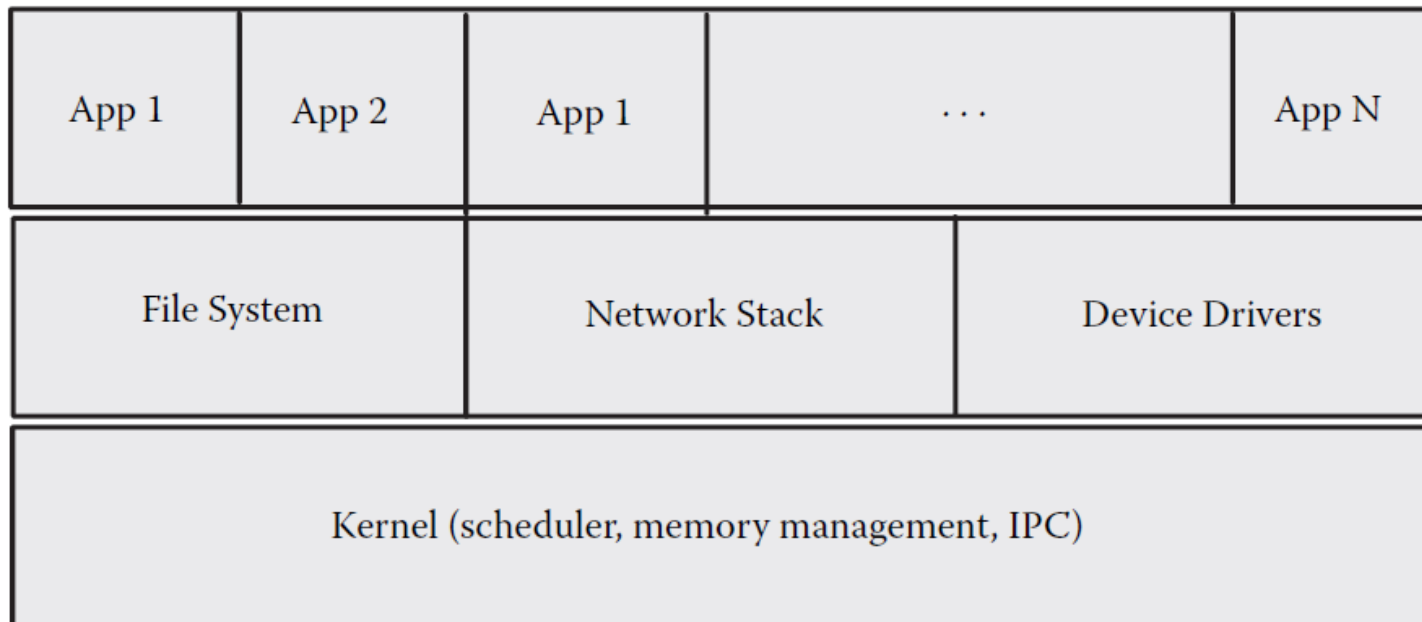

Embedded Linux Architecture

Types of Operating Systems

- Real-Time Executive
- Monolithic Kernel
- Microkernel

Real-Time Executive

- For MMU-less processors
- The entire address space is flat or linear with no memory protection between the kernel and applications



