Virtual Memory: Basics

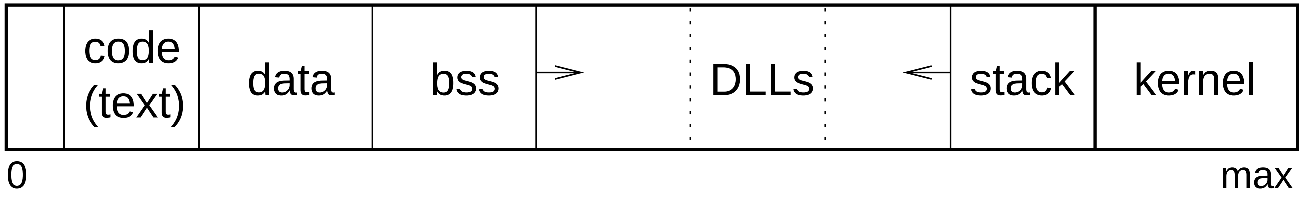
COMP3231 Operating Systems

2004/S2

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?

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

Page table for process address space::

Address	VM
0x00000	
0x01000	text
0x02000	data
0x03000	
0x04000	
0x05000	
0x06000	
0x07000	
0x08000	
0x09000	
0x0a000	
0x0b000	
0x0c000	
0x0d000	
0x0e000	stack
0x0f000	

VM

	frame #	V	M
0	–	0	0
1	4	1	0
2	2	1	0
3	7	1	1
4	–	0	0
5	–	0	0
6	–	0	0
7	–	0	0
8	–	0	0
9	–	0	0
a	–	0	0
b	–	0	0
c	–	0	0
d	–	0	0
e	5	1	1
f	0	1	1

PT

Address	PM
0x00000	stack 1
0x01000	–
0x02000	data 0
0x03000	–
0x04000	text
0x05000	stack 0
0x06000	–
0x07000	data 1
0x08000	

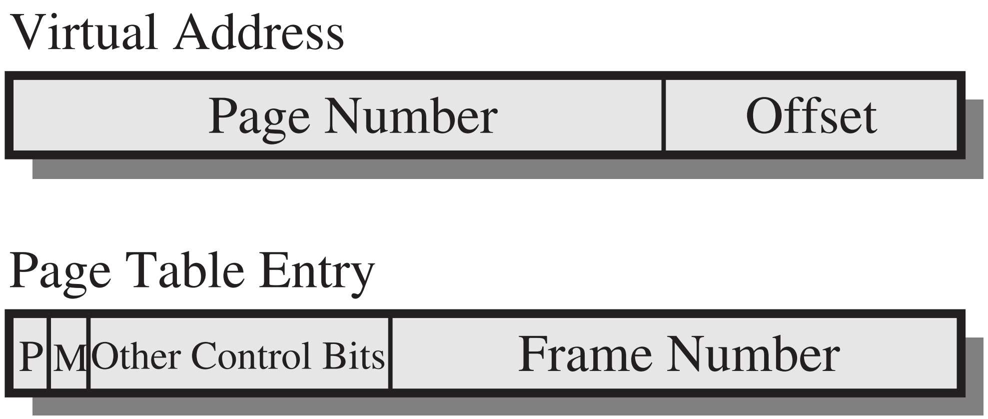
PM

Page table structure::

