Embedded drops – Every drop makes an Ocean

Sharing Embedded system ideas and concepts, that i learned in my career

Archive for October, 2012

Power Management in Linux-Based Systems

Posted in Embedded Linux on October 24, 2012 | Leave a Comment »

Implementing power management in any system is a complex task. Here’s how to manage your system’s transitions from normal run state to power-saving modes.
Power management (PM) software is a crucial component in battery-powered systems, such as PDAs and laptops, because it helps conserve power when the system is inactive. As a simple example, power may be conserved by switching off the display when a system is inactive for some time. Conserving power in this manner extends battery life, so one can work more hours before having to recharge the battery.

Hardware support is vital for power management to work, and software intelligently exercises that support. The degree of power management support available in hardware varies from device to device. Some devices, such as a display, simply provide two power states, on and off. Other devices, like the SA1110 CPU, may support more complex power-saving features, including frequency scaling.

Implementing power management in any system is a complex task, considering that several non-interacting subsystems need to be brought together under a single set of guidelines. This article explains how power management works in Linux (2.4.x) and how it can be implemented in battery-powered systems based on an APM standard, at both the device driver and application levels.

Two Power Management Standards

Power management for computer systems has matured over the years and several standards exist. The two popular ones are advanced power management (APM) and advanced configuration and power interface (ACPI). APM is a standard proposed by Microsoft and Intel for system power management, and it consists of one or more layers of software to support power management. It standardizes the information flow across those layers. In the APM model, BIOS plays a key role. ACPI is the newer of the two technologies, and it is a specification by Toshiba, Intel and Microsoft for defining power management standards. ACPI allows for more intelligent power management, as it is managed more by the OS than by the BIOS. Although both standards are more popular in x86-based systems, it is possible to implement them in other architectures.

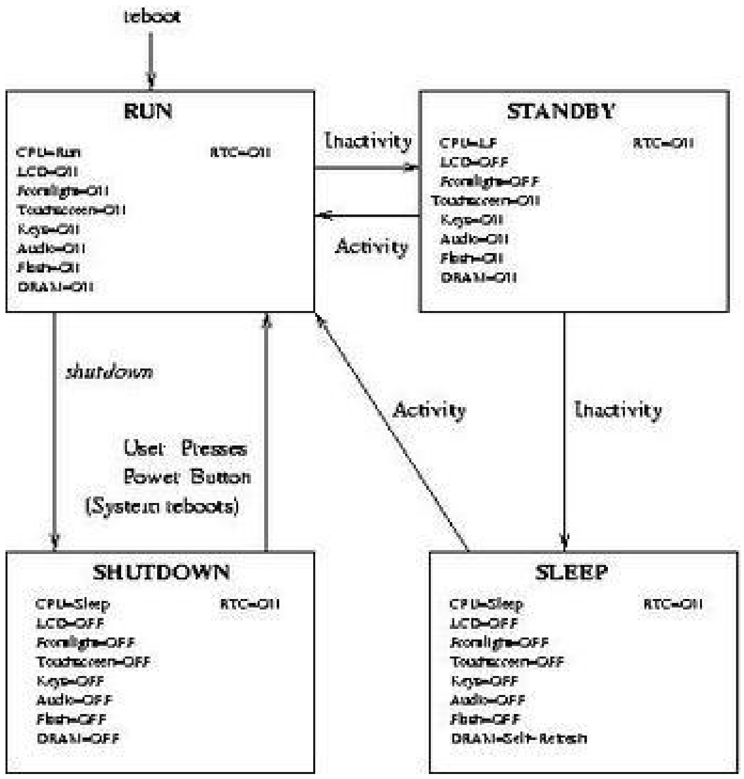
Power Management Implementation

Before implementing power management, it is important to understand what hardware support is available for saving power. One of the important goals of power management software is to keep all devices in their low power states as much as possible.

A possible approach for implementing power management is first to define a power state transition diagram. This defines several power states for the system and also defines the rules and events governing state transitions.

As an example, consider a PDA that has the following devices: Intel SA1110 CPU, real-time clock, DRAM, Flash, LCD, front light, UART, audio codec, touchscreen, keys and power button. The Intel SA1110 CPU supports several power-saving features, including frequency scaling, where the core clock frequency can be configured by software. Lowering clock frequency reduces the CPU's power consumption, but at the cost of reduced CPU speed. This CPU also supports several modes of operation:

- Run mode: the normal state of operation for the SA1110 when it is executing code. All power supplies are enabled, all clocks are running and every on-chip resource is functional.
- Idle mode: allows software to stop the CPU when not in use. In this mode, the CPU clock is stopped, representing some savings in power. All other on-chip resources are active. When an interrupt occurs, the CPU is reactivated.
- Sleep mode: offers the greatest power savings and consequently the lowest level of available functionality. In this mode, power is switched off to the majority of the processor. Some preprogrammed event, such as a power button press, wakes up the CPU from this modeAs you can see, software is responsible for transitioning the CPU either to idle mode or sleep mode.In such a PDA, DRAM cells normally are refreshed periodically by the memory controller logic present inside the CPU. In sleep state, however, the majority of the CPU is shut off, which results in DRAM cells not being refreshed, which in turn leads to loss of data in DRAM. To avoid this loss, most DRAMs support a mode called self-refresh wherein the DRAM itself takes care of refreshing its cells. In such cases, software can put DRAM in its self-refresh mode by writing to a few control registers before transitioning the CPU to its sleep mode, thereby preserving the DRAM contents.The top power-hungry devices in this PDA can be the CPU, DRAM and display back light. Hence, they should be kept in their low power states as much as possible.



(http://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/066/6699/6699f1.jpg)

Figure 1. Power State Transition Diagram

Figure 1 shows a possible power state transition diagram for this PDA. Here is a brief description of the power states:

- Run state: system falls into this default state when it reboots. Power consumption is maximum in this state, as all devices are turned on or active.
- Standby state: system falls into this state due to inactivity. LCD and display back light are turned off, and CPU clock speed is reduced to save some power.
- Sleep state: system falls into this state due to continued inactivity. Power is conserved aggressively by putting the CPU in sleep mode, which in turn powers off most devices. DRAM, however, is put in its self-refresh mode to preserve the machine state (system and application text/data loaded in memory) while the system is sleeping. The system awakens from sleep state when a preprogrammed event occurs. When it wakes up, it transitions to the run state and machine state is restored.
- Shutdown state: system falls into this state when the shutdown command is issued. The system reboots when it exits from this state. This means it is not necessary to preserve the machine state in DRAM, and hence DRAM can be powered off. The shutdown state then represents the lowest power consumption state of all.

The real-time clock is kept on in all power states to retain system time.

It is clear from this diagram that detecting inactivity and putting the devices in their low power states forms the heart of power management software.

Linux and Power Management
Power management software manages state transitions in association with device drivers and applications. It intimates all PM events, including standby transition, sleep transition and low battery, when they occur. This allows software to veto certain state transitions when it is not safe to do so.

