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
Computer Architecture

- Interface Between Hardware and Software

Applications

- Operating System
- Compiler

CPU  Memory  I/O
- Multiprocessor  Networks

This is IT

Hardware

Review: System Organization

Processor
- Cache
- Memory Bus
- Core Chip Set

I/O Bridge

Memory
- Disk Controller
- Graphics Controller
- Network Interface

Exceptions

Interrupts

© Alvin R. Lebeck
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

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`)

Solution: Atomic Sequence of Instructions

- **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

```
<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LOCK(ctr_lock)</td>
<td>ld (count)</td>
<td>add</td>
</tr>
<tr>
<td></td>
<td>switch</td>
<td>st (count+1)</td>
<td>LOCK(ctr_lock)</td>
</tr>
<tr>
<td></td>
<td>UNLOCK(ctr_lock)</td>
<td>switch</td>
<td>ld (count)</td>
</tr>
</tbody>
</table>

LOCK(x) => while (test&set(x));
UNLOCK(x) => x = 0;
```
### Alternative Solutions

- **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

<table>
<thead>
<tr>
<th>Time</th>
<th>Failure</th>
<th>Success</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td></td>
<td>begin trans</td>
<td>begin trans</td>
</tr>
<tr>
<td></td>
<td>ld (count)</td>
<td>ld (count)</td>
</tr>
<tr>
<td></td>
<td>add</td>
<td>add</td>
</tr>
<tr>
<td></td>
<td>switch</td>
<td>switch</td>
</tr>
<tr>
<td></td>
<td>st (count+1)</td>
<td>st (count+1)</td>
</tr>
<tr>
<td></td>
<td>switch</td>
<td>switch</td>
</tr>
<tr>
<td></td>
<td>st (count+1)</td>
<td>st (count+1)</td>
</tr>
<tr>
<td></td>
<td>end trans</td>
<td>end trans</td>
</tr>
<tr>
<td></td>
<td>count+1</td>
<td>count+1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>count+2</td>
</tr>
</tbody>
</table>

- **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

SRAM, Fast, Small
Expensive
~KB to MB

DRAM, Slow, Big, Cheap
(called physical or main)
~100MB to GB

=> 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?

**Extending the Memory Hierarchy**

- Very fast 1ns clock
- Multiple Instructions per cycle
- HW manages movement
- SRAM, Fast, Small
- Expensive

- DRAM, Slow, Big, Cheap
  (called physical or main memory)

- Magnetic, Really Slow,
  Really Big, Really Cheap

- SW manages movement
Virtual Memory: Motivation

- **Process = Address Space** + thread of control (PC)
- **Address space = Physical**
  - programmer controls movement from disk
  - protection?
  - relocation?
- **Linear Address space**
  - larger than physical address space
    - 32, 64 bits v.s. 28-bit physical (256MB)
- **Want Automatic management**

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 ($2^{32}$, $2^{64}$) 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
- Page 0
- Page 1
- Page 2
- Page 3
- ...
- Page N-2
- Page N-1
- Page N

Physical Memory
- Frame 0
- Frame 1
- Frame 2
- Frame 3
- Frame 4
- Frame 5

Disk
- Page 2
- Page N-1
- Page N
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

\[ V = \{0, 1, \ldots, n-1\} \text{ virtual address space} \]
\[ M = \{0, 1, \ldots, m-1\} \text{ physical address space} \]
\[ n > m \]

\[ \text{MAP: } V \rightarrow M \cup \{\text{NULL}\} \text{ address mapping function} \]

\[ \text{MAP}(a) = a' \text{ if data at virtual address } a \text{ is present in physical address } a' \text{ and } a' \text{ in } M \]

\[ = \text{NULL} \text{ if data at virtual address } a \text{ is not present in } M \]
**Page Table**

- Operating System (Kernel) data structure (per process)
- Page Table Entry (PTE)
  - VA -> 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 -> 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)
Virtual to Physical Address translation

Page size: 4KB

Virtual Address

Virtual Page Number  Page offset

Physical Address

Physical Frame Number  Page offset

How should we design the page table?

Linear Page Table

Virtual Address

Virtual Page Number  Page offset

Physical Address

Physical Frame Number  Page offset

Valid bit   Dirty bit   Access Control

Page table base

© Alvin R. Lebeck
Inverted Page Table (HP, IBM)

- One PTE per page frame
  - only one VA per physical frame
- Must search for virtual address

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
Page Replacement Algorithms

• Just like cache block replacement!
• Least Recently Used:
  ➢ selects the least recently used page for replacement
  ➢ requires knowledge about past references
  ➢ more difficult to implement
  ➢ sorted list of page table entries from most recently referenced to least recently referenced; when a page is referenced it is placed at the head of the list; the end of the list is the page to replace
• Good performance, recognizes principle of locality
• Other schemes, take OS course…

The Memory Management Unit (MMU)

• Input
  ➢ virtual address
• Output
  ➢ physical address
  ➢ access violation (exception)
• Access Violations
  ➢ not present
  ➢ user vs. kernel
  ➢ write
  ➢ read
  ➢ execute
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
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

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

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).

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
Mapping the Kernel

- Digital Unix **Kseg**
  - 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

Segmented Virtual Memory

- Virtual address \(2^{32}, 2^{64}\) to Physical Address mapping \(2^{28}\)
- Each segment
  - variable size
  - Address = base + offset
  - contiguous in both VA and PA
Intel Pentium Segmentation

Virtual Address

Segment Selector

Offset

Global Descriptor Table (GDT)

Segment Descriptor

Segment Base Address

Physical Address Space

Intel Pentium Segmentation + Paging

Virtual Address (logical address)

Segment Selector

Offset

Global Descriptor Table (GDT)

Segment Descriptor

Segment Base Address

Physical Address Space

Linear Address Space

Page Table

Page Dir

Physical Address Space

Page Table

Offset

Table
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