Linux Ia64 Documentation

The kernel development community

Jun 10, 2024

CONTENTS

CHAPTER

ONE

LINUX KERNEL RELEASE FOR THE IA-64 PLATFORM

These are the release notes for Linux since version 2.4 for IA-64 platform. This document provides information specific to IA-64 ONLY, to get additional information about the Linux kernel also read the original Linux README provided with the kernel.

1.1 Installing the Kernel

- IA-64 kernel installation is the same as the other platforms, see original README for details.

1.2 Software Requirements

Compiling and running this kernel requires an IA-64 compliant GCC compiler. And various software packages also compiled with an IA-64 compliant GCC compiler.

1.3 Configuring the Kernel

Configuration is the same, see original README for details.

Compiling the Kernel:

- Compiling this kernel doesn' t differ from other platform so read the original README for details BUT make sure you have an IA-64 compliant GCC compiler.

1.4 IA-64 Specifics

- General issues:
  - Hardly any performance tuning has been done. Obvious targets include the library routines (IP checksum, etc.). Less obvious targets include making sure we don' t flush the TLB needlessly, etc.
  - SMP locks cleanup/optimization

- IA32 support. Currently experimental. It mostly works.

CHAPTER

TWO

MEMORY ATTRIBUTE ALIASING ON IA-64

Bjorn Helgaas <bjorn.helgaas@hp.com>

May 4, 2006

2.1 Memory Attributes

Itanium supports several attributes for virtual memory references. The attribute is part of the virtual translation, i.e., it is contained in the TLB entry. The ones of most interest to the Linux kernel are:

WB	Write-back (cacheable)
UC	Uncacheable
WC	Write-coalescing

System memory typically uses the WB attribute. The UC attribute is used for memory-mapped I/O devices. The WC attribute is uncacheable like UC is, but writes may be delayed and combined to increase performance for things like frame buffers.

The Itanium architecture requires that we avoid accessing the same page with both a cacheable mapping and an uncacheable mapping[1].

The design of the chipset determines which attributes are supported on which regions of the address space. For example, some chipsets support either WB or UC access to main memory, while others support only WB access.

2.2 Memory Map

Platform firmware describes the physical memory map and the supported attributes for each region. At boot-time, the kernel uses the EFI GetMemoryMap() interface. ACPI can also describe memory devices and the attributes they support, but Linux/ia64 currently doesn’t use this information.

The kernel uses the efi_memmap table returned from GetMemoryMap() to learn the attributes supported by each region of physical address space. Unfortunately, this table does not completely describe the address space because some machines omit some or all of the MMIO regions from the map.

The kernel maintains another table, kern_memmap, which describes the memory Linux is actually using and the attribute for each region. This contains only system memory; it does not contain MMIO space.

The kern_memmap table typically contains only a subset of the system memory described by the efi_memmap. Linux/ia64 can't use all memory in the system because of constraints imposed by the identity mapping scheme.

The efi_memmap table is preserved unmodified because the original boot-time information is required for kexec.

2.3 Kernel Identify Mappings

Linux/ia64 identity mappings are done with large pages, currently either 16MB or 64MB, referred to as "granules." Cacheable mappings are speculative[2], so the processor can read any location in the page at any time, independent of the programmer' s intentions. This means that to avoid attribute aliasing, Linux can create a cacheable identity mapping only when the entire granule supports cacheable access.

Therefore, kern_memmap contains only full granule-sized regions that can referenced safely by an identity mapping.

Uncacheable mappings are not speculative, so the processor will generate UC accesses only to locations explicitly referenced by software. This allows UC identity mappings to cover granules that are only partially populated, or populated with a combination of UC and WB regions.

2.4 User Mappings

User mappings are typically done with 16K or 64K pages. The smaller page size allows more flexibility because only 16K or 64K has to be homogeneous with respect to memory attributes.

2.5 Potential Attribute Aliasing Cases

There are several ways the kernel creates new mappings:

2.5.1 mmap of /dev/mem

This uses remap_pfn_range(), which creates user mappings. These mappings may be either WB or UC. If the region being mapped happens to be in kern_memmap, meaning that it may also be mapped by a kernel identity mapping, the user mapping must use the same attribute as the kernel mapping.

If the region is not in kern_memmap, the user mapping should use an attribute reported as being supported in the EFI memory map.

Since the EFI memory map does not describe MMIO on some machines, this should use an uncacheable mapping as a fallback.

2.5.2 mmap of /sys/class/pci_bus/…/legacy_mem

This is very similar to mmap of /dev/mem, except that legacy_mem only allows mmap of the one megabyte "legacy MMIO" area for a specific PCI bus. Typically this is the first megabyte of physical address space, but it may be different on machines with several VGA devices.

"X" uses this to access VGA frame buffers. Using legacy_mem rather than /dev/mem allows multiple instances of X to talk to different VGA cards.

The /dev/mem mmap constraints apply.

2.5.3 mmap of /proc/bus/pci/…/??.?

This is an MMIO mmap of PCI functions, which additionally may or may not be requested as using the WC attribute.

If WC is requested, and the region in kern_memmap is either WC or UC, and the EFI memory map designates the region as WC, then the WC mapping is allowed.

Otherwise, the user mapping must use the same attribute as the kernel mapping.

2.5.4 read/write of /dev/mem

This uses copy_from_user(), which implicitly uses a kernel identity mapping. This is obviously safe for things in kern_memmap.

There may be corner cases of things that are not in kern_memmap, but could be accessed this way. For example, registers in MMIO space are not in kern_memmap, but could be accessed with a UC mapping. This would not cause attribute aliasing. But registers typically can be accessed only with four-byte or eight-byte accesses, and the copy_from_user() path doesn't allow any control over the access size, so this would be dangerous.

2.5.5 ioremap()

This returns a mapping for use inside the kernel.

If the region is in kern_memmap, we should use the attribute specified there.

If the EFI memory map reports that the entire granule supports WB, we should use that (granules that are partially reserved or occupied by firmware do not appear in kern_memmap).

If the granule contains non-WB memory, but we can cover the region safely with kernel page table mappings, we can use ioremap_page_range() as most other architectures do.

Failing all of the above, we have to fall back to a UC mapping.

2.6 Past Problem Cases

2.6.1 mmap of various MMIO regions from /dev/mem by "X" on Intel platforms

The EFI memory map may not report these MMIO regions.

These must be allowed so that X will work. This means that when the EFI memory map is incomplete, every /dev/mem mmap must succeed. It may create either WB or UC user mappings, depending on whether the region is in kern_memmap or the EFI memory map.

2.6.2 mmap of 0x0-0x9FFFF /dev/mem by "hwinfo" on HP sx1000 with VGA enabled

The EFI memory map reports the following attributes:

0x00000-0x9FFFF	WB only	
0xA0000-0xBFFFF	UC only	(VGA frame buffer)
0xC0000-0xFFFFF	WB only	

This mmap is done with user pages, not kernel identity mappings, so it is safe to use WB mappings.

The kernel VGA driver may ioremap the VGA frame buffer at 0xA0000, which uses a granule-sized UC mapping. This granule will cover some WB-only memory, but since UC is non-speculative, the processor will never generate an uncacheable reference to the WB-only areas unless the driver explicitly touches them.

2.6.3 mmap of 0x0-0xFFFFF legacy_mem by "X"

If the EFI memory map reports that the entire range supports the same attributes, we can allow the mmap (and we will prefer WB if supported, as is the case with HP sx[12]000 machines with VGA disabled).

If EFI reports the range as partly WB and partly UC (as on sx[12]000 machines with VGA enabled), we must fail the mmap because there' s no safe attribute to use.

If EFI reports some of the range but not all (as on Intel firmware that doesn' t report the VGA frame buffer at all), we should fail the mmap and force the user to map just the specific region of interest.

2.6.4 mmap of 0xA0000-0xBFFFF legacy_mem by "X" on HP sx1000 with VGA disabled

The EFI memory map reports the following attributes:

```
0x00000-0xFFFFF WB only (no VGA MMIO hole)
```

This is a special case of the previous case, and the mmap should fail for the same reason as above.

2.6.5 read of /sys/devices/…/rom

For VGA devices, this may cause an ioremap() of 0xC0000. This used to be done with a UC mapping, because the VGA frame buffer at 0xA0000 prevents use of a WB granule. The UC mapping causes an MCA on HP sx[12]000 chipsets.

We should use WB page table mappings to avoid covering the VGA frame buffer.

2.7 Notes

[1] SDM rev 2.2, vol 2, sec 4.4.1. [2] SDM rev 2.2, vol 2, sec 4.4.6.

CHAPTER

THREE

EFI REAL TIME CLOCK DRIVER

S. Eranian <eranian@hpl.hp.com>

March 2000

3.1 1. Introduction

This document describes the efirtc.c driver has provided for the IA-64 platform.

The purpose of this driver is to supply an API for kernel and user applications to get access to the Time Service offered by EFI version 0.92.

EFI provides 4 calls one can make once the OS is booted: GetTime(), SetTime(), GetWakeupTime(), SetWakeupTime() which are all supported by this driver. We describe those calls as well the design of the driver in the following sections.

