Memory Management Units

A MMU converts a CPU’s virtual address into a physical address in order to access memory.

* The **virtual page number** is converted to a **physical page number**.
* The **offset** is unchanged.

32 12 0

|  |  |
| --- | --- |
| Virtual Page Number | Offset |

32 ⬇ 12 0

|  |  |
| --- | --- |
| Physical Page Number | Offset |

The address space is divided into fixed sized **pages** of contiguous memory. (Usually **4KB**)

Each CPU core will have separate MMUs for instruction and data accesses.

Virtual and physical address spaces need **not** be equal.

### 32-bit Processes

Each process runs in its own 4GB (232B) **virtual address space**.

Each page in the virtual address space is mapped on demand by the **OS** to a real physical **page frame** in memory.

Pages can be:

1. Unallocated
2. Allocated in memory
3. Allocated on the paging disk (secondary storage)

### Page Tables

A **page table** stores **page table entries**. These PTEs store the physical location of the corresponding page frame in memory.

Every process has its own page table, whose address is stored in the **page table base register** of the process. (PTB0 for OS, PTB1 for current user process)

### Page Table Entries

A 32-bit page table entry may look like the following 20-12 split:

32 12 0

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Physical Page Number | | | | | | | | | | | | Unassigned | | | | R | M | PL | V |

Since we only need 20 bits to tell us the correct index of the page frame in memory, we have 10 bits remaining that we can use to store extra data about the referenced page.

* V Valid
* PL Protection Level
* M Modified
* R Referenced

The **unassigned** portion of the PTE is left to the OS for its own use.

### Flat Page Tables

If our 32-bit address supports 220 pages of size 212B, then each page table needs to contain 220 32-bit PTEs (4MB). This is a waste of memory, since there are enormous contiguous blocks of empty PTEs in the page table.

### N-Level Page Tables

N-level page tables partition the page table into blocks of PTEs. These blocks are called **secondary page tables**.

N-level page tables occupy less space than flat page tables since they only allocate secondary page tables that are actually being **used** by the process. The page table **expands** in size as the process requests more memory from the OS and more secondary page tables are allocated.

The 20-bit section of the virtual address is split up into N **segments**. Each segment acts as an index to a unique **dimension** of the N-level page table.

If we have a 2-level page table of virtual address segments 10-10-12, then we will have a primary page table of size 210 \* 4B (4KB). Secondary page tables will only be created if the process requests more memory.

Each **secondary** page table is 4KB in size also since our split is 10-10.

If our virtual address is split into segments N-M-O, then our page table will only consume

2N + P\*2M bytes where P is the number of secondary page tables allocated by the OS.

### MMU Operation

When the MMU accesses an index in a page table, the valid bit is checked. If v = 0 then no physical memory has been allocated for this PTE.

A **page fault** occurs and the instruction is aborted. The MMU interrupts the CPU.

The OS handles a page fault by:

1. Allocating a page frame.
2. Updating the associated PTEs.
3. Loading data from disk into the new page frame. (Switches context while waiting)
4. Signalling an access violation.
5. Restarting / continuing the faulting instruction.

### Process’ Page Table Structure

**Code** and **data** pages are placed at the start of the process’ page table.

The **stack** is placed at the end so that it can expand downwards in memory.

### Translation Look-Aside Buffer

A TLB is an on-chip fully-associative **cache** that provides direct mappings for the N most-recently-accessed virtual pages. (Typically 64 entries >90% hit rate)

Without a TLB, virtual to physical address translation requires 1 memory access for **each level** of the page table.

When a virtual address is being translated, the TLB is searches for the virtual page number in **parallel** for “instant” translation.

If a match is **not** found, an interrupt is generated and the page tables are walked by the CPU/MMU. The least-recently-used entry of the TLB is replaced with the new mapping.

Since processes use the same virtual addresses, all TLB entries must be **invalidated** on a context switch. PTB1 also changes.

If a PTE is changed in memory, the OS must update the TLB.

Processes may **share** the TLB if the process ID is appended to the virtual page number.

### PTE Bits

CPU/MMU automatically sets the ‘referenced’ and ‘modified’ bits in the PTEs, but it is the OS’s responsibility to clear them.

The OS can use these bits to determine good candidates for being paged out / written to the paging disk.

The unassigned bits can be used by the OS to define extra PTE types:

If **v = 1**

1. MEM Maps virtual addresses to physical addresses.
2. LOCK MEM but page is locked into physical memory.
3. SPY Maps the virtual address to a specific physical address.  
    - Can be used to map hardware device registers into a user’s process’ virtual  
    space.  
    - Allows user-level device drivers to be implemented.

If **v = 0**

1. NULL Page is not yet mapped to physical memory.
2. DISK Page is not yet mapped, but must be initialised using data stored on disk.
3. IOP Disk I/O in progress.
4. SPT Shared PTE.  
    - Allows code to be shared between processes.  
    - Contains a pointer to a PTE in another page table.

### Different Page Sizes

MMUs can support different page sizes. Large pages allow a single TLB entry to map a large virtual page onto a similar area of contiguous physical memory.

IA32 uses the first level PTE to point to a 4MB page of physical memory instead of a secondary page table. A bit is used to differentiate between large and normal pages.

### Breakpoints

The MMU can generate an interrupt if the breakpoint address (virtual or physical) is read or written (watchpoint) to or executed (breakpoint).

A debugger normally sets breakpoints / watchpoints using virtual addresses.

Hardware support is needed to set breakpoints / watchpoints in ROM.

MMU breakpoint registers are part of the process state, and are saved / restored as part of a context switch.

### Initial Mapping of Unix & Windows Processes

Text and initialised data PTEs are initialised to type DISK.

Enough real stack pages are allocated with type MEM to hold the arguments and environmental data passed to the process.

All remaining PTEs are set to type NULL.

Processes are initially allocated 5 page frames:

* 1 primary page table
* 2 secondary page tables
* 2 stack pages

When a process begins execution, a page fault will be generated as the first instruction is still on disk.

### Handling of PTE Faults

|  |  |
| --- | --- |
| **Type** | **Resolution** |
| DISK | Allocate a page of physical memory and fill with data from disk.  Code pages are READ ONLY.  Code and initialized data paged in “on demand”.  **DISK → IOP → MEM** |
| NULL | Allocate zeroed memory to the process.  **NULL → MEM** |
| MEM | Protection level fault. |
| IOP | Wait for I/O to complete. |
| SPY | Protection level fault. |
| LOCK | Protection level fault. |

Text / Code Sharing

If the same process is executing more than once, only one shared copy of the **code** needs to be in memory.

A **master page table** is created when a process is first executed.

The PTEs corresponding to the code and initialised data are initialised to type DISK, all others NULL.

A new process’ page table is created by initialising its code and initialised data PTEs to type SPT which point to entries in the master page table.

Physical pages for the process’ stack are attached to the process’ page table, others are NULL.

### SPT Page Faults

On a **shared page table** page fault, the OS follows the SPT entry to the corresponding PTE in the master page table.

The action performed depends on the PTE type in the master page table:

|  |  |
| --- | --- |
| **Type** | **Resolution** |
| DISKCODE | Allocate a page frame and fill it with data from disk.  Update PTEs in master and process page tables for allocated page.  DISKCODE → MEMCODE |
| DISKDATA | Allocate a page frame and fill it with data from disk.  Attach to master page table. This is now the master copy of the initialised data page.  Copy data from the master copy, and attach to process’ page table. |

If a process is terminated, the OS will try to keep the master page table and its attached pages in memory. New instances of the process will start up more quickly than first time.