Supporting Multiple OSes with OS Switching

Jun Sun[1], Dong Zhou, Steve Longerbeam[2]
zhou@docomolabs-usa.com
DoCoMo USA Labs
3240 Hillview Ave., Palo Alto, CA 94304, USA

Abstract—*People increasingly put more than one OSes into their computers and devices like mobile phones. Multi-boot and virtualization are two common technologies for this purpose. In this paper we promote a new approach called OS switching. With OS switching, multiple OSes time-share the same computer cooperatively. A typical implementation can reuse an OS's suspend/resume functionality with little modification. The OS switching approach promises fast native execution speed with shorter switching time than traditional multi-boot approach. We describe the design of OS switching as well as our implementation with Linux and WinCE, and evaluate its performance.*

1. Introduction

Many people nowadays run multiple OSes on their computers. For example, developers may need to test their software on different OSes and/or on different versions of the same OS. Users sometimes find that two pieces of software that they like require different OSes.

In mobile device paradigm, we are also seeing more and more applications/systems using multiple OSes. For example, VirtualLogix (formerly Jaluna) [1] provides a solution for phones that integrates a real-time communication OS with a Linux OS. The recent announcement of OSTI [2] also indicated a trend of having multiple OSes on one mobile device.

Up until now there are two main approaches to supporting multiple OSes: multi-boot and virtualization [3]. With multi-boot systems, a user installs multiple OSes into disjoint disk partitions along with a multi-OS-aware boot loader. During the boot up time, the boot loader asks the user to select the OS to boot [4]. To run an OS different from the current OS, the user exits the current OS and reboot into the other OS through the boot loader. While switching to another OS takes a long time, the approach has the advantage of each OS running directly on hardware without any modification and running with full speed and full access to hardware resources.

Virtualization technology has been very popular recently (see [5][6][7][8] as examples). In virtualization, a user typically installs Virtual Machine (VM) monitor and related management software. With the help of VM software the user can further install several different OSes onto the same computer. The VM software can typically run multiple OSes concurrently. For example, with VMWare Workstation, each OS has its display shown as a window on the host OS. Switching from one running OS to another is almost equivalent to switching between GUI applications. However, virtualization technology typically suffers from degradation in performance [5]. More importantly, it typically requires a considerable amount of work to providing a virtualizing and monitoring layer, and sometimes to modify the existing OS and its device driver code.

In this paper we promote an alternative approach, called *OS switching*. In OS switching, multiple OSes time-share the same computer cooperatively. A straightforward implementation of OS switching can make use of the suspend/resume features that already exist in modern OSes. With OS switching, at any given time, only one OS is active. When the user wants to switch to a different OS, the currently active OS is suspended to memory, and the target OS is resumed from a previously suspended state and becomes the new active OS. The OS switching approach promises native execution speed and full hardware access by all OSes, and offers relatively fast switching speed. Implementing OS switching requires only minor modifications to existing OS code, plus relatively simple code for controlling switching between OSes.

2. OS Switching with Suspend/Resume

2.1. Overview

An OS switching system has multiples OSes installed on the persistent storage (e.g. disks or flash drives). To differentiate from the term "guest OS" used in virtualization technology, we call each of these OSes a *tenant OS*. More than one tenant OSes can be loaded into disjunctive memory regions and can boot up one by one. However, at any time there is only one tenant OS actively running, and this OS is called the current *active OS*. The active OS owns completely the whole

1. Work conducted when author was with DoCoMo USA Labs. He can be contacted via email: jsun@junsun.net.
2. Work conducted when author was with DoCoMo USA Labs. He can be contacted via email: stevel@sklembedded.com.

