ITMO Advanced OS 2024

Aleksei Romanovskii, PhD,
SPb Research Center (CBG OS Lab)

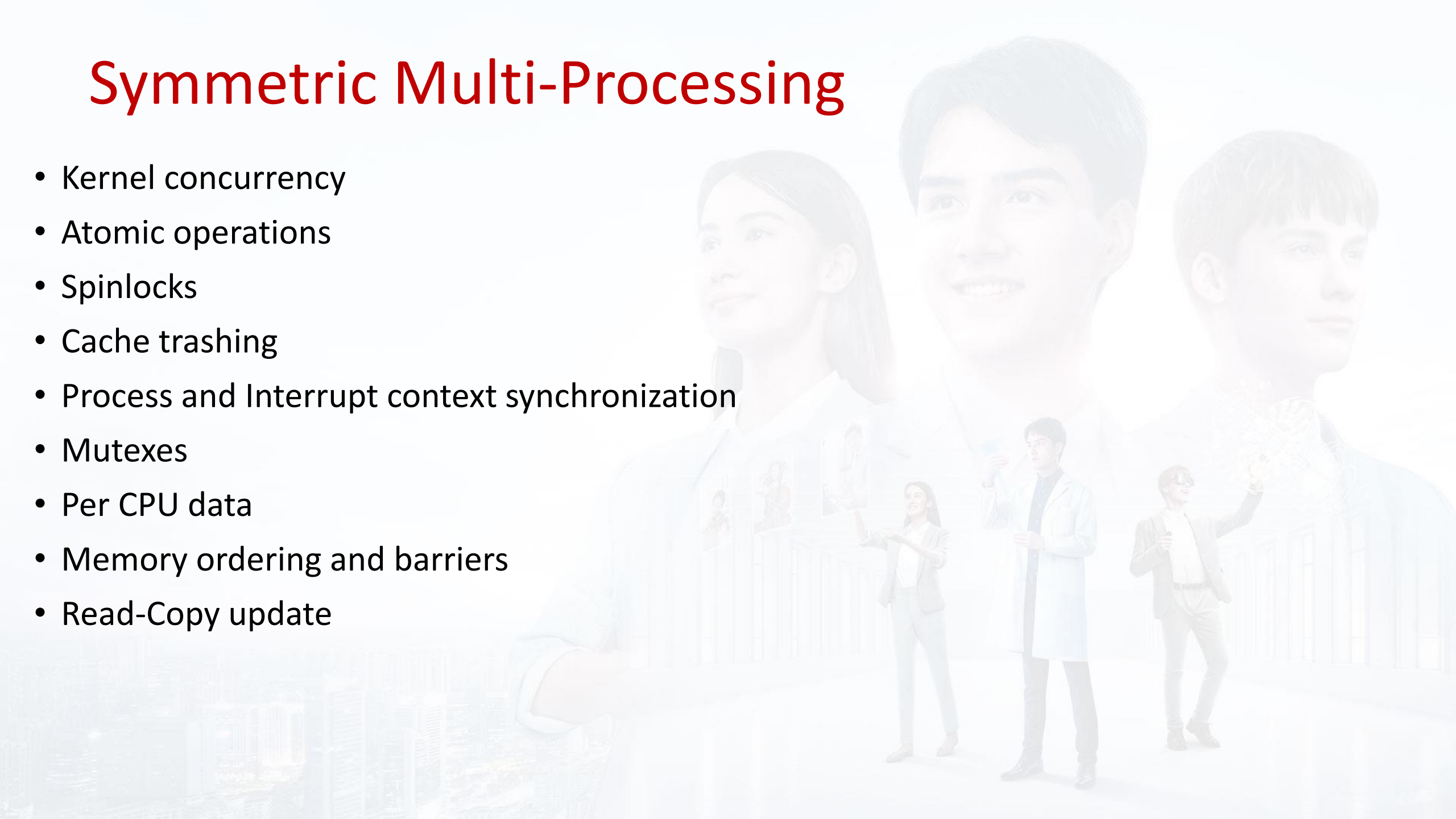
Lesson 2024.10.02

Contents

- Processes and threads
- Interrupts and exceptions
- Symmetric Multi-Processing

Symmetric Multi-Processing

- Kernel concurrency
- Atomic operations
- Spinlocks
- Cache trashing
- Process and Interrupt context synchronization
- Mutexes
- Per CPU data
- Memory ordering and barriers
- Read-Copy update

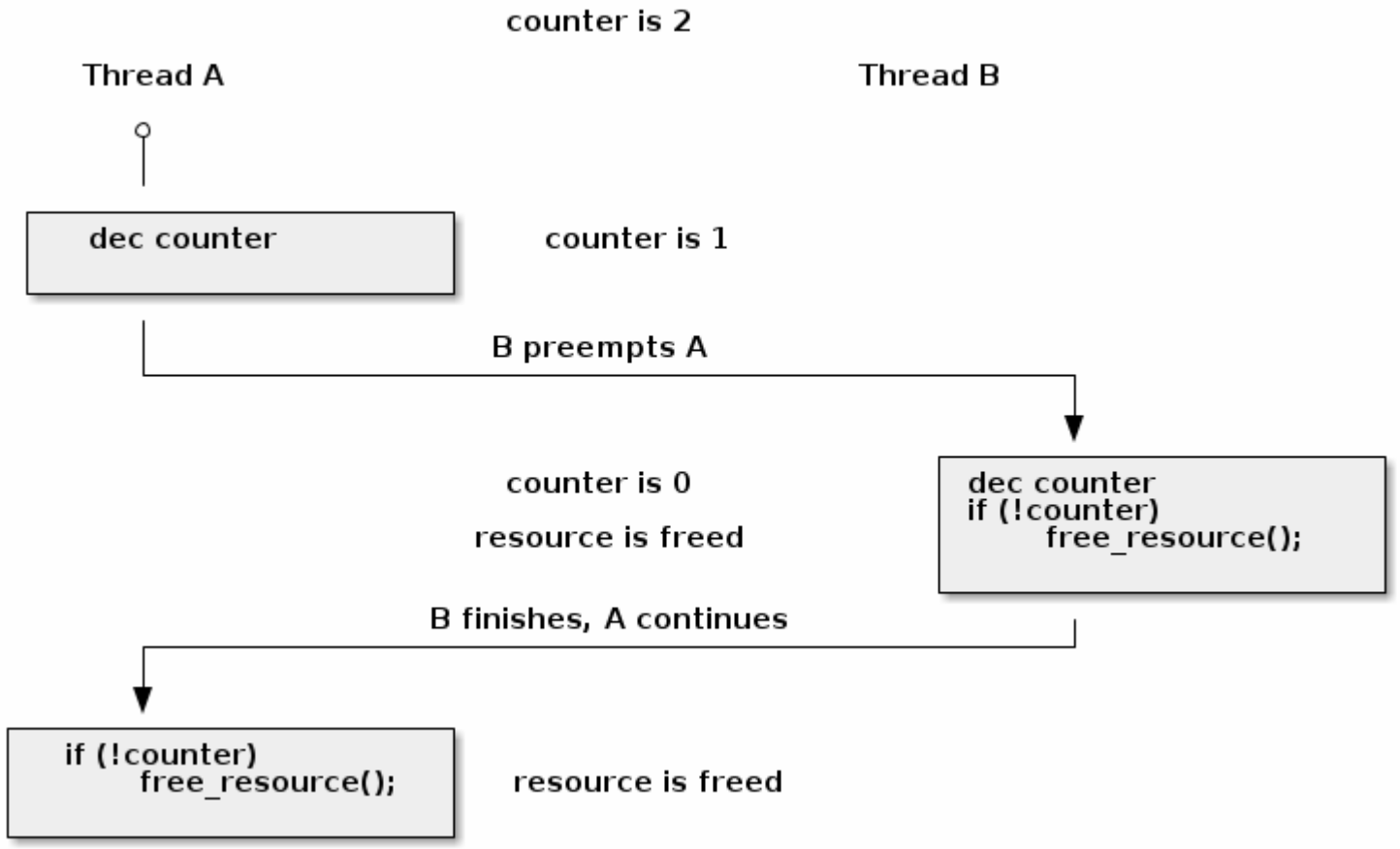


Synchronization basics

- Linux kernel supports symmetric multi-processing (SMP) and it uses a set of synchronization mechanisms to achieve predictable results, free of race conditions.
- Race conditions can occur when the following two conditions happen simultaneously:
  - there are at least two execution contexts that run in "parallel":
    - truly run in parallel (e.g. two system calls running on different processors)
    - one of the contexts can arbitrary preempt the other (e.g. an interrupt preempts a system call)
  - the execution contexts perform read-write accesses to shared memory
- Race conditions can lead to erroneous results that are hard to debug, because they manifest only when the execution contexts are scheduled on the CPU cores in a very specific order.