3.2 2. Design Decisions

The original ideas was to provide a very simple driver to get access to, at first, the time of day service. This is required in order to access, in a portable way, the CMOS clock. A program like /sbin/hwclock uses such a clock to initialize the system view of the time during boot.

Because we wanted to minimize the impact on existing user-level apps using the CMOS clock, we decided to expose an API that was very similar to the one used today with the legacy RTC driver (driver/char/rtc.c). However, because EFI provides a simpler services, not all ioctl() are available. Also new ioctl()s have been introduced for things that EFI provides but not the legacy.

EFI uses a slightly different way of representing the time, noticeably the reference date is different. Year is the using the full 4-digit format. The Epoch is January 1st 1998. For backward compatibility reasons we don’ t expose this new way of representing time. Instead we use something very similar to the struct tm, i.e. struct rtc_time, as used by hwclock. One of the reasons for doing it this way is to allow for EFI to still evolve without necessarily impacting any of the user applications. The decoupling enables flexibility and permits writing wrapper code is ncase things change.

The driver exposes two interfaces, one via the device file and a set of ioctl()s. The other is read-only via the /proc filesystem.

As of today we don’ t offer a /proc/sys interface.

To allow for a uniform interface between the legacy RTC and EFI time service, we have created the include/linux/rtc.h header file to contain only the “public” API of the two drivers. The specifics of the legacy RTC are still in include/linux/mc146818rtc.h.

3.3 3. Time of day service

The part of the driver gives access to the time of day service of EFI. Two ioctl()s, compatible with the legacy RTC calls:

Read the CMOS clock:

```
ioctl(d, RTC_RD_TIME, &rtc);
```

Write the CMOS clock:

```
ioctl(d, RTC_SET_TIME, &rtc);
```

The rtc is a pointer to a data structure defined in rtc.h which is close to a struct tm:

```
struct rtc_time {
        int tm_sec;
        int tm_min;
        int tm_hour;
        int tm_mday;
        int tm_mon;
        int tm_year;
        int tm_wday;
        int tm_yday;
        int tm_isdst;
};
```

The driver takes care of converting back an forth between the EFI time and this format.

Those two ioctl()s can be exercised with the hwclock command:

For reading:

```
# /sbin/hwclock --show
Mon Mar  6 15:32:32 2000  -0.910248 seconds
```

For setting:

```
# /sbin/hwclock --systohc
```

Root privileges are required to be able to set the time of day.

3.4 4. Wakeup Alarm service

EFI provides an API by which one can program when a machine should wakeup, i.e. reboot. This is very different from the alarm provided by the legacy RTC which is some kind of interval timer alarm. For this reason we don’t use the same ioctl()s to get access to the service. Instead we have introduced 2 news ioctl()s to the interface of an RTC.

We have added 2 new ioctl()s that are specific to the EFI driver:

Read the current state of the alarm:

```
ioctl(d, RTC_WKALM_RD, &wkt)
```

Set the alarm or change its status:

```
ioctl(d, RTC_WKALM_SET, &wkt)
```

The wkt structure encapsulates a struct rtc_time + 2 extra fields to get status information:

```
struct rtc_wkalrm {

        unsigned char enabled; /* =1 if alarm is enabled */
        unsigned char pending; /* =1 if alarm is pending  */

        struct rtc_time time;
}
```

As of today, none of the existing user-level apps supports this feature. However writing such a program should be hard by simply using those two ioctl().

Root privileges are required to be able to set the alarm.

3.5 5. References

Checkout the following Web site for more information on EFI:

http://developer.intel.com/technology/efi/

CHAPTER FOUR

IPF MACHINE CHECK (MC) ERROR INJECT TOOL

IPF Machine Check (MC) error inject tool is used to inject MC errors from Linux. The tool is a test bed for IPF MC work flow including hardware correctable error handling, OS recoverable error handling, MC event logging, etc.

The tool includes two parts: a kernel driver and a user application sample. The driver provides interface to PAL to inject error and query error injection capabilities. The driver code is in arch/ia64/kernel/err_inject.c. The application sample (shown below) provides a combination of various errors and calls the driver’ s interface (sysfs interface) to inject errors or query error injection capabilities.

The tool can be used to test Intel IPF machine MC handling capabilities. It’ s especially useful for people who can not access hardware MC injection tool to inject error. It’ s also very useful to integrate with other software test suits to do stressful testing on IPF.

Below is a sample application as part of the whole tool. The sample can be used as a working test tool. Or it can be expanded to include more features. It also can be a integrated into a library or other user application to have more thorough test.

The sample application takes err.conf as error configuration input. GCC compiles the code. After you install err_inject driver, you can run this sample application to inject errors.

Errata: Itanium 2 Processors Specification Update lists some errata against the pal_mc_error_inject PAL procedure. The following err.conf has been tested on latest Montecito PAL.

err.conf:

```
#This is configuration file for err_inject_tool.
#The format of the each line is:
#cpu, loop, interval, err_type_info, err_struct_info, err_data_
↪buffer
#where
#     cpu: logical cpu number the error will be inject in.
#     loop: times the error will be injected.
#     interval: In second. every so often one error is injected.
#     err_type_info, err_struct_info: PAL parameters.
#
#Note: All values are hex w/o or w/ 0x prefix.
```

(continues on next page)

(continued from previous page)

```
#On cpu2, inject only total 0x10 errors, interval 5 seconds
#corrected, data cache, hier-2, physical addr(assigned by tool␣
 ↪code).
#working on Montecito latest PAL.
2, 10, 5, 4101, 95

#On cpu4, inject and consume total 0x10 errors, interval 5 seconds
#corrected, data cache, hier-2, physical addr(assigned by tool␣
 ↪code).
#working on Montecito latest PAL.
4, 10, 5, 4109, 95

#On cpu15, inject and consume total 0x10 errors, interval 5 seconds
#recoverable, DTR0, hier-2.
#working on Montecito latest PAL.
0xf, 0x10, 5, 4249, 15
```

The sample application source code:

err_injection_tool.c:

```
/*
 * This program is free software; you can redistribute it and/or␣
 ↪modify
 * it under the terms of the GNU General Public License as␣
 ↪published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,␣
 ↪but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, GOOD TITLE␣
 ↪or
 * NON INFRINGEMENT.  See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 *
 * Copyright (C) 2006 Intel Co
 *	Fenghua Yu <fenghua.yu@intel.com>
 *
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <sched.h>
```

(continues on next page)

(continued from previous page)

```
#include <unistd.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <errno.h>
#include <time.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <sys/shm.h>

#define MAX_FN_SIZE           256
#define MAX_BUF_SIZE          256
#define DATA_BUF_SIZE                 256
#define NR_CPUS               512
#define MAX_TASK_NUM          2048
#define MIN_INTERVAL          5       // seconds
#define       ERR_DATA_BUFFER_SIZE    3       // Three 8-byte.
#define PARA_FIELD_NUM                5
#define MASK_SIZE             (NR_CPUS/64)
#define PATH_FORMAT "/sys/devices/system/cpu/cpu%d/err_inject/"

int sched_setaffinity(pid_t pid, unsigned int len, unsigned long␣
 ↪*mask);

int verbose;
#define vbprintf if (verbose) printf

int log_info(int cpu, const char *fmt, ...)
{
      FILE *log;
      char fn[MAX_FN_SIZE];
      char buf[MAX_BUF_SIZE];
      va_list args;

      sprintf(fn, "%d.log", cpu);
      log=fopen(fn, "a+");
      if (log==NULL) {
              perror("Error open:");
              return -1;
      }

      va_start(args, fmt);
      vprintf(fmt, args);
      memset(buf, 0, MAX_BUF_SIZE);
      vsprintf(buf, fmt, args);
      va_end(args);
```

(continues on next page)

(continued from previous page)

```
      fwrite(buf, sizeof(buf), 1, log);
      fclose(log);

      return 0;
}

typedef unsigned long u64;
typedef unsigned int  u32;

typedef union err_type_info_u {
      struct {
              u64     mode            : 3,    /* 0-2 */
                      err_inj         : 3,    /* 3-5 */
                      err_sev         : 2,    /* 6-7 */
                      err_struct      : 5,    /* 8-12 */
                      struct_hier     : 3,    /* 13-15 */
                      reserved        : 48;   /* 16-63 */
      } err_type_info_u;
      u64     err_type_info;
} err_type_info_t;

typedef union err_struct_info_u {
      struct {
              u64     siv             : 1,    /* 0     */
                      c_t             : 2,    /* 1-2   */
                      cl_p            : 3,    /* 3-5   */
                      cl_id           : 3,    /* 6-8   */
                      cl_dp           : 1,    /* 9     */
                      reserved1       : 22,   /* 10-31 */
                      tiv             : 1,    /* 32    */
                      trigger         : 4,    /* 33-36 */
                      trigger_pl      : 3,    /* 37-39 */
                      reserved2       : 24;   /* 40-63 */
      } err_struct_info_cache;
      struct {
              u64     siv             : 1,    /* 0     */
                      tt              : 2,    /* 1-2   */
                      tc_tr           : 2,    /* 3-4   */
                      tr_slot         : 8,    /* 5-12  */
                      reserved1       : 19,   /* 13-31 */
                      tiv             : 1,    /* 32    */
                      trigger         : 4,    /* 33-36 */
                      trigger_pl      : 3,    /* 37-39 */
                      reserved2       : 24;   /* 40-63 */
      } err_struct_info_tlb;
      struct {
              u64     siv             : 1,    /* 0     */
                      regfile_id      : 4,    /* 1-4   */
                      reg_num         : 7,    /* 5-11  */
```

