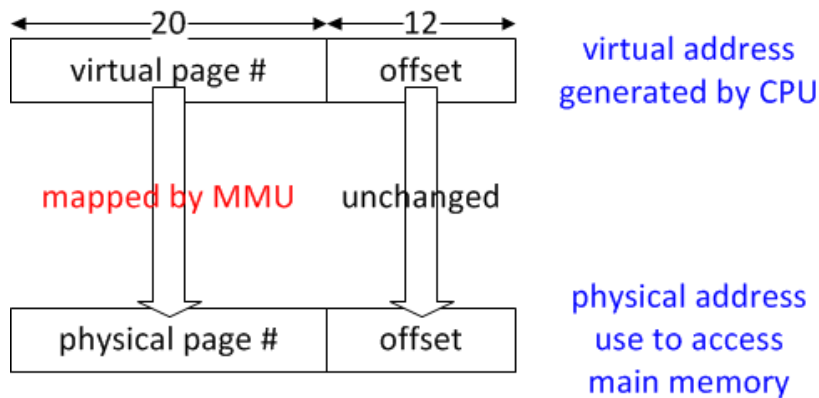


MEMORY MANAGEMENT UNITS

Memory Management Units

- memory management unit (MMU) converts a *virtual* address generated by a CPU into a *physical* address which is applied to the memory system



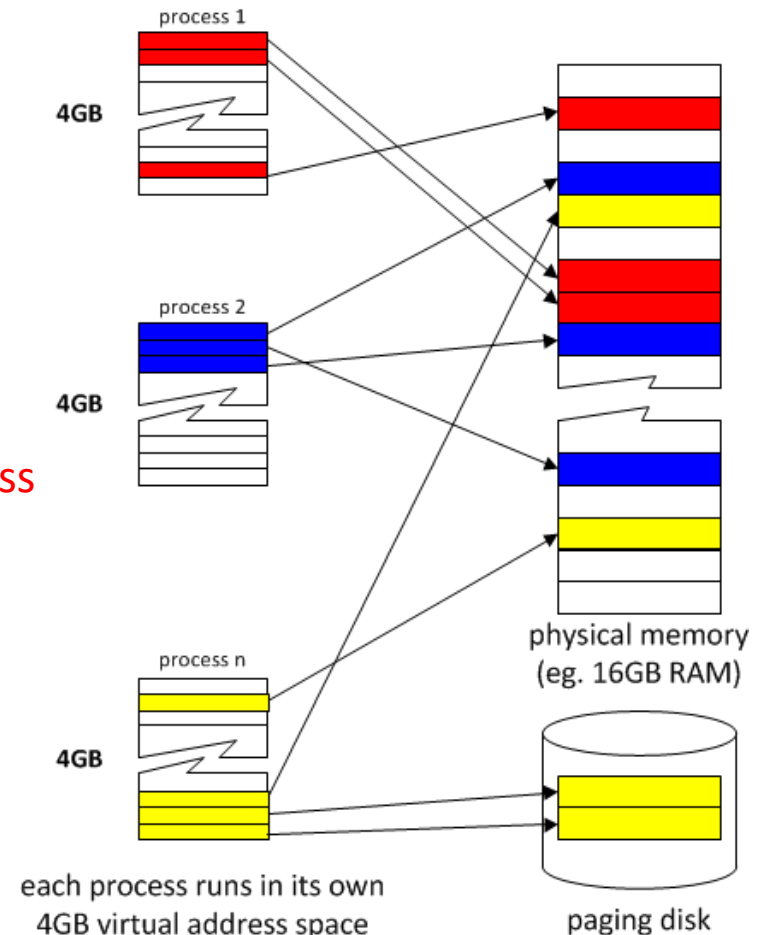
- address space divided into fixed sized pages [eg. 4Kbytes]
- low order address bits [**offset within a page**] not effected by MMU operation
- virtual page # converted into a physical page #

Memory Management Units...

