-*-Mode: outline-*-

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).

* 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)
```

* 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:

** 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

** 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.

* 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.

* 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 in little-endian mode before the first
        load/store instruction is executed. PSR.be is normally NOT
        restored upon return from an fsys-mode handler. In other
        words, user-level code must not rely on PSR.be being preserved
        across a system call.
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 that triggers at any privilege level other than 3
(user-mode).
PSR.lp Unchanged.
PSR.tb Lazy redirect. If a taken-branch trap occurs while in
        fsys-mode, the trap-handler modifies the saved machine state
        such that execution resumes in the gate page at
        syscall_via_break(), with privilege level 3. Note: the
        taken branch would occur on the branch invoking the
        fsyscall-handler, at which point, by definition, a syscall
        restart is still safe. If the system call number is invalid,
        the fsys-mode handler will return directly to user-level. This
        return will trigger a taken-branch trap, but since the trap is
        taken _after_ restoring the privilege level, the CPU has already
        left fsys-mode, so no special treatment is needed.
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 handler then modifies the saved machine
        state such that execution resumes in the gate page at
        syscall_via_break(), with privilege level 3.
PSR.ri Unchanged.
PSR.ed Unchanged. Note: This bit could only have an effect if an fsys-mode
        handler performed a speculative load that gets NaTted. If so, this
        would be the normal & expected behavior, so no special treatment is
        needed.
PSR.bn Unchanged. Note: fsys-mode handlers may clear the bit, if needed.
        Doing so requires clearing PSR.i and PSR.ic as well.
PSR.ia Unchanged. Note: the ia64 linux kernel never sets this bit.

* 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)

 o AT_SYSINFO : is the address of __kernel_syscall_via_epc()
 o 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.