CSE 323: Operating System Design

Virtual Memory (Paging and Swapping)

Salman Shamil

North South University (NSU)
Fall 2025

Original slides by Mathias Payer and Sanidhya Kashyap [EPFL]

Lecture Topics

- Abstraction: address space
- Mechanism: virtual address translation
- Mechanism: paging
- Mechanism: swapping

This slide deck covers chapters 18–22 in OSTEP.

Address Spaces

- The ***address space*** encapsulates all addressable memory
- A system has a certain amount of physical memory, this results in the available physical memory
- For simplicity, we assume that each address points to a byte
- Virtual and physical address space size can differ both ways
- The physically present memory is smaller or equal to the physical address space size

Example: modern 64-bit CPUs have a 48-bit virtual address space (with 64 bit pointers) and map to 48 bit of physical address space. Most machines have less than 256 TiB of memory.

Memory Abstraction

The seven layers of memory abstraction complexity:

- No virtualization, program runs on bare metal
- Time sharing, one program at a time
- Space sharing, relocate programs in one shared address space
- Space sharing, virtual address space through segment (base)
- Space sharing, virtual address space through segment (base+bounds)
- Space sharing, virtual address space through multiple segments
- Space sharing, virtual address space through paging

Match the Description

- One process uses all of memory
- Share address space, use a per-process offset
- Verify address is in usable part of the address space
- Several base+bounds pairs per process
- Rewrite code and data before running

Options:

base register, static relocation, segments, base & bounds, time sharing

Limits of Segmentation

Segmentation allows reasonable sharing of a physical address space among several processes. What are the drawbacks?

- Space: each segment must be fully backed by memory
- Space: physical memory area must be contiguous (fragmentation)
- ISA: instructions explicitly or implicitly encode target segment

Fragmentation

Fragmentation: *unused memory that cannot be used.*
External fragmentation: *visible to allocator (OS), e.g., between segments.*
Internal fragmentation: *visible to requester, e.g., rounding if segment size is a power of 2.*

For example: we have 16 KiB of memory. Process A has a code segment of 4 KiB starting at 2,048, a data segment of 4 KiB starting at 8,096 and a 1 KiB stack segment starting at 14,336. The system has 2+2+2+1 = 7 KiB of free memory, but the maximum contiguous space is 2 KiB. Starting process B that requires a code segment of 3 KiB would not be possible.

A More Complex MMU: Paging

- Goal: eliminate requirement that physical memory is contiguous
  - Eliminate external fragmentation
  - Grow (and shrink?) segments as needed
- Idea: break address spaces and physical memory into pages
  - Assign memory based on page granularity
  - Still prone to internal fragmentation (round to page size)

Page

- A page is the minimal unit to break up an address space
- For processes (virtual address space) this is called a (virtual) **page**
- As part of the physical address space this is called a **frame** (or physical page)
- A system with 12 bit pages implies that
  - the lowest 12 bit of the address specify the page offset
  - A page's size is 2^{12} bytes (4 KiB)

Paging

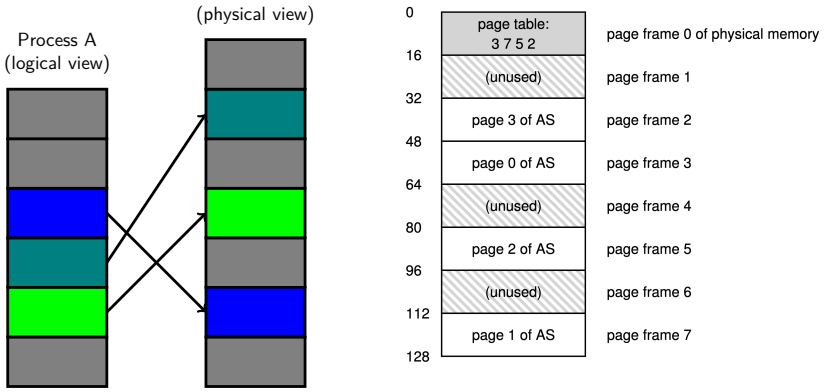
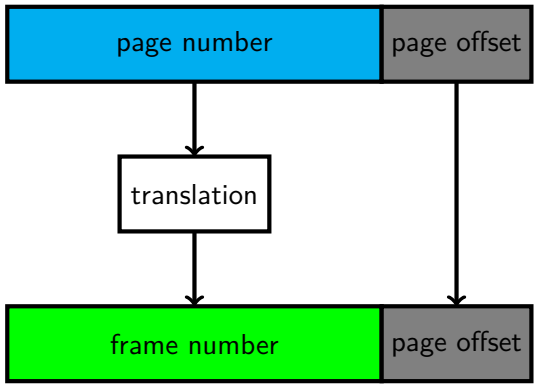


Figure 1: A toy example

Paging: Address Translation

- How can the MMU translate virtual to physical addresses?
  - High order bits designate page number
  - Low order bits designate offset in page
  - Note: size of virtual and physical address space may be different



Paging: Address Space Calculations

- Virtual address space: 4 GiB (i.e., 32 bits)
- Physical address space: 256 TiB (i.e., 48 bits)
- Page size: 4 KiB (i.e., 12 bits)

- Page numbers use 20 bits (i.e., each process may reference 2^{20} pages of 4 KiB=2^{12} size)
- Frame numbers use 36 bits (there is a maximum of 2^{36} physical 4 KiB frames in the system)

Insight: not all virtual or physical space must be allocated.