- MMUs integrated on-chip with the CPU
- each CPU core will typically have separate MMUs for instruction and data accesses
- examples as per IA32
 - 2^{32} byte [4GByte] address space divided into... 2^{20} [1,048,576] \times 2^{12} [4K] byte pages
- virtual and physical address spaces need NOT be the same size
- which would you prefer?
 - virtual > physical
- OR...
- physical > virtual

Mapping Virtual Address Spaces onto Physical Memory [IA32]

- each process runs in own 4GB virtual address space
- pages in each virtual address space mapped by MMU onto real physical pages in memory
- pages allocated and mapped on demand by Operating System (OS)
- virtual pages [in a process] may be
 - not allocated/mapped [probably because process hasn't accessed that virtual page yet]
 - allocated in physical memory
 - allocated on paging disk
- typical Windows 7 process memory usage
 - Word 43MB, IE 15MB, Firefox 27MB, ...
- small fraction of 4GB virtual address space



Mapping Virtual Address Spaces onto Physical Memory

- Atlas Computer 1962 [**Manchester University**] first to support virtual memory
 - 48bit CPU, 24bit virtual and physical address spaces, 96KB RAM, 576KB drum [**disk**]
- OS normally attempts to keep the "*working set*" of a process in physical memory to minimise the page-fault rate [**thrashing**]
- every page used in a process' virtual address space requires an equivalent page either in physical memory or on the paging disk
- 4GB [**total**] of physical memory and paging disk space needed for a program which uses/accesses all 4GB of its virtual address space [**e.g. large array**]
- can view physical memory as acting as a cache to the paging disk!

Memory Cruncher

- consider the following program outline

```
#define GB (1024*1024*1024)
```

```
char *p = malloc(4*GB);           // just moves internal OS pointer
```

```
for (size_t i = 0; i < 4*GB; i += PAGE_SIZE, p += PAGE_SIZE)  
    *p = 0;                       // access causes physical memory to be allocated
```

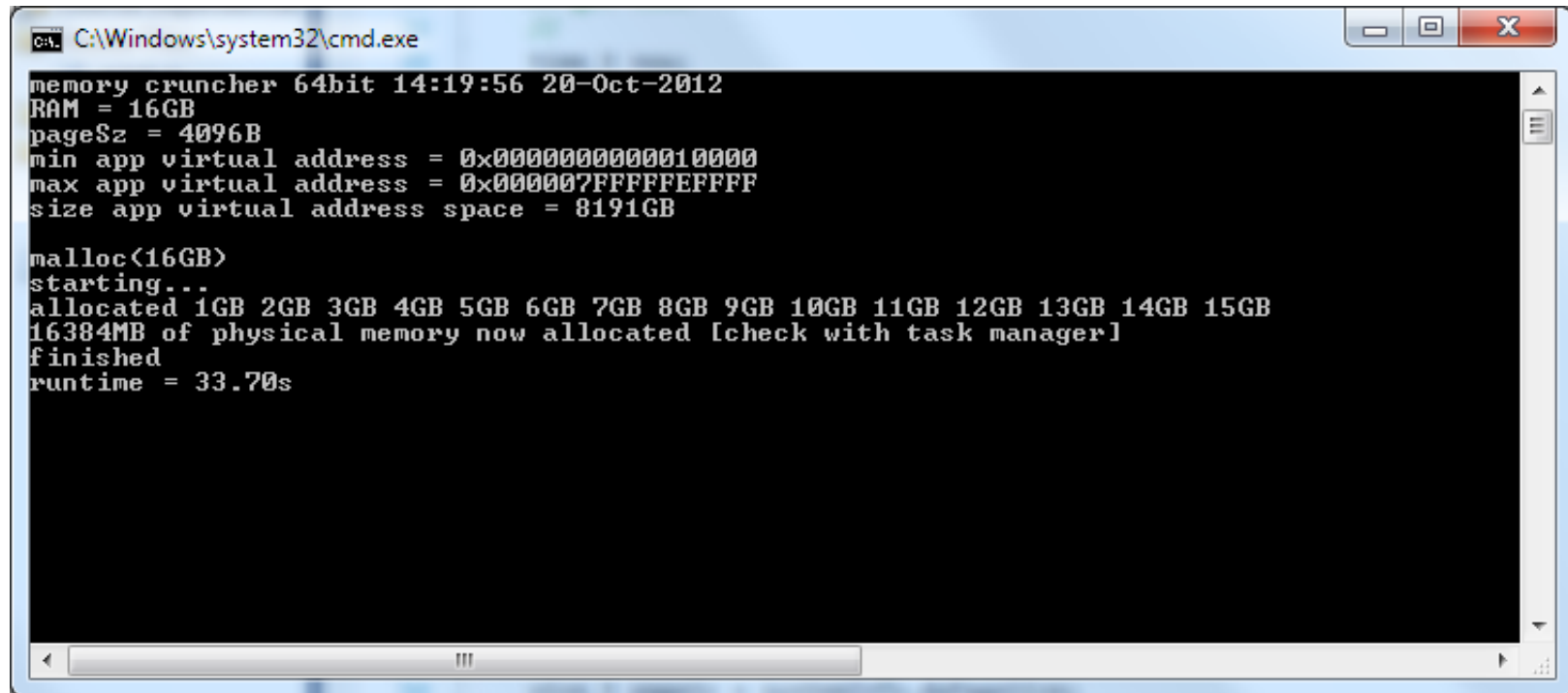
- a more complete version of [Memory Cruncher.cpp](#) is on the CS3021/3421 website
 - designed to run as a Win32 [32 bit] or x64 [64 bit] process
 - size_t is the size of an address [Win32 32 bits, x64 64 bits]
 - Windows PAGE_SIZE is 4K

