Concurrency, Atomic Sequences, Virtual Memory

Computer Science 104
Lecture 20

Admin
- Homework #5 due Wednesday
- Projects April 20

Reading
Finish Chapter 7
Start Chapter Input Output 8: 8.1, 8.2 (only first three pages), 8.4, 8.5

Review: System Organization

Review: OS, Exceptions
- Operating System
- Threads of Control
  > execution context
  > context switches
- Kernel vs. User Mode
  > mode bit, privileged instructions
- Interrupts, Exceptions and System Calls
  > what are they?
  > handling exceptions
  > extending the architecture
  > implementing control
- Concurrency
- Synchronization and Atomic Sequences

Computer Architecture
- Interface Between Hardware and Software
  - This is IT

Concurrency
- Multiple things happening simultaneously
  > logically or physically
- Causes
  > Interrupts
  > Voluntary context switch (system call/trap)
  > Hyperthreading / multicore / multiprocessor
The Trouble with Concurrency

- Two threads (T1, T2) in one address space or two processes in the kernel
- One counter (e.g., profile string_copy)

```
ld r2, count
add r1, r2, r3
st count, r1
```

Private stack per thread for locals

Shared Data

```
ld r2, count
add r1, r2, r3
st count, r1
```

Solution: Atomic Sequence of Instructions

```
begin atomic
  ld (count)
  add
  st (count+1)
end atomic
```

• Atomic Sequence
  - Appears to execute to completion without any intervening operations

HW Support for Atomic Operations

- Could provide direct support in HW
  - Atomic increment
  - Insert node into sorted list??
- Just provide low level primitives to construct atomic sequences
  - called synchronization primitives
    - LOCK(counter->lock); // begin atomic
    - counter->value = counter->value + 1;
    - UNLOCK(counter->lock); // end atomic
  - All updates to counter->value must be “protected” by Lock/Unlock
- test&set (x) instruction: returns previous value of x and sets x to “1”
  - LOCK(x) => while (test&set(x));
  - UNLOCK(x) => x = 0;

Solution: Atomic Sequence of Instructions

```
begin atomic
  ld (count)
  add
  switch
  st (count+1)
end atomic
```

• Disable interrupts
  - No context switching
  - Only works for single threaded systems (not on hyperthreading or multicore)

Transactions

- Mark begin and end of transaction
- If no conflict during transaction great
- If conflict, then need to make it look like the instructions never executed and start over
- Need support to either 1) undo operations or 2) delay updating state until transaction is guaranteed to be atomic

Transactions

```
begin trans
  ld (count)
  add
  st (count+1)
end trans
```

Conflict: Neither transaction can complete
Both will restart, eventually one completes without a conflicting Access (e.g., runs and completes without a context switch)
Review: Computer Memory

- Memory is a large linear array of bytes.
- 32-bit or 64-bit addresses

Review: Memory Hierarchy 101

Very fast 1ns clock
Multiple Instructions per cycle

<table>
<thead>
<tr>
<th>Memory</th>
<th>SRAM, Fast, Small</th>
<th>Expensive</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM, Slow, Big, Cheap</td>
<td>(called physical or main memory)</td>
<td>~100MB to GB</td>
</tr>
</tbody>
</table>

$\Rightarrow$ Cost Effective Memory System (Price/Performance)
You should know how to draw block diagram of cache (fig 7.17) and how to compute miss ratios, memory stall cycles, CPU time

Any problems with this picture?

Review: A Simple Program's Memory Layout

- What are the possible addresses generated by the program?
- How big is our DRAM?
- Is there more than one program running?
- If so, how do we allocate memory to each?

Virtual Memory

- Process = virtual address space + thread of control
- Translation
  - VA -> PA
  - What physical address does virtual address A map to
  - Is VA in physical memory?
- Protection (access control)
  - Do you have permission to access it?
Paged Virtual Memory

- Virtual address \(2^{22}, 2^{24}\) to Physical Address mapping \(2^{28}\)
  > virtual page to physical page frame
- Fixed size units for access control & translation

Virtual and Physical Memories

Virtual Memory

<table>
<thead>
<tr>
<th>Page 0</th>
<th>Page 1</th>
<th>Page 2</th>
<th>Page 3</th>
</tr>
</thead>
</table>

Physical Memory

<table>
<thead>
<tr>
<th>Frame 0</th>
<th>Frame 1</th>
<th>Frame 2</th>
<th>Frame 3</th>
</tr>
</thead>
</table>

Disk

<table>
<thead>
<tr>
<th>Page 2</th>
<th>Page 3</th>
<th>Page N-2</th>
<th>Page N-1</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Page N</th>
</tr>
</thead>
</table>

Virtual Memory: Questions

- How is data found if it is in physical memory?
- Where can data be placed in physical memory?
  - Fully Associative, Set Associative, Direct Mapped
- What data should be replaced on a miss?
- How do you handle writes?