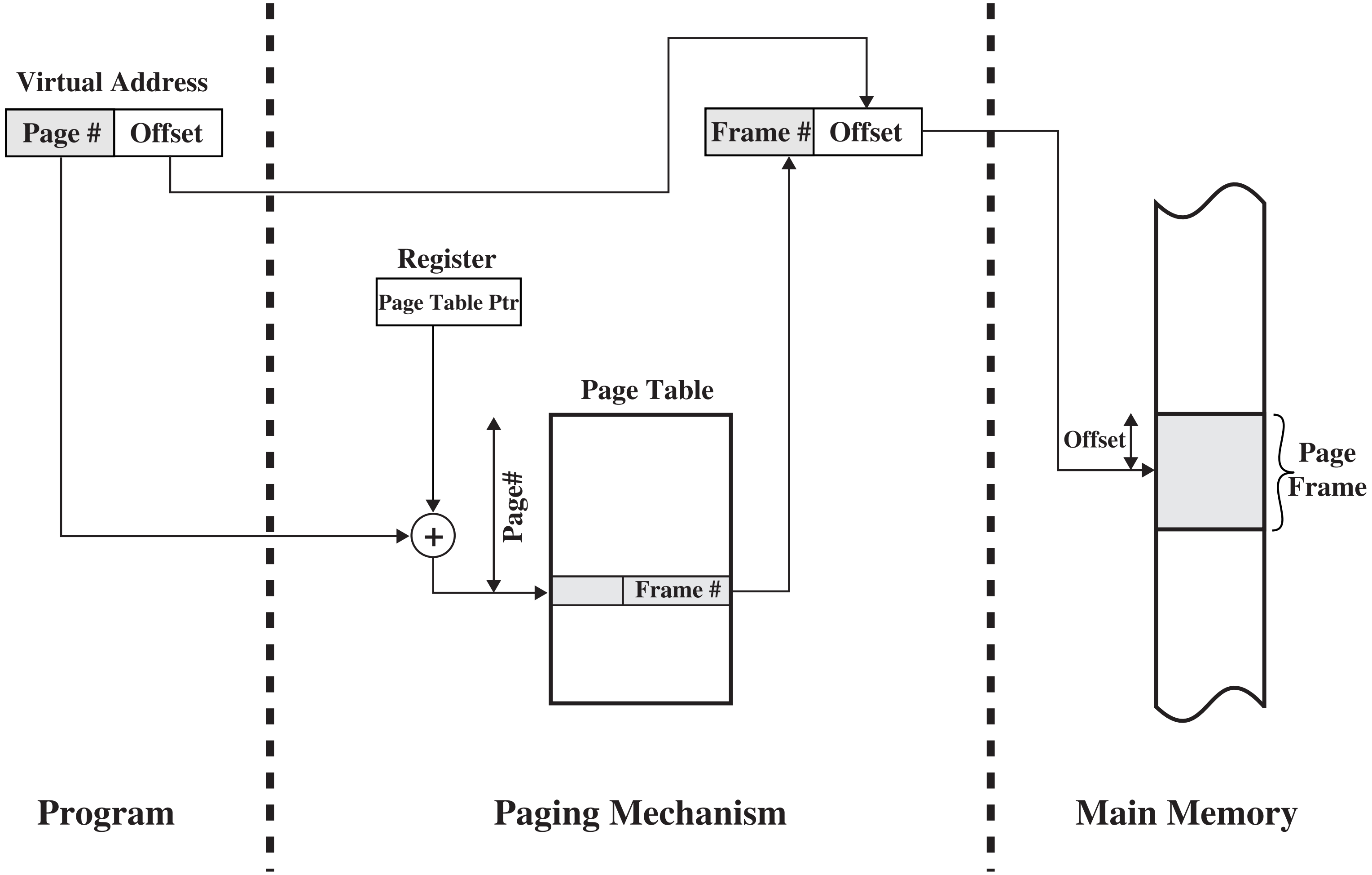
- Page table is (logically) an array of frame numbers
  - indexed by page numbers
- Each page-table entry (PTE) also has
  - A valid bit (or present bit), indicating that there is a valid mapping for the page
  - A modified bit (also called dirty bit), indicating that the page may be modified in memory
  - 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

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 #



Address translation in a paging system::



Translation Lookaside Buffer:

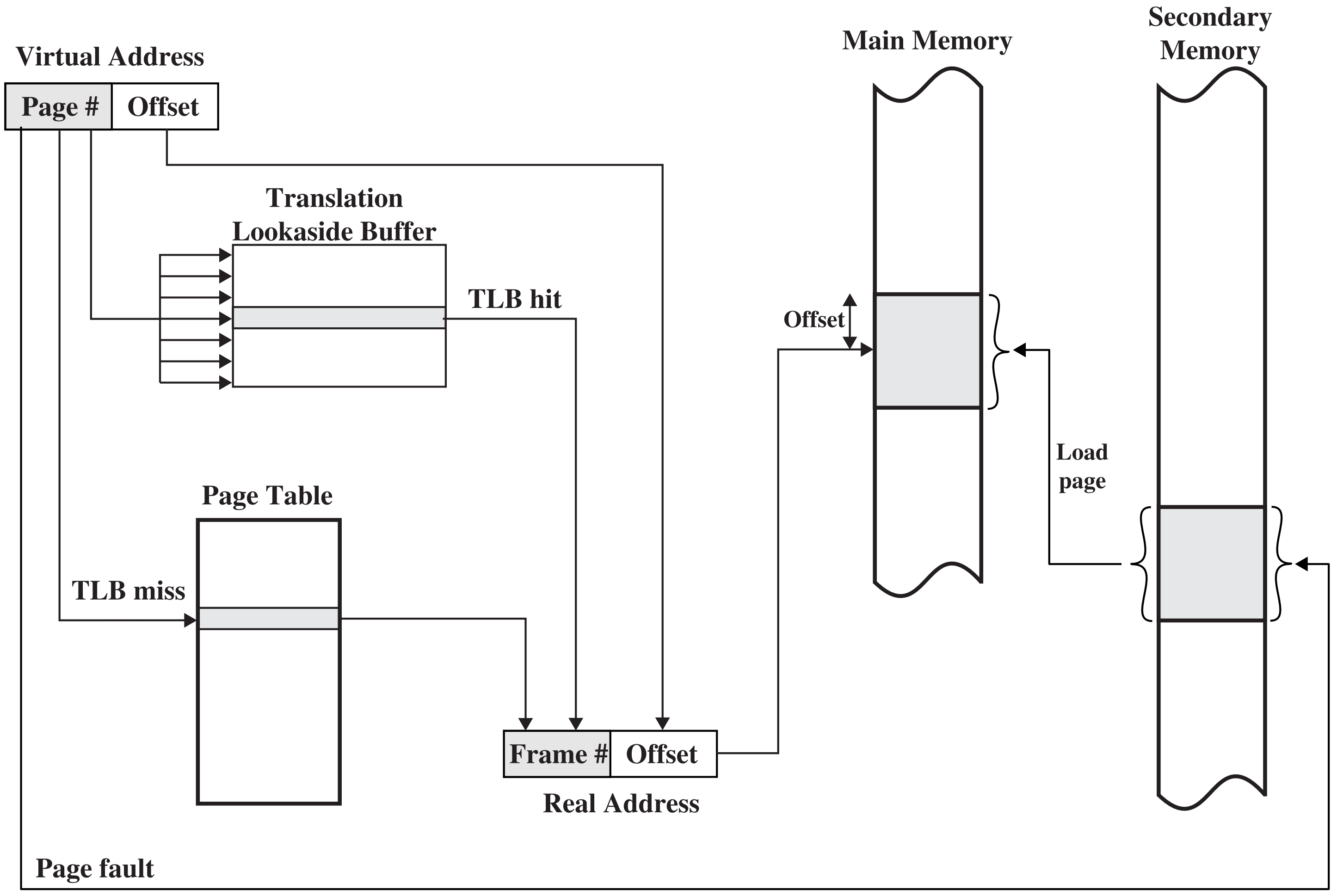
- ➜ Each virtual memory reference can cause two physical memory accesses
  - one to fetch the page table entry
  - one to fetch/store the data

  ⇒ 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)

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

TLB operation::



TLB properties::

