Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Descartes
research

Meltdown & Spectre

Offenes Informatikkolloqium SS 2018

Lukas Iffländer

28.06.2017

http://se.informatik.uni-wuerzburg.de/

Content

- How it all began
- Operating Systems and Computer Architecture (Crash Course)
- Branch Prediction
- Out-of-order Execution
- Meltdown
- Spectre
- Are we all going to die?

HOW IT ALL BEGAN

1995

There‘s always someone …

… who told you so.

The Intel 80x86 processor architecture: pitfalls for secure systems [1]

Oktober 2017

D. Gruss et. al: KASLR is Dead: Long Live KASLR [2]

(KASLR: Kernel Adress Space Layout Radomization)

‘The resulting patch set (still called "KAISER") is in its third revision and seems likely to find its way upstream in a relatively short period of time.’

Jonathan Corbet [3]

November 27, 2017

‘We’re presumably going to have to back-port these things anyway for the LTS releases, and they’re still getting comments and small fixes after the merge window is over. I’d rather have people feel like they can take the time to just get it all clean and finished (and as much testing as possible) than start merging things aggressively.’

Linus Torwalds

https://lwn.net/Articles/741882/

December 2017

- Sat Dec 16 2017: KPTI (Kernel Page Tabel Isolation) v149

  ‘In case someone wonders. V149 is my version number of the patch queue since this whole endeavour started. Hillarious, isn’t it? But alone this reshuffling created 22 new versions because I do that very fine grained and archive each step in case something goes wrong.’

  Thomas Gleixner

- Mon Dec 18 2017: KPTI v163

- Wed Dec 20 2017

  ‘This is a fundamental change to how the kernel’s memory management works and is the sort of thing that one would ordinarily expect to see debated for years, especially given its associated performance impact. KPTI remains on the fast track, though.’

  Jonathan Corbet

  https://lwn.net/Articles/741878/

January 4, 2017

‘This is not a happy time for the computing industry,‘

Notes from the Intelpocalypse [4]

What happened?

February 2, 2017

- CVE-2017-5754: non-disclosed
- CVE-2017-5753: non-disclosed
- CVE-2017-5715: non-disclosed

January, 2018

- CVE-2017-5754: **Meltdown:** rogue data cache load
- CVE-2017-5753: **Spectre v1:** bounds check bypass
- CVE-2017-5715: **Spectre v2:** branch target injection

Who reported Meltdown?

- Jann Horn (Google Project Zero)
- Werner Haas, Thomas Prescher (Cyberus Technology)
- Daniel Gruss, Moritz Lipp, Stefan Mangard, Michael Schwarz (Graz University of Technology)

Who reported Spectre?

- Jann Horn (Google Project Zero)
- Paul Kocher in collaboration with Daniel Genkin (University of Pennsylvania and University of Maryland), Mike Hamburg (Rambus), Moritz Lipp (Graz University of Technology), and Yuval Yarom (University of Adelaide and Data61)

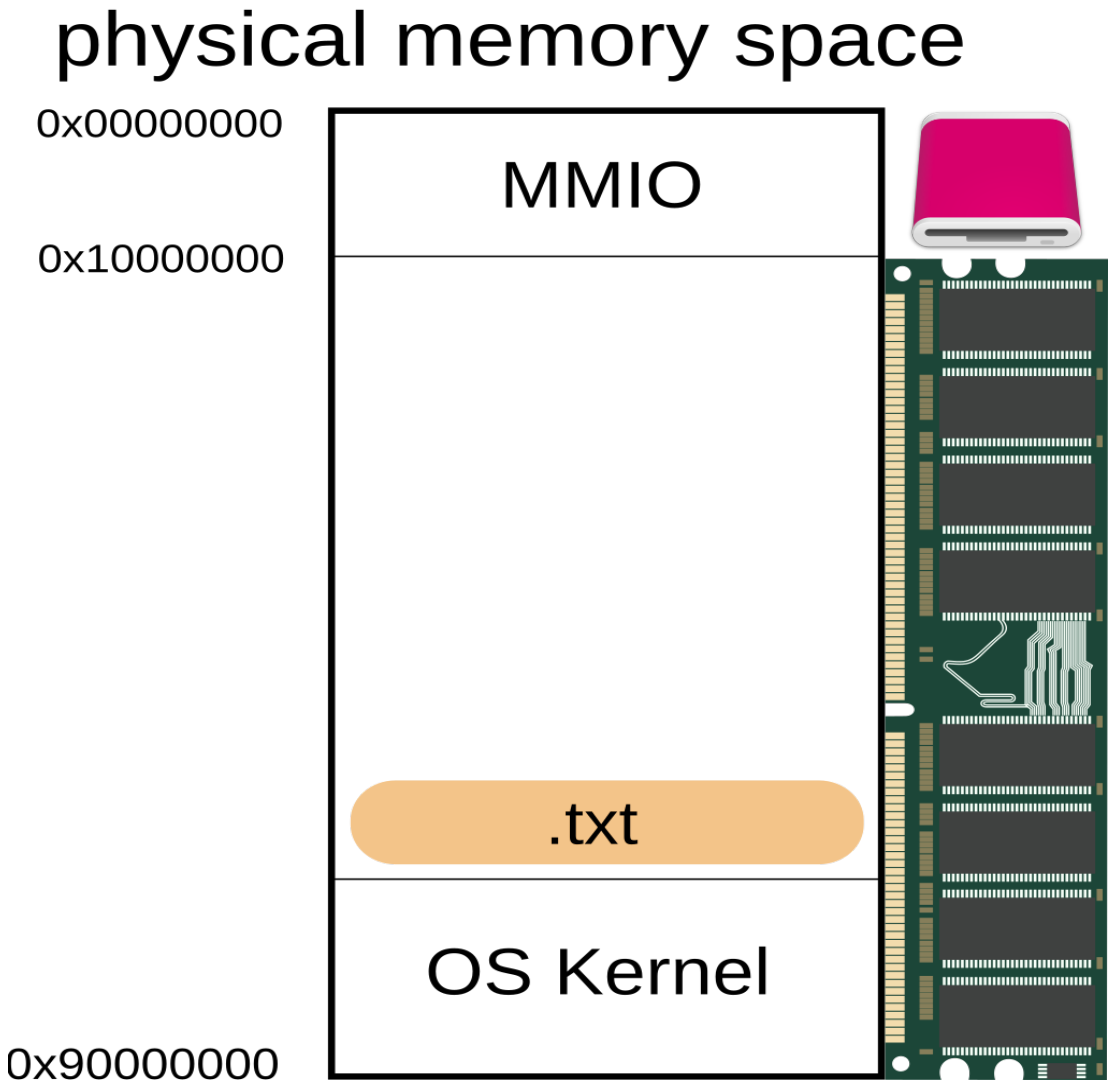
In 30 Minutes (sorry)

OPERATING SYSTEMS & COMPUTER ARCHITECTURE

OPERATING SYSTEMS

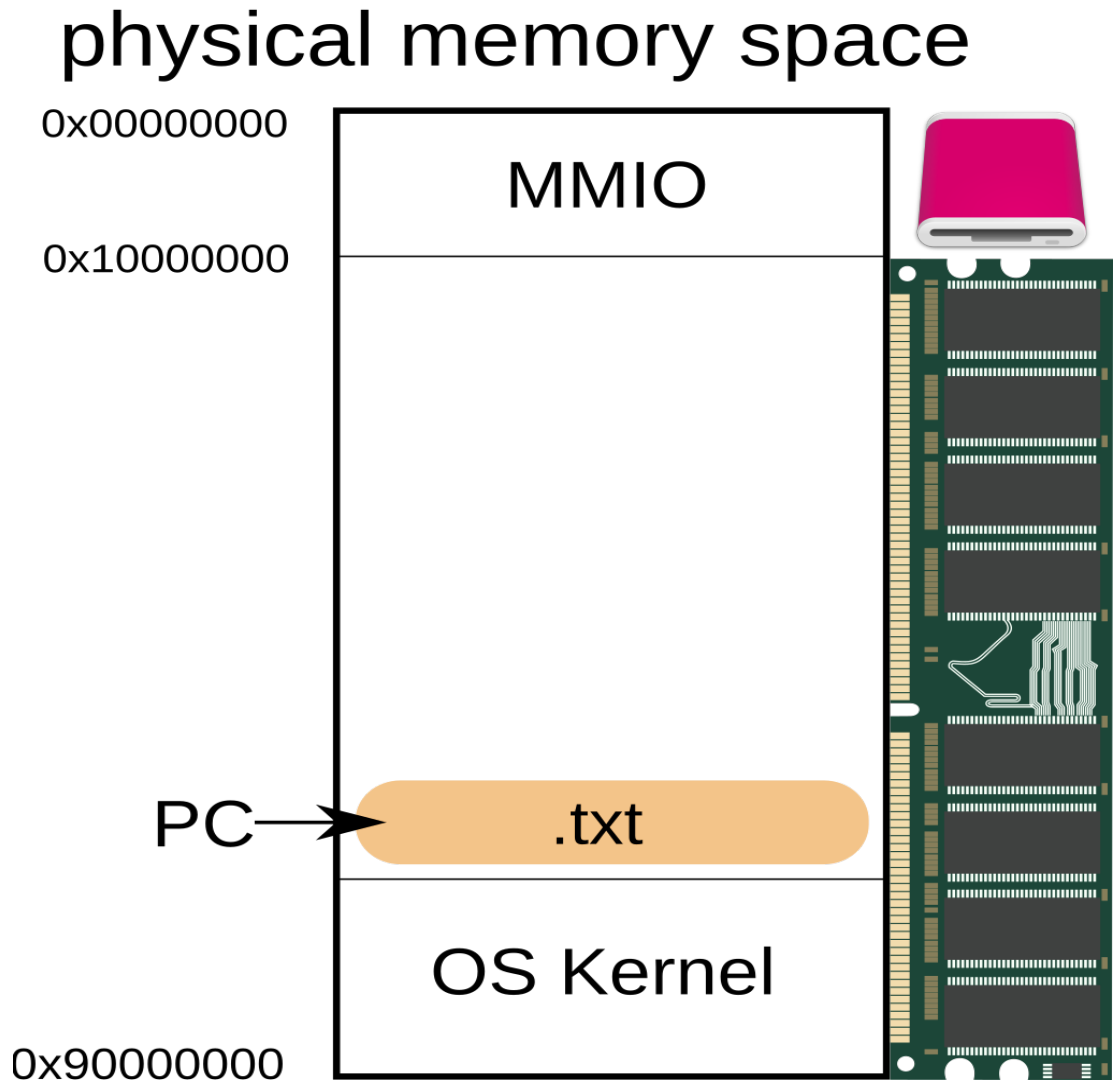
Physical Adress Space

- Raw access via system's address bus (storages)
  - RAM
  - MMIO-based Devices (USB, Audio, SATA, ...)
- Dark ages: processes used physical memory directly
- Today: often used for special-purpose systems
- But: no cross-application protection, flat address space



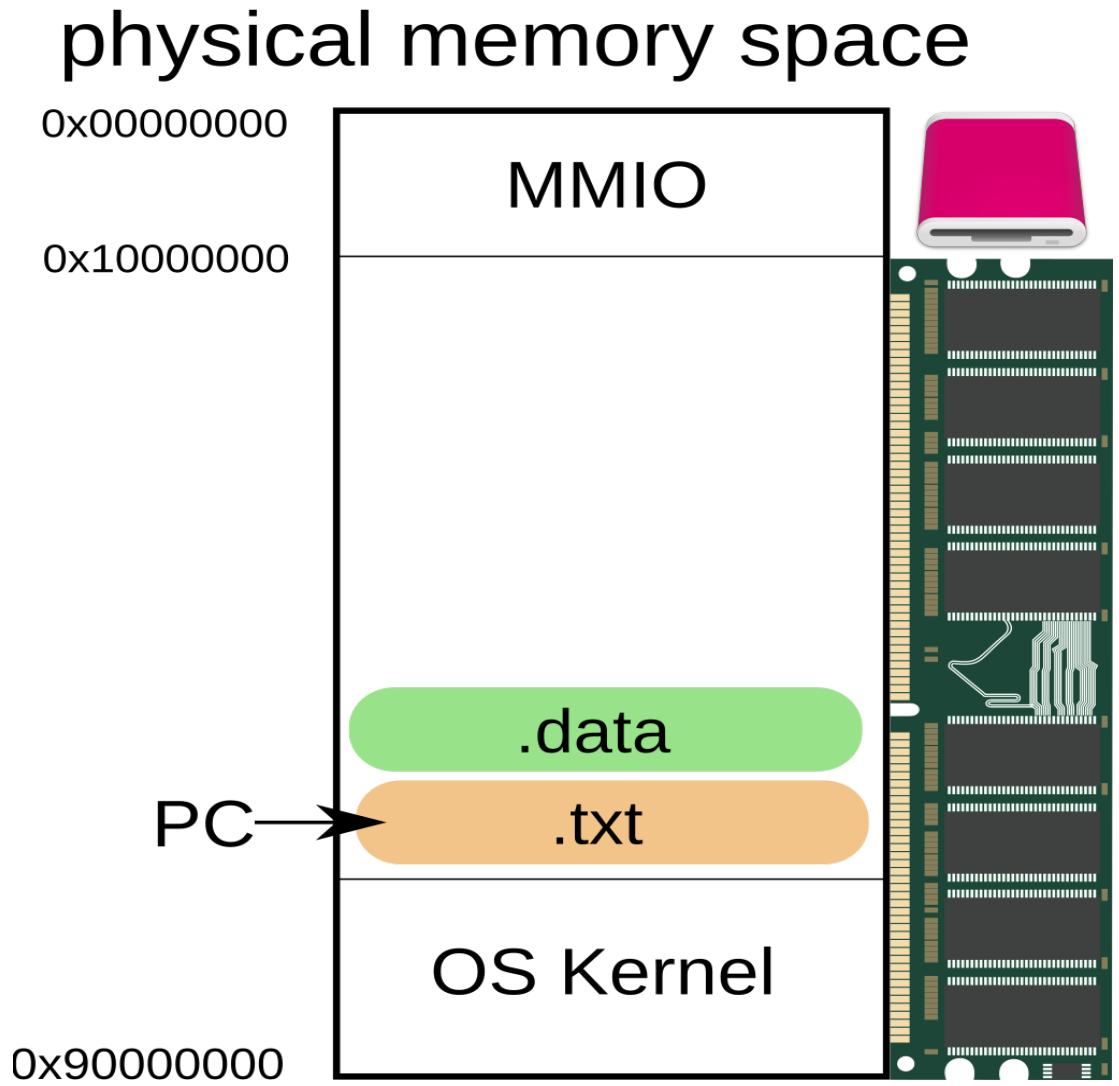
Physical Adress Space

- Raw access via system's address bus (storages)
  - RAM
  - MMIO-based Devices (USB, Audio, SATA, ...)
- Dark ages: processes used physical memory directly
- Today: often used for special-purpose systems
- But: no cross-application protection, flat address space



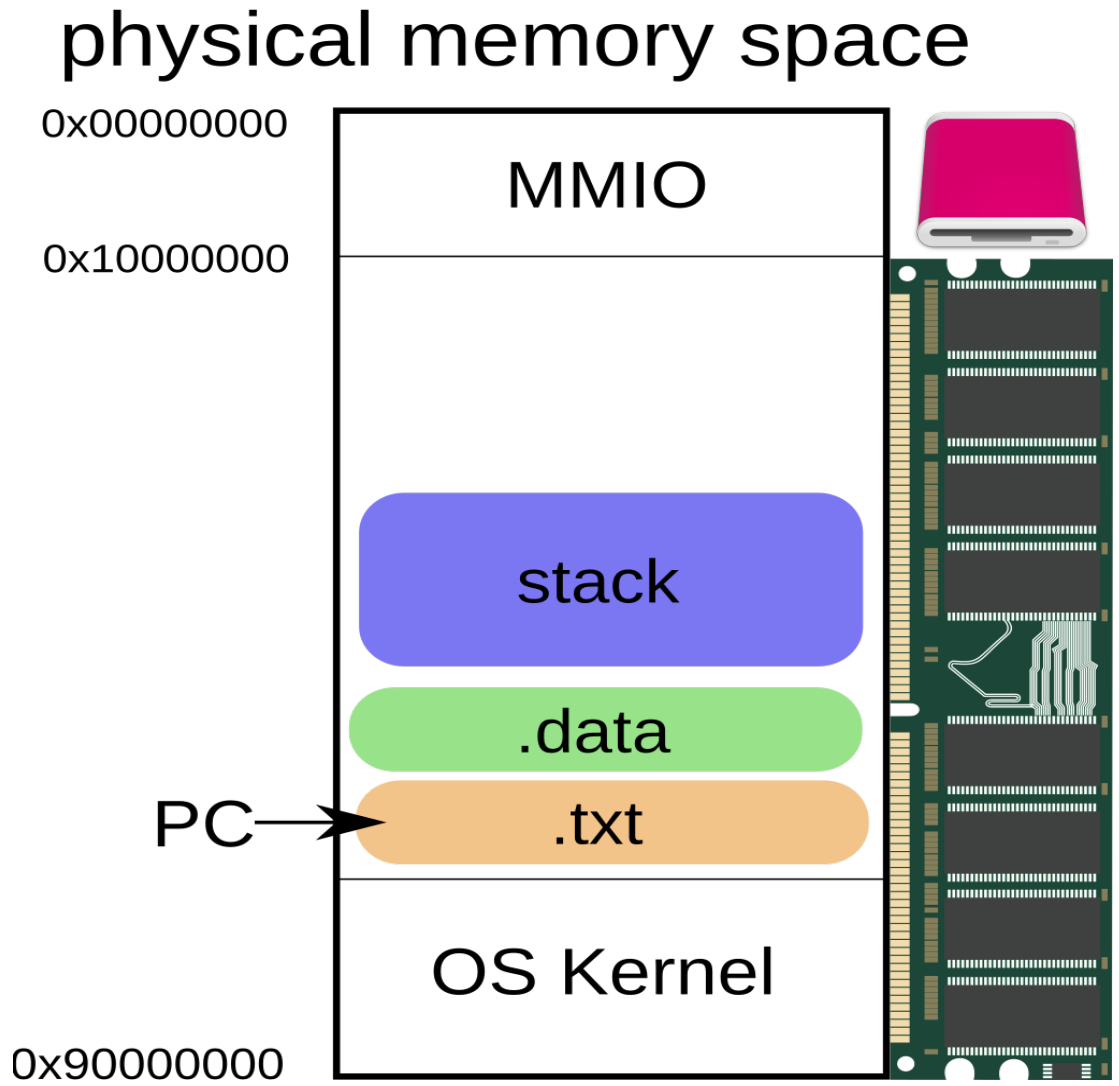
Physical Adress Space

- Raw access via system's address bus (storages)
  - RAM
  - MMIO-based Devices (USB, Audio, SATA, ...)
- Dark ages: processes used physical memory directly
- Today: often used for special-purpose systems
- But: no cross-application protection, flat address space