(continues on next page)

(continued from previous page)

```
                      reserved1       : 20,   /* 12-31 */
                      tiv             : 1,    /* 32    */
                      trigger         : 4,    /* 33-36 */
                      trigger_pl      : 3,    /* 37-39 */
                      reserved2       : 24;   /* 40-63 */
      } err_struct_info_register;
      struct {
              u64     reserved;
      } err_struct_info_bus_processor_interconnect;
      u64     err_struct_info;
} err_struct_info_t;

typedef union err_data_buffer_u {
      struct {
              u64     trigger_addr;           /* 0-63         */
              u64     inj_addr;               /* 64-127       */
              u64     way             : 5,    /* 128-132      */
                      index           : 20,   /* 133-152      */
                                      : 39;   /* 153-191      */
      } err_data_buffer_cache;
      struct {
              u64     trigger_addr;           /* 0-63         */
              u64     inj_addr;               /* 64-127       */
              u64     way             : 5,    /* 128-132      */
                      index           : 20,   /* 133-152      */
                      reserved        : 39;   /* 153-191      */
      } err_data_buffer_tlb;
      struct {
              u64     trigger_addr;           /* 0-63         */
      } err_data_buffer_register;
      struct {
              u64     reserved;               /* 0-63         */
      } err_data_buffer_bus_processor_interconnect;
      u64 err_data_buffer[ERR_DATA_BUFFER_SIZE];
} err_data_buffer_t;

typedef union capabilities_u {
      struct {
              u64     i               : 1,
                      d               : 1,
                      rv              : 1,
                      tag             : 1,
                      data            : 1,
                      mesi            : 1,
                      dp              : 1,
                      reserved1       : 3,
                      pa              : 1,
                      va              : 1,
                      wi              : 1,
```

(continues on next page)

(continued from previous page)

```
                      reserved2       : 20,
                      trigger         : 1,
                      trigger_pl      : 1,
                      reserved3       : 30;
      } capabilities_cache;
      struct {
              u64     d               : 1,
                      i               : 1,
                      rv              : 1,
                      tc              : 1,
                      tr              : 1,
                      reserved1       : 27,
                      trigger         : 1,
                      trigger_pl      : 1,
                      reserved2       : 30;
      } capabilities_tlb;
      struct {
              u64     gr_b0           : 1,
                      gr_b1           : 1,
                      fr              : 1,
                      br              : 1,
                      pr              : 1,
                      ar              : 1,
                      cr              : 1,
                      rr              : 1,
                      pkr             : 1,
                      dbr             : 1,
                      ibr             : 1,
                      pmc             : 1,
                      pmd             : 1,
                      reserved1       : 3,
                      regnum          : 1,
                      reserved2       : 15,
                      trigger         : 1,
                      trigger_pl      : 1,
                      reserved3       : 30;
      } capabilities_register;
      struct {
              u64     reserved;
      } capabilities_bus_processor_interconnect;
} capabilities_t;

typedef struct resources_s {
      u64     ibr0            : 1,
              ibr2            : 1,
              ibr4            : 1,
              ibr6            : 1,
              dbr0            : 1,
              dbr2            : 1,
```

(continues on next page)

(continued from previous page)

```
              dbr4            : 1,
              dbr6            : 1,
              reserved        : 48;
} resources_t;


long get_page_size(void)
{
      long page_size=sysconf(_SC_PAGESIZE);
      return page_size;
}

#define PAGE_SIZE (get_page_size()==-1?0x4000:get_page_size())
#define SHM_SIZE (2*PAGE_SIZE*NR_CPUS)
#define SHM_VA 0x2000000100000000

int shmid;
void *shmaddr;

int create_shm(void)
{
      key_t key;
      char fn[MAX_FN_SIZE];

      /* cpu0 is always existing */
      sprintf(fn, PATH_FORMAT, 0);
      if ((key = ftok(fn, 's')) == -1) {
              perror("ftok");
              return -1;
      }

      shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT);
      if (shmid == -1) {
              if (errno==EEXIST) {
                      shmid = shmget(key, SHM_SIZE, 0);
                      if (shmid == -1) {
                              perror("shmget");
                              return -1;
                      }
              }
              else {
                      perror("shmget");
                      return -1;
              }
      }
      vbprintf("shmid=%d", shmid);

      /* connect to the segment: */
      shmaddr = shmat(shmid, (void *)SHM_VA, 0);
```

(continues on next page)

(continued from previous page)

```
        if (shmaddr == (void*)-1) {
                perror("shmat");
                return -1;
        }

        memset(shmaddr, 0, SHM_SIZE);
        mlock(shmaddr, SHM_SIZE);

        return 0;
}

int free_shm()
{
        munlock(shmaddr, SHM_SIZE);
          shmdt(shmaddr);
        semctl(shmid, 0, IPC_RMID);

        return 0;
}

#ifdef _SEM_SEMUN_UNDEFINED
union semun
{
        int val;
        struct semid_ds *buf;
        unsigned short int *array;
        struct seminfo *__buf;
};
#endif

u32 mode=1; /* 1: physical mode; 2: virtual mode. */
int one_lock=1;
key_t key[NR_CPUS];
int semid[NR_CPUS];

int create_sem(int cpu)
{
        union semun arg;
        char fn[MAX_FN_SIZE];
        int sid;

        sprintf(fn, PATH_FORMAT, cpu);
        sprintf(fn, "%s/%s", fn, "err_type_info");
        if ((key[cpu] = ftok(fn, 'e')) == -1) {
                perror("ftok");
                return -1;
        }

        if (semid[cpu]!=0)
```

(continues on next page)

(continued from previous page)

```
              return 0;

      /* clear old semaphore */
      if ((sid = semget(key[cpu], 1, 0)) != -1)
              semctl(sid, 0, IPC_RMID);

      /* get one semaphore */
      if ((semid[cpu] = semget(key[cpu], 1, IPC_CREAT | IPC_EXCL))␣
↪== -1) {
              perror("semget");
              printf("Please remove semaphore with key=0x%lx, then␣
↪run the tool.\n",
                      (u64)key[cpu]);
              return -1;
      }

      vbprintf("semid[%d]=0x%lx, key[%d]=%lx\n",cpu,(u64)semid[cpu],
↪cpu,
              (u64)key[cpu]);
      /* initialize the semaphore to 1: */
      arg.val = 1;
      if (semctl(semid[cpu], 0, SETVAL, arg) == -1) {
              perror("semctl");
              return -1;
      }

      return 0;
}

static int lock(int cpu)
{
      struct sembuf lock;

      lock.sem_num = cpu;
      lock.sem_op = 1;
      semop(semid[cpu], &lock, 1);

        return 0;
}

static int unlock(int cpu)
{
      struct sembuf unlock;

      unlock.sem_num = cpu;
      unlock.sem_op = -1;
      semop(semid[cpu], &unlock, 1);

        return 0;
```

(continues on next page)

(continued from previous page)

```
}

void free_sem(int cpu)
{
      semctl(semid[cpu], 0, IPC_RMID);
}

int wr_multi(char *fn, unsigned long *data, int size)
{
      int fd;
      char buf[MAX_BUF_SIZE];
      int ret;

      if (size==1)
              sprintf(buf, "%lx", *data);
      else if (size==3)
              sprintf(buf, "%lx,%lx,%lx", data[0], data[1],␣
↪data[2]);
      else {
              fprintf(stderr,"write to file with wrong size!\n");
              return -1;
      }

      fd=open(fn, O_RDWR);
      if (!fd) {
              perror("Error:");
              return -1;
      }
      ret=write(fd, buf, sizeof(buf));
      close(fd);
      return ret;
}

int wr(char *fn, unsigned long data)
{
      return wr_multi(fn, &data, 1);
}

int rd(char *fn, unsigned long *data)
{
      int fd;
      char buf[MAX_BUF_SIZE];

      fd=open(fn, O_RDONLY);
      if (fd<0) {
              perror("Error:");
              return -1;
      }
      read(fd, buf, MAX_BUF_SIZE);
```

(continues on next page)

(continued from previous page)

```
        *data=strtoul(buf, NULL, 16);
        close(fd);
        return 0;
}

int rd_status(char *path, int *status)
{
        char fn[MAX_FN_SIZE];
        sprintf(fn, "%s/status", path);
        if (rd(fn, (u64*)status)<0) {
                perror("status reading error.\n");
                return -1;
        }

        return 0;
}

int rd_capabilities(char *path, u64 *capabilities)
{
        char fn[MAX_FN_SIZE];
        sprintf(fn, "%s/capabilities", path);
        if (rd(fn, capabilities)<0) {
                perror("capabilities reading error.\n");
                return -1;
        }

        return 0;
}

int rd_all(char *path)
{
        unsigned long err_type_info, err_struct_info, err_data_buffer;
        int status;
        unsigned long capabilities, resources;
        char fn[MAX_FN_SIZE];

        sprintf(fn, "%s/err_type_info", path);
        if (rd(fn, &err_type_info)<0) {
                perror("err_type_info reading error.\n");
                return -1;
        }
        printf("err_type_info=%lx\n", err_type_info);

        sprintf(fn, "%s/err_struct_info", path);
        if (rd(fn, &err_struct_info)<0) {
                perror("err_struct_info reading error.\n");
                return -1;
        }
        printf("err_struct_info=%lx\n", err_struct_info);
```

