SMP Support in the Kernel#

SMP support is provided only by the procnto-smp variant. This is built using source from:

- VARIANT_smp conditional code
- code in the ker/CPU/smp directory to override uniprocessor code in ker/CPU

The major changes for SMP are in:

- ker/CPU/kernel.s, to handle SMP system call, exception and interrupt handling
- FPU support code, to deal with lazy context switching of FPU context across multiple cpus
- the scheduler, to figure out how to assign runnable threads to each cpu
- message passing code (SMP_MSGOPT only), to allow multiple cpus to perform message passing operations

---

inkernel and cpunum#

On a uniprocessor, inkernel holds only the following:

- an interrupt nesting counter
- a set of flags indicating the kernel state

For a uniprocessor, the inkernel variable can be either in a memory location (ie. a regular variable), or in a dedicated cpu register. Using a dedicated cpu register provides faster access for inkernel manipulation.

To hide the inkernel implementation, ker/CPU/kercpu.h provides a number of macros that the cpu-independent kernel code uses to manipulate inkernel:

- HAVE_INKERNEL_STORAGE indicates whether the cpu-specific code implements the inkernel value - if this is not set, externs.h defines a global inkernel variable
- INKERNEL_BITS_SHIFT specifies where the inkernel state bits live in the inkernel value
- get_inkernel() is used to read the inkernel value
- set_inkernel() is used to set the inkernel value
- am_inkernel() is used to check if we are executing kernel code (system call, exceptions or interrupt handling)
- bitset_inkernel() is used to set inkernel state bits
- bitclr_inkernel() is used to clear inkernel state bits

For SMP, the inkernel value must be implemented as a variable in memory since multiple cpus can access it:

- HAVE_INKERNEL_STORAGE must be set to 1 and kernel.s needs to implement the inkernel storage
- inkernel now holds information about which cpu has acquired the kernel as well as the state bits and interrupt nesting counter
- am_inkernel() needs to be a bit smarter since it can't test if inkernel is non-zero since another cpu could have acquired the kernel:
  - on cpus that use a dedicated cpu mode for kernel code, this can check the cpu mode. eg. ring0 on x86 or SVC mode on ARM
  - otherwise, it must check if we are executing on the kernel stack for the cpu (only kernel code switches to the dedicated, per-cpu kernel stack)
- bitset_inkernel() and bitclr_inkernel() must operate atomically

Kernel code occasionally needs to know what cpu it is running on. It uses two macros, depending on what it needs to do:

- KERNCPU indicates the cpu that has acquired the kernel (the value of the cpunum variable)
- RUNCPU indicates the actual cpu the code is running on (using a cpu-specific get_cpunum() function)

Note that these can be different - it is possible for one cpu to be executing a kernel call with other cpus handling interrupts.

On a uniprocessor, both KERNCPU and RUNCPU are statically compiled to 0.

Initialisation#

The overall initialisation sequence for SMP is described here:

- cpu_interrupt_init() must set up the plumbing for handling IPI interrupts
- smp_start() is the kernel entry point used to initialise secondary cpus

cpu_interrupt_init()#

This is a cpu-specific function, in ker/CPU/interrupt.c, used to dynamically generate the interrupt dispatch code. It glues together:

- the exception entry code between intr_entry_start and intr_entry_end in ker/CPU/kernel.s
- the id callouts for the interrupt controller configuration described by the system page
- code to branch to the main interrupt dispatch code (intr_process_queue in kernel.s)
- the eoi callout for the interrupt source being handled
- code to branch to the main interrupt return code (intr_done in kernel.s)

For SMP, cpu_interrupt_init() also needs to plumb in the support for IPI interrupts:

- the board-specific startup is expected to define a config callout for the interrupt controller that implements the IPI interrupts
- this config callout returns INTR_CONFIG_FLAG_IPI for the IPI interrupt vector
- when cpu_interrupt_init() encounters an interrupt vector with this flag set, it generates code to branch to intr_process_ipi instead of intr_process_queue

The intr_process_queue code handles only IPI interrupts so this special handling is an optimisation to avoid going through the intr_process_queue code.

smp_start()#

This is the kernel entry point for secondary cpus and each secondary cpu begins execution here once it is released from the spin loop in the smp_spin callout.

