Slide 1

Virtual Memory: Basics

COMP3231 Operating Systems

2004/S2

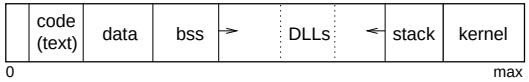
Slide 2

Basic Features of Virtual Memory

- ➜ Piecewise-linear mapping of virtual to physical addresses
  - Paging: fixed and (usually) equal-sized blocks
  - Segmentation: variable-sized blocks
- ➜ Mapping is defined at run-time, and can change
- ➜ Process does not have to be contiguous in physical memory
- ➜ Address space can have holes (unmapped regions)
- ➜ Process image may be only partially resident
  - Allows OS to swap individual pages/segments to disk
  - Saves memory for infrequently used code or data
  - What happens if program accesses non-resident memory?

Slide 3

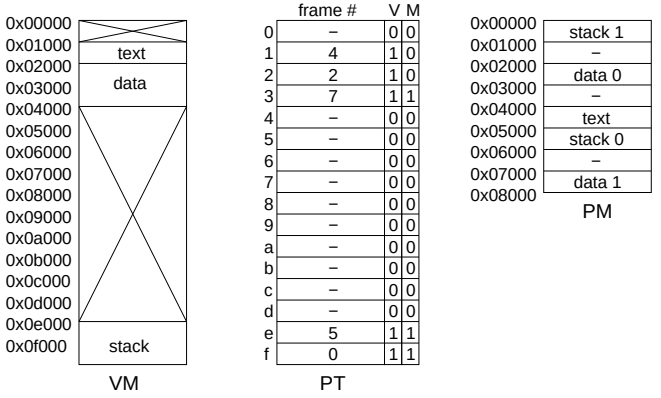
Typical Virtual Address Space Layout (UNIX):



- 0-th page typically not used
- text segment is read-only
- data segment is initialised data (partially R/O)
- bss segment is uninitialised data (heap), can grow
- shared libraries (DLLs) allocated in free middle region
- stack at top of user space, grows downward
- kernel space is in reserved (shared) region

Slide 4

Page table for process address space::



Slide 5

Page table structure::

- Page table is (logically) an array of frame numbers
  - indexed by page numbers
- Each page-table entry (PTE) also has
  - A valid bit (or present bit), indicating that there is a valid mapping for the page
  - A modified bit (also called dirty bit), indicating that the page may be modified in memory
  - Referencing an invalid page triggers a page fault:
    - illegal address (protection error)
    - page not resident, needs to be swapped in:
      ① get empty frame
      ② load page
      ③ update page table (enter frame #, set V, reset M)
      ④ restart faulting instruction

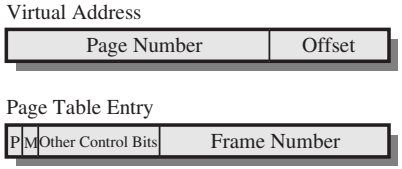
Slide 6

Address Translation (Paging)

➜ Every (virtual) memory address issued by CPU must be translated to physical
- every `load` or `store` instruction
- every instruction fetch

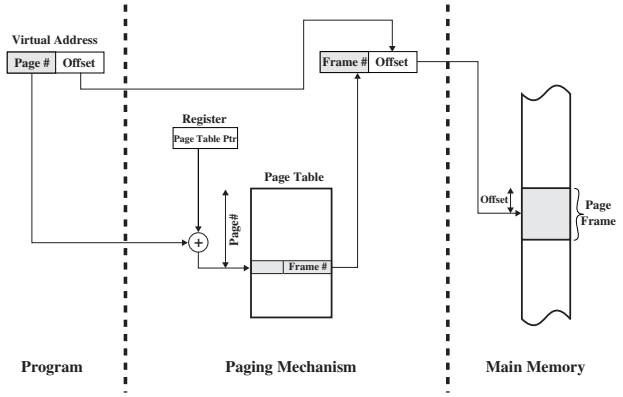
➜ Need translation hardware

➜ In paging system, translation involves replacing page # by frame #



Slide 7

Address translation in a paging system::



Slide 8

Translation Lookaside Buffer:

➜ Each virtual memory reference can cause two physical memory accesses
- one to fetch the page table entry
- one to fetch/store the data

⇒ Intolerable performance impact!

