Goals for Today

- Paging/Memory management (con’t)
- Devices and Device Drivers

Interactive is important!
Ask Questions!

Recall: R3000 pipeline includes TLB “stages”

MIPS R3000 Pipeline

<table>
<thead>
<tr>
<th>Instruction Fetch</th>
<th>Dcd/Reg</th>
<th>ALU / E.A</th>
<th>Memory</th>
<th>Write Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>I-Cache</td>
<td>RF</td>
<td>Operation</td>
<td>WB</td>
</tr>
<tr>
<td>E.A.</td>
<td>TLB</td>
<td>D-Cache</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TLB
- 64 entry, on-chip, fully associative, software TLB fault handler

Virtual Address Space

<table>
<thead>
<tr>
<th>ASID</th>
<th>V. Page Number</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>20</td>
<td>12</td>
</tr>
</tbody>
</table>

- 0x User segment (caching based on PT/TLB entry)
- 100 Kernel physical space, cached
- 101 Kernel physical space, uncached
- 11x Kernel virtual space

Allows context switching among 64 user processes without TLB flush

Reducing translation time further

- As described, TLB lookup is in serial with cache lookup:

Virtual Address

- Machines with TLBs go one step further: they overlap TLB lookup with cache access.
  - Works because offset available early
Overlapping TLB & Cache Access

- Here is how this might work with a 4K cache:

  - What if cache size is increased to 8KB?
    - Overlap not complete
    - Need to do something else. See CS152/252
  - Another option: Virtual Caches
    - Tags in cache are virtual addresses
    - Translation only happens on cache misses

Modern Pipelines: SandyBridge Pipeline

- x86 instructions turned into micro-ops
  - Cached translations are reused
- Branch prediction
  - Not entirely clear, but seems to have some combination of bi-mode, multi-level BTB, stack-based prediction for CALL/RETURN
- Predecoder
  - Finds instruction boundaries
  - Passes at least 6 instructions onto decoding infrastructure

Out-of-Order execution: Data TLB (DTLB)

- Unified Reservation Unit
  - Full OOO execution
  - Pick 6 ready µops/cycle
  - Can have two loads or stores/cycle
    - 2 address generation units (AGUs) + store data
  - Simultaneous 256-bit Multiply and Add
  - Can have 3 regular integer ops/cycle
Use of Mapping as a Cache: Demand Paging

- Modern programs require a lot of physical memory
  - Memory per system growing faster than 25%-30%/year
- But they don’t use all their memory all of the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user’s code to be in memory
- Solution: use main memory as cache for disk

![Diagram of memory hierarchy]

- Disk is larger than physical memory ⇒
  - In-use virtual memory can be bigger than physical memory
  - Combined memory of running processes much larger than physical memory
    - More programs fit into memory, allowing more concurrency
- Principle: **Transparent Level of Indirection** (page table)
  - Supports flexible placement of physical data
    - Data could be on disk or somewhere across network
  - Variable location of data transparent to user program
    - Performance issue, not correctness issue

### Review: What is in a PTE?

- **What is in a Page Table Entry (or PTE)?**
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- **Example: Intel x86 architecture PTE:**
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called “Directories”

<table>
<thead>
<tr>
<th>Page Frame Number (Physical Page Number)</th>
<th>Free (OS)</th>
<th>O</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>T</th>
<th>L</th>
<th>U</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td></td>
<td>11-9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
- **P:** Present (same as “valid” bit in other architectures)
- **W:** Writeable
- **U:** User accessible
- **PWT:** Page write transparent: external cache write-through
- **PCD:** Page cache disabled (page cannot be cached)

- **A:** Accessed: page has been accessed recently
- **D:** Dirty (PTE only): page has been modified recently
- **L:** L=1⇒4MB page (directory only)
  - Bottom 22 bits of virtual address serve as offset

- **PTE helps us implement demand paging**
  - **Valid** ⇒ Page in memory, PTE points at physical page
  - **Not Valid** ⇒ Page not in memory; use info in PTE to find it on disk when necessary