(continues on next page)

(continued from previous page)

```
      sprintf(fn, "%s/err_data_buffer", path);
      if (rd(fn, &err_data_buffer)<0) {
              perror("err_data_buffer reading error.\n");
              return -1;
      }
      printf("err_data_buffer=%lx\n", err_data_buffer);

      sprintf(fn, "%s/status", path);
      if (rd("status", (u64*)&status)<0) {
              perror("status reading error.\n");
              return -1;
      }
      printf("status=%d\n", status);

      sprintf(fn, "%s/capabilities", path);
      if (rd(fn,&capabilities)<0) {
              perror("capabilities reading error.\n");
              return -1;
      }
      printf("capabilities=%lx\n", capabilities);

      sprintf(fn, "%s/resources", path);
      if (rd(fn, &resources)<0) {
              perror("resources reading error.\n");
              return -1;
      }
      printf("resources=%lx\n", resources);

      return 0;
}

int query_capabilities(char *path, err_type_info_t err_type_info,
                      u64 *capabilities)
{
      char fn[MAX_FN_SIZE];
      err_struct_info_t err_struct_info;
      err_data_buffer_t err_data_buffer;

      err_struct_info.err_struct_info=0;
      memset(err_data_buffer.err_data_buffer, -1, ERR_DATA_BUFFER_
 ↪SIZE*8);

      sprintf(fn, "%s/err_type_info", path);
      wr(fn, err_type_info.err_type_info);
      sprintf(fn, "%s/err_struct_info", path);
      wr(fn, 0x0);
      sprintf(fn, "%s/err_data_buffer", path);
      wr_multi(fn, err_data_buffer.err_data_buffer, ERR_DATA_BUFFER_
```

(continues on next page)

(continued from previous page)

```
 ↪SIZE);

      // Fire pal_mc_error_inject procedure.
      sprintf(fn, "%s/call_start", path);
      wr(fn, mode);

      if (rd_capabilities(path, capabilities)<0)
              return -1;

      return 0;
}

int query_all_capabilities()
{
      int status;
      err_type_info_t err_type_info;
      int err_sev, err_struct, struct_hier;
      int cap=0;
      u64 capabilities;
      char path[MAX_FN_SIZE];

      err_type_info.err_type_info=0;                  // Initial
      err_type_info.err_type_info_u.mode=0;           // Query mode;
      err_type_info.err_type_info_u.err_inj=0;

      printf("All capabilities implemented in pal_mc_error_inject:\n
 ↪");
      sprintf(path, PATH_FORMAT ,0);
      for (err_sev=0;err_sev<3;err_sev++)
              for (err_struct=0;err_struct<5;err_struct++)
                      for (struct_hier=0;struct_hier<5;struct_
 ↪hier++)
      {
              status=-1;
              capabilities=0;
              err_type_info.err_type_info_u.err_sev=err_sev;
              err_type_info.err_type_info_u.err_struct=err_struct;
              err_type_info.err_type_info_u.struct_hier=struct_hier;

              if (query_capabilities(path, err_type_info, &
 ↪capabilities)<0)
                      continue;

              if (rd_status(path, &status)<0)
                      continue;

              if (status==0) {
                      cap=1;
                      printf("For err_sev=%d, err_struct=%d, struct_
```

(continues on next page)

(continued from previous page)

```
↪hier=%d: ",
                              err_sev, err_struct, struct_hier);
                      printf("capabilities 0x%lx\n", capabilities);
              }
      }
      if (!cap) {
              printf("No capabilities supported.\n");
              return 0;
      }

      return 0;
}

int err_inject(int cpu, char *path, err_type_info_t err_type_info,
              err_struct_info_t err_struct_info,
              err_data_buffer_t err_data_buffer)
{
      int status;
      char fn[MAX_FN_SIZE];

      log_info(cpu, "err_type_info=%lx, err_struct_info=%lx, ",
              err_type_info.err_type_info,
              err_struct_info.err_struct_info);
      log_info(cpu,"err_data_buffer=[%lx,%lx,%lx]\n",
              err_data_buffer.err_data_buffer[0],
              err_data_buffer.err_data_buffer[1],
              err_data_buffer.err_data_buffer[2]);
      sprintf(fn, "%s/err_type_info", path);
      wr(fn, err_type_info.err_type_info);
      sprintf(fn, "%s/err_struct_info", path);
      wr(fn, err_struct_info.err_struct_info);
      sprintf(fn, "%s/err_data_buffer", path);
      wr_multi(fn, err_data_buffer.err_data_buffer, ERR_DATA_BUFFER_
↪SIZE);

      // Fire pal_mc_error_inject procedure.
      sprintf(fn, "%s/call_start", path);
      wr(fn,mode);

      if (rd_status(path, &status)<0) {
              vbprintf("fail: read status\n");
              return -100;
      }

      if (status!=0) {
              log_info(cpu, "fail: status=%d\n", status);
              return status;
      }
```

(continues on next page)

(continued from previous page)

```
      return status;
}

static int construct_data_buf(char *path, err_type_info_t err_type_
↪info,
              err_struct_info_t err_struct_info,
              err_data_buffer_t *err_data_buffer,
              void *va1)
{
      char fn[MAX_FN_SIZE];
      u64 virt_addr=0, phys_addr=0;

      vbprintf("va1=%lx\n", (u64)va1);
      memset(&err_data_buffer->err_data_buffer_cache, 0, ERR_DATA_
↪BUFFER_SIZE*8);

      switch (err_type_info.err_type_info_u.err_struct) {
              case 1: // Cache
                      switch (err_struct_info.err_struct_info_cache.
↪cl_id) {
                              case 1: //Virtual addr
                                      err_data_buffer->err_data_
↪buffer_cache.inj_addr=(u64)va1;
                                      break;
                              case 2: //Phys addr
                                      sprintf(fn, "%s/virtual_to_
↪phys", path);
                                      virt_addr=(u64)va1;
                                      if (wr(fn,virt_addr)<0)
                                              return -1;
                                      rd(fn, &phys_addr);
                                      err_data_buffer->err_data_
↪buffer_cache.inj_addr=phys_addr;
                                      break;
                              default:
                                      printf("Not supported cl_id\n
↪");
                                      break;
                      }
                      break;
              case 2: //  TLB
                      break;
              case 3: //  Register file
                      break;
              case 4: //  Bus/system interconnect
              default:
                      printf("Not supported err_struct\n");
                      break;
      }
```

(continues on next page)

(continued from previous page)

```
      return 0;
}

typedef struct {
      u64 cpu;
      u64 loop;
      u64 interval;
      u64 err_type_info;
      u64 err_struct_info;
      u64 err_data_buffer[ERR_DATA_BUFFER_SIZE];
} parameters_t;

parameters_t line_para;
int para;

static int empty_data_buffer(u64 *err_data_buffer)
{
      int empty=1;
      int i;

      for (i=0;i<ERR_DATA_BUFFER_SIZE; i++)
         if (err_data_buffer[i]!=-1)
              empty=0;

      return empty;
}

int err_inj()
{
      err_type_info_t err_type_info;
      err_struct_info_t err_struct_info;
      err_data_buffer_t err_data_buffer;
      int count;
      FILE *fp;
      unsigned long cpu, loop, interval, err_type_info_conf, err_
 ↪struct_info_conf;
      u64 err_data_buffer_conf[ERR_DATA_BUFFER_SIZE];
      int num;
      int i;
      char path[MAX_FN_SIZE];
      parameters_t parameters[MAX_TASK_NUM]={};
      pid_t child_pid[MAX_TASK_NUM];
      time_t current_time;
      int status;

      if (!para) {
          fp=fopen("err.conf", "r");
          if (fp==NULL) {
```

(continues on next page)

(continued from previous page)

```
            perror("Error open err.conf");
            return -1;
        }

        num=0;
        while (!feof(fp)) {
            char buf[256];
            memset(buf,0,256);
            fgets(buf, 256, fp);
            count=sscanf(buf, "%lx, %lx, %lx, %lx, %lx, %lx, %lx,
↪%lx\n",
                            &cpu, &loop, &interval,&err_type_info_
↪conf,
                            &err_struct_info_conf,
                            &err_data_buffer_conf[0],
                            &err_data_buffer_conf[1],
                            &err_data_buffer_conf[2]);
            if (count!=PARA_FIELD_NUM+3) {
                    err_data_buffer_conf[0]=-1;
                    err_data_buffer_conf[1]=-1;
                    err_data_buffer_conf[2]=-1;
                    count=sscanf(buf, "%lx, %lx, %lx, %lx, %lx\n",
                            &cpu, &loop, &interval,&err_type_info_
↪conf,
                            &err_struct_info_conf);
                    if (count!=PARA_FIELD_NUM)
                            continue;
            }

            parameters[num].cpu=cpu;
            parameters[num].loop=loop;
            parameters[num].interval= interval>MIN_INTERVAL
                                      ?interval:MIN_INTERVAL;
            parameters[num].err_type_info=err_type_info_conf;
            parameters[num].err_struct_info=err_struct_info_conf;
            memcpy(parameters[num++].err_data_buffer,
                    err_data_buffer_conf,ERR_DATA_BUFFER_SIZE*8) ;

            if (num>=MAX_TASK_NUM)
                    break;
        }
    }
    else {
            parameters[0].cpu=line_para.cpu;
            parameters[0].loop=line_para.loop;
            parameters[0].interval= line_para.interval>MIN_
↪INTERVAL
                                      ?line_para.interval:MIN_
↪INTERVAL;
```

