Operating Systems

Memory Management

Sven Dziadek

October/November 2022

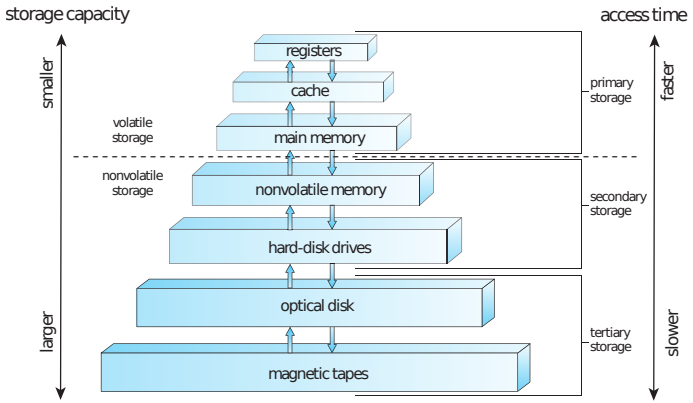
EPITA

Part I

Memory Management

Introduction

Recall: Storage



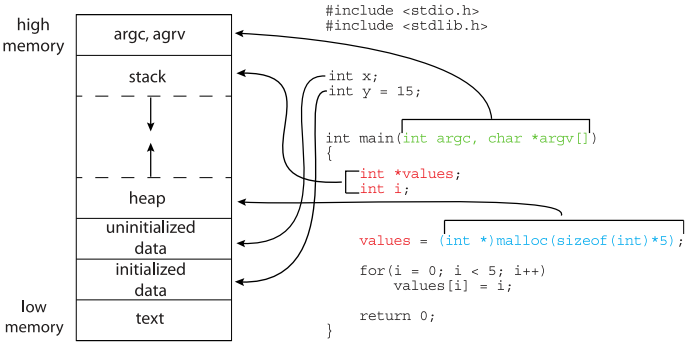
Primary storage

The only storage that the CPU can address directly

Motivation: Why do we need memory

- Central to the operation of modern computer systems
- Can be accessed directly by the CPU
- (Additional way to talk with devices, e.g., via MMIO or DMA)

Memory Layout of a Process



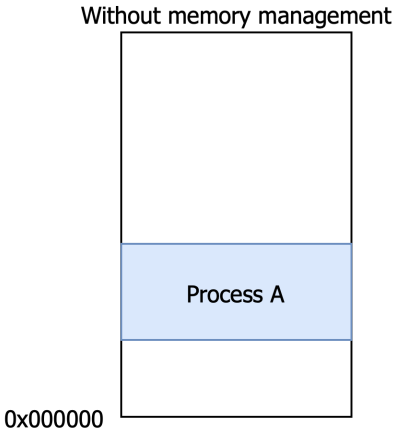
- Essentially: **code** & **data**
- Different access rights:
  x, ro, rw, ...
- Some data grows dynamically during runtime

Memory management - Some goals

- On multitasking system: more processes in memory simultaneously
- Handle hardcoded addresses in programs
- Minimize fragmentation
- Provide protection and isolation from other processes

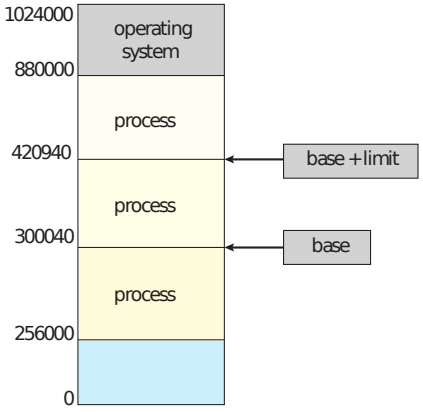
Without memory management

- Directly use physical addresses
- No memory protection
- You should not find that in any serious OS



Old Way: Segmentation

- Starting with the `i386` (1985) with the introduction of **Protected Mode**
- Allows the definition of **segments**
  - Zones of contiguous memory
  - Basic address translation
  - Basic memory protection
  - Privilege levels