- **Suppose user references page with invalid PTE?**
  - Memory Management Unit (MMU) traps to OS
    - Resulting trap is a “Page Fault”
      - Choose an old page to replace
      - If old page modified (“D=1”), write contents back to disk
      - Change its PTE and any cached TLB to be invalid
      - Load new page into memory from disk
      - Update page table entry, invalidate TLB for new entry
      - Continue thread from original faulting location

- **TLB for new page will be loaded when thread continued!**
- **While pulling pages off disk for one process, OS runs another process from ready queue**
  - Suspended process sits on wait queue
Administrivia

- Will get Exams back this week
  - Sorry for the delay!
- Final decision on Lab 2:
  - Will allow people to turn in functioning Lab 2 for extra credit (post-curve bump in grade)
  - Considering giving back slip days burned on Lab 2
    » Ok with people?
- Lab 3 – Posted
  - Get started!
  - I will adjust deadlines on the lecture page

Transparent Exceptions

- How to transparently restart faulting instructions?
  - Could we just skip it?
    » No: need to perform load or store after reconnecting physical page
- Hardware must help out by saving:
  - Faulting instruction and partial state
    » Need to know which instruction caused fault
    » Is single PC sufficient to identify faulting position????
  - Processor State: sufficient to restart user thread
    » Save/restore registers, stack, etc
- What if an instruction has side-effects?

Consider weird things that can happen

- What if an instruction has side effects?
  - Options:
    » Unwind side-effects (easy to restart)
    » Finish off side-effects (messy!)
  - Example 1: mov (sp)+, 10
    » What if page fault occurs when write to stack pointer?
    » Did sp get incremented before or after the page fault?
  - Example 2: strcpy (r1), (r2)
    » Source and destination overlap: can’t unwind in principle!
    » IBM S/370 and VAX solution: execute twice – once read-only
- What about “RISC” processors?
  - For instance delayed branches?
    » Example: bne somewhere
      ld r1, (sp)
    » Precise exception state consists of two PCs: PC and nPC
  - Delayed exceptions:
    » Example: div r1, r2, r3
      ld r1, (sp)
    » What if takes many cycles to discover divide by zero, but load has already caused page fault?

Precise Exceptions

- Precise \(\Rightarrow\) state of the machine is preserved as if program executed up to the offending instruction
  - All previous instructions completed
  - Offending instruction and all following instructions act as if they have not even started
  - Same system code will work on different implementations
  - Difficult in the presence of pipelining, out-of-order execution, ...
  - MIPS takes this position
- Imprecise \(\Rightarrow\) system software has to figure out what is where and put it all back together
- Performance goals often lead designers to forsake precise interrupts
  - System software developers, user, markets etc. usually wish they had not done this
- Modern techniques for out-of-order execution and branch prediction help implement precise interrupts
Steps in Handling a Page Fault

Demand Paging Example

- Since Demand Paging like caching, can compute average access time! ("Effective Access Time")
  - \( EAT = \text{Hit Rate} \times \text{Hit Time} + \text{Miss Rate} \times \text{Miss Time} \)
  - \( EAT = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty} \)

- Example:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
  - Suppose \( p = \text{Probability of miss} \), \( 1-p = \text{Probability of hit} \)
  - Then, we can compute \( EAT \) as follows:
    \[
    EAT = 200\text{ns} + p \times 8 \text{ ms}
    \]
    \[
    = 200\text{ns} + p \times 8,000,000\text{ns}
    \]
  - If one access out of 1,000 causes a page fault, then \( EAT = 8.2 \mu\text{s} \):
    - This is a slowdown by a factor of 40!
  - What if want slowdown by less than 10%?
    - \( 200\text{ns} \times 1.1 < EAT \Rightarrow p < 2.5 \times 10^{-6} \)
    - This is about 1 page fault in 400000!

What Factors Lead to Misses?

- Compulsory Misses:
  - Pages that have never been paged into memory before
  - How might we remove these misses?
    - Prefetching: loading them into memory before needed
    - Need to predict future somehow! More later.

