CSIT (March 2013) 1(1):67–74
DOI 10.1007/s40012-012-0005-x

ORIGINAL RESEARCH

Efficient operating system switching using mode bit and hibernation mechanism

Abhijeet Kumar · Saurabh Srivastava · R. H. Goudar

Received: 11 May 2012 / Accepted: 11 October 2012 / Published online: 30 October 2012

Abstract With the recent developments, Technology is making one independent and providing various options and varieties. As multiple options are provided, there arises a need of working in different computing environments. Though Virtualization is available but the efficient way for the process execution is not achieved as there is a limit to number of OS. In this paper, we are designing a mechanism to improve the existing multi-OS system for switching to different computing environments. To accomplish this task, we are using a register (R/W) that uses mode bit (binary values) for mapping (in boot configuration file) of different OS environments. The user is providing a choice for OS selection in GUI of the running environment via some specific software which in turn sets the mode bit value. Then it hibernates the current system and context data that is necessary to resume is saved preferably to non-volatile storage. Further, it restarts (RESET mechanism) and corresponding to mode bit value; other environment is loaded from its previous state. Data integrity is maintained between computing environments such that data from current mode cannot contaminate data in the next computing mode. Overall, these improvements will reduce time required for swapping of OS in multi-environment systems and also retains data by saving the current state. In addition, this mechanism frees user from waiting for graphical interaction for OS selection.

A. Kumar (✉) · S. Srivastava · R. H. Goudar
Graphic Era University, Computer Science and Engineering, Dehradun, India
e-mail: abhijeetchar@gmail.com

S. Srivastava
e-mail: saurabhsrivastavacs@gmail.com

R. H. Goudar
e-mail: rhgoudar@gmail.com

Keywords Virtualization · Booting · Hibernation · ACPI Specifications · Operating System Switching · Mode Bit Register · Stage-2 Booting

1 Introduction

Every business organization is taking concrete measures to provide its customers with the quality of services and tries to gain maximum customer satisfaction. The need of fast switching among operating systems is indispensible in context to present IT sector. Basically, we are following the usual concept of *Booting* and *Hibernation* technique. In fact on the ground of these two concepts, we are introducing our proposed system.

Structurally speaking, there are several steps that lead to OS Log-On services which are provided to user collectively called as Booting process. Booting phase [1] can be described as:

Figure 1 shows several steps of booting process for running multiple operating systems. Once the mother board is powered up, it initializes its own firmware-the chipset and other tidbits. The only directly executable code is a *tiny boot stub* in chipset. If all is well, one CPU from multi-core system is dynamically chosen to be the bootstrap processor (BSP) and registers in CPU have well defined values including instruction pointer (EIP) which holds the address of instruction being executed by BSP. This standard address is called Reset Vector for modern Intel CPU's. The instruction at the reset vector is a jump to memory location mapped to BIOS entry point. This memory mapping is kept in chipset (Fig. 1).

Now, CPU starts executing BIOS code which initializes some of the hardware in the machine. Afterwards, it starts power on self test (POST) which tests various components