(continues on next page)

(continued from previous page)

```
             parameters[0].err_type_info=line_para.err_type_info;
             parameters[0].err_struct_info=line_para.err_struct_
↪info;
             memcpy(parameters[0].err_data_buffer,
                     line_para.err_data_buffer,ERR_DATA_BUFFER_
↪SIZE*8) ;

             num=1;
     }

     /* Create semaphore: If one_lock, one semaphore for all␣
↪processors.
        Otherwise, one semaphore for each processor. */
     if (one_lock) {
             if (create_sem(0)) {
                     printf("Can not create semaphore...exit\n");
                     free_sem(0);
                     return -1;
             }
     }
     else {
             for (i=0;i<num;i++) {
                if (create_sem(parameters[i].cpu)) {
                     printf("Can not create semaphore for cpu%d...
↪exit\n",i);
                     free_sem(parameters[num].cpu);
                     return -1;
                }
             }
     }

     /* Create a shm segment which will be used to inject/consume␣
↪errors on.*/
     if (create_shm()==-1) {
             printf("Error to create shm...exit\n");
             return -1;
     }

     for (i=0;i<num;i++) {
             pid_t pid;

             current_time=time(NULL);
             log_info(parameters[i].cpu, "\nBegine at %s", ctime(&
↪current_time));
             log_info(parameters[i].cpu, "Configurations:\n");
             log_info(parameters[i].cpu,"On cpu%ld: loop=%lx,␣
↪interval=%lx(s)",
                     parameters[i].cpu,
                     parameters[i].loop,
```

(continues on next page)

(continued from previous page)

```
                     parameters[i].interval);
             log_info(parameters[i].cpu," err_type_info=%lx,err_
↪struct_info=%lx\n",
                     parameters[i].err_type_info,
                     parameters[i].err_struct_info);

             sprintf(path, PATH_FORMAT, (int)parameters[i].cpu);
             err_type_info.err_type_info=parameters[i].err_type_
↪info;
             err_struct_info.err_struct_info=parameters[i].err_
↪struct_info;
             memcpy(err_data_buffer.err_data_buffer,
                     parameters[i].err_data_buffer,
                     ERR_DATA_BUFFER_SIZE*8);

             pid=fork();
             if (pid==0) {
                     unsigned long mask[MASK_SIZE];
                     int j, k;

                     void *va1, *va2;

                     /* Allocate two memory areas va1 and va2 in␣
↪shm */
                     va1=shmaddr+parameters[i].cpu*PAGE_SIZE;
                     va2=shmaddr+parameters[i].cpu*PAGE_SIZE+PAGE_
↪SIZE;

                     vbprintf("va1=%lx, va2=%lx\n", (u64)va1,␣
↪(u64)va2);
                     memset(va1, 0x1, PAGE_SIZE);
                     memset(va2, 0x2, PAGE_SIZE);

                     if (empty_data_buffer(err_data_buffer.err_
↪data_buffer))
                             /* If not specified yet, construct␣
↪data buffer
                              * with va1
                              */
                             construct_data_buf(path, err_type_
↪info,
                                     err_struct_info, &err_data_
↪buffer,va1);

                     for (j=0;j<MASK_SIZE;j++)
                             mask[j]=0;

                     cpu=parameters[i].cpu;
                     k = cpu%64;
```

(continues on next page)

(continued from previous page)

```
                     j = cpu/64;
                     mask[j] = 1UL << k;

                     if (sched_setaffinity(0, MASK_SIZE*8, mask)==-
↪1) {
                             perror("Error sched_setaffinity:");
                             return -1;
                     }

                     for (j=0; j<parameters[i].loop; j++) {
                             log_info(parameters[i].cpu,"Injection
↪");
                             log_info(parameters[i].cpu,"on cpu
↪%ld: #%d/%ld ",

                                     parameters[i].cpu,j+1,␣
↪parameters[i].loop);

                             /* Hold the lock */
                             if (one_lock)
                                     lock(0);
                             else
                             /* Hold lock on this cpu */
                                     lock(parameters[i].cpu);

                             if ((status=err_inject(parameters[i].
↪cpu,
                                        path, err_type_info,
                                        err_struct_info, err_data_
↪buffer))
                                        ==0) {
                                     /* consume the error for
↪"inject only"*/
                                     memcpy(va2, va1, PAGE_SIZE);
                                     memcpy(va1, va2, PAGE_SIZE);
                                     log_info(parameters[i].cpu,
                                             "successful\n");
                             }
                             else {
                                     log_info(parameters[i].cpu,
↪"fail:");
                                     log_info(parameters[i].cpu,
                                             "status=%d\n",␣
↪status);
                                     unlock(parameters[i].cpu);
                                     break;
                             }
                             if (one_lock)
                             /* Release the lock */
```

(continues on next page)

(continued from previous page)

```
                                        unlock(0);
                                /* Release lock on this cpu */
                                else
                                        unlock(parameters[i].cpu);

                                if (j < parameters[i].loop-1)
                                        sleep(parameters[i].interval);
                        }
                        current_time=time(NULL);
                        log_info(parameters[i].cpu, "Done at %s",␣
↪ctime(&current_time));
                        return 0;
                }
                else if (pid<0) {
                        perror("Error fork:");
                        continue;
                }
                child_pid[i]=pid;
        }
        for (i=0;i<num;i++)
                waitpid(child_pid[i], NULL, 0);

        if (one_lock)
                free_sem(0);
        else
                for (i=0;i<num;i++)
                        free_sem(parameters[i].cpu);

        printf("All done.\n");

        return 0;
}

void help()
{
        printf("err_inject_tool:\n");
        printf("\t-q: query all capabilities. default: off\n");
        printf("\t-m: procedure mode. 1: physical 2: virtual.␣
↪default: 1\n");
        printf("\t-i: inject errors. default: off\n");
        printf("\t-l: one lock per cpu. default: one lock for all\n");
        printf("\t-e: error parameters:\n");
        printf("\t\tcpu,loop,interval,err_type_info,err_struct_info[,
↪err_data_buffer[0],err_data_buffer[1],err_data_buffer[2]]\n");
        printf("\t\t   cpu: logical cpu number the error will be␣
↪inject in.\n");
        printf("\t\t   loop: times the error will be injected.\n");
        printf("\t\t   interval: In second. every so often one error␣
↪is injected.\n");
```

(continues on next page)

(continued from previous page)

```
      printf("\t\t   err_type_info, err_struct_info: PAL parameters.
↪\n");
      printf("\t\t   err_data_buffer: PAL parameter. Optional. If␣
↪not present,\n");
      printf("\t\t                    it's constructed by tool␣
↪automatically. Be\n");
      printf("\t\t                    careful to provide err_data_
↪buffer and make\n");
      printf("\t\t                    sure it's working with the␣
↪environment.\n");
      printf("\t    Note:no space between error parameters.\n");
      printf("\t    default: Take error parameters from err.conf␣
↪instead of command line.\n");
      printf("\t-v: verbose. default: off\n");
      printf("\t-h: help\n\n");
      printf("The tool will take err.conf file as ");
      printf("input to inject single or multiple errors ");
      printf("on one or multiple cpus in parallel.\n");
}

int main(int argc, char **argv)
{
      char c;
      int do_err_inj=0;
      int do_query_all=0;
      int count;
      u32 m;

      /* Default one lock for all cpu's */
      one_lock=1;
      while ((c = getopt(argc, argv, "m:iqvhle:")) != EOF)
              switch (c) {
                      case 'm':       /* Procedure mode. 1: phys 2:␣
↪virt */
                              count=sscanf(optarg, "%x", &m);
                              if (count!=1 || (m!=1 && m!=2)) {
                                      printf("Wrong mode number.\n
↪");

                                      help();
                                      return -1;
                              }
                              mode=m;
                              break;
                      case 'i':       /* Inject errors */
                              do_err_inj=1;
                              break;
                      case 'q':       /* Query */
                              do_query_all=1;
                              break;
```

(continues on next page)

