Slide 1

Virtual Memory

COMP3231 Operating Systems

2005/S2

Slide 2

Basic Features of Virtual Memory

➜ Piecewise-linear mapping of virtual to physical addresses

- Paging: fixed and (usually) equal-sized blocks
  - only small internal fragmentation
  - programmer not aware of technique used
- Segmentation: variable-sized blocks
  - external fragmentation
  - programmer aware of technique used
  - can be combined with fixed-sized

Slide 3

➜ Mapping is
  - defined at run-time, and
  - can change

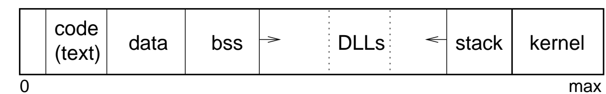
➜ Process does not have to be contiguous in physical memory

➜ 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 4

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

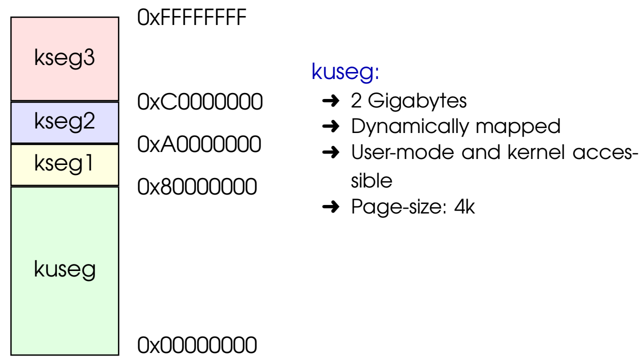
➜ shared libraries (DLLs) allocated in free middle region

➜ stack at top of user space, grows downward

➜ kernel space is in reserved (shared) region

Slide 5

MIPS R3000 Address Space Layout

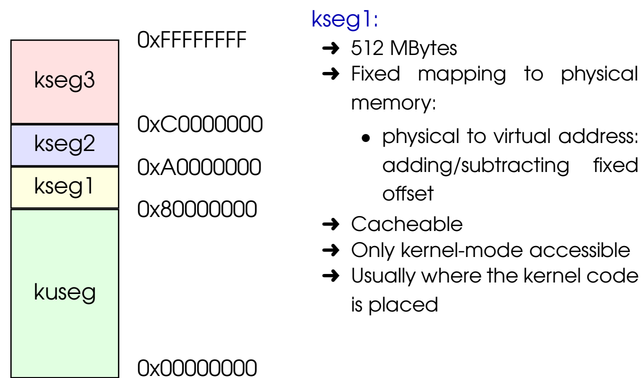


kuseg:

- ➜ 2 Gigabytes
- ➜ Dynamically mapped
- ➜ User-mode and kernel accessible
- ➜ Page-size: 4k

Slide 6

MIPS R3000 Address Space Layout

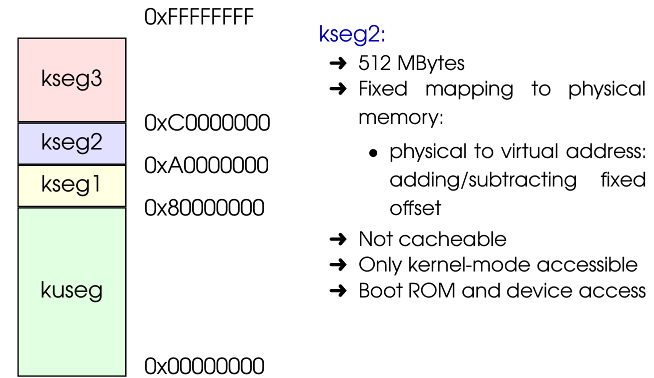


kseg1:

- ➜ 512 MBytes
- ➜ Fixed mapping to physical memory:
  - physical to virtual address: adding/subtracting fixed offset
- ➜ Cacheable
- ➜ Only kernel-mode accessible
- ➜ Usually where the kernel code is placed

Slide 7

MIPS R3000 Address Space Layout

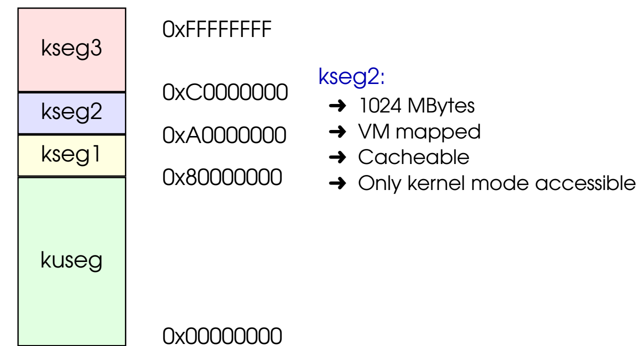


kseg2:

- ➜ 512 MBytes
- ➜ Fixed mapping to physical memory:
  - physical to virtual address: adding/subtracting fixed offset
- ➜ Not cacheable
- ➜ Only kernel-mode accessible
- ➜ Boot ROM and device access

Slide 8

MIPS R3000 Address Space Layout

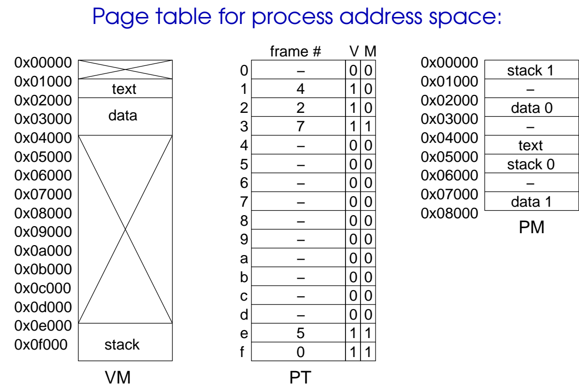


kseg2:

- ➜ 1024 MBytes
- ➜ VM mapped
- ➜ Cacheable
- ➜ Only kernel mode accessible

Slide 9

Page table for process address space:



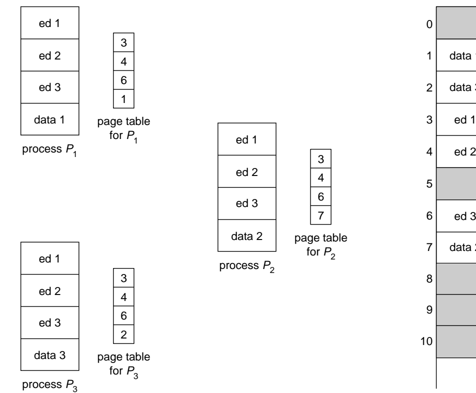
Slide 10

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 11

Example: Shared pages:



Slide 12

Page table structure:

- ➜ Page table is (logically) a mapping from page numbers to frame 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

Slide 13

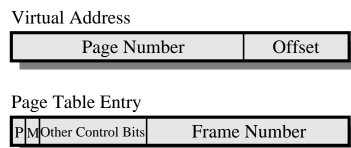
Referencing an invalid page triggers a page fault:

- ➜ 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 14

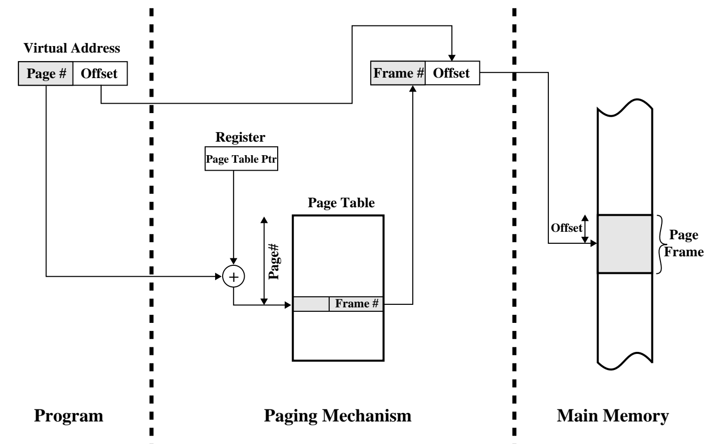
Address Translation (Paging)

- ➜ 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 15

Address translation in a paging system:



Slide 16

Page Tables

Assume we have

- ➜ 32 bit virtual addresses (4 GByte address space)
- ➜ 4 KByte page size (2^{12})
- ➜ How many entries can the page table have for one process?

Problem:

- ➜ Page table is very large
- ➜ Access has to be fast, lookup for every memory reference
- ➜ Where to store the page table?
  - register?
  - main memory?