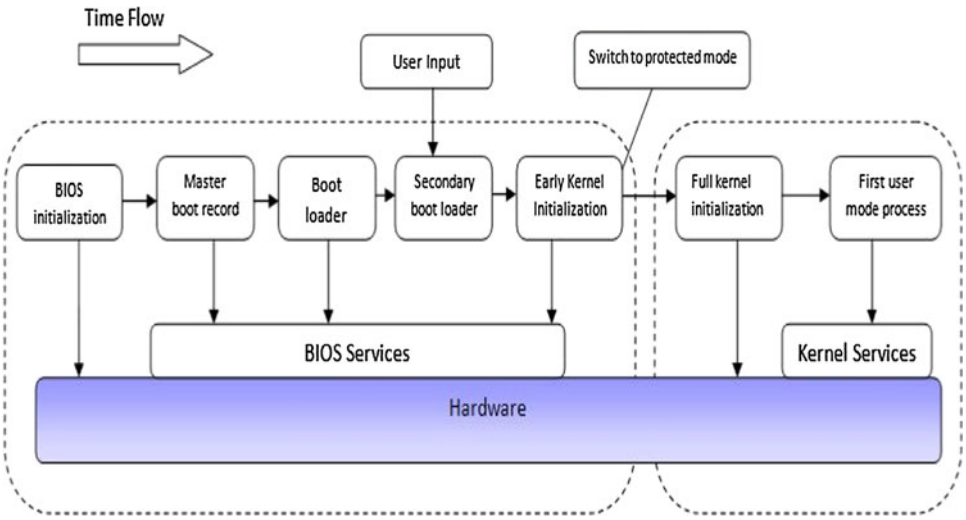

Fig. 1 Steps in booting process (Supporting multi-boot system)

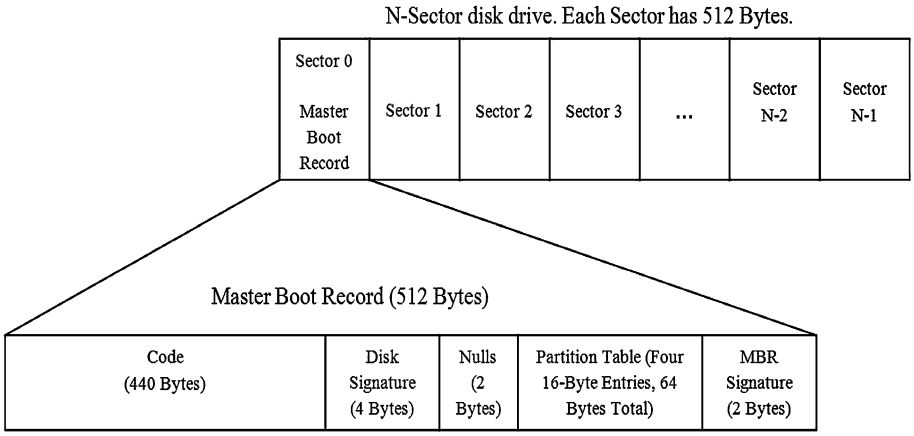

Fig. 2 Master boot record (MBR)

and peripherals attached with the system. The actual order in which BIOS seeks a boot device is user configurable (boot device priority). Once it selects the boot device (generally HDD), it now reads first 512 Bytes (Sector 0) of HDD. This sector is called Master Boot Record (MBR). The MBR sector of HDD is represented diagrammatically in Fig. 2.

It contains two vital components: A small OS-specific program (*Primary Bootstrap loader*) at the start of MBR followed by partition table of the disk. Partition table is standardized: it is a 64 Byte area with four 16 Byte entries describing the division of disk (for running multiple OS Systems). However, without caring this, BIOS simply loads the content of MBR in main memory and jumps to that location (in RAM) to start executing whatever code is in MBR. This specific code is a primary boot loader (Figs. 3, 4, 5).

Since our approach only deals with multi-OS systems (i.e. different environments) so, the primary boot loader will always load the additional bootstrap code (process known as *"Stage-2 Booting"*) in case of multi-booting. Linux boot loaders like LILO and GRUB has gotten more flexible and can handle variety of operating systems, file systems and boot configurations. The functionality of *multi-booting* goes like this:

- Primary boot loader loads additional bootstrap code as mentioned earlier (secondary bootstrap loader).
- The MBR code plus the code loaded in first step then reads a file containing second stage of boot loader. The stage-2 code then reads a boot configuration file. It then presents boot choices to the user.
- At this point, boot loader needs to load the kernel of operating system as choice selected by the user. For this, it uses the information from boot sector (first sector of partition) of partition table and reads the file containing the kernel and then loads the file into main memory and jumps to the kernel bootstrap code. This is the time where kernel starts to unfold and sets services for the user [1], [9].