(continued from previous page)

```
                     case 'v':       /* Verbose */
                             verbose=1;
                             break;
                     case 'l':       /* One lock per cpu */
                             one_lock=0;
                             break;
                     case 'e':       /* error arguments */
                             /* Take parameters:
                              * #cpu, loop, interval, err_type_
↪info, err_struct_info[, err_data_buffer]
                              * err_data_buffer is optional.␣
↪Recommend not to specify
                              * err_data_buffer. Better to use␣
↪tool to generate it.
                              */
                             count=sscanf(optarg,
                                     "%lx, %lx, %lx, %lx, %lx, %lx,
↪ %lx, %lx\n",
                                     &line_para.cpu,
                                     &line_para.loop,
                                     &line_para.interval,
                                     &line_para.err_type_info,
                                     &line_para.err_struct_info,
                                     &line_para.err_data_buffer[0],
                                     &line_para.err_data_buffer[1],
                                     &line_para.err_data_
↪buffer[2]);
                             if (count!=PARA_FIELD_NUM+3) {
                                 line_para.err_data_buffer[0]=-1,
                                 line_para.err_data_buffer[1]=-1,
                                 line_para.err_data_buffer[2]=-1;
                                 count=sscanf(optarg, "%lx, %lx,
↪%lx, %lx, %lx\n",
                                             &line_para.cpu,
                                             &line_para.loop,
                                             &line_para.interval,
                                             &line_para.err_type_
↪info,
                                             &line_para.err_struct_
↪info);
                                 if (count!=PARA_FIELD_NUM) {
                                     printf("Wrong error arguments.
↪\n");
                                     help();
                                     return -1;
                                 }
                             }
                             para=1;
                             break;
```

(continues on next page)

(continued from previous page)

```
                      continue;
                              break;
                      case 'h':
                              help();
                              return 0;
                      default:
                              break;
              }

      if (do_query_all)
              query_all_capabilities();
      if (do_err_inj)
              err_inj();

      if (!do_query_all &&  !do_err_inj)
              help();

      return 0;
}
```

CHAPTER

FIVE

LIGHT-WEIGHT SYSTEM CALLS FOR IA-64

Started: 13-Jan-2003

Last update: 27-Sep-2003

David Mosberger-Tang <davidm@hpl.hp.com>

Using the “epc” instruction effectively introduces a new mode of execution to the ia64 linux kernel. We call this mode the “fsys-mode” . To recap, the normal states of execution are:

- **kernel mode:**
  Both the register stack and the memory stack have been switched over to kernel memory. The user-level state is saved in a pt-regs structure at the top of the kernel memory stack.
- **user mode:**
  Both the register stack and the kernel stack are in user memory. The user-level state is contained in the CPU registers.
- **bank 0 interruption-handling mode:**
  This is the non-interruptible state which all interruption-handlers start execution in. The user-level state remains in the CPU registers and some kernel state may be stored in bank 0 of registers r16-r31.

In contrast, fsys-mode has the following special properties:

- execution is at privilege level 0 (most-privileged)
- CPU registers may contain a mixture of user-level and kernel-level state (it is the responsibility of the kernel to ensure that no security-sensitive kernel-level state is leaked back to user-level)
- execution is interruptible and preemptible (an fsys-mode handler can disable interrupts and avoid all other interruption-sources to avoid preemption)
- neither the memory-stack nor the register-stack can be trusted while in fsys-mode (they point to the user-level stacks, which may be invalid, or completely bogus addresses)

In summary, fsys-mode is much more similar to running in user-mode than it is to running in kernel-mode. Of course, given that the privilege level is at level 0, this means that fsys-mode requires some care (see below).

5.1 How to tell fsys-mode

Linux operates in fsys-mode when (a) the privilege level is 0 (most privileged) and (b) the stacks have NOT been switched to kernel memory yet. For convenience, the header file <asm-ia64/ptrace.h> provides three macros:

```
user_mode(regs)
user_stack(task,regs)
fsys_mode(task,regs)
```

The "regs" argument is a pointer to a pt_regs structure. The "task" argument is a pointer to the task structure to which the "regs" pointer belongs to. user_mode() returns TRUE if the CPU state pointed to by "regs" was executing in user mode (privilege level 3). user_stack() returns TRUE if the state pointed to by "regs" was executing on the user-level stack(s). Finally, fsys_mode() returns TRUE if the CPU state pointed to by "regs" was executing in fsys-mode. The fsys_mode() macro is equivalent to the expression:

```
!user_mode(regs) && user_stack(task,regs)
```

5.2 How to write an fsyscall handler

The file arch/ia64/kernel/fsys.S contains a table of fsyscall-handlers (fsyscall_table). This table contains one entry for each system call. By default, a system call is handled by fsys_fallback_syscall(). This routine takes care of entering (full) kernel mode and calling the normal Linux system call handler. For performance-critical system calls, it is possible to write a hand-tuned fsyscall_handler. For example, fsys.S contains fsys_getpid(), which is a hand-tuned version of the getpid() system call.

The entry and exit-state of an fsyscall handler is as follows:

5.2.1 Machine state on entry to fsyscall handler

r10	0
r11	saved ar.pfs (a user-level value)
r15	system call number
r16	"current" task pointer (in normal kernel-mode, this is in r13)
r32-r39	system call arguments
b6	return address (a user-level value)
ar.pfs	previous frame-state (a user-level value)
PSR.be	cleared to zero (i.e., little-endian byte order is in effect)
•	all other registers may contain values passed in from user-mode

5.2.2 Required machine state on exit to fsyscall handler

r11	saved ar.pfs (as passed into the fsyscall handler)
r15	system call number (as passed into the fsyscall handler)
r32-r39	system call arguments (as passed into the fsyscall handler)
b6	return address (as passed into the fsyscall handler)
ar.pfs	previous frame-state (as passed into the fsyscall handler)

Fsyscall handlers can execute with very little overhead, but with that speed comes a set of restrictions:

- Fsyscall-handlers MUST check for any pending work in the flags member of the thread-info structure and if any of the TIF_ALLWORK_MASK flags are set, the handler needs to fall back on doing a full system call (by calling fsys_fallback_syscall).
- Fsyscall-handlers MUST preserve incoming arguments (r32-r39, r11, r15, b6, and ar.pfs) because they will be needed in case of a system call restart. Of course, all "preserved" registers also must be preserved, in accordance to the normal calling conventions.
- Fsyscall-handlers MUST check argument registers for containing a NaT value before using them in any way that could trigger a NaT-consumption fault. If a system call argument is found to contain a NaT value, an fsyscall-handler may return immediately with r8=EINVAL, r10=-1.
- Fsyscall-handlers MUST NOT use the "alloc" instruction or perform any other operation that would trigger mandatory RSE (register-stack engine) traffic.
- Fsyscall-handlers MUST NOT write to any stacked registers because it is not safe to assume that user-level called a handler with the proper number of arguments.
- Fsyscall-handlers need to be careful when accessing per-CPU variables: unless proper safe-guards are taken (e.g., interruptions are avoided), execution may be pre-empted and resumed on another CPU at any given time.
- Fsyscall-handlers must be careful not to leak sensitive kernel' information back to user-level. In particular, before returning to user-level, care needs to be taken to clear any scratch registers that could contain sensitive information (note that the current task pointer is not considered sensitive: it' s already exposed through ar.k6).
- Fsyscall-handlers MUST NOT access user-memory without first validating access-permission (this can be done typically via probe.r.fault and/or probe.w.fault) and without guarding against memory access exceptions (this can be done with the EX() macros defined by asmmacro.h).

The above restrictions may seem draconian, but remember that it' s possible to trade off some of the restrictions by paying a slightly higher overhead. For example, if an fsyscall-handler could benefit from the shadow register bank, it could temporarily disable PSR.i and PSR.ic, switch to bank 0 (bsw.0) and then use the shadow registers as needed. In other words, following the above rules yields extremely fast system call execution (while fully preserving system call semantics), but there is also a lot of flexibility in handling more complicated cases.

5.3 Signal handling

The delivery of (asynchronous) signals must be delayed until fsys-mode is exited. This is accomplished with the help of the lower-privilege transfer trap: arch/ia64/kernel/process.c:do_notify_resume_user() checks whether the interrupted task was in fsys-mode and, if so, sets PSR.lp and returns immediately. When fsys-mode is exited via the “br.ret” instruction that lowers the privilege level, a trap will occur. The trap handler clears PSR.lp again and returns immediately. The kernel exit path then checks for and delivers any pending signals.

5.4 PSR Handling

The “epc” instruction doesn’ t change the contents of PSR at all. This is in contrast to a regular interruption, which clears almost all bits. Because of that, some care needs to be taken to ensure things work as expected. The following discussion describes how each PSR bit is handled.

