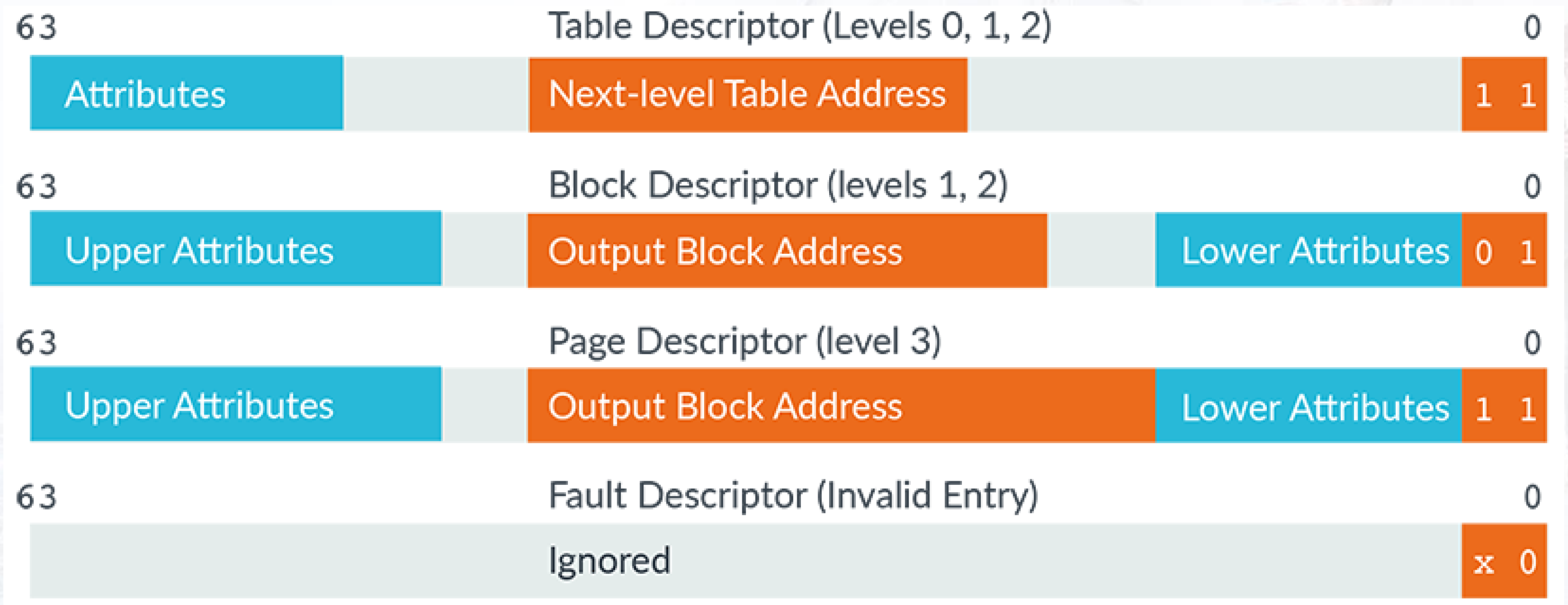
ITMO Advanced OS 2024

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

Lesson 2024.10.16

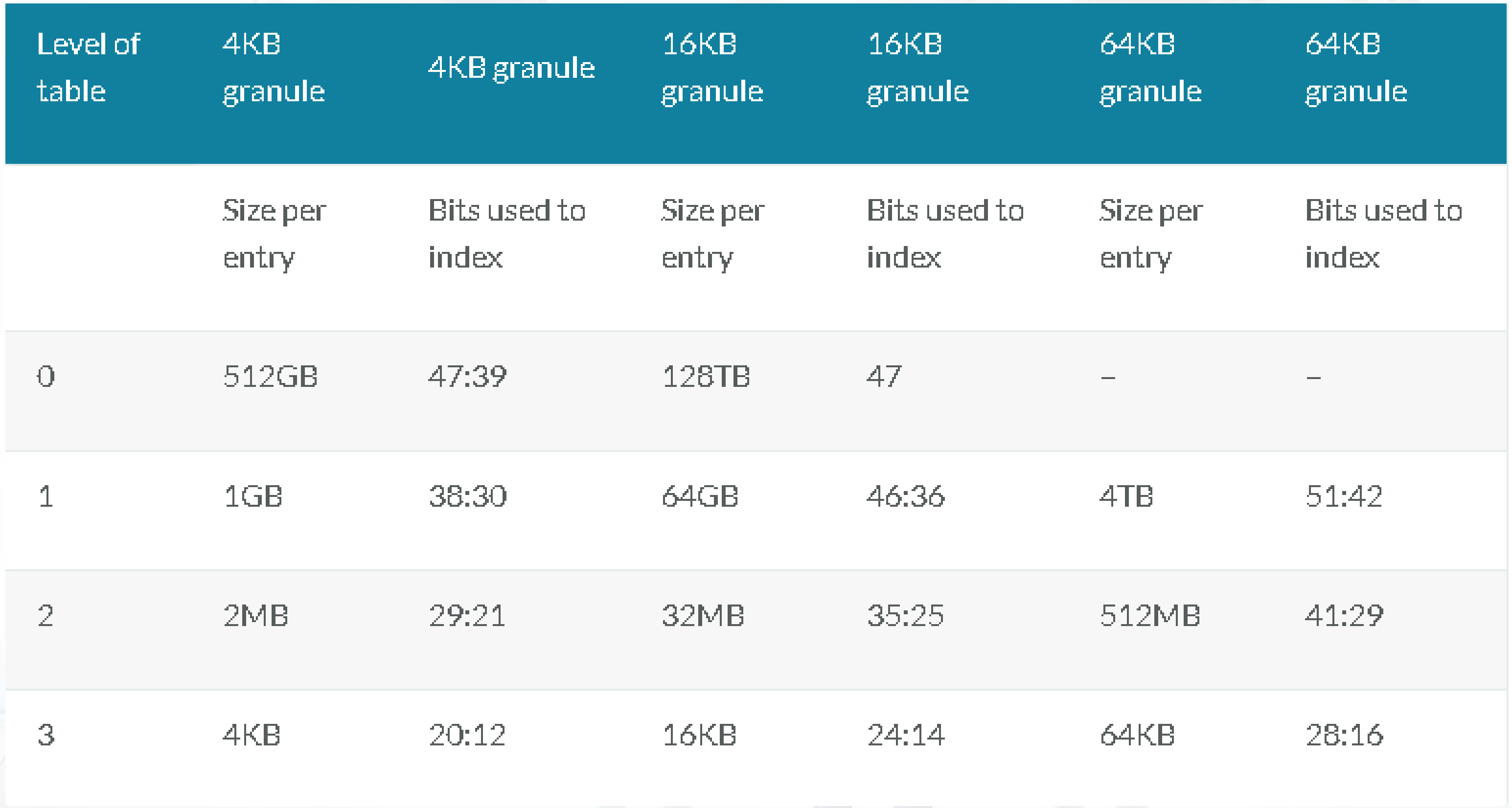
Controlling address translation

- 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.



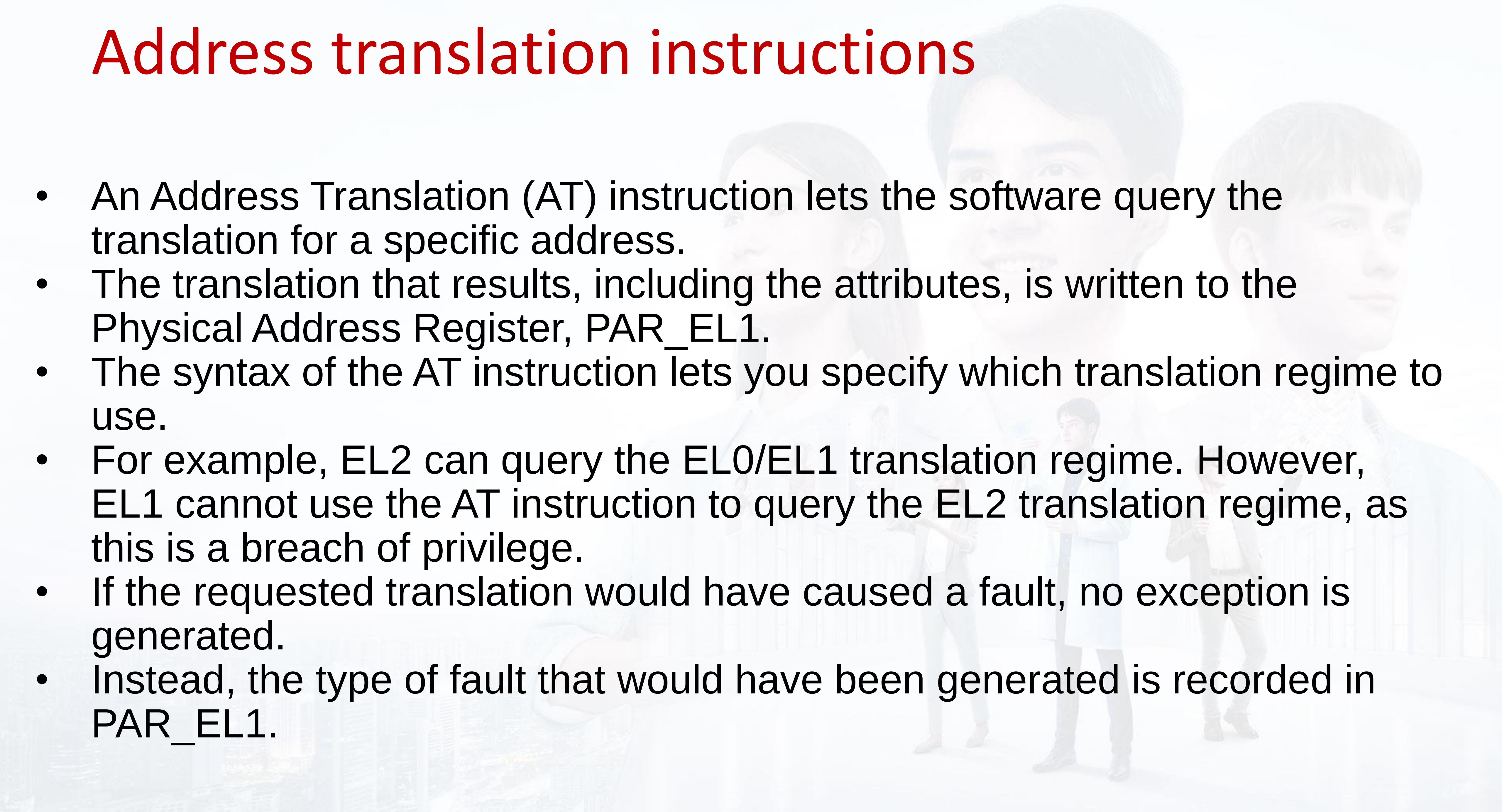
Translation granule

- 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.



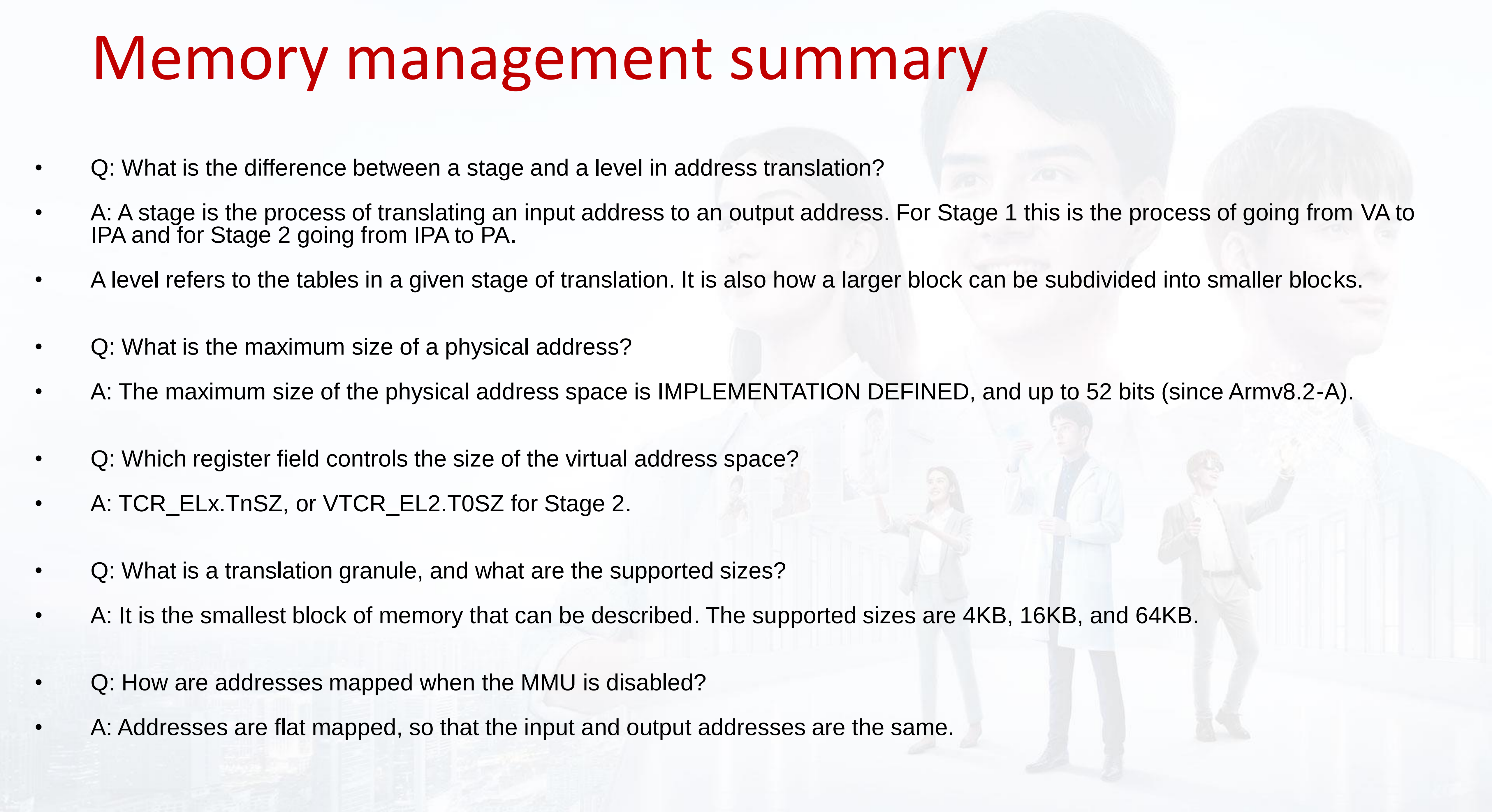
Level of table	4KB granule	4KB granule	16KB granule	16KB granule	64KB granule	64KB granule
	Size per entry	Bits used to index	Size per entry	Bits used to index	Size per entry	Bits used to index
0	512GB	47:39	128TB	47	–	–
1	1GB	38:30	64GB	46:36	4TB	51:42
2	2MB	29:21	32MB	35:25	512MB	41:29
3	4KB	20:12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

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).