Paging: How to translate from virtual to physical?

Let's assume a 16 bit virtual, a 20 bit physical address space, and 12 bit for pages.

This means there are 16 pages per process and 256 physical pages.

The simplest approach to translate from virtual to physical addresses is through a **lookup table for each process**.

The pointer to the lookup table is stored in a special register (PTBR: Page Table Base Register)

```
unsigned char v1[16];
void *toPhys(void *ptr) {
  void *offset = ptr & (1<<12-1);
  // Page for process: Virtual bits & (1 << (16-12) - 1)
  unsigned int idx = (unsigned int)((ptr>>12) & (1<<4-1));
  // Get physical offset
  return (void*)(((unsigned int)(v1[idx])<<12) | offset);
}
```


Paging: Storing other per page information

Modern systems store much more information in the page table than just the physical frame number:

- Is the mapping valid?
- Protection bits (read, write, execute)
- Is the page present?
- Is the page referenced?
- Is the page dirty?

The OS and the MMU agree on the interpretation of these bits.

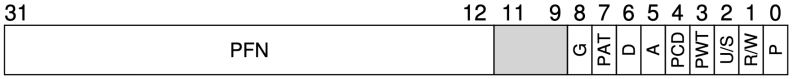
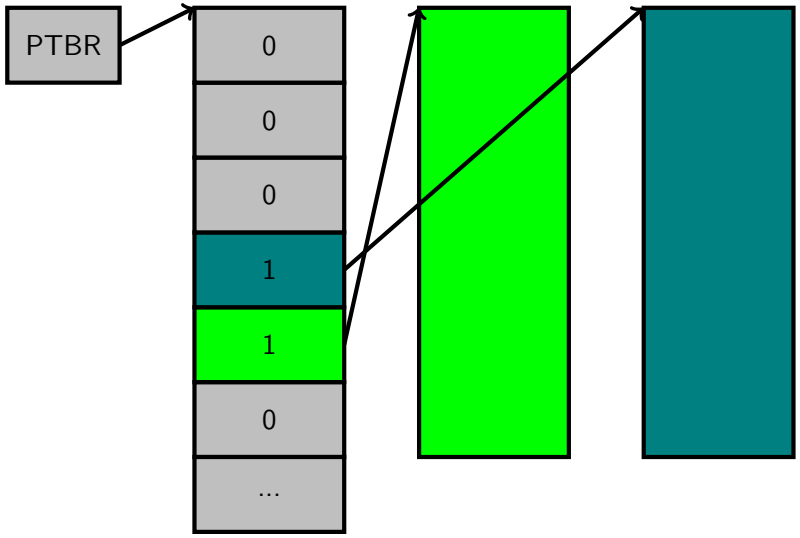


Figure 2: An x86 Page Table Entry (PTE)

Paging: example of a flat page table



Paging: Pros & Cons

- Advantages
  - No external fragmentation
  - Fast to allocate (no searching for space) and free (no coalescing)
- Disadvantages
  - Additional memory reference to page table (hint: use a cache)
  - Internal fragmentation (tension regarding page size)
  - Required space for page table may be substantial

Quiz: Find the page table size for a process with virtual address space = 4 GiB. Assume page size to be 4 KiB, where each PTE needs 4 bytes.

Paging: Avoid Additional Memory Access

- Simple idea: remember previous lookups
- Store the last accessed pages in a cache
- This cache is called **TLB** (Translation Lookaside Buffer)
- Reduces requirement memory accesses for page walk
- If there is an entry in the TLB, reuse!

Paging: The Page Size Tension

- Large pages $\rightarrow$ internal fragmentation (unused part)
- Small pages $\rightarrow$ too many pages. TLB miss more likely.
- Disk swapping issue (discussed later).

Paging: 32 bit address space

Assume 12 bit for pages, 32 bit virtual and 32 bit physical address space, 4 byte page table entries.

- What is the size of a flat page table?
- Page table size: entries $\times$ size of entry
- entries: $2^{32}/2^{12} = 2^{32-12} = 2^{20}$
- Page table size: $2^{20} * 4B = 4MiB$
- Let's assume a 32 bit virtual and 48 bit ***physical address space*** and 8 byte entries: 8 MiB
- Let's assume a ***64 bit virtual address space*** and 8 byte entries: $2^{52} * 8B$ (32 PiB). Not feasible! We need something better!