Device drivers generally are responsible for saving device states before putting them into their low power states and also for restoring the device state when the system becomes active.

Generally, applications are not involved in power management state transitions. A few specialized applications, which deal directly with some devices, may want to participate. This section explains what device drivers need to do in order to participate in power management:

- pm_dev structure: the PM subsystem in the Linux kernel maintains some information in a pm_dev structure about every registered driver. Maintaining this information allows it to notify all registered drivers about PM events.
- pm_register: device drivers first have to register themselves with the PM subsystem before participating in power management. They do this by calling pm_register:
  ```
  struct pm_dev *pm_register(pm_dev_t type, unsigned
  long id, pm_callback cbackfn);
  ```
  where *type* is the type of device being managed by the driver, *id* is the device ID and *cbackfn* is a pointer to some function in device driver. This is called as the driver's callback function.

The linux/pm.h file defines the various types and IDs that can be used by drivers. If successful, pm_register returns a pointer to a structure of type pm_dev. A driver's callback function is invoked by the PM subsystem whenever there is a PM event. The following arguments are passed to the function:

- dev: a pointer to the pm_dev structure that represents the device; the same pointer returned by pm_register.
- event: identifies the PM event type. The possible events are PM_STANDBY, meaning the system is going into standby state; PM_SUSPEND, meaning the system is going into suspend state; and PM_RESUME, meaning the system is resuming (from either standby or sleep states). Based on implementation, more events can be supplied.
- Software
  Implementing power management in any system is a complex task. Here's how to manage your system's transitions from normal run state to power-saving modes
  - Data: data, if any, associated with the request.

  Each device driver is supposed to do some processing according to the PM event type. In a PM_SUSPEND event, for example, the LCD driver is supposed to save the device state and then switch off the LCD. If it is a PM_RESUME event, the LCD driver should switch on the LCD and restore its state from the saved state.

  The callback function should return an integer value. Returning a value of zero signifies that the driver agrees to the PM event. A nonzero value signifies that the driver does not agree to the PM event. This may cause the state transition in progress to be aborted. For example, if a PM_SUSPEND event is sent to the LCD driver's PM callback function and it returns 1, the suspend operation is aborted.

  All the driver's callback functions are invoked in a predefined order. This is on a last-come-first-served basis, which can be a problem if two devices depend on each other. Let's say the interface to a Bluetooth (BT) device is through a USB host controller (HC). The Bluetooth driver needs this interface to be up before it can talk to the BT device. Because of this dependency, the USB HC driver is loaded before the BT driver. This means the USB HC driver registers with PM before the BT driver.

  Whenever a system wants to transition to sleep state, a PM_SUSPEND request is sent first to the BT driver and then to the USB driver. The USB HC driver may shut off the BT port as part of its PM_SUSPEND processing. When the system resumes, PM_RESUME is sent first to the BT driver and then to the USB HC driver. At the time when the BT driver processes this request, its interface to the BT device is not available, and hence it may have problems in resuming the BT device. One way of tackling this situation is to change the PM_RESUME order in the kernel to be on a first-come-first-served basis.

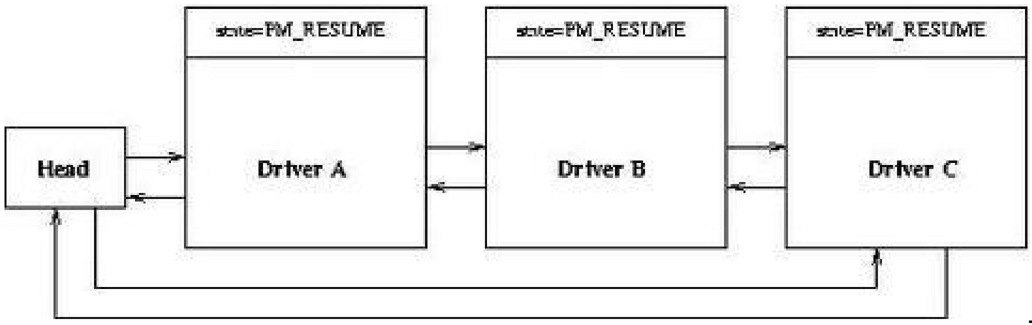
A driver stops participating in power management by calling pm_unregister:

```
pm_unregister(pm_callback cbackfn);
```

To unregister, it has to supply the pointer to the same function it used while registering. Once a driver has unregistered itself, the PM subsystem stops involving it in further PM events.

Linux also defines two interfaces, pm_access and pm_dev_idle, for drivers; pm_access should be called before accessing hardware, and pm_dev_idle has to be called when the device is not being used. These interfaces cannot be implemented on all platforms, though.

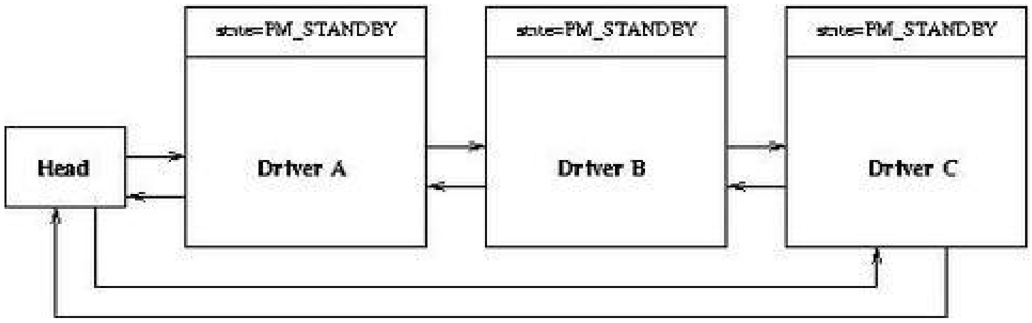
Now we illustrate how a typical state transition takes place when only device drivers are involved. The PM subsystem maintains all drivers that have registered with it in a doubly linked circular list. Figure 2 shows how this list looks when three drivers, A, B and C, have registered with it. This assumes that driver C registers first, then B and finally A.



(http://www.linu

Figure 2. System in Run State

Now, let’s say the system has to transition to standby state from run state. PM subsystem sends out a PM_STANDBY request to all three drivers, for which there are two possible outcomes. One, all drivers accept the request, and the system is put in standby state. Two, some driver rejects it. In this case, the standby transition is aborted, and the system continues to be in run state.