Real-Time Executive

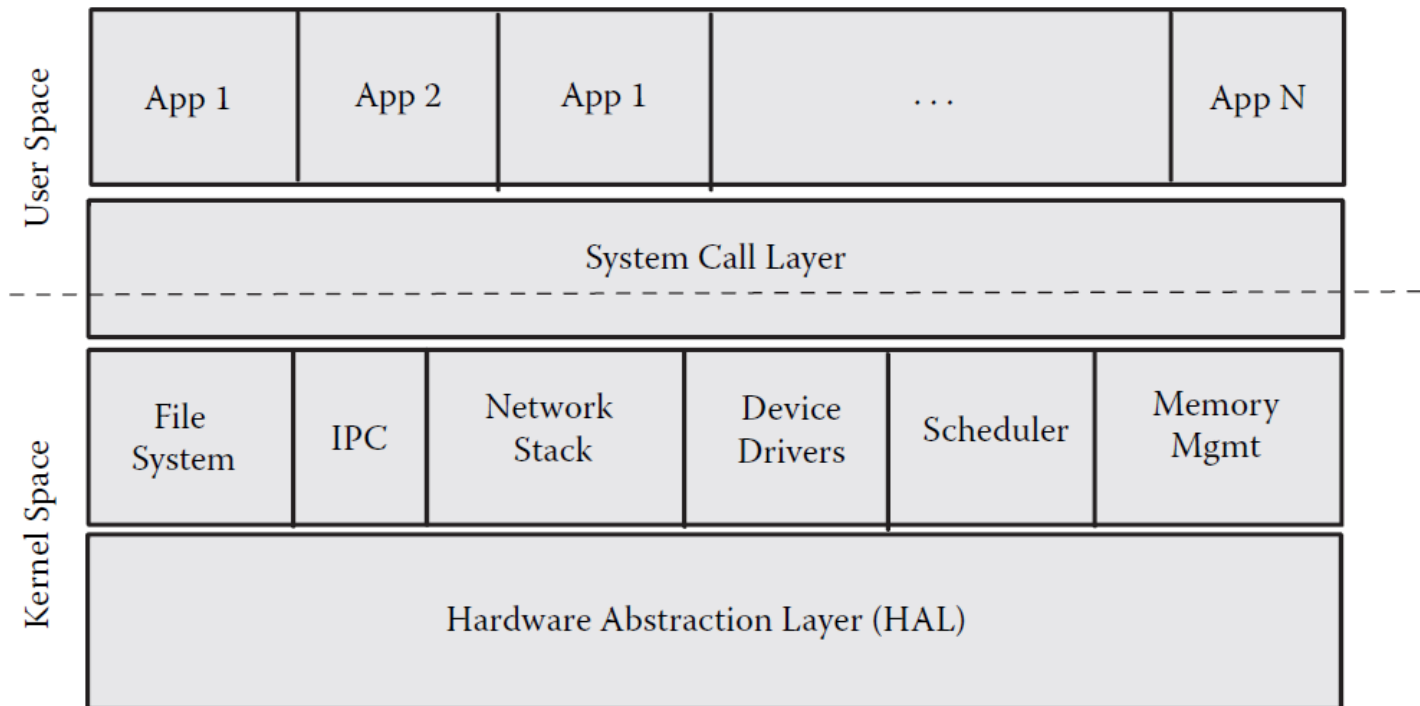
- the core kernel, kernel subsystems, and applications share the same address space.
- These operating systems have small memory and size footprint as both the OS and applications are bundled into a single image.
- real-time in nature because there is no overhead of system calls, message passing, or copying of data.
- because the OS provides no protection, all software running on the system should be foolproof.
- Reliability on real-time executives using the flat memory model was achieved by a rigorous testing process.

Monolithic Kernels

- Monolithic kernels have a distinction between the user and kernel space.
- When software runs in the user space normally it cannot access the system hardware nor can it execute privileged instructions. Using special entry points (provided by hardware), an application can enter the kernel mode from user space.
- The user space programs operate on a virtual address so that they cannot corrupt another application's or the kernel's memory.
- However, the kernel components share the same address space; so a badly written driver or module can cause the system to crash.

Monolithic Kernels

- the kernel and kernel submodules share the same address space and where the applications each have their private address spaces.



Monolithic Kernels

- Monolithic kernels can support a large application software base. Any fault in the application will cause only that application to misbehave without causing any system crash. Also applications can be added to a live system without bringing down the system. Most of the UNIX OSs are monolithic.

Microkernel

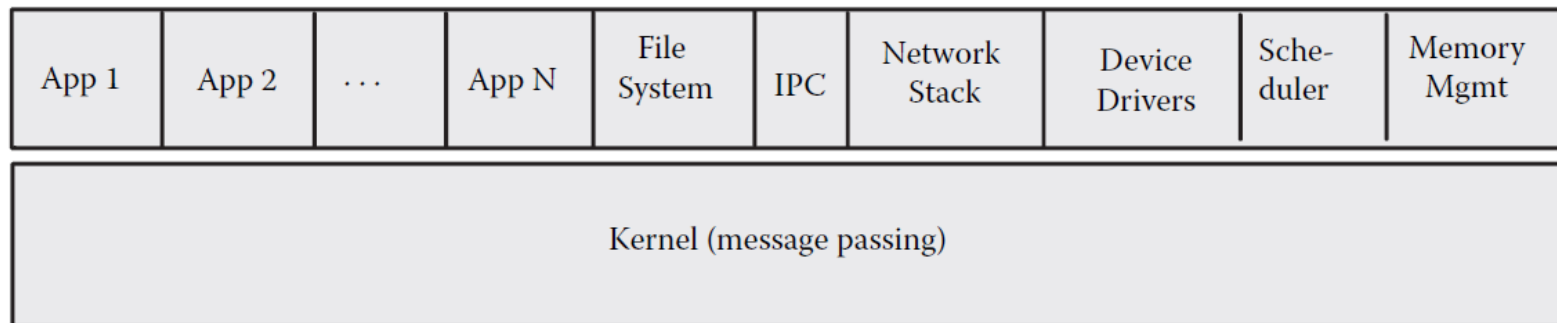
- Microkernels have been subjected to lots of research especially in the late 1980s and were considered to be the most superior with respect to OS design principles.
- The microkernel makes use of a small OS that provides the very basic service(scheduling, interrupt handling, message passing) and the rest of the kernel(file system, device drivers, networking stack) runs as applications.