Classic race condition example:

```
void release_resource() { counter--; if (!counter) free_resource(); }
```



How to avoid race conditions

- Identify the critical section that can generate a race condition. The critical section is the part of the code that reads and writes shared memory from multiple parallel contexts.
- In the example above, the minimal critical section is starting with the counter decrement and ending with checking the counter's value.
- Once the critical section has been identified race conditions can be avoided by using one of the following approaches:
- make the critical section **atomic** (e.g. use atomic instructions)
- **disable preemption** during the critical section (e.g. disable interrupts, bottom-half handlers, or thread preemption)
- **serialize the access** to the critical section (e.g. use spin locks or mutexes to allow only one context or thread in the critical section)

Linux kernel concurrency sources

- **single core systems**, **non-preemptive kernel**: the current process can be preempted by interrupts
- **single core systems**, **preemptive kernel**: above + the current process can be preempted by other processes
- **multi-core systems**: above + the current process can run in parallel with another process or with an interrupt running on another processor

Atomic operations

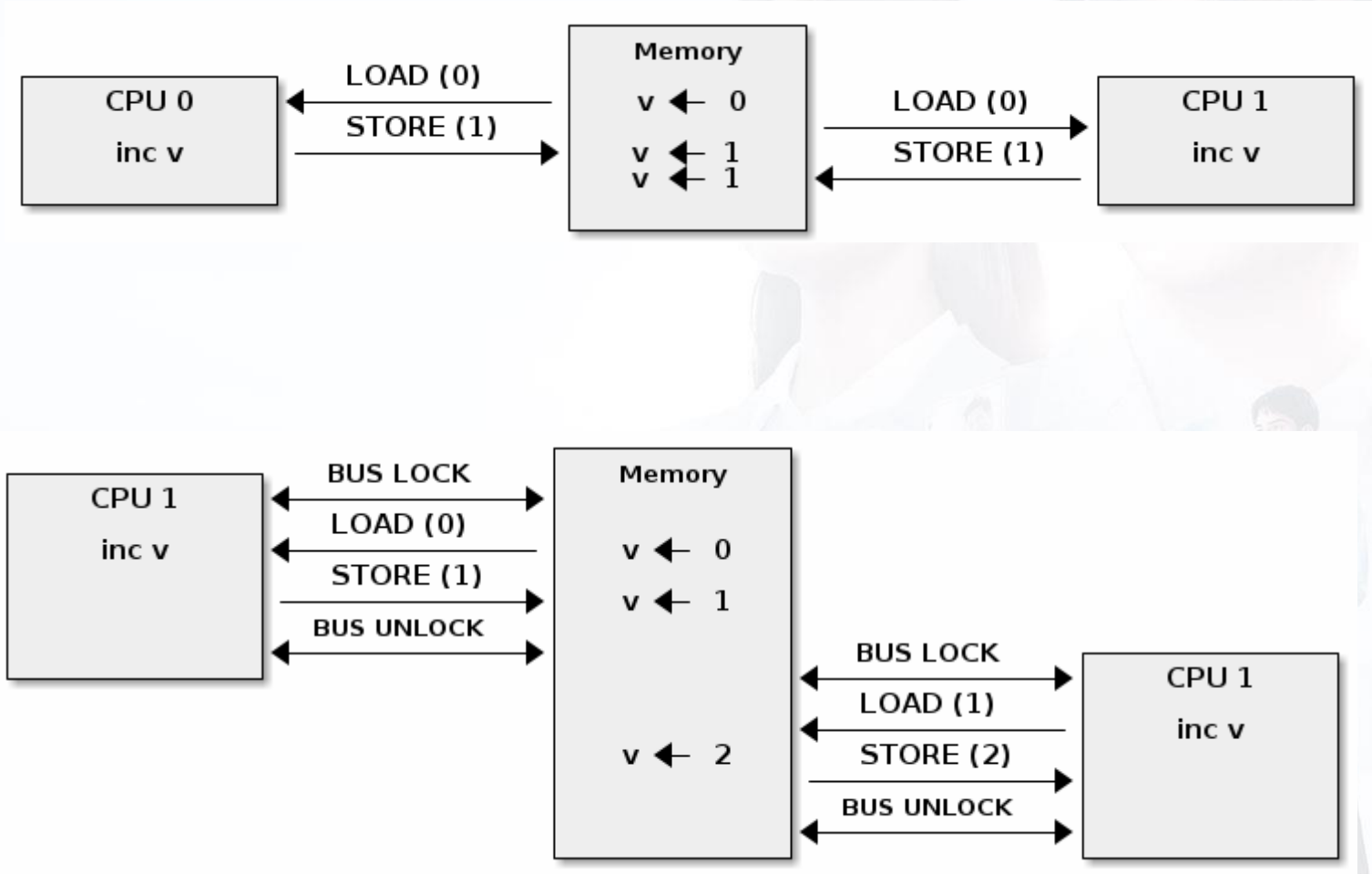
- integer based:
  - simple: **atomic_inc()**, **atomic_dec()**, **atomic_add()**, **atomic_sub()**
  - conditional: **atomic_dec_and_test()**, **atomic_sub_and_test()**
- bit based:
  - simple: **test_bit()**, **set_bit()**, **change_bit()**
  - conditional: **test_and_set_bit()**, **test_and_clear_bit()**, **test_and_change_bit()**

Using atomic_dec_and_test()

```
void release_resource()
{
   if (atomic_dec_and_test(&counter))
    free_resource();
}
```

- But… in multi-core systems an atomic operation is not longer atomic at the system level (but still atomic at the single core level)

Atomic_ may be not atomic in SMP

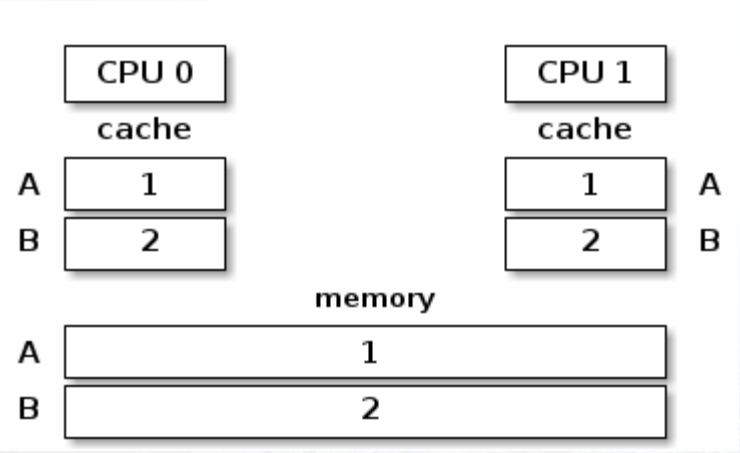


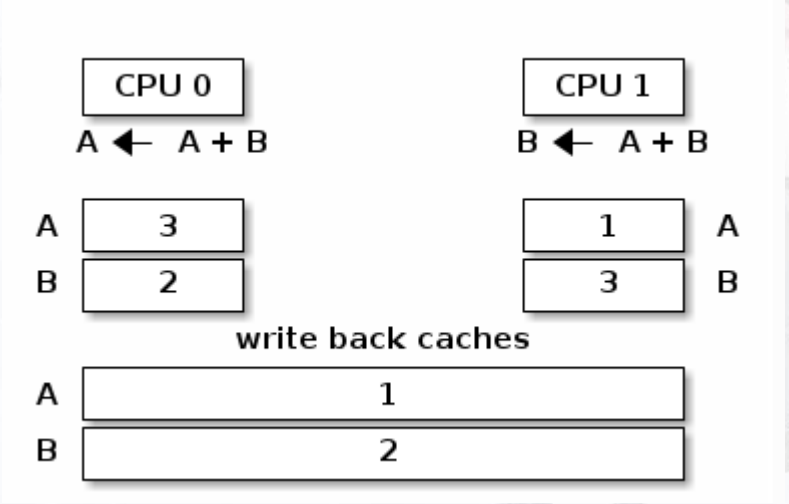
Spin locks

- Spin locks are used to serialize access to a critical section
  - spin_lock
  - spin_unlock
  - it takes CPU by “spinning “ within two instructions
- While the spin lock avoids race conditions, it can have a significant impact on the system's performance due to "lock contention":
- There is lock contention when at least one core spins trying to enter the critical section lock
- Lock contention grows with the critical section size, time spent in the critical section and the number of cores in the system
- Another negative side effect of spin locks is cache thrashing.
- Cache thrashing occurs when multiple cores are trying to read and write to the same memory resulting in excessive cache misses.
- Since spin locks continuously access memory during lock contention, cache thrashing is a common occurrence due to the way cache coherency is implemented.