PSR.be	Cleared when entering fsys-mode. A srlz.d instruction is used to ensure the CPU is
PSR.up	Unchanged.
PSR.ac	Unchanged.
PSR.mfl	Unchanged. Note: fsys-mode handlers must not write-registers!
PSR.mfh	Unchanged. Note: fsys-mode handlers must not write-registers!
PSR.ic	Unchanged. Note: fsys-mode handlers can clear the bit, if needed.
PSR.i	Unchanged. Note: fsys-mode handlers can clear the bit, if needed.
PSR.pk	Unchanged.
PSR.dt	Unchanged.
PSR.dfl	Unchanged. Note: fsys-mode handlers must not write-registers!
PSR.dfh	Unchanged. Note: fsys-mode handlers must not write-registers!
PSR.sp	Unchanged.
PSR.pp	Unchanged.
PSR.di	Unchanged.
PSR.si	Unchanged.
PSR.db	Unchanged. The kernel prevents user-level from setting a hardware breakpoint th
PSR.lp	Unchanged.
PSR.tb	Lazy redirect. If a taken-branch trap occurs while in fsys-mode, the trap-handler m
PSR.rt	Unchanged.
PSR.cpl	Cleared to 0.
PSR.is	Unchanged (guaranteed to be 0 on entry to the gate page).
PSR.mc	Unchanged.
PSR.it	Unchanged (guaranteed to be 1).
PSR.id	Unchanged. Note: the ia64 linux kernel never sets this bit.
PSR.da	Unchanged. Note: the ia64 linux kernel never sets this bit.
PSR.dd	Unchanged. Note: the ia64 linux kernel never sets this bit.
PSR.ss	Lazy redirect. If set, “epc” will cause a Single Step Trap to be taken. The trap ha
PSR.ri	Unchanged.

PSR.ed	Unchanged. Note: This bit could only have an effect if an fsys-mode handler perfor
PSR.bn	Unchanged. Note: fsys-mode handlers may clear the bit, if needed. Doing so requi
PSR.ia	Unchanged. Note: the ia64 linux kernel never sets this bit.

5.5 Using fast system calls

To use fast system calls, userspace applications need simply call __kernel_syscall_via_epc(). For example

– example fgettimeofday() call –

– fgettimeofday.S –

```
#include <asm/asmmacro.h>

GLOBAL_ENTRY(fgettimeofday)
.prologue
.save ar.pfs, r11
mov r11 = ar.pfs
.body

mov r2 = 0xa000000000020660;;  // gate address
                              // found by inspection of System.map␣
 ↪for the
                              // __kernel_syscall_via_epc() function.
 ↪  See
                              // below for how to do this for real.

mov b7 = r2
mov r15 = 1087                       // gettimeofday syscall
;;
br.call.sptk.many b6 = b7
;;

.restore sp

mov ar.pfs = r11
br.ret.sptk.many rp;;       // return to caller
END(fgettimeofday)
```

– end fgettimeofday.S –

In reality, getting the gate address is accomplished by two extra values passed via the ELF auxiliary vector (include/asm-ia64/elf.h)

- AT_SYSINFO : is the address of __kernel_syscall_via_epc()
- AT_SYSINFO_EHDR : is the address of the kernel gate ELF DSO

The ELF DSO is a pre-linked library that is mapped in by the kernel at the gate page. It is a proper ELF shared object so, with a dynamic loader that recognises the library, you should be able to make calls to the exported functions within it as with any other shared library. AT_SYSINFO points into the kernel DSO at the __kernel_syscall_via_epc() function for historical reasons (it was used before the kernel DSO) and as a convenience.

CHAPTER

SIX

IRQ AFFINITY ON IA64 PLATFORMS

07.01.2002, Erich Focht <efocht@ess.nec.de>

By writing to /proc/irq/IRQ#/smp_affinity the interrupt routing can be controlled. The behavior on IA64 platforms is slightly different from that described in Documentation/core-api/irq/irq-affinity.rst for i386 systems.

Because of the usage of SAPIC mode and physical destination mode the IRQ target is one particular CPU and cannot be a mask of several CPUs. Only the first non-zero bit is taken into account.

6.1 Usage examples

The target CPU has to be specified as a hexadecimal CPU mask. The first non-zero bit is the selected CPU. This format has been kept for compatibility reasons with i386.

Set the delivery mode of interrupt 41 to fixed and route the interrupts to CPU #3 (logical CPU number) (2^3=0x08):

```
echo "8" >/proc/irq/41/smp_affinity
```

Set the default route for IRQ number 41 to CPU 6 in lowest priority delivery mode (redirectable):

```
echo "r 40" >/proc/irq/41/smp_affinity
```

The output of the command:

```
cat /proc/irq/IRQ#/smp_affinity
```

gives the target CPU mask for the specified interrupt vector. If the CPU mask is preceded by the character "r", the interrupt is redirectable (i.e. lowest priority mode routing is used), otherwise its route is fixed.

6.2 Initialization and default behavior

If the platform features IRQ redirection (info provided by SAL) all IO-SAPIC interrupts are initialized with CPU#0 as their default target and the routing is the so called “lowest priority mode” (actually fixed SAPIC mode with hint). The XTP chipset registers are used as hints for the IRQ routing. Currently in Linux XTP registers can have three values:

- minimal for an idle task,
- normal if any other task runs,
- maximal if the CPU is going to be switched off.

The IRQ is routed to the CPU with lowest XTP register value, the search begins at the default CPU. Therefore most of the interrupts will be handled by CPU #0.

If the platform doesn’t feature interrupt redirection IOSAPIC fixed routing is used. The target CPUs are distributed in a round robin manner. IRQs will be routed only to the selected target CPUs. Check with:

```
cat /proc/interrupts
```

6.3 Comments

On large (multi-node) systems it is recommended to route the IRQs to the node to which the corresponding device is connected. For systems like the NEC AzusA we get IRQ node-affinity for free. This is because usually the chipsets on each node redirect the interrupts only to their own CPUs (as they cannot see the XTP registers on the other nodes).

CHAPTER SEVEN

AN AD-HOC COLLECTION OF NOTES ON IA64 MCA AND INIT PROCESSING

Feel free to update it with notes about any area that is not clear.

—

MCA/INIT are completely asynchronous. They can occur at any time, when the OS is in any state. Including when one of the cpus is already holding a spinlock. Trying to get any lock from MCA/INIT state is asking for deadlock. Also the state of structures that are protected by locks is indeterminate, including linked lists.

—

The complicated ia64 MCA process. All of this is mandated by Intel' s specification for ia64 SAL, error recovery and unwind, it is not as if we have a choice here.

- MCA occurs on one cpu, usually due to a double bit memory error. This is the monarch cpu.
- SAL sends an MCA rendezvous interrupt (which is a normal interrupt) to all the other cpus, the slaves.
- Slave cpus that receive the MCA interrupt call down into SAL, they end up spinning disabled while the MCA is being serviced.
- If any slave cpu was already spinning disabled when the MCA occurred then it cannot service the MCA interrupt. SAL waits ~20 seconds then sends an unmaskable INIT event to the slave cpus that have not already rendezvoused.
- Because MCA/INIT can be delivered at any time, including when the cpu is down in PAL in physical mode, the registers at the time of the event are _completely_ undefined. In particular the MCA/INIT handlers cannot rely on the thread pointer, PAL physical mode can (and does) modify TP. It is allowed to do that as long as it resets TP on return. However MCA/INIT events expose us to these PAL internal TP changes. Hence curr_task().
- If an MCA/INIT event occurs while the kernel was running (not user space) and the kernel has called PAL then the MCA/INIT handler cannot assume that the kernel stack is in a fit state to be used. Mainly because PAL may or may not maintain the stack pointer internally. Because the MCA/INIT handlers cannot trust the kernel stack, they have to use their own, per-cpu stacks. The MCA/INIT stacks are preformatted with just enough task state to let the relevant handlers do their job.

- Unlike most other architectures, the ia64 struct task is embedded in the kernel stack[1]. So switching to a new kernel stack means that we switch to a new task as well. Because various bits of the kernel assume that current points into the struct task, switching to a new stack also means a new value for current.
- Once all slaves have rendezvoused and are spinning disabled, the monarch is entered. The monarch now tries to diagnose the problem and decide if it can recover or not.
- Part of the monarch's job is to look at the state of all the other tasks. The only way to do that on ia64 is to call the unwinder, as mandated by Intel.
- The starting point for the unwind depends on whether a task is running or not. That is, whether it is on a cpu or is blocked. The monarch has to determine whether or not a task is on a cpu before it knows how to start unwinding it. The tasks that received an MCA or INIT event are no longer running, they have been converted to blocked tasks. But (and its a big but), the cpus that received the MCA rendezvous interrupt are still running on their normal kernel stacks!
- To distinguish between these two cases, the monarch must know which tasks are on a cpu and which are not. Hence each slave cpu that switches to an MCA/INIT stack, registers its new stack using set_curr_task(), so the monarch can tell that the _original_ task is no longer running on that cpu. That gives us a decent chance of getting a valid backtrace of the _original_ task.
- MCA/INIT can be nested, to a depth of 2 on any cpu. In the case of a nested error, we want diagnostics on the MCA/INIT handler that failed, not on the task that was originally running. Again this requires set_curr_task() so the MCA/INIT handlers can register their own stack as running on that cpu. Then a recursive error gets a trace of the failing handler's "task".

[1]

My (Keith Owens) original design called for ia64 to separate its struct task and the kernel stacks. Then the MCA/INIT data would be chained stacks like i386 interrupt stacks. But that required radical surgery on the rest of ia64, plus extra hard wired TLB entries with its associated performance degradation. David Mosberger vetoed that approach. Which meant that separate kernel stacks meant separate "tasks" for the MCA/INIT handlers.

—

INIT is less complicated than MCA. Pressing the nmi button or using the equivalent command on the management console sends INIT to all cpus. SAL picks one of the cpus as the monarch and the rest are slaves. All the OS INIT handlers are entered at approximately the same time. The OS monarch prints the state of all tasks and returns, after which the slaves return and the system resumes.