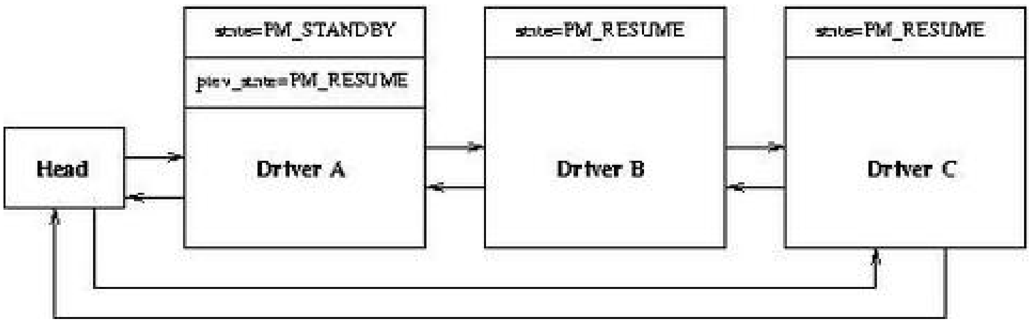


(http://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/066/6699/6699f3.jpg)

Figure 3. System in Standby State

Figure 3 shows what happens when all the drivers have accepted the PM_STANDBY request. Notice how the state field in pm_dev structure is changed when a driver accepts the request.

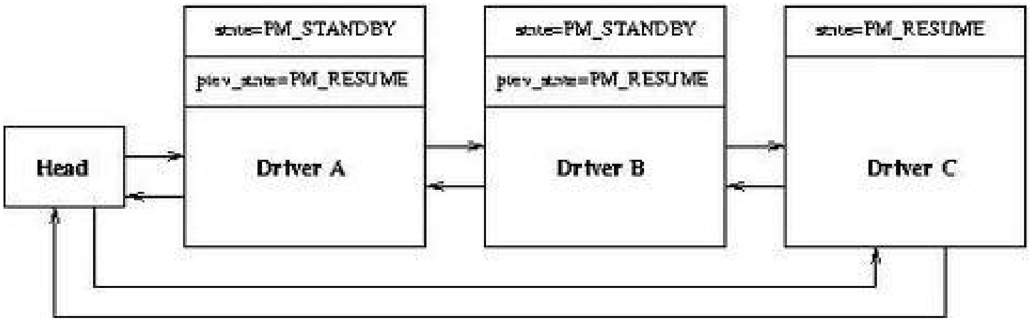
Let's now consider the case where drivers A and B accept the PM_STANDBY request, but driver C rejects it. Figure 4 shows the case after driver A has accepted the request. After driver A has agreed, the PM_STANDBY request is sent to driver B.



(http://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/066/6699/6699f4.jpg)

Figure 4. Driver A has accepted the PM_STANDBY request.

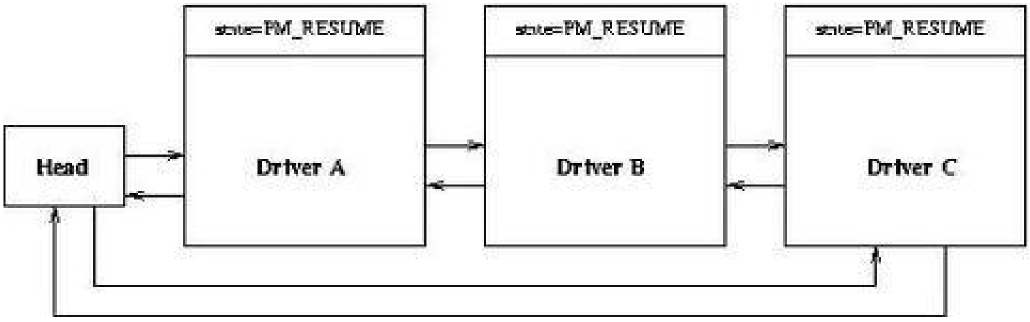
Figure 5 shows the state of the drivers after driver B also has accepted. Now both devices A and B are put in their standby state, while device C is still in its run state.



(http://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/066/6699/6699f5.jpg)

Figure 5. Driver A and driver C have accepted a PM_STANDBY request.

Next, PM_STANDBY is sent to driver C, which rejects it. In this case, the standby transition has to be aborted. Because devices A and B already have been put in their standby states, the PM subsystem has to perform an undo operation on them, so it sends a PM_RESUME request first to driver B and then to driver A. After this undo operation is done, all devices are put back in their run states, as shown in Figure 6.



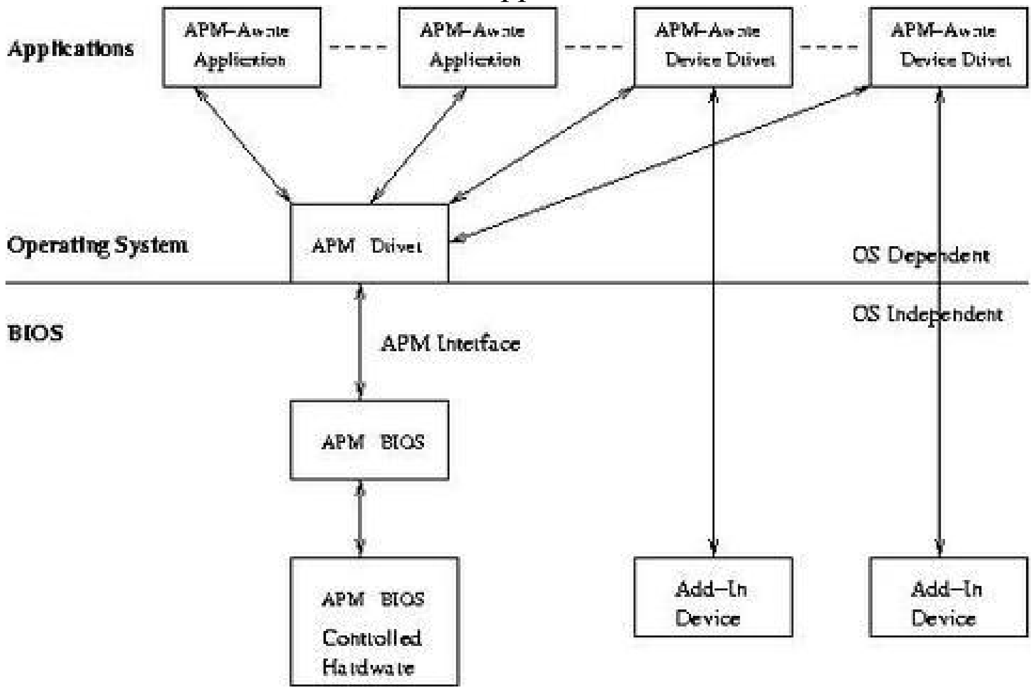
(http://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/066/6699/6699f6.jpg)

Figure 6. The system is back in run state after driver C rejected the PM_STANDBY request.

APM

Figure 7 shows the APM model. The important components of this model are:

- APM BIOS: software interface to the motherboard and its power managed devices and components. It is the lowest level of PM software in the system.
- APM driver: implements APM in a particular operating system.
- APM-aware device drivers and applications: APM driver interacts with them for all PM events.



(http://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/066/6699/6699f7.jpg)

Figure 7. APM Model

APM BIOS detects and reports various PM events, including low battery, power status change, system standby, system resume and so on. The APM driver uses polling function calls to the APM BIOS to gather information about PM events. It then processes these events in association with APM-aware drivers and applications.