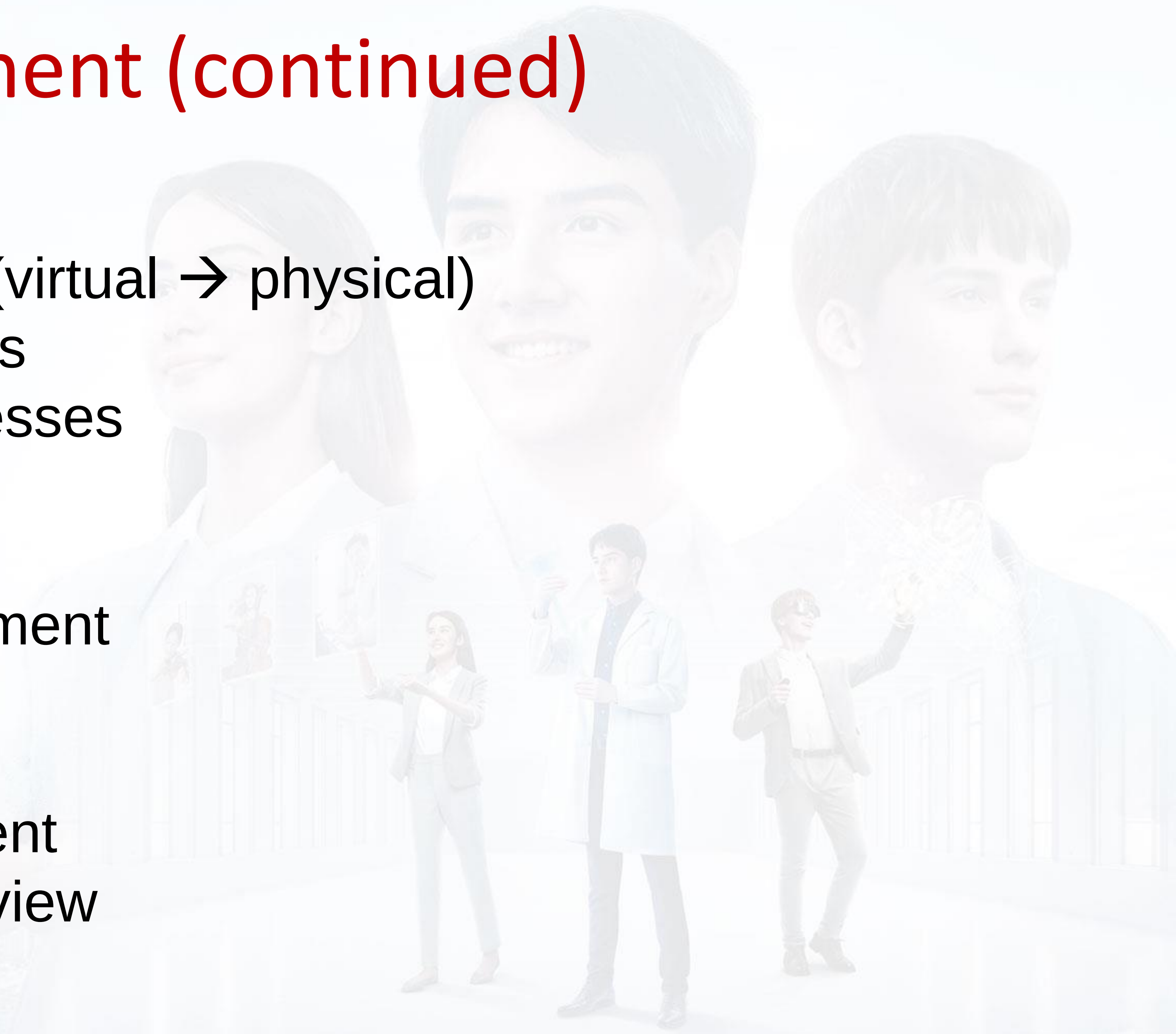
- Q: Which register field controls the size of the virtual address space?
- A: TCR_ELx.TnSZ, or VTCR_EL2.T0SZ for Stage 2.

- 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.

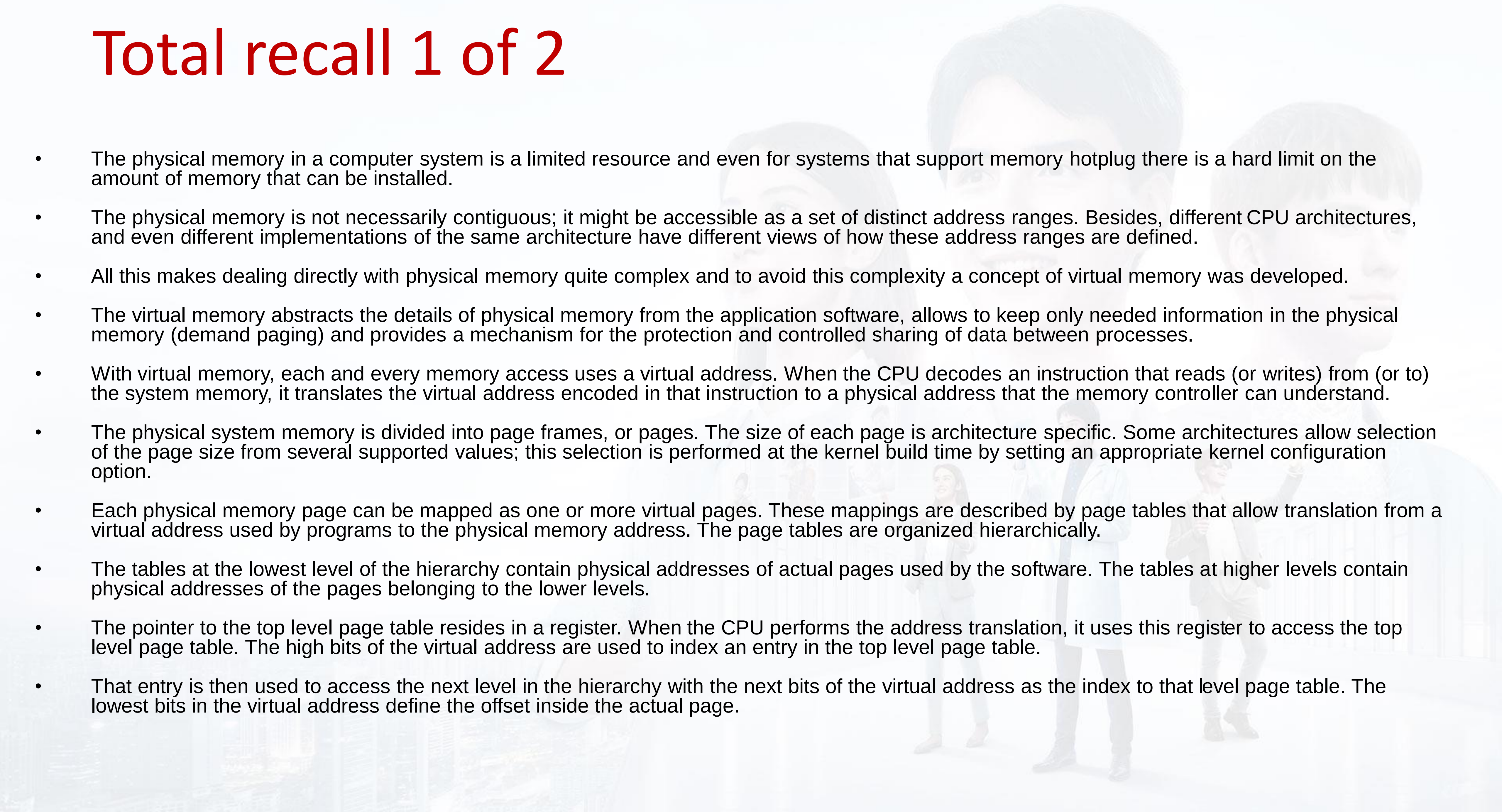
- Q: How are addresses mapped when the MMU is disabled?
- A: Addresses are flat mapped, so that the input and output addresses are the same.

Memory management (continued)

- Total recall
- Linear memory mappings (virtual → physical)
- Arbitrary memory mappings
- Fixed mapped linear addresses
- Temporary mappings
- Permanent mappings
- Physical Memory Management
  - Page allocations
  - Small allocations
- Virtual Memory Management
- Page Fault Handling Overview

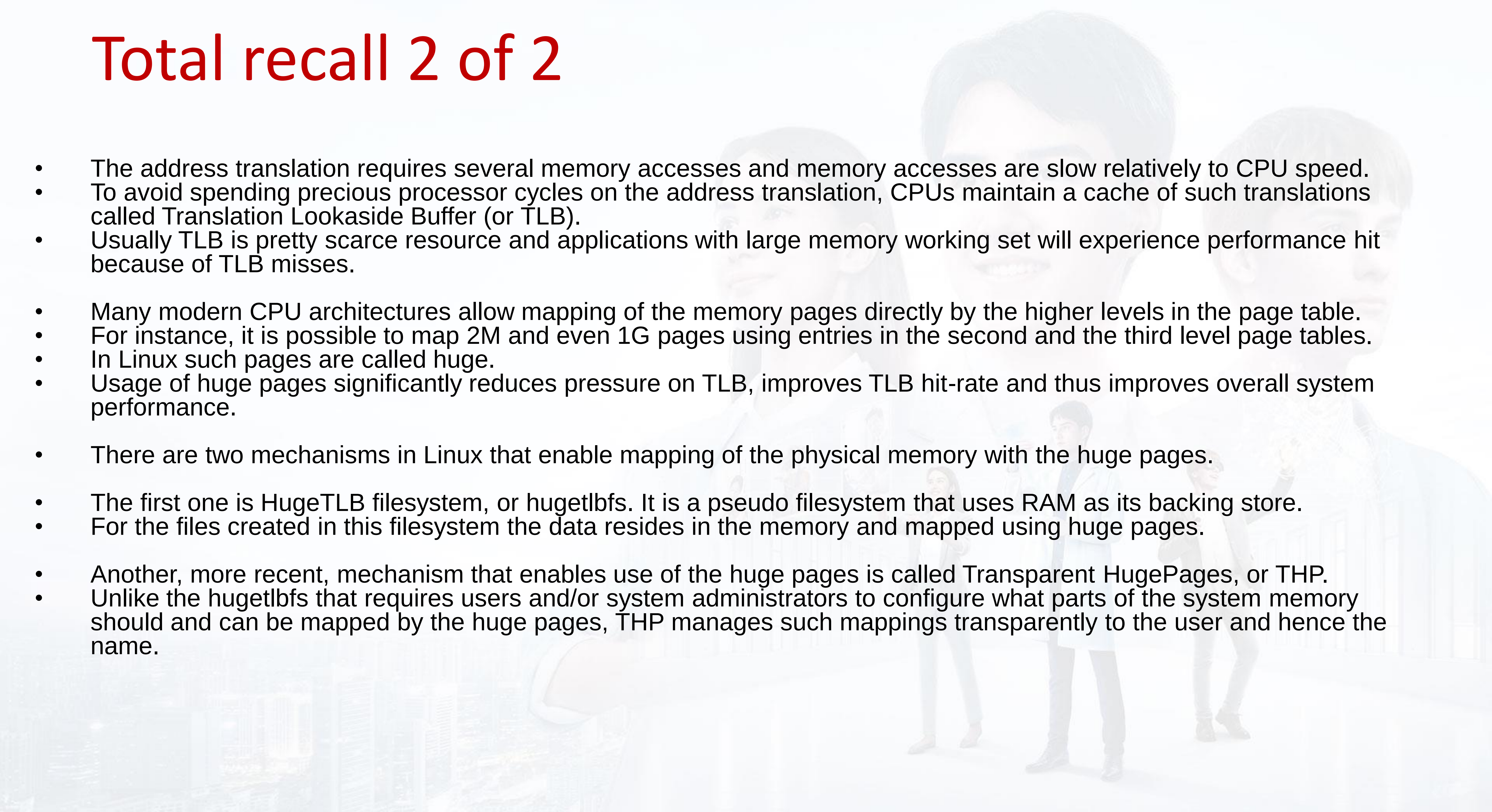


Total recall 1 of 2



- The physical memory in a computer system is a limited resource and even for systems that support memory hotplug there is a hard limit on the amount of memory that can be installed.
- The physical memory is not necessarily contiguous; it might be accessible as a set of distinct address ranges. Besides, different CPU architectures, and even different implementations of the same architecture have different views of how these address ranges are defined.
- All this makes dealing directly with physical memory quite complex and to avoid this complexity a concept of virtual memory was developed.
- The virtual memory abstracts the details of physical memory from the application software, allows to keep only needed information in the physical memory (demand paging) and provides a mechanism for the protection and controlled sharing of data between processes.
- With virtual memory, each and every memory access uses a virtual address. When the CPU decodes an instruction that reads (or writes) from (or to) the system memory, it translates the virtual address encoded in that instruction to a physical address that the memory controller can understand.
- The physical system memory is divided into page frames, or pages. The size of each page is architecture specific. Some architectures allow selection of the page size from several supported values; this selection is performed at the kernel build time by setting an appropriate kernel configuration option.
- Each physical memory page can be mapped as one or more virtual pages. These mappings are described by page tables that allow translation from a virtual address used by programs to the physical memory address. The page tables are organized hierarchically.
- The tables at the lowest level of the hierarchy contain physical addresses of actual pages used by the software. The tables at higher levels contain physical addresses of the pages belonging to the lower levels.
- The pointer to the top level page table resides in a register. When the CPU performs the address translation, it uses this register to access the top level page table. The high bits of the virtual address are used to index an entry in the top level page table.
- That entry is then used to access the next level in the hierarchy with the next bits of the virtual address as the index to that level page table. The lowest bits in the virtual address define the offset inside the actual page.

Total recall 2 of 2



- The address translation requires several memory accesses and memory accesses are slow relatively to CPU speed.
- To avoid spending precious processor cycles on the address translation, CPUs maintain a cache of such translations called Translation Lookaside Buffer (or TLB).
- Usually TLB is pretty scarce resource and applications with large memory working set will experience performance hit because of TLB misses.

- Many modern CPU architectures allow mapping of the memory pages directly by the higher levels in the page table.
- For instance, it is possible to map 2M and even 1G pages using entries in the second and the third level page tables.
- In Linux such pages are called huge.
- Usage of huge pages significantly reduces pressure on TLB, improves TLB hit-rate and thus improves overall system performance.