Microkernel

- On the usage of MMU, the real-time kernels form one extreme with no usage of MMU whereas the microkernels are placed on the other end by providing kernel subsystems with individual address space.
- The key to the microkernel is to come up with well-defined APIs for communication with the OS as well as robust message-passing schemes.

Microkernel

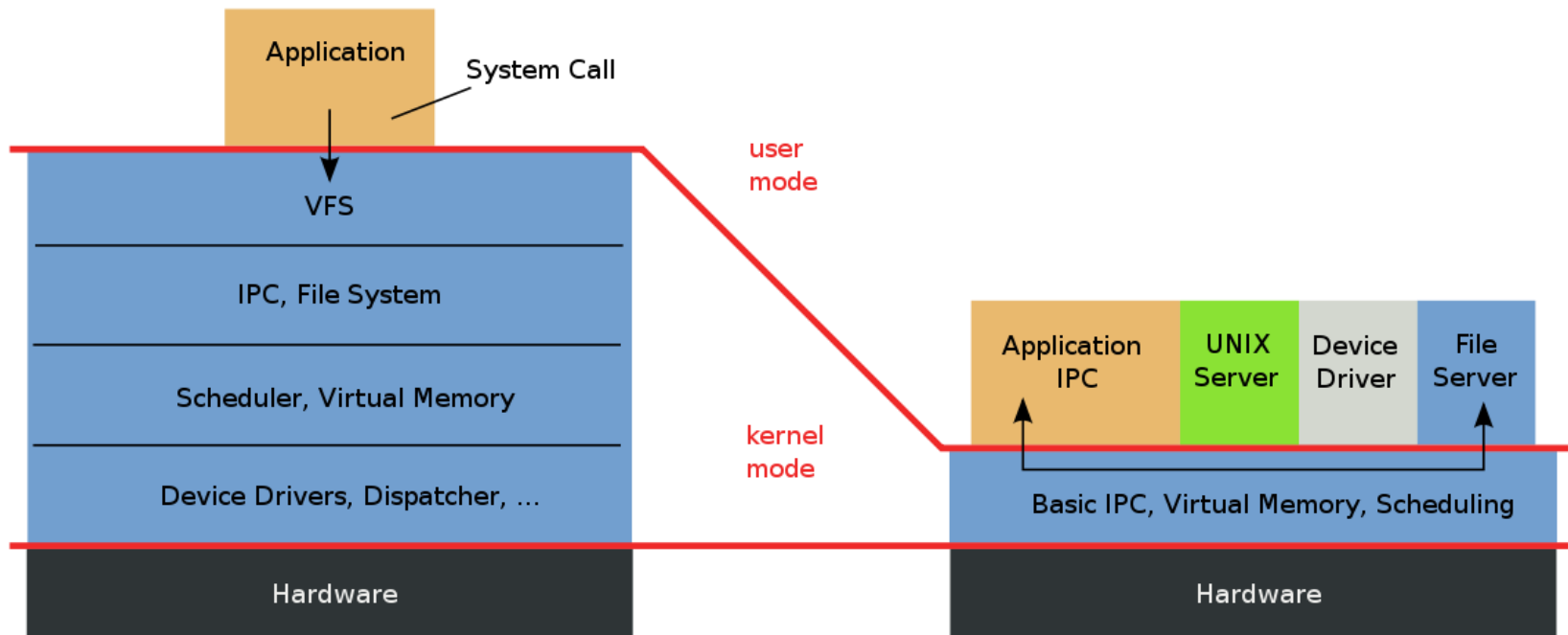
- kernel subsystems such as network stack and file systems have private address space similar to applications.



