Operating System Power Management (Windows, Linux, Android)

In recent years, the popularity of laptops and smart phones as portable computers has necessitated the advance of compact batteries and energy efficient components. These hardware changes have had a minimal effect on operating systems directly because power devices are controlled by drivers that interpret operating system instructions just as they do in personal computer and server systems. However, mobile computing has increased the interest in operating system functions and policies that decrease power consumption and have stimulated new developments in operating system functionality as a result. Software can make a more energy efficient system, and the following discussion provides some basic examples of how an operating system can control power consumption with the APIs and tools implemented to organize power management functions into working sets. Finally, I present the Android operating system's power management scheme as an example of how a total power management policy can work.

Implementing power saving measures using an API is similar across the windows family of operating systems. For the purposes of this document, we assume the Windows version is XP and the API libraries are implemented in 32-bit C# for .NET. Many of the system calls are portable to Windows 7 and 8, but I haven't tested them. Devices are controlled through messages sent through user32.dll or system calls initiated through kernel32.dll commands.

UNIX/Linux system libraries for power management vary with the user interface and the features enabled in the kernel. At the core of most system libraries is the Advanced Configuration and Power Interface (ACPI) libraries loaded into the kernel. These libraries are provided by hardware manufacturers who support the ACPI standard, which is referenced extensively in Energy Star standards.

Powering down the display and GPU

Although an emphasis on power efficiency has been advanced by mobile computing, power management isn't a new concept. Before the wide adoption of mobile computing, personal computers saved power by switching off the output to the monitor and/or switching the monitor into a power saving mode. Graphics Processing Units (GPUs) can require as much or more power than some Central Processing Units, and all monitors consume power when powered on whether a user is viewing them or not. The first power-saving mechanism implemented a command to put the monitor into a power saving mode when it was idle, i.e. when the computer was not receiving input from the computer or user. Because the monitor doesn't affect the state of running processes or write any data, powering off the monitor typically does not require special permissions.

A standard desktop or laptop monitor has three power saving modes configurable in the operating system or changed using an API, which in order of most to least power consumption are: standby (~90% power usage, backlight stays on), suspend (~15%, backlight is off), and off (~5%). Most mobile systems enable one energy efficient mode, called standby, which is equivalent to the PC suspend and leaves the display unlit until the user tries to interact with the device or an event, such as an incoming message, occurs.

For Linux, enabling the Advanced Configuration and Power Interface (ACPI) in the kernel grants access to the Energy Star Display Power Management Signaling (DPMS) parameters for the monitor, which define the monitor modes. You can control the monitor using the xset utility (available in most kernels), as shown in the following command:

```
$ xset dpms force off|suspend|standby|on
```

Where the final parameter is the desired monitor mode. The monitor returns to on mode if there is a user IO interrupt such as moving the mouse.

For Microsoft Windows, you can initiate a system call with the sendMessage() command from the user32.dll system library with the monitor state as the final parameter:

```
[DllImport("user32.dll")]
static extern IntPtr sendMessage(IntPtr hWnd,
                                 uint Msg,
                                 IntPtr wParam,
                                 IntPtr lParam);
```

Where:

- `hwnd` is a valid Handle for the window to which to send the message, which in this case does not need to be specific. You could create a new object to provide the handle, such as a form, or use a system helper method like GetDesktopWindow()
- `Msg` specifies the system message, which for monitor control is typically 0x0112
- `wParam` specifies additional message-specific controls, which for monitor power is 0xF170
- `lParam` specifies the desired monitor state; -1 = on, 2 = off, and 1 = standby

NOTE: Monitor power-saving modes are not the same as a screen saver, which doesn't save power, but was designed to extend the operating life-time of Cathode Ray Tube (CRT) monitors. If you want to save power, a screen saver does not save power, so use monitor power-saving modes instead.

Spinning down hard disks and packaging disk use

Another pre-mobile power saving concept packs or "chunks" disk access into sets of operations and spins down the hard drive in between chunks when not in use. Chunked disk access means that reads are largely done from memory and writes are stored in memory until a data size threshold is met, at which time the write to disk is done all at one time. This technology saves battery life on a laptop computer because disk access uses up more power than memory access, and can save fractions of power and cost when compounded by large numbers of computers like you might find in large server farms. The cost is in response time because spinning up the disk when access occurs takes more time than accessing information from the disk when it is already spinning.

