Virtual Memory II

Learning Outcomes
- An understanding of TLB refill:
  - in general,
  - and as implemented on the R3000
- An understanding of demand-paged virtual memory in depth, including:
  - Locality and working sets
  - Page replacement algorithms
  - Thrashing

TLB Recap
- Fast associative cache of page table entries
  - Contains a subset of the page table
  - What happens if required entry for translation is not present (a TLB miss)?

Aside: even if filled by software
- TLB still a hardware-based translator

R3000 TLB Handling
- TLB refill is handled by software
  - An exception handler
- TLB refill exceptions accessing kuseg are expected to be frequent
  - CPU optimised for handling kuseg TLB refills by having a special exception handler just for TLB refills
Exception Vectors

<table>
<thead>
<tr>
<th>Program address</th>
<th>“segment”</th>
<th>Physical Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 0000</td>
<td>0x0000</td>
<td>TLB miss on kuseg reference only.</td>
<td></td>
</tr>
<tr>
<td>0x3000 0000</td>
<td>0x0000</td>
<td>All other exceptions.</td>
<td></td>
</tr>
<tr>
<td>0x400 0100</td>
<td>0x0000</td>
<td>Uncaught alternative kuseg TLB miss entry point (used if IS bit NEV set).</td>
<td></td>
</tr>
<tr>
<td>0x0000 0100</td>
<td>0x0000</td>
<td>Exception vector for all other kuseg TLB misses (used if IS bit BEV set).</td>
<td></td>
</tr>
<tr>
<td>0x000 0000</td>
<td>0x0000</td>
<td>Assumes virtual linear array – see extended OS notes (if interested).</td>
<td></td>
</tr>
</tbody>
</table>

Special Exception Vector

- Can be optimised for TLB refill only
  - Does not need to check the exception type
  - Does not need to save any registers
    - It uses a specialised assembly routine that only uses k0 and k1.
    - Does not check if PTE exists
- Assumes virtual linear array – see extended OS notes (if interested)
- With careful data structure choice, exception handler can be made very fast

Special Exception Vector for kuseg TLB refills

MIPS VM Related Exceptions

- TLB refill
  - Handled via special exception vector
  - Needs to be very fast
- Others handled by the general exception vector
  - TLB Mod
    - TLB modify exception, attempt to write to a read-only page
    - Attempt it load from a page with an invalid translation
  - TLB Store
    - Attempt to store to a page with an invalid translation
    - Note: these can be slower as they are mostly either caused by an error, or non-resident page.
    - We never optimise for errors, and page-loads from disk dominate the fault resolution cost.

Amdahl’s law

- States that overall performance improvement is limited by the fraction of time an enhancement can be used

\[
\text{Speedup} = \frac{\text{Execution Time Without Enhancement}}{\text{Execution Time With Enhancement}}
\]

\[
\text{Time}_{\text{old}} + 0.5 \times \text{Execution Time}_{\text{Previous}} = 2 \times \text{Time}_{\text{new}}
\]

\[
\text{Fraction in enhanced mode} = 0.5 \text{ (based on old system)}
\]

\[
\text{Speedup of enhanced mode} = 2
\]

Amdahl’s law

- States that overall performance improvement is limited by the fraction of time an enhancement can be used

\[
\text{Speedup} = \frac{\text{Execution Time Without Enhancement}}{\text{Execution Time With Enhancement}}
\]

\[
\text{Time}_{\text{old}} + 0.5 \times \text{Execution Time}_{\text{Previous}} = 2 \times \text{Time}_{\text{new}}
\]

\[
\text{Fraction in enhanced mode} = 0.5 \text{ (based on old system)}
\]

\[
\text{Speedup of enhanced mode} = 2
\]

<Intermezzo>
</Intermezzo>

### c0 Registers
- **c0_EPC**
  - The address of where to restart after the exception
- **c0_status**
  - Kernel/User Mode bits, Interrupt control
- **c0_cause**
  - What caused the exception
- **c0_badvaddr**
  - The address of the fault

### The TLB and EntryHi, EntryLo
- **c0 Registers**
  - **c0_EntryHi**
  - **c0_EntryLo**
  - **c0_Index**

- **TLB**
  - Each TLB entry contains:
    - EntryHi to match page# and ASID
    - EntryLo which contains frame# and protection

- **Used to read and write individual TLB entries**

### Special TLB management Instructions
- **TLBR**
  - TLB read
    - EntryHi and EntryLo are loaded from the entry pointer to by the index register.
- **TLBP**
  - TLB probe
    - Set EntryHi to the entry you wish to match, index register is loaded with the index to the matching entry
- **TLBWR**
  - Write EntryHi and EntryLo to a pseudo-random location in the TLB
- **TLBWI**
  - Write EntryHi and EntryLo to the location in the TLB pointed to by the index register.

