CIS520 – Operating Systems
Handout 11
MIPS TLB Structure

- Case Study: A simple VM and paging system for the MIPS R3000.
- Start with architecture. Here is the memory hierarchy of the machine at hand:
  - First Level Cache. 64K bytes, direct mapped. Write allocate, write through. Physically addressed.
  - Write Buffer. 4 Entries.
  - Second Level Cache. 256K bytes, direct mapped. Write back.
  - Physical Memory. 64M bytes. 4K page frames.
  - Backing Store Disk. 256M bytes of swap space.
- Address format. Are two modes: user mode and kernel mode. Top 20 bits are Virtual Page Number, bottom 12 bits are page offset. Machine has a 6 bit current process id; process id is part of 38 bit virtual address. All user mode addresses have a top address bit of 0. How big is potential user address space? How big are pages?
- Kernel mode addresses.
  - If address starts with 0, is mapped just like current user process. So, user process address space is subset of OS address space. Called kuseg.
  - If address starts with 100, translates to bottom 512 Mbytes of physical memory and does not go through TLB. (cached and unmapped). Called kseg0. Used for kernel instructions and data.
  - If starts with 101, translates to bottom 512 Mbytes of physical memory. Is not cached (uncached, unmapped). Called kseg1. Used for disk buffers, I/O registers, ROM code.
  - If starts with 11, is mapped and cacheable. Maps differently for each process. Called kseg2. Used for kernel data structures in which there is one per address space - user page tables, etc.
- Specification of mapping process: must map 31 bit virtual address (top bit is always 0) plus 6 bit process id to 32 bit physical address. Do mapping by first mapping upper 19 bits of virtual address plus 6 bit process id to a physical page frame, then using lower 12 bits of virtual address as offset within the physical page frame.
- How do we map upper 19 bits of virtual address? Use a linear page table stored in kseg2. How big can this page table be?
- Note that we will also be paging kseg2 using a linear page table. Where do we hold the page table for kseg2? There is one for each process, and it is stored in kseg0. If the page table for the user process stored in kseg2 takes up most of the used address space, how big can the page table for kseg2 stored in kseg0 be?
- In effect, we have a two level approach. Given a 32 bit virtual address from kuseg, we get the physical address as follows:
  - Extract top 9 bits of address. Use this as an index into that process's kseg2 page table stored in kseg0. The memory in kseg0 is always there and the reference goes unmapped, so there will be no problem will this lookup. We get the physical page frame that holds the relevant part of the page table in kseg2. If the page is not resident, read it back in from disk.

  - Extract middle 10 bits of address. This is the amount you need to index one page of 4 byte physical addresses. Use the middle 10 bits to index the page table in kseg2. This lookup yields a physical page frame in kuseg. If the page is not resident, read it back in from disk. Must make sure that we are accessing the correct memory for the current process id since kseg2 maps differently for each process id.
  - Extract lower 12 bits of address. Use this as an offset into the physical page frame holding the page from kuseg. Read the memory location.