Linux (kernel 2.6.6 or later) supports laptop_mode which controls disk access by organizing access into chunks. There are some subtleties to this, but essentially laptop_mode configures the update intervals written to disk to ten minutes and spins down the disks in-between. In the kernel source tree, the documentation for laptop mode is in Documentation/laptop-mode.txt. The laptop mode control script is in this document, which you can copy and paste into /sbin/laptop_mode and change the permissions on the script to execute in root with chmod (do not enable other users!) Then run `/sbin/laptop_mode start` as root to change your configuration to chunk writes.

To save power between writes, you need to configure the hard drives to spin down after a relatively short time. The hdparm command line utility controls and monitors ATA hard disks, and you can set the idle timeout of a hard-drive in intervals of 5 seconds. For example, to set the idle timeout to 15 seconds, use the following command:

```
$ hdparm -S 4 /dev/hda
```

NOTE: Mobile devices like smart phones use flash memory instead of disk drives, so do not benefit from spinning down the hard drive.

Monitoring Mobile and Laptop Battery Life

The power supply for a mobile computer is the battery, which by technological advancements has become a device that interacts with the operating system. In personal desktop computers and rack server systems, batteries are either passive devices used to maintain the ROM when the computer is powered off or an external Uninterruptable Power Supply (UPS) device that provides power in the event of a power failure so that a computer can continue to run for some period of time and shutdown. Neither of these compares well to the mobile computer battery, which is designed to provide power to the system for extended periods of operation and is charged and discharged repeatedly during regular use like a standard power supply. In mobile systems, the battery powers the entire system bus and does not typically require process or device synchronization to use the power output. This lack of restriction allows devices attached to the bus+battery combination, such as the display, user IO devices, static memory, CPU, volatile memory, and external devices, to turn on and off without interfering with other devices. Vulnerability is avoided by keeping the CPU and volatile memory powered on, and powering on other devices as needed.

However, one of the more effective ways to manage power is to inform the user and applications of the current battery charge. Applications that require a lot of power, such as a

GPS synched map and directions finder, can warn the user when the device is low on power and likely not enough charge to complete a complete navigation session.

For Linux, enabling the Advanced Configuration and Power Interface (ACPI) in the kernel grants access to battery status and enables a full battery maintenance check, which can be run on the command line:

```
$ acpi [-V]
```

The –V parameter activates the maintenance check, which has a detailed output. Without –V, the command simply returns battery status, similar to the following:

```
Battery 0: Charging, 17%, 01:20:06 until charged
```

For Microsoft Windows, you can initiate a system call with the getSystemPowerStatus() command from the kernel32.dll system library. In code it looks like the following:

```
[DllImport("kernel32.dll")]
public static extern bool GetSystemPowerStatus(
                      ref SYSTEM_POWER_STATUS systemPowerStatus);
```

Where:

- `systemPowerStatus` is a pointer to a SYSTEM_POWER_STATUS data structure, which GetSystemPowerStatus() populates with the current battery charge information. The structure has the following definition/fields:

```
typedef struct _SYSTEM_POWER_STATUS {
  BYTE  ACLineStatus; //1 is online
  BYTE  BatteryFlag;  //8 is charging
  BYTE  BatteryLifePercent;
  BYTE  SystemStatusFlag;
  DWORD BatteryLifeTime;
  DWORD BatteryFullLifeTime;
} SYSTEM_POWER_STATUS, *LPSYSTEM_POWER_STATUS;
```

Implementation Example: Power Management in the Android System

The Android operating system is based on Linux, so builds its power management API and policies on the ACPI standard discussed above. Android extends ACPI capabilities in a driver library for power-related commands in /drivers/android/power.c, only accessible to the kernel. Requests to the kernel from user programs are implement by the Android Power Management (PM) application framework, implemented in Java.