Cache coherency in multi-processor systems

- Let us consider memory hierarchy in multi-processor systems composed of local CPU caches (L1, l2 caches), shared CPU caches (LLC caches) and the main memory. To explain cache coherency we will ignore the L2, LLC caches and only consider the L1 caches and main memory.
- Below two variables A and B fall into different cache lines and no cache coherence is assumed:





Cache coherency protocols

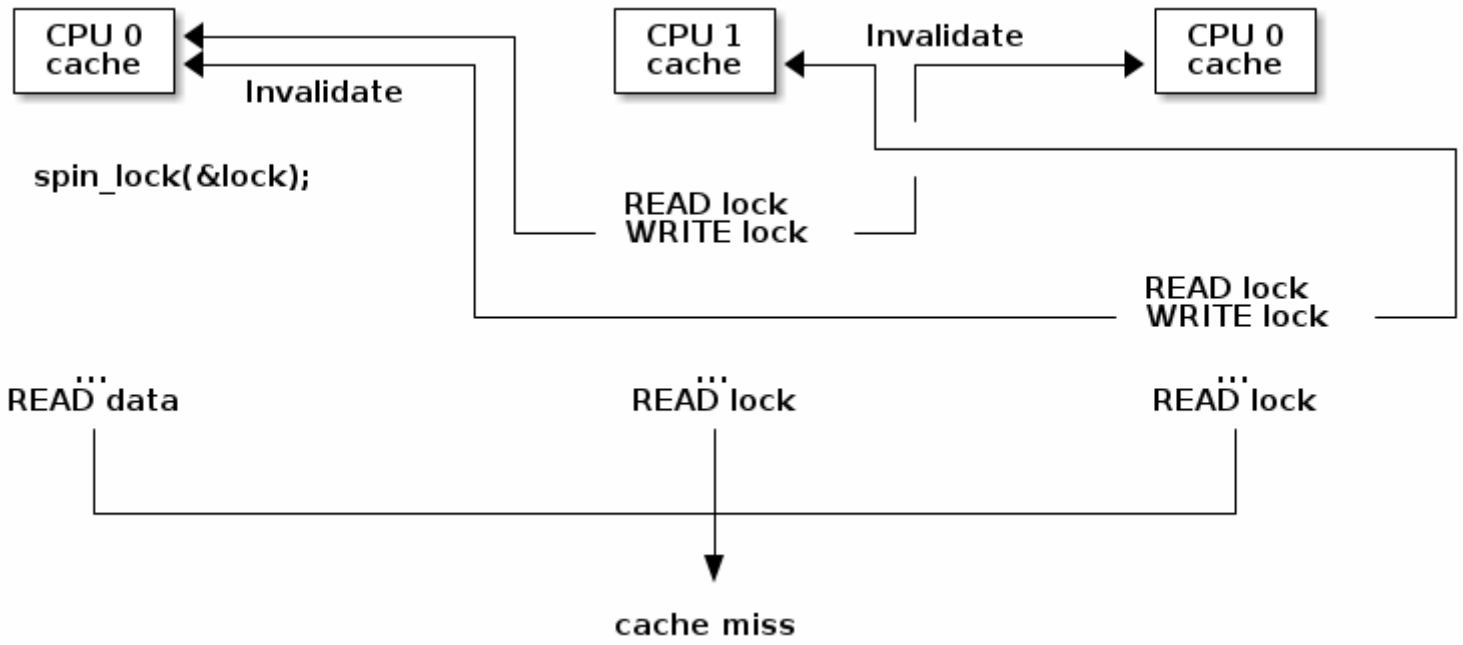
- Multi-processor systems use cache coherency protocols. There are two main types of cache coherency protocols:
- Bus snooping (sniffing) based: memory bus transactions are monitored by caches and they take actions to preserve coherency
- Directory based: there is a separate entity (directory) that maintains the state of caches; caches interact with directory to preserve coherency
- Bus snooping is simpler but it performs poorly when the number of cores goes beyond 32-64.
- Directory based cache coherence protocols scale much better (up to thousands of cores) and are usually used in NUMA systems.

MESI cache coherency protocol

- Named by cache line state names: **Modified**, **Exclusive**, **Shared** and **Invalid**
  - Modified: owned by a single core and dirty
  - Exclusive: owned by a single core and clean
  - Shared: shared between multiple cores and clean
  - Invalid : the line is not cached
- Caching policy: write back
- Issuing read or write requests from CPU cores will trigger state transitions:
  - Invalid -> Exclusive: read request, all other cores have the line in Invalid; line loaded from memory
  - Invalid -> Shared: read request, at least one core has the line in Shared or Exclusive; line loaded from sibling cache
  - Invalid/Shared/Exclusive -> Modified: write request; **all other** cores **invalidate** the line
  - Modified -> Invalid: write request from other core; line is flushed to memory

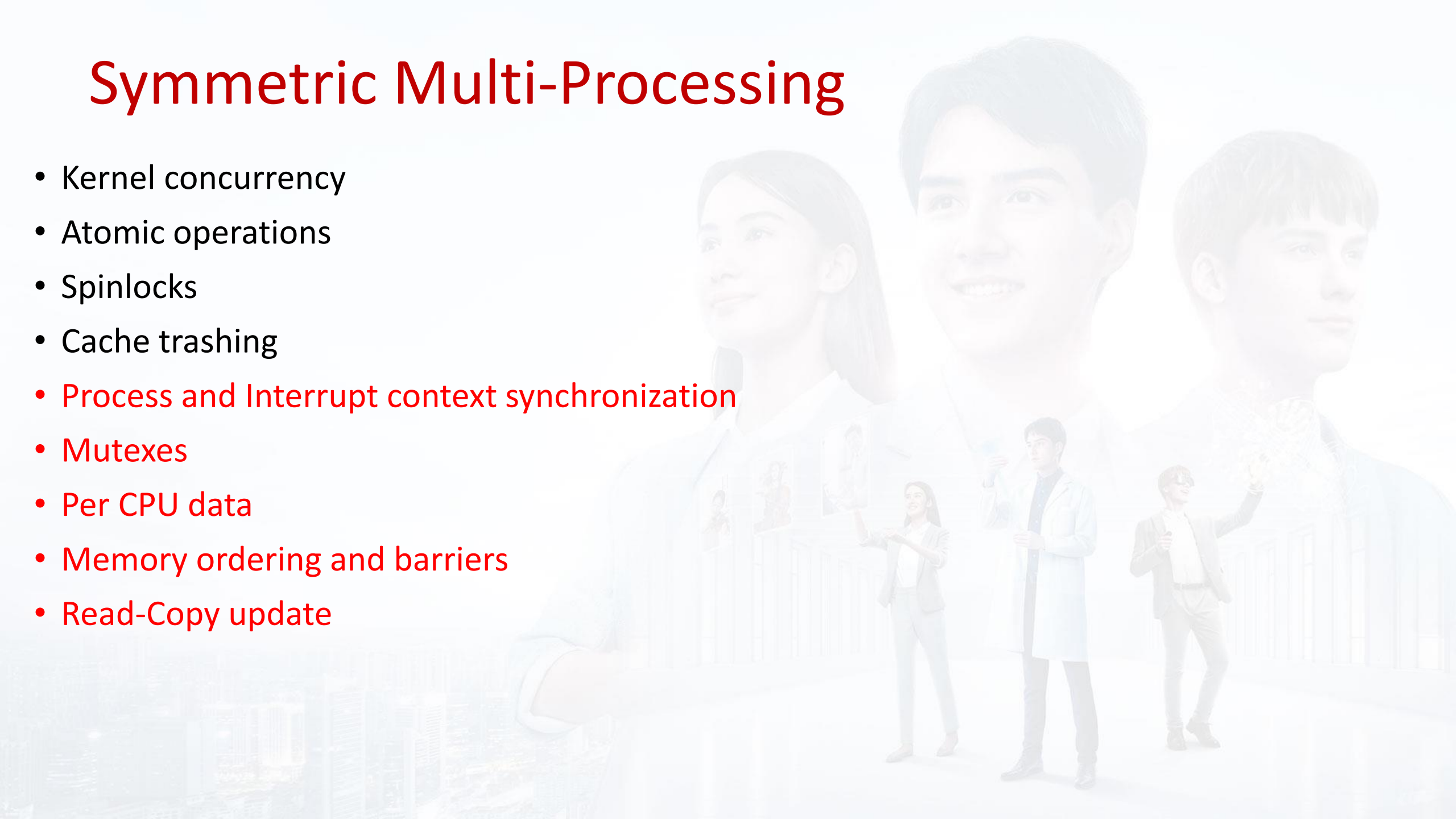
Cache trashing

- lets consider a system with three CPU cores, where the first has acquired the spin lock and it is running the critical section while the other two are spinning waiting to enter the critical section:



Symmetric Multi-Processing

- Kernel concurrency
- Atomic operations
- Spinlocks
- Cache trashing
- Process and Interrupt context synchronization
- Mutexes
- Per CPU data
- Memory ordering and barriers
- Read-Copy update



Process and Interrupt Context Sync 1 of 4

- Accessing shared data from both process and interrupt context is a relatively common scenario.
- On single core systems we can do this by disabling interrupts
- That won't work on multi-core systems - we can have the process running on one CPU core and the interrupt context running on a different CPU core.
- Using a spin lock can cause common deadlock conditions:
  - In the process context we take the spin lock
  - An interrupt occurs and it is scheduled on the same CPU core
  - The interrupt handler runs and tries to take the spin lock
  - The current CPU will deadlock

Process and Interrupt Context Sync 2 of 4

- To avoid this issue:
  - In process context:
    - disable interrupts and acquire a spin lock; this will protect both against interrupt or other CPU cores race conditions
    - spin_lock_irqsave() and spin_lock_restore() combine the two operations
  - In interrupt context:
    - take a spin lock; this will will protect against race conditions with other interrupt handlers or process context running on different processors

Process and Interrupt Context Sync 3 of 4

- The same issue for other interrupt context handlers such as softirqs, tasklets or timers (aka bottom-half)
- It is recommended to use dedicated APIs:
  - In process context use: spin_lock_bh() (which combines local_bh_disable() and spin_lock()) and spin_unlock_bh() (which combines spin_unlock() and local_bh_enable())
  - In bottom half context use: spin_lock() and spin_unlock() (or spin_lock_irqsave() and spin_lock_irqrestore() if sharing data with interrupt handlers)

Process and Interrupt Context Sync 4 of 4

- As mentioned before, another source of concurrency in the Linux kernel can be other processes, due to preemption.
- Preemption is configurable: when active it provides better latency and response time, while when deactivated it provides better throughput.
- Preemption is disabled by spin locks and mutexes but it can be manually disabled as well (by core kernel code).