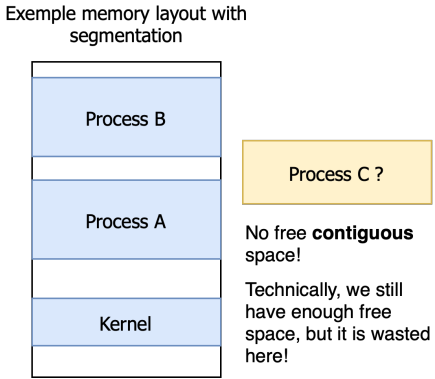


Segmentation - Issues

External Fragmentation

- Free space fragmented
- Nowhere big enough for new process

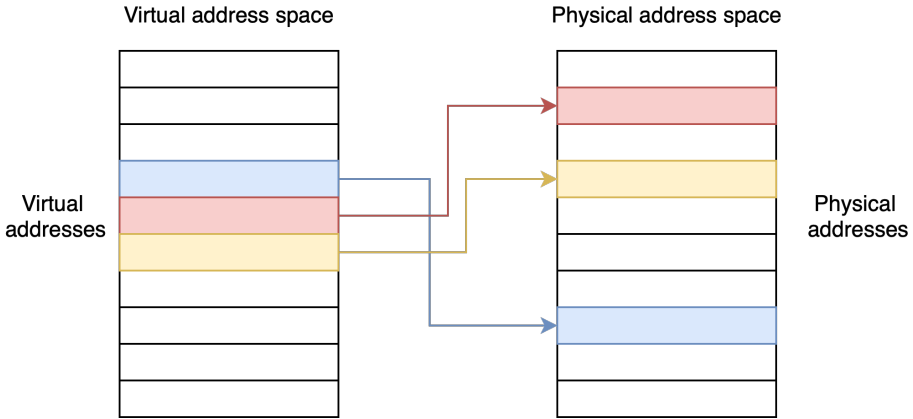
No shared memory between processes



Paging

Paging - Overview

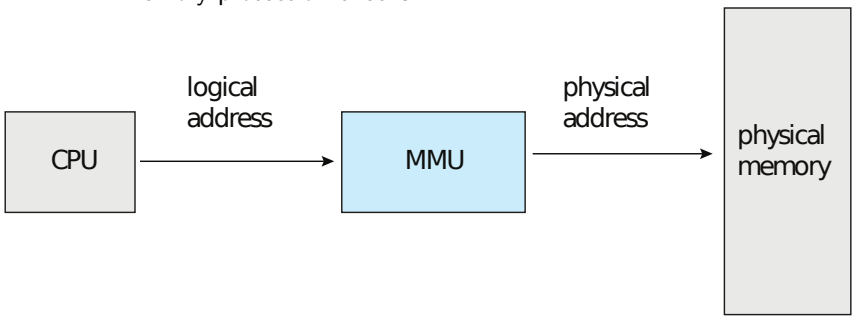
- Divide address space into `fixed-size` pages
- Divide physical memory into `fixed-size` (page) frames
- Process **sees** virtual address space



Memory Management Unit – MMU

All the address translations are done by the MMU.

- Integrated in the CPU since the `i286` (1982)
- It does:
  - Address translations when you are using segmentation or paging
  - Caching (`TLB`; more on that later)
  - Memory protection checks



Page Table

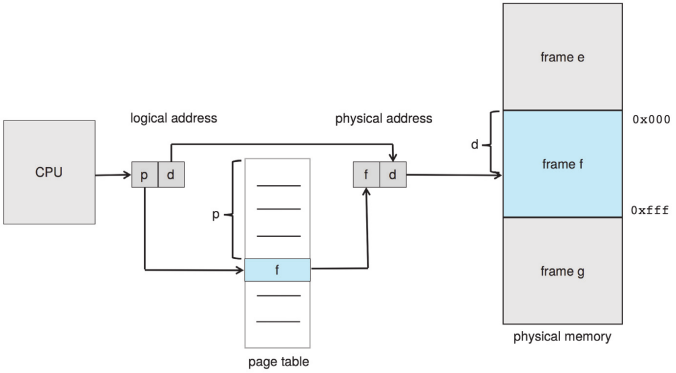
Page Table

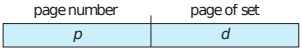
Data structure to store the mapping

- Paging depends on the CPU architecture (hardware)
- The kernel sets up and maintains them.
- Pages and frames are aligned on page sizes.

Paging – Example: 1-level

Split logical address in two parts:





Assume:

- logical address length m
- page/frame size 2^n

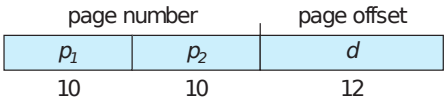
Then, we set:

Note that for **small** frames, the page table becomes **huge**.

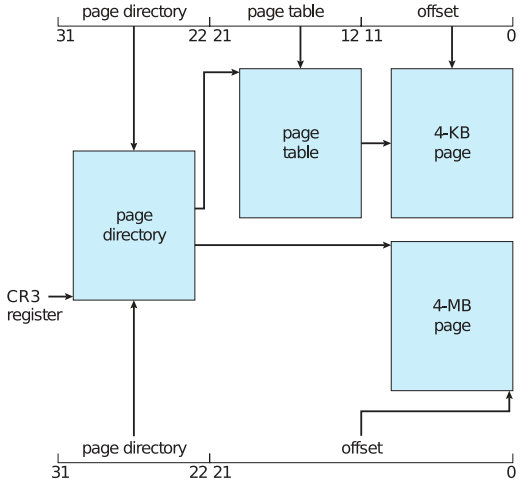
Paging – x86 (2-level)

Logical address: 32 bit = 4 B

Split in three parts:



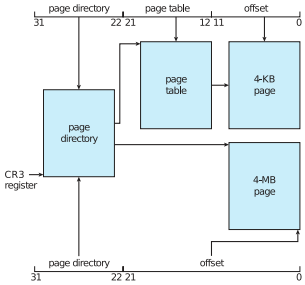
- Page size: 4 KiB (or 4 MiB) $= 2^{12}$ B (or 2^{22} B)
- **Page Table** & **Page Directory**: 4 KiB they store $2^{10} = 1024$ addresses
- Page directory stored in **CR3** register



Paging – Metadata

How to store metadata ?