The APM driver in Linux exposes two interfaces for an application’s use. The first, /proc/apm, holds information on the system power. It specifies whether the system is running on A/C power or battery. If running on battery, it also specifies the battery charge and time left for the battery to drain completely. The second interface, /dev/apm-bios, allows applications to know of and participate in PM events. It also allows them to initiate power state transitions by themselves, by issuing suitable ioctl calls. Read calls issued against this file will block until the next PM event occurs. When the read call returns, it carries information regarding the PM event about to occur.

Some of the applications that have opened /dev/apm_bios may be running with root privileges. Such applications are special to the APM driver. For some of the events, such as standby or suspend transition, APM driver informs all applications that have opened /dev/apm_bios about the event. In addition, it waits for approval from those few applications running with root privileges before the system actually is put in standby/suspend state. This approval comes when applications issue suitable ioctls.

The following ioctls normally are supported:

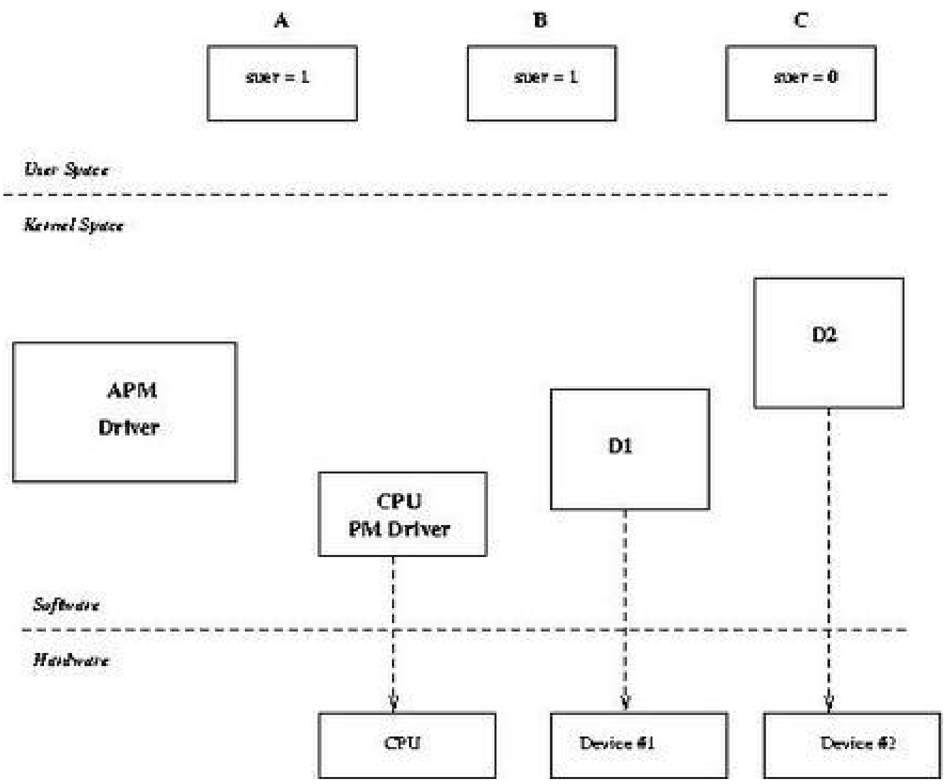
- APM_IOC_STANDBY: puts the system in standby state.
- APM_IOC_SUSPEND: puts the system in suspend state.

APM also comes with two user-space utilities. The apm command interacts with the APM subsystem in the kernel. Depending on the arguments passed, it can display system power status, or it can be used to initiate system standby/suspend transition. The apmd dæmon reports and processes various PM events and logs all PM events to /var/log/messages. In addition to logging, apmd also can take some specific actions for each type of PM event. These actions are specified in a script file (usually called apmd_proxy). This script file is invoked by the apmd dæmon with one or two arguments indicating the PM event about to occur. The following is a sample script file:

```
case 1:2 in

"standby":*)
     #System is going to Standby state because of
     #inactivity. Reduce CPU speed.
     echo 162200 > /proc/sys/cpu/0/speed
     ;;

"resume":"standby")
     #System is resuming to Run state from Standby
     #because of activity. Increase back the CPU
     #speed.
     echo 206400 > /proc/sys/cpu/0/speed
     ;;

"suspend":*)
     #System going to suspend state. Bring down
     #network interface.

     ifconfig eth0 down
     ;;

"resume":"suspend")
     #System resuming from suspend state.
     #bring up network interface and
     #increase the CPU speed and
     ifconfig eth0 up
     echo 206400 > /proc/sys/cpu/0/speed
     ;;
```

Example Power State Transition

Some of the complexities involved in power state transitions can be understood by taking the example of a state transition involving both drivers and applications. Assume the system has two drivers, D1 and D2, registered with PM and three applications, A, B and C, also participating in PM (by way of opening /dev/apm_bios). Of the three applications, A and B are running with superuser privileges, and C is not. Figure 8 depicts this scenario.



(http://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/066/6699/6699f8.jpg)

Figure 8. Example Power State Transition

Now, with this setup, let’s consider a case where the system wants to transition to sleep state from run state. The sequence of steps involved in this case begins with informing applications A, B and C about the pending transition to sleep state. This allows them to take whatever actions are necessary for this transition. Also, because A and B have superuser privileges, we have to wait for them to say okay to this sleep transition before it proceeds any further.

When A and B are done with whatever work they need to perform before the system transitions to a sleep state, they give the go-ahead to the APM driver. Now, the APM driver is ready to put the system into sleep state. It sends a PM_SUSPEND message to D1 and D2. D1 and D2 put their respective devices into sleep state and say okay to APM. After D1 and D2 are finished processing this transition, APM informs the CPU PM driver to put the CPU in sleep state. At this stage, the system transition to sleep state is complete.

Conclusion
Although APM has some drawbacks, its simplicity allows it to be implemented in almost any device. Other standards, such as ACPI, provide richer control over power management at the cost of complexity. It also is essential that all device drivers and applications implement power management support correctly. Without this proper support, a single driver may prevent the system from, say, going into suspend state. Once implemented properly, power management software greatly benefits the system in terms of enhanced battery life, leading to greater efficiency.

Courtesy: http://www.linuxjournal.com/article/6699?page=0,0 (http://www.linuxjournal.com/article/6699?page=0,0)

Read Full Post » (http://ucvbtechideas.wordpress.com/2012/10/24/power-management-in-linux-based-systems/)

Basic OS FAQ

Posted in Interview Questions on October 10, 2012 | Leave a Comment »

1. What is difference between the Process and the thread ?

Process:

- An executing instance of a program is called a process. Some operating systems use the term 'task' to refer to a program that is being executed.
- A process is always stored in the main memory also termed as the primary memory or random access memory.Therefore, a process is termed as an active entity. It disappears if the machine is rebooted.
- Several process may be associated with a same program.
- On a multiprocessor system, multiple processes can be executed in parallel.On a uni-processor system, though true parallelism is not achieved, a process scheduling algorithm is applied and the processor is scheduled to execute each process one at a time yielding an illusion of concurrency.
- **Example:** Executing multiple instances of the 'Calculator' program. Each of the instances are termed as a process.