Memory Cruncher...

- what is the largest contiguous memory block that can be allocated?
- Windows 7 Win32
 - 4GB virtual address space, bottom 2GB for user and top 2GB for OS
 - can malloc() a 1535MB contiguous memory block
 - right click on project name [Properties][Linker][System][EnableLargeAddresses]
can now malloc() a 2047MB contiguous memory block
- Windows 7 x64
 - program reports it can allocate a contiguous memory block of 8191GB or 8TB [2⁴³]
 - *mallocing* a block much greater than size of physical memory [16GB] results in PC becoming extremely unresponsive [had to reboot by turning off power]
 - RUN with caution

MEMORY MANAGEMENT UNITS

Memory Cruncher...



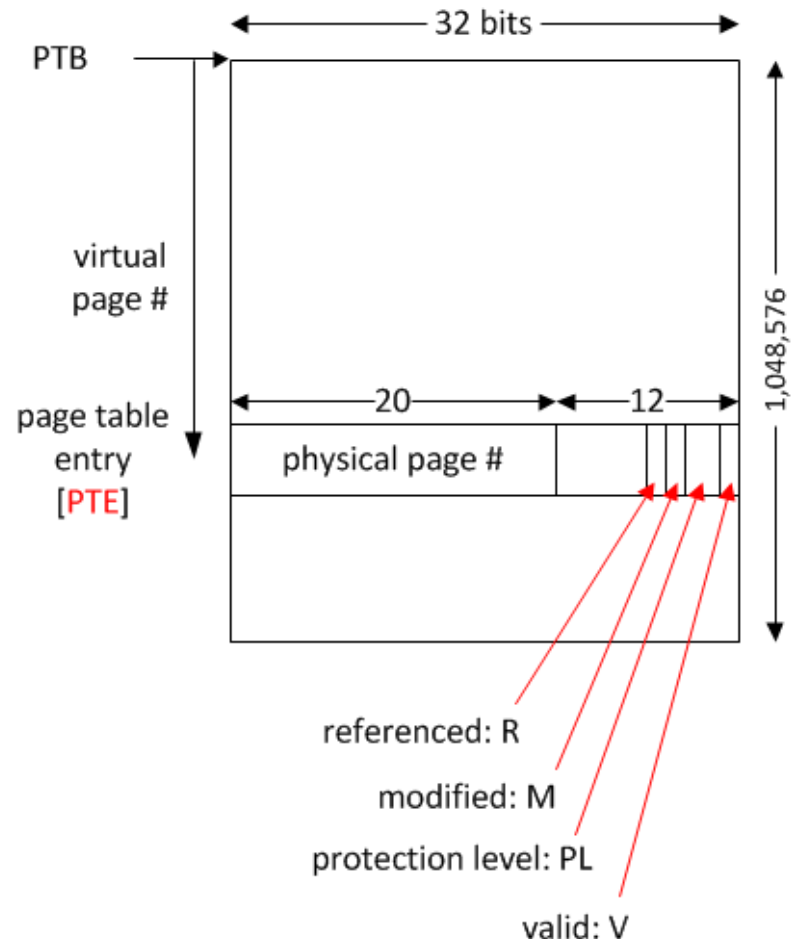
```
C:\Windows\system32\cmd.exe
memory cruncher 64bit 14:19:56 20-Oct-2012
RAM = 16GB
pageSz = 4096B
min app virtual address = 0x0000000000001000
max app virtual address = 0x000007FFFFFFF
size app virtual address space = 8191GB

malloc(16GB)
starting...
allocated 1GB 2GB 3GB 4GB 5GB 6GB 7GB 8GB 9GB 10GB 11GB 12GB 13GB 14GB 15GB
16384MB of physical memory now allocated [check with task manager]
finished
runtime = 33.70s
```