- All structures are aligned on page size ($4\,\text{KiB} = 2^{12}\,\text{B}$)
- Use lower 12 bits of page directory/table entries to store (in)valid bit, access permissions, kernel only, etc.



Note:

Virtual address space can be larger than physical address space

$\Rightarrow$ many entries will be marked `invalid`

Paging – Page fault

Page fault

Software interrupt for illegal memory access

- Tells the kernel what caused the issue
- Register `CR2` holds the address of the failed memory access
- Page fault in user mode $\Rightarrow$ kernel signals `SIGSEGV` to user process

Paging – Kernel Space

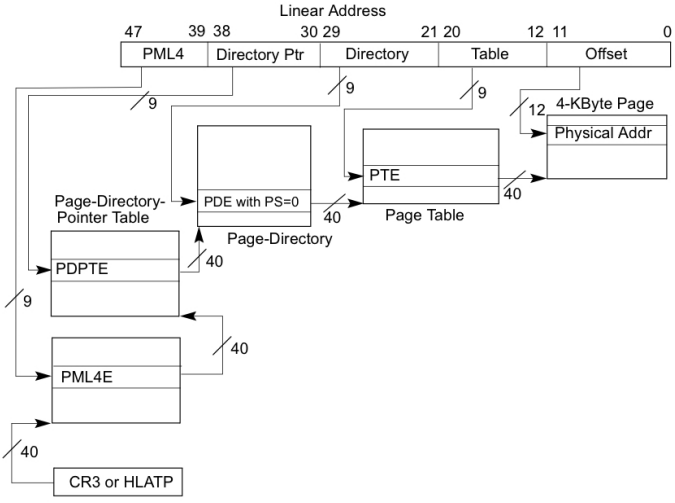
Kernel memory

Still mapped in user mode

- User applications have no access right
- Switching between kernel and user mode needs kernel code
  $\Rightarrow$ no time to unmap it

Paging – x86-64 (4-level)

- Uses 48 bit addresses
- 4-level paging scheme



Paging – Issues

Speed

Accessing memory needs multiple memory accesses:

- In x86 $\Rightarrow$ 3 accesses (2-level paging + actual data)
- In x86-64 with 4-level paging $\Rightarrow$ 5 memory accesses

That is a slowdown of 3 to 5!

$\Rightarrow$ Use cache (TLB)

Internal fragmentation

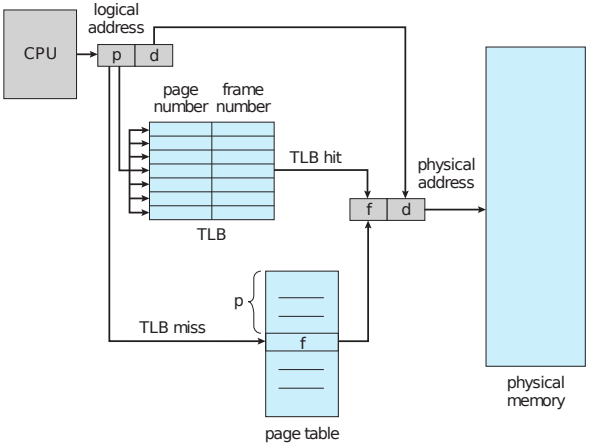
Fixed-size frames waste potentially memory

$\Rightarrow$ Use `malloc(3)`

TLB

Translation Lookaside Buffer (TLB)

Cache of the MMU



- Translated addresses are cached
- Kernel needs to invalidate TLB on page table changes
  - Writing to `CR3` invalidates entire TLB
  - Special instruction `invlpg` invalidates specific virtual address

Malloc

Dynamic memory allocations

`malloc(3)`

C standard library function for dynamic memory allocation

- `malloc(3)` returns an address with **at least** as much memory as asked for
- libc asks the kernel to map more pages into its virtual address space
- System calls for memory allocation:
  - For mapping: `mmap(2)`
  - For unmapping: `munmap(2)`

mmap(2)

```
void *mmap(void *addr, size_t len, int prot, int flags,
           int fd, off_t offset);
```

Behaves like `malloc(3)` but returns **entire** pages.

`mmap(2)` can also map files into memory (more later)

`malloc(3)`

- `mmap(2)` gives you pages
- For more finegrained allocations use `malloc(3)`:
  - Allocations within pages are managed in user space
  - Your allocator needs to keep some metadata
    - Usually kept in the pages themselves
    - Buffer overflows might corrupt allocator's metadata

`malloc(3)` reduces **internal fragmentation**