Again, Hibernation is a technology that is used in the system which provides user a choice to save system's state

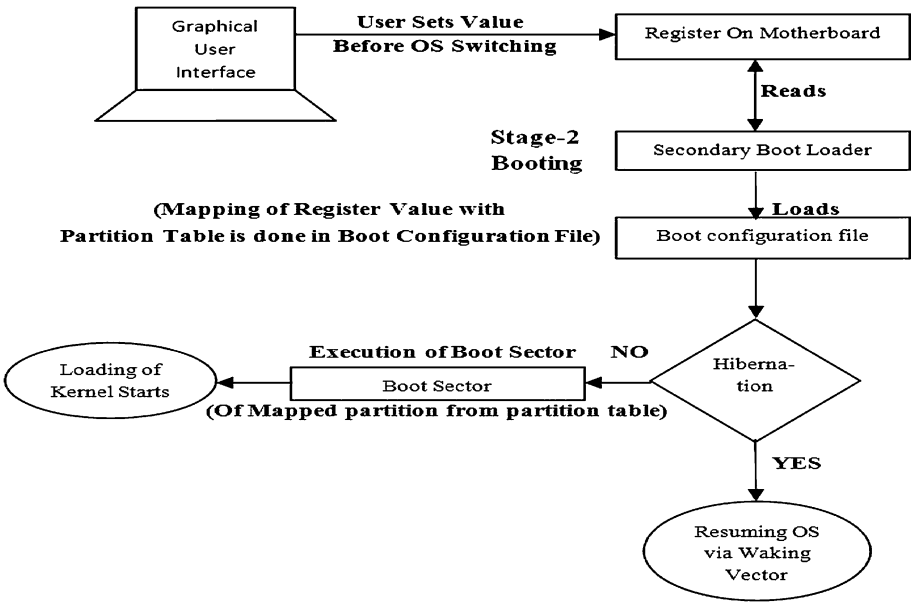

Fig. 3 OS Switching from GUI by using mode-bit register (Proposed approach)

and then restore all running programs after powering back on without any electrical power loss. Though nowadays ATX [2] motherboards and ATX power supply requires some power (standby power i.e. "5VSB") which is always on and is used to allow components of the computer (BIOS and networks adapter) to keep running some very simple software even when system is turned off.

According to the ACPI specifications [3], the standby modes have five states and hibernation state (*S4 sleeping state*) is one of them. The specification quotes this as follows [3], [4].

System S4 sleeping state is logically lower than the S3 state (*Sleep mode*) and is assumed to conserve more power. The behavior of this state is defined as follows:

- The processors are not executing instructions. The processor-complex context is not maintained.
- DRAM context is not maintained.
- Power Resources are in a state compatible with the system S4 state. All Power Resources that supply a System-Level reference of S0, S1, S2, or S3 (other states of G1 sleeping state) are in the OFF state.
- Devices that are enabled to wake the system and that can do so from their device state in S4 can initiate a hardware event that transitions the system state to S0 (Working state). This transition causes the processor to begin execution at its boot location.

There are two ways that OSPM (Operating System-Directed Configuration and Power Management) may handle the next phase of the S4 state transition; saving and restoring main memory. The first way is to use the operating system's drivers to access the disks and file system structures to save a copy of memory to disk and then initiate the hardware S4 sequence by setting the *SLP_EN register* bit. When the system wakes, the firmware performs a normal boot process and transfers control to the OS via the *firmware_waking_vector loader*. The OS then restores the system's memory and resumes execution. The alternate method for entering the S4 state is to utilize the BIOS via the S4BIOS transition. The BIOS uses firmware to save a copy of memory to disk and then initiates the hardware S4 sequence. When the system wakes, the firmware restores memory from disk and wakes OSPM by transferring control to the FACS waking vector. The alternate S4BIOS transition provides a way to achieve S4 support on operating systems that do not have support for the direct method.