Monolithic Kernel vs Micro Kernel

Monolithic Kernel
based Operating System

Microkernel
based Operating System



Three types of OS

- On one end of the spectrum we have the real-time kernel that provides no memory protection; this is done to make the system more real-time but at the cost of reliability.
- On the other end, the microkernel provides memory protection to individual kernel subsystems at the cost of complexity.
- Linux takes the middle path of monolithic kernels where the entire kernel operates on a single memory space.

Linux Kernel Architecture

- The hardware abstraction layer
- Memory manager
- Scheduler
- File system
- IO subsystem
- Networking subsystem
- IPC

Hardware Abstraction Layer (HAL)

- The hardware abstraction layer (HAL) virtualizes the platform hardware so that the different drivers can be ported easily on any hardware. The HAL is equivalent to the BSP provided on most of the RTOSs except that the BSP on commercial RTOSs normally has standard APIs that allow easy porting.
- Embedded processors (other than x86) supported on the Linux 2.6 kernel – MIPS, PowerPC, ARM, M68K, SuperH, etc.

Hardware supported by HAL

- Processor, cache, and MMU
- Setting up the memory map
- Exception and interrupt handling support
- DMA
- Timers
- System console
- Bus management
- Power management

Memory Manager

- The memory manager on Linux is responsible for controlling access to the hardware memory resources. The memory manager is responsible for providing dynamic memory to kernel subsystems such as drivers, file systems, and networking stack.
- It also implements the software necessary to provide virtual memory to user applications. Each process in the Linux subsystem operates in its separate address space called the virtual address. By using virtual address, a process can corrupt neither another process's nor the operating system's memory.

Scheduler

- The Linux scheduler provides the multitasking capabilities and is evolving over the kernel releases with the aim of providing a deterministic scheduling policy.

Scheduler

- *Kernel thread*: These are processes that do not have a user context. They execute in the kernel space as long as they live.
- *User process*: Each user process has its own address space thanks to the virtual memory. They enter into the kernel mode when an interrupt, exception, or a system call is executed. Note that when a process enters the kernel mode, it uses a totally different stack. This is referred to as the kernel stack and each process has its own kernel stack.

Scheduler

- *User thread*: The threads are different execution entities that are mapped to a single user process. The user space threads share a common text, data, and heap space. They have separate stack addresses. Other resources such as open files and signal handlers are also shared across the threads.
- The 2.6 kernel has a totally new scheduler referred to as the $O(1)$ scheduler that brings determinism into the scheduling policy. Also more real-time features such as the POSIX timers were added to the 2.6 kernel.

File System

- On Linux, the various file systems are managed by a layer called the VFS or the Virtual File System. The virtual file system provides a consistent view of data as stored on various devices on the system.
- Any Linux device, whether it's an embedded system or a server, needs at least one file system. The Linux necessity of file systems stems from two facts.
 - The applications have separate program images and hence they need to have storage space in a file system.
 - All low-level devices too are accessed as files.

-
- It is necessary for every Linux system to have a master file system, the root file system. This gets mounted at system start-up. Later many more file systems can be mounted using this file system.
 - Linux supports specialized file systems that are flash- and ROM-based for embedded systems. Also there is support for NFS on Linux, which allows a file system on a host to be mounted on the embedded system.

IO Subsystem

- The IO subsystem on Linux provides a simple and uniform interface to onboard devices. Three kinds of devices are supported by the IO subsystem.
 - **Character devices** for supporting sequential devices.
 - **Block devices** for supporting randomly accessible devices. Block devices are essential for implementing file systems.
 - **Network devices** that support a variety of link layer devices.

Networking Subsystems

Feature	Kernel Availability		
	2.2	2.4	2.6
Layer 2			
Support for bridging	Yes	Yes	Yes
X.25	Yes	Yes	Yes
LAPB	Experimental	Yes	Yes
PPP	Yes	Yes	Yes
SLIP	Yes	Yes	Yes
Ethernet	Yes	Yes	Yes
ATM	No	Yes	Yes
Bluetooth	No	Yes	Yes
Layer 3			
IPV4	Yes	Yes	Yes
IPV6	No	Yes	Yes
IP forwarding	Yes	Yes	Yes
IP multicasting	Yes	Yes	Yes
IP firewalling	Yes	Yes	Yes
IP tunneling	Yes	Yes	Yes
ICMP	Yes	Yes	Yes
ARP	Yes	Yes	Yes
NAT	Yes	Yes	Yes
IPSEC	No	No	Yes
Layer 4 (and above)			
UDP and TCP	Yes	Yes	Yes
BOOTP/RARP/DHCP	Yes	Yes	Yes

- The interprocess communication on Linux includes signals (for asynchronous communication), pipes, and sockets as well as the System V IPC mechanisms such as shared memory, message queues, and semaphores.
- The 2.6 kernel has the additional support for POSIX-type message queues.