Thread:

- A thread is a subset of the process.It is termed as a 'lightweight process', since it is similar to a real process but executes within the context of a process and shares the same resources allotted to the process by the kernel
- Usually, a process has only one thread of control – one set of machine instructions executing at a time.
- A process may also be made up of multiple threads of execution that execute instructions concurrently.
- Multiple threads of control can exploit the true parallelism possible on multiprocessor systems.
- On a uni-processor system, a thread scheduling algorithm is applied and the processor is scheduled to run each thread one at a time.
- All the threads running within a process share the same address space, file descriptor, stack and other process related attributes.
- Since the threads of a process share the same memory, synchronizing the access to the shared data withing the process gains unprecedented importance.

The major difference between threads and processes is:

1. Threads share the address space of the process that created it; processes have their own address space.
2. Threads have direct access to the data segment of its process; processes have their own copy of the data segment of the parent process.
3. Threads can directly communicate with other threads of its process; processes must use interprocess communication to communicate with sibling processes.
4. Threads have almost no overhead; processes have considerable overhead.
5. New threads are easily created; new processes require duplication of the parent process.
6. Threads can exercise considerable control over threads of the same process; processes can only

exercise control over child processes.
7. Changes to the main thread (cancellation, priority change, etc.) may affect the behavior of the other threads of the process; changes to the parent process does not affect child processes.

2. Why can't we use malloc in kernel code ?

You can't use libraries in the kernel. None whatsoever.

This means that ANY function you're calling in the kernel needs to be defined in the kernel. Linux does not define a malloc, hence you can't use it.There is a memory allocator and a family of memory allocation functions. Read the kernel docs on the memory allocator for more information.

Incidentally, there are a few functions the kernel defines which are in the standard C library as well; this is for convenience.

3. What is the major difference between kmalloc and vmalloc?

kmalloc allocates physically contiguous memory, memory which pages are laid consecutively in physical RAM. vmalloc allocates memory which is contiguous in kernel virtual memory space (that means pages allocated that way are not contiguous in RAM, but the kernel sees them as one block).

kmalloc is the preffered way, as long as you don't need very big areas. The trouble is, if you want to do DMA from/to some hardware device, you'll need to use kmalloc, and you'll probably need bigger chunk. The solution is to allocate memory as soon as possible, before memory gets fragmented.

Main reason for kmalloc being used more than vmalloc in kernel is performance. when big memory chunks are allocated using vmalloc, kernel has to map the physically non-contiguous chunks (pages) into a single contiguous virtual memory region. Since the memory is virtually contiguous and physically non-contiguous, several virtual-to-physical address mappings will have to be added to the page table. And in the worst case, there will be *(size of buffer/page size)*number of mappings added to the page table.

This also adds pressure on TLB (the cache entries storing recent virtual to physical address mappings) when accessing this buffer. This can lead to thrashing.

You only need to worry about using physically contiguous memory if the buffer will be accessed by a DMA device on a physically addressed bus (like PCI). The trouble is that many system calls have no way to know whether their buffer will eventually be passed to a DMA device: once you pass the buffer to another kernel subsystem, you really cannot know where it is going to go. Even if the kernel does not use the buffer for DMA *today,* a future development might do so.

vmalloc is often slower than kmalloc, because it may have to remap the buffer space into a virtually contiguous range. kmalloc never remaps, though if not called with GFP_ATOMIC kmalloc can block.

kmalloc is limited in the size of buffer it can provide: 128 KBytes. If you need a really big buffer, you have to use vmalloc or some other mechanism like reserving high memory at boot.

4. What is mmap?

In computing, mmap is a POSIX-compliant Unix system call that maps files or devices into memory. It is a method of memory-mapped file I/O. It naturally implements demand paging, because initially file contents are not entirely read from disk and do not use physical RAM at all. The actual reads from disk are performed in “lazy” manner, after a specific location is accessed. After the memory is not to be used, it is important to munmap the pointers to it.

5. In the linux kernel, what does the `probe()` method, that the driver provides, do? How different is it from the driver’s `init` function, i.e. why can’t the `probe()` functions actions be performed in the driver’s `init` function ?

Different device types can have probe() functions. For example, PCI and USB devices both have probe() functions.

Shorter answer, assuming PCI: The driver’s init function calls `pci_register_driver()` which gives the kernel a list of devices it is able to service, along with a pointer to the `probe()` function. The kernel then calls the driver’s `probe()` function once for each device.

This probe function starts the per-device initialization: initializing hardware, allocating resources, and registering the device with the kernel as a block or network device or whatever it is.That makes it easier for device drivers, because they never need to search for devices or worry about finding a device that was hot-plugged. The kernel handles that part and notifies the right driver when it has a device for you to handle.

6. What is the difference beteween kernel modules and kernel drivers

A kernel module is a bit of compiled code that can be inserted into the kernel at run-time, such as with insmod or modprobe.

A driver is a bit of code that runs in the kernel to talk to some hardware device. It “drives” the hardware. Most every bit of hardware in your computer has an associated driver[*]. A large part of a running kernel is driver code; the rest of the code provides generic services like memory management, IPC, scheduling, etc.

A driver may be built statically into the kernel file on disk. (The one in /boot, loaded into RAM at boot time by the boot loader early in the boot process.) A driver may also be built as a kernel module so that it can be dynamically loaded later. (And then maybe unloaded.)

Standard practice is to build drivers as kernel modules where possible, rather than link them statically to the kernel, since that gives more flexibility. There are good reasons not to, however:

- Sometimes a given driver is absolutely necessary to help the system boot up. That doesn't happen as often as you might imagine, due to the initrd feature.
- Statically built drivers may be exactly what you want in a system that is statically scoped, such as an embedded system. That is to say, if you know in advance exactly which drivers will always be needed and that this will never change, you have a good reason not to bother with dynamic kernel modules.

Not all kernel modules are drivers. For example, a relatively recent feature in the Linux kernel is that you can load a different process scheduler.

[*] One exception to this broad statement is the CPU chip, which has no "driver" *per se*. Your computer may also contain hardware for which you have no driver.

Courtesy: http://unix.stackexchange.com/questions/47208/what-is-the-difference-between-kernel-drivers-and-kernel-modules (http://unix.stackexchange.com/questions/47208/what-is-the-difference-between-kernel-drivers-and-kernel-modules)

7. What is Spinlock and what is difference between Mutex and Spinlock?

When you use regular locks (mutexes, critical sections etc), operating system puts your thread in the WAIT state and preempts it by scheduling other threads on the same core. This has a performance penalty if the wait time is really short, because your thread now has to wait for a preemption to receive CPU time again.