2 Overview of the proposed research

2.1 Proposed idea

We are proposing a mechanism for multi-OS systems which is using Mode Bit (R/W Register) that is attached to the motherboard and hibernation technology for OS switching. Functionally, we are designing a system where the user while working in current OS sets value in the mode bit register via some GUI specific software system for switching to different OS. As soon as this value is set in register, the system goes to hibernation state (all the running instances of the current OS are saved to hibernation file in non-volatile storage area or it can be saved in volatile storage area where power to the computer is not interrupted during computing mode switching) and then as a final step, instead of sending the machine to power state

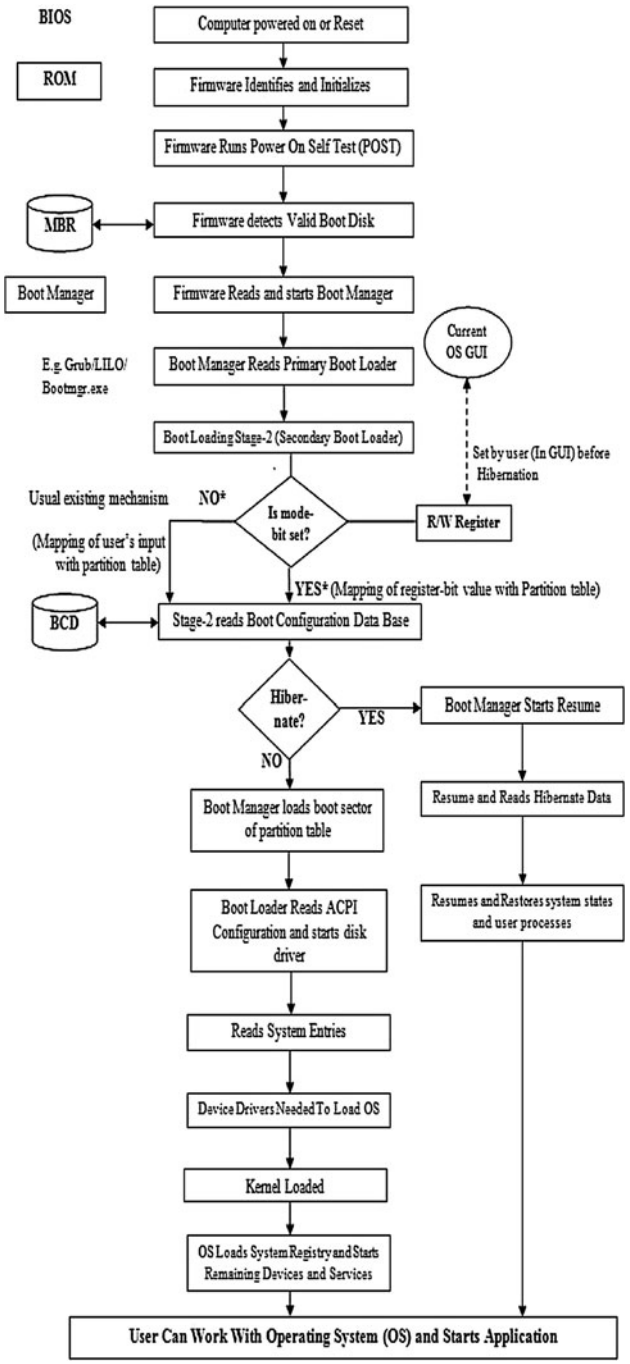

Fig. 4 Flow-diagram of proposed mechanism. *YES* Mode bit value of register is mapped with specific OS from the list in the boot configuration file which in turns loads corresponding OS partition. *NO* The usual behaviour for booting in existing systems is followed