MEMORY MANAGEMENT UNITS

Generic MMU Operation [IA32, x64, MIPS, ...]

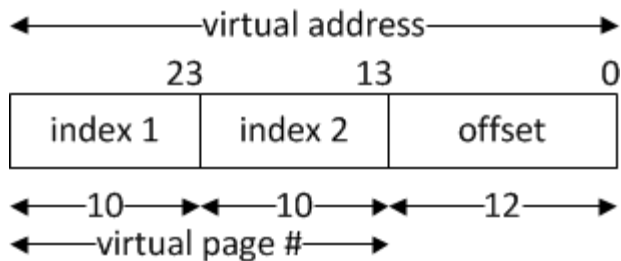
- virtual page # converted to a physical page # by table look-up
- virtual page # used as an index into a page table stored in physical memory.
- page table per process [**and sometimes one for OS**]
- page table base register PTB [**CR3 in IA32**] contains the physical address of the page table of the currently running process
- 4MB physical memory [**1,048,576 x 4**] needed for page table of every process
- **IMPRACTICAL**



MEMORY MANAGEMENT UNITS

N-level Page Table

- in order to reduce the size of the page table structure that needs to be allocated to a process, a n-level look-up table is used
- a n-level page table means that the "*larger*" the process [in terms of its use of its virtual address space], the more memory is needed for its page tables
- consider a 2-level scheme