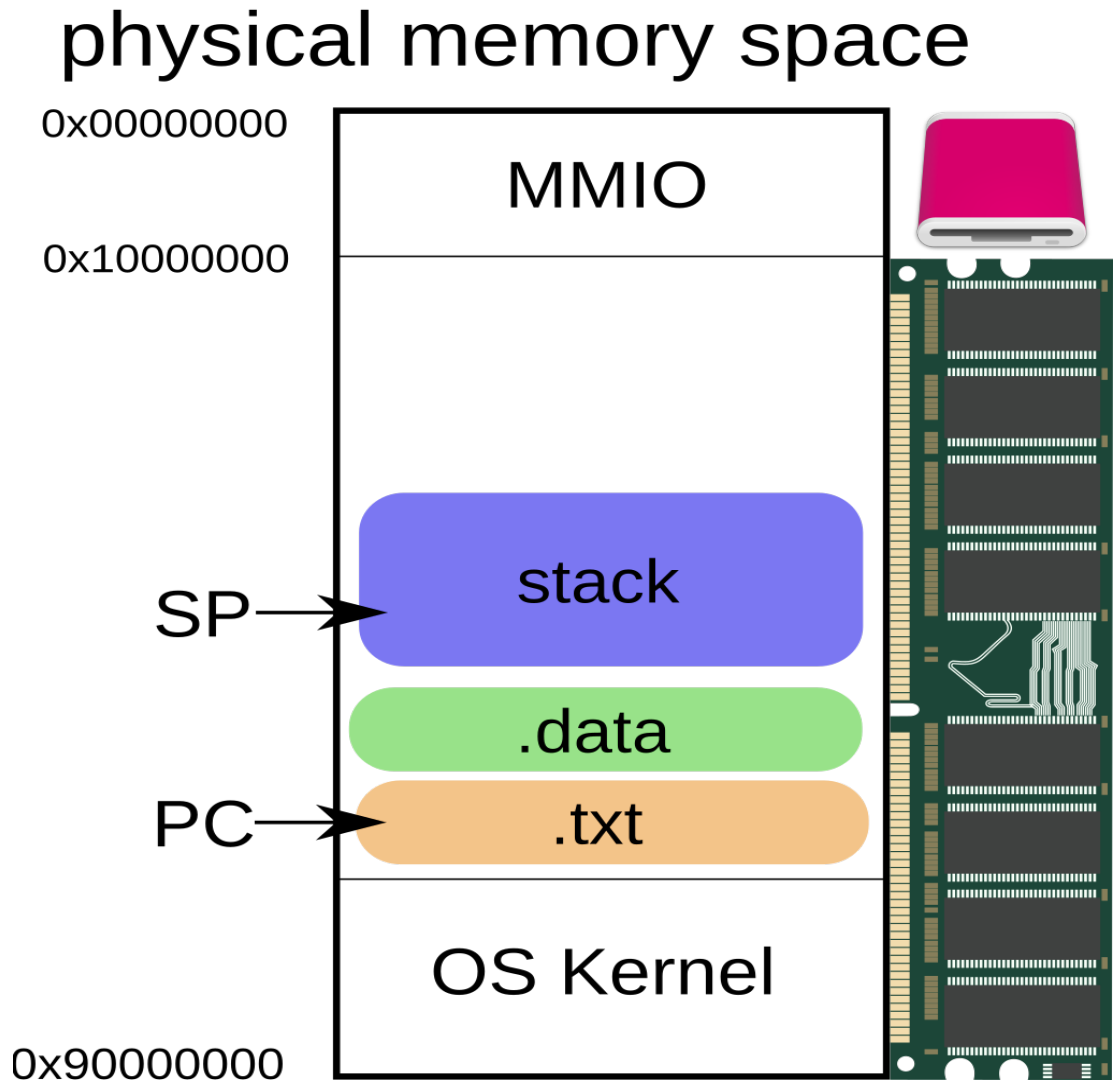
Physical Adress Space

- Raw access via system's address bus (storages)
  - RAM
  - MMIO-based Devices (USB, Audio, SATA, ...)
- Dark ages: processes used physical memory directly
- Today: often used for special-purpose systems
- But: no cross-application protection, flat address space



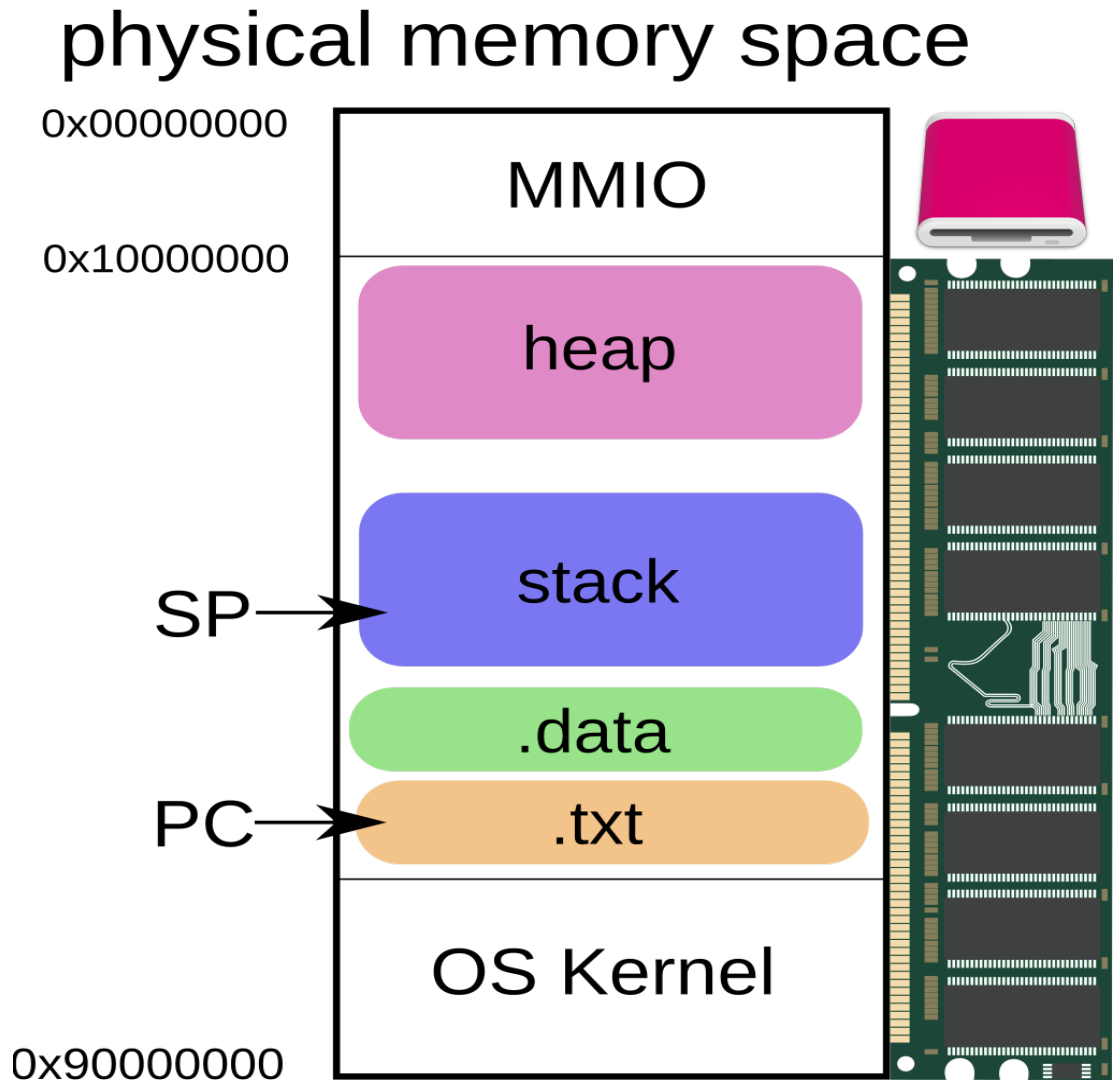
Physical Adress Space

- Raw access via system's address bus (storages)
  - RAM
  - MMIO-based Devices (USB, Audio, SATA, ...)
- Dark ages: processes used physical memory directly
- Today: often used for special-purpose systems
- But: no cross-application protection, flat address space



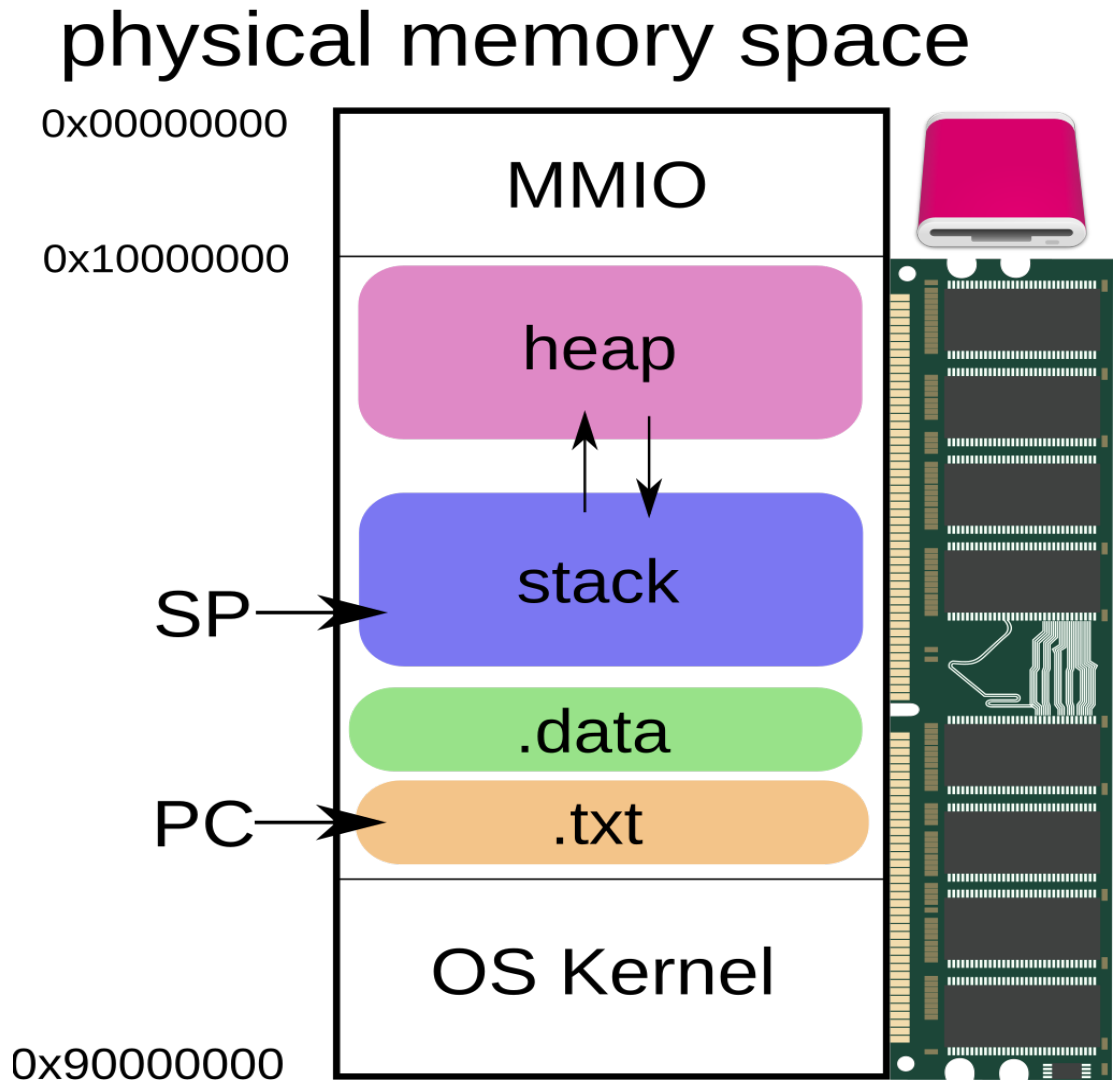
Physical Adress Space

- Raw access via system's address bus (storages)
  - RAM
  - MMIO-based Devices (USB, Audio, SATA, ...)
- Dark ages: processes used physical memory directly
- Today: often used for special-purpose systems
- But: no cross-application protection, flat address space



Physical Adress Space

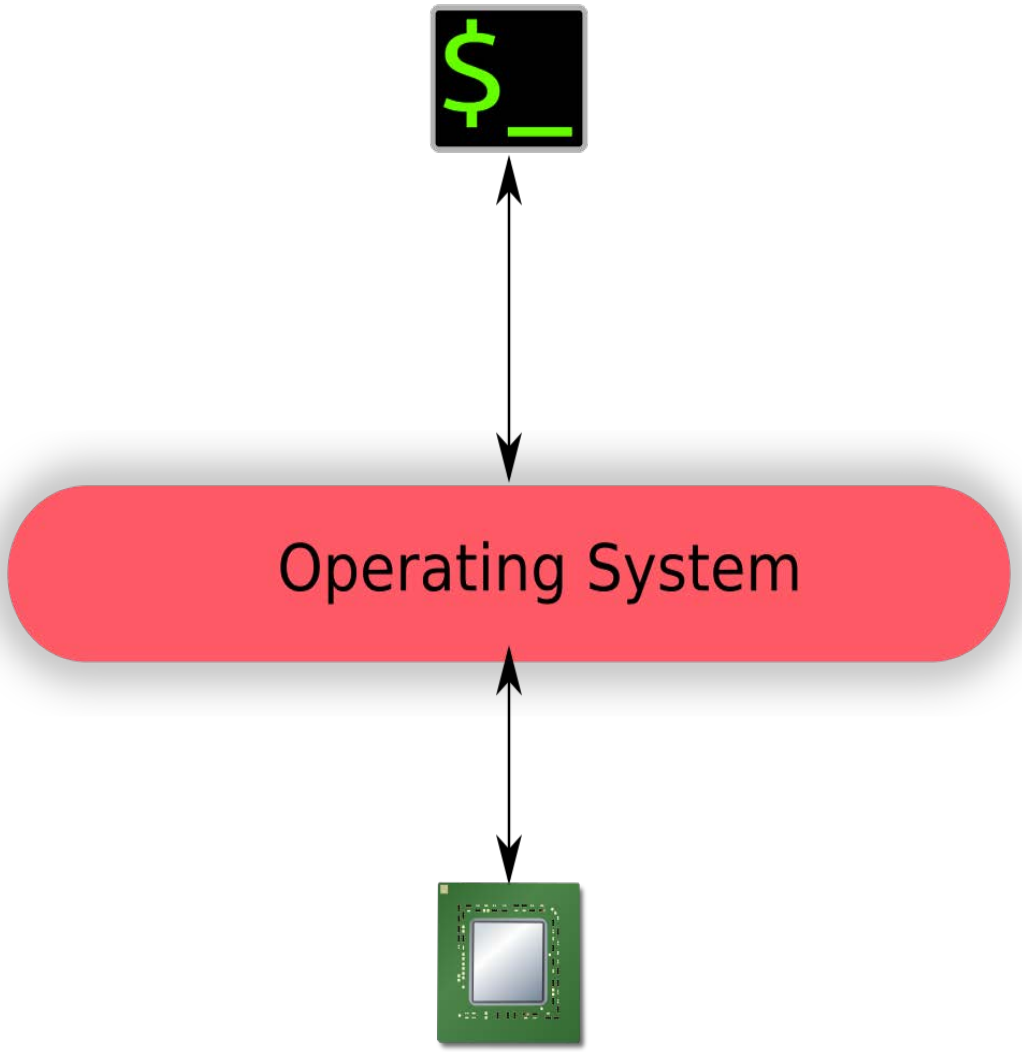
- Raw access via system's address bus (storages)
  - RAM
  - MMIO-based Devices (USB, Audio, SATA, ...)
- Dark ages: processes used physical memory directly
- Today: often used for special-purpose systems
- But: no cross-application protection, flat address space



Toward Multithreading

Deficits

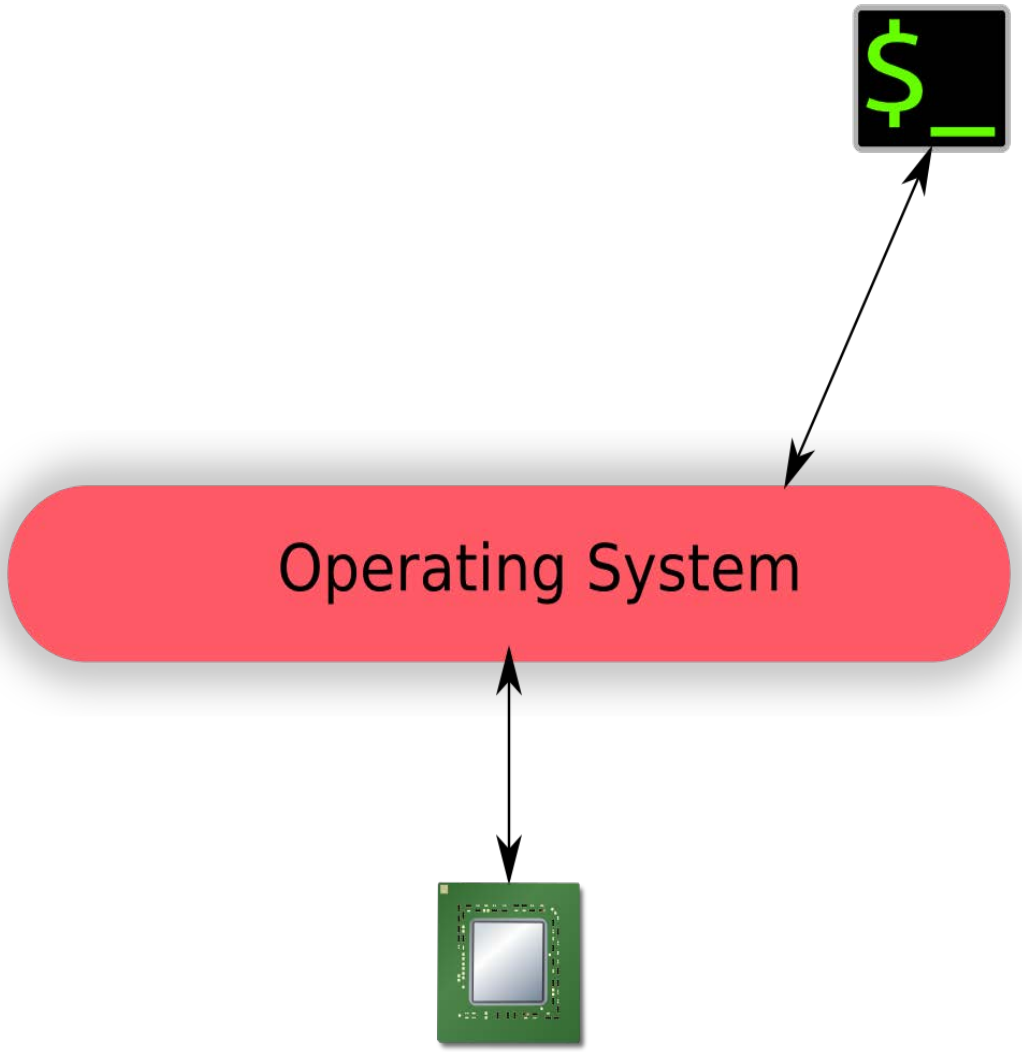
- Multitasking: run multiple programs at once
- Addresses are (were) compile-time dependant
- Protection: secure process isolation