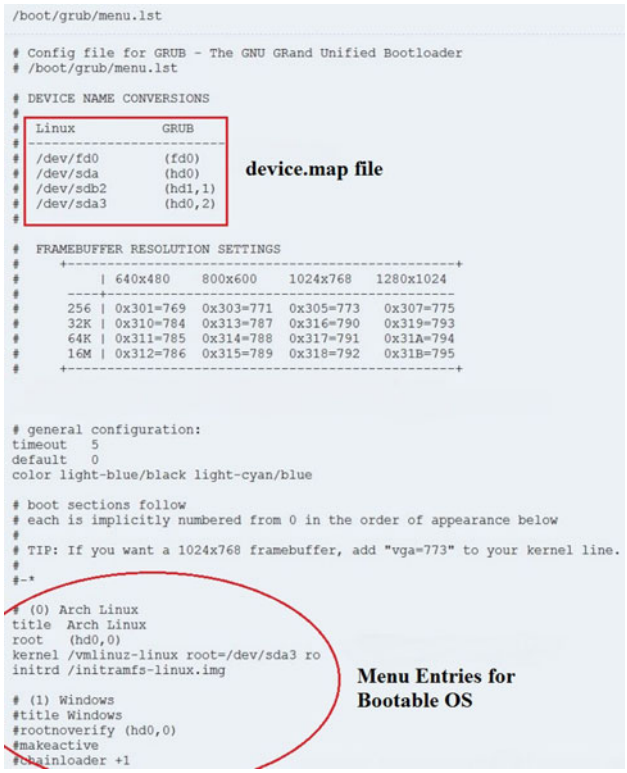

Fig. 5 Snapshot of boot configuration file

G2 (computer's "off" state), it sets the ACPI Reset Command [3]. It is generally referred to as "*Reset Register*". According to ACPI 2.0 specification, the reset mechanism must reset the entire system when implemented. This is logically equivalent to power cycling the machine, upon gaining control after reset; OSPM (Operating System-Directed Configuration and Power Management) will perform actions in like manner to a cold booting as if power supply has just being given. Now, when the system boots, Primary boot loader in turns load secondary boot loader and then it loads the boot configuration file (menu.lst in case of GRUB) where mapping of register value (mode-bit) with the partition table is done. Instead of showing GRUB graphical interface (in stage-2 booting by secondary boot loader), it reads the mode bit value which was set by the user in register before shutting down. Further, according to mapping provided in *configuration file,* it checks for hibernation state of the switched OS that whether it was previously hibernated or it is started for the first time by reading the *sleep enable register* and accordingly the further execution is done. In case of former, it resumes the previous state of OS via *waking vector* (mentioned in Introduction section). For the latter case, it loads the boot sector of partition table and reads the file containing the kernel. Afterwards, kernel reads ACPI configuration, starts kernel processes [1], loads drivers and set services for user. The sketch of our proposed approach is shown in Fig. 3.

Apart from this*, if the user shuts down or hibernates or reboots then it follows the existing usual behaviour.* Here, the synchronization or say balance with our mechanism is maintained during stage-2 booting, by checking mode-bit register; the mechanism is discussed in later section (Proposed mechanism).

2.2 Comparisons of proposed approach with existing system

Here we are discussing the benefits of our mechanism with the existing multi-OS system. As of now, we have discussed detailed description of existing system (Introduction section).

It can be inferred that:

- No state retention of previous OS is there as the current OS has to be completely shut down before selecting other environment. In our proposed mechanism, the current state of OS will be saved in a system file (e.g. *swsusp* in LINUX, *hiberfil.sys* in WINDOWS) using hibernation and then switch to different OS. Therefore, we are retaining system state.
- Time required for completely shutting down and restarting the other OS is reduced as in our proposed method, the current OS is hibernated and the other OS (to be switched) is resumed from its hibernation state except the case when it is started for the first time. Though time required for hibernation takes more time than shutting down but resuming time is comparatively very less than start of system. Overall, Time is reduced (see Observation section). Resuming time of OS from hibernation can even reduce by 10 % by a run-time page selection methodology through some approximation algorithms [5] [4].
- In existing systems, it's responsibility of the user to ensure the interaction with graphical interface at the time of restart in order to make appropriate choice for OS selection. By our method, the user is free from the responsibility of interaction with the graphical interface as the user is already making choice in current operating system. Therefore, user does not have to wait for the boot manager graphical interaction.

