Contents

Memory Management Units

- MMU simply converts *virtual* address generated by a CPU into a *physical* address which is applied to the memory system

← 20 →	← 12 →
virtual page #	offset
⇓	⇓
mapped by MMU	unchanged
⇓	⇓
physical page #	offset

- Address space divided into fixed fixed pages (e.g. $2^{12} = 4096 = 4$KB)
- Low order address bits (offset within a page) not effected by MMU operation
- Virtual page number converted into a physical page number
- MMUs integrated on-chip with the CPU
- Each CPU core will typically have seperate MMUs for instruction and data accesses
- Examples are per IA32
  - 2^{32} byte (4GByte) address space divided into 2^{20} (1MB virtual pages) $\times$ 2^{12} (4KB offset)
  - $2^{20} = 1{,}048{,}576$ pages
  - $2^{12} = 4096$ possible bytes to index
- Virtual and phsyical address spaces need **NOT** be the same size

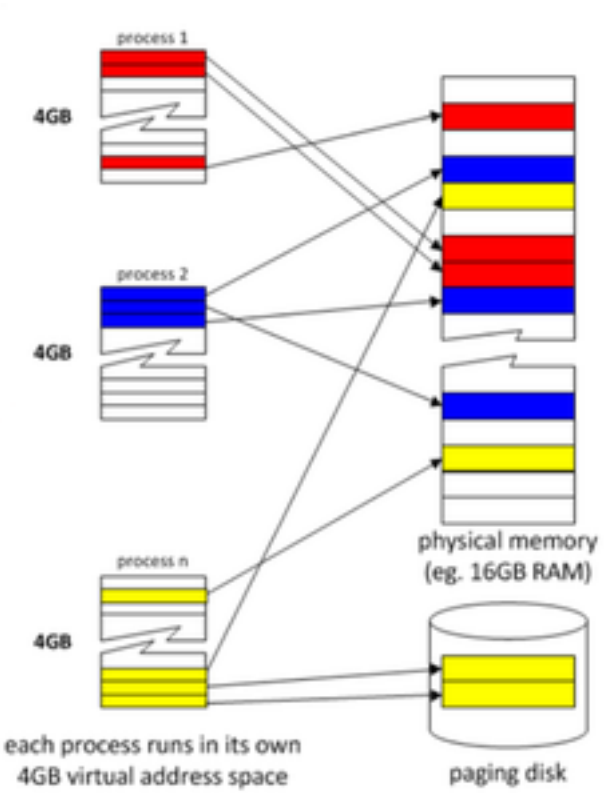

Figure 1: Virtual onto Physical

Mapping

Virtual Address Spaces onto Physical Memory

- Each process runs its 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
- Virtual pages (in a process) may be
  - Not allocated (probably because the process hasn't accessed 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
- Atlas Computer 1962 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 this 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*2014)