User Space

- **Program:** This is the image of an application. It resides on a file system. When an application needs to be run, the image is loaded into memory and run. Note that because of virtual memory the entire process image is not loaded into memory but only the required memory pages are loaded.
- **Virtual memory:** This allows each process to have its own address space. Virtual memory allows for advanced features such as shared libraries. Each process has its own memory map in the virtual address space; this is unique for any process and is totally independent of the kernel memory map.
- **System calls:** These are entry points into the kernel so that the kernel can execute services on behalf of the application.

Linux Start-Up Sequence

- ***Boot loader phase***: Typically this stage does the hardware initialization and testing, loads the kernel image, and transfers control to the Linux kernel.
- ***Kernel initialization phase***: This stage does the platform-specific initialization, brings up the kernel subsystems, turns on multitasking, mounts the root file system, and jumps to user space.
- ***User- space initialization phase***: Typically this phase brings up the services, does network initialization, and then issues a log-in prompt.

Boot Loader Phase

- Hardware Initialization
 1. Configuring the CPU speed
 2. Memory initialization, such as setting up the registers, clearing the memory, and determining the size of the onboard memory
 3. Turning on the caches
 4. Setting up the serial port for the boot console
 5. Doing the hardware diagnostics or the POST (Power On Self-Test diagnostics)

Boot Loader Phase

- Downloading Kernel Image and Initial Ram Disk
 - The boot loader needs to locate the kernel image, which may be on the system flash or may be on the network. In either case, the image needs to be loaded into memory.
- Setting Up Arguments
 - Argument passing is a very powerful option supported by the Linux kernel. Linux provides a generic way to pass arguments to the kernel across all platforms. Typically the boot loader has to set up a memory area for argument passing, initialize it with the required data structures (that can be identified by the Linux kernel), and then fill them up with the required values.

Boot Loader Phase

- Jumping to Kernel Entry Point
 - The kernel entry point is decided by the linker script when building the kernel (which is typically present in linker script in the architecture-specific directory). Once the boot loader jumps to the kernel entry point, its job is done and it is of no use.

Kernel Start-Up

- CPU/Platform-Specific Initialization
- Subsystem Initialization
 - This includes
 - Scheduler initialization
 - Memory manager initialization
 - VFS initialization
 - Note that most of the subsystem initialization is done in the `start_kernel()` function. At the end of this function, the kernel creates another process, the `init` process, to do the rest of the initialization (driver initialization, `initcalls`, mounting the root file system, and jumping to user space) and the current process becomes the idle process with process id of 0.

Kernel Start-Up

- Driver Initialization
- Mounting Root File System
 - the root file system is the master file system using which other file systems can be mounted. Its mounting marks an important process in the booting stage as the kernel can start its transition to user space.
 - There are three kinds of root file systems that are normally used on embedded systems:
 - The initial ram disk
 - Network-based file system using NFS
 - Flash-based file system

