Virtual Memory and Address Translation

Review: the Program and the Process VAS

- Process text segment is initialized directly from program text section.
- Process data segment(s) are initialized from idata and wdata sections.
- Text and data segments may be write-protected.

- Process stack and BSS (e.g., heap) segment(s) are zero-filled.
- Process BSS segment may be expanded at runtime with a system call (e.g., Unix sbrk) called by the heap manager routines.

- Text and idata segments may be write-protected.

What’s in an Object File or Executable?

- Header “magic number” indicates type of image.
- Section table an array of (offset, length, startVA) program sections.
- Sections used by linker; may be removed after final link step.

Role of MMU Hardware and OS

- VM address translation must be very cheap (on average).
  - Every instruction includes one or two memory references (including the reference to the instruction itself).
- VM translation is supported in hardware by a Memory Management Unit or MMU.
  - The addressing model is defined by the CPU architecture.
  - The MMU itself is an integral part of the CPU.

- The role of the OS is to install the virtual-physical mapping and intervene if the MMU reports that it cannot complete the translation.

The OS Directs the MMU

- The OS controls the operation of the MMU to select:
  1. The subset of possible virtual addresses that are valid for each process (the process virtual address space);
  2. The physical translations for those virtual addresses;
  3. The modes of permissible access to those virtual addresses (read/write/execute);
  4. The specific set of translations in effect at any instant.

- MMU completes a reference only if the OS “says it’s OK”.
  - MMU raises an exception if the reference is “not OK”.

- Need rapid context switch from one address space to another.
The Translation Lookaside Buffer (TLB)

An on-chip hardware translation buffer (TB or TLB) caches recently used virtual-physical translations (ptes).

Alpha 21164: 48-entry fully associative TLB.

A CPU pipeline stage probes the TLB to complete over 99% of address translations in a single cycle.

Like other memory system caches, replacement of TLB entries is simple and controlled by hardware.

e.g., Not Last Used

If a translation misses in the TLB, the entry must be fetched by accessing the page table(s) in memory.

cost: 10-500 cycles

Care and Feeding of TLBs

The OS kernel carries out its memory management functions by issuing privileged operations on the MMU.

Choice 1: OS maintains page tables examined by the MMU.

• MMU loads TLB autonomously on each TLB miss
• page table format is defined by the architecture
• OS loads page table bases and lengths into privileged memory management registers on each context switch.

Choice 2: OS controls the TLB directly.

• MMU raises exception if the needed pte is not in the TLB.
• Exception handler loads the missing pte by reading data structures in memory (software-loaded TLB).

A Simple Page Table

Each process/VAS has its own page table. Virtual addresses are translated relative to the current page table.

In this example, each VPN / maps toPFN /, but in practice any physical frame may be used for any virtual page.

The page tables are themselves stored in memory; a protected register holds a pointer to the current page table.

Page Tables (2)

32-bit address with two-level page tables

(from Tanenbaum)

Alpha Page Tables (Forward Mapped)

This is (roughly) what a MIPS/Nachos page table entry (pte) looks like.

A Page Table Entry (PTE)
Virtual Address Translation

Example: typical 32-bit architecture with 8KB pages.

Virtual address translation maps a virtual page number (VPN) to a physical page frame number (PFN); the rest is easy.

Deliver exception to OS if translation is not valid and accessible in requested mode.

Physical address =PFN + offset

VPN

PFN

offset

Virtual address

physical address

What You Should Know

• Basics of paged memory management
• Typical address space layout
• Basics of address translation
• Architectural mechanisms to support paged memory
• Importance for kernel protection and process isolation
• Why the simple page table is inadequate
• Motivation for and structure of hierarchical tables
• Motivation for and structure of hashed (inverted) tables

Background

The remaining slides provide background from CPS 110.

Be sure you understand why page-based memory allocation is more memory-efficient than the old way: allocating contiguous physical memory for each address space (partitioning).

• Two partitioning strategies: fixed and variable
• How to make partitioning transparent to programs
• How to protect memory in a partitioned system
• Fragmentation: internal and external
• Fragmentation issues for each strategy
• Relevance to heap managers today
• Approaches to variable partitioning: First fit, best fit, etc., and the role of compaction.

Memory Management 101

Once upon a time...memory was called "core", and programs ("jobs") were loaded and executed one by one.

• load image in contiguous physical memory
• start execution at a known physical location
• allocate space in high memory for stack and data
• address text and data using physical addresses
• prelink executables for known start address
• run to completion

Memory and Multiprogramming

One day, IBM decided to load multiple jobs in memory at once.

