Lab 4: Multi-core and Locks

18307130024 Jimmy Tan

1 Multi-processing Systems

This section is based on ARMv8-A Programmer Guide.

1.1 Getting the information about the running core

Software may need to know which core its code is executing upon. `ARMv8` provides the *Multi-Processor Affinity Register* (`MPIDR_EL1`) to identify the core thus satisfying the demand.

To configure a virtual machine, `EL2` and `EL3` can set `MPIDR_EL1` to different values at run-time. `MPIDR_EL3` is based on the physical core and its value cannot be modified.

1.2 Symmetric multi-processing

In symmetric multi-processing (`SMP`), each core has the same view of memory and of shared hardware. There is a layer of abstraction from the softwares' perspective because the operating system hides the complexity of scheduling the tasks within or between the cores.

There are several trade-offs on whether to spread the tasks on more or less cores:

- Energy. On one hand, scheduling tasks on fewer cores can provide more idle resources. On the other hand, to achieve the same performance of a single core, spreading tasks on multiple cores requires lower frequencies, thus saving the power.
- Interrupt handling. Interrupts can be handled by multiple cores, which can reduce interrupt latency and the time spent on context switching. Similarly, only allowing interrupts to be handled on a few certain cores can reduce complexity.

1.3 Timers

The System Timer Architecture involves a global system counter at a fixed clock frequency. There are also local timers, secure and insecure ones, and ones for virtualization purposes. To implement the timer interrupt, each channel has a comparator and generates a timer interrupt when the count of the global counter is greater than or equal the the comparator.

Although the frequency of the global timer is fixed, the step of incrementing can be modified, to either 10 to 100 per clock tick, or every 10 to 100 clock ticks.

The `CNTFRQ_EL0` register reports the frequency of the system timer. The `CNTPCT_EL0` register reports the current count value. The `CNTKCTL_EL1` register controls whether `EL0` can access the system timer.

1.4 Synchronization

The `aarch64` instruction set has three instructions for synchronization:

- Load Exclusive (`LDXR`)
- Store Exclusive (`STXR`). The instruction gives a signal about whether the store completed successfully.
- Clear Exclusive access monitor (`CLREX`)

Although the architecture and the hardware support implementations of exclusive access, the programmers should keep the right behavior on the software level.

2 Homework

2.1 Multi-core Booting

I will divide the `entry.S` into five parts, according to the symbols in this file.

2.1.1 _start

The `_start` section is located at address `0x80000`. In this section, the CPU stores the address of the next section `mp_start` somewhere in the memory.

This section is only executed by one CPU core (in our labs, CPU #0 executes this part). The CPU core has the mission to wake other the CPU cores up, which is divided into two parts: one is store the address of `mp_start` in $n - 1$ locations where n means the total CPU cores in the machines (in our labs, n is 4); the other is synchronize data and instruction memory via `dsb` and `isb` instruction, and wake other CPU cores up via `sev` instruction.

For the other cores, they drop in an infinite loop executing `wfe` instruction before being waken. This instruction would set the processor in a low power mode and wait for being waken.

The below parts of instruction is executed by all cores.

2.1.2 mp_start

This part is multi-processor start part. The processor gets the information of the current exception level and branches to the corresponding part.

2.1.3 `EL3`

This part configures the `SCR_EL3`, the `SPSR_EL3` and `ELR_EL3` register and then jump to exception level 2 via `eret` instruction.

2.1.4 `EL2`

This part configures the `HCR_EL2`, the `SCTLR_EL1`, the `SPSR_EL2` and the `ELR_EL2` register and then jump to exception level 1 via `eret` instruction.

2.1.5 `EL1`

In this part:

- The kernel page table is configured and the `MMU` is enabled, which means the instructions above all use physical address. The related registers are `TTBR0_EL1`, `TTBR1_EL1`, `TCR_EL1`, `MAIR_EL1` and `SCTLR_EL1`.
- The stack pointer is configured for all cores. Each stack pointer is located at the beginning of different pages. The address of the pages are below the `_start` section in order to avoid conflicts.

Finally, the processor branches to main function.

2.2 Spin-locks and Interrupts

Reference of this part: xv6-armv8

When a task acquires a lock, the interrupt should be disabled. Otherwise, deadlocks would exist. During the period that a task holds a lock, if an interrupt stops the task, the interrupt handler probably acquires the same lock, which would not be released before the processor continues to execute this task.

So we should call `cli()` in the `acquire` function.

In `release` function, we should restore the interrupt settings. As a result, we need to store whether the interrupt is enabled in the `acquire` function.

Considering a task may hold multiple locks, and the `acquire` and `release` operations are similar to `push` and `pop` operations in a stack, I maintain a number of the locks hold by tasks on a CPU.

```
struct cpu {
    int lock_num; // record the num of locks
    unsigned char prev_int_enabled; // record the interrupt settings before the first clock acquired
};
```

In `acquire` function, I check if there are any locks on this CPU; if so, record the current settings of the interrupt. Then, I get the lock numbers of the CPU plus 1 and stop the interrupt.

```
int id = cpuid();
if (cpus[id].lock_num++ == 0) {
    cpus[id].prev_int_enabled = enabled;
}
```

In `release` function, I check if there are any locks on this CPU after this lock is released. If so, do nothing and return; otherwise, I restore the interrupt settings.

```c
int id = cpuid();
cpus[id].lock_num--;
if ((cpus[id].lock_num == 0) && cpus[id].prev_int_enabled) {
    sti();
}
```

2.3 Add locks

Several functions is `main` should be executed only on one CPU. So I use a global variable to count how many times those functions are executed. Accessing this value should acquire a lock.

From my perspective, only `alloc_init` should be executed once to avoid a single page to be added multiple times in the free list. The other functions should be executed on each core.

The code is:

```c
struct check_once {
    int count;
    struct spinlock lock;
}; // Records the count and the lock
struct check_once alloc_once = { 0 }; // Initialized as 0
void
main()
{
    // ...
    acquire(&alloc_once.lock);
    if (!alloc_once.count) {
        // Do alloc_init
    }
    release(&alloc_once.lock);
    // ...
}
```

In `kalloc.c`, the `kfree` and `kalloc` function access and modifies the value of a global variable `kmem`, so a lock should be added on it.