Toward Multithreading

Deficits

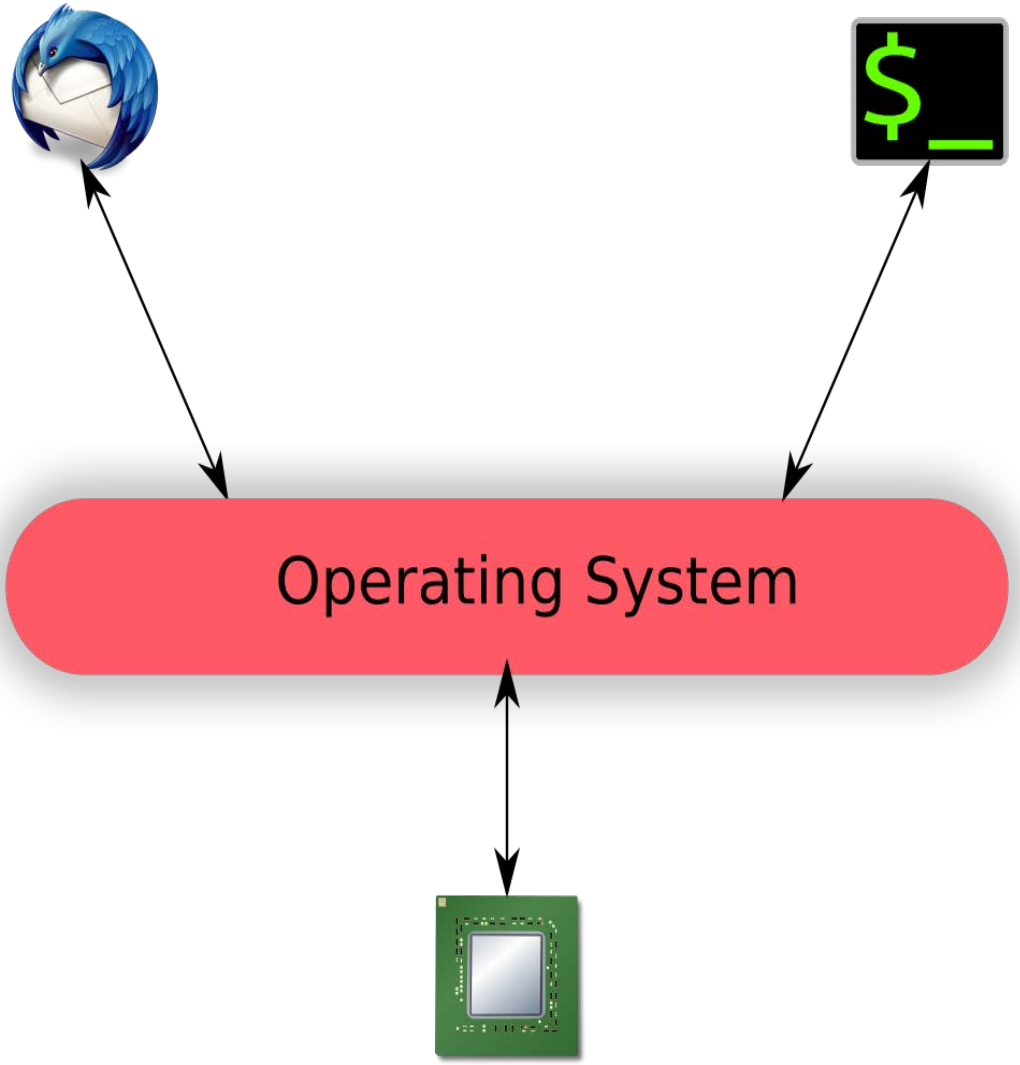
- Multitasking: run multiple programs at once
- Addresses are (were) compile-time dependant
- Protection: secure process isolation



Toward Multithreading

Deficits

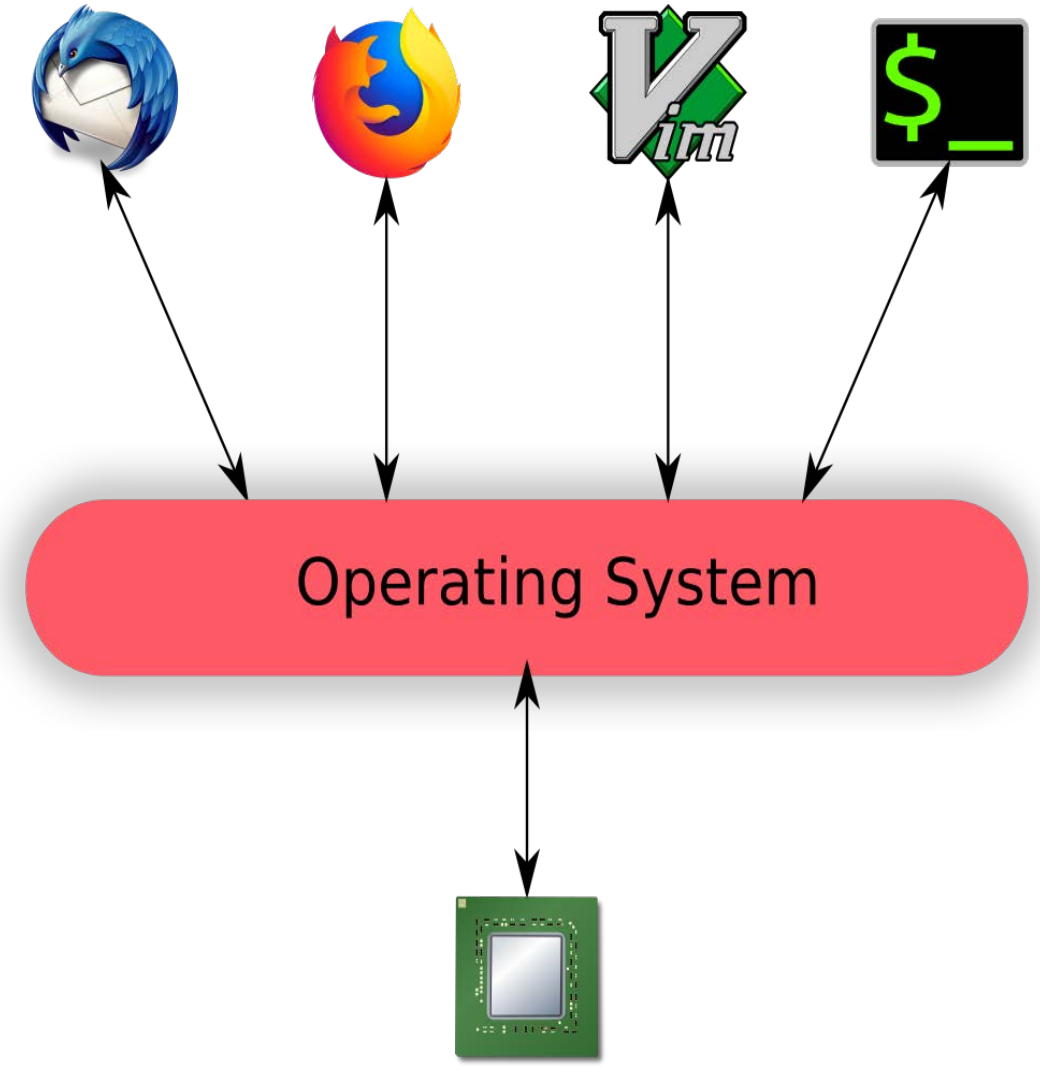
- Multitasking: run multiple programs at once
- Addresses are (were) compile-time dependant
- Protection: secure process isolation



Toward Multithreading

Deficits

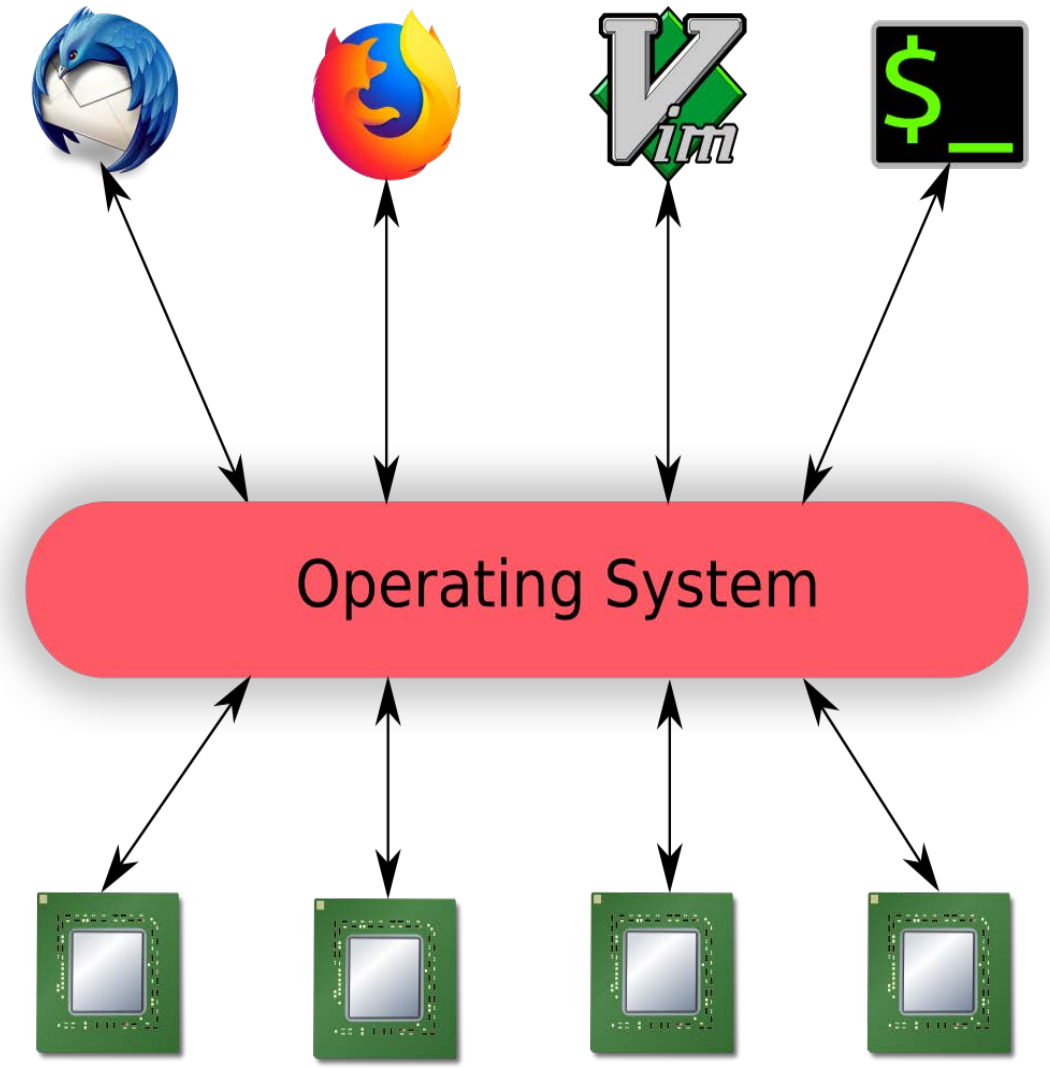
- Multitasking: run multiple programs at once
- Addresses are (were) compile-time dependant
- Protection: secure process isolation



Toward Multithreading

Deficits

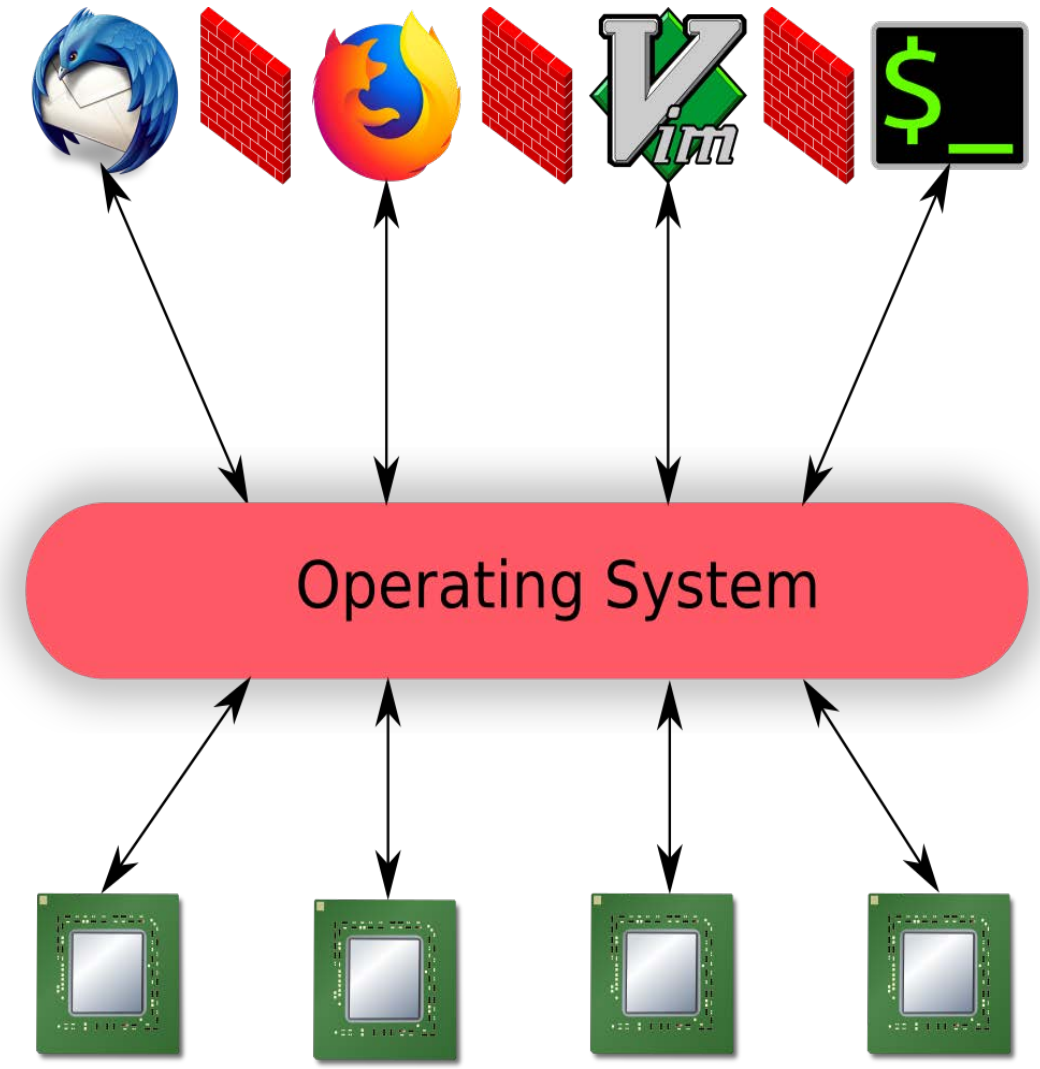
- Multitasking: run multiple programs at once
- Addresses are (were) compile-time dependant
- Protection: secure process isolation



Toward Multithreading

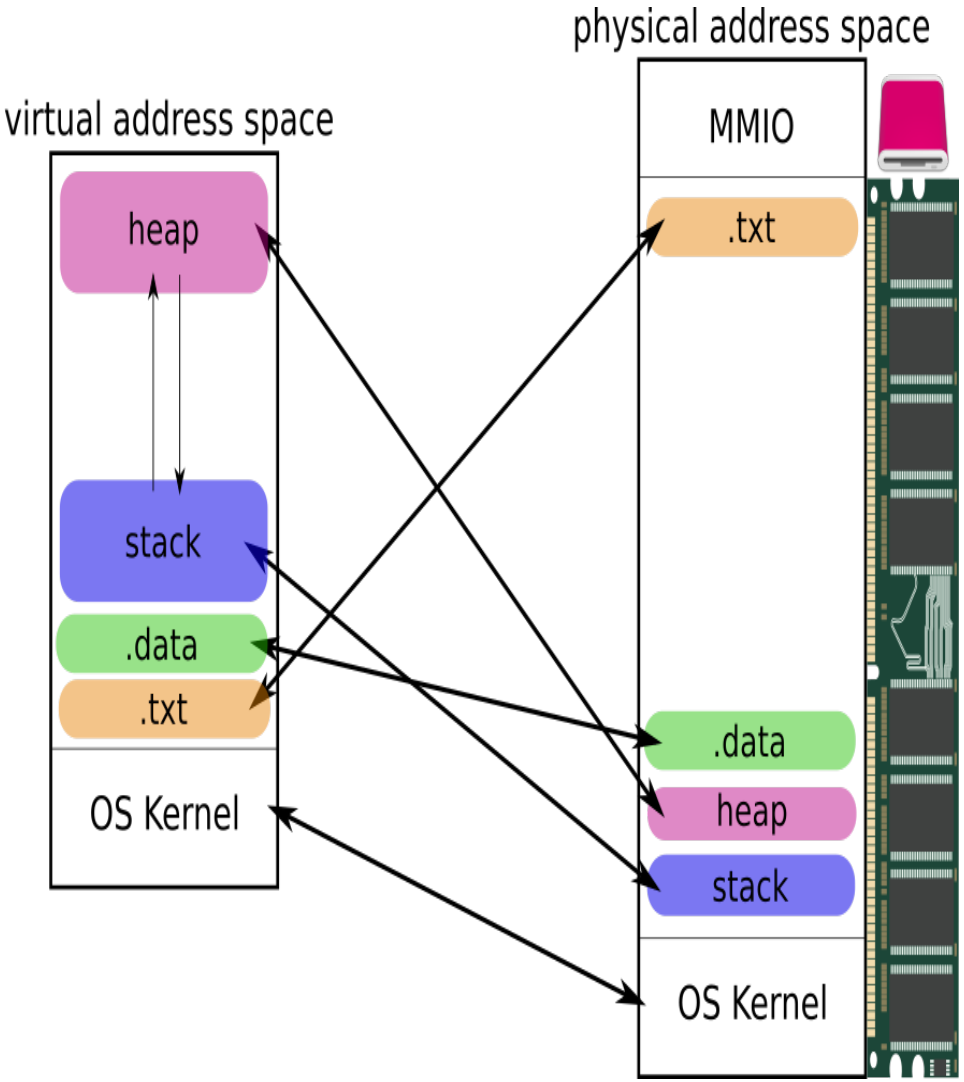
Deficits

- Multitasking: run multiple programs at once
- Addresses are (were) compile-time dependant
- Protection: secure process isolation