• improve utilization of that expensive CPU
• improve system throughput

Problem 1: how do programs address their memory space?
load-time relocation?

Problem 2: how does the OS protect memory from rogue programs?
???
**Base and Bound Registers**

**Goal:** Isolate jobs from one another, and from their placement in the machine memory.

- Addresses are offsets from the job's base address stored in a machine base register.
  - Machine computes effective address on each reference initialized by OS when job is loaded.
- Machine checks each offset against job size placed by OS in a bound register.

**Base and Bound: Pros and Cons**

**Pro:**
- Each job is physically contiguous.
- Simple hardware and software.
- No need for load-time relocation of linked addresses.
- OS may swap or move jobs as it sees fit.

**Con:**
- Memory allocation is a royal pain.
- Job size is limited by available memory.

**Variable Partitioning**

Variable partitioning is the strategy of parking differently sized cars along a street with no marked parking space dividers.

**Wasted space** from **external fragmentation**

**Fixed Partitioning**

Wasted space from **internal fragmentation**

**Completing a VM Reference**

If an unmapped page is referenced, the machine passes control to the kernel exception handler (page fault).

1. Handler initializes a page frame, updates pte, and restarts.
2. If a disk access is required, the OS may switch to another process after initiating the I/O.
3. Page faults are delivered at IPL 0, just like a system call trap.
4. Fault handler executes in context of faulted process, blocks on a semaphore or condition variable awaiting I/O completion.

**Demand Paging and Page Faults**

OS may leave some virtual physical translations unspecified.

Mark the pte for a virtual page as invalid.

If an unmapped page is referenced, the machine passes control to the kernel exception handler (page fault).

Passes faulting virtual address and attempted access mode.

Handler initializes a page frame, updates pte, and restarts.

If a disk access is required, the OS may switch to another process after initiating the I/O.

Page faults are delivered at IPL 0, just like a system call trap.

Fault handler executes in context of faulted process, blocks on a semaphore or condition variable awaiting I/O completion.
Where Pages Come From

- Fetches for clean text or data are typically fill-from-file.
- Modified (dirty) pages are pushed to backing store (swap) on eviction.
- Paged-out pages are fetched from backing store when needed.

Faults

- Faults are similar to system calls in some respects:
  - Faults occur as a result of a process executing an instruction.
  - Fault handlers execute on the process kernel stack; the fault handler may block (sleep) in the kernel.
  - The completed fault handler may return to the faulted context.
- But faults are different from system calls in other respects:
  - Syscalls are deliberate, but faults are “accidents”.
  - System calls involve divide-by-zero, dereference invalid pointer, memory page fault
  - Not every execution of the faulting instruction results in a fault.
    - May depend on memory state or register contents.

Options for Handling a Fault (1)

1. Some faults are handled by “patching things up” and returning to the faulted context.
   - Example: the kernel may resolve an address fault (virtual memory fault) by installing a new virtual-physical translation.
   - The fault handler may adjust the saved PC to re-execute the faulting instruction after returning from the fault.

2. Some faults are handled by notifying the process that the fault occurred, so it may recover in its own way.
   - Fault handler munges the saved user context (PC, SP) to transfer control to a registered user-mode handler on return from the fault.
   - Example: Unix signals or Microsoft NT user-mode A synchronous Procedure Calls (APCs).

Options for Handling a Fault (2)

3. The kernel may handle unrecoverable faults by killing the user process.
   - Program fault with no registered usermode handler?
     - Destroy the process, release its resources, maybe write the memory image to a file, and find another ready process/thread to run.
     - In Unix this is the default action for many signals (e.g., SEGV).

4. How to handle faults generated by the kernel itself?
   - Kernel follows a bogus pointer?  Divides by zero?  Executes an instruction that is undefined or reserved to user mode?
     - These are generally fatal operating system errors resulting in a system crash, e.g., panic().

Issues for Paged Memory Management

The OS tries to minimize page fault costs incurred by all processes, balancing fairness, system throughput, etc.

1. *fetch policy*: When are pages brought into memory?
   - Prepaging: reduce page faults by bringing pages in before needed
   - Clustering: reduce seeks on backing storage

2. *replacement policy*: How and when does the system select victim pages to be evicted/discarded from memory?

3. *backing storage policy*:
   - Where does the system store evicted pages?
   - When is the backing storage allocated?
   - When does the system write modified pages to backing store?