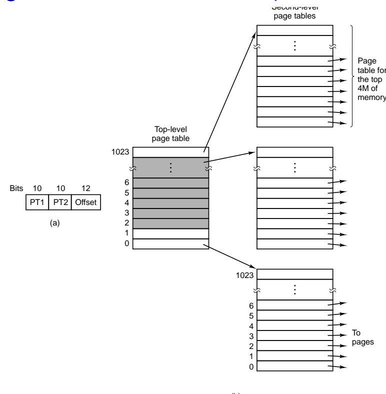
Slide 17

PAGE TABLES

- ➜ 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 18

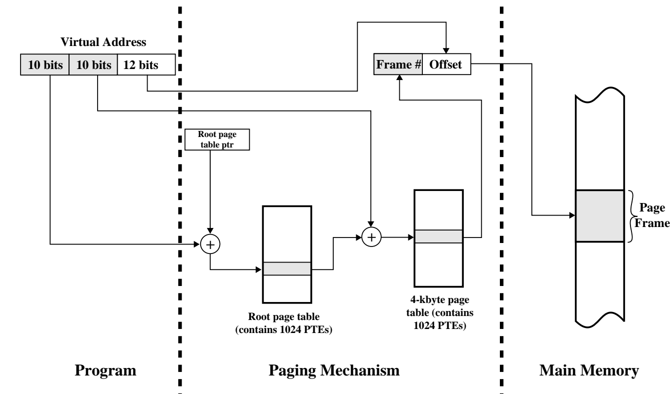
Two-level page table (32-bit address spaces):



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

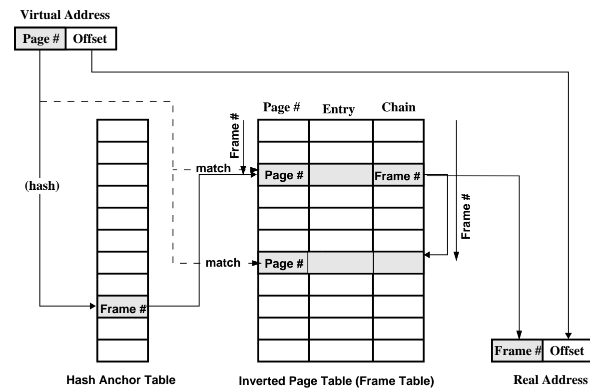
Slide 19

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



Slide 20

Alternative: Inverted page table



Slide 21

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

Slide 22

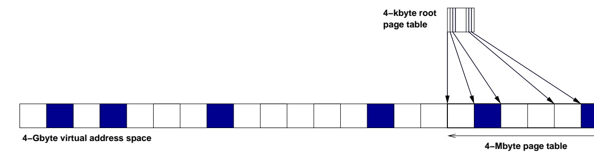
Properties of Inverted Page Tables

- ➜ 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
- ➜ Saves vast amount of space (esp. in 64bit Address systems)
- ➜ Searching does not come for free—efficiency?
- ➜ Currently used in some IBM and HP workkstations

Slide 23

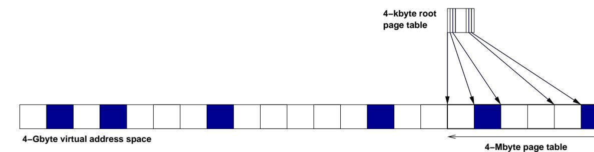
Alternative: Virtual linear array page table:

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



Slide 24

Virtual linear array page table operation:



- ➜ Index into 2nd 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 25

Problem:

- 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)

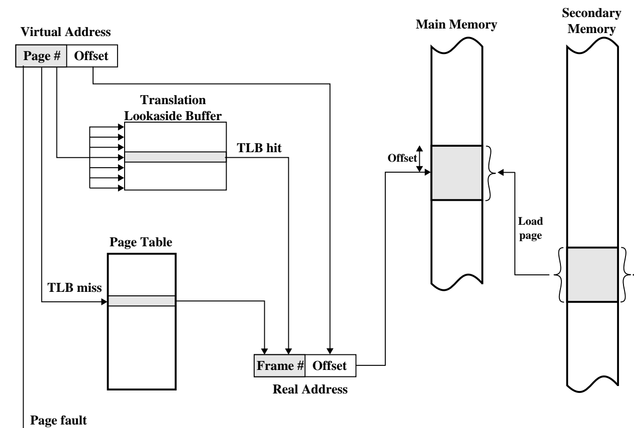
Slide 26

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 27

TLB operation::



Slide 28

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
- TLB can also be used for inverted page tables

