Operating Systems

Tutorial 1

Michael Tänzer

os-tut@nhng.de
http://os-tut.nhng.de

21st December 2010

Outline

True or False

- If the system is in a safe state all resource requests from existing processes which don’t exceed their maximum can be granted and still no deadlock can occur
- A circle in a WFG indicates a deadlock
- Bounded waiting is a necessary condition for a deadlock

Basic idea of paging

- Logical address space is contiguous while physical space occupied doesn’t need to be
- Logical memory is divided into fixed size blocks called pages
- Physical memory is divided into blocks of the same size which are called frames
- The page table maps pages to frames

Address translation using a single-level page table

1. Each logical address is divided into two parts
   - Page number
   - Page offset
2. The page number is used as an index into the page table which contains
   - Frame number
   - (Status bits, e. g. protection, valid)
3. Physical address := frame number $\times$ frame size + offset

Typically the page/frame size is a power of two $\Rightarrow$ frame number $\times$ frame size is a simple shift operation

What is the disadvantage of single-level page tables?

- Size(page table) = Size(page table entry) $\times$ (#logical addresses $\div$ page size)
- Increases linearly with the size of the address space

$\Rightarrow$ Page tables become quite large (especially with 64-bit processors)

Space needed for a page table on 32-bit processors

- Page size = 4k
- Page table entry size = 4 bytes

$$\Rightarrow \frac{2^{32}}{2^{12}} \cdot 2^2 = 2^{22} = 4MB$$

- And that's per process/address space

What alternatives to single-level page tables exist?

Strengths and weaknesses

Multi-level Page Tables

- Entries in higher levels point to another page table instead of a frame
- Logical address is divided into $n + 1$ parts
- Each part except the last one, is used as an index into a page table on the i-th level

+ Address spaces are usually sparse $\Rightarrow$ not all page tables on all levels are needed (entries in the higher tables are marked invalid)

+ Sub-tables can be swapped out to disk

– Memory accesses required to do the address translation increases with the levels of page tables

What alternatives to single-level page tables exist?

Strengths and weaknesses

Hashed Page Tables

- Use a hash table instead of an array
- Collisions are resolved using chaining

+ Only those entries that are used need to be present

– Each entry needs to contain the page and frame number

– If there are many collisions the time to traverse the collision chain may become critical

⇒ Need to find the right™ size for the hash table (dynamic growing/shrinking possible)

What alternatives to single-level page tables exist?

Strengths and weaknesses

Inverted Page Tables

- Frames used as index instead of pages
- Entries contain the address space ID and page number to which the frame is mapped

+ Size only depends on size of physical memory

– Requires linear search to do the address translation

⇒ Can use an additional hash table to make it more efficient

– Sharing of memory becomes difficult (need to keep track of all address spaces which point to the frame)

Demand Paging vs. Pre-Paging

Demand Paging

- Pages are only mapped on page fault

– High number of page faults on application start up

Pre-Paging

- Speculatively map pages that might be needed

– Some of the preloaded pages might not be used ⇒ waste of time and memory

+ Reading large chunks from hard disk is more efficient

What is a TLB?

- Hardware cache to speed up address translation
- Contains pairs of logical (key) and the corresponding physical address (value)
- On TLB miss the address has to be looked up the ‘normal’ way (e. g. by traversing page tables)
- TLB miss handling can be done in
  - Hardware
    - The hardware does the lookup, only if the entry isn’t present in the page table (e. g. because the page is swapped out) an exception is raised (page fault)

    $\Rightarrow$ The hardware has to know the page table layout
  - Software
    - Hardware indicates an exception $\Rightarrow$ the kernel handles the exception in any way it likes to and fills the TLB with the pair needed

    $\Rightarrow$ Flexible but slow

What is the purpose of the valid (alias present) bit?

- Indicates whether the page is currently in memory
- If set the entry can be used for deriving the physical address
- If not set a page fault is raised and the page fault handler is called
- Unset valid bit does not necessarily mean that access to the page is forbidden (could be just swapped out)

$\Rightarrow$ OS needs to keep track of whether the access is legal or not

What is copy-on-write?

How can it be implemented?

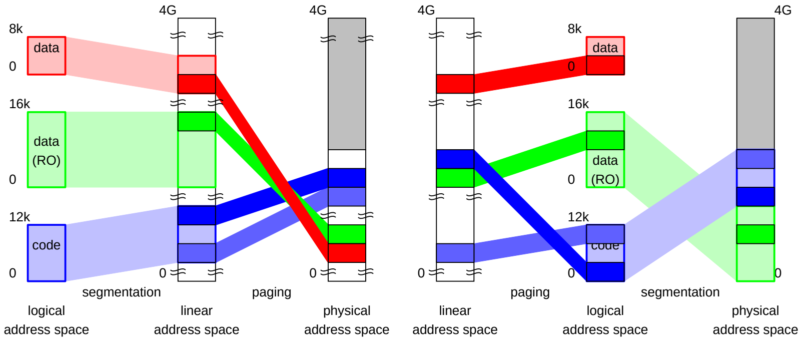
- Avoid unnecessary copies of memory pages by referencing the same frame and only copy the frame when writing to it
- Common example: `fork`
  - Whole address space has to be copied
  - Some pages are never modified ⇒ saves space
  - When followed by `exec` the address space is replaced anyway
- Page tables point to the same frames but the entries are marked read only

⇒ When writing to it a page fault is raised

⇒ The OS can copy the frame, map the page to the new frame and mark it writeable

- If the frame was only mapped to two pages the other page can also be marked writeable again

Compare the two approaches



(a) (b)

Why is loading the CR3 register (address of top-level page table) expensive?