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
}
```

- Windows PAGESIZE is 4K
- What is the largest contiguous memory block that can be allocated?
- Windows 7 Win 32
  - 4GB virtual address space, bottom 2GB for user and top 2GB for OS
  - Can `malloc` 2047MB contiguous memory block
- Windows 7 x64
  - Program reports it can allocate a contiguous memory block of 8191GB or 8TB
  - `malloc`ing a block much greater than size of physcal memory (16GB) results in PC becoming extremely unresponsive
  - RUN with caution

Generic Memory Management Unit Operation (IA32, x64, MIPS, ...)

- Virtual page number converted to a physical page number by page table look-up
- Virtual page number used as an index into a page table stored in *physical memory*
- Page table *per* process (and sometimes per 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 (2^{20}=1MB (virtual pages) $\times$ 4 (32 bits = 4 bytes)) needed for page table of every process

N-level Page Table

- In order to reduce the size of the page table structure that needs to be allocated to a process, an *n-level* look-up table is used
- An n-level page table means that the *"larger"* the process (in terms of its use of the virtual address space), the more memory is needed for its page tables

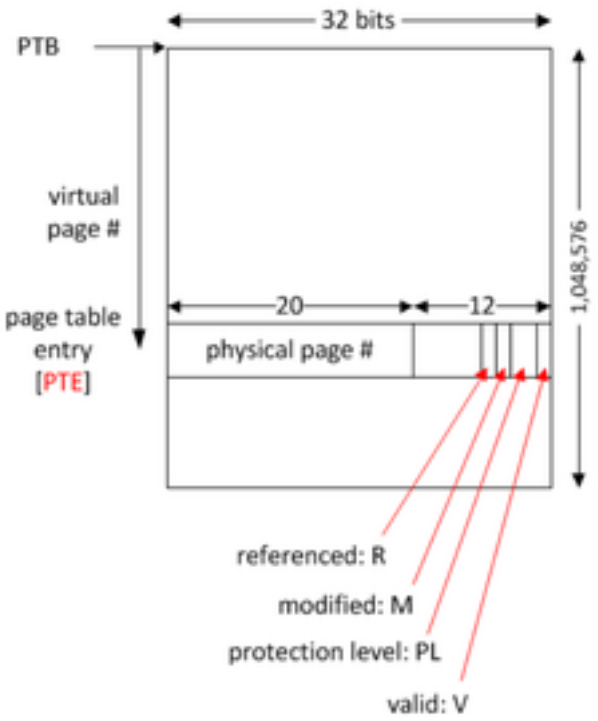

Figure 2: MMU Operation

- Consider a 2-level scheme

← 10 →	← 10 →	← 12 →
index 1	index 2	offset

- `index 1` is used to index into a primary page table, `index 2` into a secondary page table, and so on…
- PTB points to primary page table
- A valid primary page table entry points to a secondary page table
- Each process has one primary page table + multiple secondary page tables
- A secondary page tables created on demand (depends on how much of its vrtual address space the process uses)
- **NB:** size of page tables is 4KB - the page size itself

Valid Bit

- When MMU accesses a page table entry it checks the **V**alid bit
  - **if** `V==0` and accessing a primary page table entry
    - **then** NO physical memory allocated for corresponding secondary page table
  - **if** `V==0` and accessing a secondary page table entry
    - **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
  - Allocating a page of physical memory for the refeenced page
  - Updating the associated page table entry/entries
  - Reading code or initialised data from disk to initalise the page contents (context switches to another process while waiting)
  - Signalling an access violation (e.g. writing to a read-only code page)
  - Restarting (or continuing) the faulting instruction

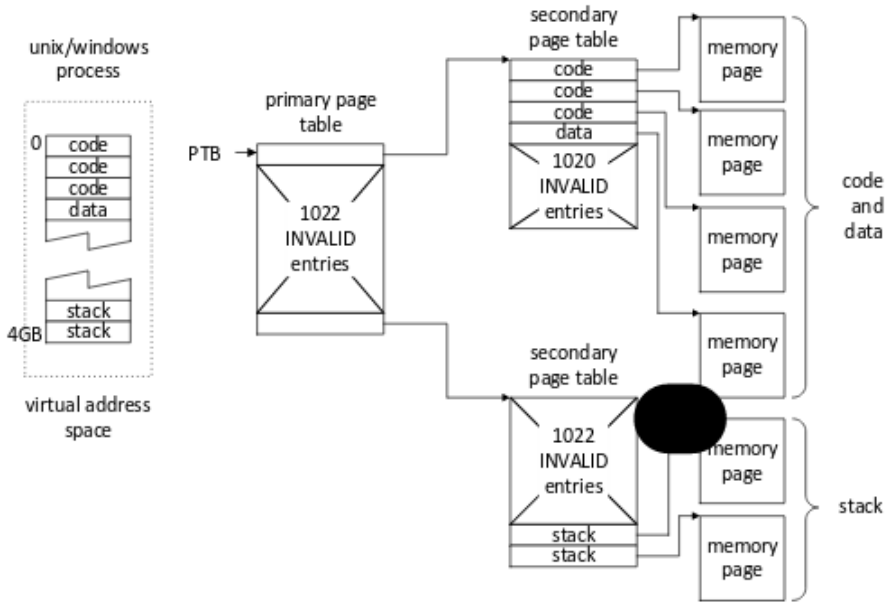

Figure 3: Example Process

Process Page Table Structure

- Example 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 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 × 4K pages = 4MB
- Need ONLY 2 secondary page tables providing program doesn't use moe than 4MB of code/data and 4MB of stack space

Translation Look Aside Buffer

- 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
- When a virtual address is sent to the TLB, the virtual page number is compared with ALL `m` tag entries in the TLD *in parallel* (a fully associative cache)
- If a match is found (TLB hit) the corresponding cached secondary page table entry is ouput by the TLB/MMU to provide the physical address
  - The address translation is completed *"insantanously"*
- If a match if 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 approach is taken by many RISCs, a TLB miss generates an interrupt and the CPU walks the page table using ordinary instructions
- 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 (e.g. could implement a hash table)
- Need a CPU instructions to replace the LRU TLB entry
- TLBs are normally small
- A typical 64 entry fully associative TLB has a hit rate of >90%
- A CPU would typically have a MMU for instruction access 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 contains current mappings of virtual to physical processes
- TLB is 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 really is shared

- **ALL** TLB entries referring to the old process must be invalidated on a context switch *otherwise* the new process will acces the memory pages of the old process
- Normally the OS (if it runs in its owen virtual address space) and *one* user process can share the entries in the TLB
- User/supervisor bit appended to TLB tag
- Whenever the page table base register (e.g. PTB0 for OS or PTB1 for 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 replected in the TLB
  - Must be able to invalidate old PTEs in the TLB
  - CPU instruction to do this (e.g. IA32 "INVAL va" will invalidate PTE entry corresponding to virtual address if va is present in TLB)
- Also need to keep TLBs in a multicore CPU coherent

Multiple Processes sharing TLB

- Possible for processes to share TLB is a process ID is appended to the virtual page number 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 limited (e.g. 8 bits)

Referenced and Modified Bits