Spin locks don't cause preemption but wait in a loop ("spin") till the other core releases the lock. This prevents the thread from losing it's quantum and continue as soon as the lock gets released. The simple mechanism of spinlocks allow a kernel to utilize it in almost any state.

Courtesy: http://stackoverflow.com/questions/1957398/what-exactly-are-spin-locks (http://stackoverflow.com/questions/1957398/what-exactly-are-spin-locks)

Difference:

In theory, when a thread tries to lock a mutex and it does not succeed, because the mutex is already locked, it will go to sleep, immediately allowing another thread to run. It will continue to sleep until being woken up, which will be the case once the mutex is being unlocked by whatever thread was holding the lock before. When a tread tries to lock a spinlock and it does not succeed, it will continuously re-try locking it, until it finally succeeds; thus it will not allow another thread to take its place (however, the operating system will forcefully switch to another thread, once the CPU runtime quantum of the current thread has been exceeded, of course).

Read more@

Courtesy: http://stackoverflow.com/questions/5869825/when-should-one-use-a-spinlock-instead-of-mutex (http://stackoverflow.com/questions/5869825/when-should-one-use-a-spinlock-instead-of-mutex)

Read Full Post » (http://ucvbtechideas.wordpress.com/2012/10/10/basic-os-faq/)

Inside the Linux boot process

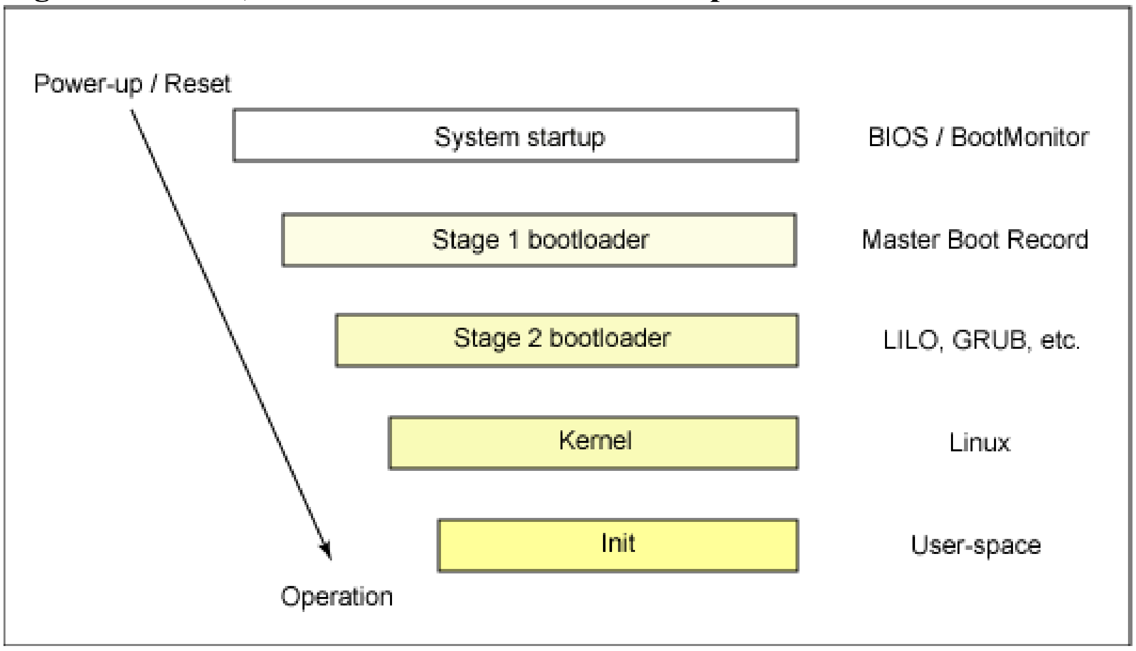
Posted in Bootloaders on October 8, 2012 | Leave a Comment »

Inside the Linux boot process

Take a guided tour from the Master Boot Record to the first user-space application

Summary: The process of booting a Linux® system consists of a number of stages. But whether you're booting a standard x86 desktop or a deeply embedded PowerPC® target, much of the flow is surprisingly similar. This article explores the Linux boot process from the initial bootstrap to the start of the first user-space application. Along the way, you'll learn about various other boot-related topics such as the boot loaders, kernel decompression, the initial RAM disk, and other elements of Linux boot.

In the early days, bootstrapping a computer meant feeding a paper tape containing a boot program or manually loading a boot program using the front panel address/data/control switches. Today's computers are equipped with facilities to simplify the boot process, but that doesn't necessarily make it simple.Let's start with a high-level view of Linux boot so you can see the entire landscape. Then we'll review what's going on at each of the individual steps. Source references along the way will help you navigate the kernel tree and dig in further.OverviewFigure 1 gives you the 20,000-foot view.
Figure 1. The 20,000-foot view of the Linux boot process



When a system is first booted, or is reset, the processor executes code at a well-known location. In a personal computer (PC), this location is in the basic input/output system (BIOS), which is stored in flash memory on the motherboard. The central processing unit (CPU) in an embedded system invokes the reset vector to start a program at a known address in flash/ROM. In either case, the result is the same. Because PCs offer so much flexibility, the BIOS must determine which devices are candidates for boot. We'll look at this in more detail later.

When a boot device is found, the first-stage boot loader is loaded into RAM and executed. This boot loader is less than 512 bytes in length (a single sector), and its job is to load the second-stage boot loader.

When the second-stage boot loader is in RAM and executing, a splash screen is commonly displayed, and Linux and an optional initial RAM disk (temporary root file system) are loaded into memory. When the images are loaded, the second-stage boot loader passes control to the kernel image and the kernel is decompressed and initialized. At this stage, the second-stage boot loader checks the system hardware, enumerates the attached hardware devices, mounts the root device, and then loads the necessary kernel modules. When complete, the first user-space program (`init`) starts, and high-level system initialization is performed.

That's Linux boot in a nutshell. Now let's dig in a little further and explore some of the details of the Linux boot process.

System startup

The system startup stage depends on the hardware that Linux is being booted on. On an embedded platform, a bootstrap environment is used when the system is powered on, or reset. Examples include U-Boot, RedBoot, and MicroMonitor from Lucent. Embedded platforms are commonly shipped with a boot monitor. These programs reside in special region of flash memory on the target hardware and provide the means to download a Linux kernel image into flash memory and subsequently execute it. In addition to having the ability to store and boot a Linux image, these boot monitors perform some level of system test and hardware initialization. In an embedded target, these boot monitors commonly cover both the first- and second-stage boot loaders.

Extracting the MBR

To see the contents of your MBR, use this command:

```
# dd if=/dev/hda of=mbr.bin bs=512 count=1
# od -xa mbr.bin
```

The `dd` command, which needs to be run from root, reads the first 512 bytes from /dev/hda (the first Integrated Drive Electronics, or IDE drive) and writes them to the `mbr.bin` file. The `od` command prints the binary file in hex and ASCII formats.