Mutexes

- Mutexes are used to protect against race conditions from other CPU cores but they can only be used in process context.
- With mutexes (as opposed to spin locks) while a thread is waiting to enter the critical section it will not use CPU time, instead it will be added to a waiting queue until the critical section is vacated.
- Since mutexes and spin locks usage intersect, it is useful to compare the two:
  - Mutexes don't "waste" CPU cycles; system throughput is better than spin locks if context switch overhead is lower than medium spinning time
  - Mutexes can't be used in interrupt context
  - Mutexes have a higher latency than spin locks

Per-CPU data

- Per CPU data avoids race conditions by avoiding to use shared data.
- Instead, an array sized to the maximum possible CPU cores is used and each core will use its own array entry to read and write data.
- Advantages:
  - No need to synchronize to access the data
  - No contention, no performance impact
  - Well suited for distributed processing where aggregation is only seldom necessary (e.g. statistics counters)

Memory Ordering and Barriers

- Modern processors and compilers employ out-of-order execution to improve performance.
- For example, processors can execute "future" instructions while waiting for current instruction data to be fetched from memory.
- When executing instructions out of order the processor makes sure that data dependency is observed:
  - it won't execute instructions whose input depend on the output of a previous instruction that has not been execute.
- Here is an example of out of order compiler generated code:

C code	Compiler generated code

`a = 1;`
`b = 2;`
	`MOV R10, 1`
	`MOV R11, 2`
	`STORE R11, b`
	`STORE R10, a`

Barriers to order memory operations

- A read barrier (rmb(), smp_rmb()) is used to make sure that no read operation crosses the barrier;
  - all read operation before the barrier are complete before executing the first instruction after the barrier
- A write barrier (wmb(), smp_wmb()) is used to make sure that no write operation crosses the barrier
- A rw barrier (mb(), smp_mb()) is used to make sure that no write or read operation crosses the barrier

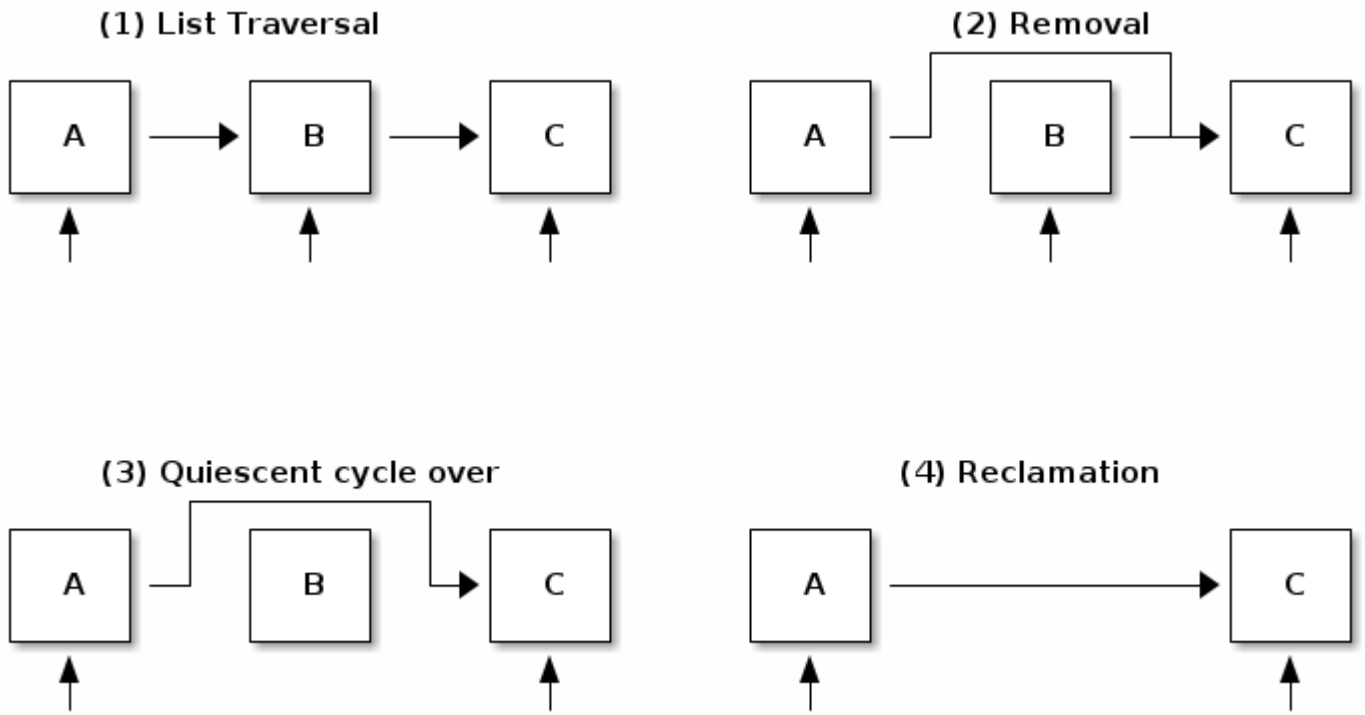
Read-Copy-Update (RCU)

- RCU is a special synchronization mechanism with significant improvements over read-write locks (and some limitations):
  - Read-only lock-less access at the same time with write access
  - Write accesses still requires locks in order to avoid races between writers
  - Requires unidirectional traversal by readers
- In fact, the read-write locks in the Linux kernel have been deprecated and then removed, in favor of RCU.

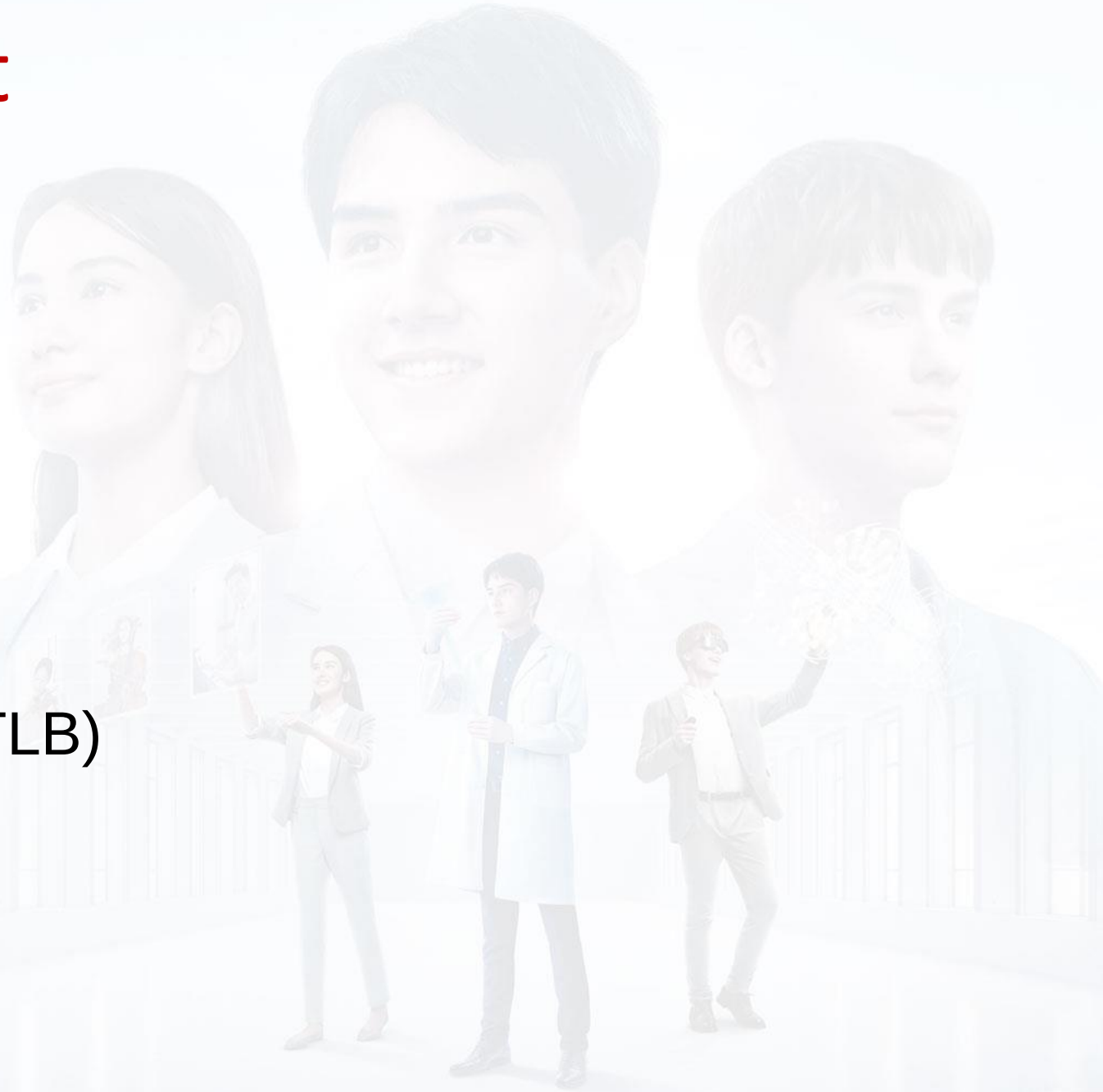
- Implementing RCU for a new data structure is difficult, but a few common data structures (lists, queues, trees) do have RCU APIs that can be used.