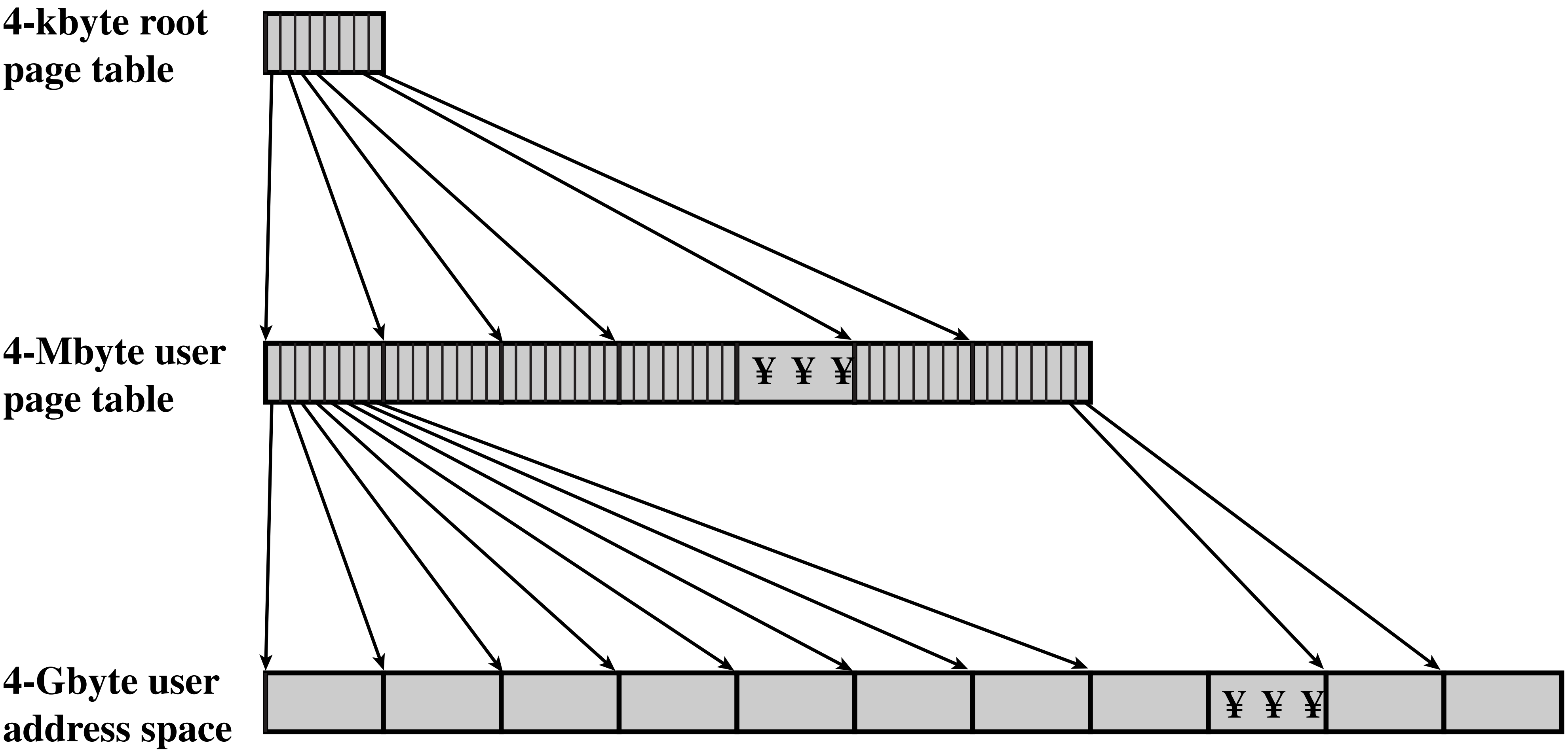
- TLB entries contain (per-page) write-protect bits
- TLB may or may not be under OS control:
  - Hardware-loaded TLB:
    - On miss, hardware performs PT lookup and reloads TLB
    - Example: Pentium, most 32-bit architectures, PowerPC
  - Software-loaded TLB:
    - On miss, hardware generates a TLB 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