### c0 Index Register
- **Used as an index to TLB entries**
  - Single TLB entries are manipulated/viewed through EntryHi and EntryLo0 registers
  - Index register specifies which TLB entry to change/view
Coprocesor 0 registers on a refill exception

c0.EPC ← PC
c0.cause.ExcCode ← TLBL ; if read fault
c0.cause.ExcCode ← TLBS ; if write fault
c0.BadVaddr ← faulting address
c0.EntryHi.VPN ← page number of faulting address
c0.status ← kernel mode, interrupts disabled.
c0.PC ← 0x8000 0000

Outline of TLB miss handling

• Software does:
  – Look up PTE corresponding to the faulting address
  – If found:
    • load c0_EntryLo with translation
    • load TLB using TLBWR instruction
    • return from exception
  – Else, page fault
• The TLB entry (i.e. c0_EntryLo) can be:
  – (theoretically) created on the fly, or
  – stored completely in the right format in page table
  • more efficient

OS/161 Refill Handler

• After switch to kernel stack, it simply calls the common exception handler
  – Stacks all registers
  – Can (and does) call ‘C’ code
  – Unoptimised
  – Goal is ease of kernel programming, not efficiency
• Does not have a page table
  – It uses the 64 TLB entries and then panics when it runs out.
    • Only support 256K user-level address space

Demand Paging

• With VM, only parts of the program image need to be resident in memory for execution.
• Can transfer presently unused pages/segments to disk
• Reload non-resident pages/segment on demand.
  – Reload is triggered by a page or segment fault
  – Faulting process is blocked and another scheduled
  – When page/segment is resident, faulting process is restarted
  – May require freeing up memory first
    • Replace current resident page/segment
    • How determine replacement “victim”?  
      – If victim is unmodified (“clean”) can simply discard it
        • This is reason for maintaining a “dirty” bit in the PT
• Why does demand paging work?
  – Program executes at full speed only when accessing the resident set.
  – TLB misses introduce delays of several microseconds
  – Page/segment faults introduce delays of several milliseconds
  – Why do it?
• Answer
  – Less physical memory required per process
    • Can fit more processes in memory
    • Improved chance of finding a runnable one
  – Principle of locality
**Principle of Locality**

- An important observation comes from empirical studies of the properties of programs.
  - Programs tend to reuse data and instructions they have used recently.
  - **90/10 rule**
    - A program spends 90% of its time in 10% of its code
- We can exploit this **locality of references**
- An implication of locality is that we can reasonably predict what **instructions** and **data** a program will use in the near future based on its accesses in the recent past.

**Two different types** of locality have been observed:
- **Temporal locality**: states that recently accessed items are likely to be accessed in the near future.
- **Spatial locality**: says that items whose addresses are near one another tend to be referenced close together in time.

**Locality In A Memory-Reference Pattern**

- The pages/segments required by an application in a time window (∆) is called its memory **working set**.
- Working set is an approximation of a program’s locality
  - If ∆ too small will not encompass entire locality.
  - If ∆ too large will encompass several localities.
  - If ∆ ≈ ∞ will encompass entire program.
  - ∆’s size is an application specific tradeoff
- System should keep resident at least a process’s working set
  - Process executes while it remains in its working set.
  - Working set tends to change gradually
    - Get only a few page/segment faults during a time window
    - Possible (but hard) to make intelligent guesses about which pieces will be needed in the future
      - May be able to pre-fetch pages/segments

**Working Set Example**

**Thrashing**

- CPU utilisation tends to increase with the degree of multiprogramming
  - Number of processes in system
- Higher degrees of multiprogramming – less memory available per process
- Some process’s working sets may no longer fit in RAM
  - Implies an increasing page fault rate
- Eventually many processes have insufficient memory
  - Can’t always find a runnable process
  - Decreasing CPU utilisation
  - System become I/O limited
- This is called **thrashing**.
Thrashing

- Why does thrashing occur?
  \( \sum \text{working set sizes} > \text{total physical memory size} \)

Recovery From Thrashing

- In the presence of increasing page fault frequency and decreasing CPU utilisation
  - Suspend a few processes to reduce degree of multiprogramming
  - Resident pages of suspended processes will migrate to backing store
  - More physical memory becomes available
  - Less faults, faster progress for runnable processes
  - Resume suspended processes later when memory pressure eases

What is the difference?

```c
/* reset array */
int array[10000][10000];
int i, j;
for (i = 0; i < 10000; i++) {
    for (j = 0; j < 10000; j++) {
        array[i][j] = 0;
    }
}
```

VM Management Policies

- Operation and performance of VM system is dependent on a number of policies:
  - Page table format (may be dictated by hardware)
    - Multi-level
    - Inverted/Hashed
  - Page size (may be dictated by hardware)
  - Fetch Policy
  - Replacement policy
  - Resident set size
    - Minimum allocation
    - Local versus global allocation
  - Page cleaning policy

Page Size

Increasing page size
- Increases internal fragmentation
  - reduces adaptability to working set size
- Decreases number of pages
  - Reduces size of page tables
  - Increases TLB coverage
  - Reduces number of TLB misses
- Increases page fault latency
  - Need to read more from disk before restarting process
  - Increases swapping I/O throughput
  - Small I/O are dominated by seek/rotation delays
  - Optimal page size is a (work-load dependent) trade-off.
Working Set Size Generally Increases with Increasing Page Size: True/False?