- Capacity Misses:
  - Not enough memory. Must somehow increase size.
  - Can we do this?
    - One option: Increase amount of DRAM (not quick fix!)
    - Another option: If multiple processes in memory: adjust percentage of memory allocated to each one!

- Conflict Misses:
  - Technically, conflict misses don’t exist in virtual memory, since it is a “fully-associative” cache

- Policy Misses:
  - Caused when pages were in memory, but kicked out prematurely because of the replacement policy
  - How to fix? Better replacement policy

Page Replacement Policies

- Why do we care about Replacement Policy?
  - Replacement is an issue with any cache
  - Particularly important with pages
    - The cost of being wrong is high: must go to disk
    - Must keep important pages in memory, not toss them out

- FIFO (First In, First Out)
  - Throw out oldest page. Be fair – let every page live in memory for same amount of time.
  - Bad, because throws out heavily used pages instead of infrequently used pages

- MIN (Minimum):
  - Replace page that won’t be used for the longest time
  - Great, but can’t really know future...
  - Makes good comparison case, however

- RANDOM:
  - Pick random page for every replacement
  - Typical solution for TLB’s. Simple hardware
  - Pretty unpredictable – makes it hard to make real-time guarantees
Replacement Policies (Con't)

- LRU (Least Recently Used):
  - Replace page that hasn't been used for the longest time.
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.

- How to implement LRU? Use a list!
  - On each use, remove page from list and place at head
  - LRU page is at tail

- Problems with this scheme for paging?
  - Need to know immediately when each page used so that can change position in list...
  - Many instructions for each hardware access

- In practice, people approximate LRU (more later)

Implementing LRU

- Perfect:
  - Timestamp page on each reference
  - Keep list of pages ordered by time of reference
  - Too expensive to implement in reality for many reasons

- Clock Algorithm: Arrange physical pages in circle with single clock hand
  - Approximate LRU (approx to approx to MIN)
  - Replace an old page, not the oldest page

- Details:
  - Hardware “use” bit per physical page:
    » Hardware sets use bit on each reference
    » If use bit isn't set, means not referenced in a long time
    » Nachos hardware sets use bit in the TLB; you have to copy this back to page table when TLB entry gets replaced
  - On page fault:
    » Advance clock hand (not real time)
    » Check use bit: 1=used recently; clear and leave alone
      0=selected candidate for replacement
    - Will always find a page or loop forever?
      » Even if all use bits set, will eventually loop around ⇒ FIFO

Clock Algorithm: Not Recently Used

- Set of all pages in Memory
- Single Clock Hand: Advances only on page fault!
  Check for pages not used recently
  Mark pages as not used recently

- What if hand moving slowly?
  - Good sign or bad sign?
    » Not many page faults and/or find page quickly
- What if hand is moving quickly?
  - Lots of page faults and/or lots of reference bits set
- One way to view clock algorithm:
  - Crude partitioning of pages into two groups: young and old
  - Why not partition into more than 2 groups?

N\textsuperscript{th} Chance version of Clock Algorithm

- N\textsuperscript{th} chance algorithm: Give page N chances
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks use bit:
    » 1⇒clear use and also clear counter (used in last sweep)
    » 0⇒increment counter; if count=N, replace page
  - Means that clock hand has to sweep by N times without page being used before page is replaced

- How do we pick N?
  - Why pick large N? Better approx to LRU
    » If N ~ 1K, really good approximation
  - Why pick small N? More efficient
    » Otherwise might have to look a long way to find free page

- What about dirty pages?
  - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
  - Common approach:
    » Clean pages, use N=1
    » Dirty pages, use N=2 (and write back to disk when N=1)
Clock Algorithms: Details

- Which bits of a PTE entry are useful to us?
  - **Use**: Set when page is referenced; cleared by clock algorithm.
  - **Modified**: Set when page is modified, cleared when page written to disk.
  - **Valid**: Ok for program to reference this page.
  - **Read-only**: Ok for program to read page, but not modify.
    » For example for catching modifications to code pages!

