OS Lab4 Report

Author: Chengzu Ou (14302010037)

Description

In the first exercise, we are asked to implement `mmio_map_region` in `kern/pmap.c` to allocate space from MMIO region and map device memory to it. Note that we should check if the memory exceed `MMIOLIM`.

In the second exercise, `boot_aps()` first copies code in `mpentry.S` to memory at `MPENTRY_PADDR`. Then, it start a process for every `cpu` which is `APS`. At last, it jumps to execute `mpentry.S`. The function of `mpentry.S` is similar to boot loader. It will jump to `mp_main` and do some initialization work.

In the third exercise, we are asked to allocate memory for each CPU. And in exercise four, we should init every CPU.

To prevent race condition when multiple CPUs run kernel code simultaneously, we should implement a kernel lock. In this lab, environments in user mode can run concurrently on any available CPUs, but no more than one environment can run in kernel mode; any other environments that try to enter kernel mode are forced to wait.

Answers

Q1: Compare `kern/mpentry.S` side by side with `boot/boot.S`. Bearing in mind that `kern/mpentry.S` is compiled and linked to run above `KERNBASE` just like everything else in the kernel, what is the purpose of macro `MPBOOTPHYS`? Why is it necessary in `kern/mpentry.S` but not in `boot/boot.S`? In other words, what could go wrong if it were omitted in kern/`mpentry.S`?

The function of `MPBOOTPHYS` is to map high address to low address. When executing this code, we are still in real mode but the address in the code is already in protected mode. So we have to translate it from high address to low address.

Q2: It seems that using the big kernel lock guarantees that only one CPU can run the kernel code at a time. Why do we still need separate kernel stacks for each CPU? Describe a scenario in which using a shared kernel stack will go wrong, even with the protection of the big kernel lock.

When the interrupt happens, it will push to the stack and we have not got the lock yet. When multiple CPUs have interrupt at the same time, shared kernel stack would cause error.

Q3: In your implementation of `env_run()` you should have called `lcr3()`. Before and after the call to `lcr3()`, your code makes references (at least it should) to the variable `e`, the argument to `env_run`. Upon loading the `%cr3` register, the addressing context used by the MMU is instantly changed. But a virtual address (namely `e`) has meaning relative to a given address context--the address context specifies the physical address to which the virtual address maps. Why can the pointer `e` be dereferenced both before and after the addressing switch?

This is because we did static mapping in `envs` and `mem_init()` uses `kern_pgdir` as a template so that `pgdir` in every `env` is copied from `kern_pgdir` and also has mapping.

Q4: Whenever the kernel switches from one environment to another, it must ensure the old environment's registers are saved so they can be restored properly later. Why? Where does this happen?

It is absolutely necessary to save old registers when doing context switching otherwise when switching back, the CPU would have no idea what state it is in and where is the next instruction.

The saving of context is done when interrupts happens. We copied a `Trapframe` in `trap()`.