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... $2^{20} \times 2^{12} [4K]$ 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

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

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

for (size_t i = 0; i < 4*GB; i += PAGESIZE, p += PAGESIZE)
    *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 PAGESIZE 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 \([2^{43}]\)
  - `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 Cruncher...

```plaintext
memory cruncher 64bit 14:19:56 20-Oct-2012
RAM = 16GB
pageSz = 4096B
min app virtual address = 0x000000000010000
max app virtual address = 0x0000000F0000F000
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
```
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
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...
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

NB: page tables are page aligned

AS MANY AS NEEDED
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
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
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 \([\text{TLB miss} \equiv \text{page fault}]\)

• 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 \([\text{eg. could implement a hash table}]\)

• 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 \([\text{needed for parallel accesses to the instruction and data caches}]\)
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 "INVLPD 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
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]
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*

- as all addresses go through the MMU, it 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
Integrating MMU and Operating System

- page table entries normally have a number of bits set aside for use by the OS implementer [i.e. not altered by hardware]

- IA32 PTEs have 3 such bits

![Diagram of page table entry](image)

- use spare bits to store OS specific PTE types

- consider the OS specific PTE types used in a hypothetical Unix implementation [closely modelled on GENIX for the NS32000 microprocessor which was the first demand paged microprocessor Unix implementation]

- uses 2 spare bits in PTE to define four PTE types when V == 0 and four when V == 1
Types when $V == 1$ [VALID]

- **MEM** - maps virtual address to a physical address

- **LOCK** - same as MEM except page is locked into physical memory
  - $vlock(va)$ system call [superuser ONLY]
  - software, not hardware, locking
  - really a hint to OS

- **SPY** - maps virtual address [$va$] to a specific physical address [$pa$]
  - can be used to map hardware device registers into a user process' virtual address space
  - $vspy(va, pa)$ system call [superuser ONLY]
  - allows user level device drivers to be implemented
Types when V == 0 [INVALID]

- **NULL** - page NOT yet mapped to physical memory
- **DISK** - page not mapped to physical memory, but when mapped the page must be initialised using data stored on disk
  - when V == 0, the PTE physical page # field contains a disk block number where the data is located on disk
  - assuming a 20 bit physical page # field, a 4K page size and a 4K disk block size it is possible to accommodate a $2^{20} \times 2^{12} = 4$GB disk [limiting with current disk sizes]
- **IOP** - indicates that the disk I/O is in progress
- **SPT** shared PTE [explained in next section of notes]
  - allows code to be shared between processes
  - contains a pointer to a PTE in another page table
Initial Mapping of Unix/Windows Process

- need to create a virtual address space + page table for process
Initial Mapping of Unix/Windows Process...

- Unix/windows process
- PTB
- Primary page table
- Secondary page table
- Disk block
- Memory page
Initial Mapping of Unix/Windows Process...

- text and initialised data PTEs initialised to type DISK
  - disk block number allows data to be quickly located on disk
- enough real stack pages allocated [type MEM] to hold the arguments and environmental data passed to the process
- **ALL** remaining PTEs initialised to type NULL
- process allocated ONLY 5 pages of physical memory initially
  - primary page table
  - 2 secondary page tables
  - 2 stack pages
- further pages allocated to process on demand
Initial Execution of Unix/Windows Process

- after the initial page table is created the process starts execution [start address in .exe header]

- will instantly generate a page fault as page containing first instruction is still on disk

- page faults will continue to occur as the process executes and each PTE type fault will be handled as follows:

<table>
<thead>
<tr>
<th>PTE type</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISK</td>
<td>allocate a page of physical memory [OS maintains a free list] and fill with data read from disk [context switch while waiting for disk] code pages normally read ONLY, initialised data pages typically read/write code and initialised data paged in &quot;on demand&quot;</td>
</tr>
</tbody>
</table>

DISK → IOP → MEM
### Initial Execution of Unix/Windows Process...