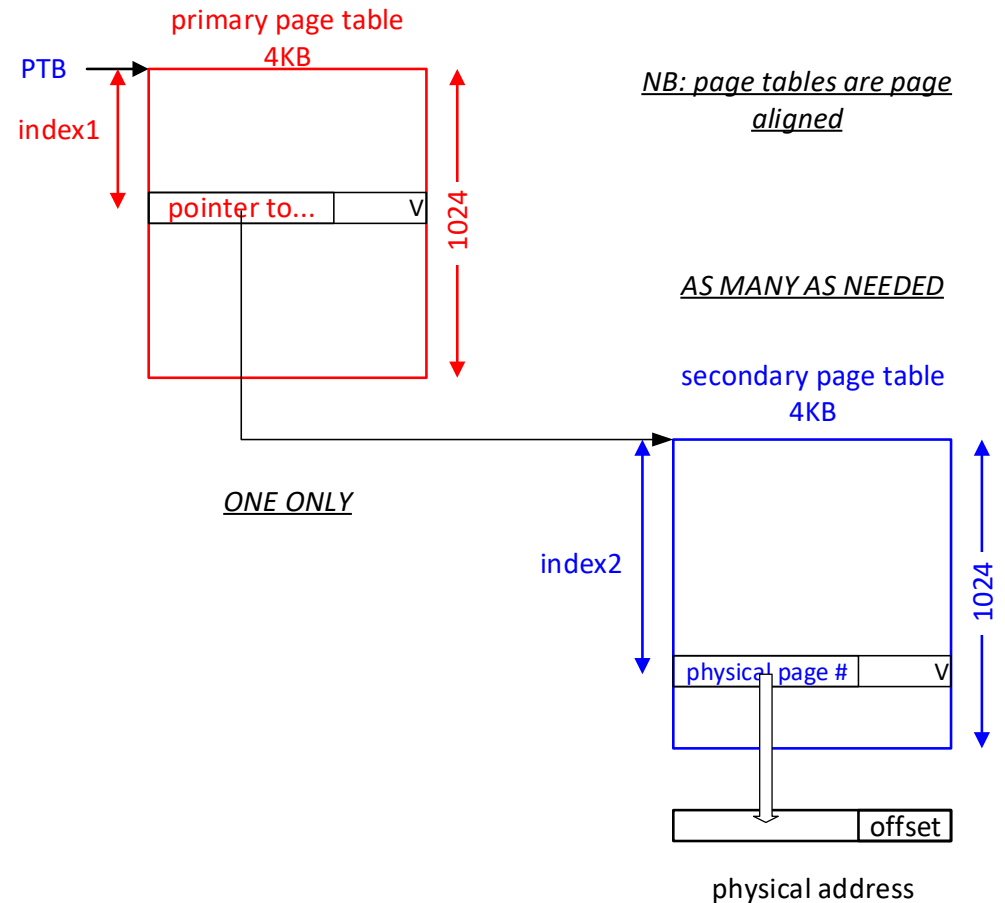


- *index1* is used to index into a primary page table, *index2* into a secondary page table and so on...

MEMORY MANAGEMENT UNITS

N-Level Page Table...

- PTB points to primary page table of currently running process
- a valid primary page table (PTE) entry points to a secondary page table
- each process has one primary page table + multiple secondary page tables
- secondary page tables created on demand [**depends on how much of its virtual address space the process uses**]
- NB: size of page tables is 4KB - the page size itself



Generic MMU Operation...

- when MMU accesses a PTE it checks the Valid bit
- if accessing a primary PTE and $V == 0$ (i.e. PTE is invalid)
 - then NO physical memory allocated for corresponding secondary page table
- If accessing a secondary PTE and $V == 0$
 - then NO physical memory allocated for referenced page [i.e. virtual address NOT mapped to physical memory]
- in both cases a "page fault" occurs, the instruction is aborted and the MMU interrupts the CPU