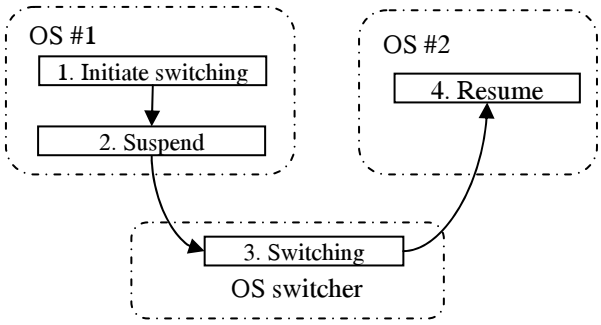

Figure 1. OS Switching Overview

system except for the portions of memory and disk reserved for other tenant OSes.

An active OS goes into dormant state through OS switching. OS switching is typically initiated by the end user (e.g., by pushing a switching button). It is also possible that OS switching is initiated through software triggered events.

After an OS switching is initiated, the current active OS (or outgoing OS in this context) performs necessary preparation, typically including saving necessary states for later resumption and putting hardware into a known and agreed-upon states for the next OS (or incoming OS).

Once the outgoing OS finished its preparation, an actual switch will happen. This step can be simply jumping to the resume path of the incoming OS. For OSes that support Memory Management Unit (MMU), however, this step may involve tearing down the outgoing OS' page mapping and setting up the new page table for the incoming OS.

The last step in OS switching is to restore incoming OS into an actively running state. This step involves retrieving states saved in system RAM and re-initialize hardware into a working state. Device drivers and application processes are re-activated, and system will continue to run from the state when the OS was last time suspended.

The four steps in OS switching is illustrated in Figure 1.

2.2. Suspend and resume in modern OSes

Most modern operating systems with advanced power management support a power-saving state where all hardware (including CPU and peripherals) are powered off except for the system RAM. In Windows this state is called *standby* state. Mac OS X calls it *sleep* state, while Linux refers to it as *suspend-to-RAM* (STR). When a computer performs suspend-to-RAM operation, the OS stops applications, drivers and kernel in order, and stores all necessary information in the RAM. The system then enters a low-power state while the RAM enters a low power self-refreshing state. Most of other hardware devices are turned off to save energy. When the system resumes, it retrieves operating state from memory and restores the whole system to the state when it was suspended.

Different OSes implement STR differently. The following text describes general steps involved in a typical suspend/resume process.

`Suspend Process:`

1. Suspend initiated
2. Applications are notified of the imminent suspend operation through callbacks. Certain applications may save data, or complete networking operations, etc.
3. Subsystems (such as file system, networking, daemons) are notified of the imminent suspend operation. For example, NFS domain may close its connection and save the connection information for later resumption.
4. The suspend routines in device drivers are called. Such suspend routines typically do two things: disabling its service to higher-level software and turn off the device (e.g., flushing and disabling DMA, disabling interrupts). Sometimes the suspend routine may also save certain information for later resumption.
5. Save system core state, including bus controller and CPU register states.
6. Turn off power to all hardware except system RAM, and enter sleep mode.

`Resume process:`

1. Resume is initiated through some pre-configured external events (pushing button, RTC timer expiration, etc). CPU control typically jump-starts from a pre-set address.
2. Restore CPU and system core states.
3. Invoke device drivers' resume() functions. The device driver resume functions typically enable the devices and make their services available for higher-level code.
4. Invoke the resume() functions of subsystems.
5. Notify applications of the resumption of operation. In Unix-like OSes, this can be achieved through signals.
6. System resumes full operation.

2.3. OS switching based on suspend/resume

If all tenant OSes support the suspend-to-RAM feature, we can re-use a large part of the suspend/resume code to implement OS switching. Conceptually, instead of turning off the power at the