➜ Solution: high-speed cache for page table entries (PTEs)
- Called the translation lookaside buffer (TLB)
- Contains recently used page table entries
- Associative high-speed memory, similar to a memory cache
- May be under OS control (unlike memory cache)

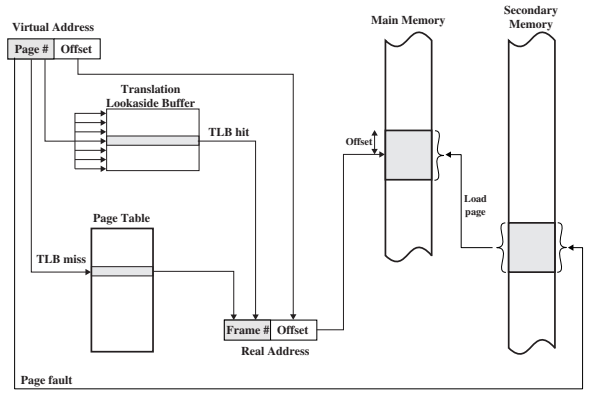
Slide 9

Translation Lookaside Buffer:

- Given a virtual address, processor examines the TLB
- If matching PTE is found (TLB hit), the address is translated
- Otherwise (TLB miss), the page number is used to index the process page table
  - If PT contains a valid entry, reload TLB and restart
  - Otherwise (page fault) check if page is on disk
    - If on disk, swap in
    - Otherwise allocate a new page or raise an invalid address exception

Slide 10

TLB operation::



Slide 11

TLB properties::

- TLB entries contain (per-page) write-protect bits
- TLB may or may not be under OS control:
  - Hardware-loaded TLB:
    - On miss, hardware performs PT lookup and reloads TLB
    - Example: Pentium, most 32-bit architectures, PowerPC
  - Software-loaded TLB:
    - On miss, hardware generates a TLB miss exception, and exception handler reloads TLB.
    - Example: MIPS, and most modern architectures
- TLB size: typically 64–128 entries
- Modern architectures generally have separate TLBs for

Slide 12

instruction fetch and data access

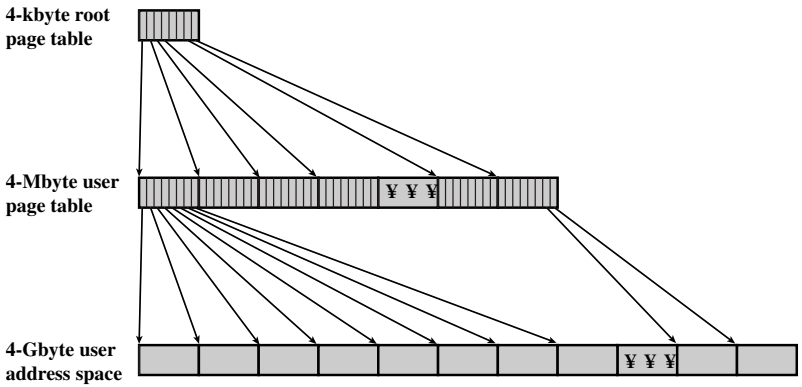
PAGE TABLES

Slide 13

- Even a single process' page table is big: ≥ 4MB for 32-bit address space
- Most processes do not use a full 4GB address space
  - e.g., 0.1–1MB text, 0.1–10MB data, 0.1MB stack
- Need compact representation that doesn't waste space
- Three basic schemes:
  - use data structures which adapt to sparsity
  - use data structures which only represent resident pages
  - use VM techniques for page tables

Slide 14

Two-level page table (32-bit address spaces):

