OPERATING SYSTEMS

ASSIGNMENT 5

QEMUSabe : CS13B051 , CS13B055

Introduction

This lab aims to do the following :

A. To build a multiprocessor support with preemptive multitasking.
B. Implement a fork library function to create a child process that mimics the parent process.
C. Add Inter Process Communication support to send and receive signals using a simple IPC algorithm.

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?

Ans : In boot/boot.s the link address and the load address are the same and are in the lower memory region. So there is no need for any conversions.

However in kern/mpentry.s is linked to execute at a higher address than is accessible in real mode , but is of course loaded in the lower address range. Hence we need to get the load address from the higher address.

That means we need to subtract the start address i.e *mpentry_start (f0106908 <mpentry_start>)* and add the *MPENTRY_PADDR (the physical address where it is loaded.)* .

This is what the following macro precisely does :

MPBOOTPHYS(s) ((s) - mpentry_start + MPENTRY_PADDR)

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.

Ans : The big kernel lock ensures that only one process can enter the kernel mode. However this is not sufficient. Since this does not prevent the processor from pushing the *cs* , *ss* , *eip* , *esp* , *eflags* . Consider the following scenario where a process executing some handler in kerel mode pushing arguments onto the stack and another process gets an interrupt. The processor pushes the 5 arguments of the second process onto the same stack ,and thus corrupting it.

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?

Ans: In env_run(), after loading the cr3() the address space changes to the new process' . But the variable 'e' is still accessible before and after changing the page directory.

This is because in *env_setup_vm* which is called from *env_alloc* maps all the Virtual memory in kernel page directory.(i.e. all used memory

above UTOP). This includes the read-only pages mapped at UENVS which contains read-only information about environments. Hence it is accessible even after changing cr3().

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?

Ans: The old environment's registers must be saved , so that it can continue execution exactly at the instruction where it left off and with same state(same values in registers,etc.) . The state is saved in trapframe of the process which is pushed onto kernel stack.

Initially in *trapentry.S* at its entry point(known from IDT entry), *alltraps* before going for a trap handler, we push trapframe onto the stack.

It is then in *trap.c* ,this trapframe is saved in the environment's *env_tf* in the function *void trap*(*struct Trapframe *tf*) . (Line no: 472 in kern/trap.c)