1 Booting the kernel

1.1 Qemu

Qemu starts by executing the code stored in the ROM (0x1000). This code does:

- a0 = mhartid
- a1 = PHYSTOP
- a2 = end of the ROM code #useless
- pc = KERNBASE (PHYSTART)

So, the kernel code should be loaded at KERNBASE (0x80000000). As Qemu loads the virtual address space of the binary file, we need to link so that the code starts at KERNBASE. This step is done in the linking phase.

1.2 entry.S

entry.S is loaded at PHYBASE (0x80000000). The code aims at providing each core with a stack and then calling the kernel code (**start.c**).

The stacks are defined in the C code as global arrays.

1.3 start.c

start.c is called by **entry.S**. This is the very first piece of kernel code which is executed.

The code is executed is **machine mode** as the microprocessor starts is machine mode.

This code aims at configuring the microprocessor before passing the baton to the supervisor mode.

The following tasks are done:

- machine trap configuration : interrupt/exception delegation, interrupt enabling, machine trap vector
- physical memory protection
- preparing return the supervisor mode

Then, the kernel code runs and configures the kernel's services in **main.c**.

1.4 main.c

main.c calls the configuration function of all kernel's services, namely:

- UART
- CLINT (timer interrupts)
- PLIC (external interrupts)
- page allocation

- virtual memory
- processes
- supervisor trap (in kernel mode)

Some of these services need to be configured only once. So, this is only done by **hart 0**.

1.5 alloc.c

alloc.c gathers all the code related to the page allocation, namely:

- *uint64* hd_pagelist*: linked list of free pages. Each page in the free page list stores the adress of the next free page in its 8 first bytes of the page.
- *void allocinit()*: divides the unused DRAM memory into chunks of *PAGESIZE*
- *void* alloc()*: removes a free page from the page list and returns it
- *void free(void* p)*: resets the page p to 0 and appends it at the begining of the free page list

1.6 vm.c

vm.c gathers all the code related to the virtual memory management, namely:

- *t_pagetable kernel_pagetable*: kernel's page table
- *mvoid mappages(t_pagetable kernel_pagetable, void* va, uint64 sz, void* pa, uint64 perm)*: maps pages starting from virtual address *va* until *va + sz* to physical pages starting from physical address *pa* with permission *perm*
- *void kmvinit()*: creates the kernel page table
- *void kvminihart()*: loads the kernel's page table in satp register

1.7 proc.c

proc.c gathers all code related to processes, namely:

- *void procinit()*: creates for each process its pagetable.
- *void proclaunch()*: runs the process on the hart

2 Core Local Interruptor (CLINT)

The CLINT is responsible of timer interrupts. It contains a **MTIME** register which is incremented automatically and **MTIMECMP** registers for each hart.

Whenever MTIMECMP ¿= MTIME, a **timer interrupt** is raised for the hart. This interrupt is reflected in **MTIP** bit of the **mip** register. MTIMECMP

needs to be incremented to clear the timer interrupt.

So, the CLINT can only generate timer interrupt for the **machine** mode.

hart's x mtimecmp register 64 bits @ base + 0x4000 + 8 * x
mtime register 64 bits @ base + 0xbff8

3 Platform Local Interrupt Controller (PLIC)

3.1 Internals

The PLIC makes a communication link from **external devices** to the **microprocessor**.

3.1.1 f

or the hartInterrupt Gateways The interrupt gateways are responsible for converting global interrupt signals into a common interrupt request format, and for controlling the flow of interrupt requests to the **PLIC core**.

External devices are connected to the **Gateway** through an **irq** line to send interrupt signals (either level trigger or edge trigger).

Each source has an **Interrupt Identifier**. Interrupt indentifier 0 is reserved (so, it starts from 1).

The gateway only forwards a new interrupt request to the PLIC core after receiving notification that the interrupt handler servicing the previous interrupt request from the same source has completed (or if it is the very first interrupt from the source).

At most one interrupt request per interrupt source can be pending in the PLIC core at any time, indicated for the PLIC core by setting the **source's IP bit**.

3.1.2 Interrupt priority

Each interrupt source has a priority, which is a 32 bits integer. A priority equals to 0 means "never interrupt" and interrupt priority increases with increasing integer values. The priority is stored in a memory-mapped register.

interrupt source x priority: 32 bits @ base + 4 * x

3.1.3 Interrupt targets

An interrupt target is a given privilege mode on a given hart. For instance, machine mode on hart 1, Supervisor mode on hart 1 ...