- Do we really need hardware-supported “modified” bit?
  - No. Can emulate it (BSD Unix) using read-only bit.
    » Initially, mark all pages as read-only, even data pages.
    » On write, trap to OS. OS sets software “modified” bit, and marks page as read-write.
    » Whenever page comes back in from disk, mark read-only.

- Do we really need a hardware-supported “use” bit?
  - No. Can emulate it similar to above:
    » Mark all pages as invalid, even if in memory.
    » On read to invalid page, trap to OS.
    » On write, trap to OS (either invalid or read-only).
    » Set use and modified bits, mark page read-write.
  - Get modified bit in same way as previous:
    » On write, trap to OS (either invalid or read-only).
    » When clock hand passes by, reset use and modified bits and mark page as invalid again.

- Remember, however, that clock is just an approximation of LRU.
  - Can we do a better approximation, given that we have to take page faults on some reads and writes to collect use information?
    » Need to identify an old page, not oldest page!
    » Answer: second chance list.

Second-Chance List Algorithm (VAX/VMS)

- Split memory in two: Active list (RW), SC list (Invalid).
- Access pages in Active list at full speed.
- Otherwise, Page Fault.
  - Always move overflow page from end of Active list to front of Second-chance list (SC) and mark invalid.
  - Desired Page On SC List: move to front of Active list, mark RW.
  - Not on SC list: page in to front of Active list, mark RW; page out LRU victim at end of SC list.

- How many pages for second chance list?
  - If 0 ⇒ FIFO
  - If all ⇒ LRU, but page fault on every page reference.
- Pick intermediate value. Result is:
  - Pro: Few disk accesses (page only goes to disk if unused for a long time).
  - Con: Increased overhead trapping to OS (software / hardware tradeoff).
- With page translation, we can adapt to any kind of access the program makes.
  - Later, we will show how to use page translation / protection to share memory between threads on widely separated machines.
- Question: why didn’t VAX include “use” bit?
  - Strecker (architect) asked OS people, they said they didn’t need it, so didn’t implement it.
  - He later got blamed, but VAX did OK anyway.
Free List

- Keep set of free pages ready for use in demand paging
  - Freelist filled in background by Clock algorithm or other technique ("Pageout demon")
  - Dirty pages start copying back to disk when enter list
- Like VAX second-chance list
  - If page needed before reused, just return to active set
- Advantage: Faster for page fault
  - Can always use page (or pages) immediately on fault

Reverse Page Mapping (Sometimes called "Coremap")

- Physical page frames often shared by many different address spaces/page tables
  - All children forked from given process
  - Shared memory pages between processes
- Whatever reverse mapping mechanism that is in place must be very fast
  - Must hunt down all page tables pointing at given page frame when freeing a page
  - Must hunt down all PTEs when seeing if pages "active"
- Implementation options:
  - For every page descriptor, keep linked list of page table entries that point to it
    » Management nightmare - expensive
  - Linux 2.6: Object-based reverse mapping
    » Link together memory region descriptors instead (much coarser granularity)

What Actually Happens in Linux?

- Memory management in Linux considerably more complex that the previous indications
- Memory Zones: physical memory categories
  - ZONE_DMA: < 16MB memory, DMAable on ISA bus
  - ZONE_NORMAL: 16MB ⇒ 768GB (mapped at 0xC0000000)
  - ZONE_HIGMEM: Everything else (> 768GB)
- Each zone has 1 freelist, 2 LRU lists (Active/Inactive)
- Many different types of allocation
  - SLAB allocators, per-page allocators, mapped/unmapped
- Many different types of allocated memory:
  - Anonymous memory (not backed by a file, heap/stack)
  - Mapped memory (backed by a file)
- Allocation priorities
  - Is blocking allowed/etc

Linux Virtual memory map

- 32-Bit Virtual Address Space
- 64-Bit Virtual Address Space
- Kernel Addresses
- User Addresses
- Empty Space
- "Canonical Hole"
Virtual Map (Details)

