COL331 Operating Systems
Homework 3

Tanay Asija
2016EE10040

Assignment: Paging

VA = 0x80100000 , PA = 0x00100000

CR3 points to pgdirectory, first 10 bits of VA are used to index into pgdirectory.

First 10 bits of VA = $\underbrace{1000}_{8}\underbrace{0000}_{0}\overbrace{00}^{\text{first 2 bits of 1}}_{b} = 512_d$

⇒ Offset in page directory = 512_d

Value at 512th offset = Address (physical) of page table which should be page aligned (i.e last 12 bits are assumed to zero, so only first 20 bits are needed to store in the page directory entry) (PPE)

Permissions for PPE can be set to Read/Write/Present as they can be set individually for pages it points to in the page table entry. Also it should be set as Kernel page (∵ it is a high VA)

⇒ Offset in page table is the next 10 bits of VA

Next 10 bits of VA = $\underbrace{01}_{\text{last 2 bits of 1}}\underbrace{0000}_{0}\underbrace{0000}_{0} = 256_d$

So offset in page table = 256_d

Value at 256th entry = Most significant 20 bits of the physical address ie 0x00100000 or 0x00100, the flags should be set to Read and Present and Kernel (∵ it is a high VA)

Assignment : Page Table Reload

(gdb) print/x kpgdir[0]

Q1 Why is this zero?

Ans The kernel wants to switch pgdirectories at this point by setting up the page directory in such a way that only the high addresses ie above KERNBASE (=2GB) are mapped and the rest are available for user processes. These are all virtual addresses. Since initially it was running on entrypgdir which also mapped VA 0-4MB identically to PA 0-4MB along with the high addresses VA 2GB - 2GB+4MB to PA 0-4MB as it was executing in the low addresses. However at the point of invoking kvmalloc, it has switched to high addresses and can remove these low address mappings. kpgdir[0] would be the first entry corresponding to the page directory entry that would be neffered for a VA with first 10 bits equal to 0. Clearly it should be set to zero to indicate that no mapping exists for this VA (i.e. Present flag is "0").

(gdb) x kpgdir[0]

Q2. How would we translate 0x80107beb to a PA?

Ans The VA 0x80107beb is translated to a PA by subtracting KERNBASE (=2GB or 0x80000000) from this

PA = V2P(0x80107beb) = 0x80107beb - 0x80000000

= 0x107beb = 1080299 (in decimal)

This is because kernel maps itself one-to-one from VA 0x80000000 to PA 0x0.

(gdb) print /x kpgdir [0x200]

o/p : $60x114007

Q3. What is this?

Ans. This is the entry at the offset 0x200 in the page pointed to by kpgdir. Once it (kpgdir) is loaded into CR3 (after converting to its PA) this will serve as the 0x200'th page directory entry, ie it will be the PDE consulted when translating 0x80107beb after kpgdir is loaded into CR3.

Q4. What is the PPN?

Ans. The PPN is of 20 bits, as it is at page aligned boundries. The first 12 bits of 0x114007 ie "0x007" are the flags, and the PPN is 0x114 = 276d ie the 276th page in physical memory.

Q5. What does 7 mean?

Ans. The "7" indicates the flags and when translated to binary is "111". These are the last three bits in the PDE and indicate that the permissions on this PDE are user, present and writable. (Bits 0, 1, 2 of PDE and PTE are present, writable, user flags respectively).

(gdb) print/x ((int *) 0x114000)[0x107]

o/p $12 = 0x107001

Q6. What is this?

Ans. This is the "0x107"th entry in the page that is pointed to by the PA 0x114000. This will serve as the PTE when translating the VA 0x80107beb, after the system initialises kpgdir as the page directory (currently it is running on entrypgdir). The PTE points to the 107th page.

Q7. Why 1 in the low bits?

Ans. These are the flags set to indicate that it is pointing to a page that is present, read-only and accessible only by the kernel. (Least significant bits 0, 1, 2 are present, writable and user flags respectively)

Q8. Why did the physical address work in gdb?

Ans. Currently (ie at the point kvmalloc returns & before calling switchkvm) the system is running on "entrypgdir", not on kpgdir. "entrypgdir" maps the low virtual addresses from 0-4MB identically to Physical addresses 0-4MB since the system booted in the low addresses. Hence only all Virtual addresses b/w 0-4MB are valid at this point.

Q9. Why? (0x107beb won't work)

Ans. The system has switched to kpgdir which has no mappings for the low virtual addresses (ie VA that are less than 2GB). Since 0x107beb is a low virtual address it is no longer accessible once kpgdir is loaded into the CR3 register (after converting to its physical address).