Its responsibilities are to:

- perform any memmgr initialisation required for this cpu
- plumb in the send_ipi callout so that the kernel's send_ipi() function can deliver IPIs
- call ker_start() to resume execution in the cpu's idle thread

---

Kernel Entry and Exit#

IPI Handling#

Interprocessor interrupts are used for a number of situations where an activity one cpu requires something to occur on other cpus:

- the initiating cpu uses the send_ipi() function to send the "command" to another cpu
- send_ipi() uses the board-specific send_ipi callout provided by the startup program to trigger the interrupt
- the target cpu handles the interrupt in the intr_process_ipi routine in ker/CPU/kernel.s

There are a couple of things to note:

- the protocol is asynchronous. If the initiating cpu needs to wait for the action to be performed before proceeding it must explicitly manage the synchronisation.
- the IPI commands are implemented as single bits. This means that:
  - the initiator can simply (atomically) set a bit to indicate a pending command
  - commands are not queued
  - commands cannot take parameters

The most common commands are:

- IPI_TLB_FLUSH, invoked by the memmgr to flush TLB entries when address translations are changed
- IPI_TIMESLICE, invoked by the clock handler if the thread running on this cpu has used up its scheduling timeslice
- IPI_RESCHED, invoked by the scheduler if it decides that this cpu needs to perform a rescheduling operation
- IPI_CONTEXT_SAVE, invoked when the active FPU on this cpu needs to be saved (eg. to load the context on another cpu)

intr_process_queue is invoked directly by the interrupt dispatch code generated by cpu_interrupt_init():

- it is called with interrupts disabled and must return with interrupts disabled
- it is called with the intr_slock locked, and must return with the intr_slock locked

Although it is littered with cpu-specific details, the operation is essentially:

```
release the intr_slock
save_state = cpupageptr[RUNCPU]
cpupageptr[RUNCPU]->state = 1;
cmd = _smp_xchg(&ipi_cmds[RUNCPU], 0);
if (cmd & IPI_TLB_SAFE) {
   set_safe_aspace();
}
if (cmd & IPI_CLOCK_LOAD) {
   clock_load();
}
if (cmd & IPI_INTR_MASK) {
   interrupt_smp_sync(INTR_FLAG_SMP_BROADCAST_MASK);
}
if (cmd & IPI_INTR_UNMASK) {
   interrupt_smp_sync(INTR_FLAG_SMP_BROADCAST_UNMASK);
}
if (cmd & IPI_TLB_FLUSH) {
   cpu-specific actions for flushing TLBs
}
pending_async_flags = 0;
if (cmd & IPI_TIMESLICE) {
   pending_async_flags |= _NTO_ATF_TIMESLICE;
```

```
}
if (cmd & IPI_RESCHED) {
   pending_async_flags |= _NTO_ATF_SMP_RESCHED;
}
if (cmd & IPI_CONTEXT_SAVE) {
   save FPU context in actives_fpu[RUNCPU]->fpudata
   clear BUSY/CPU bits  in actives_fpu[RUNCPU]->fpudata
   actives_fpu[RUNCPU] = 0;
}
if (pending_async_flags) {
   if (interrupted user mode) {
      pending_async_flags |= _NTO_ATF_FORCED_KERNEL;
   }
   else if (interrupted __ker_entry code spinning while waiting to acquire kernel) {
      SETKIP(act, KIP(act) - KER_ENTRY_SIZE);
      set state so intr_done thinks we entered from user mode
      pending_async_flags |= _NTO_ATF_FORCED_KERNEL;
   }
   old_flags = _smp_xchg(&act->async_flags, pending_async_flags);
   if ((pending_async_flags & _NTO_ATF_FORCED_KERNEL) && !(old_flags & _NTO_ATF_FORCED_KERNEL)) {
      act->args.async.save_ip = KIP(act);
      act->args.async.save_type = KTYPE(act);
      SETKTYPE(act, __KER_NOP);
      SETKIP(act, kercallptr);
   }
}
cpupageptr[RUNCPU]->state = save_state
lock intr_slock
```

The handling of async_flags deserves a little more explanation. The intention of these IPI commands is to cause some form of rescheduling, which will be performed via __ker_exit when it checks the thread's async_flags.