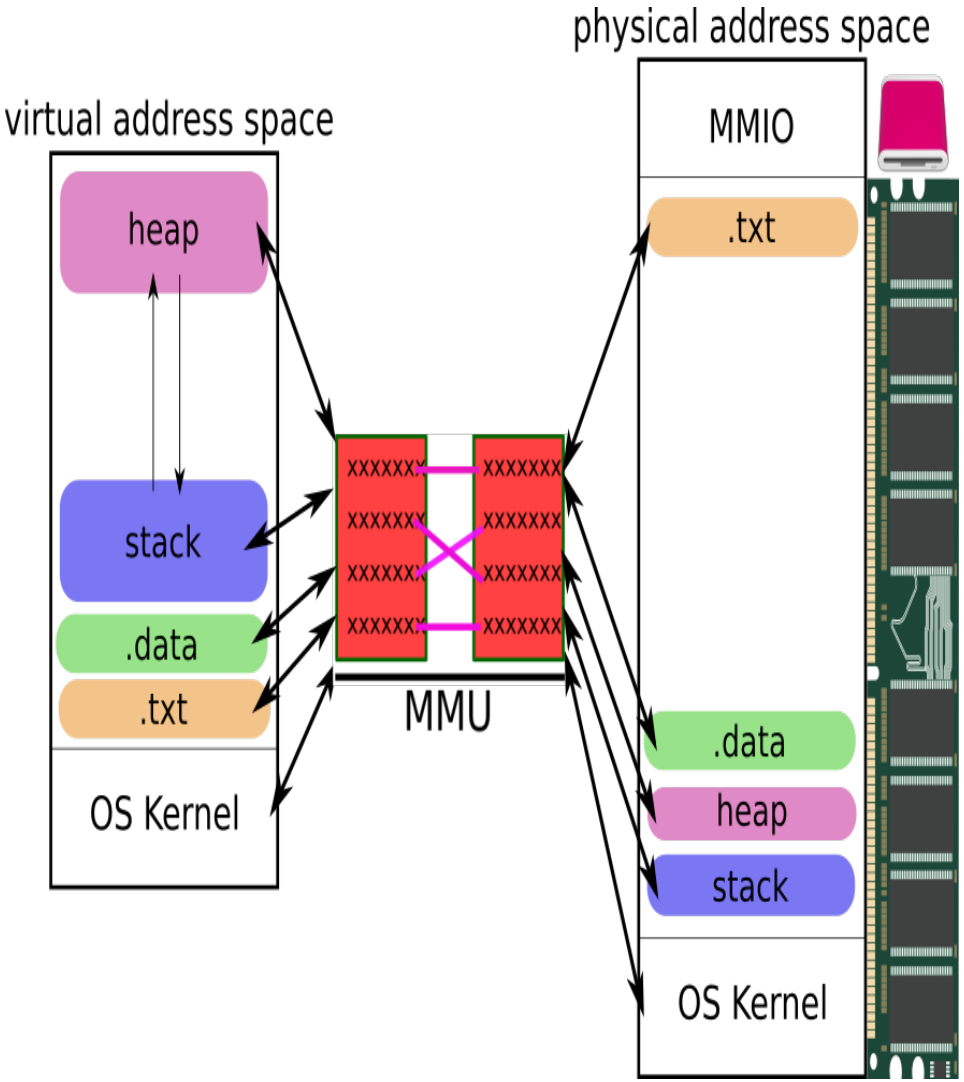
Paging / Virtual Memory

- Introduce new memory space and provide separate memory space for every process
- Transparent for CPU access
- MMU translates virtual to physical addresses
- Fast translations / lookups
- BTW: shared memory: virtual memory of different processes point to the same physical page



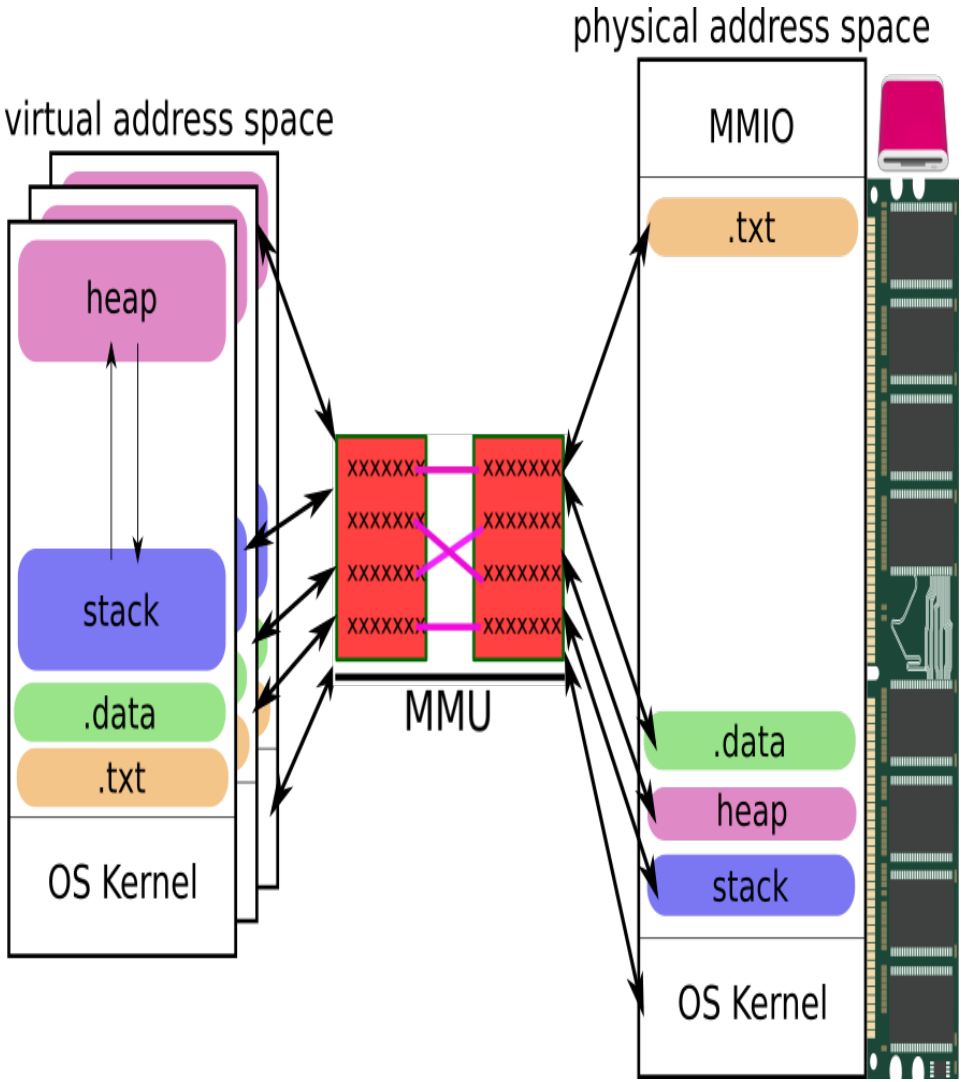
Paging / Virtual Memory

- Introduce new memory space and provide separate memory space for every process
- Transparent for CPU access
- MMU translates virtual to physical addresses
- Fast translations / lookups
- BTW: shared memory: virtual memory of different processes point to the same physical page



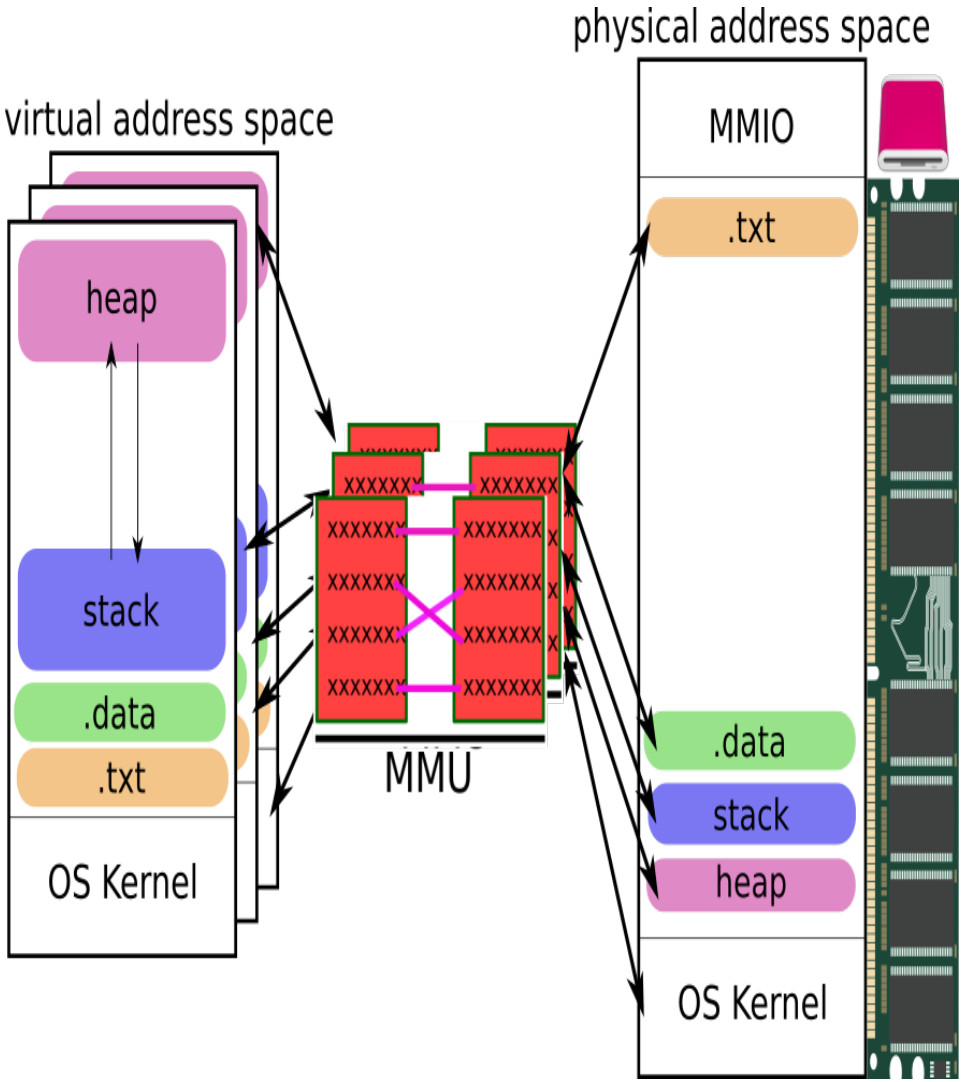
Paging / Virtual Memory

- Introduce new memory space and provide separate memory space for every process
- Transparent for CPU access
- MMU translates virtual to physical addresses
- Fast translations / lookups
- BTW: shared memory: virtual memory of different processes point to the same physical page



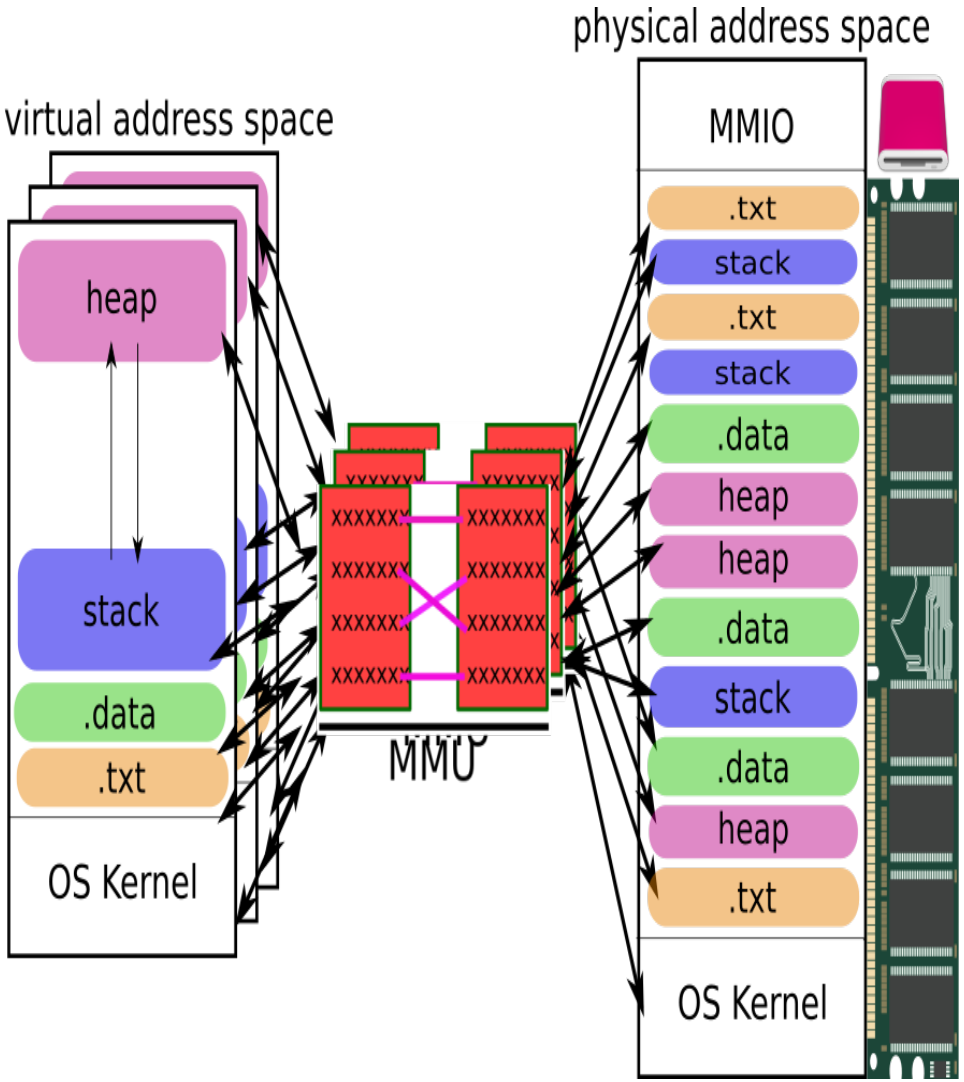
Paging / Virtual Memory

- Introduce new memory space and provide separate memory space for every process
- Transparent for CPU access
- MMU translates virtual to physical addresses
- Fast translations / lookups
- BTW: shared memory: virtual memory of different processes point to the same physical page



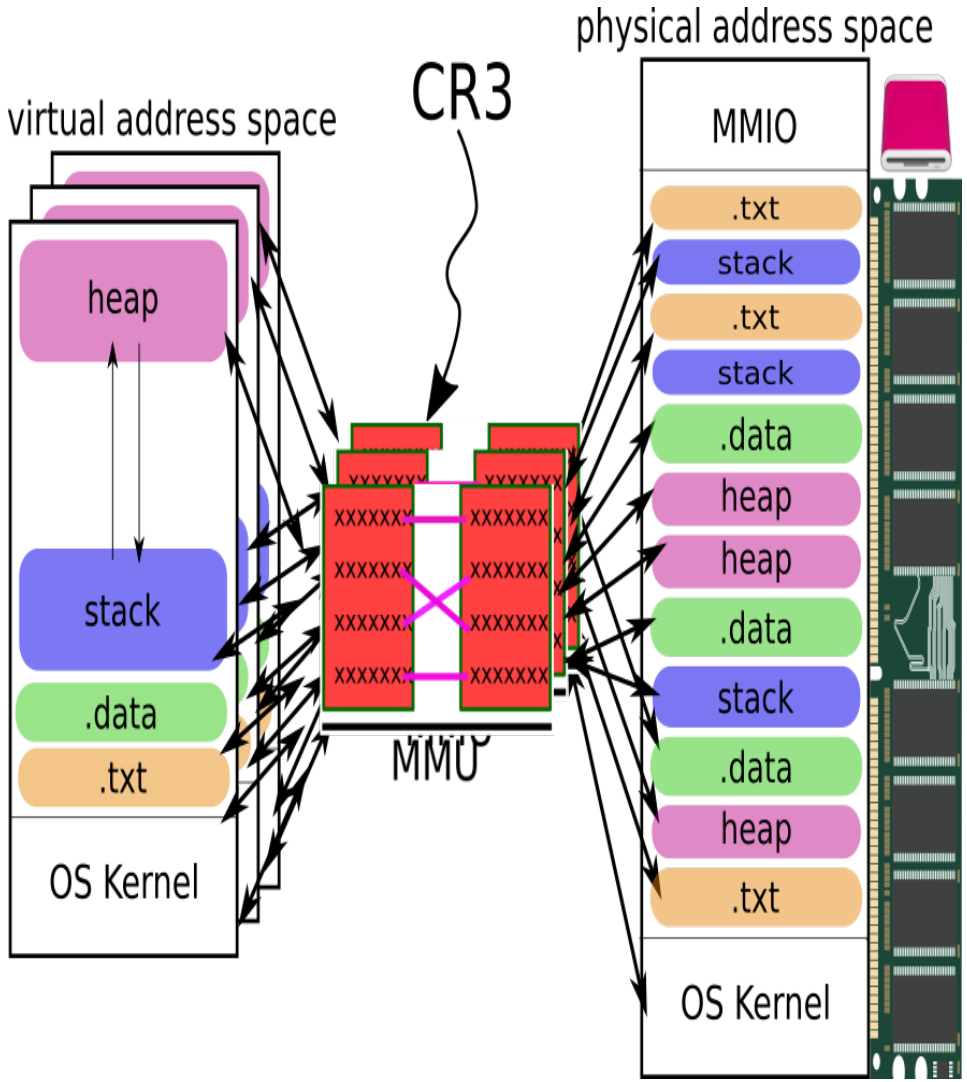
Paging / Virtual Memory

- Introduce new memory space and provide separate memory space for every process
- Transparent for CPU access
- MMU translates virtual to physical addresses
- Fast translations / lookups
- BTW: shared memory: virtual memory of different processes point to the same physical page