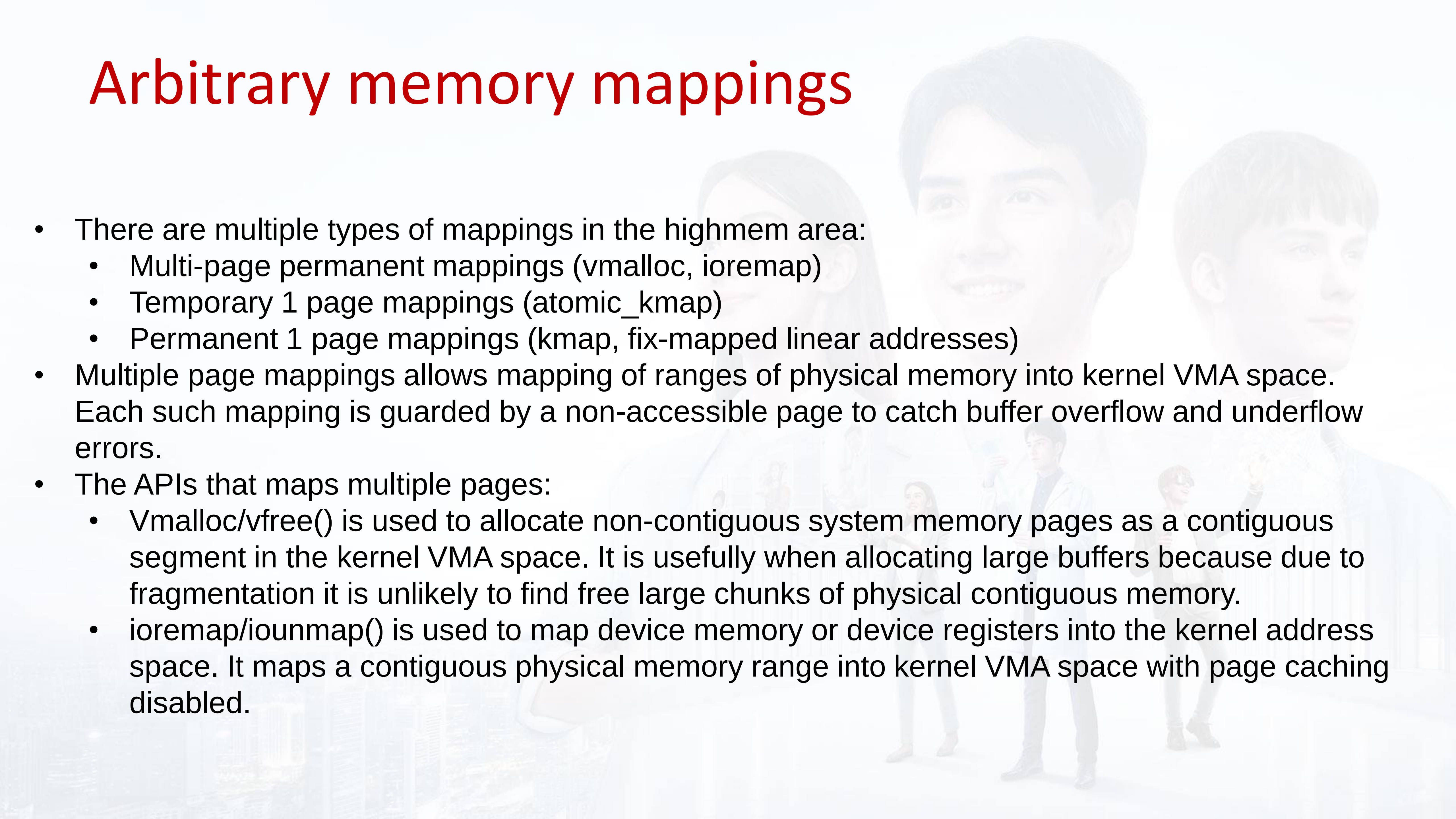
- There are two mechanisms in Linux that enable mapping of the physical memory with the huge pages.

- The first one is HugeTLB filesystem, or hugetlbfs. It is a pseudo filesystem that uses RAM as its backing store.
- For the files created in this filesystem the data resides in the memory and mapped using huge pages.

- Another, more recent, mechanism that enables use of the huge pages is called Transparent HugePages, or THP.
- Unlike the hugetlbfs that requires users and/or system administrators to configure what parts of the system memory should and can be mapped by the huge pages, THP manages such mappings transparently to the user and hence the name.

Linear memory mappings (virtual → physical)

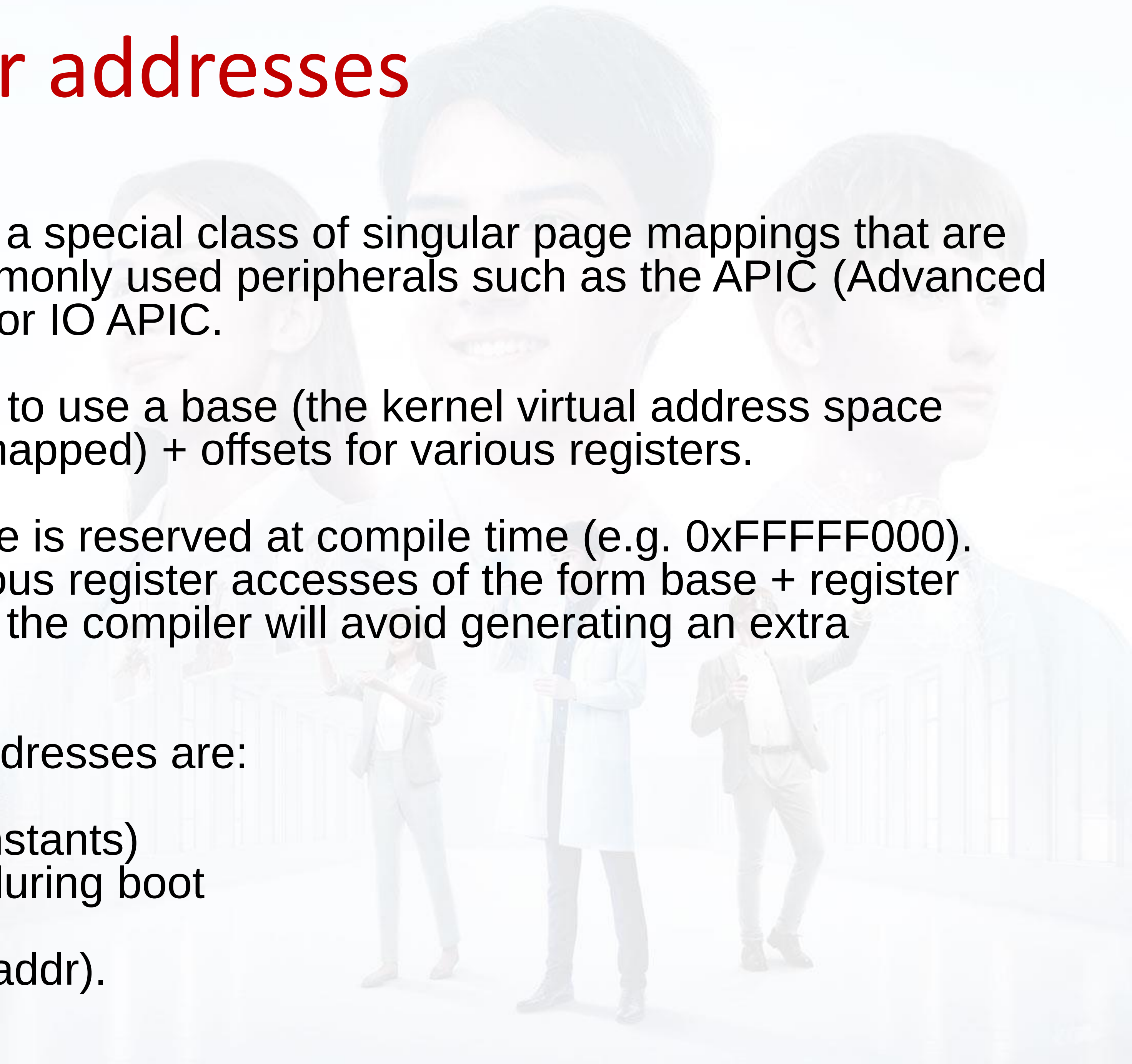
- Linear mappings refer to particular way of mapping virtual pages to physical pages, where virtual page V, V + 1, ... V + n is mapped to physical pages P, P + 1, ..., P + n.
- To understand the necessity of linear mappings, we should look at common kernel operations that involves using both the virtual and physical address of a page such as an I/O transfer:
  - Use the virtual address of a kernel buffer in order to copy to data from user space
  - Walk the page tables to transform the kernel buffer virtual address to a physical address
  - Use the physical address of the kernel buffer to start a DMA transfer
- However, if we use linear mappings and the kernel buffers are in the linear mapping area, then:
  - Virtual to physical address space translation is reduced to one operation (instead of walking the page tables)
  - Less memory is used to create the page tables
  - Less TLB entries are used for the kernel memory



Arbitrary memory mappings

- There are multiple types of mappings in the highmem area:
  - Multi-page permanent mappings (vmalloc, ioremap)
  - Temporary 1 page mappings (atomic_kmap)
  - Permanent 1 page mappings (kmap, fix-mapped linear addresses)
- Multiple page mappings allows mapping of ranges of physical memory into kernel VMA space. Each such mapping is guarded by a non-accessible page to catch buffer overflow and underflow errors.
- The APIs that maps multiple pages:
  - Vmalloc/vfree() is used to allocate non-contiguous system memory pages as a contiguous segment in the kernel VMA space. It is usefully when allocating large buffers because due to fragmentation it is unlikely to find free large chunks of physical contiguous memory.
  - ioremap/iounmap() is used to map device memory or device registers into the kernel address space. It maps a contiguous physical memory range into kernel VMA space with page caching disabled.

Fixed mapped linear addresses



- Fixed-mapped linear addresses are a special class of singular page mappings that are used for accessing registers of commonly used peripherals such as the APIC (Advanced Programmable Interrupt Controller) or IO APIC.

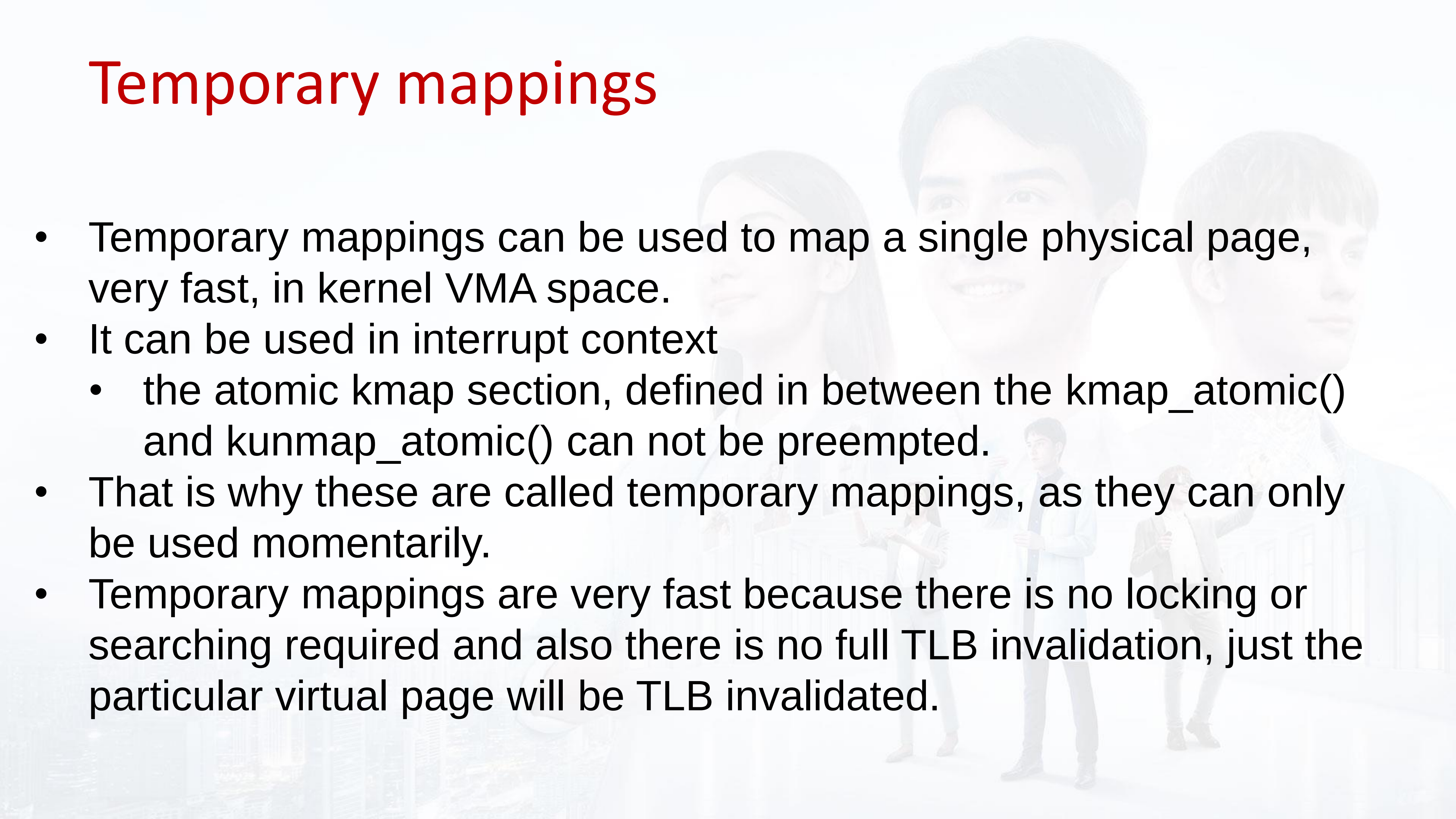
- Typical I/O access for peripherals is to use a base (the kernel virtual address space where the peripheral registers are mapped) + offsets for various registers.

- In order to optimize access, the base is reserved at compile time (e.g. 0xFFFFF000). Since the base is constant, the various register accesses of the form base + register offset will also be constant and thus the compiler will avoid generating an extra instruction.

