Virtual Memory and Address Translation

**Virtual Memory Concept**
- **Key problem:** How can one support programs that require more memory than is physically available?
- **Solution:**
  - Hide all physical aspects of memory from users
  - Memory is a logically unbounded virtual address space of $2^n$ bytes
  - Only portions of VAS are in physical memory at any one time

**Issues**
- **Placement strategies**
  - Where to place programs in physical memory
- **Replacement strategies**
  - What to do when there exist more processes than can fit in memory
- **Load control strategies**
  - Determining how many processes can be in memory at one time

---

**Realizing Virtual Memory**

**Paging**
- Physical memory partitioned into equal sized page frames
- A memory address is a pair $(f, o)$
  - $f$ — frame number ($F_{\max}$ frames)
  - $o$ — frame offset ($O_{\max}$ bytes/frames)
  - Physical address = $O_{\max} \times f + o$

**Physical Address Specifications**

- Example: A 16-bit address space with $(O_{\max} = 512$ byte page frames
- Addressing location $(3, 6) = 1,542$

PA: $1,542 = 1011110010$

$(0, 0)$

$(f, o)$

$F_{\max}$

$O_{\max}$

$2^n - 1$
Realizing Virtual Memory

A process's virtual address space is partitioned into equal-sized pages.

A virtual address is a pair \((p, o)\) where:
- \(p\) — page number (\(p_{\text{MAX}}\) pages)
- \(o\) — page offset (\(o_{\text{MAX}}\) bytes/pages)

Virtual address = \(o_{\text{MAX}} \times p + o\)

Paging

Mapping virtual addresses to physical addresses

Pages map to frames.
- Pages are contiguous in a VAS...
- But pages are arbitrarily located in physical memory, and
- Not all pages mapped at all times

Virtual Address Translation Details

Page table structure

1 table per process:
- Part of process's state

- Contents:
  - Flags — dirty bit, resident bit, clock/reference bit
  - Frame number

A page table maps virtual pages to physical frames.

CPU

Virtual Addresses

Program

Page Table

Virtual Address Space

Physical Addresses

Physical Memory

Page Table

Virtual Addresses

Physical Addresses
**Virtual Address Translation Details**

**Example**

A system with 16-bit addresses
- 32 KB of physical memory
- 1024 byte pages

---

**Virtual Address Translation**

**Performance Issues**

- Problem — VM reference requires 2 memory references!
  - One access to get the page table entry
  - One access to get the data
- Page table can be very large; a part of the page table can be on disk!
  - For a machine with 64-bit addresses and 1024 byte pages, what is the size of a page table?
- What to do?
  - Most computing problems are solved by some form of...
    - Caching
    - Indirection

---

**Virtual Address Translation Using TLBs to Speedup Address Translation**

- Cache recently accessed page-to-frame translations in a TLB
  - For TLB hit, physical page number obtained in 1 cycle
  - For TLB miss, translation is updated in TLB
  - Has better than 99% hit ratio! (why?)

---

**Dealing With Large Page Tables**

**Multi-level paging**

- Add additional levels of indirection to the page table by sub-dividing page number into k parts
  - Create a “tree” of page tables
Dealing With Large Page Tables

Multi-level paging

- Example: Two-level paging

Virtual Address Translation
Using Page Registers (aka Inverted Page Tables)

- Each frame is associated with a register containing
  - Residence bit: whether or not the frame is occupied
  - Occupier: page number of the page occupying frame
  - Protection bits

- Page registers: an example
  - Physical memory size: 16 MB
  - Page size: 4096 bytes
  - Number of frames: 4096
  - Space used for page registers (assuming 8 bytes/register): 32 Kbytes
  - Percentage overhead introduced by page registers: 0.2%
  - Size of virtual memory: irrelevant

Page Registers
Tradeoffs

- Advantages:
  - Size of translation table occupies a very small fraction of physical memory
  - Size of translation table is independent of VM size

- Disadvantages:
  - We have reverse of the information that we need...
  - How do we perform translation?
  - Search the translation table for the desired page number

Inverted Page Tables
Searching for a Virtual Page

- If the number of frames is small, the page registers can be placed in an associative memory

- Virtual page number looked up in associative memory
  - Hit: frame number is extracted
  - Miss: results in page fault

- Limitations:
  - Large associative memories are expensive
  - Memory expansion is non-trivial
Dealing With Large Inverted Page Tables

Using Hash Tables

- Use a proven fast search technique: Hash Tables
- Hash page numbers to find corresponding frame numbers in a "frame" table with one entry per page frame

![Hash Table Diagram]

Searching Inverted Page Tables

Using Hash Tables

- Page registers are placed in an array
- Page $i$ is placed in frame $f(i)$ where $f$ is an agreed-upon "hashing function"

- To lookup page $i$, perform the following:
  - Compute $f(i)$ and use it as an index into the table of page registers
  - Extract the corresponding page register
  - Check if the register contains $i$, if so, we have a hit
  - Otherwise, we have a miss

Searching the Inverted Page Table

Using Hash Tables (Cont’d.)

- Minor complication
  - Since the number of pages is usually larger than the number of slots in a hash table, two or more items may hash to the same location
- Two different entries that map to same location are said to collide
- Many standard techniques for dealing with collisions
  - Use a linked list of items that hash to a particular table entry
  - Rehash index until the key is found or an empty table entry is reached
  - …

Virtual Memory (Paging)

The bigger picture

- A process's VAS is its context
  - Contains its code, data, and stack
- Code pages are stored in a user's file on disk
  - Some are currently residing in memory; most are not
- Data and stack pages are also stored in a file
  - Although this file is typically not visible to users
  - File only exists while a program is executing
- OS determines which portions of a process's VAS are mapped in memory at any one time
Virtual Memory

Page fault handling

- References to non-mapped pages generate a page fault

Page fault handling steps:
- Service the fault
- Block the running process
- Read in the unmapped page
- Resume/initiate some other process
- Map the missing page into memory
- Restart the faulting process

Virtual Memory Performance

Page fault handling analysis

- To understand the overhead of paging, compute the effective memory access time (EAT)
  \[ EAT = \text{memory access time} \times \text{probability of a page hit} + \text{page fault service time} \times \text{probability of a page fault} \]

Example:
- Memory access time: 20 ms
- Disk access time: 25 ms
- Let \( p \) be the probability of a page fault
  \[ EAT = 20(1-p) + 25,000,000p \]

- To realize an \( EAT \) within 5% of minimum, what is the largest value of \( p \) we can tolerate?

Virtual Memory

Summary

- Physical and virtual memory partitioned into equal size units
- Size of VAS unrelated to size of physical memory
- Virtual pages are mapped to physical frames
- Simple placement strategy
- There is no external fragmentation
- Key to good performance is minimizing page faults