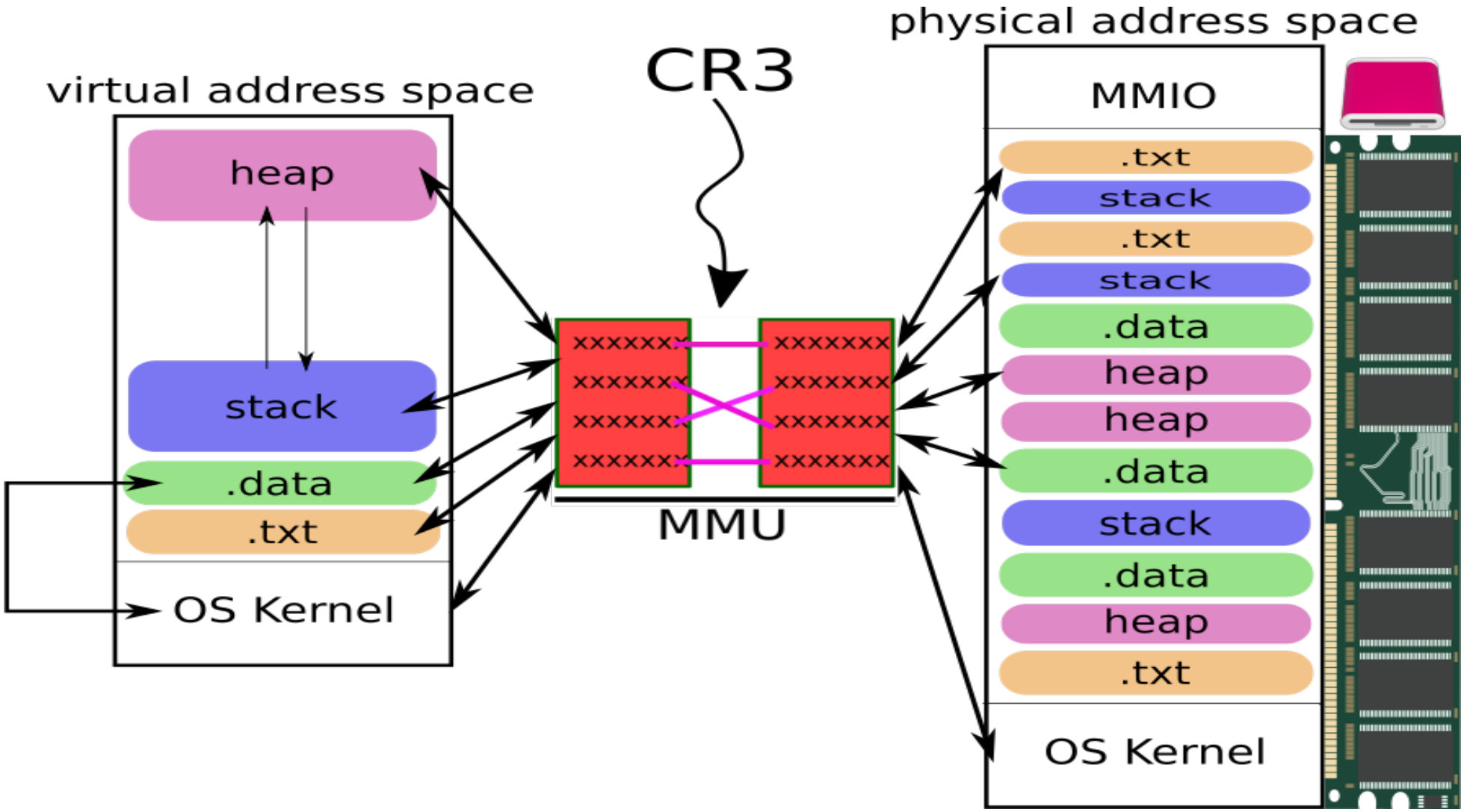


Paging / Virtual Memory

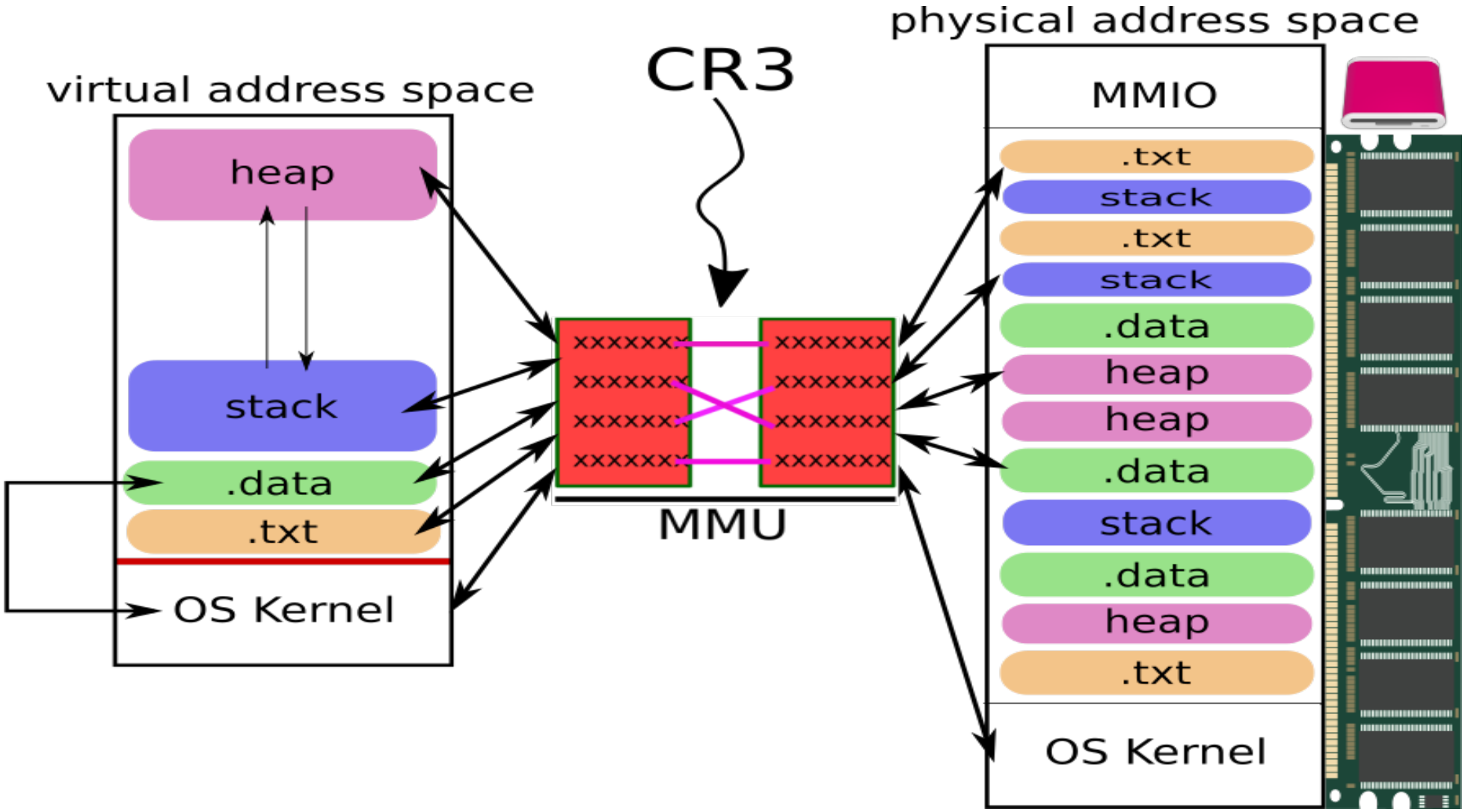
- Introduce new memory space and provide separate memory space for every process
- Transparent for CPU access
- MMU translates virtual to physical addresses
- Fast translations / lookups
- BTW: shared memory: virtual memory of different processes point to the same physical page



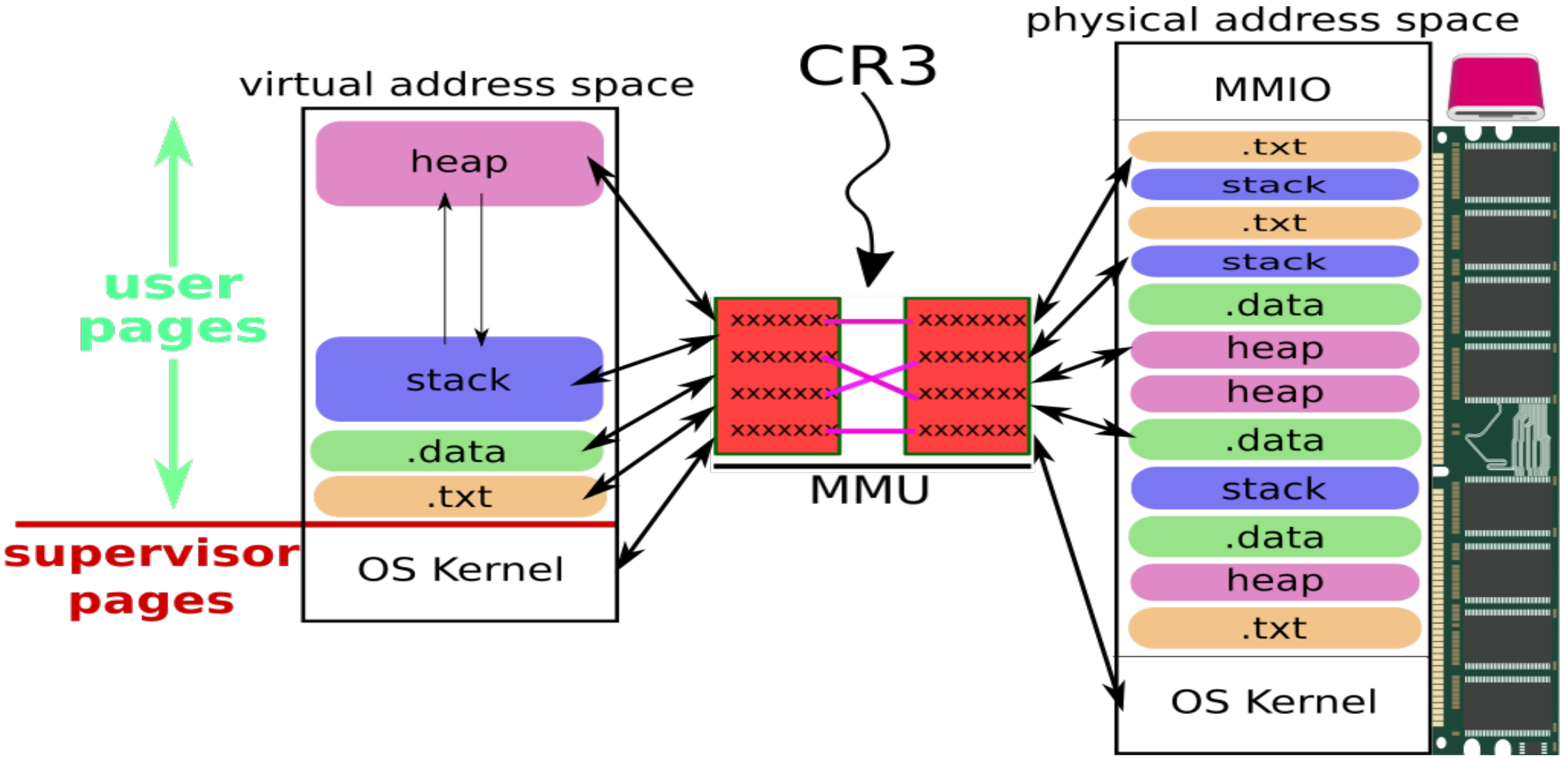
Paging (Access)



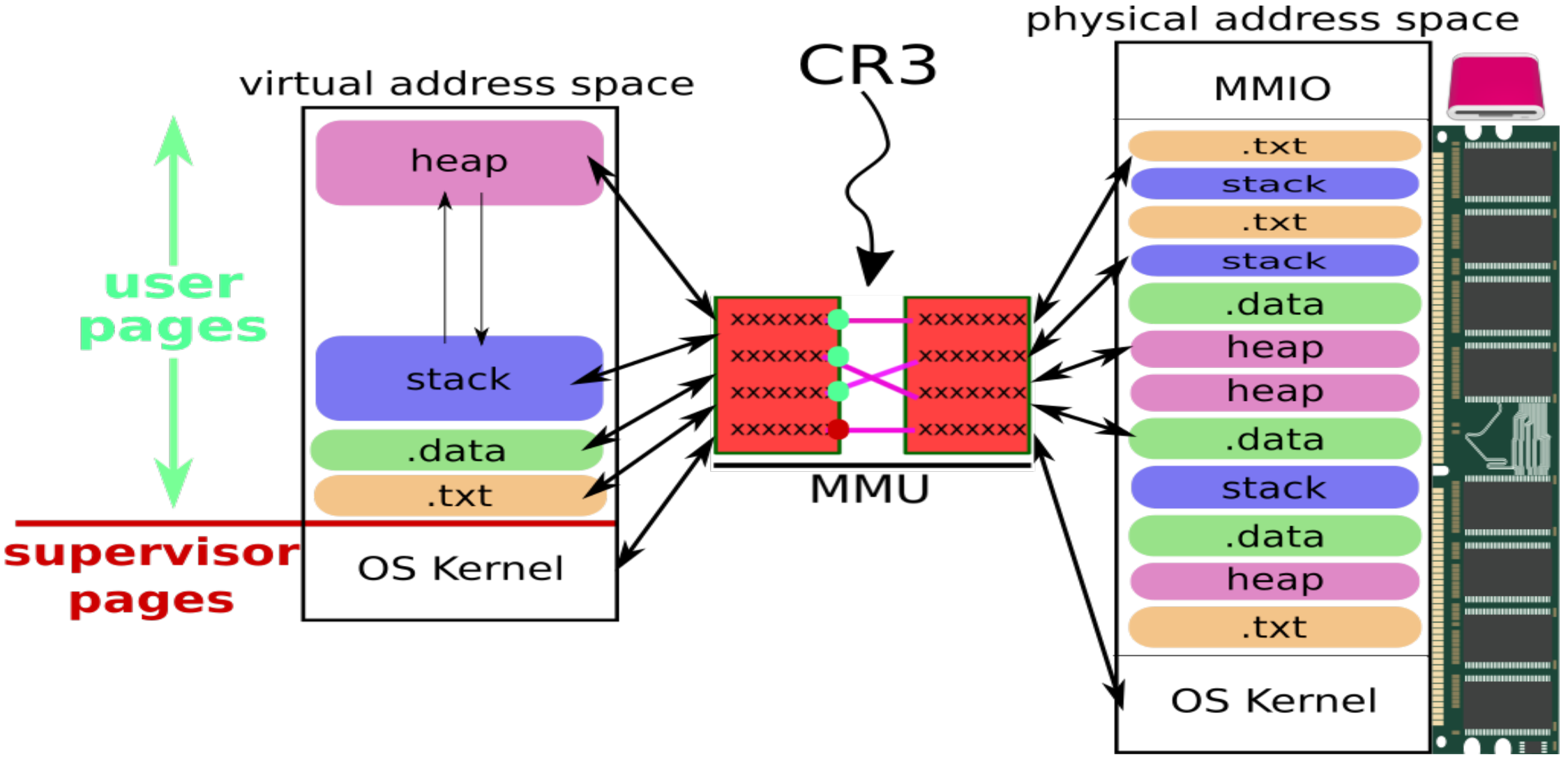
Paging (Access)



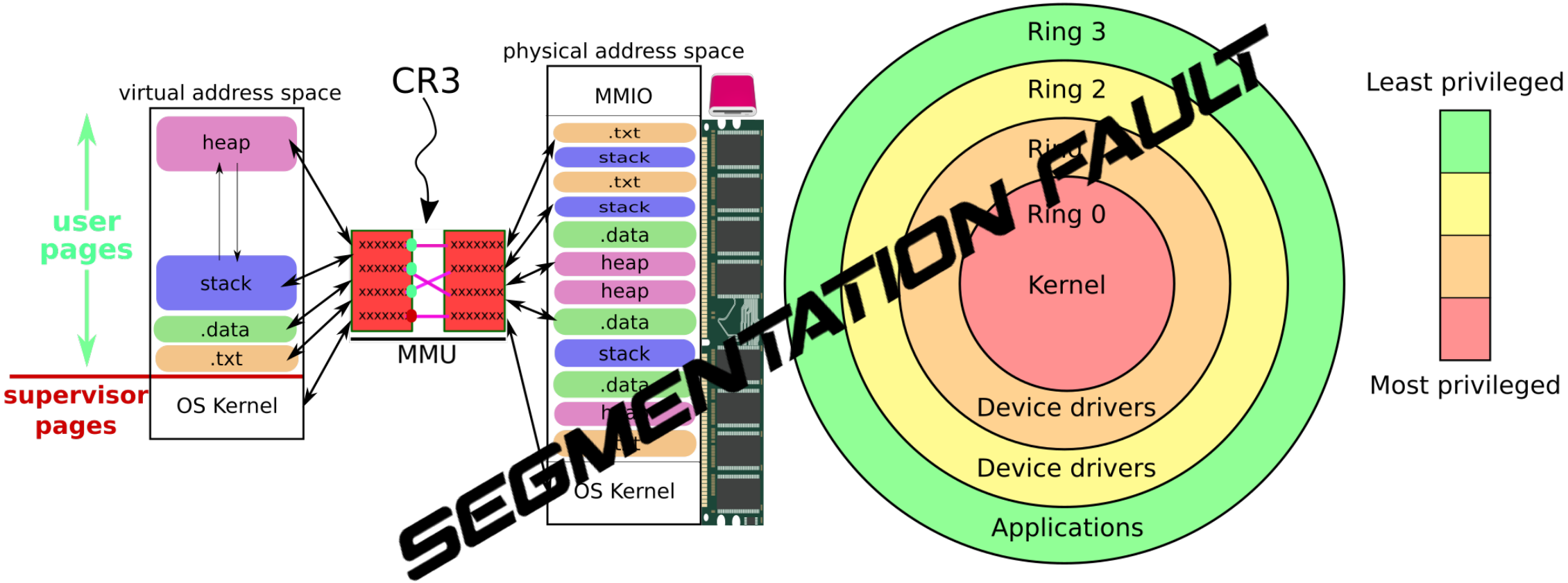
Paging (Access)



Paging (Access)



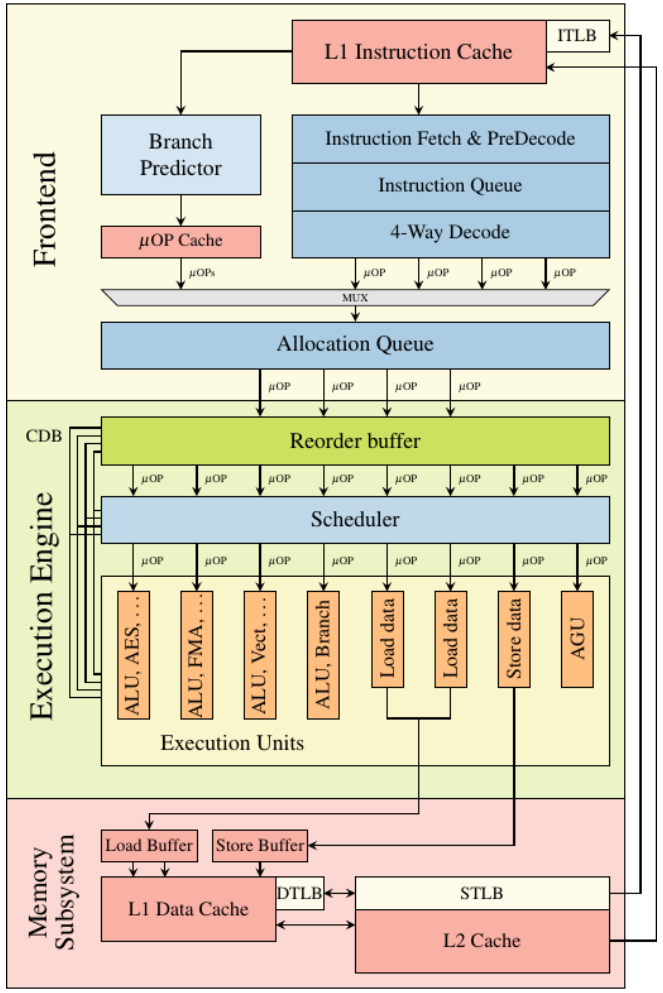
Protection Rings



CPU ARCHITECTURE

Overview

- Chip designs have come a long way:
  - Pipelining (1961: IBM 7030 Stretch)
  - Caches (1968: IBM IBM System/360 Model 85)
  - Branch Prediction (broad adoption with superscalar architectures)
  - Out-of-order Execution (broad adoption in the mid 90s)
- Modern CPU designs do things that the most programmers are not aware of

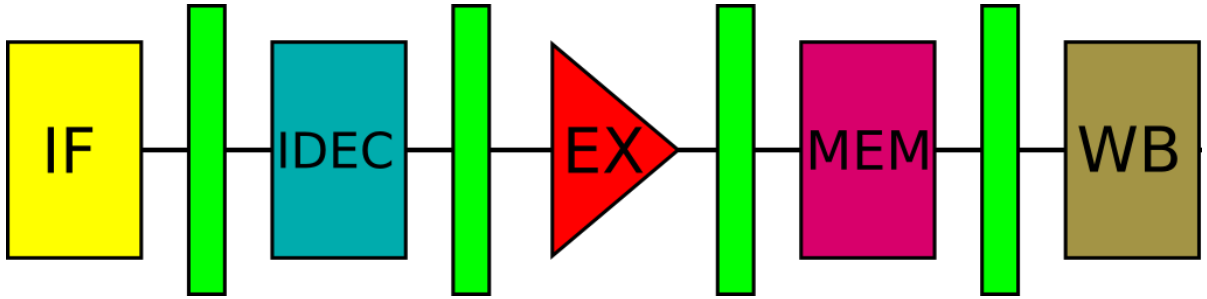


Intel Skylake Microarchitecture

© Meltdown authors [5]