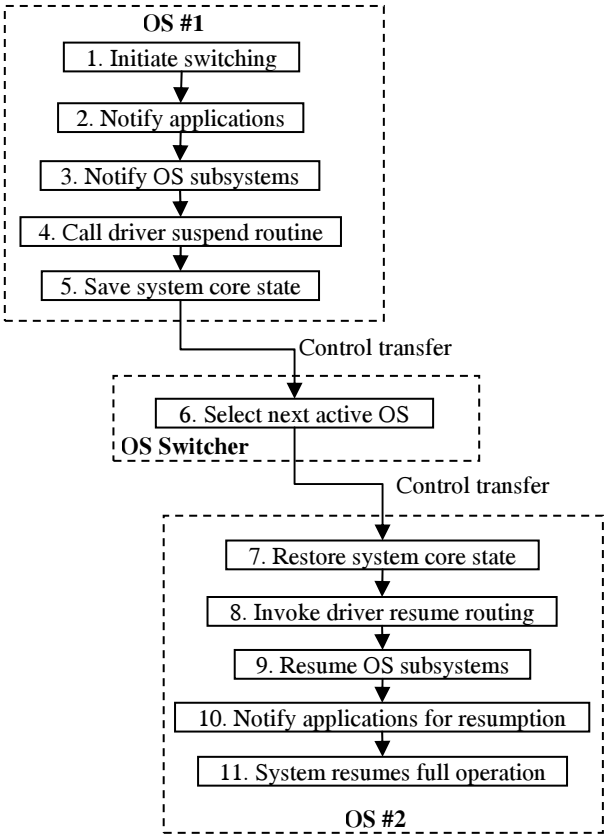

Figure 2. Flow of control in suspend/resume based implementation of OS switching

last step of the suspend process, we simply jump to the resumption path of the incoming OS.

Figure 2 shows the control flow of suspend/resume based OS switching.

If we are switching between two instances of the same OS with the same version, we are guaranteed that the hardware state at step 7 is exactly what the incoming OS expects. If we are switching between different OSes, however, the hardware states may be different from what the incoming OS expects. In that case, some form of "state adaptation" is needed (see section 2.5).

We implemented suspend/resume based OS switching in following environments:

- Linux-Linux switching on Sandgate 2P, an Intel PXA270 (ARM9) based handheld prototype device [10].
- Linux-WinCE switching on Sandgate 2P

Detailed implementation notes and performance data can be found in section 3. In the rest of this section, we will assume that there are two tenant OSes and discuss in general several other design issues.

2.4. Loading and booting of subsequent OSes

While the first OS in an OS switching system can be loaded and booted as usual (except that the loader and the OSes needs to be modified so that the OSes live in restricted memory regions), there are several design choices for loading and booting subsequent OSes:

1) The boot loader loads both OSes into RAM. One OS boots first and the second OS boots when it is selected to run for the first time.
2) The boot loader loads and boots the first OS. From the first OS an application loads the second OS. The second OS boots up when it is activated for the first time.
3) Boot loader loads and boots the first OS. From the first OS, an application installs the RAM content of the second OS previously captured when it went into suspended state.

Both methods 1) and 2) require special code that handles the booting of the second OS. Typically the loader and the OS mutually agree on the start-up system state. When for the first time we switch to the second tenant OS, the hardware state is usually different from what the loader would have set to. For example when Linux boots on ARM it expects MMU is turned off and first serial port is turned on. It also expects kernel command line arguments passed in through register r2. In order to boot the second OS correctly, we can either set the hardware states in the switcher so that they conform to the protocol, or we can modify the boot-up code in the second OS. We like the first approach as it is less intrusive in terms of changes to tenant OSes, and is more reusable when we switch among multiple different OSes.

Method 3) avoids the above problem. However, it requires a tool for capturing the memory content of the second OS in suspended state. In addition, the memory image, even when compressed, may be too large for systems where persistent storage space is scarce.

2.5. Switching

In theory, the actual control transfer is as simple as a jump instruction from the outgoing OS to the incoming OS.

In reality, this process is very complicated. The actual suspend process varies quite a bit for different CPUs, systems and OSes. In some OSes there are multiple suspend states. For example, Linux on VMPlayer has two suspend states corresponding to ACPI's S1 and S4 states [11].

Therefore, the first implementation decision is to choose a suspend path as the default OS switching path. Different suspend path puts hardware into

different suspend states and have different resume points. For example, Linux on VMPlayer support "standby" state, a shallower power saving mode where CPU context, including MMU and program counter, are preserved during suspend. Resume starts from the last instruction that puts the CUP into suspend state. By comparison, Linux on Sandgate 2P supports "memory" state, a deeper power saving mode where all CPU context are lost. The resumption point is remembered in a non-volatile register and CPU resumes in physical addressing mode.