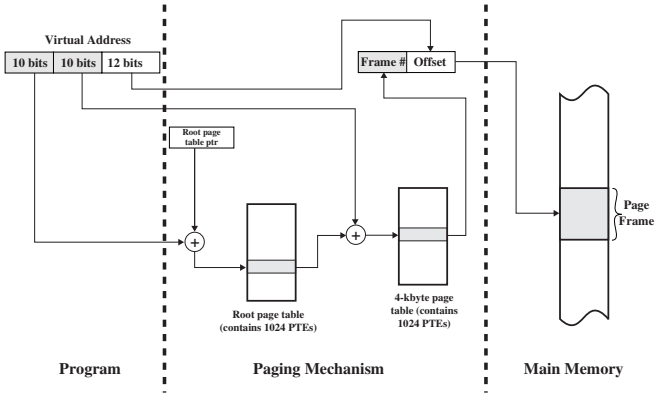


Slide 15

- Note: Unused PT pages not allocated (NULL pointer in root page table)

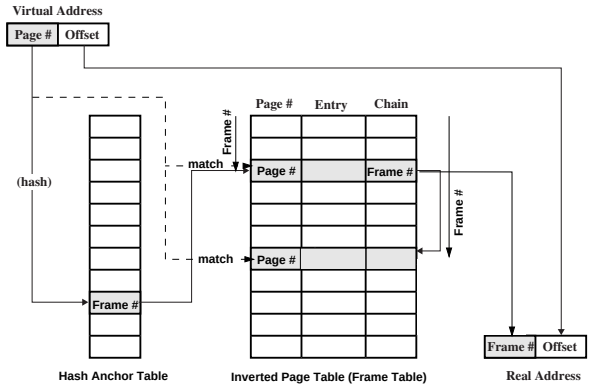
Slide 16

Address translation with 2-level PT (2LPT)::



Slide 17

Alternative: Inverted page table:



Slide 18

Inverted page table (IPT) operation::

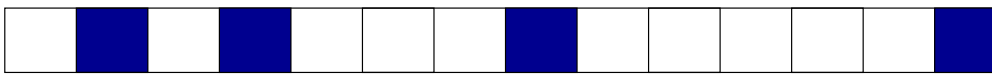
- "Inverted page table" is an array of page numbers sorted by frame number (it's a frame table)
- Algorithm:
  ① Compute hash of page number (usually least significant bits)
  ② us this to index into the hash anchor table (HAT)
  ③ HAT contains candidate frame number
  ④ use this to index into frame table
  ⑤ match the page number in FT entry to original
  ⑥ if match, use frame # for translation
  ⑦ if no match, get next candidate frame # from chain field
  ⑧ if NULL chain entry ⇒ page fault
- IPT grows with size of RAM, not virtual address space!
- Frame table is needed anyway (for page replacement)
- Need separate data structure for non-resident pages

Slide 19

Alternative: Virtual linear array page table:

- Assume a 2-level PT
- Assume 2^{nd} level PT nodes are in virtual memory
- Assume all 2^{nd} level nodes are allocated contiguously
  ⇒ 2^{nd} level nodes then form a contiguous array

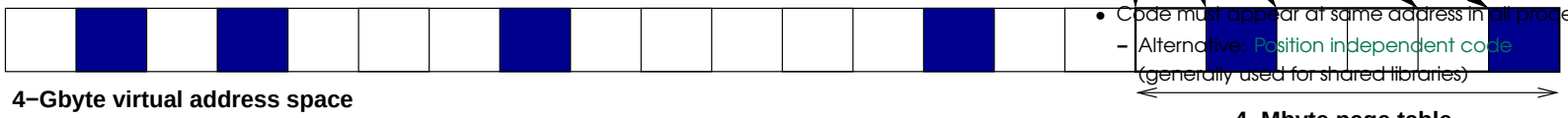
Slide 20

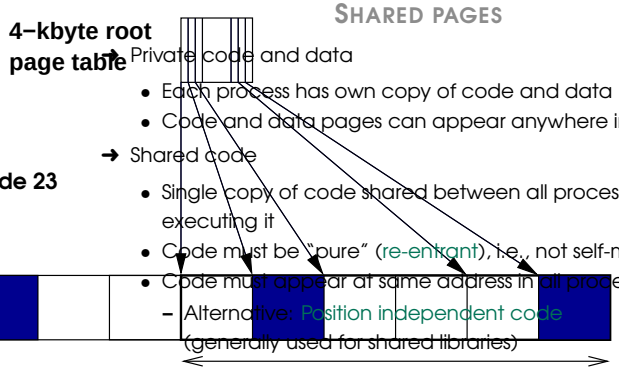


4–Gbyte virtual address space

Slide 21

Virtual linear array page table operation:





Slide 22

- ➜ Index into 2^{nd} level page table without referring to root PT!
- ➜ Simply use the full page number as the PT index!
- ➜ Leave unused parts of PT unmapped!
- ➜ If access is attempted to unmapped part of PT, a secondary page fault is triggered
  - This will load the mapping for the PT from the root PT
  - Root PT is kept in physical memory (cannot trigger page faults)

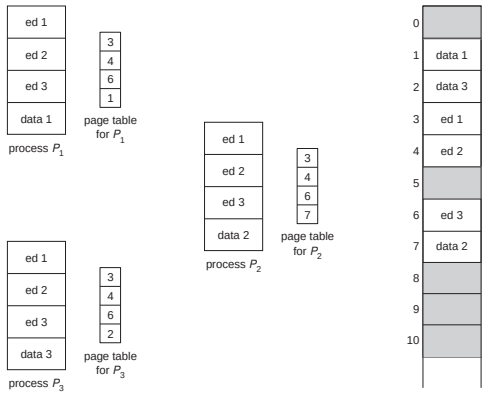
Slide 23

SHARED PAGES

- ➜ Private code and data
  - Each process has own copy of code and data
  - Code and data pages can appear anywhere in the VAS
- ➜ Shared code
  - Single copy of code shared between all processes executing it
  - Code must be "pure" (re-entrant), i.e., not self-modifying
  - Code must appear at same address in all processes
    - Alternative: Position independent code (generally used for shared libraries)

Slide 24

Example: Shared pages:



Slide 25

PAGE TABLES AND THE TLB

- ➜ Page table is (logically) an array of frame #'s indexed by page #
- ➜ The TLB holds a (recently used) subset of PT entries
  - each TLB entry must be identified (tagged) with the page # it translates
  - access is by associate lookup:
    - all TLB entries' tags are concurrently compared to the page #
  - TLB is associative (or context-addressable) memory

$page\ \#$	$frame\ \#$	V	W
...	...	.	.
...	...	.	.

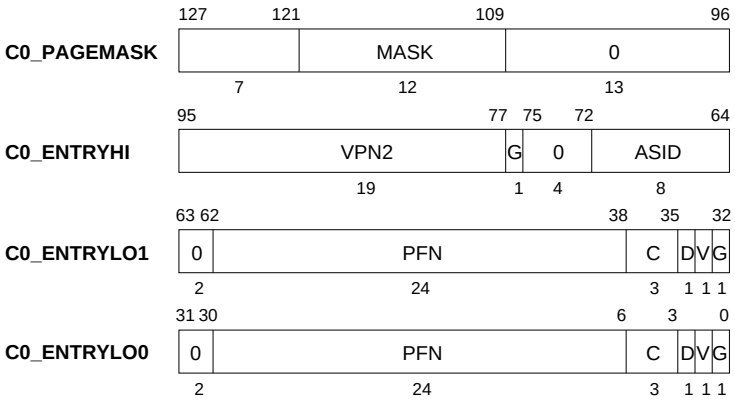
Slide 26

Page tables and the TLB...:

- ➜ TLB is a shared piece of hardware
- ➜ Page tables are per-process (address space)
- ➜ TLB entries are process-specific
  - On context switch need to flush the TLB (invalidate all entries)
    - high context-switching overhead (ix86)
  - **or** tag entries with address-space ID (ASID)
    - called a tagged TLB
    - used (in some form) on all modern architectures
  - TLB entry: ASID, page #, frame #, valid and write-protect bits
  - On TLB load:
    - construct entry on-the-fly from data in PTE, or
    - load a complete TLB entry from PT