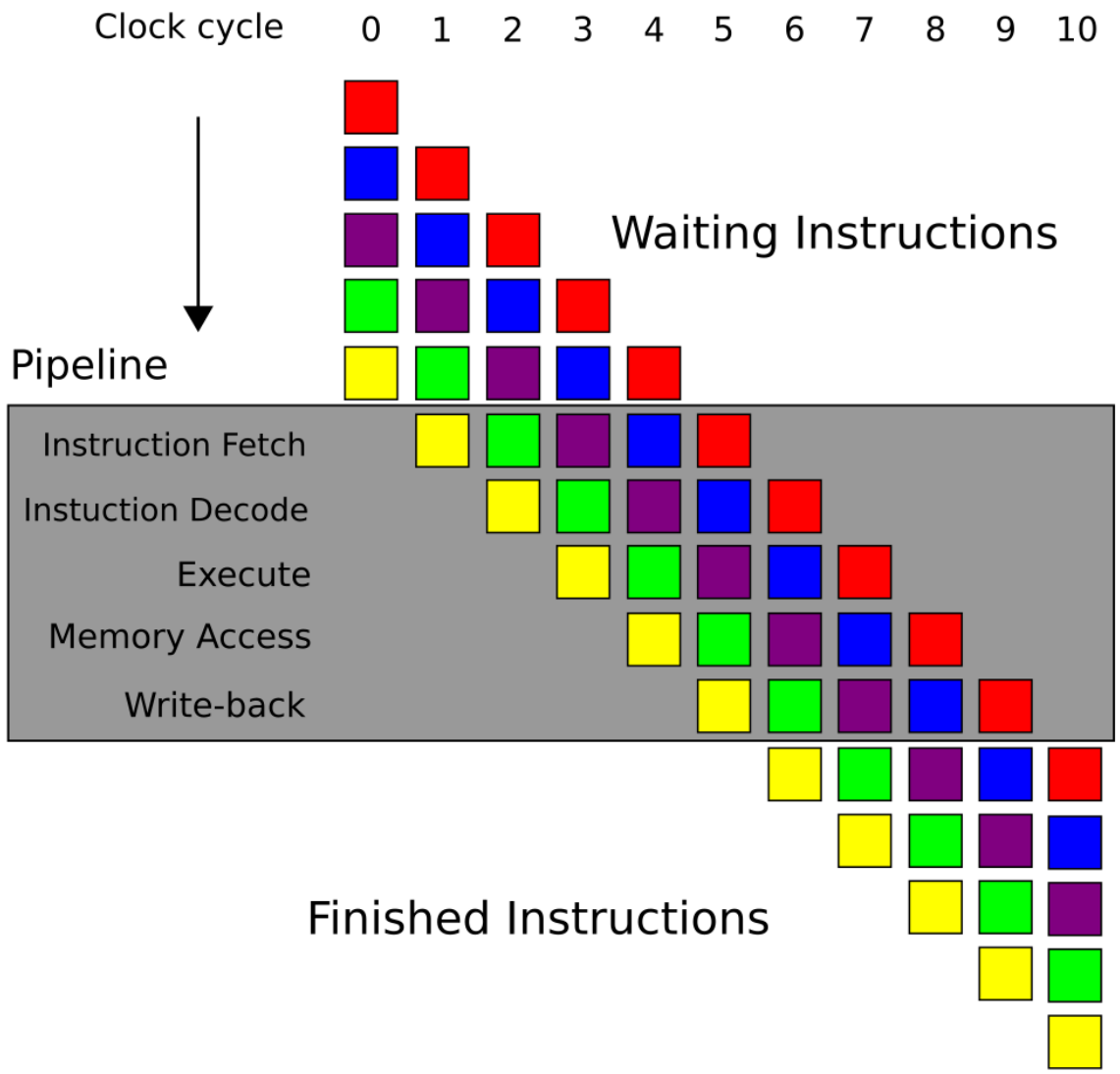
Pipelining I

- Technique for parallelism on instruction level
- Typical organization of pipeline
  - Instruction Fetch (IF)
  - Instruction Decode (IDEC)
  - Instruction Execute (EX)
  - Memory Access (MEM)
  - Result Writeback (WB)



- Pipeline stages are separated by registers to become independent
- Example: While the CPU executes an instruction (EX), it can load the next one (IF)

Pipelining II



Pipelining III

- More pipeline stages enable higher clock speeds → higher performance
- But more pipeline stages cause performance drawback for pipeline-flushes and bubbles
- Additional CPU features might require more pipeline stages
- Some examples of pipeline lengths:
  - AMD64 (K8): 12 stages
  - AMD Bulldozer: 19 ~22 stages
  - AMD Ryzen: ~18 Stages
  - Intel Pentium IV: 20 stages (Northwood), 31 stages (Prescott)
  - Intel Core2 (Merom/Penryn): ~ 14 stages
  - Intel Core i (Nehalem): ~ 17 stages
  - Intel Core i (Sandy Bridge/Ivy Bridge): ~ 14 stages
  - ARM Cortex (32bit): 8 stages (A7/A9), 15 stages (A15)
  - ARM Cortex (64bit): 10 stages (A53), 15 stages (A57/A72), 11 stages (A73)

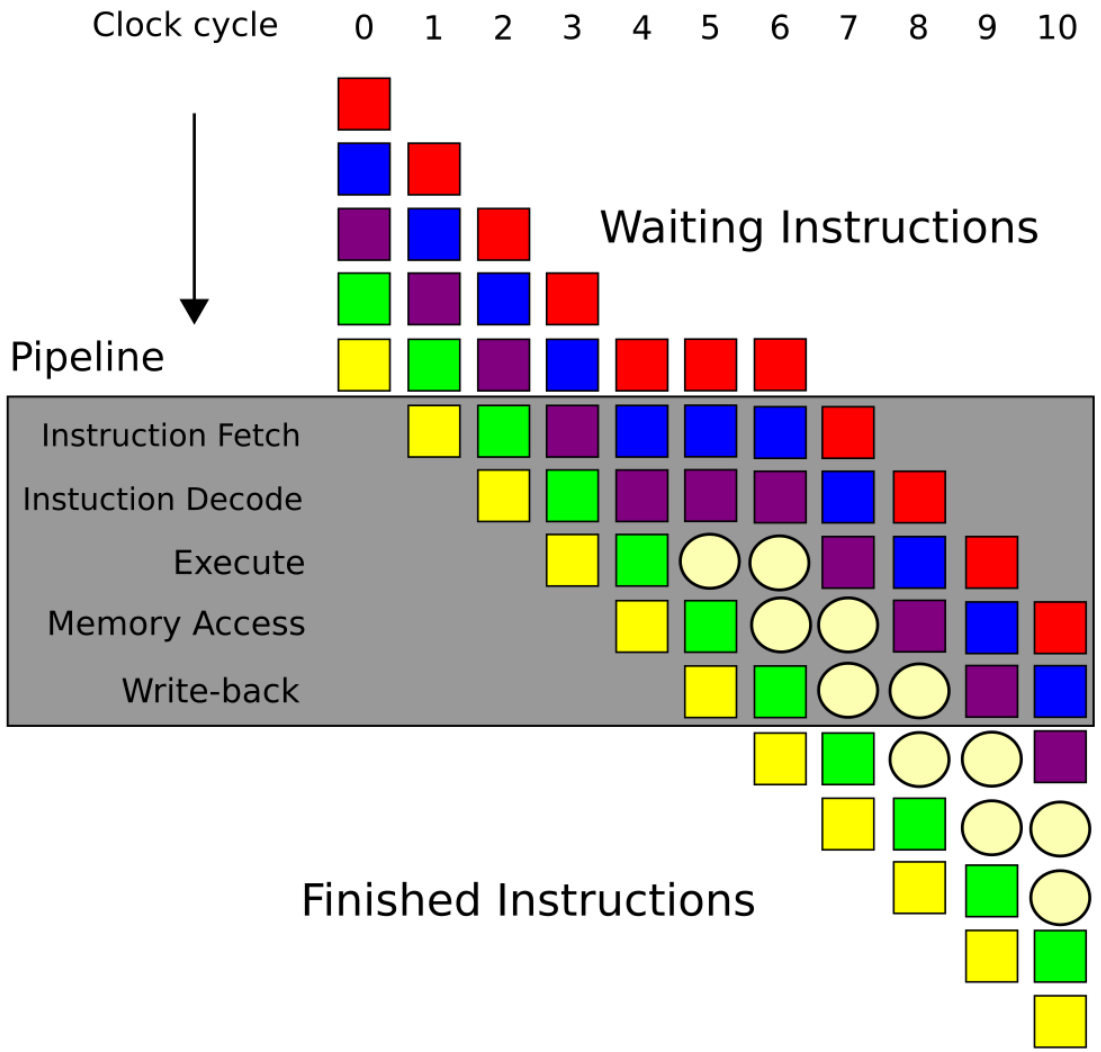
Pipelining IV

- Pipelining is controlled by Control Unit (CU)
- The CU directs the execution of instruction
- The CU detects and resolves pipeline hazards
- There are several types of pipeline hazards:
  - Data hazards: necessary operand data is not written back or is not available
  - Structural hazards: a hardware resource of the CPU is not available
  - Control hazards: outcome of branches are not resolved yet
- Example (data hazard):

```
imul edi, esi
imul edi, edx
```

- Inserting NOPs is a common way to resolve pipeline hazards

Pipelining IV

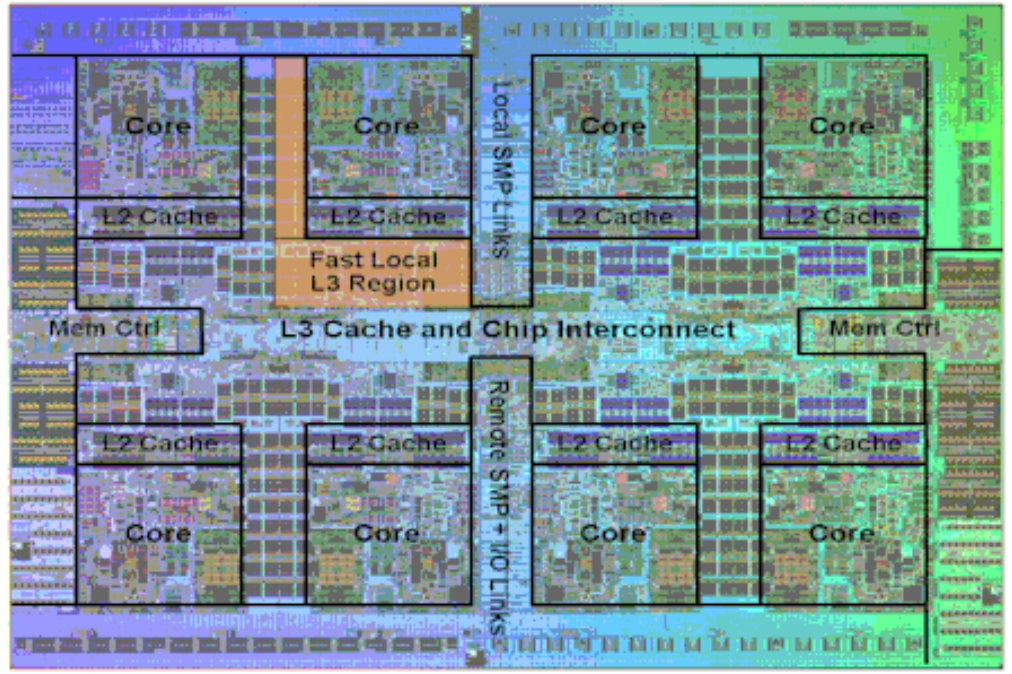


Pipeline hazards avoidance techniques

- For data hazards:
  - Out-of-order execution (arrange the instruction stream)
  - Caches (faster memory accesses)
- For structural hazards:
  - Implement additional hardware resources (e.g. more LS units)
- For control hazards:
  - Branch prediction (predict which route to take)

Caches - main memory is slow

- Cache is a on-chip/off-chip memory located closer to the CPU
- Cache is a partial reflection of the main memory
- Higher bandwidth vs. lower latency
- Little space (few MiB ↔ many GiB)
- Multiple levels of cache varying in speed and size
- Cache performance is defined by multiple factors (cache associativity, separate/unified)
- Hierarchical cache structures can be inclusive or exclusive



IBM eight core Power 7 die

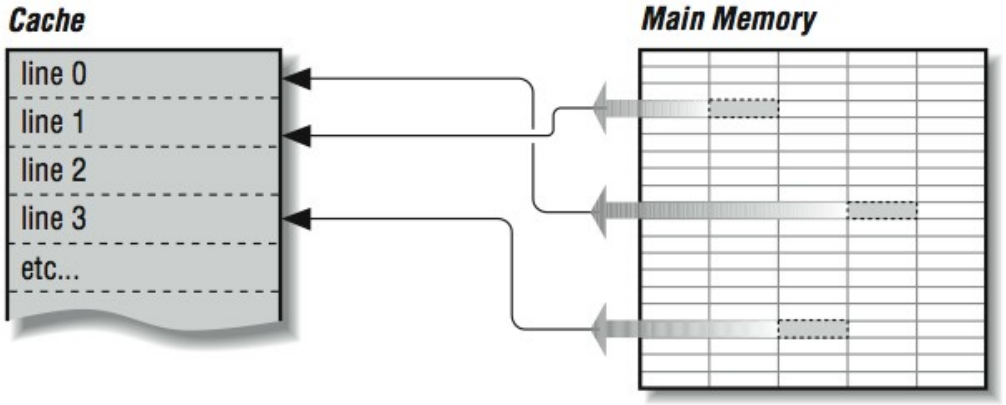
© theregister.co.uk

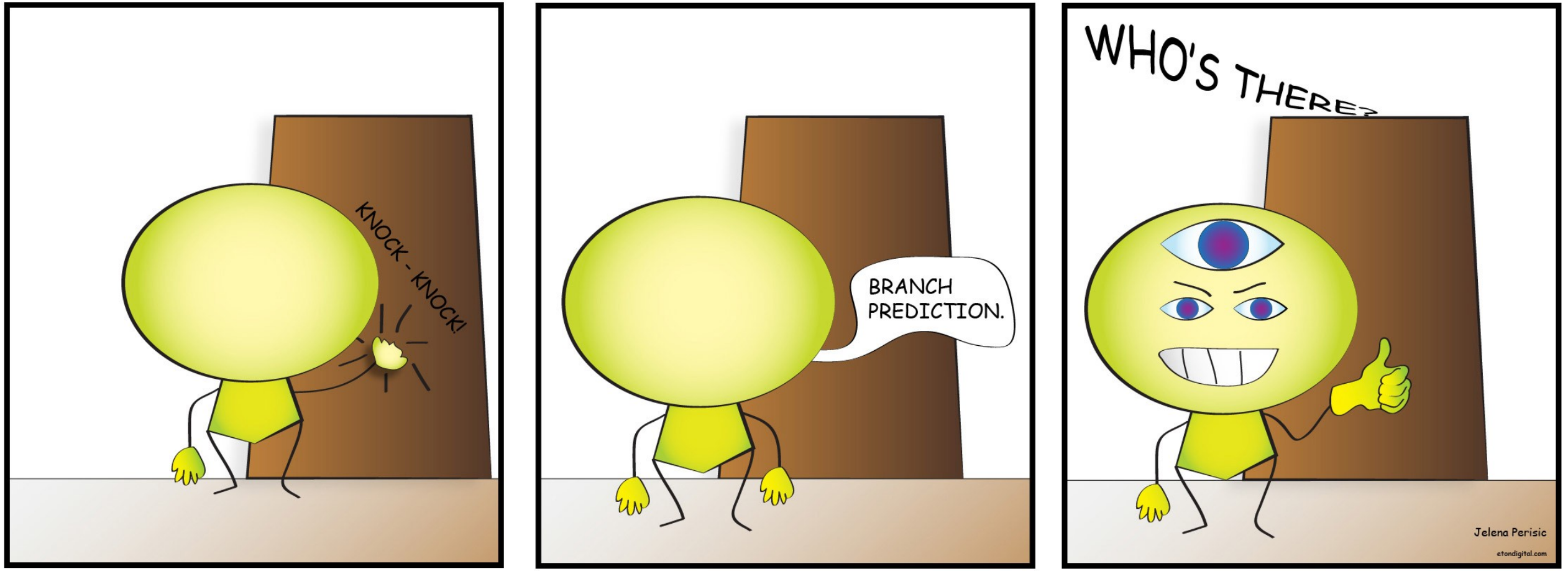
Cache overview

© cybercomputing.co.uk

Caches - the little auxiliary memory

- Cache makes use of the spatial neighborhood of data in memory
  - If one information is accessed it is likely that the next information will also be accessed
  - Always transfer cache lines/blocks between cache ↔ memory (typically 64B)
- Cache is fully transparent to the programmer, but can be flushed
- Memory accesses can be measured (cache-hit vs. cache-miss)
- Cache eviction policies choose the next cache line for eviction (LRU, LFU, FIFO)

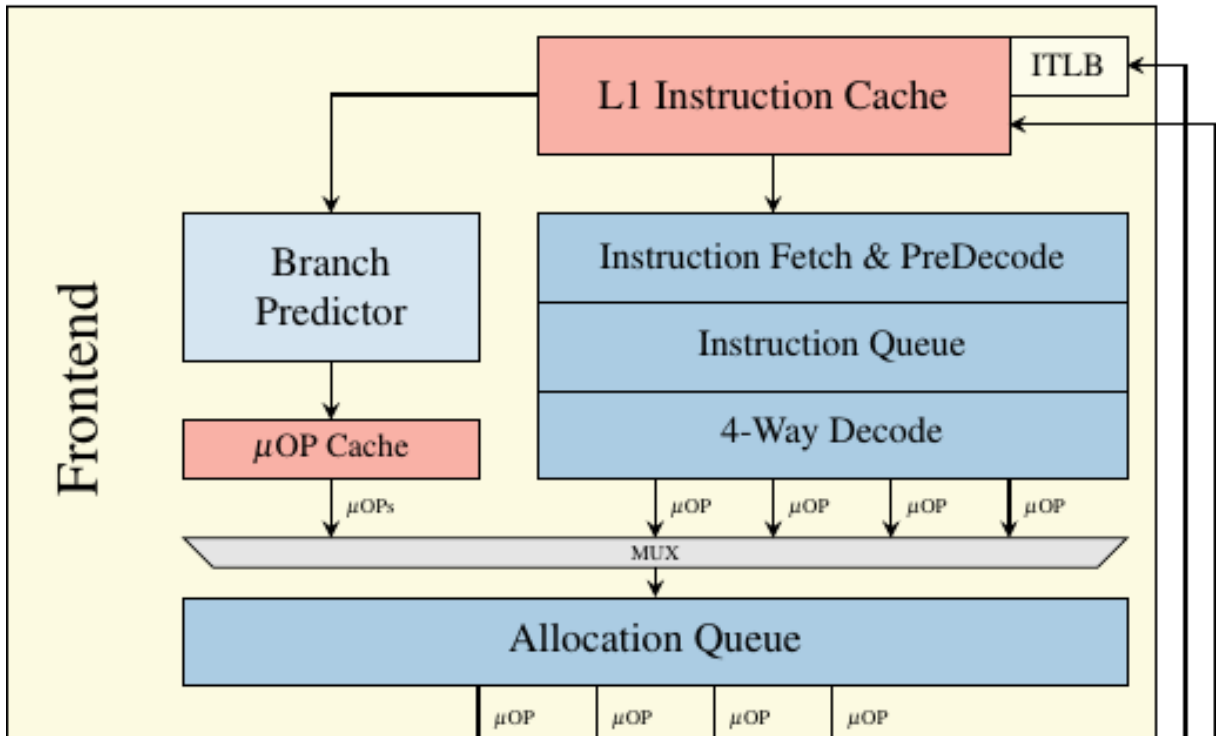




WHAT IS BRANCH PREDICTION?