- CPU/MMU automatically updates the PTE **R**eferenced and **M**odified bits (IA32/x64 **A**ccessed and **D**irty bits) in the PTEs
- PTE changes "written through" to correspondin PTE in physical memory
  - CPU/MMU automatically executes these bus cycles
- CPU/MMU never clears the reference and modified bits
  - Up to the OS (perhaps a background process regularly clearing the referenced bits?)
- OS can use the Referenced and Modified bits to determine
  - Which pages are good candidates for being paged out (ones that have not been referenced for a while)
  - Whether pages have to be written to the paging disk (may be unchanged since last write)

Support for Different Page Sizes

- Often useful is 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 2^{nd} level page table)
  - Bit set in primary PTE to indicate that it points to a *large* page (not a 2^{nd} 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*
- The MMU can generate an interrupt if the breakpoint address (virtual or physical) is read or written (watchpoint) or executed (breakpoint)
- Debugger normally sets breakpoints and watchpoints using virtual address
- Used to implement real-time debugger brakpoints 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 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
- 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 number* friend contains a disk block number where the data is located on disk
  - Assuming a 20 bit *physical page number* friend, a 4K page size and a 4K disk block size it is possible to accommodate a $2^{20} \times 2^{12} = 4$GB disk (limited with current disk sizes)
- *IOP* - indicates that the disk I/O is in progress
- *SPT* shared PTE
  - Allows code to be shared between processes
  - Contains a pointer to a PTE in another page table

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 intially
  - Primary page table
  - 2 secondary page tables
  - 2 stack pages
- Further pages allocated to process on demand
- After the initial page table is created the process starts execution (start address in .exe header)
- Will instantly generate a page fault as the first insutrction is still on disk
- Page faults will continue to occur as the process executes and each PTE type fault will be handled as follows:

DISK

Allocaye 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, intialised data pages typically read/write

Code and initialised data paged in "on demand"

DISK -> IOP -> MEM

NULL

Physical memory has not yet been allocated

The virtual fault address is checked to see if it's sensible/in range

If page fault virtual address not in initialised data, heap or stack *then* it is considered to be an illegal memory access and a memory access violation is signallted *otherwise* a page of (zeroed) physical memory is allocated by OS

NULL -> MEM

MEM

Potection level fault (e.g. writing to text via a NULL pointer)

IOP

Wait for I/O to complete (see DISK type fault)

SPY

Protection level fault?

LOCK

Protection level fault?

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
- When a process is executed *for the first time*, a master page table is created
  - Never used by a process, just used as a master copy
- The PTEs corresponding to the code and initialised data are initialised to type DISK
- Remaining PTEs set to type NULL
- A process page table is created by initialising its code and initialised data PTEs to type SPT
- On a SPT page fault, the OS follows the sPT entry to the corresponding PTE in the master page table
- Action performed dependson master page table PTE type
- $DISK_{\text{CODE}}$
  - Allocate page of physical memory
  - Fill with data read from disk
  - Update PTEs in master and process page tables to point to allocated page
- $DISK_{\text{IDATA}}$
  - Allocate page of physical memory
  - Fill it with data read from disk
  - Attach to master page table
  - Now we have a read-only *master copy* of the initialised data page
  - Allocate page of physical memory

  - Copy data from master
  - Attach to process page table
  - Process now has its own copy of the initialised data page which it is free to over write
  - Could implement *copy-on-write* instead of *copy-on-access*
    * Makes very little difference

- If all processes terminate, the OS will try to keep the master page 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 page master copies attached to the master page table
- This is why a process run, for a second time, often starts up more quickly

Memory Management inside x64 Processes

- Pragmatic implementation (not currently realistic to implement 2^{64} virtual and physical address spaces, just think of the cost of 2^{64} bytes RAM)
  - Virtual Address = Linear Address in Intel terminology
- 2^{48} byte birtual and 2^{52} byte physical address spaces
- 4 level page table structure 9-9-9-9-12
- Page table sizes 2^9 * 8 each PTE is 64 bits
- PTE comprises 40 bit physical address + 12 house keeping bits
- How many bits of the 52 bit physical address actually used depends on the CPU model