Page Tables and the TLB

Slide 29

- ➜ 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 30

Page tables and the TLB:

- ➜ TLB is a shared piece of hardware
- ➜ Page tables are per-process (address space)
- ➜ TLB entries are process-specific

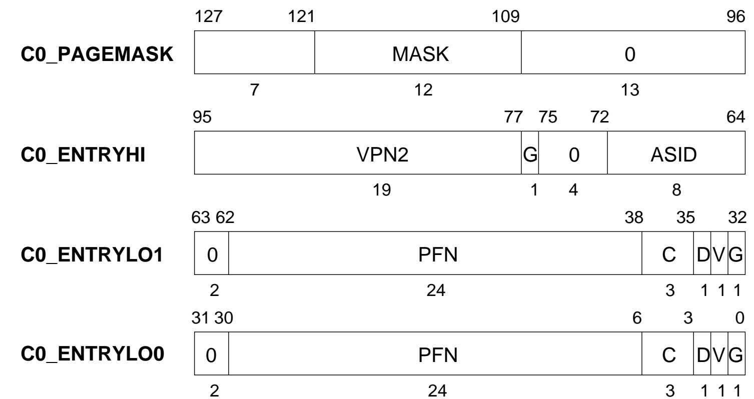
Solutions:

① On context switch need to flush the TLB (invalidate all entries)
- high context-switching overhead (ix86)

② **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 31

Example: MIPS R4x00 TLB entry (32-bit mode): (R3000 similar, see Hardware Guide on course web page)



Slide 32

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×`VPN2`)
  - `ENTRYLO1` maps page (2×`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

Slide 33

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 (≥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 34

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 35

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 36

TLB miss handling...:

- Software does:
  ① Look up PTE corresponding to fault address
  ② if found:
    ① load `C0_ENTRYL00` and `C0_ENTRYL01` with translations
    ② load TLB using `tlbwr` instruction
    ③ return from exception
  ③ else handle page fault
- TLB entry (i.e., `C0_ENTRYL00`, `C0_ENTRYL01`) can be:
  - created on the fly, or
  - stored completely and in the right format in the PT

Slide 37

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

Slide 38

OS/161 Refill Handler:

➜ Switches to kernel stack

➜ Calls common exception handler

- Unoptimised
- Written for ease of kernel programming, not efficiency

➜ Does not have a page table

- If 64 TLB entries are full, kernel panics
- supports only 256K user-level address space

Slide 39

SEGMENTATION

➜ Variable-sized segments instead of fixed-size pages

➜ Size may be dynamic (changeable at run time)

➜ Natural support for modularisation (dynamic libraries)

➜ Lends itself to sharing along logical module boundaries

➜ Lends itself to module-based protection

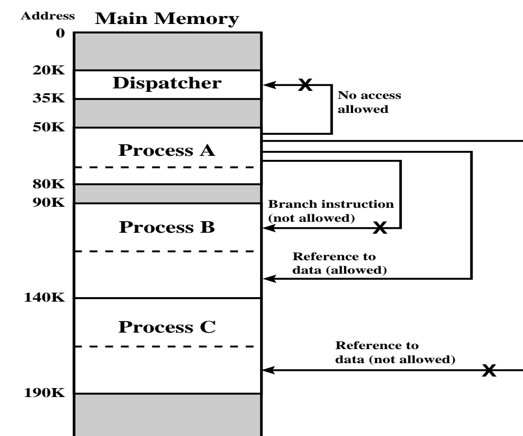
➜ External fragmentation but no internal fragmentation

➜ Introduce a non-linear address space (segment #, offset)

➜ Per-process segment table instead of page table

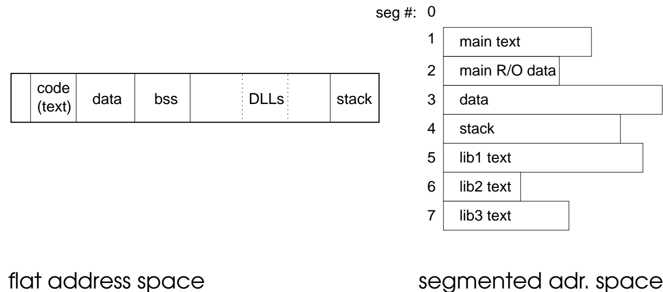
Slide 40

Protection using segments:



Slide 41

Flat vs. segmented address space::



flat address space | segmented adr. space

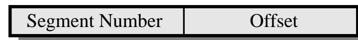
no protection... | protection...