- In summary, fixed-mapped linear addresses are:

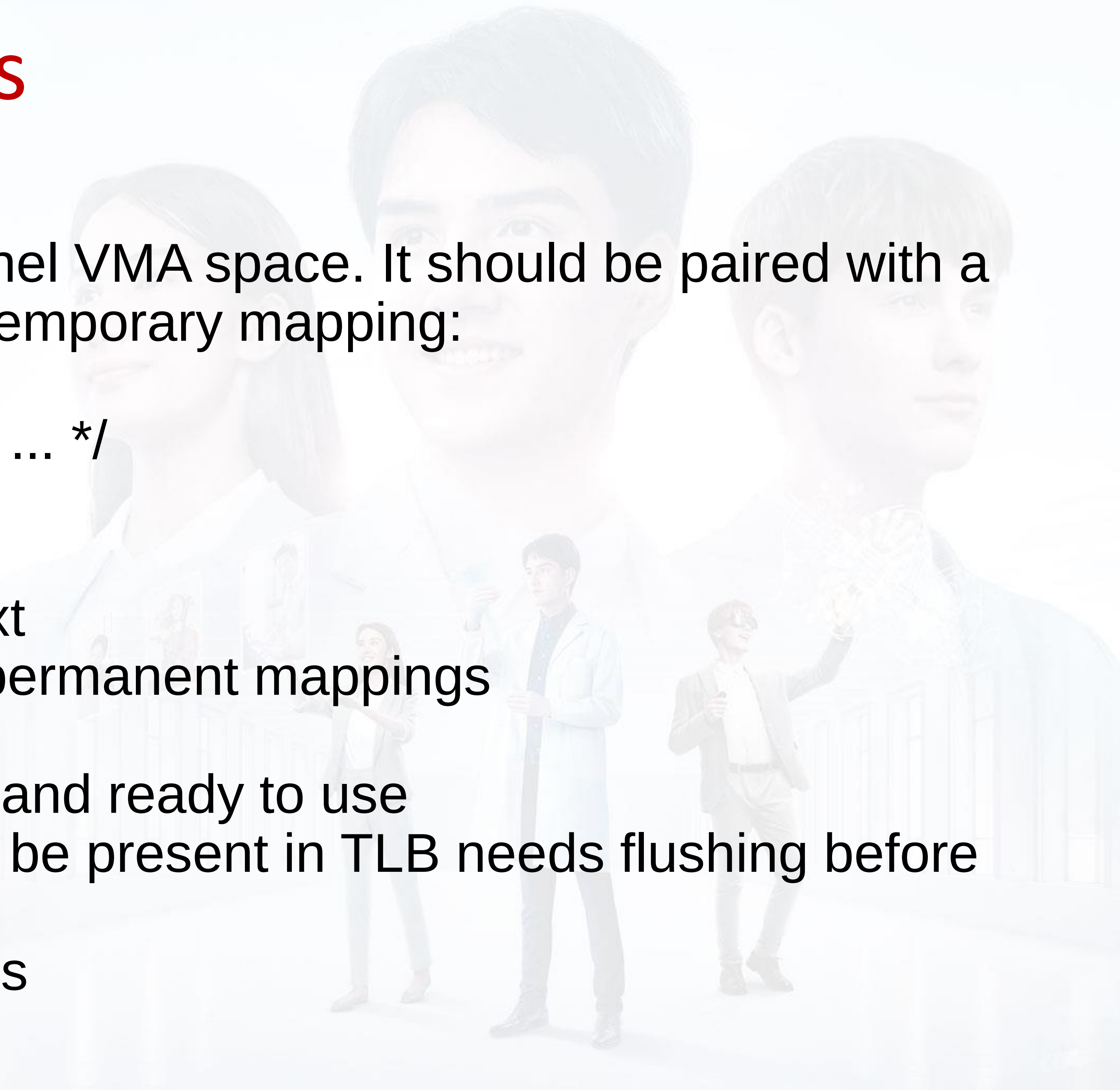
  - Reserved virtual addresses (constants)
  - Mapped to physical addresses during boot
  - set_fixmap(idx, phys_addr)
  - set_fixmap_nocache(idx, phys_addr).

Temporary mappings

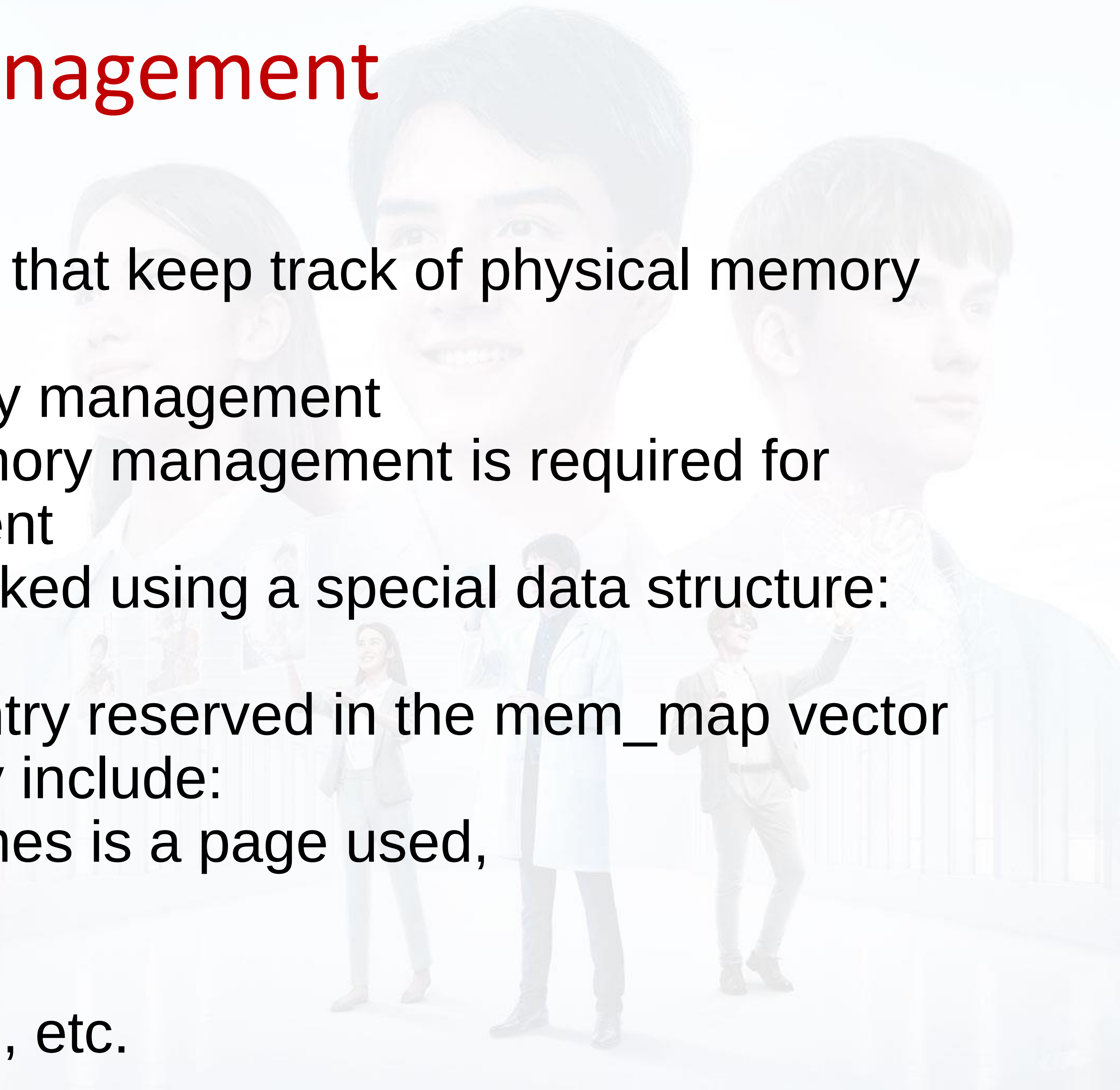


- Temporary mappings can be used to map a single physical page, very fast, in kernel VMA space.
- It can be used in interrupt context
  - the atomic kmap section, defined in between the kmap_atomic() and kunmap_atomic() can not be preempted.
- That is why these are called temporary mappings, as they can only be used momentarily.
- Temporary mappings are very fast because there is no locking or searching required and also there is no full TLB invalidation, just the particular virtual page will be TLB invalidated.

Permanent mappings



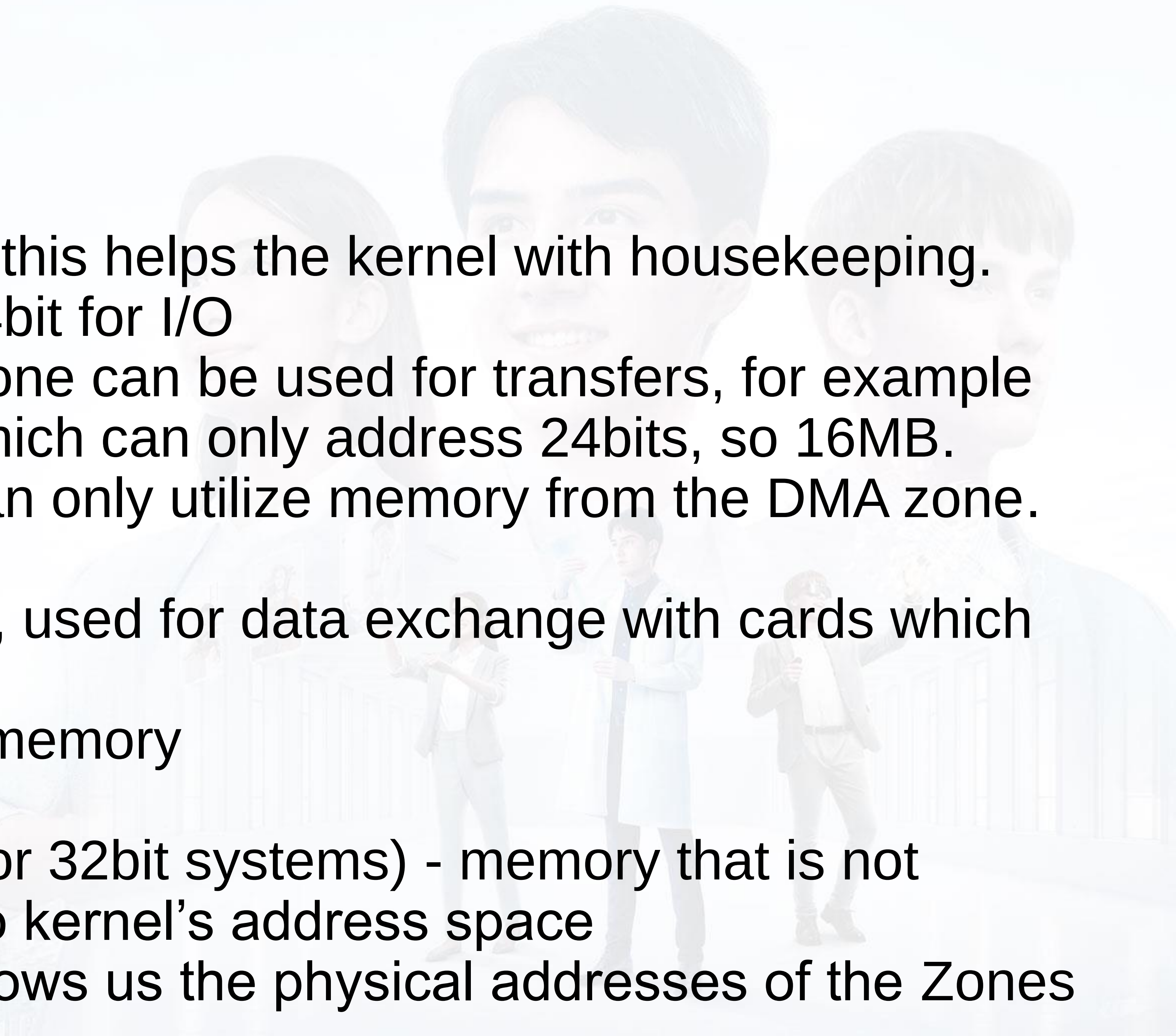
- kmap() will map a page into kernel VMA space. It should be paired with a call to kunmap() to release the temporary mapping:
  - p2 = kmap(pages[0]);
  - /* do something with p2 here ... */
  - kunmap(p2);
- Context switches are allowed
- Only available in process context
- One page table is reserved for permanent mappings
- Page counter
  - 0 - page is not mapped, free and ready to use
  - 1 - page is not mapped, may be present in TLB needs flushing before using
  - N - page is mapped N-1 times



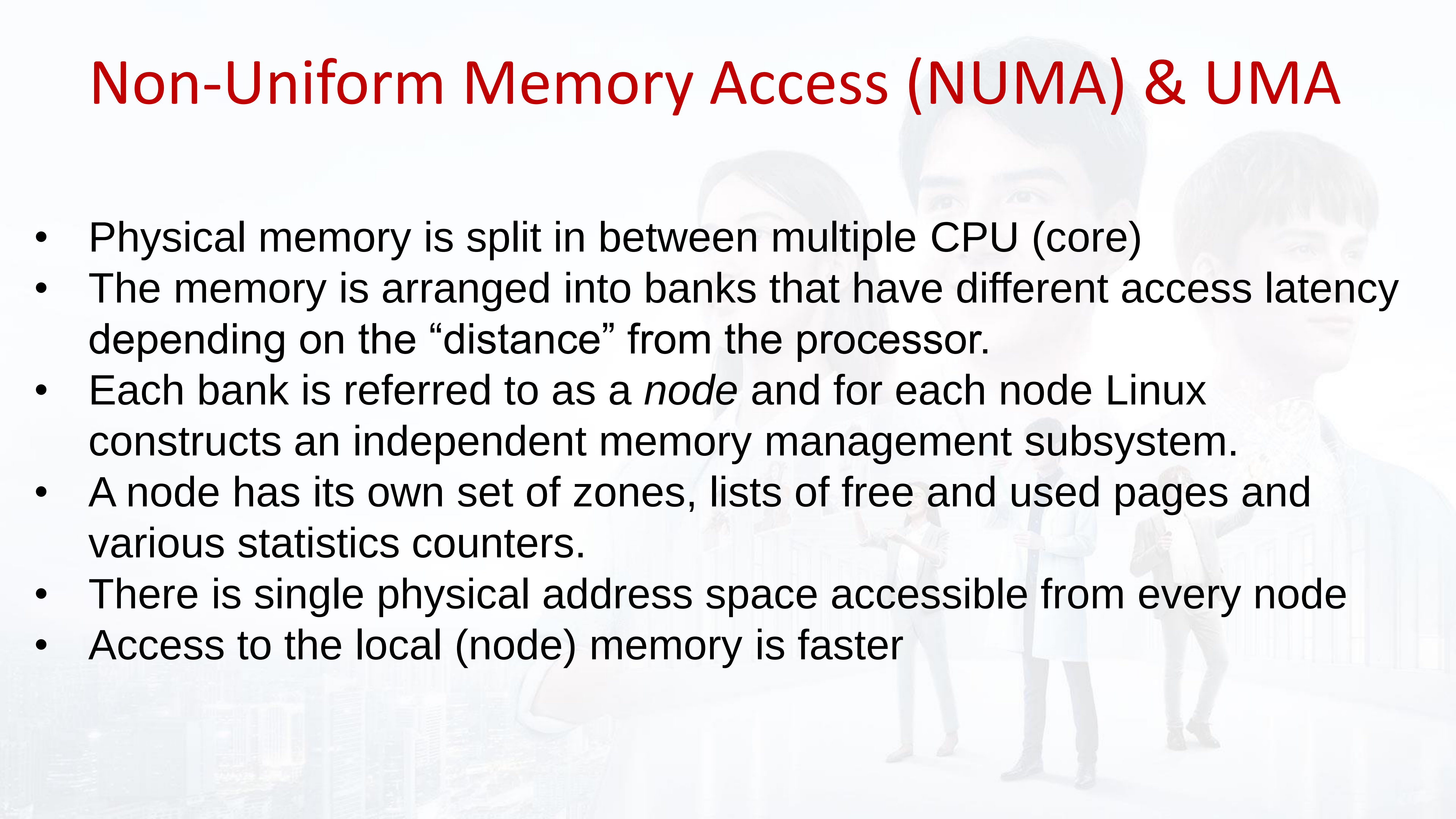
Physical memory management

- Algorithms and data structure that keep track of physical memory pages
- Independent of virtual memory management
- Both virtual and physical memory management is required for complete memory management
- Physical pages are being tracked using a special data structure: struct page
- All physical pages have an entry reserved in the mem_map vector
- The physical page status may include:
  - a counter for how many times is a page used,
  - position in swap or file,
  - buffers for this page,
  - position int the page cache, etc.