- Kernel memory not generally visible to user
  - Exception: special VDSO facility that maps kernel code into user space to aid in system calls (and to provide certain actual system calls such as gettimeofday).
- Every physical page described by a “page” structure
  - Collected together in lower physical memory
  - Can be accessed in kernel virtual space
  - Linked together in various “LRU” lists
- For 32-bit virtual memory architectures:
  - When physical memory < 896GB
    » All physical memory mapped at 0xC0000000
  - When physical memory >= 896GB
    » Not all physical memory mapped in kernel space all the time
    » Can be temporarily mapped with addresses > 0xCC000000
- For 64-bit virtual memory architectures:
  - All physical memory mapped above 0xFFFF800000000000

Allocating Memory

- One mechanism for requesting pages: everything else on top of this mechanism:
  - Allocate contiguous group of pages of size $2^{order}$ bytes given the specified mask:
    ```c
    struct page * alloc_pages(gfp_t gfp_mask, 
    unsigned in order)
    ```
  - Allocate one page:
    ```c
    struct page * alloc_page(gfp_t gfp_mask)
    ```
  - Convert page to logical address (assuming mapped):
    ```c
    void * page_address(struct page *page)
    ```
- Also routines for freeing pages
- Zone allocator uses “buddy” allocator that tries to keep memory unfragmented
- Allocation routines pick from proper zone, given flags

Allocation flags

- Possible allocation type flags:
  - GFP_ATOMIC: Allocation high-priority and must never sleep. Use in interrupt handlers, bottom halves, while holding locks, or other times cannot sleep
  - GFP_NOWAIT: Like GFP_ATOMIC, except call will not fall back on emergency memory pools. Increases likely hood of failure
  - GFP_NOIO: Allocation can block but must not initiate disk I/O.
  - GFP_NOFS: Can block, and can initiate disk I/O, but will not initiate filesystem ops.
  - GFP_KERNEL: Normal allocation, might block. Use in process context when safe to sleep. This should be default choice
  - GFP_USER: Normal allocation for processes
  - GFP_HIGHUSER: Allocation from ZONE_HIGHMEM
  - GFP_DMA: Allocation from ZONE_DMA. Use in combination with a previous flag

Slab Allocator

- Replacement for typical free-lists that are hand-coded by users
  - Consolidation of all of this code under kernel control
  - Efficient when objects allocated and freed frequently
- Objects segregated into “Caches”
  - Each cache stores different type of object
  - Data inside cache divided into “slabs”, which are continuous groups of pages (often only 1 page)
  - Key idea: avoid memory fragmentation
- When allocate from cache, chunk of memory is returned of requested size
  - Multiple objects will fit in each slab
  - Cache keeps track of which portions of slab are free
- Interface:
  ```c
  void * kem_cache_alloc(struct kmem_cache *cachep, 
  gfp_t flags)
  ```
Summary (1/2)

- **PTE**: Page Table Entries
  - Includes physical page number
  - Control info (valid bit, writeable, dirty, user, etc)
- A cache of translations called a “Translation Lookaside Buffer” (TLB)
  - Relatively small number of entries (< 512)
  - Fully Associative (Since conflict misses expensive)
  - TLB entries contain PTE and optional process ID
- On TLB miss, page table must be traversed
  - If located PTE is invalid, cause Page Fault
- On context switch/change in page table
  - TLB entries must be invalidated somehow
- TLB is logically in front of cache
  - Thus, needs to be overlapped with cache access to be really fast

Summary (2/2)

- **Clock Algorithm**: Approximation to LRU
  - Arrange all pages in circular list
  - Sweep through them, marking as not “in use”
    - If page not “in use” for one pass, than can replace
- **Nth-chance clock algorithm**: Another approx LRU
  - Give pages multiple passes of clock hand before replacing
- **Second-Chance List algorithm**: Yet another approx LRU
  - Divide pages into two groups, one of which is truly LRU and managed on page faults
- **Reverse Page Mapping**
  - Efficient way to hunt down all PTEs associated with given page frame