A Design of the Persistent Operating System with Non-volatile Memory

Ren Ohmura Nobuyuki Yamasaki Yuichiro Anzai
Graduate School of
Science for Open and Environmental Systems
Keio University, Japan
{ren,yamasaki,anzai}@ayu.ics.keio.ac.jp

Abstract

In today's computing environment, novel memory devices with non-volatile characteristics are increasing in practicality when used as the main memory, due to the persistence with no additional battery that significantly enhances usability of personal devices.

In our research, we have built a persistent operating system using non-volatile main memory. This paper describes our strategy in detail on how atomicity of execution is maintained for each device driver method so that the state of peripheral devices can also recovered consistently. The method was implemented on the Linux kernel using a UART device driver. We have confirmed correct system recovery through our experiments.

1 Introduction

Many computer systems today lose their application context when they are faced with an unpredictable power failure. Additional power supplies (e.g. UPS or second battery) are needed to protect the system against this problem. However, this increases the size and cost of the system especially in personal devices such as PDAs or Set-Top-Boxes.

Past experiments on *persistent operating systems* make the application's execution states and data recoverable significantly increasing the reliability and usefulness of computer systems. Based on this system, a user can continue his/her task after an unpredictable power failure with minimum data loss without rebooting the OS and re-executing each application, which takes enough time to interrupt the user's work. Furthermore, if persistence is achieved in a pretty short periods, a computer system can be made to run with an unstable power source such as solar power. However, past experiments have assumed that permanent devices in the system are either disks or tapes, known to be "slow devices". It was thus difficult to preserve the system state in a short period without decreasing system performance.

Recently, memory devices with non-volatile characteristic that can operate without any batteries (e.g. FeRAM and MRAM) are becoming more practical. As these memory devices increase in their capacity and speed, they can be used as the main memory.

Therefore, we are currently developing a persistent operating system on a non-volatile main memory system towards fine grain persistence without any additional battery. Even on such a system, not only does the main memory state but also the CPU and peripheral device states must be recovered consistently after power failure. This paper describes our basic concept on how to recover the consistent system state while ensuring atomicity of the device driver method execution. Our experiment was implemeted the Linux kernel.

2 Motivation

Most existing research on the persistent operating system focus on when and how to store the main memory state with the CPU state into the permanent storage devices such as disks[3, 4, 5, 7].

On a system with non-volatile main memory, the CPU automatically saves the current main memory state from each store instruction. The CPU state can be saved in a similar way to the context switch, by writing to the main memory. We developed a method based on this to save both states consistently with low overhead using explicit timing called "checkpoints". It copies the memory space modified since the previous checkpoint to free space, similar to the "side file" scheme, while arranging the memory management structure to reduce overhead. Unfortunately, in the worst-case scenario, this would require twice as much as the normal memory space and needs time to save all the changed memory space and the CPU state, which is unavoidable according to the side file scheme. Even if logging scheme is used, the overhead increases even more because logs have to be taken on each instruction.

Few experiments examine the peripheral devices state. In order to continue system execution, the all states of the main memory, CPU and peripheral devices must be recovered because all are intertwined with each other.

Thus, our goals are:

- to be able to save and restore the entire consistent state of the system including peripheral devices
- to reduce memory space and the time required to store system states

To achieve these goals, we focused on controlling the execution of device drivers since there are power management schemes that make suspend/resume and hibernation possible, such as APM and ACPI [2, 1]. However, they are unable to handle an unpredictable power failure because they save the state of peripheral devices during special time when they can.

3 Design

The type of CPU determines the amount of the CPU state that needs to be saved. The system hardware can be designed to allow sufficient time to store the state during power failure, which can be detected by an interrupt in a circuit monitoring the power level, by slowing the power attenuation with the device like capacitors. If the system consisted of only the (non-volatile) main memory and CPU registers (used for calculation), it would recover the execution correctly with the CPU state stored during power failure and the memory state existing on the main memory. Hence, only the CPU state is stored when a power failure occurs. Existing memory state, except the devices driver areas, is used for recovery, which reduces overhead.

However, the amount of peripheral device states required to be saved cannot be estimated at system hardware design time. Most systems allow users to add peripheral devices to extend the system(e.g. PCMCIA cards). Some states of devices cannot be read by the CPU, which makes it impossible to store all of the states of peripheral devices during power failure. Furthermore, many devices require some access to begin and perform asynchronously, so they may act incorrectly or harm the system in the worst case due to the loss of the previous access and states of peripheral devices if the system restarts from the precise point of power failure.

Therefore, callback functions written in device drivers like APM and ACPI are used for recovering the states of devices at that explicit point. We defined that point as the head of each device driver method. The state of the CPU and the memory space used by a device driver are saved at the beginning of each device driver method. If the CPU context is in a device driver when a power failure occurs, the recovery operation restores this memory state, calls the callback function to recover the consistent device state, and then restarts the system with the saved CPU state. This way, the system restarts from the head of the device driver method, and repeat it with the consistent device state as each unit of request for the devices.