Memory zones



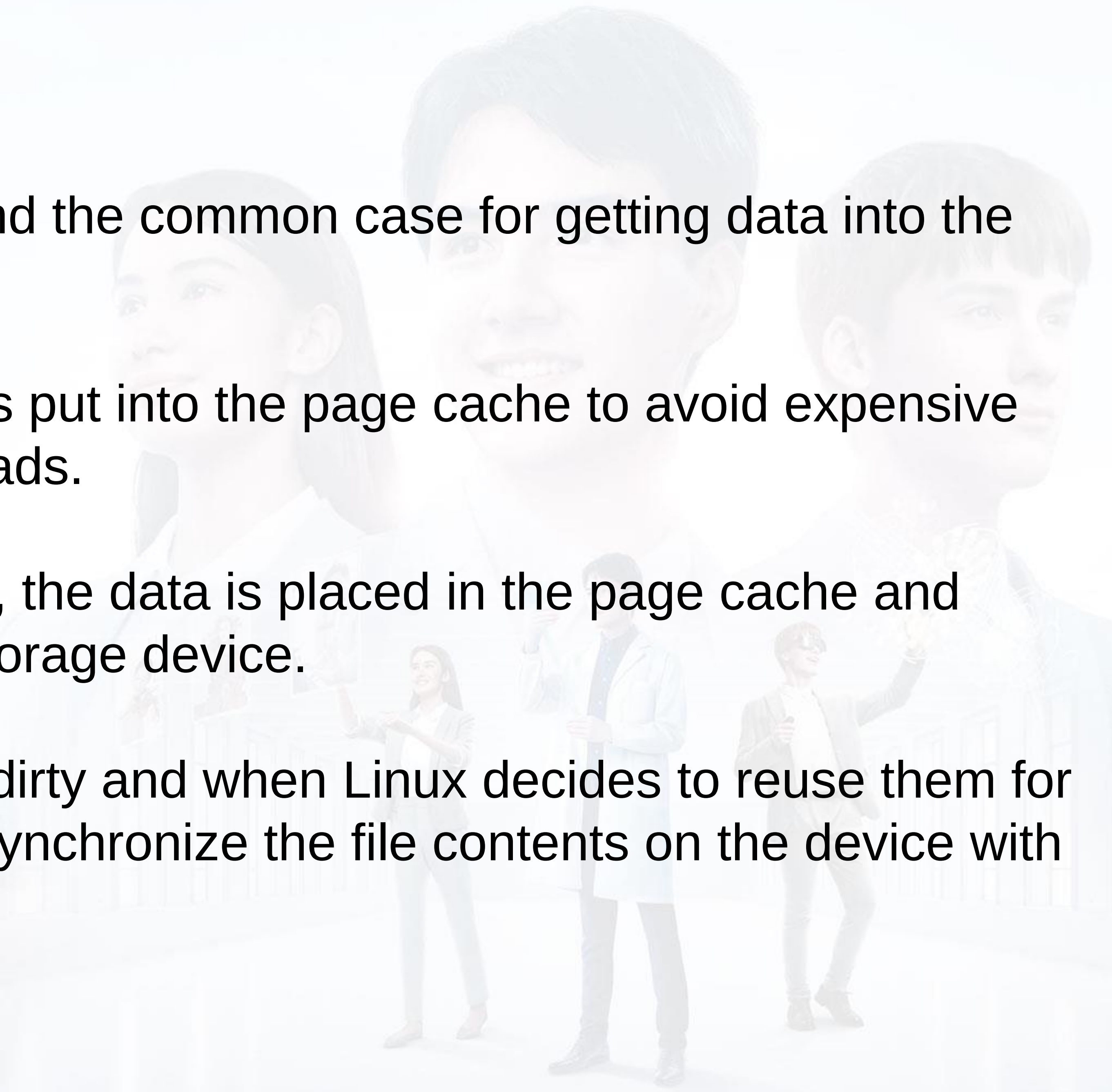
- Dividing the memory like this helps the kernel with housekeeping.
- Zone DMA: first 16MB/24bit for I/O
  - Memory in the DMA zone can be used for transfers, for example with network cards, which can only address 24bits, so 16MB.
  - Some cards/drivers can only utilize memory from the DMA zone.
- Zone DMA32: 4GB I/O
  - DMA32 is 4GB in size, used for data exchange with cards which can address 32bits.
- Zone Normal: all further memory
  - is used for processes
- Zone HiMem (historical for 32bit systems) - memory that is not permanently mapped into kernel’s address space
- After booting, ‘dmesg’ shows us the physical addresses of the Zones



Non-Uniform Memory Access (NUMA) & UMA

- Physical memory is split in between multiple CPU (core)
- The memory is arranged into banks that have different access latency depending on the “distance” from the processor.
- Each bank is referred to as a *node* and for each node Linux constructs an independent memory management subsystem.
- A node has its own set of zones, lists of free and used pages and various statistics counters.
- There is single physical address space accessible from every node
- Access to the local (node) memory is faster

Page cache



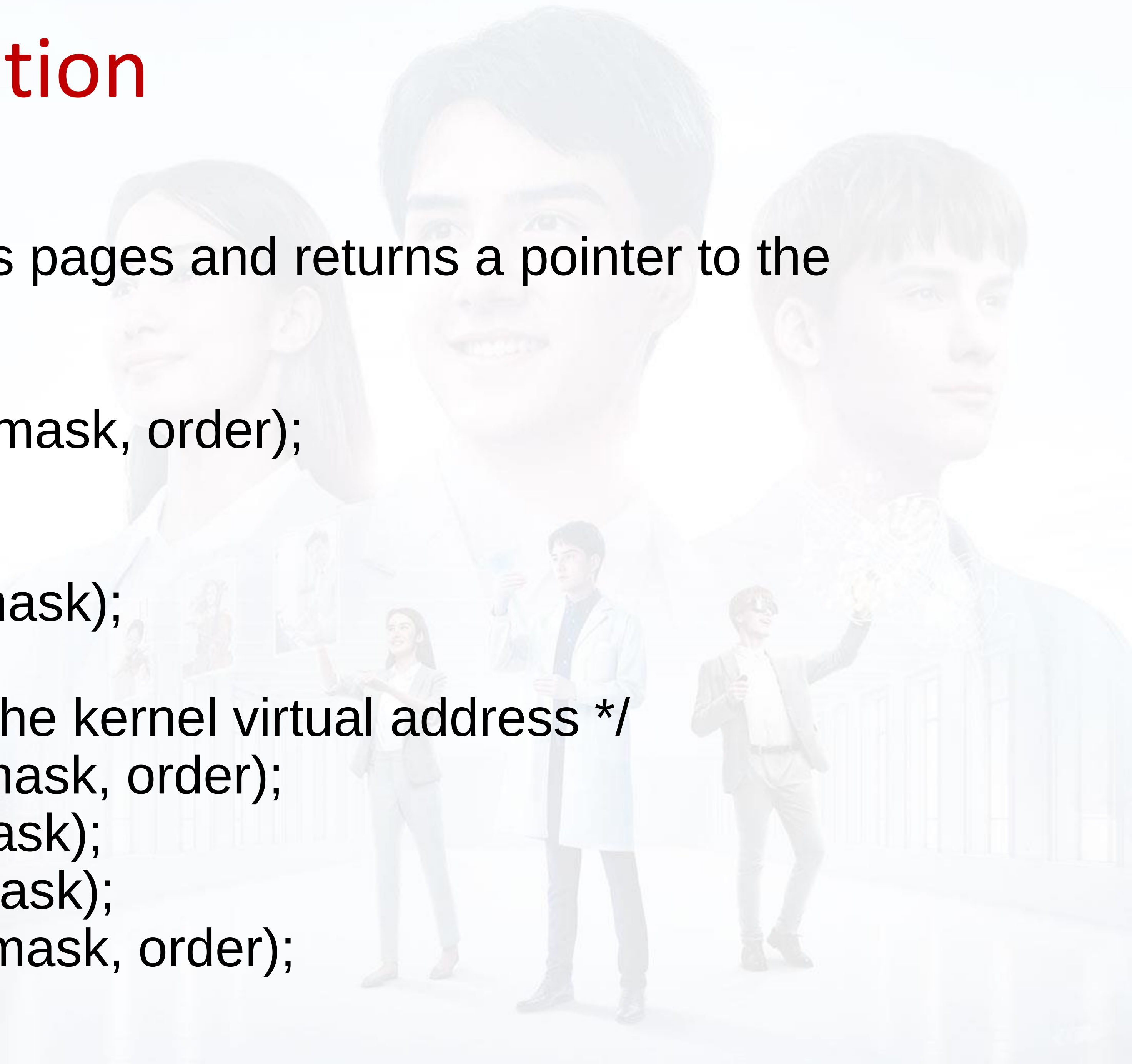
- The physical memory is volatile and the common case for getting data into the memory is to read it from files.

- Whenever a file is read, the data is put into the page cache to avoid expensive disk access on the subsequent reads.

- Similarly, when one writes to a file, the data is placed in the page cache and eventually gets into the backing storage device.

- The written pages are marked as dirty and when Linux decides to reuse them for other purposes, it makes sure to synchronize the file contents on the device with the updated data.

Physical page allocation

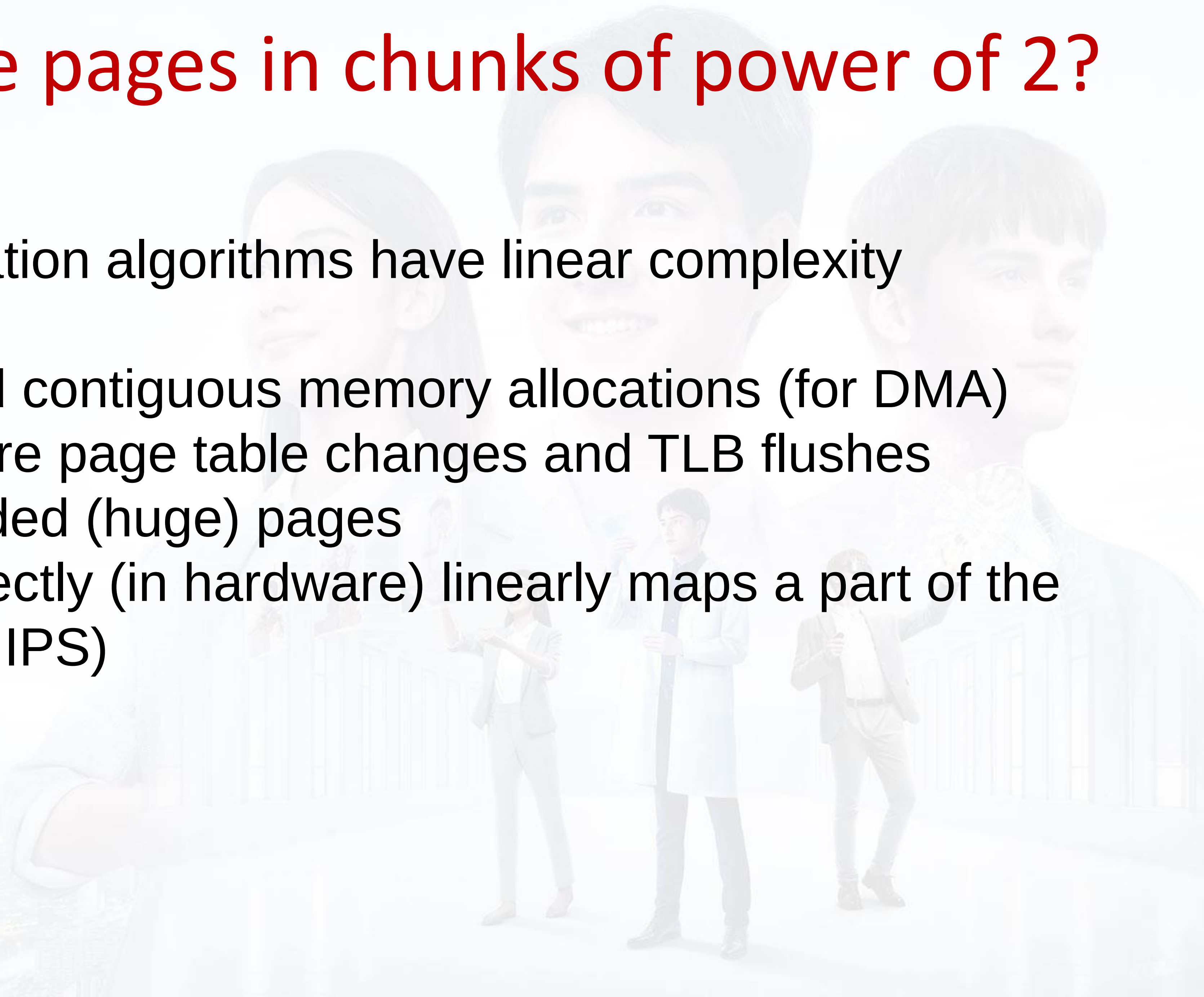


- /* Allocates 2^order contiguous pages and returns a pointer to the
- * descriptor for the first page
- */
- struct page *alloc_pages(gfp_mask, order);

- /* allocates a single page */
- struct page *alloc_page(gfp_mask);

- /* helper functions that return the kernel virtual address */
- void *__get_free_pages(gfp_mask, order);
- void *__get_free_page(gfp_mask);
- void *__get_zero_page(gfp_mask);
- void *__get_dma_pages(gfp_mask, order);