Assignment: Addressing

The bootloader cannot just place the kernel wherever it likes. This is because kernel symbols have been compiled and linked assuming a certain address space where it takes the validity of certain addresses as guaranteed. For instance, after loading the kernel at 2MB in physical memory instead of 1MB, the bootloader jumps to entry.S which would be the address assuming the kernel was loaded at 1MB, since it is obtained from the ELF file. However there is nothing loaded b/w 1-2MB or it is garbage from the kernel standpoint as now the kernel starts from 2MB. Clearly bad things will happen on jumping to entry.S. Even if the "entry" address is offset by 1MB, there will be problems faced later on when the kernel starts to run the "main" fn; since the kernel assumes the bootloader loaded it starting at 1MB.

The kernel will setup its page tables according to that, mapping the virtual addresses to include the 1MB - 2MB region as well. Clearly address translation of the kernel's code (or what it thought were its code) will yield unexpected results.

Assignment :. Traps

Q. Is it possible to have exactly 3 sets of registers on the kstack?

Ans. A process can have three sets of saved registers on its kstack i.e 2 trapframes and one context. This is possible when the process makes a system call (the first trapframe gets pushed), there is a transition to kernel mode. While executing in kernel mode, there is a timer interrupt (second trapframe gets pushed). The kernel continues to execute but now follows a different call chain than the one for the system call, calls the scheduler, which results in the "context" getting pushed, and the kernel switching to a different kstack.

Q. Is it possible to have two "context" structure and one "trapframe"?

Ans. No, such a kstack will never exist. This is because a context structure is only pushed when the process calls switch(). The code in switch will initialize the "context" and then load another kstack. As the context structure gets pushed by the switch function which after pushing the context on the current will switch kstacks and unwind the context on that kstack, it is not possible to have two "context" structures on the kstack.

Q. Is it possible to have two trapframes and one context structure?

Ans. Yes. This is possible if the process makes a system call (first tf gets pushed) followed by a timer interrupt (second tf gets pushed) which then calls switch() which is when "context" gets pushed.

Q. Is it possible to have more than three sets of saved registers?

Ans. Technically it is possible to have more than three sets of saved registers if there are certain external interrupt handlers that run with interrupts enabled. There is a scenario possible in which process executes syscall (first tf gets pushed), then there is an external interrupt (second tf gets pushed) whose handler does not disable interrupts which means when the timer interrupt occurs a third tf gets pushed. The code that follows calls switch() which means context gets pushed, resulting in 4 sets of saved registers. However in order to limit kstack sizes, handlers for external interrupts run with interrupts disabled. Even then there is the possibility that a third tf gets pushed if the kernel code itself gives rise to an exception such as page fault or divide-by-zero, and so more than 3 sets of registers (3 tf + 1 context) will be there. (syscall → exception → timer → switch) so as long as interrupts are disabled for external interrupts and the kernel itself does not give rise to exceptions, there will not be more than three sets of saved registers on the kstack.

Assignment: Context Switching

Q1. The stack that sched executes on is the process ~~that~~ kernel stack that was interrupted and made the switch from user mode to kernel mode.

Q2. The stack that scheduler executes on is the private stack of the CPU the process was executing on. This is a per-CPU stack that does not belong to any process and is used by the scheduler.

Q3. The call to switch does not return immediately. Let's say P1 called switch, and the stacks shift to the scheduler's stack. The scheduler picks P2 and switches from the scheduler stack to P2's Kstack. At some point P2 will call switch, and shift to the scheduler's stack, which will then say pick P1 and switch to P1's Kstack. At this point the call made by P1 to switch returns. In other words the call to switch does return but after few scheduler rounds.

Q4. No it is not possible to reduce the work done by switch or reduce the size of a struct context. Switch as well as context only consists of the callee save registers, which have to be necessarily saved because on a context switch the kernel returns to another thread whose fn call chain could possibly alter those registers. It is however possible to remove "eip" from context because "eip" will be ~~saved~~ pushed automatically by the H/W on calling switch, and if one needs to access it then one know exactly how many registers or bytes above context it is depending on the remaining registers in the context. Say if context is ebp, ebx, esi and edi are in context, then context+16 would yield eip.

Q5. The four character pattern printed is "badc"

Q6. The very first characters are "ac". This is because before any user process gets to run, the kernel is setting itself up. It calls the scheduler, running on the scheduler's stack that does not belong to any process. The scheduler picks the first process to run (that is when "a" gets printed), which has been created by the kernel, and calls switch to transition from the scheduler's stack to the first process's kstack ("the init process"). The first process make an exec system call to the "init" process, which makes more exec system calls. Finally one of them is interrupted

and calls swtch and to print transition from their process's kstack to the scheduler's stack on the CPU they were running on. That is when the scheduler's swtch returns and prints "b".

and calls swtch (that is when "c" gets printed) to transition from the process's kstack to the scheduler's stack which is private to the CPU it is running on.