In essence, our basic concept for maintaining consistency of the entire system state is to ensure atomicity of execution of each device driver method. In the following sections, we will describe the Linux device driver ,the implementation target, and discuss in detail the design of our strategy.

3.1 Linux device driver

Although there are many peripheral devices and methods to implement them, we observed a more simple case. Each peripheral device is managed as a file in Linux distinguished by a `kdev_t` value, which is a set of major and minor numbers [1]. When a user application calls a device handling function, such as `open`, `read`, `write`, or `close`, the file operation handler looks up the appropriate device driver and calls a suitable method in it.

Regular kernel functions used in device drivers are generally fixed. For example, `wakeup` and `down` and other similar functions are used to wake up and put threads to sleep; `kill_fasync` is used for handling asynchronous device signals; and `queue_task` is used for queuing the thread of other kernel components called the *bottom half handler* or *tasklet*, such as protocol stacks.

3.2 Basic Operations

Figure 1 shows the basic sequence of the file operation handler extended by our scheme while entering into and returning from a device driver method. When a user issues a request to a device, shown in step (1) in Figure 1, the file operation handler looks up the appropriate device driver. Before calling the method, it saves the memory of this device driver and the CPU states to the area specified by `kdev_t` value, shown in step (2) and (3). Then, it calls the method in step (4). After returning from the device driver, it destroys the CPU state saved in step (3), shown in step (5), and returns to the user level shown in step (6).

The recovery process looks for the CPU state saved in (3) at first. Next, if there is no CPU state relating with any device driver, the recovery process restarts the system with the CPU state saved during power failure. If there is one or more CPU states, the recovery operation restores the device driver's memory state saved at (2) and then restarts the system with the CPU states saved at (3).

Nevertheless, we have to consider following conditions:

[1]Although network devices are not managed as a file, we leave this matter to next paper.

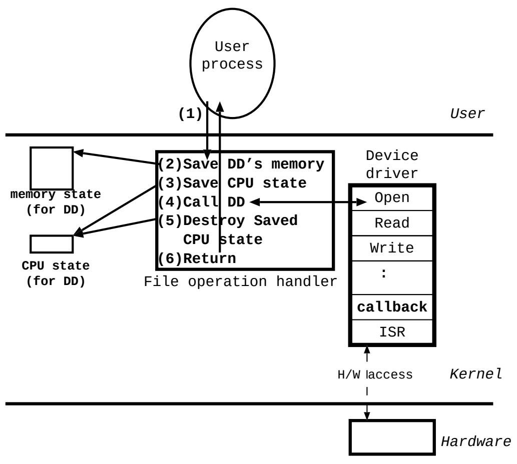

Figure 1. The operations in the file handler

context switch in a device driver
cooperation with external components
interrupt handling

3.3 Context switch

There are many cases that a context switch occurs in the device driver due to the lack of required data, mutual exclusion, and so on.

If a thread context switches in a device driver and a power failure occurs while another thread is running outside, the CPU state saved during power failure is that of the outside thread. In order to restart the thread inside the device driver from the start of that method, the recovery process overwrites the CPU state of this thread on the context table with that saved at step (3) in Figure 1. The subsequent scheduling point of this thread is changed from the inside to the head of the device driver by this operation.

In addition, the state of this thread (e.g. *running* or *blocked*) has to be recovered. Therefore, the thread state is also recorded by step (3), and the recovery process overwrites it on the context table.

3.4 Cooperation with external components

Generally, device drivers cooperate at the user level or with other kernel components, writing data to or reading data from the buffers. These buffers are not required to be restored during the recovery process.

If the device driver method reads from a buffer, the current data at the buffer is the same before power failure. Then, the operations of the method can re-execute with the same data after recovery. When writing to a buffer, the recovered request executes the same operations and gets the same or more appropriate data. Then external components work correctly. Thus, the buffer concerning external components does not need to save and recover. This observation decreases the overhead of our scheme.

However, pointers and index variables of the buffers need to be recovered as indicating the same position of the buffer after recovery.

3.5 Interrupt

When a power failure occurs while handling an interrupt, the interrupt acts as if it had not been executed.

Interrupts from peripheral devices can be roughly classified into two groups. One is the result of a request from the device driver. The other is an active event from a device such as a button press or a network packet arrival. In the first case, the same interrupt is expected by repeated requests. In the second case, it does not matter since the interrupt acts as if it had never occurred.

A typical interrupt service routine does one or a combination of the following operations with the kernel functions mentioned in section 3.1. The routine registers a request for the execution of other kernel components; it sends a signal to the corresponding thread as notification of asynchronous I/O; and it wakes the sleeping thread waiting for an event (data) from a device.