...against segment overrun

Slide 42

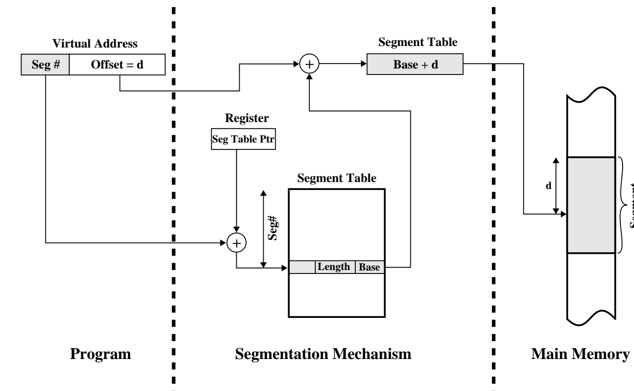
Segment Tables:

- ➜ Translates segment number to main memory address
- ➜ Each entry contains:
  - physical start address of segment
  - segment length
  - valid bit
  - dirty bit



Slide 43

Address translation in a segmentation system:



Slide 44

Combined Paging and Segmentation

- ➜ Paging is transparent to the programmer
- ➜ Paging eliminates external fragmentation
- ➜ Paging supports (relatively) fine-grain memory management
- ➜ Segmentation is visible to the programmer
- ➜ Segmentation allows for growing data structures, modularity, and module-based support for sharing and protection
- ➜ Combination: Segments broken into fixed-size pages
- ➜ Examples: Pentium, PowerPC, HP PA-Risc, IA-64

Slide 45

Combined segmentation and paging:

➜ Per-process segment table
➜ Per-segment page table

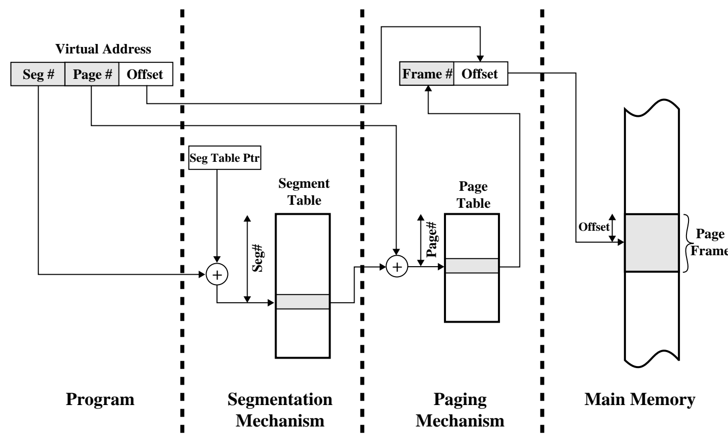
Virtual Address

Segment Table Entry

Page Table Entry

Slide 46

Address translation with paged segments:



Slide 47

Address-space switch

➜ The address space is defined by page or segment table(s)
➜ Part of that information is cached in the TLB
➜ Data and instructions are cached in the CPU caches
➜ Switching AS (during a context switch):

- The page/segment table pointer needs to be changed
- **if** the TLB is tagged:
  - The ASID register needs to be reloaded (`C0_TLBHI` on MIPS)
- **otherwise**:
  - The TLB must be flushed (all entries invalidated)
- **if** the CPU caches are **not** tagged:
  - The caches must be flushed

Slide 48

Demand Paging/Segmentation

➜ With VM, only parts of the program image need to be resident for execution
➜ Can swap presently unused pages/segments to disk
➜ Reload non-resident pages/segments 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 currently resident page/segment
  - How determine replacement "victim"?
- If victim is unmodified ("clean") can simply discard
  - This is the reason for maintaining the "dirty" bit in the PT

Slide 49

Why does demand paging/segmentation work?

- ➜ Program executes at full speed while only accessing resident set of pages or segments
- ➜ TLB miss introduces delay of several micro-seconds
- ➜ Page/segment fault introduces delay of several milli-seconds
- ➜ Why do it?