What is branch prediction?

- Branch prediction is achieved by a separate hardware unit on the CPU
- The branch prediction unit evaluates the instruction stream for branches and selects (hopefully) the correct next instructions to be executed
- Example:
  - ```
    int sum = 0;
    for(int c = 0; c < size; c++) {
            sum += arr[c];
    }
    ```
  - For each iteration the branch condition needs to be evaluated
  - Branch prediction circumvents this issue by automatically predicting the next instructions



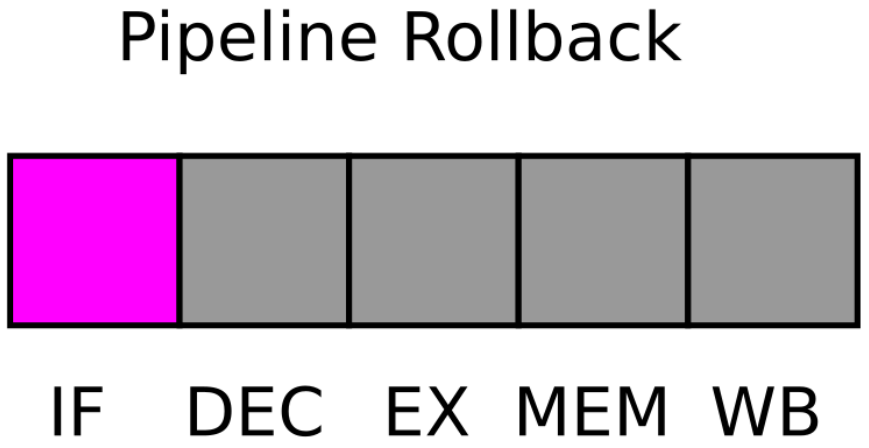
Intel Skylake Microarchitecture
© Meltdown authors [5]
    ```

Branch prediction - in depth

- Assembly:

```
        lea rdx, [rdi+4+rax*4]
        xor eax, eax
 loop:
        add eax, DWORD PTR [rdi]
        add rdi, 4
        cmp rdi, rdx
        jne loop
        rep ret
```

- Branch prediction efficiently prevents pipeline bubbles
- If branch prediction fails, draw-back is a pipeline flush ↔ same impact as no branch prediction was present



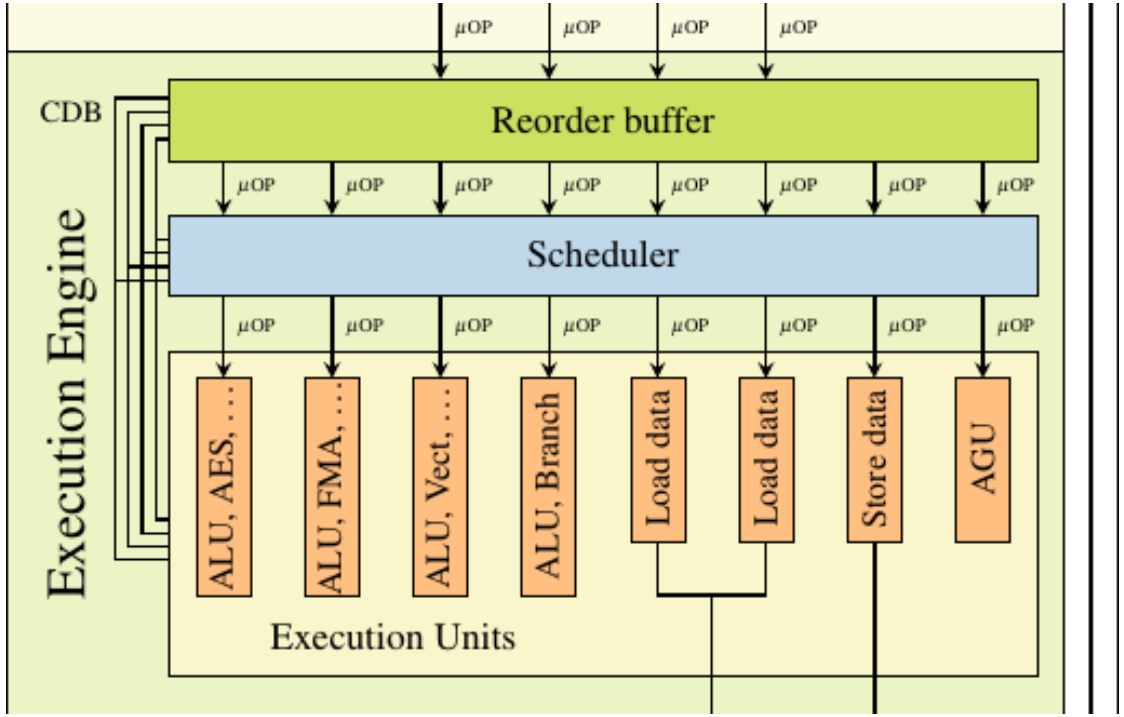
Branch prediction - in depth cont'd

- There multiple implementations of branch prediction
- Static branch prediction:
  - Simplest branch prediction implementation
  - Branch behavior is predicted at compile time (f.e. loop unroll helps for data hazards)
- Dynamic branch prediction:
  - Uses the information gathered at runtime (will be trained on taken branches)
  - If a branch is taken very often, it is likely to be taken at the next occurence as well
- For dynamic branch prediction there exist multiple (hybrid) implementations. Commonly used are multilevel branch and perceptron branch prediction in modern CPU architectures
- Speculative execution: although the evaluation of a branch is not finished yet, instructions can be executed ahead of time

OUT-OF-ORDER EXECUTION

Out-of-order execution

- Reorder independent instructions efficiently
- Achieved by additional pipeline stages
- Basic principles:
  - Decoded instructions are buffered in the reservation stations, which can be dedicated to a single or multiple functional units
  - If the operands of a instruction are ready, they can be scheduled to an appropriate functional unit
  - As soon the execution is finished the result is transmitted back to the reservation stations over the common data bus (CDB)
  - Independent instructions can be executed independently → resolves data hazards



Intel Skylake Microarchitecture
© Meltdown authors [5]

Example

- Code:

```
float do_something(float * arr, int entry, float x, float y) {
        float value;
        value = arr[entry];
        value += x*y; //fma
        return value;
}
```

- Loads from memory independent multiplication of x and y, and can therefore be executed independently.
- As soon as the loads from memory are returned, the final addition can be executed.

Rogue Data Cache Load

MELTDOWN

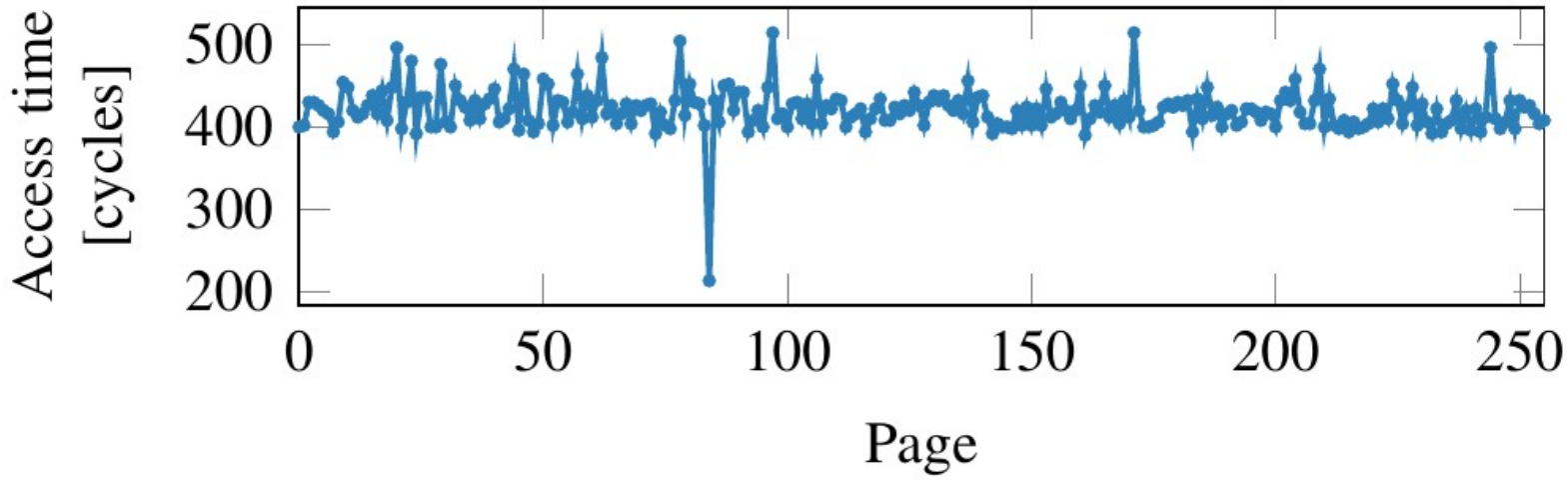
THE ATTACK

Meltdown

- Exploit out-of-order execution and pipelines: place transient instruction
- Let CPU run ahead and warm up a cache line
- Covert channel: number of the warm cache line leaks the actual secret
- Intel: Cache disregards MMU privilege levels
- Leak arbitrary memory from kernel space

Simplified Implementation

```
char mysecret = 'F';
char probe[4096 * 256];
```



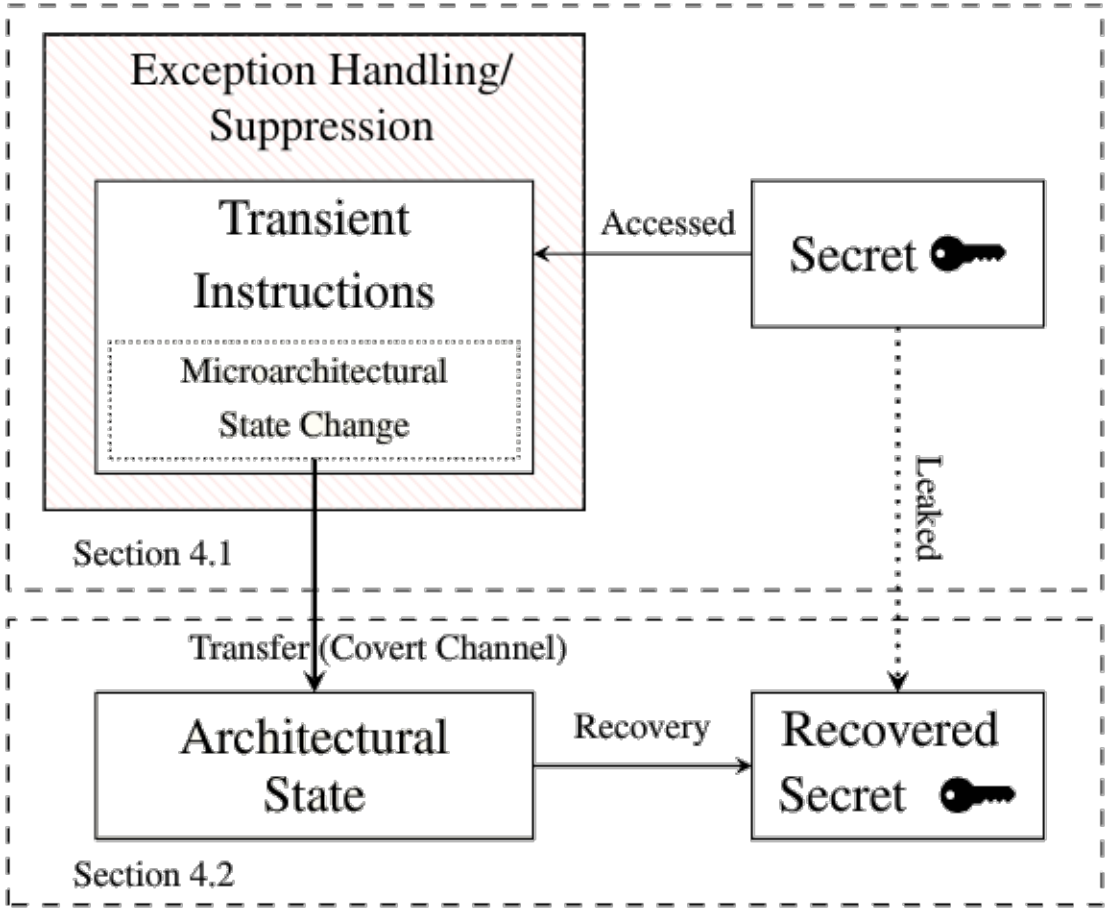