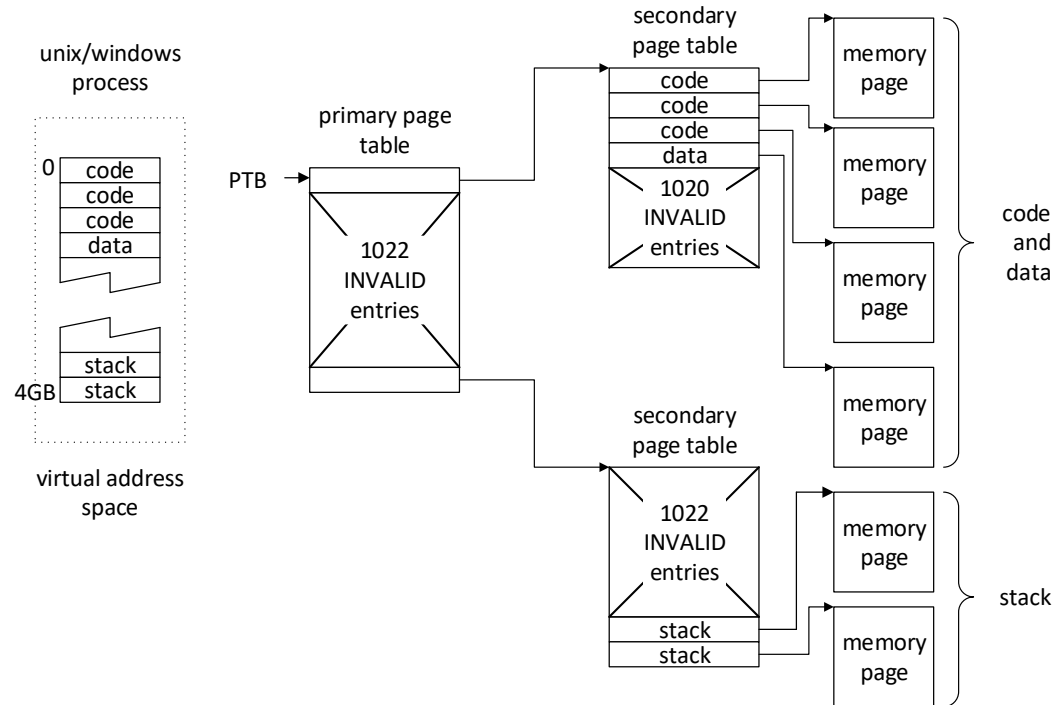
Page fault handling

- OS must resolve page fault by performing one OR more of the following actions:
 - allocating a page of physical memory for use as a secondary page table [**from an OS maintained list of free memory pages**]
 - allocating a page of physical memory for the referenced page
 - updating the associated page table entry/entries
 - reading code or initialised data from disk to initialise the page contents [**context switches to another process while waiting**]
 - signalling an access violation [**eg. writing to a read-only code page**]
 - restarting [**or continuing**] the faulting instruction

MEMORY MANAGEMENT UNITS

Process Page Table Structure

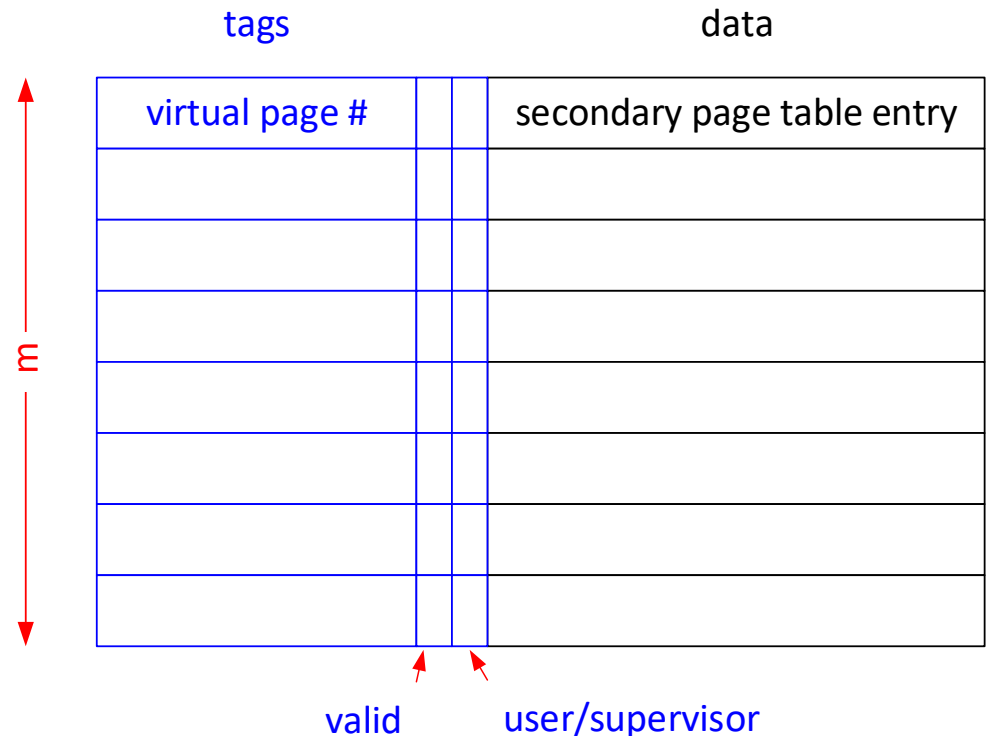
- example small process needs 3 code pages [12K], 1 data page [4K] and 2 stack pages [8K]
- code and data pages start at virtual address 0 with the stack at top of virtual address space
- require 2 secondary page tables to map code and stack areas [as at opposite ends of the virtual address space]
- a secondary page table can map $1024 \times 4K$ pages = 4MB
- need ONLY 2 secondary page tables providing program doesn't use more than 4MB of code/data and 4MB of stack space



MEMORY MANAGEMENT UNITS

Translation Look Aside Buffer [TLB]

- without an internal TLB, each virtual to physical address translation requires 1 memory access for each level of page table [2 accesses for a 2 level scheme]
- MMU contains an m-entry on-chip translation cache [TLB] which provides direct mappings for the m most recently accessed virtual pages



Translation Look Aside Buffer [TLB]...