Answers:

① Less physical memory required per process

- ⇒ can fit more processes into memory
- ⇒ improved chance of finding a runnable process

② Principle of locality

Slide 50

Principle of Locality:

- ➜ Process' program and data references tend to cluster
- ➜ Only a small number of pages/segments will be needed during a (short) time window
  - Called the memory working set of a process
- ➜ System keeps at least working set of process resident
  - ➨ process can execute while it stays within working set
  - Working set tends to change gradually
    - Get only a few page/segment faults during time window
  - Possible to make intelligent guesses about which pieces will be needed in the future
    - May be able to pre-fetch pages/segments

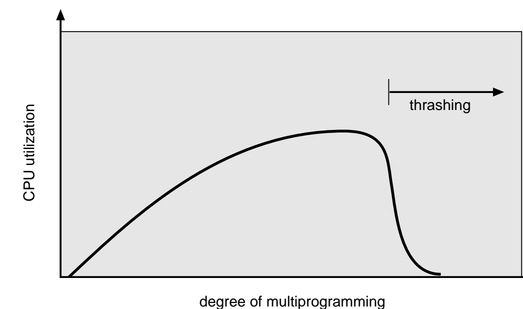
Slide 51

Thrashing:

- ➜ CPU utilisation tends to increase with the degree of multiprogramming
  - number of processes in the system
- ➜ Higher degree of multiprogramming—less memory available per process
- ➜ Some process's working sets may no longer fit in RAM
  - ➨ increasing page fault rate
- ➜ Eventually many processes have insufficient memory
  - can't always find runnable process
  - decreasing CPU utilisation
  - system is I/O limited
- ➜ This is called thrashing

Slide 52

Thrashing...:



Thrashing: Σ resident sets $<$ Σ working sets

Slide 53

Remember this?:

```
void ResetArray (int array[10000][10000]) {
    int i, j;

    for (i=0; i<10000; i++) {
        for (j=0; j<10000; j++) {
            array[i][j] = 0;
            /* OR array[j][i] = 0 ??? */
        }
    }
}
```

What's the difference?

Slide 54

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
  - hashed
- ➜ Page size (may be dictated by hardware)
- ➜ Fetch policy
- ➜ Replacement policy
- ➜ Resident set size
  - minimum allocation
  - local vs global allocation
- ➜ Page cleaning policy
- ➜ Degree of multiprogramming ...

Slide 55

Page Size

Increasing page size:

- ✘ increases internal fragmentation
  - reduces adaptability to working set
- ✔ 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/Os are dominated by seek time/rotational delays

Optimal page size is a (workload-dependent) tradeoff!

Slide 56

Example Page Sizes:

Architecture	Page Size
Atlas	512 words (48-bit)
Honeywell/Multics	1k words (36-bit)
IBM 370/XA, 370/ESA	4k bytes
DEC VAX	512 bytes
IBM AS/400	512 bytes
Intel Pentium	4k and 4M bytes
ARM	4k and 64k bytes
MIPS R4x00	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 plus "blocks"
Intel IA-64	4k–256M bytes in powers of 4, etc

Slide 57

Page Size:

- ➜ Multiple page sizes provide the flexibility to optimise the use of the TLB
- ➜ E.g.:
  - large pages can be used for code
  - small pages for thread stacks
- ➜ Most operating system support only one page size
  - Dealing with multiple page sizes is hard!

Slide 58

Fetch Policy

- ➜ Determines when a page should be brought into memory
  - Demand paging only loads pages in response to page faults
    - Many page faults when process first started
  - Pre-paging brings in more pages than needed at the moment
    - improve I/O performance by reading larger chunks
    - pre-fetch when disk is idle
    - wastes I/O bandwidth if pre-fetched pages aren't used

Slide 59

Replacement Policy

- ➜ Which page is chosen to be tossed out?
  - Page removed should be the page least likely to be referenced in the near future
  - Most policies attempt to predict the future behavior on the basis of past behaviour
- ➜ Constraint: locked frames:
  - kernel code
  - main kernel data structures
  - I/O buffers
  - performance-critical user pages (e.g. for DBMS)
- ➜ Frame table has lock bit

Slide 60

Basic Replacement Policies

Optimal:

- ➜ Toss the page that won't be used for the longest time
- ➜ Impossible to implement
- ➜ Only good as a theoretical 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?

Slide 61

Basic Replacement Algorithms: FIFO:

- ➜ First-in, first-out: Toss oldest page
  - ✔ Easy to implement
  - ✘ Age of a page isn’t necessarily related to its usage

Example:

- ➜ reference string: 1 2 3 4 1 2 5 1 2 3 4 5
- ➜ four frames
- ➜ how many page faults?
- ➜ three frames

Belady’s anomaly: more frames $\not\Rightarrow$ fewer page faults

Slide 62

Basic Replacement Algorithms: LRU

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

How many page faults for example sequence?