Address Mapping

- Operating System (Kernel) data structure (per process)
- Page Table Entry (PTE)
  > VA \(\rightarrow\) PA translations
  > Valid bit = 1 if valid translation, else "page fault exception"
  > access rights (Read, Write, Execute, User/Kernel, cached/uncached)
  > operating system modifies these, not general programs \(\rightarrow\) mode
  > reference, dirty bits (help when replacing a page)
- Many designs
  > Linear, Forward mapped, Inverted, Hashed, Clustered
- Design Issue
  > time to obtain translation

Page Table

- Operating System (Kernel) data structure (per process)
- Page Table Entry (PTE)
  > VA \(\rightarrow\) PA translations
  > Valid bit = 1 if valid translation, else "page fault exception"
  > access rights (Read, Write, Execute, User/Kernel, cached/uncached)
  > operating system modifies these, not general programs \(\rightarrow\) mode
  > reference, dirty bits (help when replacing a page)
- Many designs
  > Linear, Forward mapped, Inverted, Hashed, Clustered
- Design Issue
  > time to obtain translation

Address Mapping Algorithm

- If Valid = 1
  > then page is in main memory at frame address stored in table
  > else page is in secondary storage
- Page Fault
  > page not resident in physical memory (Valid = 0)
  > causes an exception
  > operating system initiates fetch from secondary storage
  > usually accompanied by a context switch: current process suspended while page is fetched from secondary storage
- Access Control
  > R = Read-only, R/W = read/write, X = execute only
  > Protection Fault
  > access control violation
  > access type (load/store) not allowed by specified access rights (read/write)
Choosing a Page Size

What if page is too small?
- Too many misses
- BIG page tables

What if page is too big?
- Fragmentation
  - don't use all of the page, but can't use that DRAM for other pages
  - want to minimize fragmentation (get good utilization of physical memory)
- Smaller page tables
  - Trend is, slowly, toward larger pages
  - increasing gap between CPU/DRAM/DISK

Choosing a Page Size

What if page is too small?
- Too many misses
- BIG page tables

What if page is too big?
- Fragmentation
  - don't use all of the page, but can't use that DRAM for other pages
  - want to minimize fragmentation (get good utilization of physical memory)
- Smaller page tables
  - Trend is, slowly, toward larger pages
  - increasing gap between CPU/DRAM/DISK

Choosing a Page Size

What if page is too small?
- Too many misses
- BIG page tables

What if page is too big?
- Fragmentation
  - don't use all of the page, but can't use that DRAM for other pages
  - want to minimize fragmentation (get good utilization of physical memory)
- Smaller page tables
  - Trend is, slowly, toward larger pages
  - increasing gap between CPU/DRAM/DISK

Choosing a Page Size

What if page is too small?
- Too many misses
- BIG page tables

What if page is too big?
- Fragmentation
  - don't use all of the page, but can't use that DRAM for other pages
  - want to minimize fragmentation (get good utilization of physical memory)
- Smaller page tables
  - Trend is, slowly, toward larger pages
  - increasing gap between CPU/DRAM/DISK

Choosing a Page Size

What if page is too small?
- Too many misses
- BIG page tables

What if page is too big?
- Fragmentation
  - don't use all of the page, but can't use that DRAM for other pages
  - want to minimize fragmentation (get good utilization of physical memory)
- Smaller page tables
  - Trend is, slowly, toward larger pages
  - increasing gap between CPU/DRAM/DISK

Choosing a Page Size

What if page is too small?
- Too many misses
- BIG page tables

What if page is too big?
- Fragmentation
  - don't use all of the page, but can't use that DRAM for other pages
  - want to minimize fragmentation (get good utilization of physical memory)
- Smaller page tables
  - Trend is, slowly, toward larger pages
  - increasing gap between CPU/DRAM/DISK

Choosing a Page Size

What if page is too small?
- Too many misses
- BIG page tables

What if page is too big?
- Fragmentation
  - don't use all of the page, but can't use that DRAM for other pages
  - want to minimize fragmentation (get good utilization of physical memory)
- Smaller page tables
  - Trend is, slowly, toward larger pages
  - increasing gap between CPU/DRAM/DISK

Choosing a Page Size

What if page is too small?
- Too many misses
- BIG page tables

What if page is too big?
- Fragmentation
  - don't use all of the page, but can't use that DRAM for other pages
  - want to minimize fragmentation (get good utilization of physical memory)
- Smaller page tables
  - Trend is, slowly, toward larger pages
  - increasing gap between CPU/DRAM/DISK