What change to the hardware would significantly reduce this cost?

- TLB and virtually tagged caches have to be flushed
- When looking up the logical address also check an additional address space ID
  - If ID same as ID of the current address space ⇒ entry found
  - If ID not equal to the current address space ID ⇒ entry belongs to another address space, keep searching

What's the difference between the use of hardware- and software-walked page tables?

How does this relate to TLBs?

Hardware-Walked Page Tables

- Page table lookup is done by the MMU
- If the page table entry or page directory entry is invalid a page fault is raised

⇒ Has to be handled by the OS (page fault handler: install the appropriate mapping or abort application)

What's the difference between the use of hardware- and software-walked page tables?

How does this relate to TLBs?

Software-Walked Page Tables

- If the MMU doesn't find a matching TLB entry a TLB fault exception (also called TLB miss exception) is raised

$\Rightarrow$ OS has to do the lookup in the page table

- If no valid entry is found the TLB fault handler calls the page fault handler

What's the difference in the contents of HW- vs. SW-walked multi-level page tables?

Hardware-Walked Page Tables

- Fixed layout
- Structure defined by the hardware vendor
- Need to use separate data structures for extra information that doesn't fit in the page table (e. g. for shared memory or copy-on-write)

Software-Walked Page Tables

- Flexible Layout
- Structure can be defined by the OS designer
- Can put as much information in the page table as you want

Under what circumstances are TLB miss or page fault handlers invoked?

Hardware-Walked Page Tables

- Page fault: if the page table doesn't contain a valid entry for the requested page or the desired access is not allowed

Software-Walked Page Tables

- TLB miss: if the TLB doesn't contain a valid entry for the requested page
- TLB write/read/execute fault: if the desired access is not allowed according to the TLB entry

How can HW-walked multi-level page tables be set up or accessed from software running on truly paged virtual memory?

- First level has to contain physical addresses of lower levels
  $\Rightarrow$ chicken vs. egg
- Solution: In bootstrap use fixed offset for the mapping
- After bootstrap true paging is turned on
- Non-solution: Turning paging off for every page table access
  - Too much overhead
  - As soon as paging is disabled all addresses are interpreted as physical ones

  $\Rightarrow$ When the instruction counter is incremented the address is interpreted as physical address

  $\Rightarrow$ Code for turning off (and on) paging would have to be mapped 1:1

Enumerate the different kinds of cache misses

Compulsory miss	The accessed data block has not been in the cache before
Capacity miss	The cache is too small for all the required data to fit
Conflict miss	Different data blocks are mapped to the same location in the cache

Write-through vs. write-back

Write-through

- Update all upstream caches and main memory on write
- + No inconsistencies between caches and memory
- − Upstream memory is slow $\Rightarrow$ buffer

Write-back

- No upstream propagation of changes on write
- When a dirty cache line is evicted from the cache it is written back to the upstream cache or memory
- + Less traffic
- − Often need two upstream accesses on a read

How can writes to data not in cache be handled?

Write-allocate

- Data is fetched into the cache
- Write is performed in the cache according to it's write policy

Write-to-memory

- The write is performed directly on the upstream cache or memory

What are the two main problems with virtually indexed and tagged caches?

How can we avoid or solve these?

Ambiguity

- The same virtual address points to different physical addresses at different points in time (e. g. other address space loaded)
- Cache might indicate a hit for the virtual address but contain data from a different physical address than the current mapping would imply

$\Rightarrow$ Invalidate cache lines when the mapping changes

$\Rightarrow$ On an address space switch the whole cache has to be invalidated

What are the two main problems with virtually indexed and tagged caches?

How can we avoid or solve these?

Aliases

- Different virtual addresses map to the same physical address
- Writing to one cache line leaves the other unmodified $\Rightarrow$ inconsistency
- “Solutions”:
  - Prohibit mapping one frame to multiple pages within the same address space
  - Disable caching for these pages
  - Only pages that map to the same cache line may refer to the same frame (only works with direct-mapped caches)

Do virtually indexed, physically tagged caches solve those problems?

- Solves ambiguity:
  - When the mapping changes the virtual index will still point to the same cache line
  - But the physical tag will no longer match $\Rightarrow$ data will be fetched from upstream
- Doesn't solve problems with aliases:
  - The virtual index possibly points to different cache lines $\Rightarrow$ same problem as before
  - The physical tag is the same on all those cache lines but that doesn't help

Questions & Comments

Any questions or comments?

Merry Christmas and a Happy New Year