- when a virtual address is sent to the TLB, the virtual page # is compared with ALL m tag entries in the TLB in parallel [a fully associative cache]
- if a match is found [TLB hit], the corresponding cached secondary page table entry is output by the TLB/MMU to provide the physical address
 - the address translation is completed "*instantaneously*"
- if a match is NOT found [TLB miss], page tables *walked* by CPU/MMU
 - IA32/x64 page tables walked by a hardware state machine hardwired into CPU/MMU
- the "*least recently used*" [LRU] TLB entry is replaced with the new mapping
- how can the hardware find the LRU entry *SIMPLY and QUICKLY?*

RISC TLB Miss Handling

- REMEMBER that the page tables are just data structures held in main memory and can be walked by a CPU using ordinary instructions
- this is the approach taken by many RISCs, a TLB miss generates an interrupt and the CPU walks the page table using ordinary instructions [TLB miss \equiv page fault]
- in such cases the organisation of page table structure is more flexible since it can be set by software and is NOT hard-wired into CPU/MMU [eg. could implement a hash table]
- need a CPU instruction to replace the LRU TLB entry
- TLBs are normally small
- a typical 64 entry fully associative TLB has a hit rate $> 90\%$
- a CPU would typically have a MMU for instruction accesses and a MMU for data accesses [needed for parallel accesses to the instruction and data caches]

TLB Coherency OS implications

- what happens on a process switch?
- TLB looked up by virtual address
- **ALL** processes use the same virtual addresses...

*e.g. process 0 virtual address 0x1000 is **NOT** mapped to the same physical memory location as process N virtual address 0x1000 unless the page is really shared*

- **ALL** TLB entries referring to the old process must be invalidated on a context switch otherwise the new process will access the memory pages of the old process
- normally the OS [**if it runs in its own virtual address space**] and one user process can share the entries in the TLB
- user/supervisor bit appended to TLB tag [**see diagram slide 14**]

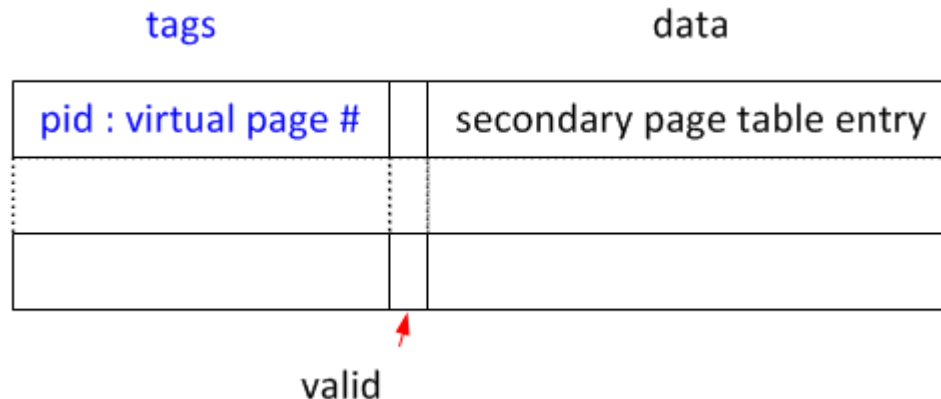
TLB Coherency OS implications...

- whenever the page table base register [eg. PTB0 for OS or PTB1 for the user process] is changed ALL corresponding TLB entries are invalidated
 - PTB1 changed every time there is a context switch between processes
 - PTB0 unlikely to change
- if a page table entry is changed in main memory [when handling a page fault], the OS must make sure that this change is reflected in the TLB
 - must be able to invalidate old PTEs in the TLB by executing an instruction
 - CPUs have an instruction to do this [eg. IA32 "*INVLPG va*" will invalidate PTE entry corresponding to the processes' virtual address *va* if present in TLB]
- also need to keep TLB entries in a multicore CPU coherent

MEMORY MANAGEMENT UNITS

Multiple Processes sharing TLB

- possible for processes to share TLB if a process ID (PID) is appended to the virtual page # as part of the TLB tag