In a PC, booting Linux begins in the BIOS at address 0xFFFF0. The first step of the BIOS is the power-on self test (POST). The job of the POST is to perform a check of the hardware. The second step of the BIOS is local device enumeration and initialization.

Given the different uses of BIOS functions, the BIOS is made up of two parts: the POST code and runtime services. After the POST is complete, it is flushed from memory, but the BIOS runtime services remain and are available to the target operating system.

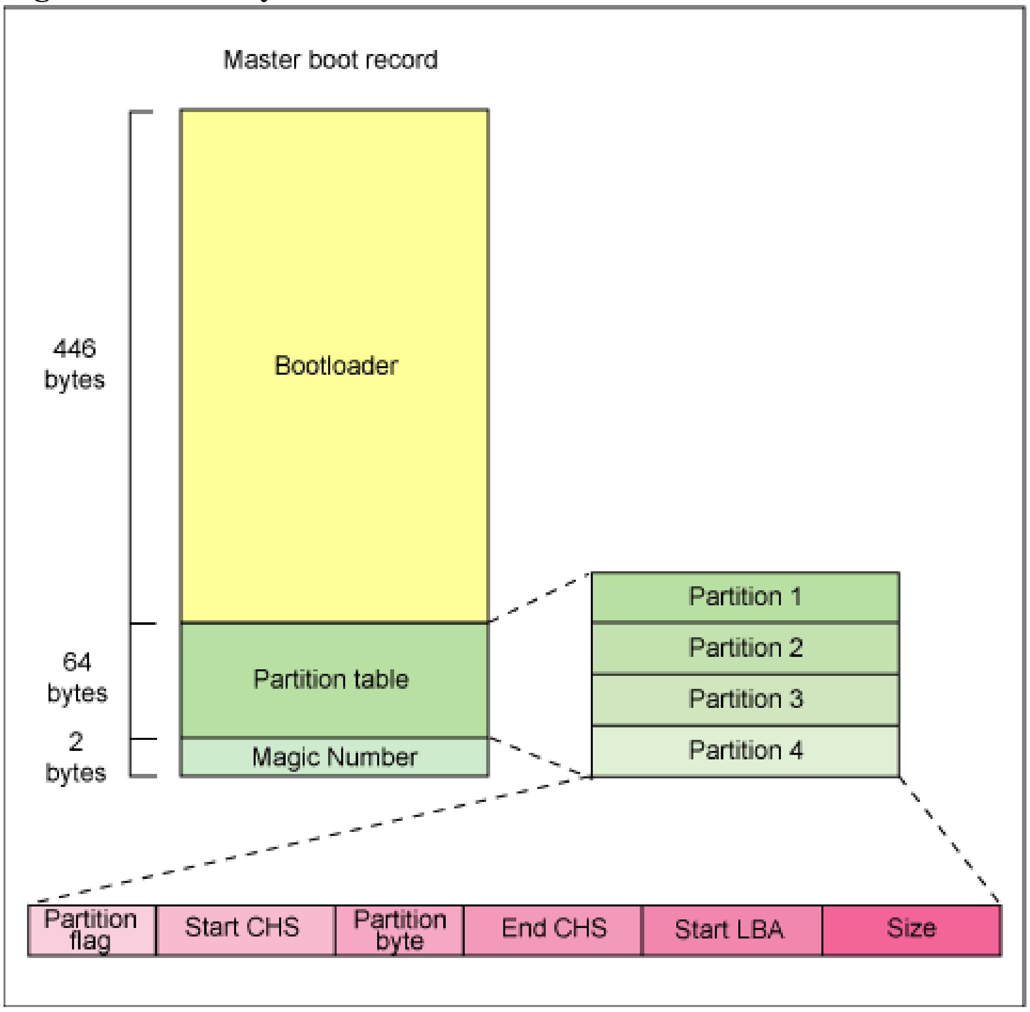
To boot an operating system, the BIOS runtime searches for devices that are both active and bootable in the order of preference defined by the complementary metal oxide semiconductor (CMOS) settings. A boot device can be a floppy disk, a CD-ROM, a partition on a hard disk, a device on the network, or even a USB flash memory stick.

Commonly, Linux is booted from a hard disk, where the Master Boot Record (MBR) contains the primary boot loader. The MBR is a 512-byte sector, located in the first sector on the disk (sector 1 of cylinder 0, head 0). After the MBR is loaded into RAM, the BIOS yields control to it.

Stage 1 boot loader

The primary boot loader that resides in the MBR is a 512-byte image containing both program code and a small partition table (see Figure 2). The first 446 bytes are the primary boot loader, which contains both executable code and error message text. The next sixty-four bytes are the partition table, which contains a record for each of four partitions (sixteen bytes each). The MBR ends with two bytes that are defined as the magic number (0xAA55). The magic number serves as a validation check of the MBR.

Figure 2. Anatomy of the MBR



The job of the primary boot loader is to find and load the secondary boot loader (stage 2). It does this by looking through the partition table for an active partition. When it finds an active partition, it scans the remaining partitions in the table to ensure that they're all inactive. When this is verified, the active partition's boot record is read from the device into RAM and executed.

Stage 2 boot loader

The secondary, or second-stage, boot loader could be more aptly called the kernel loader. The task at this stage is to load the Linux kernel and optional initial RAM disk.

GRUB stage boot loaders

The `/boot/grub` directory contains the `stage1`, `stage1.5`, and `stage2` boot loaders, as well as a number of alternate loaders (for example, CR-ROMs use the `iso9660_stage_1_5`).

The first- and second-stage boot loaders combined are called Linux Loader (LILO) or GRand Unified Bootloader (GRUB) in the x86 PC environment. Because LILO has some disadvantages that were corrected in GRUB, let's look into GRUB. (See many additional resources on GRUB, LILO, and related topics in the Resources (http://www.ibm.com/developerworks/linux/library/l-linuxboot/#resources) section later in this article.)

The great thing about GRUB is that it includes knowledge of Linux file systems. Instead of using raw sectors on the disk, as LILO does, GRUB can load a Linux kernel from an ext2 or ext3 file system. It does this by making the two-stage boot loader into a three-stage boot loader. Stage 1 (MBR) boots a stage 1.5 boot loader that understands the particular file system containing the Linux kernel image. Examples include `reiserfs_stage1_5` (to load from a Reiser journaling file system) or `e2fs_stage1_5` (to load from an ext2 or ext3 file system). When the stage 1.5 boot loader is loaded and running, the stage 2 boot loader can be loaded.

With stage 2 loaded, GRUB can, upon request, display a list of available kernels (defined in `/etc/grub.conf`, with soft links from `/etc/grub/menu.lst` and `/etc/grub.conf`). You can select a kernel and even amend it with additional kernel parameters. Optionally, you can use a command-line shell for greater manual control over the boot process.