2.3 Comparisons of proposed mechanism with virtualization (Type 2)

2.3.1 Virtualization

It's a technology that creates a platform that emulates a hardware platform and allows multiple instances of an OS to use that platform, as though they have full and exclusive access to the underlying hardware. The guest OS uses Hypervisor technology [6] (also called a virtual machine manager (VMM)) for execution of its processes via host OS. A hypervisor is a program that allows multiple OS to share a single hardware host. Each OS appears to have the host's processor, memory, and other resources all to itself. However, the hypervisor is actually controlling the host processor and resources and making sure that the guest operating systems (*called virtual machines*) cannot disrupt each other. This can be seen in Fig. 6 [6] below.

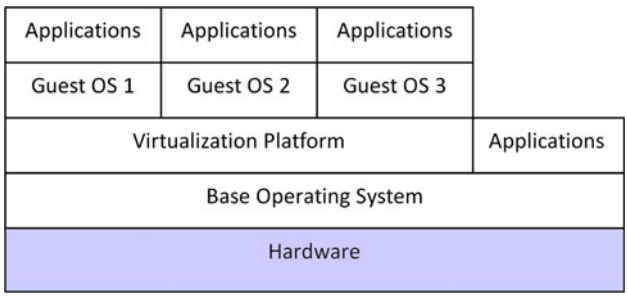

Fig. 6 Type 2 virtualization

Here, we are comparing hosted virtualization (type-2) with our proposed mechanism.

It can be observed that

- As it is layered approach (Fig. 6), it reads in code, looking for basic blocks, then inspect basic block to find sensitive instructions (system call). If found, replace with VM call (process called binary translation). This binary translation lowers the efficiency of process execution, but since our system is based on existing multi-OS systems where the OS is running on hardware directly. So, process execution is fast, making the system efficient.

Though basic blocks are cached and then executed in virtualization and eventually all basic blocks will be modified and cached, and will run at *near native* speed but then we require cache for that (cost factor).

- The process running in the guest OS cannot make use of all hardware available but only uses resources provided to guest OS as these virtual machines share hardware resources. For e.g. at the time of creating virtual machine, user has to allocate main memory (RAM) for guest OS say 512 MB then this virtualized OS can only use 512 MB of RAM. In our proposed technique, as at a particular instant of time, only one OS is running so the process being executed will use all the resource of system as in case of multi-OS systems.
- In order to make program execution efficient in virtualization, System requires a good amount of main memory (RAM) and more cache than usual. In our system, it requires storage space on secondary memory (HDD) for saving the state of OS (for Hibernation). The cost of our mechanism is balanced in comparison to virtualization.
- As a whole, Virtualization (Type-1 server virtualization) has big advantage in server consolidation (server farm), reducing space and power consumption which cannot be overlooked. At the same time, flexibility (running application in different environment simultaneously)

Table 1 Observed time in different systems

S. no.	Various system	Time taken to shutting down and restart	Time taken for hibernation and resume (s)
1.	Sony-VAIO	1 min and 15 s	55
2.	Lenovo	1 min and 10 s	54
3.	Acer	1 min and 12 s	44

Note these observations vary from one system to another. Here, observations are performed practically for windows; these are not precise figure but approx. figures

is achieved by Type-2 virtualization, but as we are reducing switching time among different OS to a certain extent (Table 1. in Observation section), keeping existing system in mind this is certainly an edge over hosted virtualization (Type-2).

- In our proposed system, the computer context data is stored in a storage area and these data are prevented from being accessed by any computing mode other than one from which context data was saved. This limitation of accessing the contextual data from other computing environments minimizes the chance of data contamination and computer security and integrity is maintained. This is not the case with hosted virtualization as security is an issue.
- There are several use cases where virtualization is not appropriate to use. There are situations in research and development field where researchers need to run a program or software directly on hardware resources in order to analyze the efficiency of the software. Even for educational purposes where students need to work on various environments, system memory (RAM) and other hardware resources like processor etc. are a bottleneck as it is not affordable in educational field.