However, the return from interrupt code for SMP may not necessarily directly return via __ker_exit so the easiest way to ensure that we will run through __ker_exit at some point is to force the current thread to make a null system call. We can only do this if:

- we interrupted user mode. In this case we need to
  - save the registers used for invoking a system call
  - since we interrupted user mode, intr_done will cause the thread to resume at kercallptr to invoke the __KER_NOP system call
- we interrupted the __ker_entry code while it was spinning waiting to the acquire the kernel for a system call. In this case, we need to:
  - modify the thread's saved context so that it will restart the system call
  - modify the (cpu-specific) state to make intr_done think we interrupted user mode so that the thread will resume at kercallptr and invoke the __KER_NOP system call

If we interrupted any other kernel code, we simply set the thread's async_flags and expect to process them when the kernel code eventually returns to user mode.

When this forced __KER_NOP call returns through __ker_exit, the _NTO_ATF_FORCED_KERNEL flag will still be set, so the code knows that it must restore the KTYPE and KIP registers that were saved in the thread's args.async fields:

- if we had interrupted user mode, we will resume at that original user code on return from the kernel
- if we had interrupted the __ker_entry entry sequence, we will restart the original system call

FPU Handling#

To avoid having to save/restore FPU context on each context switch, the kernel implements a lazy context switch mechanism:

- actives_fpu[RUNCPU] indicates which thread currently has active FPU on that cpu
- in __ker_exit, the FPU is disabled if actives_fpu[RUNCPU] != actives[RUNCPU]

This means that if the thread accesses the FPU, it will generate an exception:

- if the thread has no FPU save area, the exception handler arranges to allocate one by setting _NTO_ATF_FPUSAVE_ALLOC in the thread's async_flags:
  - when the thread returns through __ker_exit, fpusave_alloc() is called to allocate and initialise the context in the save area
  - the FPU still remains disabled so the thread will re-execute the instruction and take another exception, but this time it will have a save area

- if the thread has a save area, an FPU context switch needs to be performed:
  - if actives_fpu[RUNCPU] is non-0, the FPU context is saved in that thread's save area
  - the FPU context is restored from the faulting thread's save area and actives_fpu[RUNCPU] is set the new thread
  - the FPU is now enabled, so when the thread is resumed, the FPU operation is restarted with the correct context

For SMP, this lazy scheme means that it's possible for a thread to have used the FPU on one cpu but then be rescheduled onto another cpu. If the thread then uses the FPU again, it needs to save the FPU context on the old cpu so that it can be restored on the new cpu.

Similarly, when a thread terminates and its save area needs to be freed, it may have active FPU context on another cpu that must be flushed.

To handle this, the thread's fpudata pointer has some additional information to indicate if the context is active on a cpu:

- FPUDATA_INUSE() indicates that the FPU context is active on a cpu
- FPUDATA_CPU() indicates which cpu holds the active FPU context
- FPUDATA_PTR() is a macro to remove the cpu/busy bits to generate a pointer to the save area

So, overall, the context switch algorithm in the FPU exception handler is:

```
if (thp->fpudata == 0) {
    atomic_set(&thp->async_flags, _NTO_ATF_FPUSAVE_ALLOC);
    return;
}
if (actives_fpu[RUNCPU] != 0) {
    save FPU context in FPUDATA_PTR(actives_fpu[RUNCPU]->fpudata);
    actives_fpu[RUNCPU] = 0;
}
if (FPUDATA_INUSE(thp->fpudata) && FPUDATA_CPU(thp->fpudata) != RUNCPU) {
    // send an IPI to the cpu with the FPU context and then wait until it has been saved
    SENDIPI(FPUDATA_CPU(thp->fpudata), IPI_CONTEXT_SAVE);
    bitset_inkernel(INKERNEL_EXIT);
    unlock_kernel();
    while (FPUDATA_INUSE(thp->fpudata))
        ;
    lock_kernel();
```

```
}
load FPU context from FPUDATA_PTR(thp->fpudata);

// indicate FPU context is active on this cpu
thp->fpudata |= FPUDATA_BUSY | RUNCPU;
actives_fpu[RUNCPU] = thp;
```

Note that the kernel is typically locked (INKERNEL_LOCK is set) during exception handling so we need to:

- unlock the kernel so we can be preempted while we spin
- set INKERNEL_EXIT so that the faulting instruction is restarted if we do get preeempted

---

CPU Scheduling#

Message Passing (SMP_MSGOPT)#