As a result the switcher would need to manage all these differences and ensure the control transfer happen smoothly among tenant OSes. The OS switcher typically performs:

1. Saving any contexts that were assumed to persist during normal suspend but will be lost during OS switching.
2. Tearing down current MMU mapping and switching to physical addressing mode. If incoming OS resumes from virtual addressing mode, setup MMU mapping for the incoming OS.
3. Restoring any context for the incoming OS that were assumed to persist during normal suspend but was lost during OS switching.

3. Implementation and Performance

In this section, we describe our implementation of OS switching, and evaluate and analyze OS switching performance. As mentioned earlier, we have implemented two prototypes. In this section we focus on the Linux-WinCE Sandgate 2P prototype.

3.1. Loading and booting

In our Linux-WinCE Sandgate 2P prototype, we chose the first approach for subsequent tenant OS loading and booting, i.e., we modified the WinCE loader, eboot, to load both OSes into RAM. Eboot sets up the environment for WinCE to boot up first. When for the first time we switch from WinCE to Linux, the OS switcher sets up proper hardware state so that Linux can boot up successfully. Special setting include turning off MMU, turning on serial port, and preparing kernel boot arguments.

3.2. Switching on the Linux side

Our Linux kernel base is 2.6.16. Intel has supplied board specific support for Sandgate 2P. In this implementation, Linux supports two power saving states, "standby" and "memory". We decided to modify the suspend-to-memory execution path for OS switching purpose.

A side button on the prototype device is designated as the OS switching button. The keypad driver sends a signal to the APM daemon when a button pressing event is detected. Upon receiving this signal, the APM daemon performs the suspend process, including calling each driver's suspend() function. At the end of this process, instead of going into suspended state, the daemon jumps into the switcher and calls the switching function. In section 3.4 we will discuss in detail what the switcher does.

3.3. Switching on WinCE side

WinCE supports several power saving states including idle and suspend. Unfortunately the WinCE BSP we obtained from vendor does not fully support them. While some drivers have their own suspend/resume routines, some do not. In addition, there is no system-wide suspend/resume routines.

Our implementation effort starts with supplying those suspend/resume functions for various drivers including display driver. Similar to the Linux case, when the keypad driver detects an OS-switching button pressing event, it changes the system power state into suspend state, which starts the standard suspend process. The standard suspend process invokes the OEMPowerOff() function after all devices are suspended. The OEMPowerOff() function in turn invokes our real OS switching function.

3.4. Implementation of OS switcher

For practicality reasons, OS switcher is implemented inside eboot. Thus it also uses eboot's address mapping, which is different from either Linux's or WinCE's.

When we switch from Linux to WinCE, the switcher will save the CPU context, including MMU, general registers, system control registers, etc. It will then restore WinCE's CPU context. Since WinCE suspend/resume is not complete, the OS switcher performs additional saving and restoring for peripheral devices such as LCD, audio, etc.

When we switch from WinCE to Linux, the OS switcher performs similar steps. Again, for WinCE, the OS switcher saves additional context for peripheral devices. This saving is necessary as states for peripheral devices presumably will be altered once Linux becomes active.

3.5. Evaluation

In this section we present and discuss timing data for OS switching. The numbers presented in this section are obtained through instrumentation of Linux (2.6.16 kernel) and WinCE 5.0 source code. Since we

Table 1. Breaking down OS switching time (Linux)

Suspend Steps	Time Used (us)	Resume Steps	Time Used (us)
Freeze processes	8500	Thaw processes	3226
Device suspend	5845	Device resume	1384313
Other	32	Other	4796
Total	14377	Total	1392335

didn't have access to the full source code of WinCE 5.0, we could only measure suspend and resume time costs of major device drivers for WinCE side.