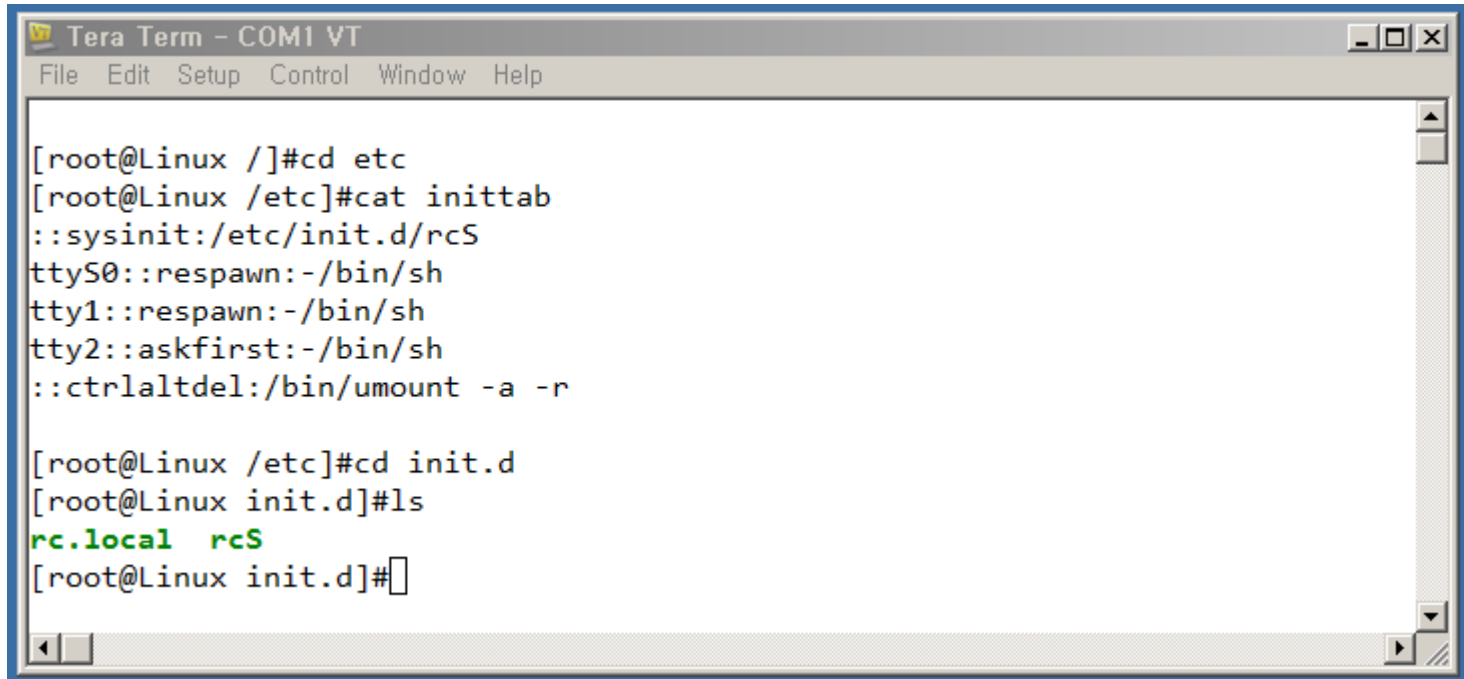
Kernel Start-Up

- Doing Initcall and Freeing Initial Memory
- Moving to User Space
 - The kernel that is executing in the context of the init process jumps to the user space by overlaying itself (using `execve`) with the executable image of a special program also referred to as `init`. This executable normally resides in the root file system in the directory `/sbin`. Note that the user can specify the `init` program using a command line argument to the kernel.

User Space Initialization

- User space initialization is distribution dependent. The responsibility of the kernel ends with the transition to the init process. What the init process does and how it starts the services is dependent on the distribution.
- The `/sbin/init` Process and `/etc/inittab`
 - The init process can be configured on any system using the inittab file, which typically resides in the `/etc` directory. init reads the inittab file and does the actions accordingly in a sequential manner.