Choosing a Page Size

What if page is too small?
- Too many misses
- BIG page tables

What if page is too big?
- Fragmentation
  - don't use all of the page, but can't use that DRAM for other pages
  - want to minimize fragmentation (get good utilization of physical memory)
- Smaller page tables
  - Trend is, slowly, toward larger pages
  - increasing gap between CPU/DRAM/DISK

Choosing a Page Size

What if page is too small?
- Too many misses
- BIG page tables

What if page is too big?
- Fragmentation
  - don't use all of the page, but can't use that DRAM for other pages
  - want to minimize fragmentation (get good utilization of physical memory)
- Smaller page tables
  - Trend is, slowly, toward larger pages
  - increasing gap between CPU/DRAM/DISK
Translation Lookaside Buffers (TLB)

- Need to perform address translation on every memory reference
  - 30% of instructions are memory references
  - at least one memory reference per cycle on today’s processors
- Make Common Case Fast, others correct
- If you just accessed a virtual page and hence determined its physical frame, you’re likely to need that same translation again in the near future
  - LOCALITY!
- Throw hardware at the problem
- Cache PTEs

Fast Translation: Translation Buffer

- Cache of translated addresses
- 64 entry fully associative

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Number</td>
<td>phys frame</td>
</tr>
<tr>
<td>Page Offset</td>
<td>1 2 tag</td>
</tr>
<tr>
<td>offset</td>
<td>2 3</td>
</tr>
<tr>
<td>64x1 mux</td>
<td>4</td>
</tr>
</tbody>
</table>

Hardware Managed TLBs

- Hardware Handles TLB miss
- Dictates page table organization
- Complicated state machine to “walk page table”
  - Multiple levels for forward mapped
  - Linked list for inverted
- Exception only if access violation

Software Managed TLBs

- Software Handles TLB miss
  - OS reads translations from Page Table and puts them in TLB
  - special instructions
- Flexible page table organization
- Simple Hardware to detect Hit or Miss
- Exception if TLB miss or access violation

TLB Design

- Must be fast, not increase critical path
- Must achieve high hit ratio
- Generally small # of entries with high associative
- Mapping change
  - page removed from physical memory
  - processor must invalidate the TLB entry (special privileged instructions)
- PTE is per process entity
  - Multiple processes with same virtual addresses
  - Context Switches?
- Flush TLB
- Add Address space identifier (ASID or PID)
  - part of processor state, must be set on context switch

Virtual Memory

- Provides illusion of very large memory
  - Sum of the memory of many jobs greater than physical memory
  - Address space of each job larger than physical memory
- Good utilization of available physical memory.
- Simplifies memory management: code and data movement, protection, ...
  - (main reason today)
- Exploits memory hierarchy to keep average access time low.
  - Involves at least two storage levels: main and secondary

Virtual Address -- address used by the programmer
Virtual Address Space -- collection of such addresses
Memory Address -- address in physical memory also known as “physical address” or “real address"
Paged Virtual Memory: Main Idea
- Divide memory (virtual and physical) into fixed size blocks (Pages, Frames).
  - Pages in Virtual space.
  - Frames in Physical space.
- Make page size a power of 2: \((\text{page size} = 2^k)\)
- All pages in the virtual address space are contiguous.
- Pages can be mapped into any physical Frame
- Some pages in main memory (DRAM), some pages on secondary memory (disk).

Segmented Virtual Memory
- Virtual address \((2^{32}, 2^{40})\) to Physical Address mapping \((2^{28})\)
- Each segment
  - variable size
  - Address = base + offset
  - contiguous in both VA and PA

Paged Virtual Memory: Main Idea (Cont)
- All programs are written using Virtual Memory Address Space.
- The hardware does on-the-fly translation between virtual and physical address spaces.
- Use a Page Table to translate between Virtual and Physical addresses
- Translation Lookaside Buffer (TLB) expedites address translation
- Must select “good” page size to minimize fragmentation

Intel Pentium Segmentation
- Virtual Address
- Physical Address Space

Intel Pentium Segmentation + Paging
- Virtual Address
- Linear Address Space
- Physical Address Space

Mapping the Kernel
- Digital Unix \(\text{kseg}\)
  - \(\text{kseg}\) (bit 63 = 1, 62 = 0)
- Kernel has direct access to physical memory
- One VA->PA mapping for entire Kernel
- Lock (pin) TLB entry
  - or special HW detection
Summary and Next Time

Summary
- Virtual Memory provides illusion of large contiguous address space
- Protection, code and data movement
- Address Translation
  - Page Table
  - Translation lookaside buffer (TLB)

Next Time
- Virtual Memory & Caches
- Start I/O