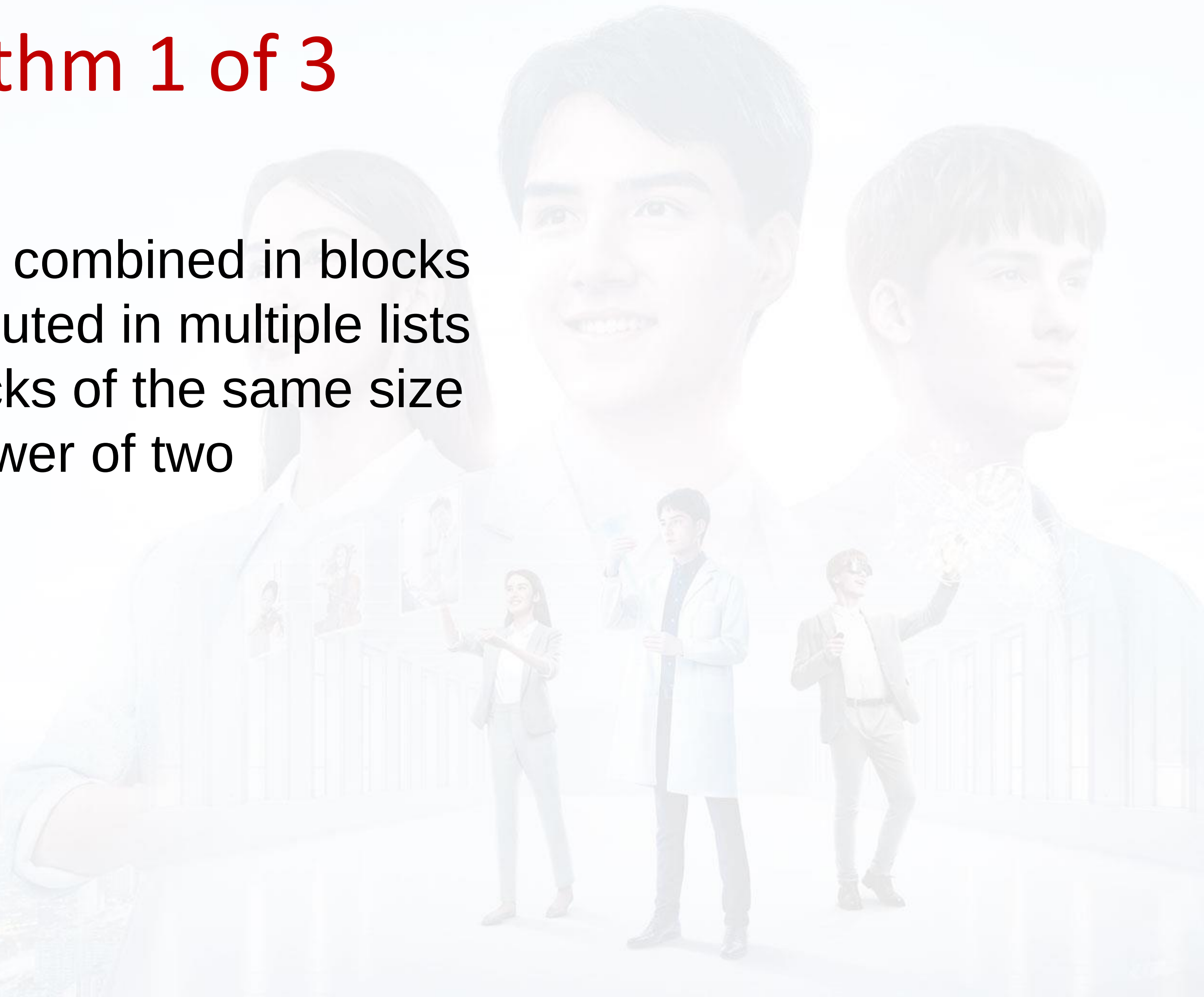
Why only allocate pages in chunks of power of 2?



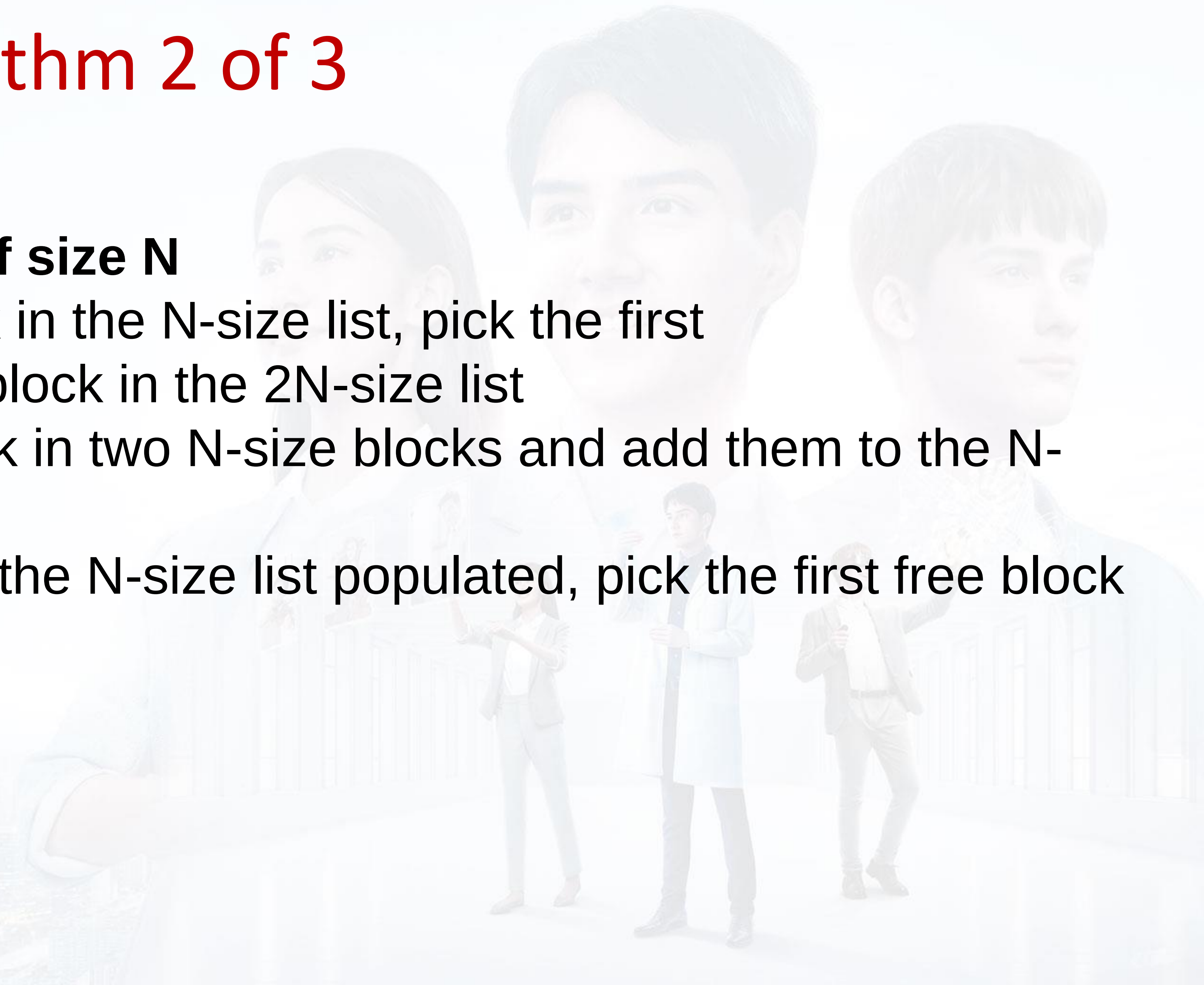
- Typical memory allocation algorithms have linear complexity
- Why not use paging?
- Sometime we do need contiguous memory allocations (for DMA)
- Allocation would require page table changes and TLB flushes
- Not able to use extended (huge) pages
- Some architecture directly (in hardware) linearly maps a part of the address space (e.g. MIPS)

The buddy algorithm 1 of 3

- Contiguous pages are combined in blocks
- Free blocks are distributed in multiple lists
- Each list contains blocks of the same size
- The block size is a power of two



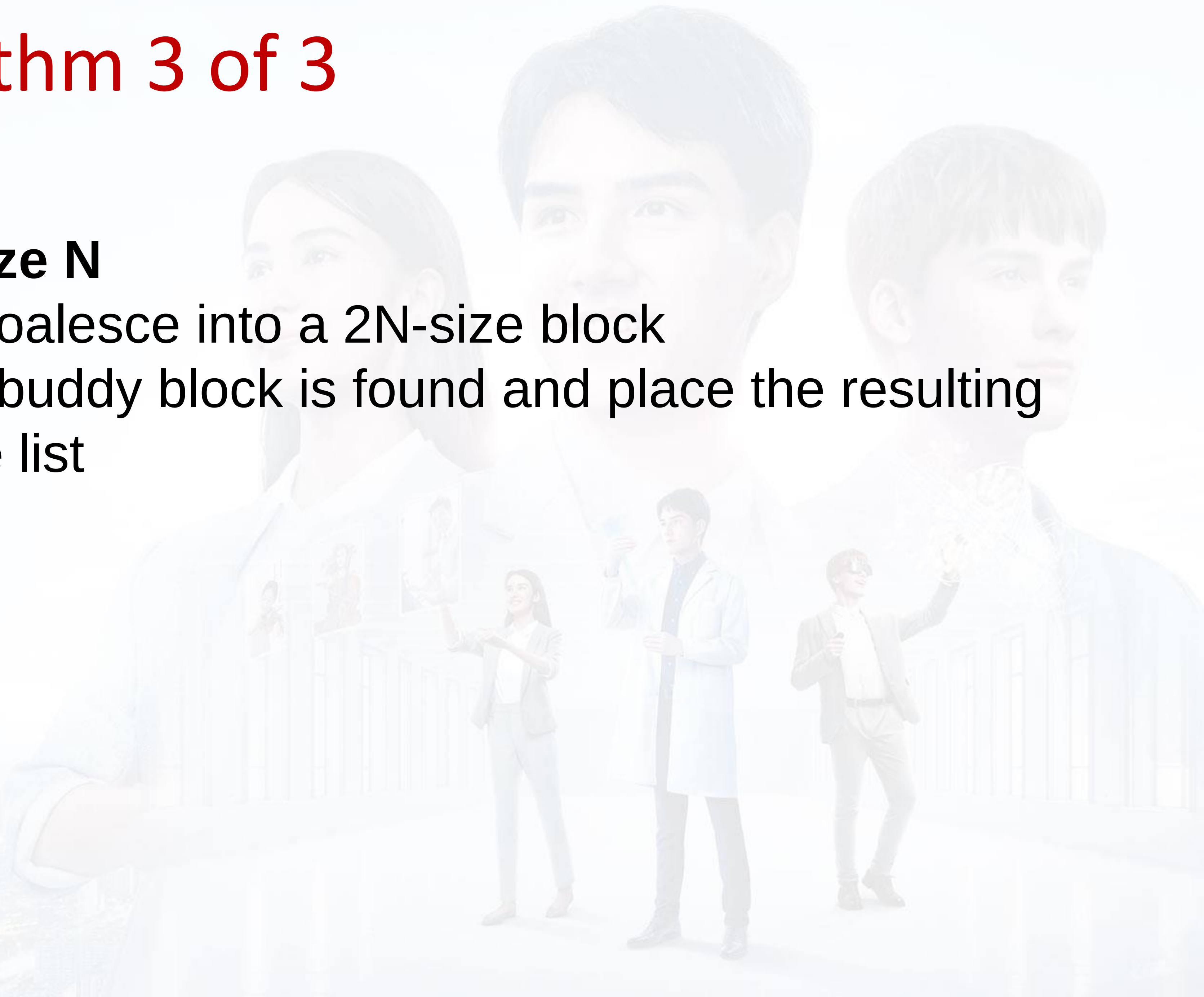
The buddy algorithm 2 of 3



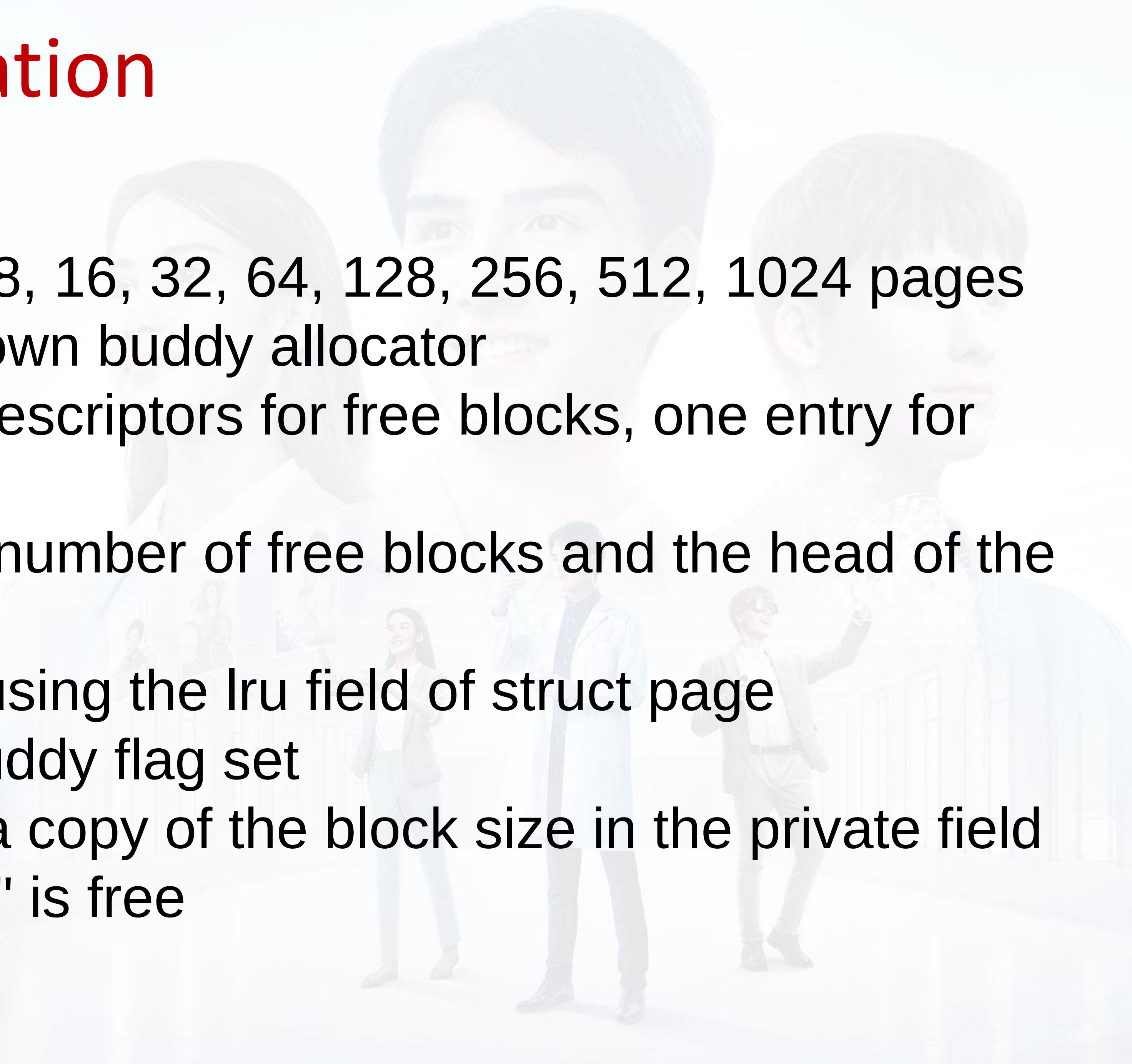
- **Allocating a block of size N**
- If there is a free block in the N-size list, pick the first
- If not, look for a free block in the 2N-size list
- Split the 2N-size block in two N-size blocks and add them to the N-size block
  - Now that we have the N-size list populated, pick the first free block from that list

