Virtual Memory and Address Translation
Program addresses are virtual addresses.
- Relative offset of program regions can not change during program execution. E.g., heap can not move further from code.
- Virtual addresses == physical address inconvenient.
  - Program location is compiled into the program.

A single offset register allows the OS to place a process’ virtual address space anywhere in physical memory.
- Virtual address space must be smaller than physical.
- Program is swapped out of old location and swapped into new.

Segmentation creates external fragmentation and requires large regions of contiguous physical memory.
- We look to fixed sized units, memory pages, to solve the problem.
Virtual Memory
Concept

Key problem: How can one support programs that require more memory than is physically available?
- How can we support programs that do not use all of their memory at once?

Hide physical size of memory from users
- Memory is a “large” virtual address space of $2^n$ bytes
- Only portions of VAS are in physical memory at any one time (increase memory utilization).

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
  - Page frames avoid external fragmentation.

A memory address is a pair \((f, o)\)

- \(f\) — frame number (\(f_{\text{max}}\) frames)
- \(o\) — frame offset (\(o_{\text{max}}\) bytes/frames)

Physical address = \(o_{\text{max}} \times f + o\)

PA: \[ \begin{array}{cccccccc}
\log_2 (f_{\text{max}} \times o_{\text{max}}) & \log_2 o_{\text{max}} & 1 \\
\hline & f & o \\
(0,0) & (f_{\text{MAX}}-1, o_{\text{MAX}}-1)
\end{array} \]
Physical Address Specifications
Frame/Offset pair v. An absolute index

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

![Diagram showing physical memory and addressing location](image)
Questions

- The offset is the same in a virtual address and a physical address.
  - A. True
  - B. False
Realizing Virtual Memory

Paging

- A process’s virtual address space is partitioned into equal sized *pages*.
  - \(|\text{page}| = |\text{page frame}|\)

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

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

\[2^{n-1} = (p_{\text{MAX}} - 1, o_{\text{MAX}} - 1)\]
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
Frames and pages

- Only mapping virtual pages that are in use does what?
  - A. Increases memory utilization.
  - B. Increases performance for user applications.
  - C. Allows an OS to run more programs concurrently.
  - D. Gives the OS freedom to move virtual pages in the virtual address space.

- Address translation and changing address mappings are
  - A. Frequent and frequent
  - B. Frequent and infrequent
  - C. Infrequent and frequent
  - D. Infrequent and infrequent
A page table maps virtual pages to physical frames.
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

Diagram:
- CPU
- Virtual Addresses
- Physical Addresses
- PTBR
- Page Table
Virtual Address Translation Details

Example

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

**Virtual Address Space**

**Physical Memory**

**Page Table**

- **CPU**
- **Physical Addresses**
- **Virtual Addresses**
- **Physical Addresses**

(4, 0)
(3, 1023)
(4, 1023)
(0, 0)
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 high 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
  - TLB still used, just not shown
  - The architecture determines the number of levels of page table
Dealing With Large Page Tables

Multi-level paging

- **Example:** Two-level paging

![Diagram of two-level paging](image)
The Problem of Large Address Spaces

- With large address spaces (64-bits) forward mapped page tables become cumbersome.
  - E.g. 5 levels of tables.

- Instead of making tables proportional to size of virtual address space, make them proportional to the size of physical address space.
  - Virtual address space is growing faster than physical.

- Use one entry for each physical page with a hash table
  - Translation table occupies a very small fraction of physical memory
  - Size of translation table is independent of VM size

- Page table has 1 entry per virtual page
- Hashed/Inverted page table has 1 entry per physical frame
Virtual Address Translation
Using Page Registers (aka Hashed/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/registry): 32 Kbytes
  - Percentage overhead introduced by page registers: 0.2%
  - Size of virtual memory: irrelevant
CPU generates virtual addresses, where is the physical page?
- Hash the virtual address
- Must deal with conflicts

TLB caches recent translations, so page lookup can take several steps
- Hash the address
- Check the tag of the entry
- Possibly rehash/traverse list of conflicting entries

TLB is limited in size
- Difficult to make large and accessible in a single cycle.
- They consume a lot of power (27% of on-chip for StrongARM)
Indexing Hashed Page Tables
Using Hash Tables

- Hash page numbers to find corresponding frame number
  - Page frame number is not explicitly stored (1 frame per entry)
  - Protection, dirty, used, resident bits also in entry

$\text{PTBR} \times h(PID, p) \rightarrow \text{Inverted Page Table}$

Virtual Address

CPU

Memory

Physical Addresses

$\text{CPU}$

$\text{Virtual Address}$

$\text{PID}$

$\text{running}$

$\text{PTBR} \times h(PID, p) \rightarrow \text{Inverted Page Table}$

$\text{PID}$

$\text{page}$

$011$

$\text{Inverted Page Table}$

$\text{tag check}$

$\text{PTBR} \times h(PID, p) \rightarrow \text{Inverted Page Table}$

$\text{Physical Addresses}$

$\text{CPU}$

$\text{Memory}$

$\text{PTBR} \times h(PID, p) \rightarrow \text{Inverted Page Table}$
Searching Hashed Page Tables

Using Hash Tables

- Page registers are placed in an array
- Page \( i \) is placed in slot \( f(i) \) where \( f \) is an agreed-upon hash 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 tag contains \( i \), if so, we have a hit
  - Otherwise, we have a miss
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 the 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 (open hashing)
Questions

Why use hashed/inverted page tables?

- A. Forward mapped page tables are too slow.
- B. Forward mapped page tables don’t scale to larger virtual address spaces.
- C. Inverted pages tables have a simpler lookup algorithm, so the hardware that implements them is simpler.
- D. Inverted page tables allow a virtual page to be anywhere in physical memory.
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:
- Processor runs the interrupt handler
- OS blocks the running process
- OS starts read of the unmapped page
- OS resumes/initiates some other process
- Read of page completes
- OS maps the missing page into memory
- OS 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: 60 ns
  - Disk access time: 25 ms
  - Let \(p\) = the probability of a page fault
  \[ EAT = 60(1-p) + 25,000,000p \]

- 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
Segmentation vs. Paging

- **Segmentation has what advantages over paging?**
  - A. Fine-grained protection.
  - B. Easier to manage transfer of segments to/from the disk.
  - C. Requires less hardware support.
  - D. No external fragmentation.

- **Paging has what advantages over segmentation?**
  - A. Fine-grained protection.
  - B. Easier to manage transfer of pages to/from the disk.
  - C. Requires less hardware support.
  - D. No external fragmentation.