```
}
```

```
char probe_array(void) {
    int i;
    return min(histogram);
}
```

Meltdown

1. Initialise exception handling, register SIGSEGV
2. Execute transient instruction
    1. Load attacker-chosen (virtual) address into a register <- SIGSEGV
    2. Let CPU run ahead
    3. Reference address → register
    4. Index of probe array depends on result
3. Catch SIGSEGV
    1. CPU will roll back architectural state
    2. Probe access times of probe array to determine cache line (Flush+Reload)
4. The number of the warm cache line leaks the secret
5. That’s it, folks.

Rogue Data Cache Load



Meltdown attack scheme

© Meltdown authors [5]

- Microarchitectural state is rolled back when exceptions occur
- Pipeline is rolled back, transient instructed will be undone
- Intel CPUs: Architectural state can be recovered: no rollback of cache's state
- The number of the warm cache line (1B 0-255) is the secret byte in kernel memory

Implications

- Ok, so we can and do leak kernel's memory... Bloody hell!
- **Q:** But how to leak arbitrary physical memory?
- **A:** Direct Physical Mapping. [5]

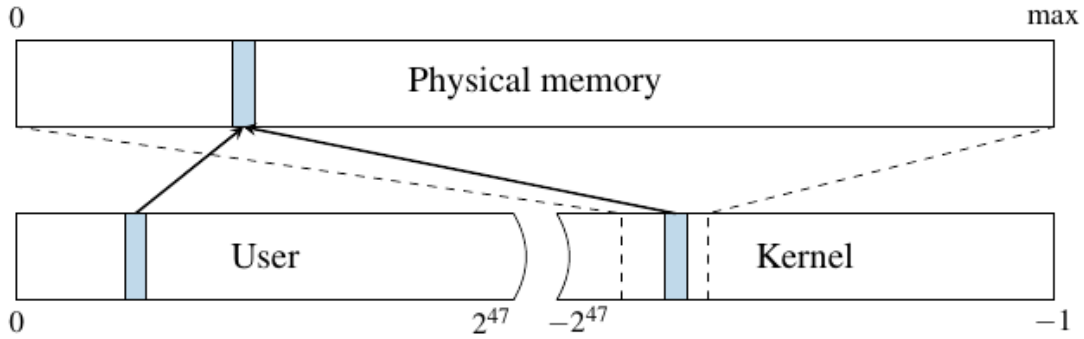


Figure 2: The physical memory is directly mapped in the kernel at a certain offset. A physical address (blue) which is mapped accessible for the user space is also mapped in the kernel space through the direct mapping.

© Meltdown authors [5]

MITIGATION

FUCKWIT, aka KAISER, aka KPTI

Implementation

- `FUCKWIT:`

  Forcefully **U**nmap **C**omplete **K**ernel **W**ith **I**nterrupt **T**rampolines

- `KAISER:`

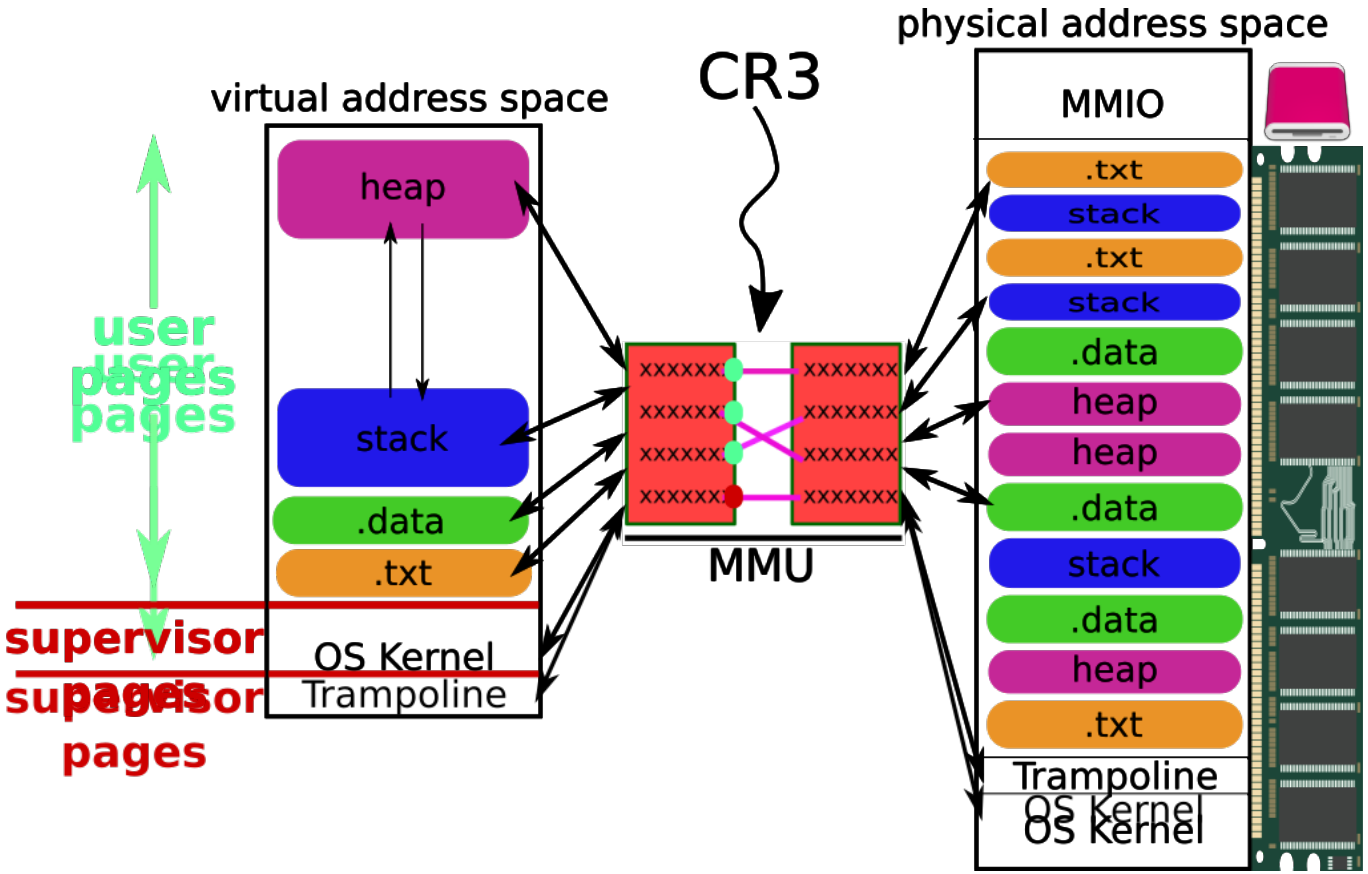
  Kernel **A**ddress **I**solation to have **S**ide-channels **E**fficiently **R**emoved

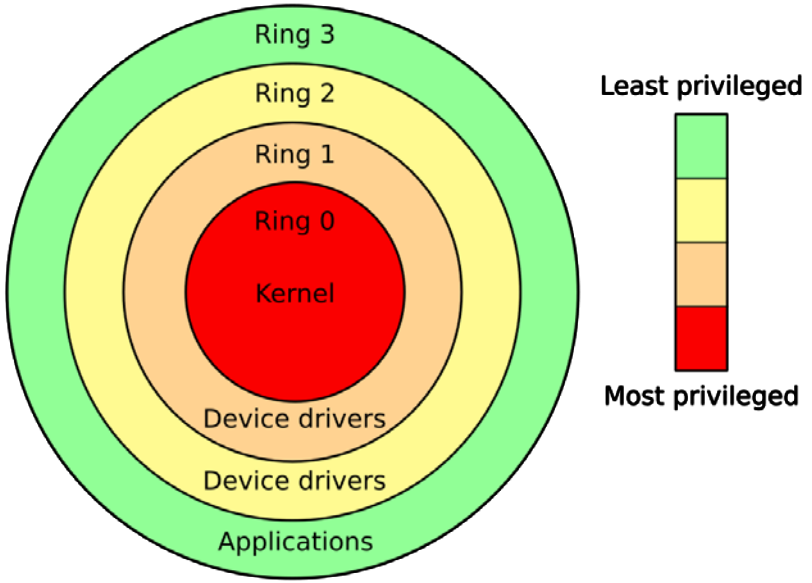
- `KPTI:`

  Kernel **P**age **T**able **I**solation

- Mitigation
- Shadow address space
- Two tables per process: user table and kernel table
- Minimize kernel footprint in userspace
  - Trampoline code switches to kernel's page table
  - Only Kernel's page table maps kernel pages
  - Return to userspace: change to user's table

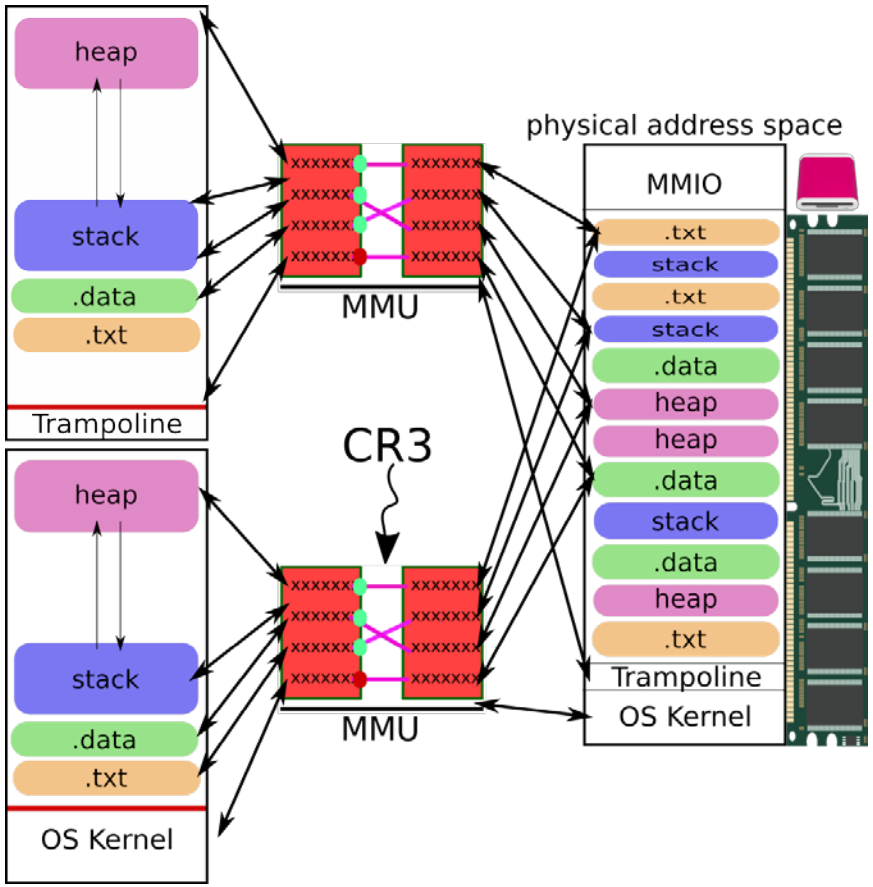
KPTI: Kernel space



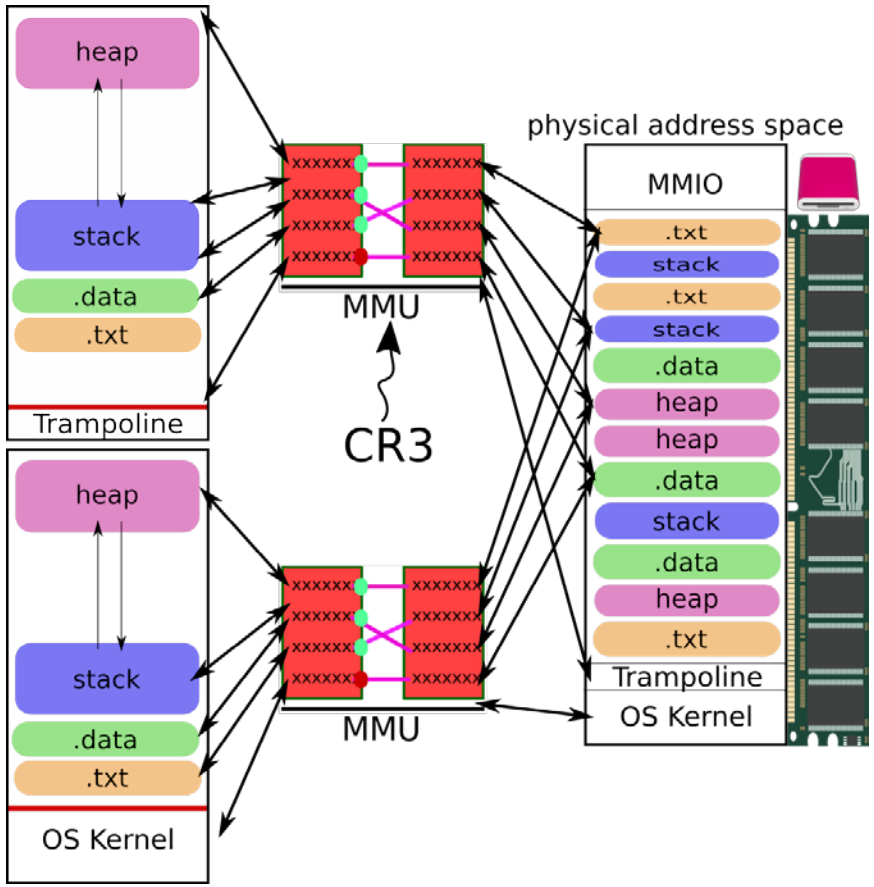


Two modes

KPTI: kernel space



KPTI: user space



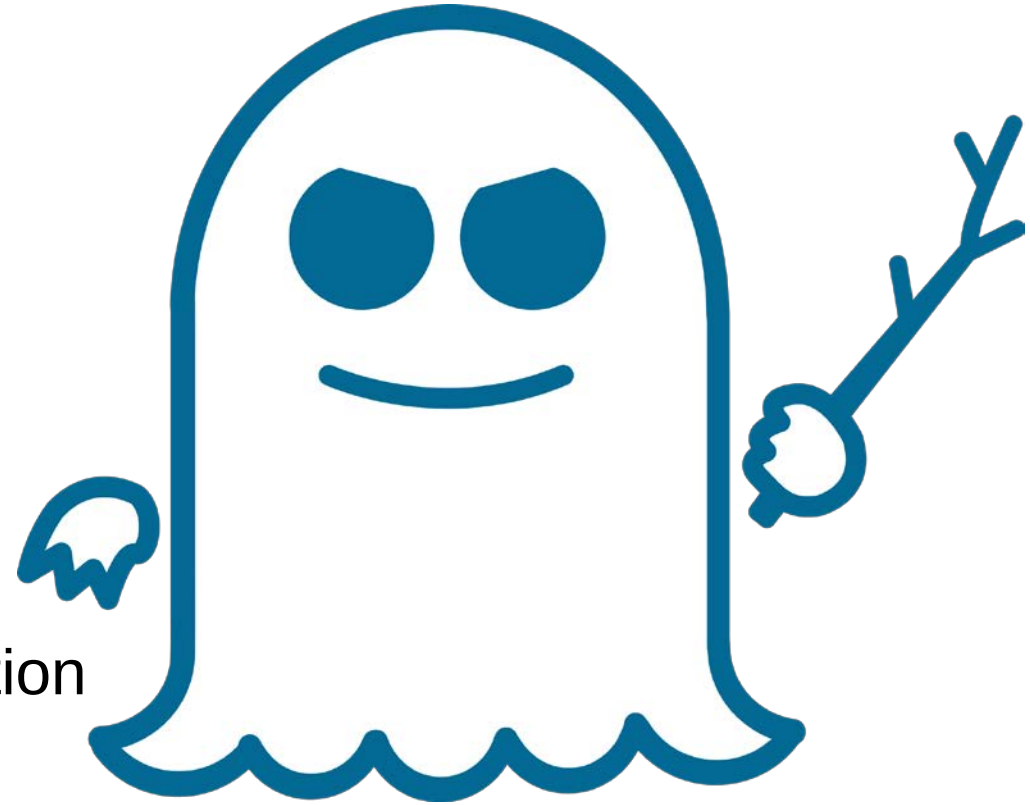
How much is KPTI?

- Switch of page tables is expensive (TLB flush)
- Performance. Overhead?
- According to...
  - authors: 0.25%
  - Linux devs: 5%
  - some scenarios: 30%+ [6]
- Systems with high IO load
- KPTI is only enabled on affected CPUs

Wait, there’s still another bug...

Bounds Check Bypass & Branch Target Injection

SPECTRE [7]



Spectre

- Exploit speculative dynamic branch prediction and pipelining: place transient instruction
- Train the speculative branch prediction to access arbitrary code
- Static branch prediction is not affected, because it cannot be trained
- Covert channel: number of the warm cache line leaks the actual secret
- Spectre variants:
  - v1: bounds check bypass
  - v2: branch target injection

v1: bounds check bypass

- CPU mispredicts a branch and speculatively executes code
- Train the branch prediction that offset will be likely within bounds
- Attack: set offset out of bounds
- Boundary check variables need to be uncached(!) → load from memory
- Data of `val` will be leaked, because the `arr2` is cached
- Finally `arr1->length` arrives from memory and an incorrect branch was detected

```
if (offset < arr1->length) {
    char val = arr1->data[offset]
    long idx = ((val&1)*0x100) + 0x200;
    if (idx < arr2->length)
        char val2 = arr2->data[idx];
}
```

v2: branch target injection

- Indirect branches are unconditional jumps based on an address: `jmp eax`
- Branch prediction for indirect jumps uses cached results in a separate buffer
- Attacker exploits similiar jump addresses by training invalid addresses
- Victim will jump to an invalid location
  - → CPU will detect invalid jump
  - → CPU will restore its previous state
  - BUT: speculative execution leaves traces in the memory cache

Mitigation

- Prediction cannot be disabled
- Speculation barriers (injected by compiler)
  - Cache flushes
  - Execution serialization: memory barriers (fences)
  - CPU Microcode Patching
  - Intel:
    - IBRS Indirect Branch Restricted Speculation: Marketing-speak for "we flipped a chicken bit", presumably.
    - IBPB Indirect Branch Predictor Barrier instruction prevents leakage of indirect branch predictor state across contexts
    - STIBP Single Thread Indirect Branch Predictors isolates branch prediction state between two hyperthreads
  - AMD: "near zero risk" for branch target injection, but for bounds check bypass
  - Disable indirect branch prediction entirely by using an alternative instruction sequence: **Retpoline**

Retpoline I

- **Ret**urn tram**poline**
- Replaces JMPs/CALLs with unspeculative JMPs/CALLs
- Quite hacky
- Performance horror!
- Requires recompiling!

```
...
xorq  %rax, %rax

mov 0x0(%rip),%rdi mov
    $0x1,%edx

mov $0x3,%esi

jmpq  21c0
```

Retpoline II

- `call` pushes the current instruction pointer
- `lea` resets the stack pointer to TARGET which is read from the stack
- `ret` jumps to `*TARGET` and resets the stack pointer to the beginning of the call stack, reversing the hack from before

```
push TARGET
call retpoline_call_target
2:
lfence /* stop speculation */
jmp 2b
retpoline_call_target:
lea 8(%rsp), %rsp
ret
```

Impact

- Still possible to run vulnerable code
- RETPOLINE is a hack
- Massive performance impacts (>50%)
- Alan Cox: “... your performance will resemble that of a 2012 atom processor at best”

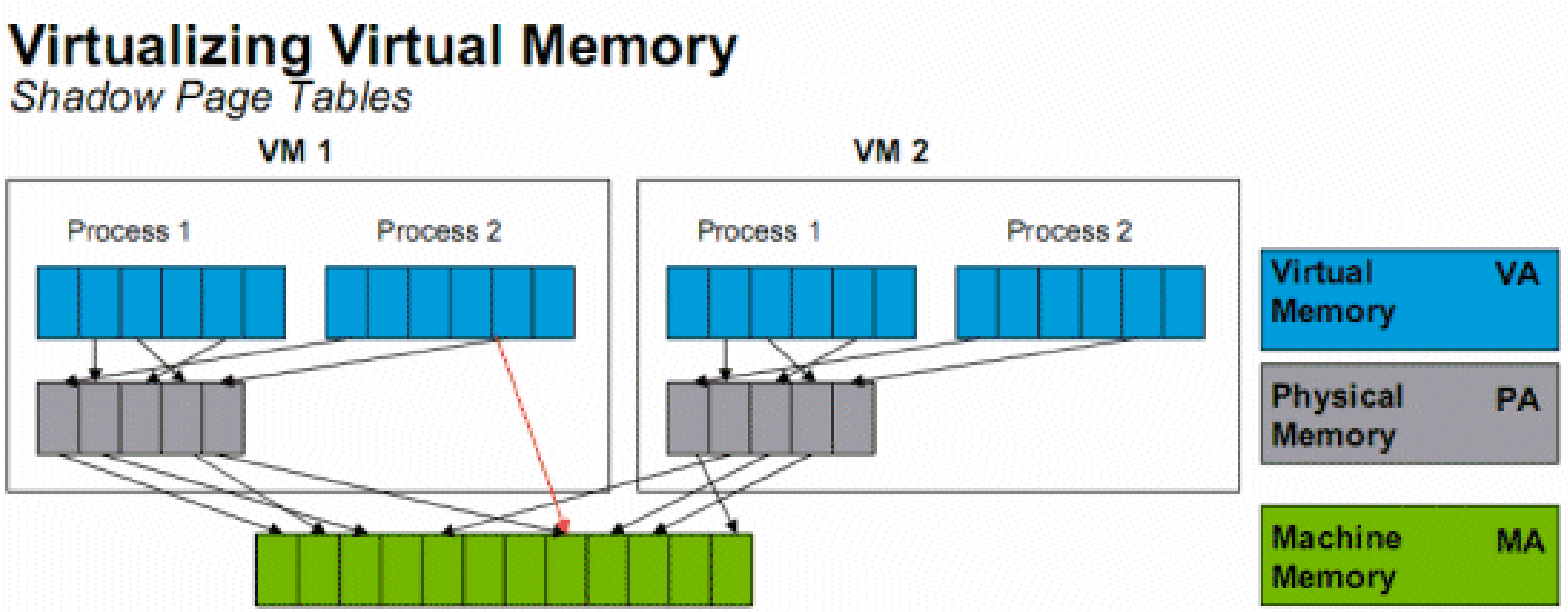
WERDEN WIR ALLE STERBEN?

Meltdown

- Only Intel CPUs are affected
- Fixable. Patches are available! DO PATCH!
- Malign applications show benign behaviour (AV)
- Libraries are available: libkdump
- Exploit requires to execute code (JavaScript!)
- Imaginable attack scenarios:
  - Browser’s JavaScript leaks content of another tab
  - Browser’s JavaScript leaks content of any other application
  - Android’s harmful app leaks other app’s content
- Virtualisation Technologies? Can a VM leak memory of its neighbour?

Meltdown with VMs

- Cross-VM
  - Two stage address translation
  - NOT affected by Meltdown: Xen, QEMU/KVM/, VMware, VirtualBox, Jailhouse ...
  - ONLY guests are affected!



Spectre v1 (bounds check bypass)

- Sandbox escapes (same context leak)
  - Easy
  - JITs/interpreters
  - Shared memory/threads
  - Browser's JavaScript leaks content from another tab
- Same-CPU cross-process
  - Medium
  - Attacker triggers vulnerable code in the vulnerable process
  - Get a signal from the cache directly (e.g. by timing accesses to memory which has colliding cache tags on the same CPU core or sharing a level of cache)
  - Includes attacks on the kernel and on hypervisors
- Remote
  - Hard
  - Remotely trigger vulnerable code
  - Get timing signal back from the relevant cache lines
  - Not practical

Spectre v2 (branch target injection)

- Sandbox escapes (same context leak)
  - Tricky
  - JIT
  - Maybe with careful instruction massaging?
- Same-CPU cross-process
  - Feasible
  - Includes attacks on the kernel/hypervisor (if hypervisor in involved)
- Remote: not possible

What‘s next?

New Exploits

- JavaScript Malware
- More covert channels / side channel attacks
- SMT technologies [8]
  - Known as ’Intel Hyperthreading’
  - share CPUs
  - share Caches, ...
  - share ALUs
- Yet more TSX exploits

How to solve?

- Reduce CPU complexity!
- More features → more bugs → more side

Thank you for your attention!

Credits

- Ralf Ramsauer & Joachim Weber for their slides to their talk at OTH Regensburg
- Alexandra Dmitrienko for her lecture slides

Sources

1) Olin Sibert, Phillip A Porras, and Robert Lindell. ‘The Intel 80x86 processor architecture: pitfalls for secure systems’. In: Security and Privacy, 1995. Proceedings., 1995 IEEE Symposium on. IEEE. 1995, pp. 211–222.

2) Clémentine Maurice and Stefan Mangard. ‘KASLR is Dead: Long Live KASLR’. In: Engineering Secure Software and Systems: 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings. Vol. 10379. Springer. 2017, p. 161.

3) Jonathan Corbet. The current state of kernel page-table isolation. Oct. 2017. URL: https://lwn.net/Articles/741878/.

4) Jonathan Corbet. Notes from the Intelpocalypse. Jan. 2018. URL: https://lwn.net/Articles/738975/.

5) Moritz Lipp, Michael Schwarz, Daniel Gruss, et al. ‘Meltdown’. In: ArXiv e-prints (Jan. 2018). arXiv: 1801.01207.

5) Jonathan Corbet. KAISER: hiding the kernel from user space. Nov. 2017. URL: https://lwn.net/Articles/738975/.

6) Paul Kocher, Daniel Genkin, Daniel Gruss, et al. ‘Spectre Attacks: Exploiting Speculative Execution’. In: ArXiv e-prints (Jan. 2018). arXiv: 1801.01203.

7) Sophia M D’Antoine. ‘Exploiting processor side channels to enable cross VM malicious code execution’. PhD thesis. Rensselaer Polytechnic Institute, 2015.