Slide 27

Example: MIPS R4x00 TLB entry (32-bit mode):



Slide 28

Explanation:

- Current TLBE is mirrored in CP0 registers `C0_PAGEMASK`, `C0_ENTRYHI`, `C0_ENTRYLO1`, `C0_ENTRYLO0`
- Each entry (of 48) maps a pair of pages (buddies) with common value of `VPN2` (= virtual page #/2)
  - `ENTRYLO0` maps page $(2\times$`VPN2`$)$
  - `ENTRYLO1` maps page $(2\times$`VPN2`$+1)$

Slide 29

`MASK`	defines size of page (0: page size is 4kB)
`ASID`	address-space ID tag
`PFN`	physical frame number
`C`	page cacheability/coherency
`D`	dirty: page is writable
`V`	mapping is valid
`G`	global: ignore ASID (in tag word in TLB, in `ENTRYLO` in CP0)
`0`	must be zero

Slide 30

MIPS addressing (highly simplified)::

- virtual addressing:
  - addresses in the bottom half of the VAS ($<$0x8000 0000) are translated by the TLB
- quasi-physical addressing:
  - addresses in the upper half of the VAS ($\geq$0x8000 0000) are translated by masking out the top bits:

    ```
    pa = va&0xfff ffff
    ```
  - these addresses are called `kseg0` addresses
  - `kseg0` addresses are only available in kernel mode
  - typical use: `(pa+K0SEG_BASE)` to refer to `pa`
  - used e.g. by exception handlers

Slide 31

Exception handling on MIPS:

- MIPS has 4 different exception handlers:
  - 32-bit mode fast TLB miss handler `TLB`
    - at PA 0x000 (VA 0x8000 0000)
    - unless already in exception mode
  - 64-bit mode fast TLB miss handler `XTLB`
    - at PA 0x080 (VA 0x8000 0080)
    - unless already in exception mode
  - cache error handler `CACHE`
    - at PA 0x100 (VA 0x8000 0100)
  - general exception handler `GENERAL`
    - at PA 0x180 (VA 0x8000 0180)
    - all other exceptions
    - including TLB exceptions in exception mode

Slide 32

32-bit TLB miss handling on MIPS (simplified):

- TLB refill exception: TLB miss in non-exception mode
- MIPS processor does the following:

```
CP0.EPC             ← PC
CP0.CAUSE.ExcCode   ← TLBL          ; if read fault
CP0.CAUSE.ExcCode   ← TLBS          ; if write fault
CP0.BadVaddr        ← faulting address
CP0.EntryHi.VPN2    ← faulting address/(2*pagesize)
CP0.STATUS.EXL      ← 1             ; enter exception mode
CP0.PC              ← 0x8000 0000   ; fast TLB miss handler
```

Note: ASID is already contained in `CP0.EntryHi.ASID`

Slide 33

TLB miss handling...:

➜ Software does:

① Look up PTE corresponding to fault address

② if found:

① load `C0_ENTRYLO0` and `C0_ENTRYLO1` with translations

② load TLB using `tlbwr` instruction

③ return from exception

③ else handle page fault

➜ TLB entry (i.e., `C0_ENTRYLO0`, `C0_ENTRYLO1`) can be:

- created on the fly, or
- stored completely and in the right format in the PT

Slide 34

Other TLB exceptions:

➜ vectored to `GENERAL` handler:

- TLB invalid: entry exists but `V` flag off
- TLB modified: write to page with `D` flag off

➜ Processor handles as for TLB miss, except:

```
CP0.CAUSE.ExcCode  ← TLBL        ; if read invalid fault
CP0.CAUSE.ExcCode  ← TLBS        ; if write invalid fault
CP0.CAUSE.ExcCode  ← Mod         ; if write protect fault
CP0.PC             ← 0x8000 0180 ; gen excpt handler
```

➜ Software must:

- interpret and fix the cause of the invalid access
- fix PT entry
- reload TLB