Lab 4 - Part A: Multiprocessor Support and Cooperative Multitasking

Sriram V - CS11B058

Exercise 1

- Implemented `mmio_map_region()` in `kern/pmap.c`.
- The passed `size` is rounded up and checks are made to ensure that the limit doesn't exceed `MMIOLIM`.
- `boot_map_region()` is then called to map at `MMIOBASE` the passed physical address with the appropriate permissions.

Exercise 2

- The only change that is made to `page_init()` here is a special check for `MPENTRY _PADDR`.
- If the page is the page corresponding to this address, the `pp_ref` is incremented for this page, and its link is set to `NULL`.
- Now the AP bootstrap code can be safely copied and run at this physical address.
- The code now successfully passes the updated `check_page_free_list()`.

Questions

1. **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`? Hint: recall the differences between the link address and the load address that we have discussed in Lab 1.**

- MPBOOTPHYS(s) ((s) - mpentry_start + MPENTRY_PADDR) to calculate absolute addresses of its symbols, rather than relying on the linker to fill them
- If it was exactly like boot.S, then the linker would put the symbols not where we want it, but elsewhere. We have control in this way.

Exercise 3

- In `mem_init_mp()` in `kern/pmap.c`, for each of the `NCPU`s we use `boot_map_region()` to map into the kernel page directory the physical address of each CPU's stack.
- The subtracted amount after each mapping is `KSTKSIZE + KSTKGAP`. This is because we need to account for both the stack size as well as the gap maintained between each stack (so that a stack overflow will not overwrite other stacks).
- The code now passes `check_pgdir()`.

Exercise 4

- Here we modify the `trap_init_percpu()` function in `kern/trap.c` to handle multiple CPUs by primarily modifying the global `ts` variable into a local variable that is equal to the current CPU's Taskstate.
- `ss0` remains as `GD_KD` as the kernel's data segment won't change.
- However, the stack used by each CPU is different, and consequently the `esp0` value is modified from `KSTACKTOP` to `KSTACKTOP - (KSTKSIZE + KSTKGAP) * cpu_id`.

- The index of the `GDT` and its offset vary from CPU to CPU as well, and thus the `TSS` of each CPU is calculated as `GD_TSS0 + (cpu_id << 3)`.
- The rest of the code remains the same.

Exercise 5

- This exercise involves locking and unlocking the kernel at appropritate points to address race conditions when multiple CPUs run the kernel code simultaneously.
- In `i386_init()` in `kern/init.c`, the kernel lock is acquired before the APs are wokern up.
- In `mp_main()`, to ensure that only one CPU can enter the scheduler at a time, we lock the kernel before calling `sched_yield()`.
- In `trap()`, the lock is inserted if `tf_cs` is equal to `3`.
- In `env_run()`, the kernel is unlocked right before popping the new environment's Trapframe.

Questions

2. **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.**

- Can happen in the case of context switch. P1 pushes some data onto kernel stack to be accessed later. Critical data it is. Context switch to P2
- Now P2 also pushes some critical data to access later. now context switch to p1. Now when P1 wants to access it's data that it pushed, it ends up having access to p2's data, a security vulnerability.

Exercise 6

- Round robin scheduler is implemented in `sched_yield()`.
- The index is initialized to `-1`, so that if no current environment exists, the increment inside the loop will set index to zero.
- Else, the index will be set inside the loop to the next environment.
- We loop through all environments and search for an `ENV_RUNNABLE` environment.
- If no such environemnt is found, and the current environment is not null and is `ENV_RUNNING`, then we `env_run()` the current environment itself.
- `yield.c` running on 3 environemnts now uses the scheduler succesfully.

Questions

3. **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 in every environment we map all the virtual addresses above UTOP (the kernel addresses) to be the same. Hence, we can continue to dereference the pointer before and after the context switch.
- Because both page directories (kernel and user mode) has the same entries for the kernel addresses mapping.
- That way reloading %cr3 register makes no difference, since the 'e' pointer will have a kernel address, which is correctly mapped in the just loaded page directory.

4. **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?**

- Called a context switch. Done so that an environment can resume from the pt where it switched. The Trapframe struct contains details of all regs for an env. env's pgdir and tf are changed when `env_run` is called.

Exercise 7

- The system calls implemented are primarily wrapper functions for already existing functions.
- Hence, the only code that had to be written are the required sanity checks before calling these functions.
- `dumbfork.c` now works successfully.