The buddy algorithm 3 of 3

- **Freeing a block of size N**
- If the "buddy" is free coalesce into a 2N-size block
- Try until no more free buddy block is found and place the resulting block in the respective list

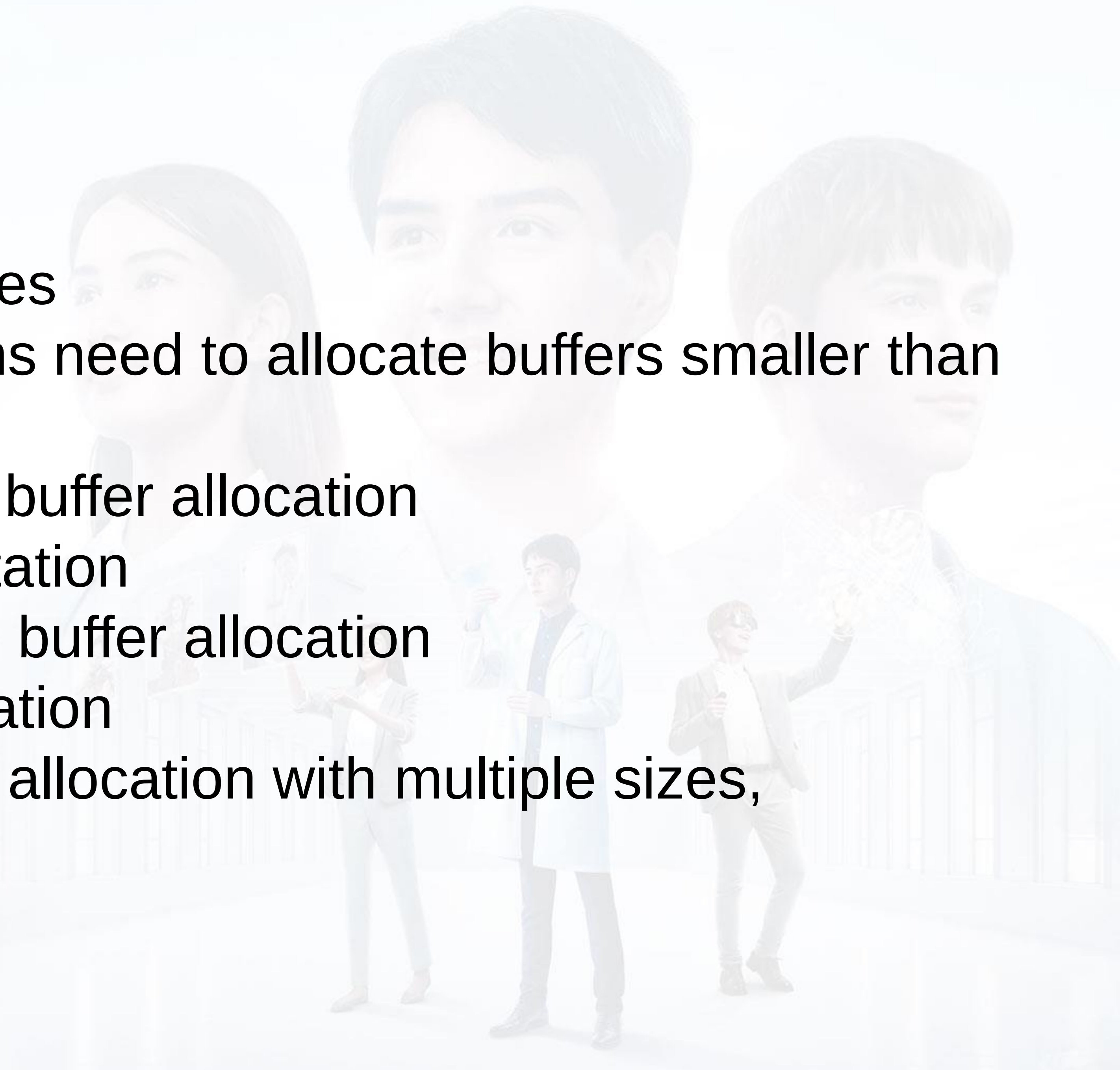


The linux implementation

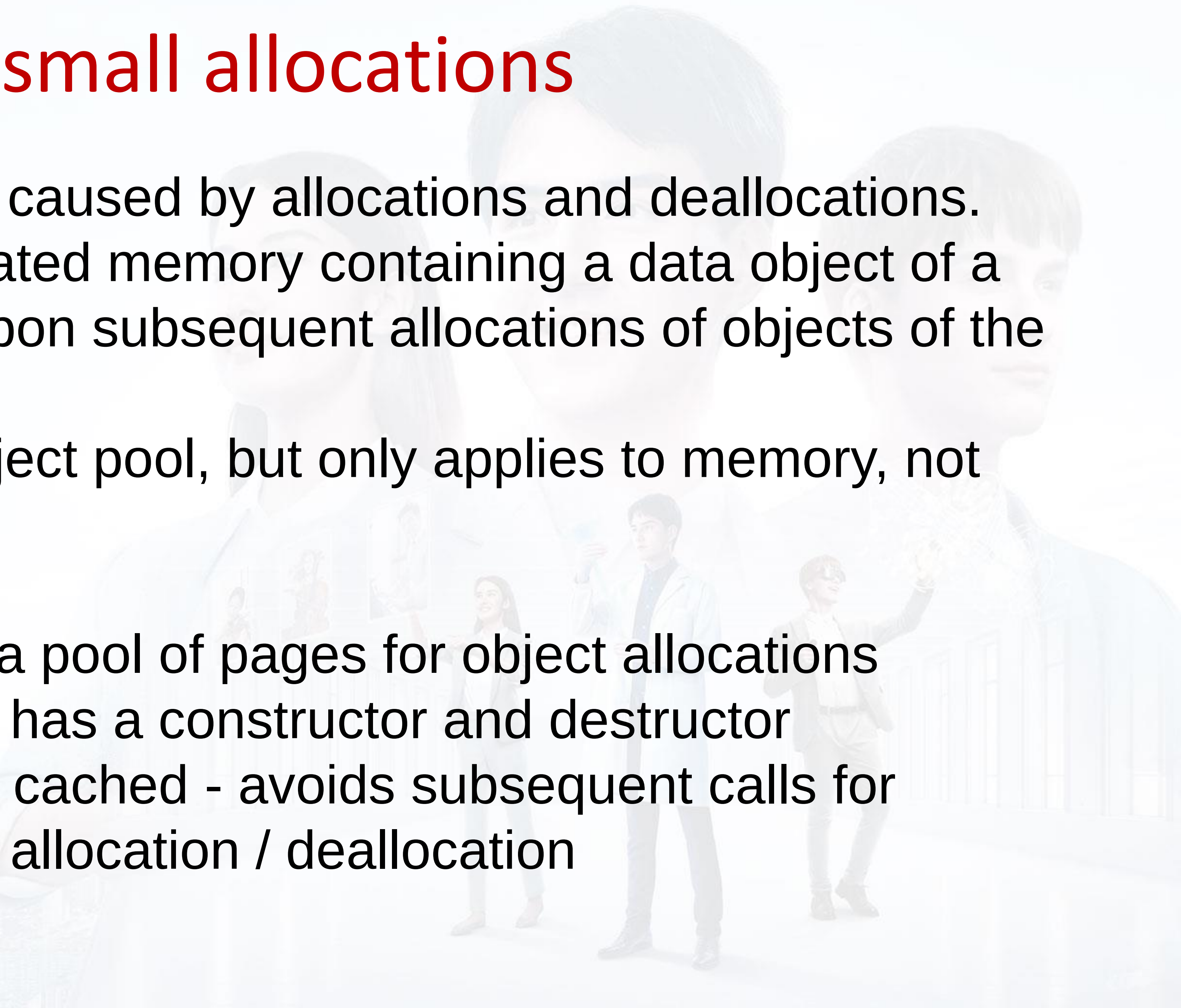


- 11 lists for blocks of 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 pages
- Each memory zone has its own buddy allocator
- Each zone has a vector of descriptors for free blocks, one entry for each size
- The descriptor contains the number of free blocks and the head of the list
- Blocks are linked in the list using the lru field of struct page
- Free pages have the PG_buddy flag set
- The page descriptor keeps a copy of the block size in the private field to easily check if the "buddy" is free

Small allocations



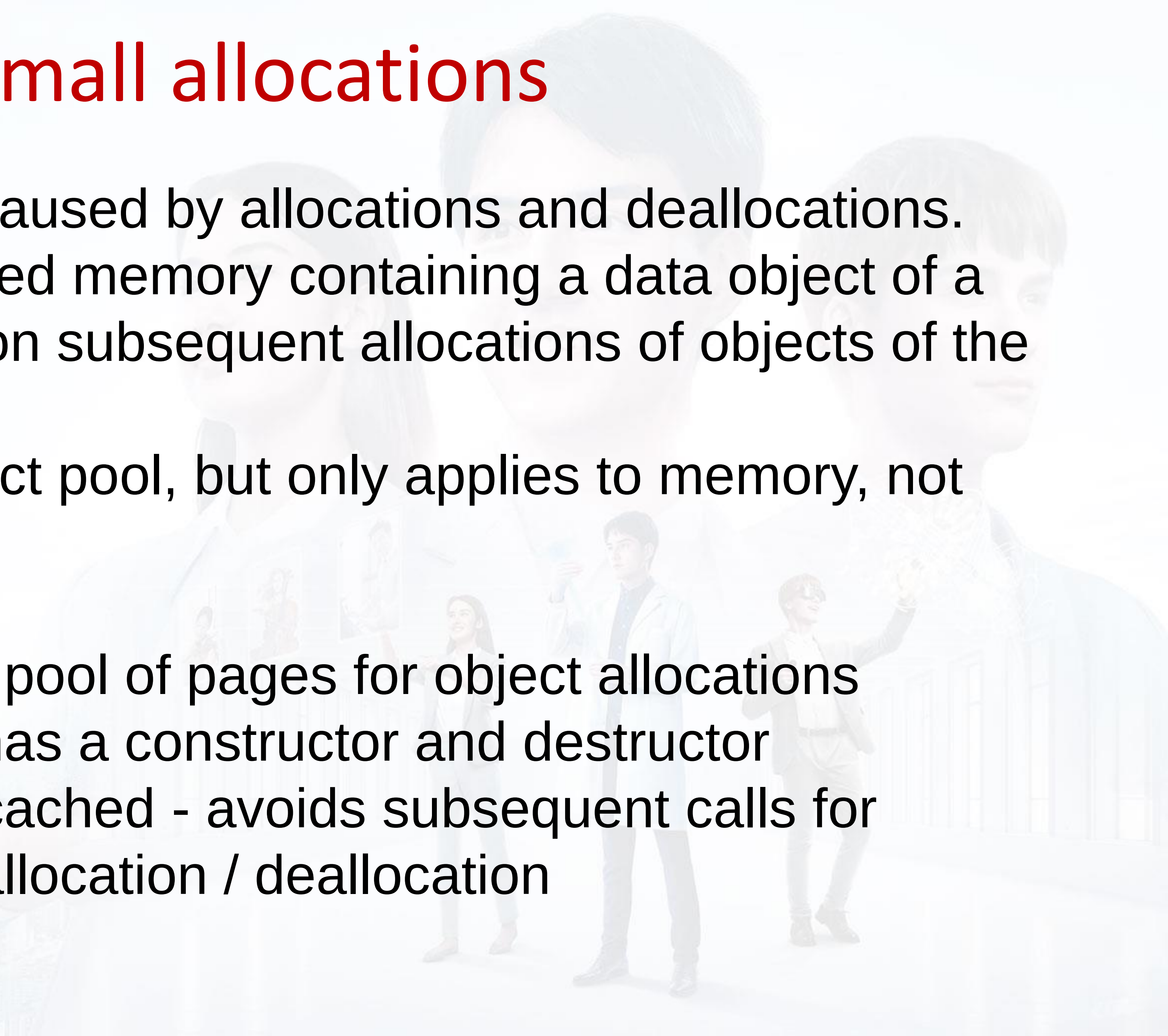
- Buddy is used to allocate pages
- Many of the kernel subsystems need to allocate buffers smaller than a page
- Typical solution: variable size buffer allocation
  - Leads to external fragmentation
- Alternative solution: fixed size buffer allocation
  - Leads to internal fragmentation
- Compromise: fixed size block allocation with multiple sizes, logarithmically distributed
  - e.g.: 32, 64, ..., 131056



SLAB allocator for small allocations

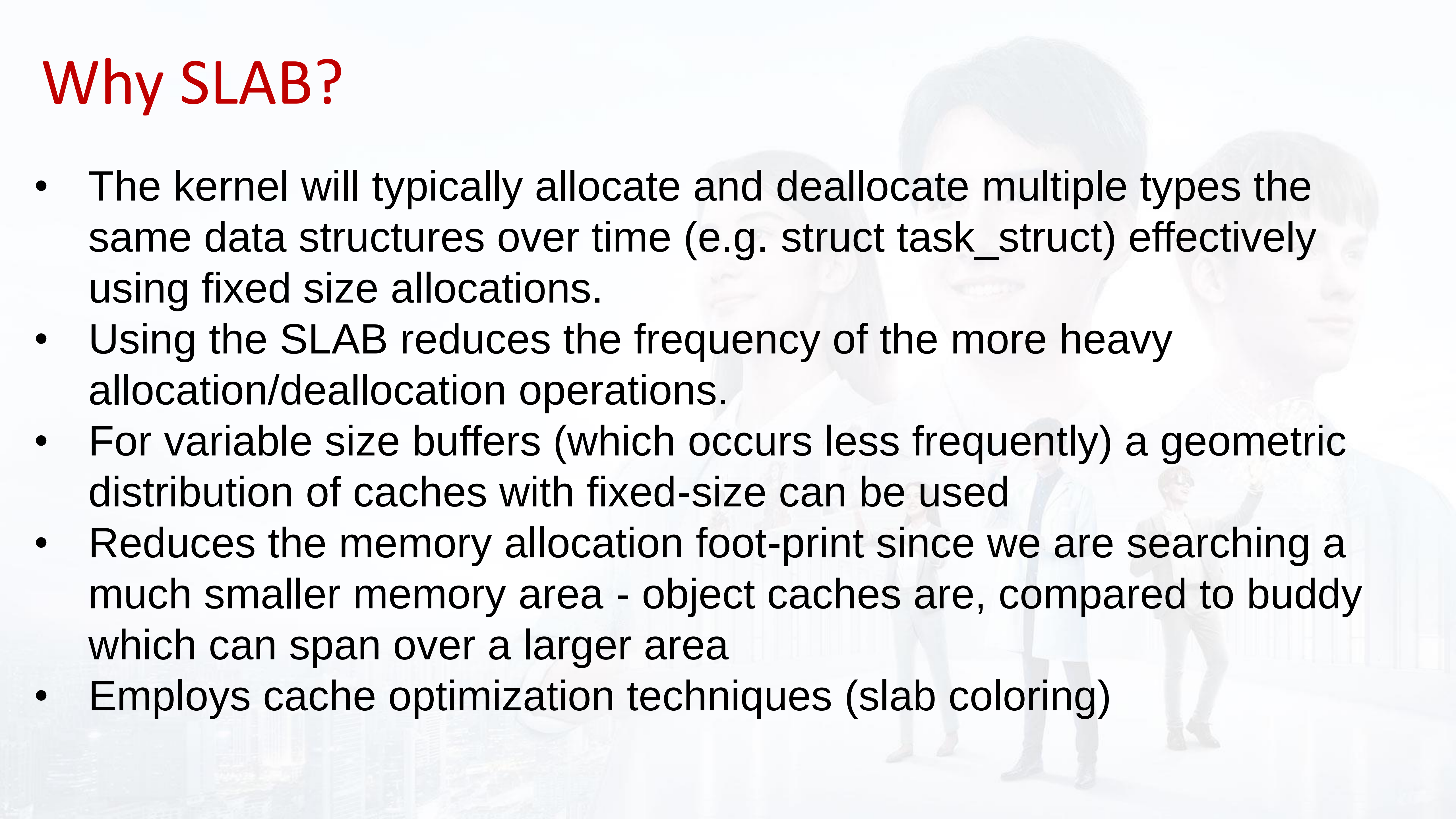
- Reduces fragmentation caused by allocations and deallocations.
- Used for retaining allocated memory containing a data object of a certain type for reuse upon subsequent allocations of objects of the same type.
- It is analogous to an object pool, but only applies to memory, not other resources.
- Buffers = objects
- Uses buddy to allocate a pool of pages for object allocations
- Each object (optionally) has a constructor and destructor
- Deallocated objects are cached - avoids subsequent calls for constructors and buddy allocation / deallocation