At least that is what is supposed to happen. Alas there are broken versions of SAL out there. Some drive all the cpus as monarchs. Some drive them all as slaves. Some drive one cpu as monarch, wait for that cpu to return from the OS then drive the rest as slaves. Some versions of SAL cannot even cope with returning from the OS, they spin inside SAL on resume. The OS INIT code has workarounds for some of these broken SAL symptoms, but some simply cannot be fixed from the OS side.

—

The scheduler hooks used by ia64 (curr_task, set_curr_task) are layer violations. Unfortunately MCA/INIT start off as massive layer violations (can occur at _any_ time) and they build from there.

At least ia64 makes an attempt at recovering from hardware errors, but it is a difficult problem because of the asynchronous nature of these errors. When processing an unmaskable interrupt we sometimes need special code to cope with our inability to take any locks.

—

How is ia64 MCA/INIT different from x86 NMI?

- x86 NMI typically gets delivered to one cpu. MCA/INIT gets sent to all cpus.
- x86 NMI cannot be nested. MCA/INIT can be nested, to a depth of 2 per cpu.
- x86 has a separate struct task which points to one of multiple kernel stacks. ia64 has the struct task embedded in the single kernel stack, so switching stack means switching task.
- x86 does not call the BIOS so the NMI handler does not have to worry about any registers having changed. MCA/INIT can occur while the cpu is in PAL in physical mode, with undefined registers and an undefined kernel stack.
- i386 backtrace is not very sensitive to whether a process is running or not. ia64 unwind is very, very sensitive to whether a process is running or not.

—

What happens when MCA/INIT is delivered what a cpu is running user space code?

The user mode registers are stored in the RSE area of the MCA/INIT on entry to the OS and are restored from there on return to SAL, so user mode registers are preserved across a recoverable MCA/INIT. Since the OS has no idea what unwind data is available for the user space stack, MCA/INIT never tries to backtrace user space. Which means that the OS does not bother making the user space process look like a blocked task, i.e. the OS does not copy pt_regs and switch_stack to the user space stack. Also the OS has no idea how big the user space RSE and memory stacks are, which makes it too risky to copy the saved state to a user mode stack.

—

How do we get a backtrace on the tasks that were running when MCA/INIT was delivered?

mca.c:::ia64_mca_modify_original_stack(). That identifies and verifies the original kernel stack, copies the dirty registers from the MCA/INIT stack' s RSE to the original stack' s RSE, copies the skeleton struct pt_regs and switch_stack to the original stack, fills in the skeleton structures from the PAL minstate area and updates the original stack' s thread.ksp. That makes the original stack look exactly like any other blocked task, i.e. it now appears to be sleeping. To get a backtrace, just start with thread.ksp for the original task and unwind like any other sleeping task.

—

How do we identify the tasks that were running when MCA/INIT was delivered?

If the previous task has been verified and converted to a blocked state, then sos->prev_task on the MCA/INIT stack is updated to point to the previous task. You can look at that field in dumps or debuggers. To help distinguish between the handler and the original tasks, handlers have _TIF_MCA_INIT set in thread_info.flags.

The sos data is always in the MCA/INIT handler stack, at offset MCA_SOS_OFFSET. You can get that value from mca_asm.h or calculate it as KERNEL_STACK_SIZE - sizeof(struct pt_regs) - sizeof(struct ia64_sal_os_state), with 16 byte alignment for all structures.

Also the comm field of the MCA/INIT task is modified to include the pid of the original task, for humans to use. For example, a comm field of ‘MCA 12159’ means that pid 12159 was running when the MCA was delivered.

CHAPTER

EIGHT

SERIAL DEVICES

8.1 Serial Device Naming

As of 2.6.10, serial devices on ia64 are named based on the order of ACPI and PCI enumeration. The first device in the ACPI namespace (if any) becomes /dev/ttyS0, the second becomes /dev/ttyS1, etc., and PCI devices are named sequentially starting after the ACPI devices.

Prior to 2.6.10, there were confusing exceptions to this:

- Firmware on some machines (mostly from HP) provides an HCDP table[1] that tells the kernel about devices that can be used as a serial console. If the user specified "console=ttyS0" or the EFI ConOut path contained only UART devices, the kernel registered the device described by the HCDP as /dev/ttyS0.
- If there was no HCDP, we assumed there were UARTs at the legacy COM port addresses (I/O ports 0x3f8 and 0x2f8), so the kernel registered those as /dev/ttyS0 and /dev/ttyS1.

Any additional ACPI or PCI devices were registered sequentially after /dev/ttyS0 as they were discovered.

With an HCDP, device names changed depending on EFI configuration and "console=" arguments. Without an HCDP, device names didn' t change, but we registered devices that might not really exist.

For example, an HP rx1600 with a single built-in serial port (described in the ACPI namespace) plus an MP[2] (a PCI device) has these ports:

Type	MMIO address	pre-2.6.10 (EFI console on builtin)	pre-2.6.10 (EFI console on MP port)	2.6.10+
builtin	0xff5e0(	ttyS0	ttyS1	ttyS0
MP UPS	0xf8031	ttyS1	ttyS2	ttyS1
MP Console	0xf8030	ttyS2	ttyS0	ttyS2
MP 2	0xf8030	ttyS3	ttyS3	ttyS3
MP 3	0xf8030	ttyS4	ttyS4	ttyS4

8.2 Console Selection

EFI knows what your console devices are, but it doesn’t tell the kernel quite enough to actually locate them. The DIG64 HCDP table[1] does tell the kernel where potential serial console devices are, but not all firmware supplies it. Also, EFI supports multiple simultaneous consoles and doesn’t tell the kernel which should be the “primary” one.

So how do you tell Linux which console device to use?

- If your firmware supplies the HCDP, it is simplest to configure EFI with a single device (either a UART or a VGA card) as the console. Then you don’t need to tell Linux anything; the kernel will automatically use the EFI console.

  (This works only in 2.6.6 or later; prior to that you had to specify “console=ttyS0” to get a serial console.)

- Without an HCDP, Linux defaults to a VGA console unless you specify a “console=” argument.

NOTE: Don’t assume that a serial console device will be /dev/ttyS0. It might be ttyS1, ttyS2, etc. Make sure you have the appropriate entries in /etc/inittab (for getty) and /etc/securetty (to allow root login).

8.3 Early Serial Console

The kernel can’t start using a serial console until it knows where the device lives. Normally this happens when the driver enumerates all the serial devices, which can happen a minute or more after the kernel starts booting.

2.6.10 and later kernels have an “early uart” driver that works very early in the boot process. The kernel will automatically use this if the user supplies an argument like “console=uart,io,0x3f8”, or if the EFI console path contains only a UART device and the firmware supplies an HCDP.

8.4 Troubleshooting Serial Console Problems

No kernel output after elilo prints “Uncompressing Linux…done”:

- You specified “console=ttyS0”but Linux changed the device to which ttyS0 refers. Configure exactly one EFI console device[3] and remove the “console=” option.

- The EFI console path contains both a VGA device and a UART. EFI and elilo use both, but Linux defaults to VGA. Remove the VGA device from the EFI console path[3].

- Multiple UARTs selected as EFI console devices. EFI and elilo use all selected devices, but Linux uses only one. Make sure only one UART is selected in the EFI console path[3].
- You' re connected to an HP MP port[2] but have a non-MP UART selected as EFI console device. EFI uses the MP as a console device even when it isn' t explicitly selected. Either move the console cable to the non-MP UART, or change the EFI console path[3] to the MP UART.

Long pause (60+ seconds) between "Uncompressing Linux…done" and start of kernel output:

- No early console because you used "console=ttyS<n>" . Remove the "console=" option if your firmware supplies an HCDP.
- If you don' t have an HCDP, the kernel doesn' t know where your console lives until the driver discovers serial devices. Use "console=uart,io,0x3f8" (or appropriate address for your machine).

Kernel and init script output works fine, but no "login:" prompt:

- Add getty entry to /etc/inittab for console tty. Look for the "Adding console on ttyS<n>" message that tells you which device is the console.

"login:" prompt, but can' t login as root:

- Add entry to /etc/securetty for console tty.

No ACPI serial devices found in 2.6.17 or later:

- Turn on CONFIG_PNP and CONFIG_PNPACPI. Prior to 2.6.17, ACPI serial devices were discovered by 8250_acpi. In 2.6.17, 8250_acpi was replaced by the combination of 8250_pnp and CONFIG_PNPACPI.

[1]

http://www.dig64.org/specifications/agreement The table was originally defined as the "HCDP" for "Headless Console/Debug Port." The current version is the "PCDP" for "Primary Console and Debug Port Devices."

[2]

The HP MP (management processor) is a PCI device that provides several UARTs. One of the UARTs is often used as a console; the EFI Boot Manager identifies it as "Acpi(HWP0002,700)/Pci(…)/Uart" . The external connection is usually a 25-pin connector, and a special dongle converts that to three 9-pin connectors, one of which is labelled "Console."

[3]

EFI console devices are configured using the EFI Boot Manager "Boot option maintenance" menu. You may have to interrupt the boot sequence to use this menu, and you will have to reset the box after changing console configuration.