With the second-stage boot loader in memory, the file system is consulted, and the default kernel image and `initrd` image are loaded into memory. With the images ready, the stage 2 boot loader invokes the kernel image.

Kernel

Manual boot in GRUB

From the GRUB command-line, you can boot a specific kernel with a named `initrd` image as follows:

```
grub> kernel /bzImage-2.6.14.2
[Linux-bzImage, setup=0x1400, size=0x29672e]
```

grub> **initrd /initrd-2.6.14.2.img**
[Linux-initrd @ 0x5f13000, 0xcc199 bytes]

grub> **boot**

Uncompressing Linux… Ok, booting the kernel.

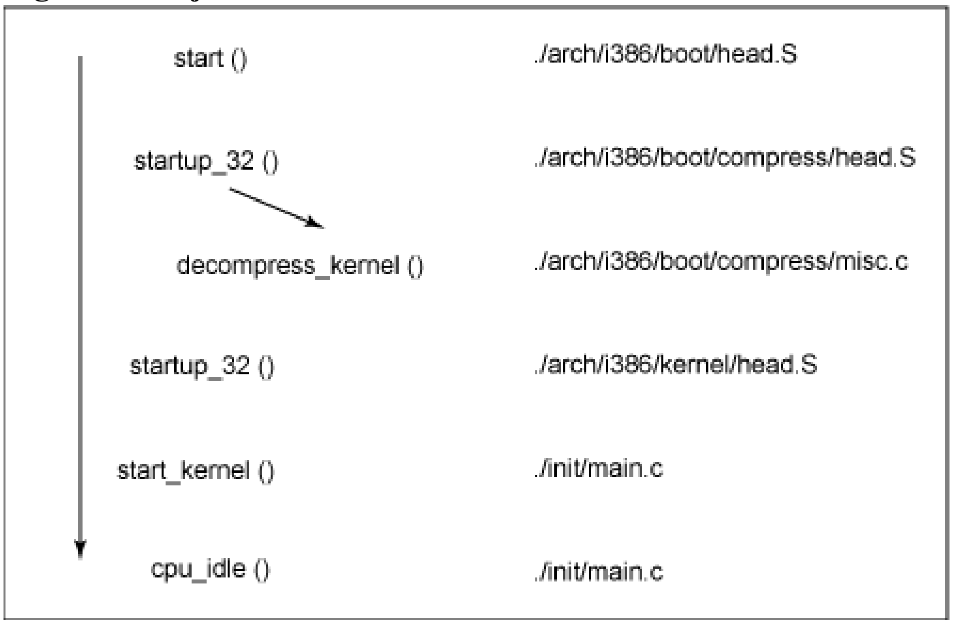
If you don't know the name of the kernel to boot, just type a forward slash (/) and press the Tab key. GRUB will display the list of kernels and `initrd` images.

With the kernel image in memory and control given from the stage 2 boot loader, the kernel stage begins. The kernel image isn't so much an executable kernel, but a compressed kernel image. Typically this is a zImage (compressed image, less than 512KB) or a bzImage (big compressed image, greater than 512KB), that has been previously compressed with zlib. At the head of this kernel image is a routine that does some minimal amount of hardware setup and then decompresses the kernel contained within the kernel image and places it into high memory. If an initial RAM disk image is present, this routine moves it into memory and notes it for later use. The routine then calls the kernel and the kernel boot begins.

When the bzImage (for an i386 image) is invoked, you begin at `./arch/i386/boot/head.S` in the `start` assembly routine (see Figure 3 for the major flow). This routine does some basic hardware setup and invokes the `startup_32` routine in `./arch/i386/boot/compressed/head.S`. This routine sets up a basic environment (stack, etc.) and clears the Block Started by Symbol (BSS). The kernel is then decompressed through a call to a C function called `decompress_kernel` (located in `./arch/i386/boot/compressed/misc.c`). When the kernel is decompressed into memory, it is called. This is yet another `startup_32` function, but this function is in `./arch/i386/kernel/head.S.`

In the new `startup_32` function (also called the swapper or process 0), the page tables are initialized and memory paging is enabled. The type of CPU is detected along with any optional floating-point unit (FPU) and stored away for later use. The `start_kernel` function is then invoked (`init/main.c`), which takes you to the non-architecture specific Linux kernel. This is, in essence, the `main` function for the Linux kernel.

Figure 3. Major functions flow for the Linux kernel i386 boot



With the call to `start_kernel`, a long list of initialization functions are called to set up interrupts, perform further memory configuration, and load the initial RAM disk. In the end, a call is made to `kernel_thread` (in `arch/i386/kernel/process.c`) to start the `init` function, which is the first user-space process. Finally, the idle task is started and the scheduler can now take control (after the call to `cpu_idle`). With interrupts enabled, the pre-emptive scheduler periodically takes control to provide multitasking.

During the boot of the kernel, the initial-RAM disk (`initrd`) that was loaded into memory by the stage 2 boot loader is copied into RAM and mounted. This `initrd` serves as a temporary root file system in RAM and allows the kernel to fully boot without having to mount any physical disks. Since the necessary modules needed to interface with peripherals can be part of the `initrd`, the kernel can be very small, but still support a large number of possible hardware configurations. After the kernel is booted, the root file system is pivoted (via `pivot_root`) where the `initrd` root file system is unmounted and the real root file system is mounted.

decompress_kernel output

The `decompress_kernel` function is where you see the usual decompression messages emitted to the display:

```
Uncompressing Linux... Ok, booting the kernel.
```

The `initrd` function allows you to create a small Linux kernel with drivers compiled as loadable modules. These loadable modules give the kernel the means to access disks and the file systems on those disks, as well as drivers for other hardware assets. Because the root file system is a *file system* on a disk, the `initrd` function provides a means of bootstrapping to gain access to the disk and mount the real root file system. In an embedded target without a hard disk, the `initrd` can be the final root file system, or the final root file system can be mounted via the Network File System (NFS).

Init

After the kernel is booted and initialized, the kernel starts the first user-space application. This is the first program invoked that is compiled with the standard C library. Prior to this point in the process, no standard C applications have been executed.

In a desktop Linux system, the first application started is commonly `/sbin/init`. But it need not be. Rarely do embedded systems require the extensive initialization provided by `init` (as configured through `/etc/inittab`). In many cases, you can invoke a simple shell script that starts the necessary embedded applications.

Summary

Much like Linux itself, the Linux boot process is highly flexible, supporting a huge number of processors and hardware platforms. In the beginning, the loadlin boot loader provided a simple way to boot Linux without any frills. The LILO boot loader expanded the boot capabilities, but lacked any file system awareness. The latest generation of boot loaders, such as GRUB, permits Linux to boot from a range of file systems (from Minix to Reiser).

Courtesy: http://www.ibm.com/developerworks/linux/library/l-linuxboot/ (http://www.ibm.com/developerworks/linux/library/l-linuxboot/)

Read Full Post » (http://ucvbtechideas.wordpress.com/2012/10/08/inside-the-linux-boot-process/)