3 Observations

Here, the proposed system executing the current environment is hibernated and then it resumes the other environment. In sum, there are two points to ponder upon:

- The time required for the hibernation is more than shutting down the OS as hibernation takes some overhead for writing the content of RAM in a file on HDD.
- The time required for the resuming is very less than starting the system as in latter case kernel along with its services and several drivers has to be loaded where as former simply reads the content of file to RAM.

Taking above points to consideration, we observed this time in several systems, we found that:

Time taken for hibernation was more than shut down but on the contrary, resuming takes very less time compared to start of the system. The time differences in latter is considerably higher than former. On the whole we are getting time difference which can be seen in the table given below.

Here by calculating the reduction, we concluded that the reduction in time is nearly 20–30 % (from Restart to Hibernate and resume).

4 The proposed mechanism

The flow diagram of our proposed system is shown in the following diagram given in Fig. 4.

The basic concept that constitutes our system is Reset mechanism (ACPI routine) with hibernation mode and inclusion of R/W register for OS switching. The default value is there in register in case when user does not go for OS switching otherwise user sets the mode-bit value in register via some specific software system. The specific software will detect the installed operating systems in the system and provides an interface through which user can select option for switching to different OS. The menu.lst file (in case of GRUB) or BCD file (boot configuration data in case of windows, earlier known as boot.ini) is responsible for keeping information about all installed computing environments and are used for OS selection at the time of booting. This file is also used to point to the locations of each of the operating systems. These configuration files are located in primary disks and can be used by our software for providing various choices to user for switching. After this, the system state is saved by hibernation mechanism as discussed in Introduction section. Every OS supports hibernation if hardware is ACPI complaint, which nowadays every system is. For example: Windows has *hiberfil.sys*, Linux has *swsusp* and *uswsusp*, Mac Os uses *Safe Sleep Feature.* After saving state, instead of going to power off state (power state G2) as in case of hibernation, reset mechanism starts. This is an ACPI command which sets *Reset Register* and following steps are happened in sequence:

All logic is reset. This means sending the respective reset commands to various bits of hardware including the CPU, memory controller, peripheral controllers, etc. The computer is then bootstrapped. This is the "perform actions in like manner to a cold boot" part. The motherboard performs the same steps as it would if the power supply had just become ready after the power button being pressed.

Our mode-bit register attached to the motherboard is not being reset in the first step of above sequence. It retains its value as power source for this register is small button sized cell. Further, following second step booting starts. At stage-2 booting, the boot manager will check the mode-bit

value of register that whether it's set or not. If it is not set (default value) then it loads OS in existing fashion and takes user's input for OS selection from graphical interface interaction of boot manager (GRUB). At the same time, user's input is mapped with specific OS in boot configuration file which in turns mapped with boot sector of partition table. Then following the further booting sequence, selected OS is loaded and user can work with it.

If the value is set in register, then mapping of bit value with specific OS is done in boot configuration file instead of showing graphical interface for user's input. Here, it checks the specific OS was previously hibernated or not by checking (SLP_EN register) *sleep enable register* (as it would be set at time of hibernation). If it was earlier hibernated then it resumes saved state via *waking_vector_loader* [3] otherwise it loads boot sector of partition table to load the specified OS and finally user can work with it.

Now we are taking the GRUB as an example (*case study*) to illustrate the mapping of user's input for selecting OS with partition table. The purpose of showing this mapping is to tell how precisely the mapping is done in existing system and what our proposed idea is. Actually GRUB configuration is based on four files that are discussed below in brief [7]:

/boot/grub/menu.lst: This file contains all information about operating systems that can be booted with GRUB.

/boot/grub/device.map: This file is used for translating device names from GRUB and BIOS to Linux devices.