<table>
<thead>
<tr>
<th>PTE type</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>physical memory has not yet been allocated</td>
</tr>
<tr>
<td></td>
<td>the virtual fault address is checked to see if it's sensible / in range</td>
</tr>
<tr>
<td></td>
<td>if page fault virtual address not in uninitialised data, heap or stack then it is considered to be an illegal memory access and a memory access violation is signalled otherwise a page of [zeroed] physical memory is allocated by OS</td>
</tr>
<tr>
<td>NULL → MEM</td>
<td></td>
</tr>
</tbody>
</table>

| MEM     | protection level fault [e.g. writing to text via a NULL pointer] if OS gets confused and cannot resolve page fault it calls panic() which reboots OS |
| IOP     | wait for I/O to complete [see DISK type fault] |
| SPY     | protection level fault? |
| LOCK    | protection level fault? |
Diagram shows page table after the following pages have been accessed by the process:

- 1 code page (1)
- 1 initialised data page (2)
- 1 uninitialised data page (3)
- 1 stack page (4)
Text/Code Sharing

- if the same process is executing more than once, ONLY a single shared copy of the code need be in memory

- NB: each process still needs its own pages for its data, heap and stack

- NB: initialised data can be shared if read-only

- consider the following technique for sharing pages

- when a process is executed for the first time, a master page table is created

- the PTEs corresponding to the code and initialised data are initialised to type DISK

- remaining PTEs set to type NULL
Text/Code Sharing..

- A process page table is created by initialising its code and initialised data PTEs to type SPT.
- The SPT PTEs point to their corresponding entries in the master page table.
- Physical pages for its initial stack are attached to the process page table.
- Remaining PTEs set to type NULL.
Text/Code Sharing..

- on a SPT page fault, the OS follows the SPT entry to the corresponding PTE in the master page table
- action performed depends on master page table PTE type
- \text{DISK}_{\text{CODE}}
  - allocate page of physical memory
  - fill with data read from disk
  - update PTEs in master and process page tables to point to allocated page \([\text{MEM}_{\text{CODE}}]\)
Text/Code Sharing..

- **DISK\_IDATA**
  - allocate page of physical memory (1)
  - fill with data read from disk
  - attach to master page table [MEM]
  - now have a read-only *master copy* of the initialised data page

  - allocate page of physical memory (2)
  - copy data from master copy
  - attach to process page table [MEM]
  - process now has its own copy of the initialised data page which it is free to overwrite

  - could implement *copy-on-write* instead of *copy-on-access*
Text/Code Sharing..

- diagram shows how another process page table is created from the master page table
- the MEM\text{code} entries are copied thus sharing the code
- the remaining PTEs for the code and initialised data are set to the SPT type and point to corresponding entry in the master page table
- the remaining PTEs are initialised as per the non-shared case since each instance needs its own its uninitialised data, heap and stack
Text/Code Sharing..

• if all processes terminate, the OS will try to keep the master table and its attached pages in memory

• if another instance of the process is then created, it can quickly attach to the code pages already in memory

• it can also make its own copies of the initialised data pages, as needed, from the master copies attached to the master page table

• this is why when a process runs for a second time, it often starts up more quickly
IA-32e address spaces > $2^{32}$ bytes [x64]

- pragmatic implementation [not currently realistic to implement $2^{64}$ virtual and physical address spaces – just think of the cost of $2^{64}$ bytes RAM]
IA-32e address spaces > $2^{32}$ bytes...

- $2^{48}$ byte virtual [linear in Intel terminology] and $2^{52}$ byte physical address spaces

- 4 level page table structure 9-9-9-9-12 [Intel naming: PML4, Directory Ptr, Directory, Table]

- page table sizes $2^9 \times 8$ as each PTE is 64 bits [4K]

- PTE comprises 40 bit physical page # + 12 house keeping bits [52 bits]

- how many bits of the 52 bit physical address actually used depends on CPU model [$2^{40} = 1$TB, $2^{42} = 4$TB, $2^{50} = 1$PB and $2^{52} = 4$PB]
Summary

• you are now able to:
  ▪ explain the concept and benefits of virtual memory
  ▪ explain the operation of an n-level page table
  ▪ construct the contents of an n-level page table
  ▪ explain the operation of a TLB
  ▪ calculate the TLB hit rate
  ▪ explain how a MMU and an OS together support on-demand paging
  ▪ explain how code and initialised data can be shared between processes