- RCU splits removal updates to the data structures in two phases:
  - Removal: removes references to elements. Some old readers may still see the old reference so we can't free the element.
  - Elimination: free the element. This action is postponed until all existing readers finish traversal (quiescent cycle). New readers won't affect the quiescent cycle.

RCU example (list)



Memory management



- What is it and why it is needed?
- Virtual and physical adresses
- MMU
- Address spaces
- Controlling address translation
- Translation granule
- Translation Look-aside Buffer (TLB)
- Address translation instructions

What is MM and why it is needed

- MM describes how access to memory in a system is controlled.
- The hardware performs memory management every time that memory is accessed by either the OS or applications.
- MM is a way of dynamically allocating regions of memory to applications.
- Application processors are designed to run an OS, e.g. Linux, and to support virtual memory systems.
- Software that executes on the processor only sees virtual addresses, which the processor translates into physical addresses.
- These physical addresses are presented to the memory system and point to the actual physical locations in memory.

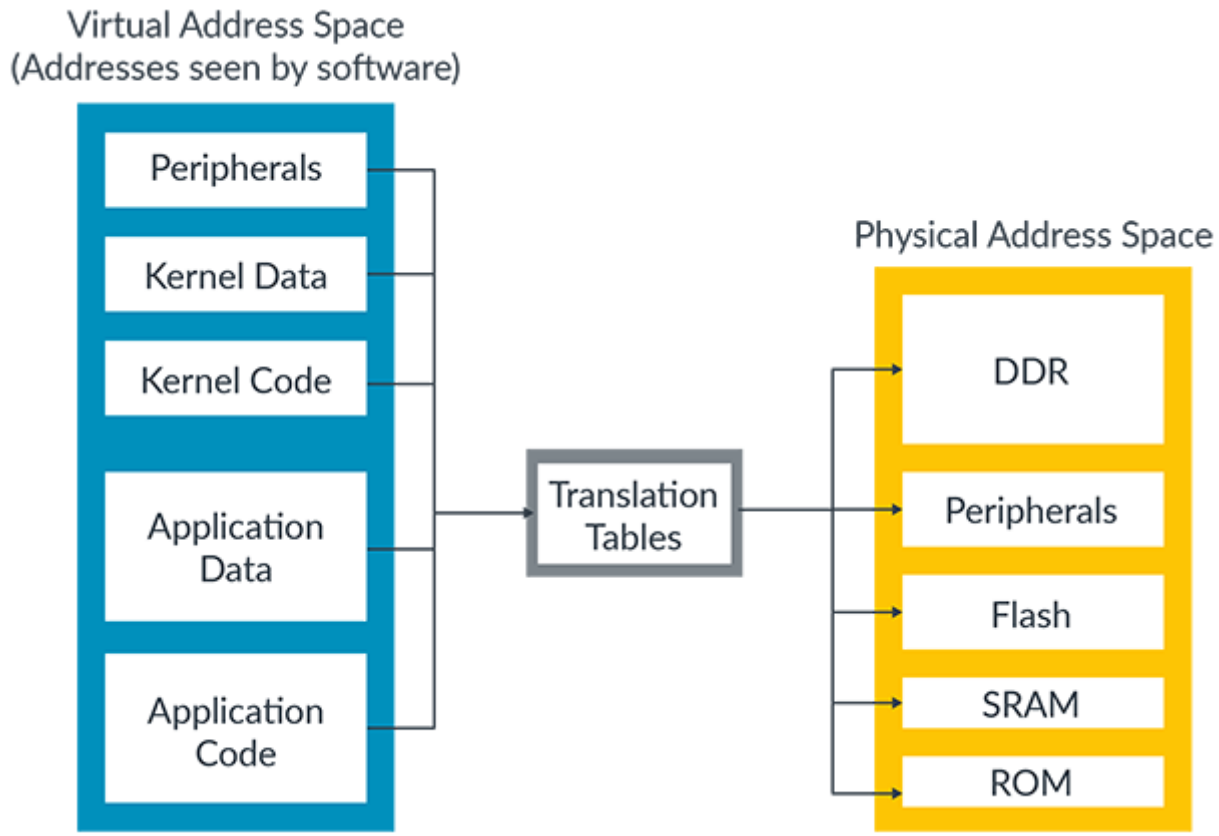
Virtual and physical adresses 1 of 2

- The benefit of using virtual addresses is that it allows OS to control the view of memory that is presented to software.
- The OS can control what memory is visible, the virtual address (VA) at which that memory is visible, and what accesses are permitted to that memory.
- This allows the OS to sandbox applications (hiding the resources of one application from another application) and to provide abstraction from the underlying hardware.
- One benefit of using VA is that an OS can present multiple fragmented physical regions of memory as a single, contiguous VA space to an application.
- VA benefita software developers, who will not know a system's exact memory addresses when writing their application.
- With VA software developers do not need to concern themselves with the physical memory. The application knows that it is up to the OS and the hardware to work together to perform the address translation.
- In practice, each application can use its own set of VA that will be mapped to different locations in the physical system.
- As the operating system switches between different applications it re-programs the map. This means that the VA for the current application will map to the correct physical location in memory.

Virtual and physical adresses 2 of 2

- Translation tables are in memory and are managed by software, typically an OS or hypervisor.
-
- The translations tables are not static, and the tables can be updated as the needs of software change.

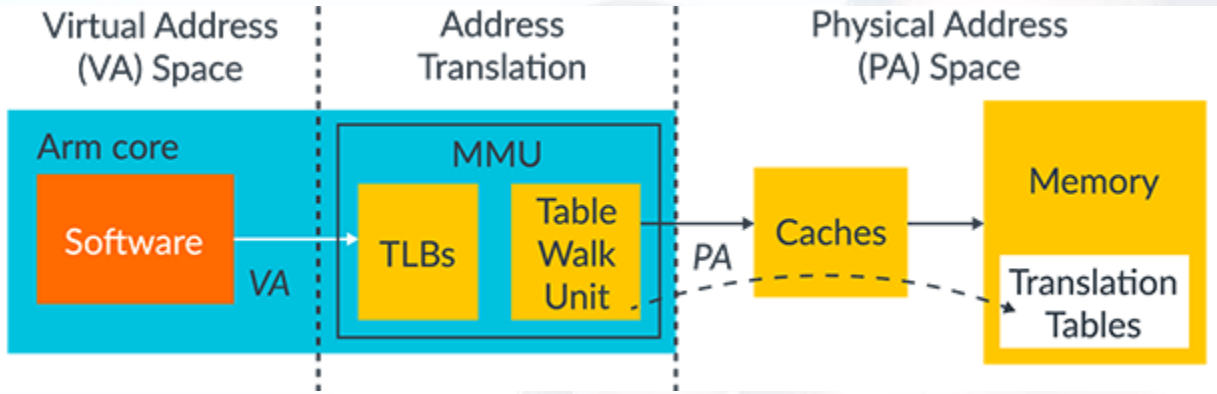
- This changes the mapping between virtual and physical addresses.



MMU 1 of 4

- The Memory Management Unit (MMU) is responsible for the translation of virtual addresses used by software to physical addresses used in the memory system.
- The MMU contains the following:
  - The table walk unit, which contains logic that reads the translation tables from memory.
  - Translation Lookaside Buffers (TLBs), which cache recently used translations.
- All memory addresses that are issued by software are virtual.
- These memory addresses are passed to the MMU, which checks the TLBs for a recently used cached translation.

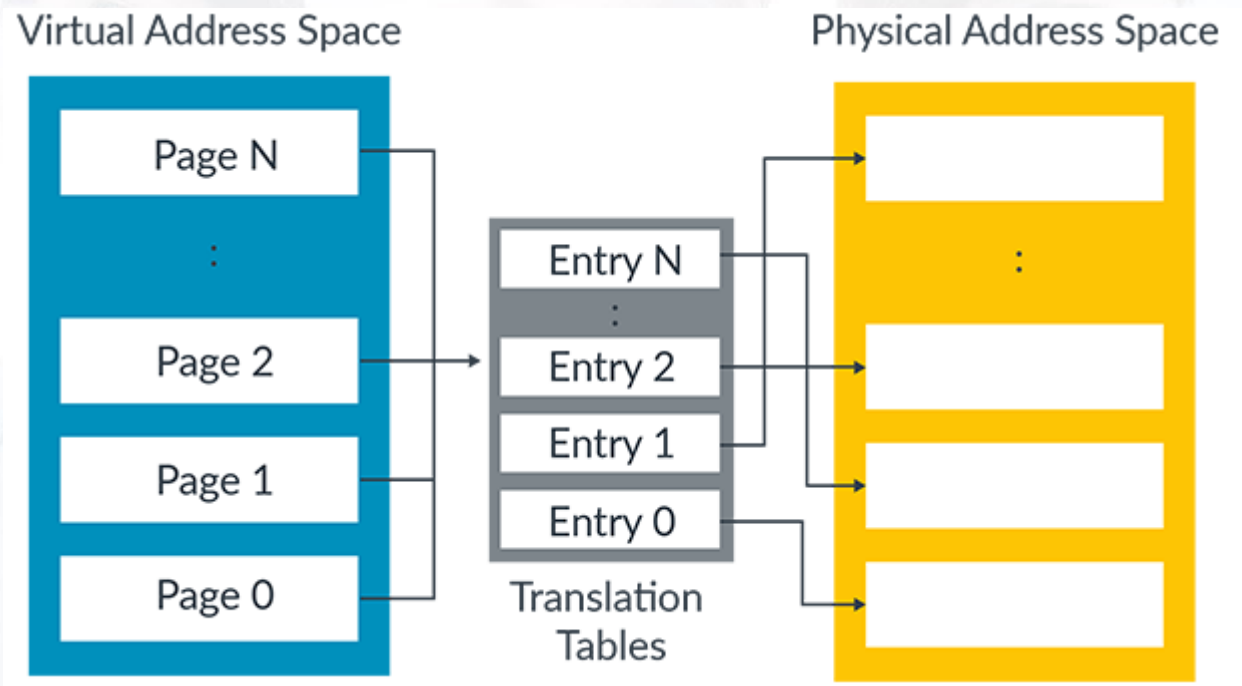
MMU 2 of 4



- If the MMU does not find a recently cached translation, the table walk unit reads the appropriate table entry, or entries, from memory, as shown above
- A virtual address must be translated to a physical address before a memory access can take place (because we must know which physical memory location we are accessing).
- This need for translation also applies to cached data, because on Arm processors, the data caches store data using the physical address (addresses that are physically tagged).
- Therefore, the address must be translated before a cache lookup can complete.