- Why divide the lookup into two stages? So we can page the virtual address space AND page the page table. The page table for virtual address space is stored in the part of kernel address space that is mapped differently for different processes. The page table page table is stored in unmapped but cached kernel memory.
- Seems inefficient - 3 memory accesses for one user memory access. So, speed it up with a TLB. 64 entry fully associative TLB. Each entry maps one virtual address to one physical page frame.
- TLB entry format. Each TLB entry is 64 bits long.
  - Top 20 bits: VPN.
  - Next 6 bits: PID.
  - Next 6 bits: unused.
  - Next 20 bits: Physical page frame.
  - Next bit: N bit. If set, memory access bypasses the cache. If not set, memory access goes through the cache.
  - Next bit: D bit. If set, memory is writeable. If not set, memory is not writeable.
  - Next bit: V bit. If set, entry is valid.
  - Next bit: G bit. If set, TLB does not check PID for translation.
- How does lookup work? Basic idea: match on upper half of TLB entry, use lower half of TLB entry. Can generate three different kinds of TLB misses, each with its own exception handler.
  - UTLB miss - generated when the access is to kuseg and there is no matching mapping loaded into the TLB.
  - TLB miss - generated when the access is to kseg0, kseg1, or kseg2 and there is no mapping loaded into TLB. Also generated when the mapping is loaded into TLB, but valid bit is not set.
  - TLB mod - generated when the mapping is loaded, but access is a write and the D bit is not set.
- Here is the TLB lookup algorithm:
  - If MSB is 1 and in user mode, generate an address error exception.
  - Is there a VPN match? If no, generate a TLB miss exception if MSB is 1, otherwise generate a UTLB miss.
  - Does the PID match or is the global bit set? If no, generate a TLB miss (if MSB is 1) or UTLB miss (if MSB is 0).
  - Is valid bit set? If no, generate a TLB miss.
  - If D bit is not set and the access is a write, generate a TLB mod exception.
  - If N bit is set, access memory, otherwise access cache (which may refer access to memory).
- The PID field allows multiple processes to share the TLB. What if there was no PID field? The PID field is only 6 bits long. What if create more than 64 user processes?
- Manipulating TLB entries. Processor must be able to load new entries into TLB. Basic Mechanism: Two 32 bit TLB registers: TLB EntryHi, TLB EntryLow. Bits are the same as for 64 bit TLB entry. EntryHi register holds current PID that is part of all virtual addresses. Also have an Index register: 6 bits that can be set by software, and a Random register: 6 bit register decremented every clock cycle. Constrained not to point to first 8 entries.

- Can load into TLB entry registers under program control, then store contents of Entry registers either to TLB entry to which index register points, or to which random register points.
- TLB instructions:
  - mtc0 - loads one of TLB registers with contents of a general register.
  - mfc0 - reads one of TLB registers into a general register.
  - tlbp - probes the TLB to see if an entry matches EntryHi. If so, loads index register with index of TLB entry that matched. If no match, sets upper bit of index register.
  - tlbr - loads EntryHi and EntryLow with contents of TLB entry that index register points to.
  - tlbwi - writes TLB entry that index points to with contents of EntryHi and EntryLo registers.
  - tlbwr - writes TLB entry that random register points to with contents of EntryHi and EntryLo registers.
- What happens when there is a UTLB or TLB miss? OS must reload TLB and restart the faulting process. Note - the UTLB and TLB miss exceptions branch to different handlers.
- Machine state for exceptions:
  - EPC register: points to instruction that caused fault, unless faulting instruction was in branch delay slot. If so, points to branch before branch delay slot instruction. Basic idea: when fix up exception and return to user code, will branch to EPC.
  - Cause register. Tells what caused exception, and maintains some state about interrupts.
  - Status register. Contains information about status of machine. Important bits: Kernel/User mode bit, Interrupt Enable bit. OS maintains a 3 deep stack of these bits, shifting them over on an exception. So, can take two exceptions without having to extract and store the bits.
  - BadVaddr register - stores virtual address that caused last exception.
  - Context register. Upper 11 bits - set under program control. Next 19 bits - set to VPN of address that caused exception (omits top bit). Last 2 bits - always 0.
- What does machine do on a UTLB miss?
  - Sets EPC register.
  - Sets Cause register.
  - Sets Status register. Shifts K/U and IE bits over one, and clears current Kernel/User and Interrupt Enable bits. So - processor is in kernel mode with interrupts turned off.
  - Sets BadVaddr register - stores virtual address that caused exception.
  - Sets Context register. Upper 11 bits - left alone. Next 19 bits - set to VPN of address that caused exception (omits top bit).
  - Sets TLB EntryHi register to contain VPN of faulting address.
- What does OS do in UTLB handler?
  - Store EPC register to kt1 register (software convention, OS has two registers reserved for its use).
  - Load context register into kt0 register.
  - Load contents of memory address that kt0 points to into kt0. Into what part of address space does kt0 point?
  - Load kt0 into entry low TLB register.
  - Load TLB entry registers into TLB entry that random register points to.
  - JR kt1; rfe instruction in branch delay slot. rfe instruction pops bits in Status Register.
- What is going on? OS uses a linear page table for each process, starting at address stored in upper 11 bits of context register. Each page table entry is the 32 lower bits of a TLB entry. So, OS just fetched the TLB entry and stored it into a random location in TLB, then started up the program again.

