Operating Systems Notes

SubmergedDuck

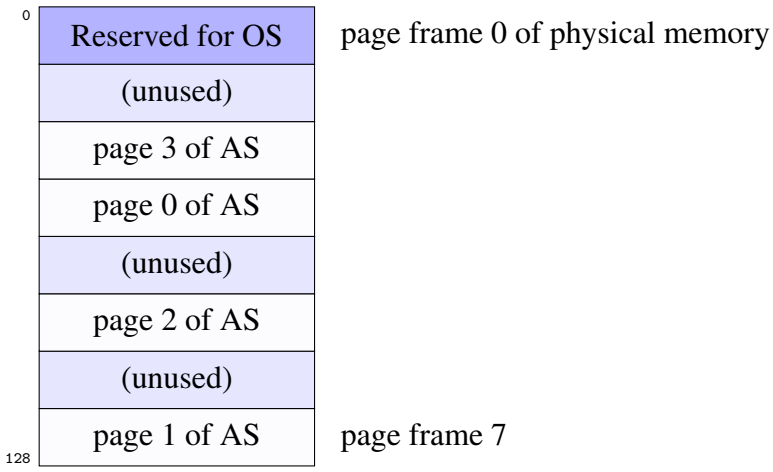
October 30, 2025

1 Page Tables: OSTEP

Paging. When memory is split into fixed-sized pieces. Inside one page, the offset tells "which byte within this page."

Page. A fixed size unit in a process's address space.

Fig 1. 64-Byte Address Space in 128-Byte Physical Memory. TBC.



Notice how each page here is $128 \div 8 = 16$ bytes each. Notice also how the address space is can be scattered throughout physical memory.

Free List. When an OS wishes to place a process's address space into physical memory, it keeps a free list of all free pages/page frames of the physical memory.

Page Table. A data structure to record each where each virtual page of the address space is located in physical memory. Each process has their own page table.

Address Translations. What the page table stores for each virtual page of the address space.

> **Eg.** Virtual Page 1 $\rightarrow$ Physical Frame 7
>
> **Eg.** To translate a virtual address to a physical address, we keep the offset (the right-most bits), but change the VPN to PFN. Using Fig 1, since the virtual page number is 1 (01 in binary) and the physical frame number is 7 (111 in binary) then our physical adress would be 111 | [offset] while our virtual adress would be 01 | [offset].

Offset. Indicates which byte a virtual address is on a virtual page.

Page Table Entry (PTE). A 20-bit VPN implies there are 2^{20} address translations for each process. We need 4 bytes per PTE to hold the physical translation plus other stuff. So for a 32-bit address space there are $2^{20} * 4$ bytes = 4MB of memory needed for each page table.

Note. So the number of page table entries is equal to the number of address translations. The number of address translations is related to how many bits a VPN has.

2 Page Tables: Other

Offset Bits. How many bits to count all bytes in one page.

Swap Offset. The index of the page-sized slot in the swap area (file/partion) where the whole page was written when it was swapped out.

> **Eg.** A 4KB page has 4096 bytes. Since $4096 = 2^{12}$, the page offset uses 12 bits.

Virtual Page Number (VPN). The index of a virtual page. These are the high bits of the virtual address.)

> **Eg.** Given a 32-bit virtual address and 4KB pages, there would be 20 bits left for the VPN (cause $4KB = 2^{12}$ bytes). This implies that there are 2^{20} virtual pages if there are 20 VPN bits.

Page Table Entry (PTE). One record in the page table that maps a virtual page to a physical frame and bits like valid/present, R/W/X permissions, dirty, accessed, etc. A single second-level page table holds 1024 PTEs.

> **Note.** Each page table entry has a Present(P) bit saying whether that virtual page is currently in RAM. $P = 1 \Rightarrow$ the mapping is valid and you can use the page table number. $P = 0 \Rightarrow$ otherwise (could be swapped out or not mapped.)

Virtual Page. A fixed-size chunk (eg. 4KB) of a process's virtual address space.

> **Eg.** With 4KB pages, every virtual address splits into a virtual page number (which page) + offset (which byte inside that page)
>
> **Eg 2.** Let virtual address = 0x12345ABC. Then the offset would be 0xABC (low 12 bits) and the VPN would be 0x12345 given 4KB pages.

Single-level Page Table. One big array indexed by the VPN; each slot is a PTE that says where that virtual page lives (which physical frame).

Second-level Page Table. Splits the VPN into PDI | PTI (10 | 10).

> **Eg.** There are 2^{20} page table entries, each entry is 4 bytes. The total size of the single page table would be $2^{20} \times 4$ bytes.

Physical Frame. A 4KB chunck of RAM (or same size as page) where a virtual page's data lives when it's in memory.

Physical Frame Number (PFN). The index of a 4KB physical frame in RAM. After page-table lookup, the PTE gives you the PFN plus status bits. Eg. Given 4KB pages, VA = [PFN | 12-bit Offset]

MiB. A mebibyte. 1 MiB = 2^{20} bytes

Page Directory Index (PDI). The first 10 bits of virtual address (VA). Selects which page table to use.

Page Table Index (PTI). The next 10 bits after PDI of VA. The Selects which PTE inside that page table.

Bit Shifting

```
#define PAGE_SHIFT 12 // 4KB Pages
uint64_t physical_address = ((uint64_t)pfn << PAGE_SHIFT)|
   offset;
```

Translation Lookaside Buffer (TLB). A tiny, very fast cache inside the MMU that stores recent virtual page number (VPN) to physical frame number translations (plus R/W/X/valid bits).