3.1.4 Interrupt enable

Each target can enable interrupts comming from every sources by setting a 1 in the target's **enable regsiter**.

The PLIC will mask the interrupts from sources not enabled by the target.

enbale bits for source x on target y: bit x % 8 @ base + 0x2000 + 0x80 * y + x / 8

3.1.5 Interrupt threshold

Each target sets a threshold by writting the target's **threshold register**. The PLIC core will mask interrupts whose priority is not greater than the target's threshold.

target x threshold: 32 bits @ base + 0x2000 + 0x1000 * x

3.1.6 Interrupt notification

Each target has a **EIP bit** in the PLIC core thats indicates that the corresponding target has an interrupt pending. The EIP bit will be reflected in the hart's **interrupt pending register**.

In order to have the EIP bit in the PLIC core set for a target, it requires:

- an interrupt request has been sent to the PLIC core by the Gateway
- the interrupt's source is enabled for the target
- the interrupt's priority is greater than the target's threshold

3.1.7 Interrupt claim process

Once a target has been notified about an interrupt (interrupt pending register), the target must **claim** the PLIC to service the interrupt. The **claim** action corresponds to reading the target's **claim/complete register** in order to get the **Interruption Indentifier** responsible of the interrupt.

On receiving a claim, the PLIC will determine the ID if the highest priority pending interrupt for the target. 0 will be returned in case there is no pending interrupt. This case can happen if several targets have been notified for an interrupt and one of them has already claimed the interrupt.

On receiving a claim, the PLIC will clear the **source's IP bit** partially responsible for setting the **EIP** bit of the target. In fact, other source's IP bits can still set the target's EIP bit to 1. It is designed like that so that the target can claim multiple interrupts in one shot until the interrupt pending register is 0 (prevent many context switches). Indeed, claiming can be done at any time.

target x claim/complete register : 32 bits @ base + 0x2000 + 0x1000 * x + 0x4

3.1.8 Interrupt complete

Once the target has completed the interrupt, it must **complete** the interrupt to the PLIC, so that the PLIC will forward the complete message to the Gateway which in turn would be able to forward new interrupt requests.

Completing consists in writting the **interrupt indentifier** to the **claim/complete register**.

target x claim/complete register : 32 bits @ base + 0x2000 + 0x1000 * x + 0x4

3.2 Configuration

In order to configure the PLIC, we need to:

- think about sources
- think about targets
- set sources' priority
- set targets' thrshold
- enable sources for targets

In order to handle the external interrupt, we need to:

- claim
- if ID == 0 :
  - do nothing
- else :
  - do some stuff related to ID
- complete

3.3 Sources

- https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc#interrupt-notifications

4 Traps

Traps indicate exception (raised by executing an instruction) and interruptions (raised by another module).

An exception will be automatically taken by the corresponding mode (indicated by Xedelg register), if trap are enabled in mode X.

An interrupt will be serviced by mode x if:

- the interrupt has been delegated to mode x
- interrupt bit is set in xip
- interrupt is enabled in xie
- interrupts are enabled globally. It is the case if ie bit is set in xstatus register or if the current privilege mode is lower than x.

4.1 Trap flow

Whenever a trap occurs in mode Y and is handled in mode X (and X enabled traps), the following actions happen:

- Xepc ¡- pc
- Xstatus.ie ¡- 0
- Xstatus.XPP ¡- Y
- Xcause ¡- ...
- Xtval ¡- ...
- pc ¡- Xtvec
- new mode is X

4.2 Machine trap

mtvec points to **mtrapvec** in **kernelvec.S**.

matrpvec saves all registers to per hart save area and loads the stack pointer with a per hart stack dedicated for trap handling, and finally jumps to **mtraphandler**.

mtraphandler handles the trap and passes the baton to **mtrapvecret**.

mtrapvecret does exactly the contrary of **mtrapvec**, namely, loading all saved registers.

As there is only one save area per hart, it is mandatory that the trap handling in machine mode can not be interrupted again (otherwise the regsiters would be overwritten).

A machine trap is usefull during kernel development as it enables to catch all interrupts from kernel mode easily (no satp).

4.3 Trap from user mode

trampoline.S gathers the code for switching context between user mode and supervisor mode. It contains 2 functions, namely:

- *uservec*: saves all registers in the *trapframe*, whose address is stored in *sscrtach*, and the loads registers for supervisor mode, and finally changes the *pagetable* to that of kernel.
- *uservecret*: does the countrary of *uservec*.

The code in **trampoline.S** needs to be located at the same address in both user and kernel pagetable, as when we switch pagetables, the program counter points to the next instruction.