instruction fetch and data access

Page Tables

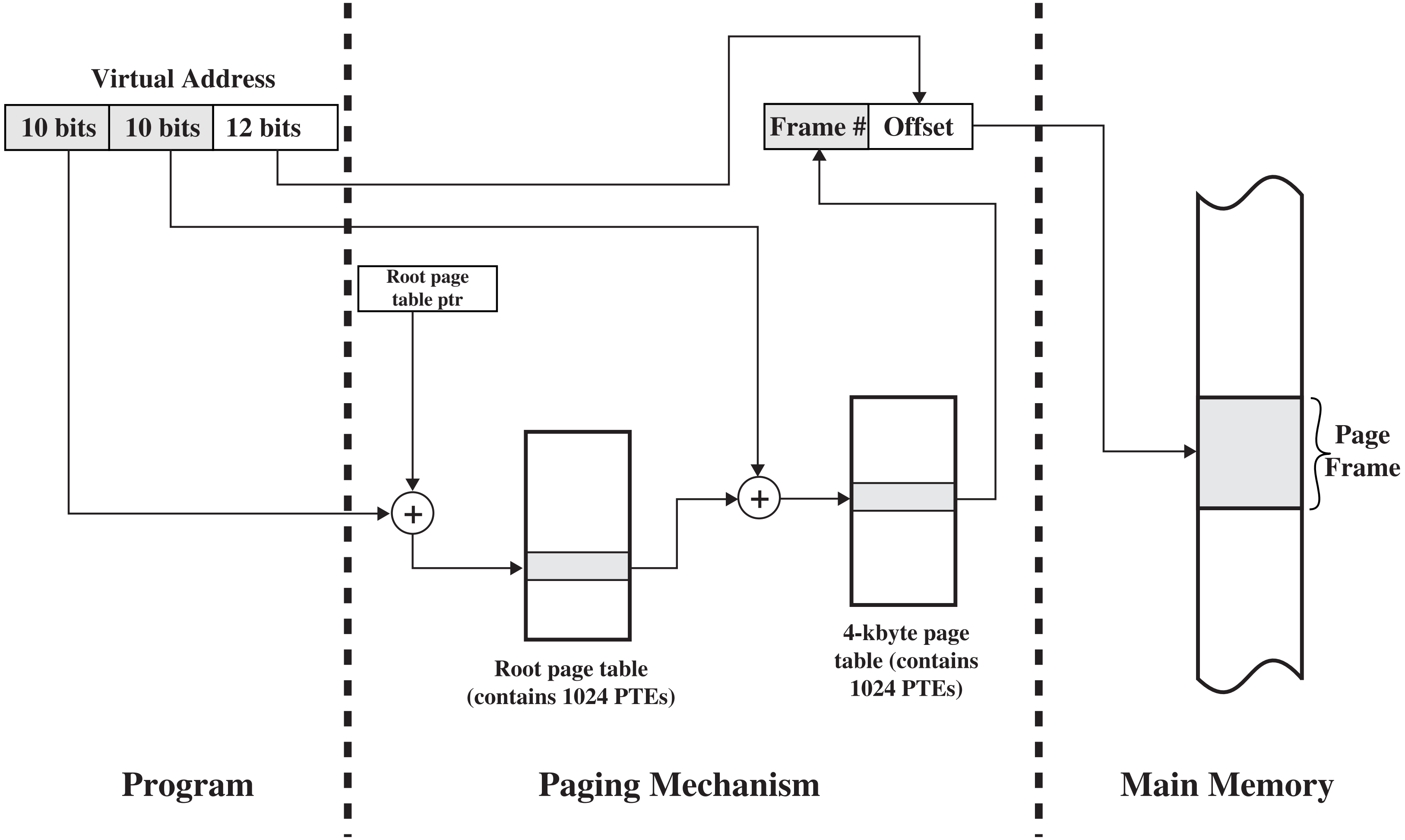
- Even a single process' page table is big: ≥ 4MB for 32-bit address space
- Most processes do not use a full 4GB address space
  - e.g., 0.1–1MB text, 0.1–10MB data, 0.1MB stack
- Need compact representation that doesn't waste space
- Three basic schemes:
  - use data structures which adapt to sparsity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

