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.