Android's power management policies dramatically increase a mobile device's battery life and function with the following rules:

1. All components (display, SIM card, processor, camera, etc.) default to the off state.

2. Applications and services must request the CPU resource with wake locks through the Android application framework or native Linux libraries to keep the CPU powered on.
3. At regular intervals, the system checks the power state and attempts to enter a lower power state. If there are no active wake locks, the CPU powers down.
4. A full wake lock keeps the display, keyboard and CPU powered-on. A partial wake lock only maintains CPU power.
5. User-level power control implemented in the Android power management (PM) application framework can request wake locks and implements time out mechanisms to switch the system power state.

Control over the power state of an Android device, including wake locks, are managed by the PowerManager class. To obtain an instance of PowerManager, call the Context.getSystemService() factory method, which retrieves a PowerManager object from the pool generated at startup in the POWER_SERVICE kernel context. Nearly all of the methods require appropriate permissions to succeed (consult the android.permission class reference for more information). The general controls of the device state are controlled with the following PowerManager methods:

- goToSleep(long time) puts the device into the sleep state in which all components are powered off for a specified amount of time. This method requires a wake lock and does not release any wake locks.
- wakeUp(long time) powers on all the device components for a specified amount of time. This method requires a wake lock.
- Reboot(String reason) restarts the device. This method releases all wake locks.
- newWakeLock(int levelAndFlags, string tag) creates a new wakelock, which turns the CPU and display on by acquiring a wakelock and off by releasing the wakelock, as shown in the following code:

```
PowerManager pm = (PowerManager)
    getSystemService(Context.POWER_SERVICE);
PowerManager.WakeLock wl =
    pm.newWakeLock(PowerManager.SCREEN_DIM_WAKE_LOCK, "My Tag");
wl.acquire();
   ... //screen stays powered on during this section
wl.release();
```

The first argument of the newWakeLock() method is a constant selected from the list of mutually exclusive wake lock levels, which are shown in the following table.

Flag Value	CPU	Screen	Keyboard
PARTIAL_WAKE_LOCK	On*	Off	Off
SCREEN_DIM_WAKE_LOCK	On	Dim	Off
SCREEN_BRIGHT_WAKE_LOCK	On	Bright	Off
FULL_WAKE_LOCK	On	Bright	Bright

NOTE*: A partial wake lock maintains power to the CPU regardless of any display timeouts, the display state, and even after the user presses the power button. For all other wake locks, the CPU runs, but the user can still put the device to sleep using the power button.

The Android operating system exposes the device's battery status with BatteryManager.java. BatteryManager responds to event messages generated by Linux kernel power supply class, sys/class/power_supply, that runs on top of the battery driver. The BatteryManager class is abstract, so does not need to be instantiated. In the Android context, events are gathered in an Intent, which is a passive data structure holding an abstract description of an action to be performed. The following implementation uses a system defined Intent, ACTION_BATTERY_CHANGED, to process BatteryManager information because it provides the correct parsing method, getIntExtra(), to parse BatteryManager status fields.

Because power is of less concern if the battery is charging or fully charged, a typical application uses the BatteryManager to determine whether the battery is or is not charging. To determine whether the battery is charging, move the battery status information from BatteryManager into an ACTION_BATTERY_CHANGED intent to determine the battery state, as shown in the following code:

```
IntentFilter ifilter = new
    IntentFilter(Intent.ACTION_BATTERY_CHANGED);
Intent batteryStatus = context.registerReceiver(null, ifilter);

// Are we charging / charged?
int status =
    batteryStatus.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
boolean isCharging = status == BatteryManager.BATTERY_STATUS_CHARGING
                  || status == BatteryManager.BATTERY_STATUS_FULL;
```

In this way, Android implements all the essential power management functions discussed previously for Linux and Windows operating systems, and power management is implementable by user-space applications as a result. The "always off" policy of Android sets it apart from other Linux operating systems, which directly addresses the power consumption constraints of the mobile devices Android runs.

Conclusion

Operating System power management is not a new concept, but has increased visibility and importance for mobile computing implementations. Mobile operating systems, like Android examined here, take advantage of existing power management functionality to implement policies that optimize power consumption. In addition, new system functions that access the battery as an interactive device implement system functionality that can extend battery life with strategic policies. The flexibility of these functions and policies determines the success of a power management policy, which can be controlled by user applications to optimize performance in application programming as well as system programming.