- Atlas: 512 words (48-bit)
- Honeywell/Multics: 1K words (36-bit)
- IBM 370/XA: 4K bytes
- DEC VAX: 512 bytes
- IBM AS/400: 512 bytes
- Intel Pentium: 4K and 4M bytes
- ARM: 4K and 64K bytes
- MIPS R4000: 4K – 16M bytes in powers of 4
- DEC Alpha: 8K - 4M bytes in powers of 8
- UltraSPARC: 8K – 4M bytes in powers of 8
- PowerPC: 4K bytes + “blocks”
- Intel IA-64: 4K – 256M bytes in powers of 4

Page Size

- Multiple page sizes provide flexibility to optimise the use of the TLB
- Example:
  - Large page sizes can be use for code
  - Small page size for thread stacks
- Most operating systems have limited support for only a single page size
  - Dealing with multiple page sizes is hard!

Fetch Policy

- Determines when a page should be brought into memory
  - Demand paging only loads pages in response to page faults
  - Pre-paging brings in more pages than needed at the moment
  - Pre-fetch when disk is idle
  - Wastes I/O bandwidth if pre-fetched pages aren’t used
  - Especially bad if we eject pages in working set in order to pre-fetch unused pages.
  - Hard to get right in practice.

Replacement Policy

- Which page is chosen to be tossed out?
  - Page removed should be the page least likely to be references in the near future
  - Most policies attempt to predict the future behaviour on the basis of past behaviour
- Constraint: locked frames
  - Kernel code
  - Main kernel data structure
  - I/O buffers
  - Performance-critical user-pages (e.g. for DBMS)
- Frame table has a lock (or pinned) bit
Optimal Replacement policy

- Toss the page that won’t be used for the longest time
- Impossible to implement
- Only good as a theoretic reference point:
  - The closer a practical algorithm gets to optimal, the better
- Example:
  - Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - Four frames
  - How many page faults?

FIFO Replacement Policy

- First-in, first-out: Toss the oldest page
  - Easy to implement
  - Age of a page isn’t necessarily related to usage
- Example:
  - Reference string: 1,2,3,4,1,2,5,1,2,3,4,5
  - Four frames

Least Recently Used (LRU)

- Toss the least recently used page
  - Assumes that page that has not been referenced for a long time is unlikely to be referenced in the near future
  - Will work if locality holds
  - Implementation requires a time stamp to be kept for each page, updated on every reference
  - Impossible to implement efficiently
  - Most practical algorithms are approximations of LRU

Clock Page Replacement

- Clock policy, also called second chance
  - Employs a usage or reference bit in the frame table.
  - Set to one when page is used
  - While scanning for a victim, reset all the reference bits
  - Toss the first page with a zero reference bit.

Figure 8.16 Example of Clock Policy Operation
Issue

- How do we know when a page is referenced?
- Use the valid bit in the PTE:
  - When a page is mapped (valid bit set), set the reference bit
  - When resetting the reference bit, invalidate the PTE entry
  - On page fault
    - Turn on valid bit in PTE
    - Turn on reference bit
- We thus simulate a reference bit in software

Performance

- It terms of selecting the most appropriate replacement, they rank as follows
  1. Optimal
  2. LRU
  3. Clock
  4. FIFO
  - Note there are other algorithms (Working Set, WSclock, Ageing, NFU, NRU)
  - We don’t expect you to know them in this course

Resident Set Size

- How many frames should each process have?
  - Fixed Allocation
    - Gives a process a fixed number of pages within which to execute.
    - Isolates process memory usage from each other
    - When a page fault occurs, one of the pages of that process must be replaced.
    - Achieving high utilisation is an issue.
      - Some processes have high fault rate while others don’t use their allocation.
  - Variable Allocation
    - Number of pages allocated to a process varies over the lifetime of the process
Variable Allocation, Global Scope

– Easiest to implement
– Adopted by many operating systems
– Operating system keeps global list of free frames
– Free frame is added to resident set of process when a page fault occurs
– If no free frame, replaces one from any process

• Pro/Cons
  – Automatic balancing across system
  – Does not provide guarantees for important activities

Variable Allocation, Local Scope

• Allocate number of page frames to a new process based on
  – Application type
  – Program request
  – Other criteria (priority)
• When a page fault occurs, select a page from among the resident set of the process that suffers the page fault
• Re-evaluate allocation from time to time!

Page-Fault Frequency Scheme

• Establish “acceptable” page-fault rate.
  – If actual rate too low, process loses frame.
  – If actual rate too high, process gains frame.

Cleaning Policy

• Observation
  – Clean pages are much cheaper to replace than dirty pages
• Demand cleaning
  – A page is written out only when it has been selected for replacement
  – High latency between the decision to replace and availability of free frame.
• Precleaning
  – Pages are written out in batches (in the background, the pagedaemon)
  – Increases likelihood of replacing clean frames
  – Overlap I/O with current activity