If a power failure occurs after they are operated, the recovery process invalidates the operations. For this reason, kernel functions, such as `queue_task`, `kill_fasync` and `wakeup`, record these requests as a log.

4 Implementation

4.1 Kernel Function

As noted in section 3.4, it does not need to save all of the memory in a device driver. To reduce overhead and maintain flexibility of device driver implementation, we added a new kernel function that registers the memory space to save when entering the device driver. The arguments in this function are the `kdev_t` , the head address of the memory, and the size. The initialization code in a device driver, which is only executed during the first boot sequence, registers the memory space using this function.

We also extended some kernel functions, such as `queue_task`, `fasync_kill`, `wakeup`, et cetera, as mentioned above.

4.2 Callback Function

The difference between the callback functions in our scheme and the APM and ACPI is that they have the responsibility to recover complete states of peripheral devices

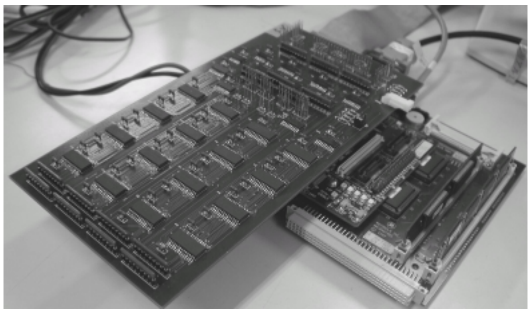

Figure 2. FeRAM board and MPC860FADS

at the head of the device driver method. In order to do that, the device driver has to be written as storing the each change of the device state to the registered main memory and callback functions reflect it after initializing the device.

For example, with the UART device, current configurations, such as the baud rate, the number of stop bits, and the number of parity bits, are kept in the registered main memory. The callback function, called after restoring the memory state, has to set these to the device as the current (at the head of the method) state.

4.3 Recovery Operation

The steps for recovery operation are:

1. Recover the memory state of device drivers where the saved CPU contexts exist
2. Call the callback functions of device drivers
3. Overwrite appropriate contexts on the context table with the CPU state at the beginning of the device driver method (section 3.3)
4. Change thread states which had their context switched in device drivers (section 3.3)
5. Undo requests from the device driver based on logs(section 3.5)

Then, if the CPU state saved during power failure is neither in the device driver nor in the interrupt handler, the system restarts with this CPU state. Otherwise, if it is in the device driver, the system restarts with the same thread in the CPU state, saved at the head of the device driver method. If it is in the interrupt handler, the scheduler restarts.

5 Experimentation

We tested our strategy on the MPC860FADS, an evaluation board of the PowerPC core CPU for embedded system, with FeRAM boards as the main memory (Figure 2). The prototype was implemented on Linux 2.4.4 with a UART device driver.

We ran a simple process, which incremented a counter value and printed it on the UART, and then, shut off the power supply. We confirmed that the system showed the next value before power failure on the UART and continued performing correctly. At that time, the registered memory space was only 152 bytes and the increased execution time ratio measured by the `time` command was less than 0.5%.

6 Summary and Future Work

We illustrated our strategy of recovering an entire system state consistently given the condition that the main memory is non-volatile, focusing on the device driver. The basic concept of our scheme is to ensure atomicity of the device driver method, observing whether there is thread present in the device driver at power failure. We implemented our prototype on the Linux kernel and its UART device driver, and confirmed the system including the actual device performed correctly after recovery.

The experimentation we conducted was only a simple case, assuming that only one thread is in the device driver. However, more than one thread usually enters the same device driver simultaneously, especially in sharable devices. Also, we did not consider some device driver method such as `mmap`. We plan to solve these questions towards the computer system performing with unstable power supply.

References

[1] *Advanced Configuration and Power Interface Specification Revision 2.0*, July 2000. http://acpi.info/spec.htm.

[2] *Advanced Power Management BIOS Interface Specification Revision 1.2*, February 1996. http://www.microsoft.com/hwdev/archive/ BUSBIOS/amp_12.asp.

[3] A. Dearle and D. Hulse. Operating system support for persistent systems: past, present and future. *Software – Practice-and-Experience*, 30(4):295–324, 2000.

[4] K. Elphinstone, S. Russell, G. Heiser, and J. Liedtke. Supporting Persistent Object Systems in a Single Address Space. In *Proc.7th POS*, 1996. http://www.cse.unsw.edu.au/~disy/papers/ index.html.

[5] A. Lindstrom, A. Dearle, R. di Bona, S. Norris, J. Rosenberg, and F. Vaugha. Persistence in Grasshopper Kernel. In *Proc. of the Eighteenth Australian Computer Science Conf.*, pages 329–338, 1995.

[6] A. Rubini and J. Corbet. *Linux Device Drivers, 2nd Edition*. O'Reilly & Associates, Inc., June 2001.

[7] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS:a fast capability system. In *17th ACM Symposium on Operating System Principles(SOSP '99)*, pages 170–185, 1999.