> **Hit.** MMU find the mapping in the TLB so it can form the physical address without reading the page tables from memory. Memory Accesses Needed: 1.
>
> **Miss.** MMU (or OS) must walk the page tables in memory to find the PTE, then it can access the data. Memory Accesses Needed: 3.

Memory Management Unit (MMU). The hardware between the CPU and RAM that translates virtual addresses to physical using TLB and page tables. Also enforces R/W/X permissions and raises page faults if needed and updates accessed/dirty bits.

Accessed Bit (A/R). Set by the hardware on any read or write to the page. OS uses it to tell which pages were recently used (eg. for replacement).

Dirty Bit (D/modified). Set on any write. If a dirty page is evicted, it must be written back to disk; if clean it can be dropped.

Dirty Page. A memory page that's been modified (written to) since it was loaded (its dirty bit is set.) If the OS evicts it, it must write it back to disk; a clean page (non-modified) can be dropped without writing.

Page Fault. The PTE says the page isn't in RAM (invalid/not present), thus trapping to the OS to bring the page in.

3 Lecture 15/16: Page Replacement Algorithms

Policy Decisions for Virtual Memory Management
There are three types of policies we need to know.

Fetch Policy. Policies that we use to decide when to fetch a page.

Placement Policy. Policies that we use to decide where to put the page.

Replacement Policy. Policies that we use to decide what page to evict to make room.

Page Fetch Policies. There are two types of fetch policies.

1. **Demand Paging.** This is a when we only fetch a page when we actually reference it.
2. **Prepaging.** This is when we fetch a page, we will fetch a little bit more, so when the next page fault comes around the data/content that we need for that page is already there. This is so we don't have the issue of "stalling twice."

Page Placement Policies. Modern memory managemnent systems allow for any physical frames to hold any virtual page. This means that on most computers, any of the physical frames are equally good to hold any virtual page. This also means that memory management hardware can translate any virtual-to-physical mapping equally well.

Why would we prefer some virtual-to-physical over others?

1. **NUMA Multiprocessors.** NUMA stands for non-uniform memory access. When we have a NUMA processor, some memory locations results in faster memory access than others. Eg. Any processor can access entire memory, but local memory is faster.
2. **Cache Performance.** You want to pick physical frames that does not result in bad behaviour. Thus we have to choose physical pages to minimize cache conflicts.

Note. Placement policy will have a much smaller effect than fetch and replacement polciies if memory is limited.

Page Replacement Policies. The main goal of a replacement algorithm is to reduce the **fault rate** (how frequent a page fault occurs). We want to evict pages that will never be used again in the future, but this is hard to do cause we don't know the future, so we will pick to evict the page that won't be used for the longest period of time.

How are replacement algorithms evaluated?

We take an input reference string (a list of addresses in the order that a program references them) and we count how many page faults occured, the lower the better.

What does it mean to reference an address?

It means to use a variable or pointer to locate and access data in memory.

Timestamp: 5:19

4 Lecture 12: Paging and Address Translation

Compaction. The idea that we have something called a physical memory (RAM) and as we are assigning physical memory to processes, we can end up with holes inbetween allocations of memory. The purpose of compaction is to eliminate the small fragments in physical memory by relocating allocated space by consolidating them so we get 1 big free space in physical memory that we can use for other processes.

> **Note.** Compaction is not possible with the C memory allocation library. The C library uses something called dynamic partitioning. It's not possible because once malloc() returns an address to you, that address would be out of reach by the malloc library—this means that the malloc library cannot change the address that was returned and update it to something else.
>
> **Dynamic Partitioning.** Opposite of fixed partitions. Video

Dynamic Relocation. Also known as execution-time binding of addresses. The purpose of this is to be able to do **swapping** and **compaction** at runtime. Dynamic relocation allows us to take a virtual address that the user process is using as its executing instructions and translating them to physical addresses.

> **Note.** This allows the OS to change the mapping between virtual and physical addresses whenever it wants—thus able to move data from one region of memory to another region of memory.
>
> **Regarding Compaction.** We are able to change some of the virtual-to-physical address translations.
>
> **What's the minimum requirements to relocate fixed or dynamic partitions?**
> All memory used by the process must be contiguous, which means no holes.

Relocation Registers. What some CPU architectures, commonly, the older ones have. The basic idea is to translate physical addresses by doing an add opertation to the virtual address to get the corressponding physical address.

> **Eg. Translating Addresses w/ Reloc. Reg.** Let's say base address is 10, and we want to translate the virtual address 5. Then the physical address you get is 10+5 = 15.
>
> **Note.** We also have to check that the base address is within the limits of a process's address space to protect overwriting memory that we lost to other processes. Sometimes the generated physical address does not even exist, eg. -200 physical address.
>
> **Base and Limit Reg.** On CPUs that have a "base" and "limit" registers, the MMU uses these registers to store the base address and also a limit. These registers are updated whenever we do a kernel level context switch.
>
> 1 **Base.** When we restore registers for the next process, we load the **base** register with the starting address of the physical address of that process.

2 **Limit.** Limit register is set to the last logical address of that process. Purpose is to cap what range of physical addresses are valid.

3 **Load/Store.** When we execute any memory reference instruction like load and store, the hardware will add the base address to the logical address (limit)

Kernel-Level Context Switch. When we switch from one process to another.

Logical Address. It's the same thing as a virtual address.

Timestamp: 8:22