/etc./*grub.conf:* This file contains commands for installing the boot loader correctly.

/etc./*sysconfig/bootloader:* This file is used for configuring the boot loader every time the new kernel is installed and updating it to bootloader configuration file (menu.lst).

NOTE GRUB defines any storage device as hdX, Device and partition numbering begin at zero. For e.g. first boot priority (hard disk) recognized will be hd0, second device will be hd1. This also applies to partition of particular device for e.g. (hd0,0): first partition of primary hard disk, (hd0,3): fourth partition of primary hard disk (usually an extended partition), (hd0,4): first logical partition of primary hard disk, (hd1,0): first partition of secondary hard disk [8].

A snapshot of*/boot/grub/menu.lst* is given in Fig. 5 [7]; actually this is loaded when system is booted, so that GRUB does not need to be reinstalled after every change to the file. In the figure, we can see the *menu entries* for several OS that are installed in the system (shown by circle). Similarly, we can also see Device Name Conversions in *device.map* file (shown in the figure by square), here logical root of specific OS is translated to actual device name. In case of windows, it maps with device name where *chainloader* is present which in turns loads another bootloader (rather than kernel image).

Moreover, we can see the general configuration of GRUB like *timeout: 5* and *default*. This shows that after 5 s without user input, GRUB automatically boots the default entry. Further, with the specific menu entry shown below, it shows that kernel is located in*/dev/sda7* and in case of resume (from Hibernation), it loads*/dev/sda9* from device file.

```
title linux
    root (hd0,4)
    kernel /boot/vmlinuz root=/dev/sda7 vga=791 resume=/dev/sda9
    initrd /boot/initrd
```

So, in our proposed mechanism similar type of translation is done, instead of taking user's input it will take the bit value from register to map with specified 'title' of OS. The above description of translation will be followed when switching is not done and system starts normally.

5 Conclusion

With our proposed mechanism, we are improving the trends of existing multi-OS system by achieving the retention of previous system state using hibernation technique and reducing time required while switching among different operating system environments. Moreover, by this approach, it eliminates the waiting time for the user to interact with the GRUB interface. Though, here we are not achieving simultaneous execution of programs running in multiple operating environments as in case of virtualization but efficient execution of processes are carried along with the full utilization of hardware resources. The hardware device containing mode bit (binary value) which contains simple register and a small button sized cell which is cost effective and durable.

Acknowledgments We acknowledge the efforts of our Professor Rayan Goudar of Graphic Era University, Dehradun, India whose encouragement, guidance and support from the initial to the final level. The Computer science department of University is gratefully acknowledged for its laboratory facilities. Our well-wishers are appreciated for their various contributions to the success of this work.

References

1. 6 Stages of Linux Boot Process, Retrieved April 05, 2012 from http://www.thegeekstuff.com/2011/02/linux-boot-process/.
2. Intel Corporation (2004) "ATX Specification Version 2.2".
3. Hewlett-Packard, Intel, Microsoft, Phoenix and Toshiba (2011) "Advance configuration and power interface specification, Revision 5.0," Retrieved March 28, 2012 from http://www.acpi.info/.
4. Jiong Z (2006) Linux kernel complete analysis [M]. China Machine Press, Beijing.
5. A run-time page selection methodology for efficient quality based resuming. 17th International IEEE conference on embedded and real time computing systems and applications (2011) IEEE.

6. King ST, Dunlap GW, Chen PM Operating system support for virtual machines, Department of Electrical Engineering and Computer Science, University of Michigan.
7. GRUB Configuration file Retrieved March 02, 2012 from http://archlinux.org/GRUB.
8. Getting to know LILO and GRUB Retrieved March 02, 2012 from www.ibm.com/developerworks/linux/library/.
9. Bai YW, Hsu HT (2007) Design and implementation of an instantaneous turning-on mechanism for PCs, Revised manuscript received 18 Sep 2007 IEEE.