Table 1 breaks down the costs for switching out of and into Linux. Since process freeze/thaw and device suspend/resume times dominate the total cost, we omitted listing the costs of other individual steps. Note that the costs for freezing and thawing processes depend on those specific processes. In our experiment we only had a few basic processes running. As we can see from the table, the resume cost in Linux is much higher than the suspend cost, and this resume cost is dominated by the cost for resuming devices. Overall, the resuming process takes close to 1.4 seconds, and the total time for switching from a Linux OS to another Linux OS is slightly over 1.4 seconds (Linux suspend time plus Linux resume time).

Table 2. further breaks down the suspend/resume costs of individual Linux device drivers. Note that the resuming costs of the PCMCIA driver, the frame buffer driver, and the WiFi driver, dominates the total cost of device resumption.

On WinCE side, excluding GWES' (Graphics, Windowing and Events Subsystem) asynchronous handling of power on/off events, almost all the costs of OS switching is in device suspend and resume. Table 3 shows the suspend/resume costs of four drivers that were used on the WinCE side of the prototype: the drivers for display, touch screen, keypad, and audio. The total suspend time for the drivers is about 188ms, and the total resume time is about 341ms. We can thus infer that the time for Linux-to-WinCE switch is around 0.3-0.4 second, while the time for a WinCE-to-Linux switch is around 1.6 seconds.

Table 2. Suspend and resume cost of Linux device drivers

Driver	Suspend Time (us)	Resume Time (us)	Comment
ak4650-ts	2	3036	Touchscreen
ak4650-core	325	165	core for TS and audio
hostap_cs	1378	254231	Wifi driver
pxa2xx-pcmcia	26	769261	PCMCIA driver
pxa2xx-fb	3770	344926	Frame buffer
pxa2xx-ac97	328	12683	AC97 controller
Other drivers	16	11	
total	5845	1384313	

Table 3. Suspend/resume cost of WinCE device drivers

Device	Suspend Time (us)	Resume Time (us)
Display	153829	276103
Touchscreen	3439	4380
Keypad	3414	57316
Audio	27500	2862
Total	188182	340661

We also measured *user perceived switching time*, defined as the time from display going into blank in the outgoing OS, to the time display resumes in the incoming OS. It is measured as the time between the end of display driver suspend on one side, to the end of the display driver resume on the other side. In our setup, the user perceived switching time from WinCE to Linux is about 398.8ms, while the user perceived switching time from Linux to WinCE is about 328.3ms.

4. Related Work

The idea of using suspend/resume (and also shutdown/reboot) to support multiple OSes first appeared in a patent [9] by Shimotono. The patent generally assumes PC-like computing systems where BIOS performs the major part of switching. The outgoing OS completely shuts off the whole system and the incoming OS will start from reset state with a flag to indicate it is the resumption instead of a regular booting. From OS perspective there is no difference between a real suspend/resume and an OS switching.

Clearly this scheme does not work for non-PC systems where there is no BIOS standard and power management standard (such as APM or ACPI). In this paper we extend and broaden this idea to a more general OS switching approach where tenant OSes work cooperatively to time-share the same computing device. Shimotono's patent is a special case where all tenant OSes must suspend and shut off all hardware (even including CPU) completely before a switching can happen on the rebooting path in BIOS. In this paper we demonstrate that multiple OSes can suspend differently into different states and adapt through the OS switcher with a flexible scheme. Another directly related approach is the multi-boot approach [4]. Like OS switching approach, hard disks or permanent storage are partitioned among tenant OSes, and there is only one active OS at any time. Unlike OS switching approach, the active OS owns the whole system RAM instead sharing with other tenant OSes. However, OS switching offers much faster switching time.

Compared with virtualization technologies, OS switching lacks concurrency and hence is not suitable for application scenarios where multiple OSes need to run concurrently (for example, telnet from one tenant OS to another). In addition, OS switching depends on corporation among OSes and is consequently less robust against faulty OS implementations. On the other hand, OS switching offers native execution speed, which gives better performance than virtualization (especially traditional full virtualization). In addition, many application scenarios (such as multi-OS driver development) require native hardware access which is not possible in virtualization.