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

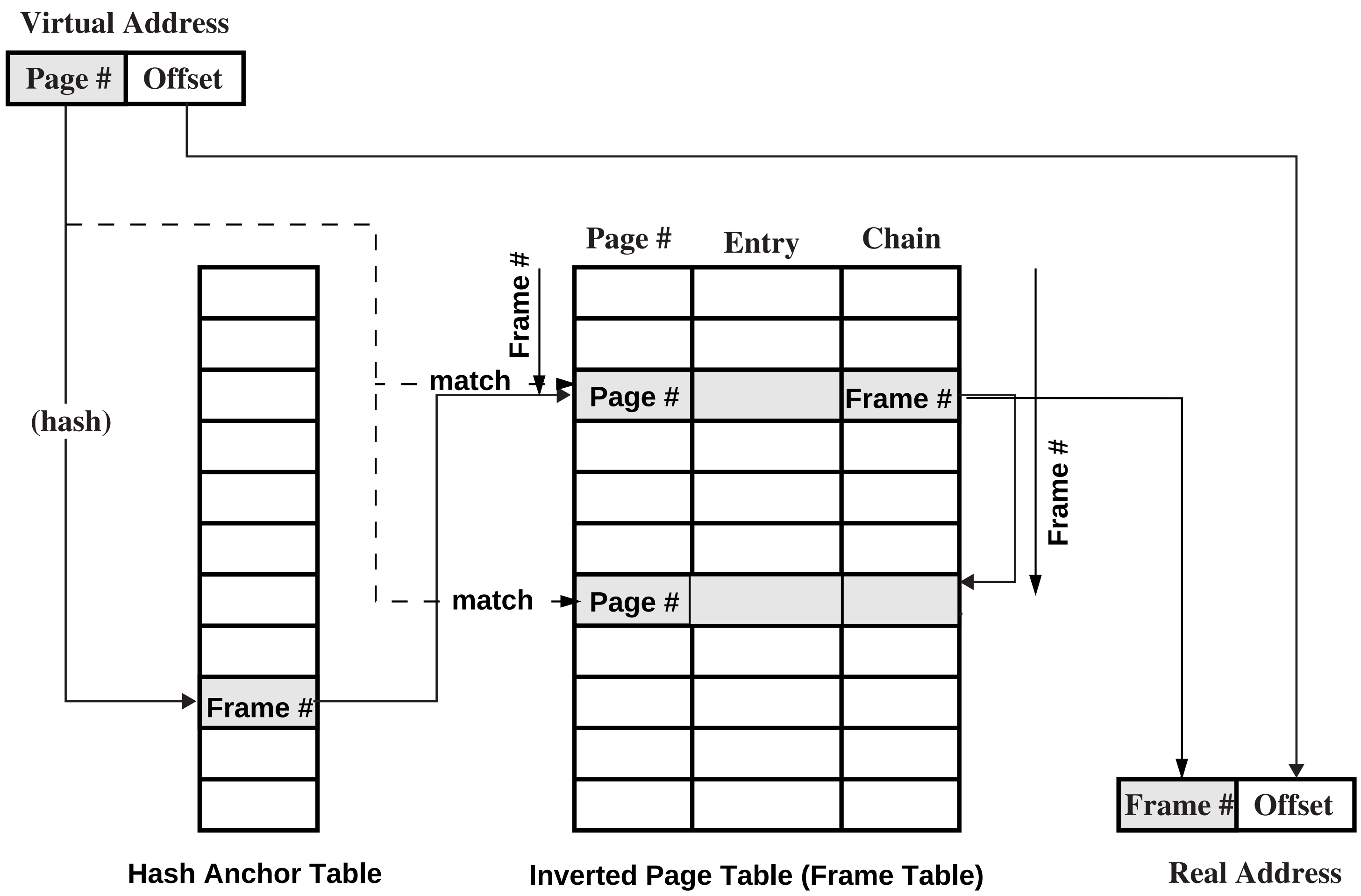


- Note: Unused PT pages not allocated (NULL pointer in root page table)

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



Alternative: Inverted page table:



Inverted page table (IPT) operation::

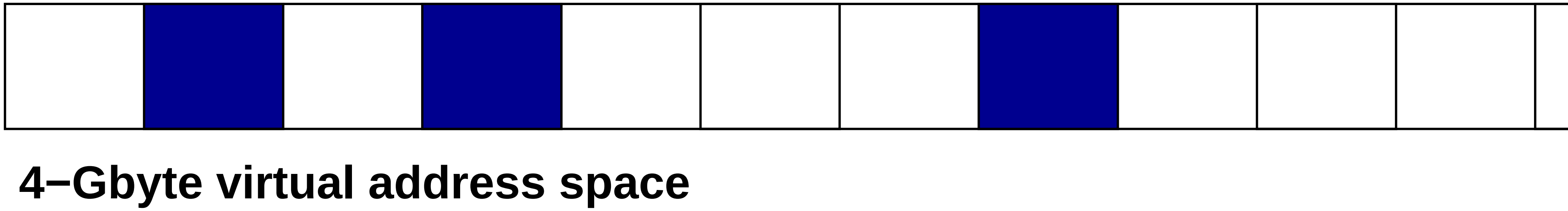
- "Inverted page table" is an array of page numbers sorted by frame number (it's a frame table)
- Algorithm:
  1. Compute hash of page number (usually least significant bits)
  2. us this to index into the hash anchor table (HAT)
  3. HAT contains candidate frame number
  4. use this to index into frame 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 $\Rightarrow$ 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

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
  $\Rightarrow$ 2^{nd} level nodes then form a contiguous array

4−Gbyte virtual address space

Virtual linear array page table operation:



- ➜ 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)