/etc/inittab

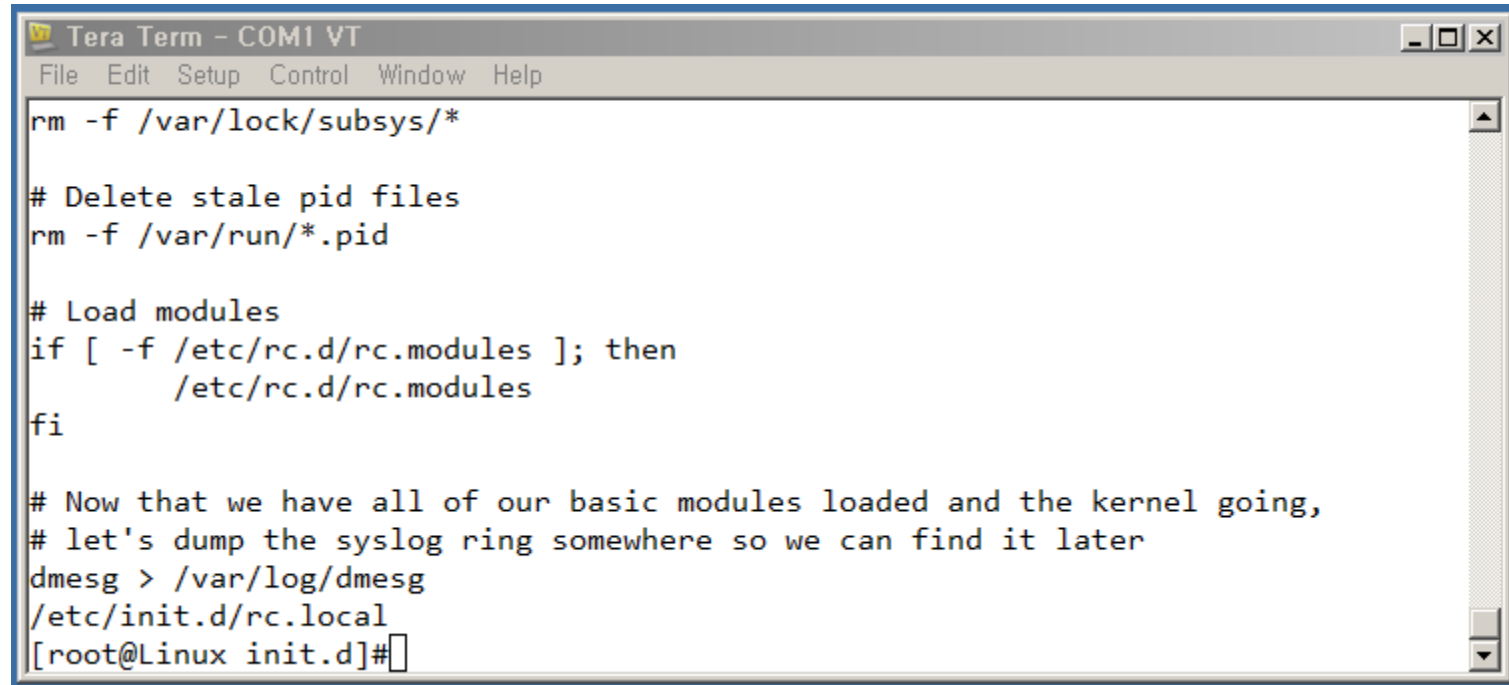


```
Tera Term - COM1 VT
File Edit Setup Control Window Help

[root@Linux /]#cd etc
[root@Linux /etc]#cat inittab
::sysinit:/etc/init.d/rcS
ttyS0::respawn:-/bin/sh
tty1::respawn:-/bin/sh
tty2::askfirst:-/bin/sh
::ctrlaltdel:/bin/umount -a -r

[root@Linux /etc]#cd init.d
[root@Linux init.d]#ls
rc.local rcS
[root@Linux init.d]#
```