Compared with para-virtualization approaches such as Xen [5], OS switching requires less kernel modification. For example, our kernel patch for Linux/WinCE switching on ARM changes, excluding device driver changes, 26 lines of WinCE code and 139 lines of Linux, plus around 60 assembly instructions. In addition, OS switching only needs to change so-called BSP part of kernel, not as intrusive as other para-virtualization approaches. Because of this attribute, we are able to enable Linux-WinCE switching even though we don't have the full source of WinCE kernel.

The implementation of OS switching closely resembles cooperative VM approach in that all kernels have privileged access to the whole system and the cooperative relation among OS kernels. Cooperative Linux [12] modifies Linux kernel to run inside the host OS's kernel. The guest Linux kernel runs as a process on top of the host OS. MMU is time-shared between the host kernel and guest Linux kernel. Peripheral hardware access is virtualized through host OS's support. Jaluna's OSware [1] integrates two or more OSes and multiplexes hardware interrupts and CPU usage among them. Hardware resources are exclusively partitioned among OSes. Virtualized hardware access is possible if the owner OS exports the resource and the client OS has the virtual driver which knows how to talk to the owner OS. Compared with cooperative VM approach, OS switching approach trades multi-OS concurrency for implementation simplicity and full native access to hardware.

5. Summary and Conclusion

OS switching enables multiple OSes time-share the same computer in a cooperative manner. Its implementation typically reuses suspend/resume functionalities already existing in modern OSes and result in little modification to existing kernels. Compared with multi-boot approach, OS switching offers much faster switching time. Compared with virtualization approach OS switching offers simplicity, native execution speed and native hardware access.

In this paper we generalize the OS switching notion and present our study on its design, implementation, and performance. Despite some of its limitations we believe OS switching is a useful alternative to multi-boot approach and virtualization approach for many application scenarios where simplicity, performance, native hardware access and switching time are important. We would like to promote this approach and are hopeful to see wider applications of OS switching technology.

Acknowledgements

We would like to thank many of our colleagues for providing insights to our OS switching work, including Ken Ohta, Takehiro Nakayama, Jane Inamura, and Atsushi Takeshita. We would also like to thank Hiroshi Inamura for initiating the idea of putting multiple OSes on a mobile phone for improved system dependability.

References

[1] Jaluna. *Jaluna OSware*. Web site: http://www.jaluna.com.

[2] Intel and NTT DoCoMo, "Open and Secure Terminal Initiative (OSTI)," http://www.nttdocomo.co.jp/english/corporate/technology/osti/

[3] R. J. Creasy, "The origin of the VM/370 time-sharing system," *IBM Journal of Research and Development*, vol. 25, pp. 483-490, 1981.

[4] GNU. GNU Grub Project. Website: http://www.gnu.org/software/grub/.

[5] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, "Xen and the Art of Virtualization," in *Proceedings of the ACM Symposium on Operating Systems Principles*, October, 2003.

[6] Intel. *Intel Virtualization Techonology*. Web site: http://www.intel.com/technology/computing/vptech/.

[7] A. Whitaker, M. Shaw, and S. Bribble, "Denali: Lightweight virtual machines for distributed networked applications," in *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[8] VMWare. *VMWare home page*. Web site: http://www.vmware.com

[9] S. Shimotono, "Computer system, operating system switching system, operating system mounting method, operating system switching method, storage medium, and program transmission apparatus", US patent application number US20010018717A1, Aug 30, 2001.

[10] Sophia Systems. *Sandgate 2P reference design.* Web site: http://www.sophia.com/Products/SG2P.html.

[11] Advanced Configuration & Power Interface. "ACPI Specification". http://www.acpi.info/spec.htm.

[12] D. Aloni, "Cooperative Linux", in *Proceedings of the Linux Symposium (vol 2)*, pp.23-31, Ottawa, Ontario, July 21st-24th, 2004.