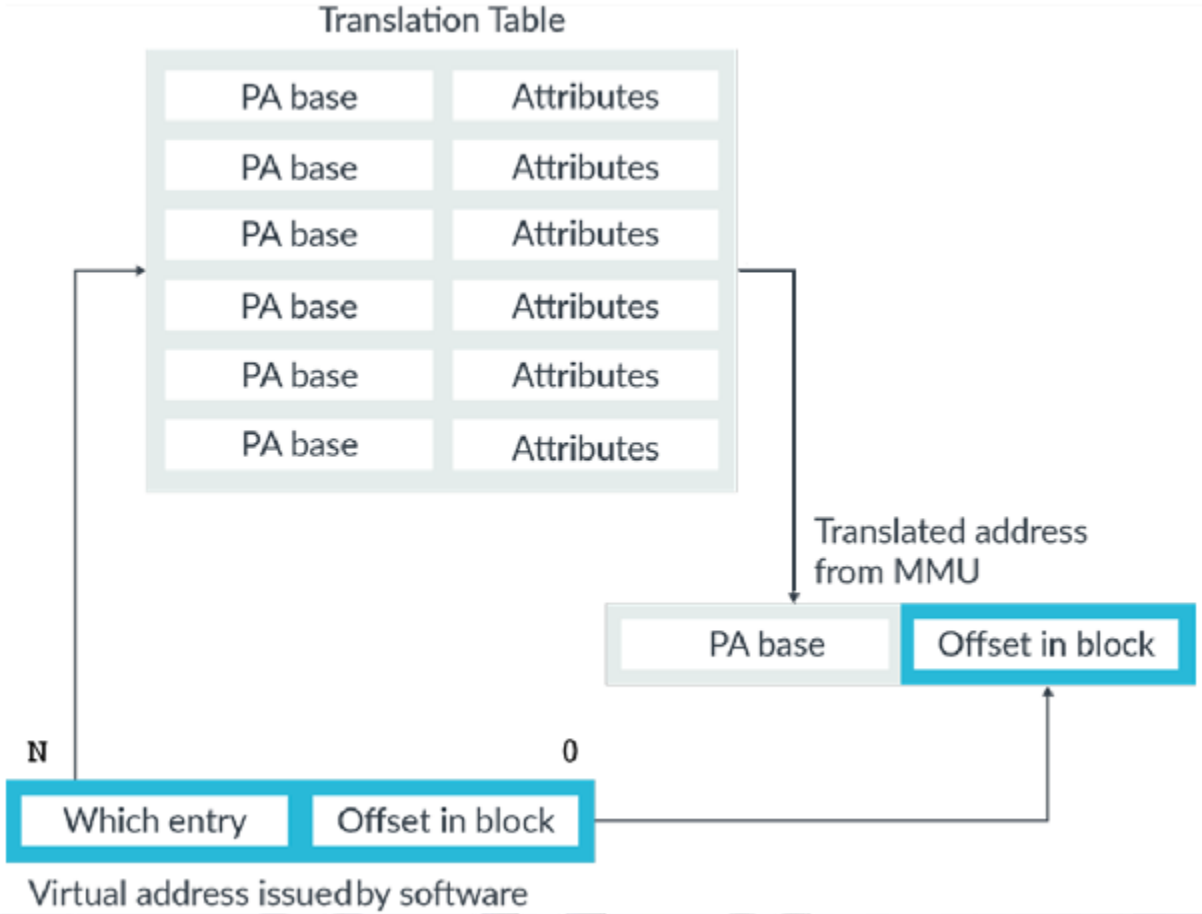
MMU 3 of 4: Table entry

- The translation tables work by dividing the virtual address space into equal-sized blocks and by providing one entry in the table per block.
- Entry 0 in the table provides the mapping for block 0, entry 1 provides the mapping for block 1, and so on.
- Each entry contains the address of a corresponding block of physical memory and the attributes to use when accessing the physical address.



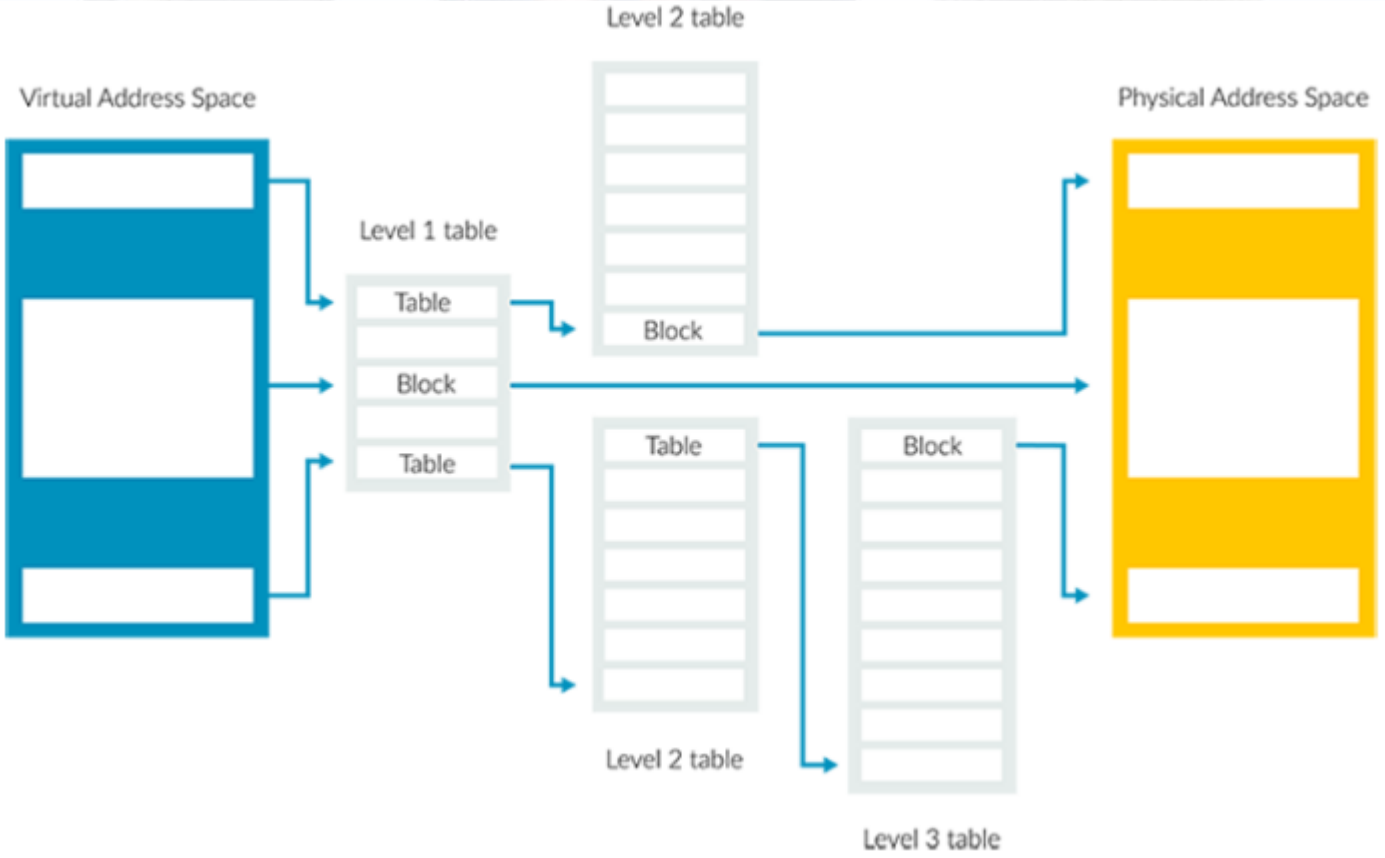
MMU 4 of 4: Table lookup

- A table lookup occurs when a translation takes place.
- When a translation happens, the virtual address that is issued by the software is split in two, as shown in the right
- This diagram shows a single-level lookup(!)
- The upper-order bits, which are labelled ‘Which entry’ in the diagram, tell you which block entry to look in and they are used as an index into the table.
- This entry block contains the physical address for the virtual address.
- The lower-order bits, which are labelled ‘Offset in block’ in the diagram, are an offset within that block and are not changed by the translation.