- What are upper two bits of context register? 11 - so, this is kernel memory that is mapped separately for each process. Next 9 bits are base of page table in mapped, process-specific kernel space. So, each process has its own page table.
- Error cases:
  - What if page is not in memory? Then OS will store a zero in the valid bit of page table entry. Program will reexecute faulting instruction, generating a TLB miss exception. (NOT a UTLB miss exception).
  - What if address is out of bounds? OS stores nothing above the page table in address space, so will get a TLB miss (NOT a UTLB miss). This generates a double fault that OS will handle in general exception handler.
  - What if page table page is not mapped or it is not in memory? Another double fault.
  - What if faulting instruction was in a branch delay slot? EPC points to branch, so will reexecute branch instruction. No problem - in R3000, all branch instructions are reexecutable with same effect.
  - What if are inside kernel when take a UTLB miss? How does miss handler know which state to return to? Is stored automatically in Status register, and manipulated by exceptions and rfe instruction.
- R3000 carefully designed to support this efficient UTLB reload mechanism.
  - Some kernel addresses are mapped differently for different processes. So, can store per-process page tables there.
  - TLB entry format laid out so that it matches possible page table entries.
  - All branches are restartable - they do not depend on machine state.
  - UTLB miss handler is in a different location than normal exception handler - supports fast code. Don't have to decode cause of exception.
  - Sets context register appropriately.
  - Supports three levels of Kernel/User mode and interrupt enable/disable bits, so can take two faults in a row without needing to save state. Supports double fault mechanism for fast handling of uncommon cases.
- What must OS do when it switches contexts? Must set EntryHi of TLB to contain current process id. Must also load top 11 bits of context register with page table base.
- What happens on a TLB miss (as opposed to a UTLB miss).
  - Sets cause register - can be TLB mod miss, or TLB miss. TLB mod miss is when TLB entry matches but operation was a store and D bit was not set.
  - Sets BadVaddr register.
  - Sets EPC.
  - Shifts bits in Status register.
  - Sets context register.
  - Sets TLB EntryHi register.
  - Branches to general exception handler (different from UTLB miss handler).
- What OS does on TLB miss:
  - First determine what caused miss.
  - If TLB mod miss, check to see if process has right to write page. Usually stored in page table entry in one of unused bits. If has can write it, mark physical page as dirty in OS data structures, set D bit in TLB entry and restart process. Note: can't use random register to write TLB entry back in. There is already a TLB entry with the matching VPN and PID. Must use tlbp to load index register with the index of the matching entry, then store new page table entry into Entry Low register, then tlbwi to store new TLB entry back into the TLB.

  - If TLB caused by double miss from UTLB miss handler, (find this out by seeing of EPC points inside UTLB miss handler), first determine if given address is valid. Determine this by checking if it is within range of process virtual address space. Then determine if page table page is resident. If so, construct mapping for page table page and put it into TLB. Use this mapping to get TLB entry for page. Insert this entry into TLB and return to user program.
  - If page table page not resident (this is for a normal kernel TLB miss in per-process kernel space), read it in from disk. When page arrives, set up page table entry. Set valid bit, make PFN point to page frame where it was read in. Clear D bit. Map page table page into TLB and proceed as above.
  - If TLB caused by kernel mode reference not mapped, find the PTE for the kernel address and put it into TLB, using entry hi and lo registers and random register. Return to code that caused miss.
  - If miss caused by reference to invalid page in user address space, read page in from disk, set valid bit and PFN in page table. Also, be sure to clear D bit so that any write reference will cause a trap. The OS will use this trap to mark the PFN dirty.

- Alternatives:
  - The OS may run completely unmapped. Problem: can't page OS data structures.
  - The OS may have a separate address space from user. Problem: can't as easily access user space.
  - Cache may be virtually addressed. Problem: can't map same memory in different addresses in same process. How do implement Unix mmap facility, which demands that different virtual addresses map to same physical address? Also, if cache does not have PID field, must flush cache on context switch (!).
  - TLB may not have PID field. Problem: must flush TLB on context switch.
  - TLB reload may be done automatically in hardware. Each page table entry is 32 bits (lower 32 bits of TLB entry above), and hardware will automatically reload TLB. Context switch must load page table base and bounds registers. Problem: locks OS into using a specific data structure for page table.