntoskrnl.exe
XNU K32/K64