Multi-level translation 1 of 2

- The first table (Level 1 table) divides the virtual address space into large blocks.
- Each entry in this table can point to an equal-sized block of physical memory or it can point to another table which subdivides the block into smaller blocks.
- We call this type of table a ‘multilevel table’.
- In Armv8-A, the maximum number of levels is four, and the levels are numbered 0 to 3. This multilevel approach allows both larger blocks and smaller blocks to be described.

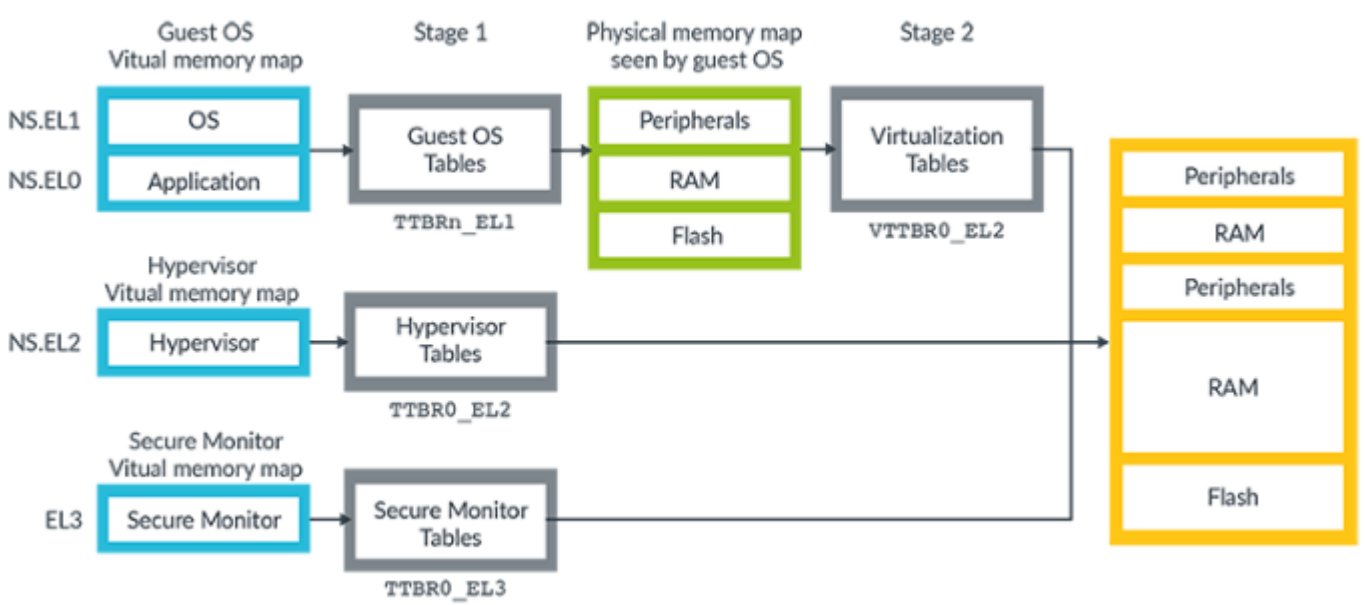


Multi-level translation 2 of 2

- The characteristics of large and small blocks are as follows:
  - Large blocks require fewer levels of reads to translate than small blocks.
  - Plus, large blocks are more efficient to cache in the TLBs.
  - Small blocks give software fine-grain control over memory allocation.
  - However, small blocks are less efficient to cache in the TLBs.
  - Caching is less efficient because small blocks require multiple reads through the levels to translate.
- To manage this trade-off, an OS must balance the efficiency of using large mappings against the flexibility of using smaller mappings for optimum performance.
- The processor does not know the size of the translation when it starts the table lookup.
- The processor works out the size of the block that is being translated by performing the table walk.

Address spaces

- The diagram shows three virtual address spaces:
  - Non-secure EL0 and EL1.
  - Non-secure EL2.
  - EL3.
- Each of these virtual address spaces is independent, and has its own settings and tables.
- We often call these settings and tables ‘translation regimes’.
- There are also virtual address spaces for Secure EL0, Secure EL1 and Secure EL2, but they are not shown in the diagram.
- The diagram also shows that the virtual addresses from Non-secure EL0 and Non-secure EL1 go through two sets of tables.
- These tables support virtualization and allow the hypervisor to virtualize the view of physical memory that is seen by a virtual machine (VM).

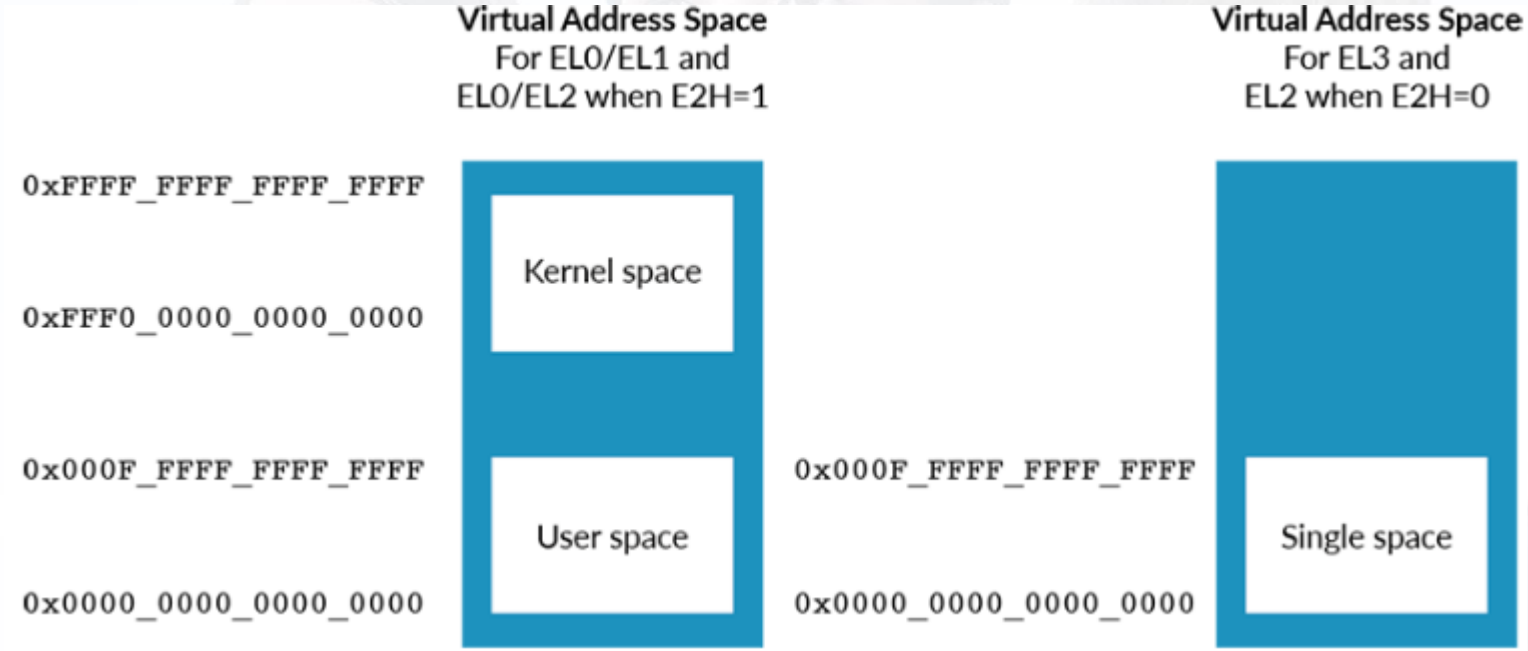


Physical addresses

- As well as multiple virtual address spaces, AArch64 also has multiple physical address spaces (PAS):
  - Non-secure PAS0
  - Secure PAS
  - Realm PAS (Armv9-A only)
  - Root PAS (Armv9-A only)
- Which physical address space, or spaces, a virtual address can be mapped to depends on the current Security state of the processor.
- The following list shows the Security states with its corresponding virtual address mapping destinations:
  - Non-secure state: virtual addresses can only map to Non-secure physical addresses.
  - Secure state: virtual addresses can map to Secure or Non-secure physical addresses.
  - Realm state: virtual addresses can map to Realm or Non-secure physical addresses
  - Root state: virtual address can map to any physical address space.