Paging: A Multi-Level Table

- Insight: most processes only need a fraction of the address space.
- Goal: only allocate metadata for this fraction
- Mechanism: a multi-level lookup table.
  - One or more levels of indirection allow space efficient encoding
  - Each level adds one more memory lookups during address translation
  - What is a good trade-off?

Paging: A Multi-Level Table

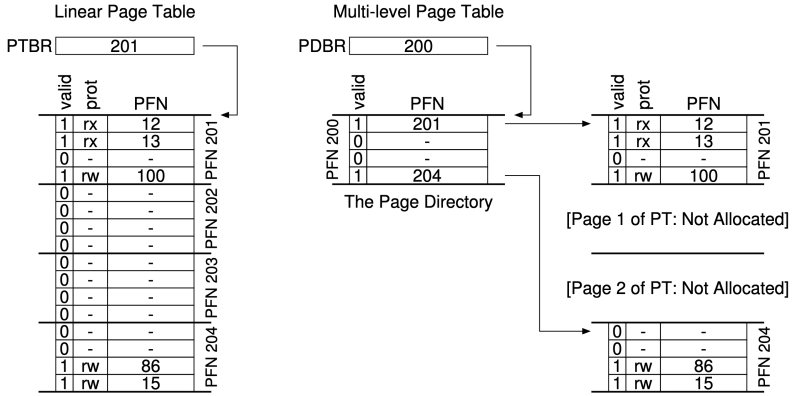


Figure 3: Linear/Flat *vs.* Multi-Level Page Tables

Paging: A Multi-Level Table

What size should the individual levels be?

1st level	2nd level	offset

Assumptions: 32-bit virtual address, 4 KiB pages, 4 B page table entries, two-level page table.

- **Page size & offset:** 4 KiB pages → 12-bit page offset
- **Entries per table:** 4 B per entry → 1024 entries per page → 10-bit index
- **Two-level page table:** 32-bit virtual address → 10 bits first-level, 10 bits second-level, 12 bits offset
- **Hex representation:** `0x3ff 0x3ff 0xfff` (max for each field)

Lookup flow: 1st-level index → select 2nd-level table → 2nd-level index → select page frame → offset → byte

Paging: A Multi-Level Table

- Let's move to a 64 bit virtual/physical address space, 4 KiB pages, PTE size 8 B
  - 4 KiB pages have space for 512 entries (9 bit)
  - $64 - 12 = 52$, i.e., we would need 6 levels of page tables
- Let's shrink the address space to 48 bit!
  - $48 - 12 = 36$, i.e., we need 4 levels of page tables
  - Page walks are still expensive, need high TLB hit rate for efficiency

Swapping: When Main Memory Runs Out

- Observation: main memory may not be enough for all memory of all processes
- Idea: store unused pages of address space on the disk
  - Allows the OS to reclaim memory when necessary
  - Allows the OS to over-provision
    - Hand out more memory than physically available
    - Process doesn't need to worry about limited memory
  - When needed, the OS finds and pushes unused pages to disk
  - Careful strategy needed to avoid performance degradation

Swapping: Page Fault

- The MMU translates virtual to physical addresses using the OS provided data structures (page tables)
- The present bit for each page table entry at each level indicates if the reference is valid, i.e., resides in memory, or not.
  - MMU checks present bit during translation
  - If a page is not present then the MMU triggers a page fault
  - The OS then enforces its policy to handle the page fault
- Virtual to physical translation is transparent to executed instructions, requires HW support

Swapping: Page Fault Handling

- MMU signals CPU to trap and switch to the OS
- If a page is on disk, the OS uses the PTE to locate it and issues a disk read to bring it into memory.
- Once the page is loaded, the OS updates the PTE to mark it present and store its in-memory frame number.
- The OS then retries the faulting instruction; this may cause a TLB miss, which is serviced to update the TLB.
- On the next attempt, the translation is in the TLB, and the CPU can access the desired memory location.

Swapping: Page Fault Handler

- Virtual to physical translation happens for every memory access
- Why does the MMU switch to the OS during a page fault?
- OS handles ***policy***, MMU is the ***mechanism***
- MMU handles common case: path to page is valid, page is present
  - Page walk can be highly optimized
  - OS and MMU agree on data structures
- OS handles policy decisions
  - Allows implementation of flexible policies
  - OS decides which pages are allocated and replaced
- Page fault handler need not be highly optimized; the real bottleneck is disk I/O.

Page Replacement Policy

- Determines which page to evict when memory is full
- Common policies:
  - **Optimal**: remove page not used for longest future time (theoretical)
  - **FIFO**: remove oldest page
  - **Random**: remove a random page
  - **LRU**: remove least recently used page
  - **Clock**: approximates LRU efficiently

Summary

- Fragmentation: space lost due to internal or external padding
- Paging: MMU fully translates between virtual and physical addresses
  - One flat page table (array)
  - Multi-level page table
  - Pros? Cons? What are size requirements?
- Paging and swapping allows process to execute with only the working set resident in memory, remaining pages can be stored on the disk