- ➜ reference string: 1 2 3 4 1 2 5 1 2 3 4 5

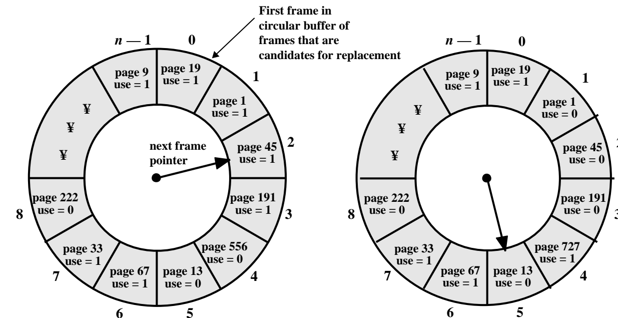
Slide 63

Basic Replacement Algorithms: Clock

- ➜ Clock policy, also called second chance
  - Employs a usage or reference bit in frame table
  - Set to one when page is used
  - When scanning for a victim, reset all reference bits
  - Toss first page with zero reference bit.
- ➜ How do we know when a page has been referenced?
- ➜ Use the valid bit in the PTE:
  - when page is mapped (valid bit set), set reference bit
  - when resetting reference bit, invalidate PTE entry
  - on TLB fault:
    - turn on valid bit in PTE
    - turn on reference bit in frame table

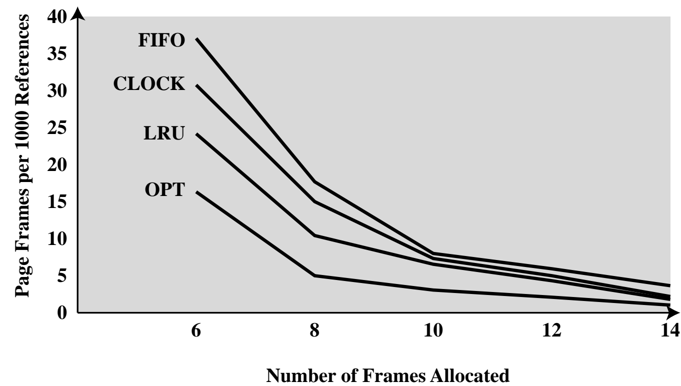
Slide 64

Example of operation of clock policy:



Slide 65

Performance



Slide 66

Note: there are other algorithms (working set, ageing, NFU, . . .) — we don't expect you to know them in this course)

Slide 67

Basic Replacement Algorithms: Page buffering

- ➜ Replace pages before running out of memory
- ➜ "Replaced" frame is added to one of two lists
  - free-frame list if page has not been modified
  - modified-frame list if dirty
- ➜ clean (write back) modified frames asynchronously
  - migrates frames from modified to free list
- ➜ when in need of a frame, get it from free-frame list
- ➜ on page fault check both lists first

Slide 68

Resident Set Size

- ➜ Fixed allocation
  - gives a process a fixed number of pages within which to execute
  - when a page fault occurs, one of the pages of that process must be replaced
- ➜ Variable allocation
  - number of pages allocated to a process varies over the lifetime of the process

Slide 69

Variable Allocation, Global Scope:

- ➜ Easiest to implement
- ➜ Adopted by many operating systems
- ➜ Operating system keeps 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

Slide 70

Variable Allocation, Local Scope:

- ➜ Allocate number of page frames to new processes based on
  - application type
  - program request,
  - other criteria...
- ➜ When page fault occurs, select page from among the resident set of the process that suffers the fault
- ➜ Reevaluate allocation from time to time

Slide 71

CLEANING POLICY

- ➜ Demand cleaning
  - a page is written out only when it has been selected for replacement
- ➜ Precleaning
  - pages are written out in batches
  - generally used with frame buffering

Slide 72

LOAD CONTROL (DEGREE OF MULTIPROGRAMMING)

- ➜ Determines the number of runnable processes
- ➜ Controlled by:
  - Admission control:
    - only let new process's thread proceed from new to ready state if enough memory available
  - Suspension:
    - move all threads of some processes into special suspended state
    - swap complete process image of suspended processes to disk
- ➜ Tradeoff:
  - Too many processes will lead to thrashing
  - Too few will lead to idle CPU/excessive swapping