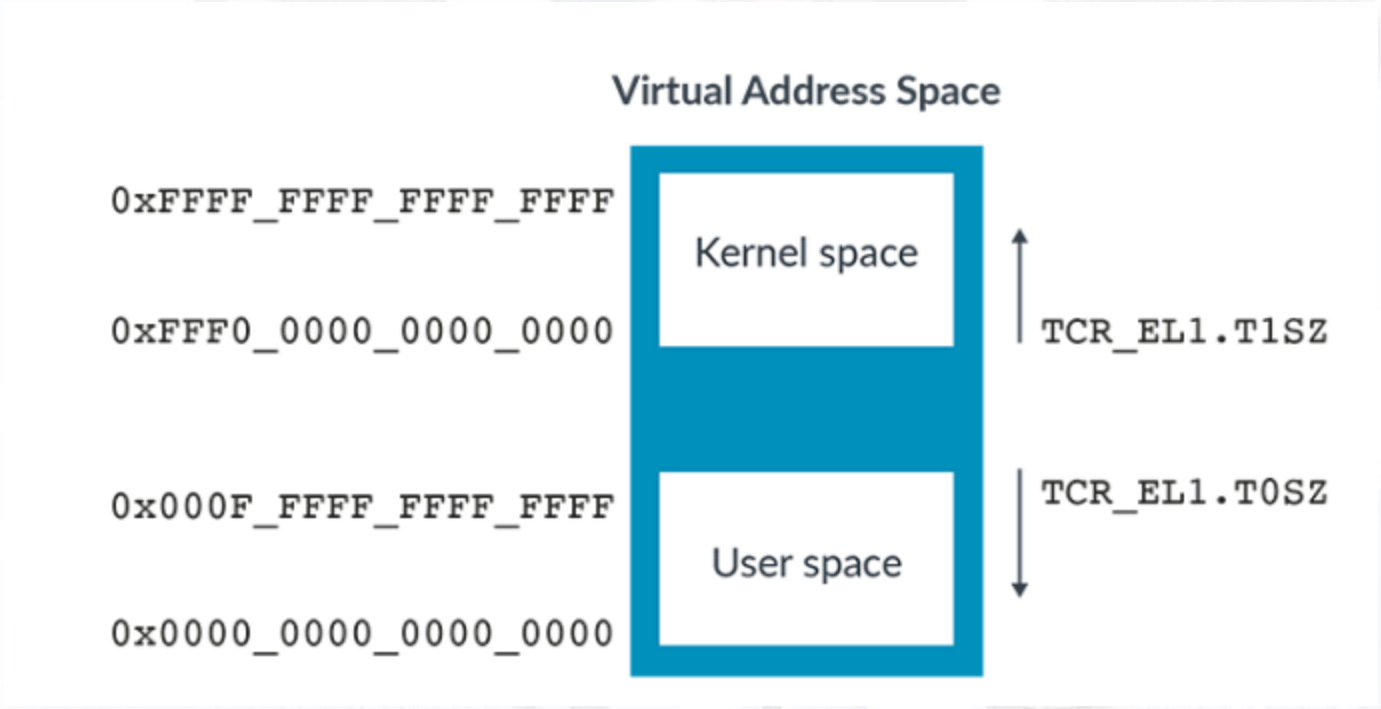
Address sizes 1 of 2

- Virtual addresses are stored in a 64-bit format.
- As a result, the address in load instructions (LDR) and store instructions (STR) is always specified in an X register.
- However, not all of the addresses in the X register are valid.
- There are two regions for the EL0/EL1 virtual address space: kernel space and application space.
- These two regions are shown on the left-hand side of the diagram, with kernel space at the top, and application space, which is labelled ‘User space’, at the bottom of the address space.
- Kernel space and user space have separate translation tables and this means that their mappings can be kept separate.
- There is a single region at the bottom of the address space for all other Exception levels.
- This region is shown on the right-hand side of the diagram and is the box with no text in it.



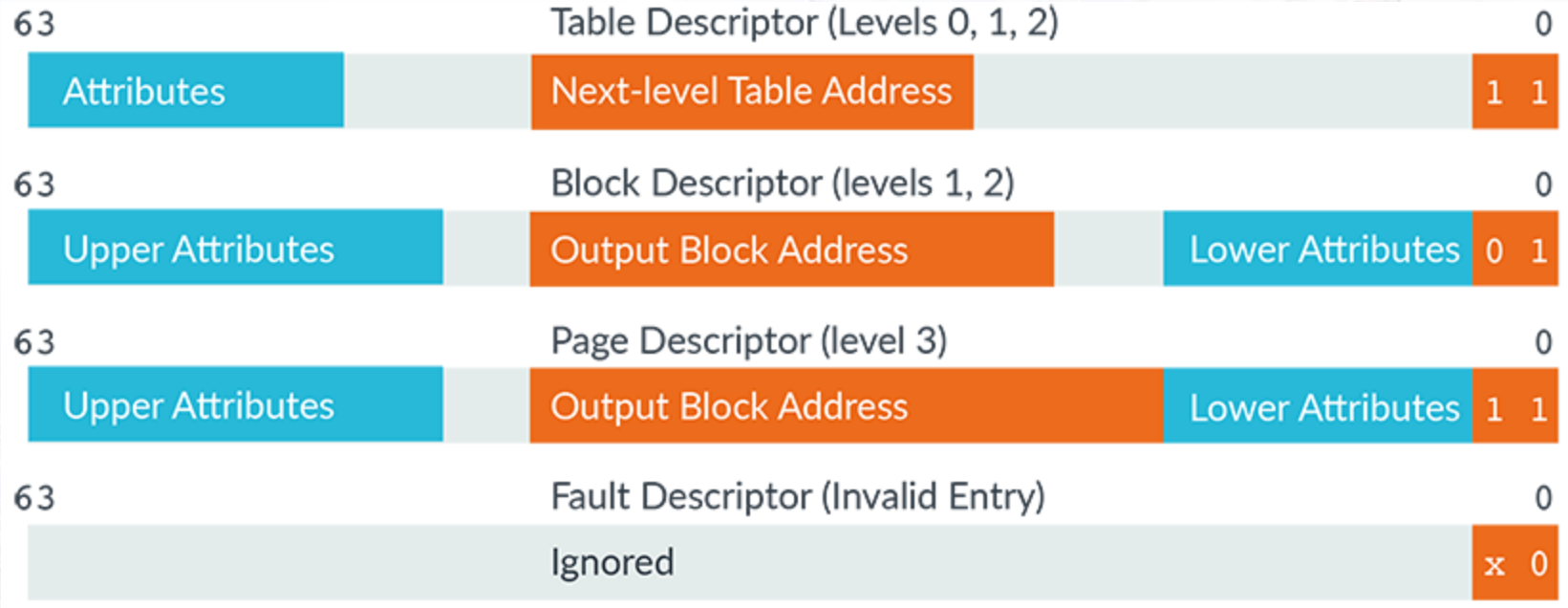
Address sizes 2 of 2

- Each region of address space has a size of up to 52-bits.
- However, each region can be independently shrunk to a smaller size.
- The TnSZ fields in the TCR_ELx registers control the size of the virtual address space.
- For example, this diagram shows that TCR_EL1 controls the EL0/EL1 virtual address space



Controlling address translation

- Each entry is 64 bits and the bottom two bits determine the type of entry.
- Notice that some of the table entries are only valid at specific levels.
- The maximum number of levels of tables is four, which is why there is no table descriptor for level 3 (or the fourth level), tables.
- Similarly, there are no Block descriptors or Page descriptors for level 0. Because level 0 entry covers a large region of virtual address space, it does not make sense to allow blocks.
- The encoding for the Table descriptor at levels 0-2 is the same as the Page descriptor at level 3.
- This encoding allows ‘recursive tables’, which point back to themselves.
- This is useful because it makes it easy to calculate the virtual address of a particular page table entry so that it can be updated.



Translation granule

- A translation granule is the smallest block of memory that can be described.
- Nothing smaller can be described, only larger blocks, which are multiples of the granule.
- AArch64 supports three different granule sizes: 4KB, 16KB, and 64KB.
- The granule sizes that a processor supports are IMPLEMENTATION DEFINED and are reported by ID_AA64MMFR0_EL1.
- All Arm Cortex-A processors support 4KB and 64KB.
- The selected granule is the smallest block that can be described in the latest level table.
- Larger blocks can also be described.

Level of table	4KB granule	4KB granule	16KB granule	16KB granule	64KB granule	64KB granule
	Size per entry	Bits used to index	Size per entry	Bits used to index	Size per entry	Bits used to index
0	512GB	47:39	128TB	47	–	–
1	1GB	38:30	64GB	46:36	4TB	51:42
2	2MB	29:21	32MB	35:25	512MB	41:29
3	4KB	20:12	16KB	24:14	64KB	28:16

Translation Look-aside Buffer (TLB)

- The Translation Lookaside Buffers (TLBs) cache recently used translations.
- This caching allows the translations to be reused by subsequent lookups without needing to reread the tables.
- If you change a translation table entry, or the controls that affect how entries are interpreted, then you need to invalidate the affected entries in the TLB.
- If you do not invalidate those entries, then the processor might continue to use the old translation.

Address translation instructions

- An Address Translation (AT) instruction lets the software query the translation for a specific address.
- The translation that results, including the attributes, is written to the Physical Address Register, PAR_EL1.
- The syntax of the AT instruction lets you specify which translation regime to use.
- For example, EL2 can query the EL0/EL1 translation regime. However, EL1 cannot use the AT instruction to query the EL2 translation regime, as this is a breach of privilege.
- If the requested translation would have caused a fault, no exception is generated.
- Instead, the type of fault that would have been generated is recorded in PAR_EL1.

Memory management summary

- Q: What is the difference between a stage and a level in address translation?
- A: A stage is the process of translating an input address to an output address. For Stage 1 this is the process of going from VA to IPA and for Stage 2 going from IPA to PA.
- A level refers to the tables in a given stage of translation. It is also how a larger block can be subdivided into smaller blocks.

- Q: What is the maximum size of a physical address?
- A: The maximum size of the physical address space is IMPLEMENTATION DEFINED, and up to 52 bits (since Armv8.2-A).

- Q: Which register field controls the size of the virtual address space?
- A: TCR_ELx.TnSZ, or VTCR_EL2.T0SZ for Stage 2.

- Q: What is a translation granule, and what are the supported sizes?
- A: It is the smallest block of memory that can be described. The supported sizes are 4KB, 16KB, and 64KB.

- Q: How are addresses mapped when the MMU is disabled?
- A: Addresses are flat mapped, so that the input and output addresses are the same.