Shared pages

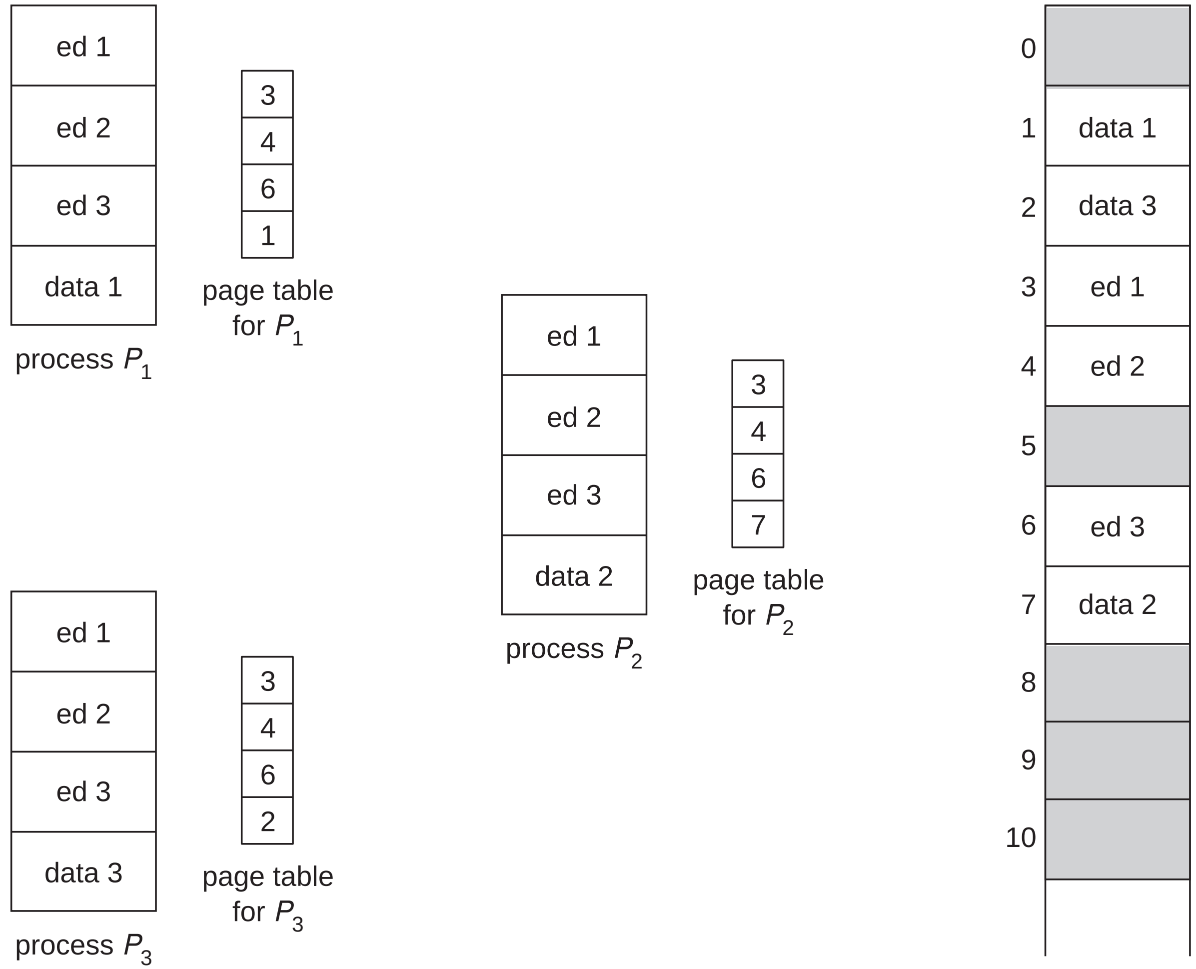
➜ Private code and data

- 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)

Example: Shared pages:



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
$\ldots$	$\ldots$	.	.
$\ldots$	$\ldots$	.	.

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

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

C0_PAGEMASK

Bits	127–121	120–109	108–96
Field		MASK	0
Width	7	12	13

C0_ENTRYHI

Bits	95–77	76	75–72	71–64
Field	VPN2	G	0	ASID
Width	19	1	4	8

C0_ENTRYLO1

Bits	63 62	62–38	38–35			32
Field	0	PFN	C	D	V	G
Width	2	24	3	1	1	1

C0_ENTRYLO0

Bits	31 30	30–6	6–3			0
Field	0	PFN	C	D	V	G
Width	2	24	3	1	1	1

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 $(2\times\texttt{VPN2})$
  - `ENTRYLO1` maps page $(2\times\texttt{VPN2}+1)$

`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

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

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

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

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

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

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

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

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	$\leftarrow$	TLBL	; *if read invalid fault*
CP0.CAUSE.ExcCode	$\leftarrow$	TLBS	; *if write invalid fault*
CP0.CAUSE.ExcCode	$\leftarrow$	Mod	; *if write protect fault*
CP0.PC	$\leftarrow$	0x8000 0180	; *gen excpt handler*

➜ Software must:

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