/etc/init.d/rcS



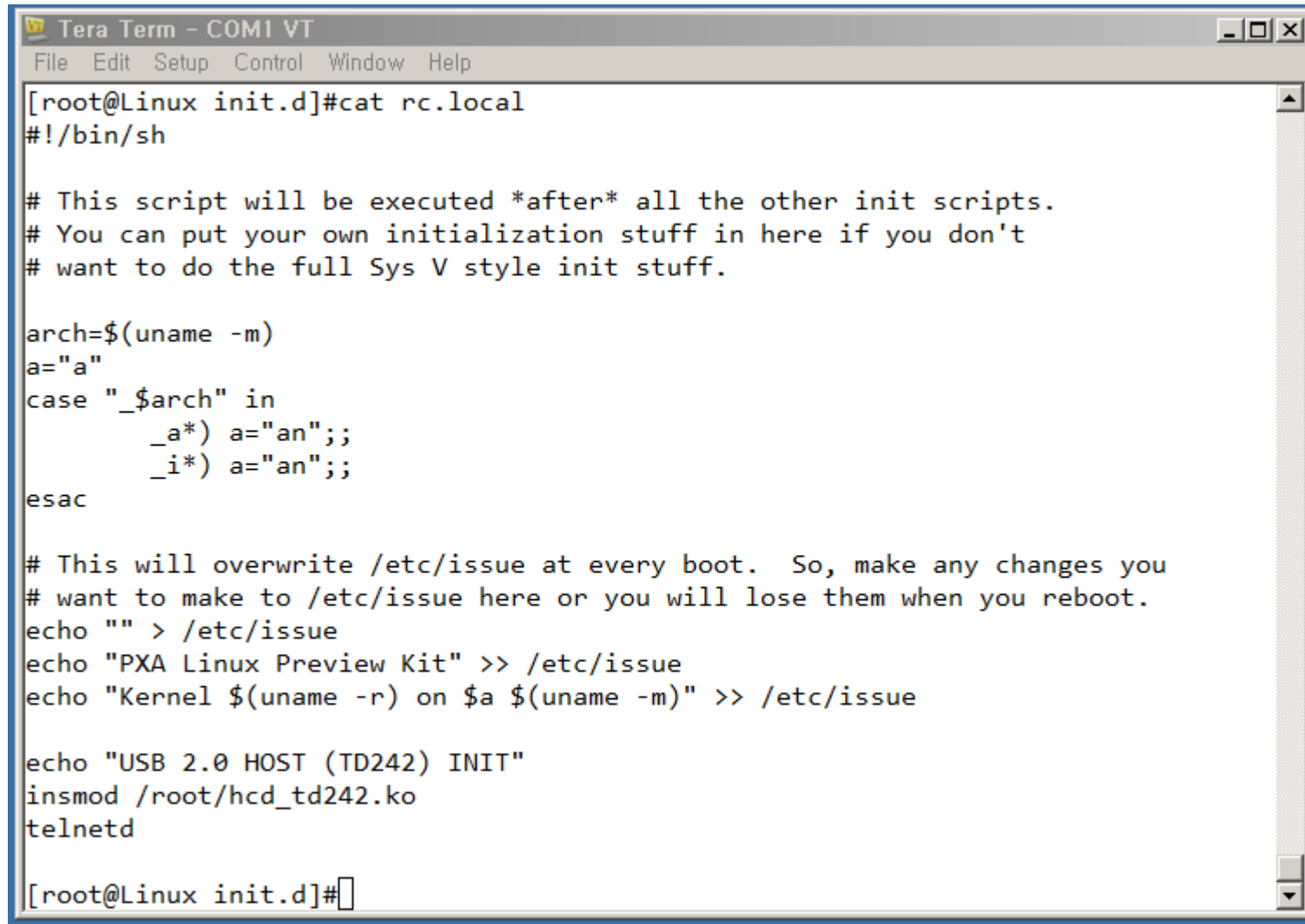
```
Tera Term - COM1 VT
File Edit Setup Control Window Help
rm -f /var/lock/subsys/*

# Delete stale pid files
rm -f /var/run/*.pid

# Load modules
if [ -f /etc/rc.d/rc.modules ]; then
    /etc/rc.d/rc.modules
fi

# Now that we have all of our basic modules loaded and the kernel going,
# let's dump the syslog ring somewhere so we can find it later
dmesg > /var/log/dmesg
/etc/init.d/rc.local
[root@Linux init.d]#
```

/etc/init.d/rc.local



```
Tera Term - COM1 VT
File Edit Setup Control Window Help
[root@Linux init.d]#cat rc.local
#!/bin/sh

# This script will be executed *after* all the other init scripts.
# You can put your own initialization stuff in here if you don't
# want to do the full Sys V style init stuff.

arch=$(uname -m)
a="a"
case "$arch" in
    _a*) a="an";;
    _i*) a="an";;
esac

# This will overwrite /etc/issue at every boot. So, make any changes you
# want to make to /etc/issue here or you will lose them when you reboot.
echo "" > /etc/issue
echo "PXA Linux Preview Kit" >> /etc/issue
echo "Kernel $(uname -r) on $a $(uname -m)" >> /etc/issue

echo "USB 2.0 HOST (TD242) INIT"
insmod /root/hcd_td242.ko
telnetd

[root@Linux init.d]#
```

GNU Cross-Platform Toolchain

- Binutils: Binutils are a set of programs necessary for compilation/linking/ assembling and other debugging operations.
- GNU C compiler: The basic C compiler used for generating object code (both kernel and applications).
- GNU C library: This library implements the system call APIs such as open, read, and so on, and other support functions. All applications that are developed need to be linked against this base library.

GNU Cross-Platform Toolchain

- ARM: <http://www.emdebian.org/>
- PPC: [http:// www.emdebian.org/](http://www.emdebian.org/)
- MIPS: <http://www.linux-mips.org/>
- M68K: <http://www.uclinux.org/>

Target Name

- arm-linux: Support for ARM processors such as armV4, armv5t, and so on.
- mips-linux: Support for various MIPS core such as r3000, r4000, and so on.
- ppc-linux: Linux/PowerPC combination with support for various PPC chips.
- m68k-linux: This targets Linux running on the Motorola 68k processor.