- extension of user/supervisor bit as part of tag
- need to handle PID reuse as number of bits used for PID may be limited [e.g. 8 bits]

Referenced and Modified Bits

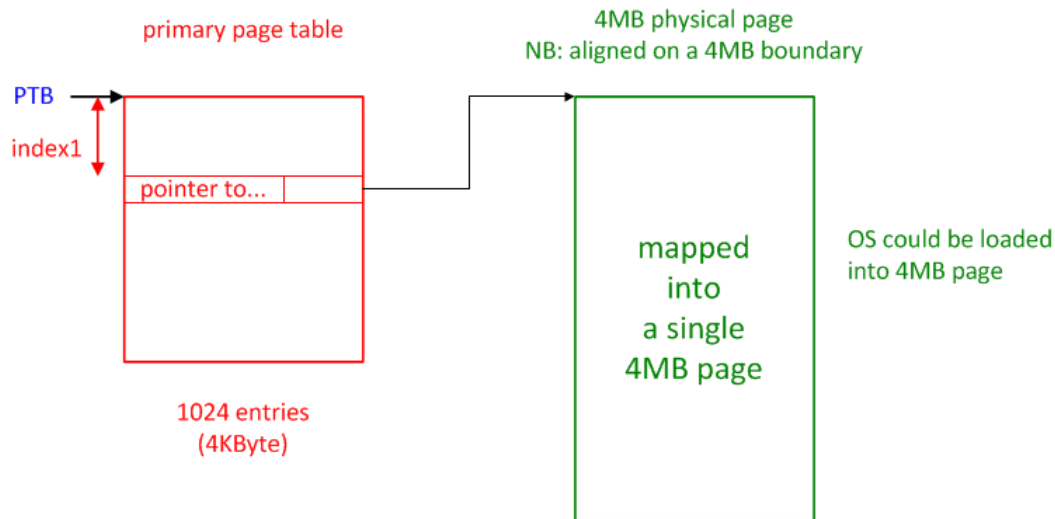
- CPU/MMU automatically updates the PTE Referenced and Modified bits [IA32/x64 Accessed and Dirty bits] in the PTEs
- PTE changes "*written through*" to corresponding PTE in physical memory
 - IA32/x64 CPU/MMU automatically executes these bus cycles
- IA32/x64 CPU/MMU never clears the reference and modified bits
 - up to the OS [eg. a background process that regularly clears the referenced bits]
- OS can use the Referenced and Modified bits to determine
 - which pages are good candidates for being paged out [eg. ones that have not been referenced recently]
 - if pages have to be written to the paging disk [eg. no need to save if a read only code page in executable file or may be unchanged since last saved]

Support for Different Page Sizes

- often useful if MMU supports a number of different page sizes
- one reason is that a TLB typically contains very few entries [32 or 64]
- *large pages* allows a single TLB entry map a *large* virtual page onto similar sized area of contiguous physical memory
 - OS could be loaded into a contiguous area of physical memory which could then be mapped using a single TLB entry
 - similarly for a memory mapped graphics buffer
- IA32 solution
 - first level PTE points to a 4MB page of physical memory [not a 2nd level page table]
 - bit set in primary PTE to indicate that it points to a *large* page [not a 2nd level page table]

MEMORY MANAGEMENT UNITS

IA32 Support for Large Pages



- corresponding TLB entry maps 4MB virtual page to a 4MB page of physical memory
- 4MB page aligned on a 4MB boundary in virtual and physical address spaces
- TLB operation needs to be modified to accommodate these *large 4MB* TLB entries

Breakpoints Registers

- the MMU typically supports a number of *breakpoint address registers* and *breakpoint control registers*
- the MMU can generate an interrupt if the breakpoint address [**virtual or physical**] is read or written [**watchpoint**] or executed [**breakpoint**]
- debugger normal sets breakpoints and watchpoints using virtual addresses
- used to implement real-time debugger breakpoints and watchpoints
- hardware support needed to set breakpoints in ROM and for watchpoints
- MMU breakpoint registers are part of the process state
 - save/restored as part of the context switch
 - hence more than one processes can be debugged *at the same time*
- used by Linux ptrace system call