SLAB allocator for small allocations



- Reduces fragmentation caused by allocations and deallocations.
- Used for retaining allocated memory containing a data object of a certain type for reuse upon subsequent allocations of objects of the same type.
- It is analogous to an object pool, but only applies to memory, not other resources.
- Buffers = objects
- Uses buddy to allocate a pool of pages for object allocations
- Each object (optionally) has a constructor and destructor
- Deallocated objects are cached - avoids subsequent calls for constructors and buddy allocation / deallocation

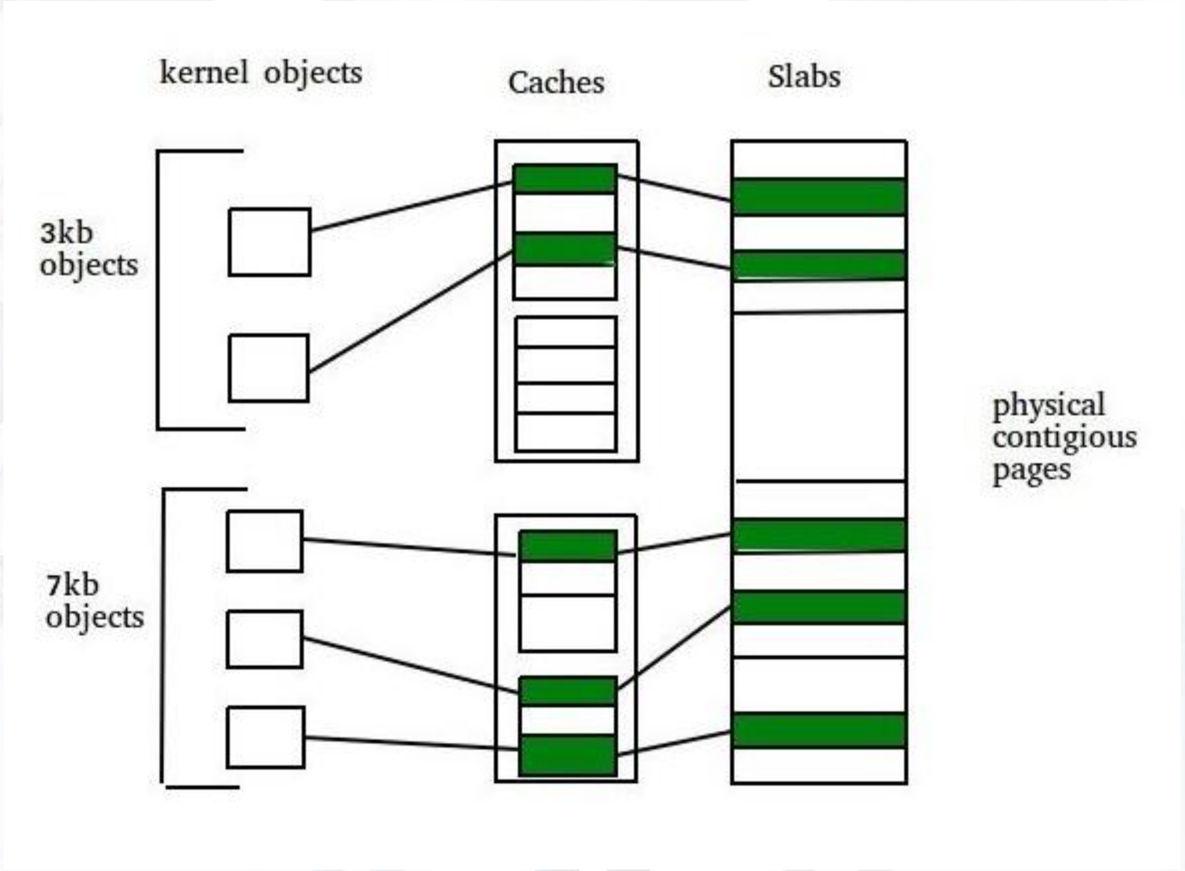
Why SLAB?



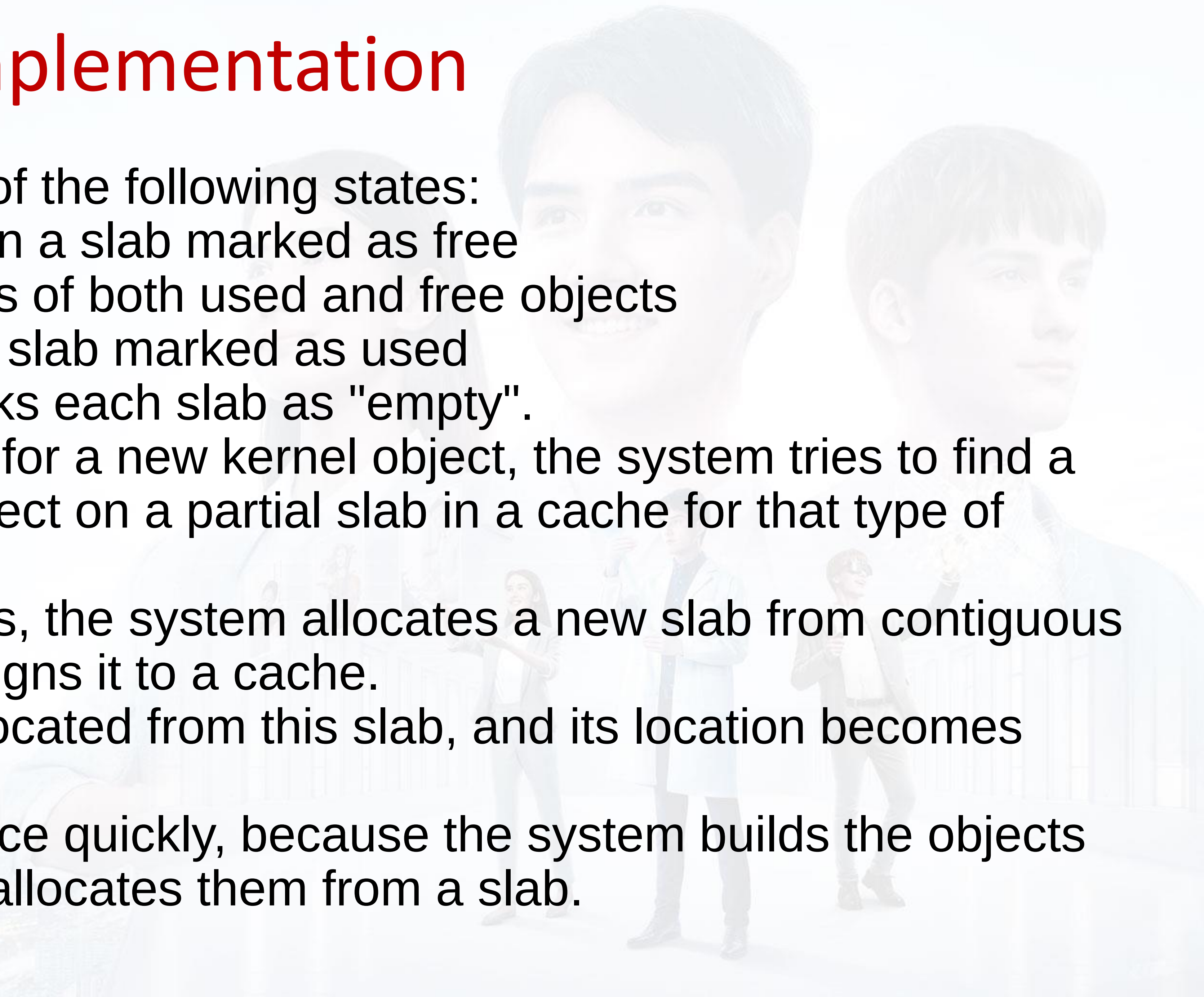
- The kernel will typically allocate and deallocate multiple types the same data structures over time (e.g. struct task_struct) effectively using fixed size allocations.
- Using the SLAB reduces the frequency of the more heavy allocation/deallocation operations.
- For variable size buffers (which occurs less frequently) a geometric distribution of caches with fixed-size can be used
- Reduces the memory allocation foot-print since we are searching a much smaller memory area - object caches are, compared to buddy which can span over a larger area
- Employs cache optimization techniques (slab coloring)

SLAB architecture

- Cache: a small amount of very fast memory. A cache is a storage for a specific type of object, such as semaphores, process descriptors, file objects, etc.
- Slab: a contiguous piece of memory, usually made of several physically contiguous pages.
- The slab is the actual container of data associated with objects of the specific kind of the containing cache.
- Cache, setup: allocate a number of objects to the slabs associated with that cache. This number depends on the size of the associated slabs.



SLAB allocator implementation

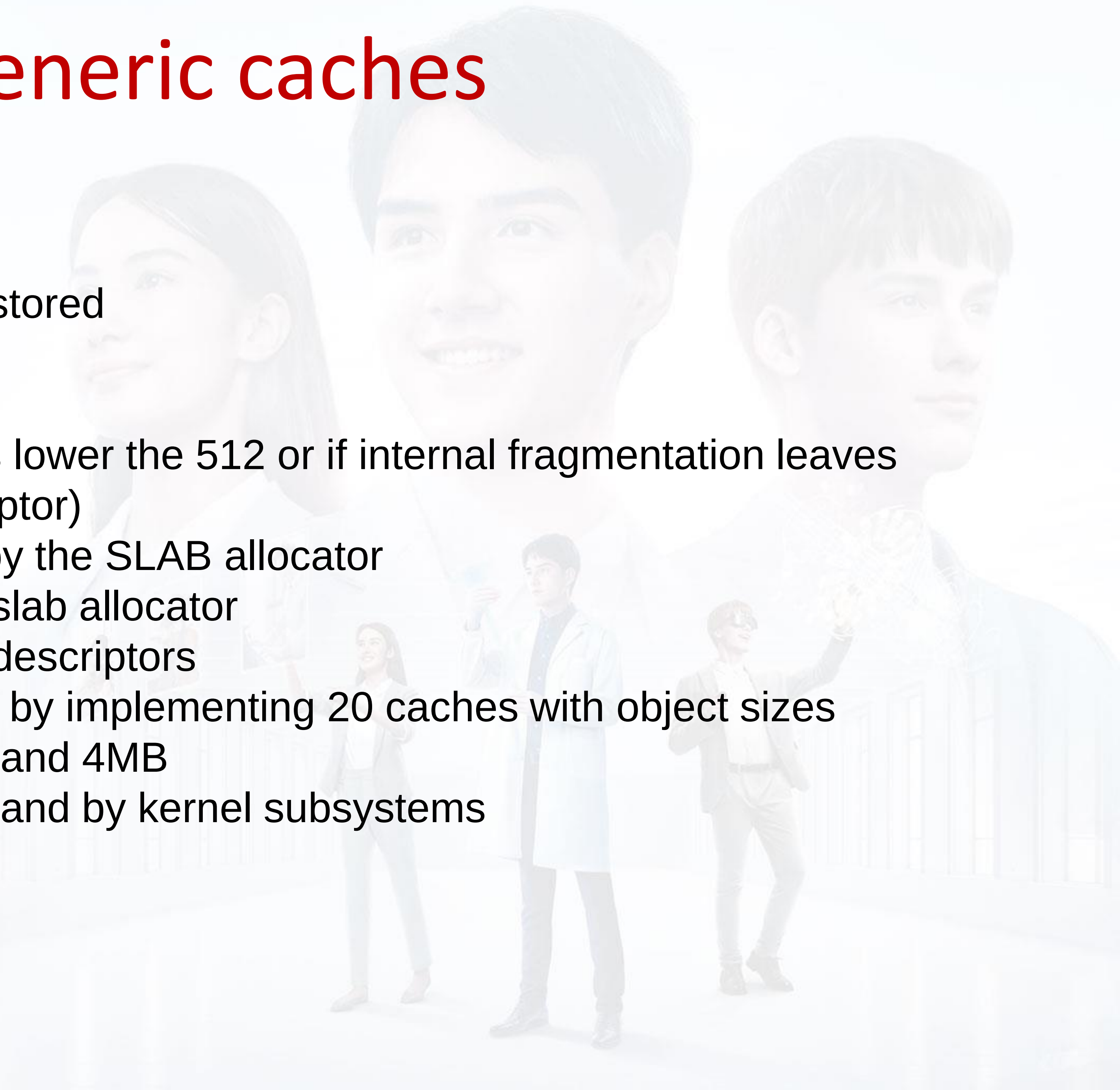


- Slabs may exist in one of the following states:
  - empty – all objects on a slab marked as free
  - partial – slab consists of both used and free objects
  - full – all objects on a slab marked as used
- Initially, the system marks each slab as "empty".
- When the process calls for a new kernel object, the system tries to find a free location for that object on a partial slab in a cache for that type of object.
- If no such location exists, the system allocates a new slab from contiguous physical pages and assigns it to a cache.
- The new object gets allocated from this slab, and its location becomes marked as "partial".
- The allocation takes place quickly, because the system builds the objects in advance and readily allocates them from a slab.

Cache descriptor

- A name to identify the cache for stats
- object constructor and destructor functions
- size of the objects
- Flags
- Size of the slab in power of 2 pages
- GFP (Get Free Pages) masks
- One or mores slabs, grouped by state: full, partially full, empty

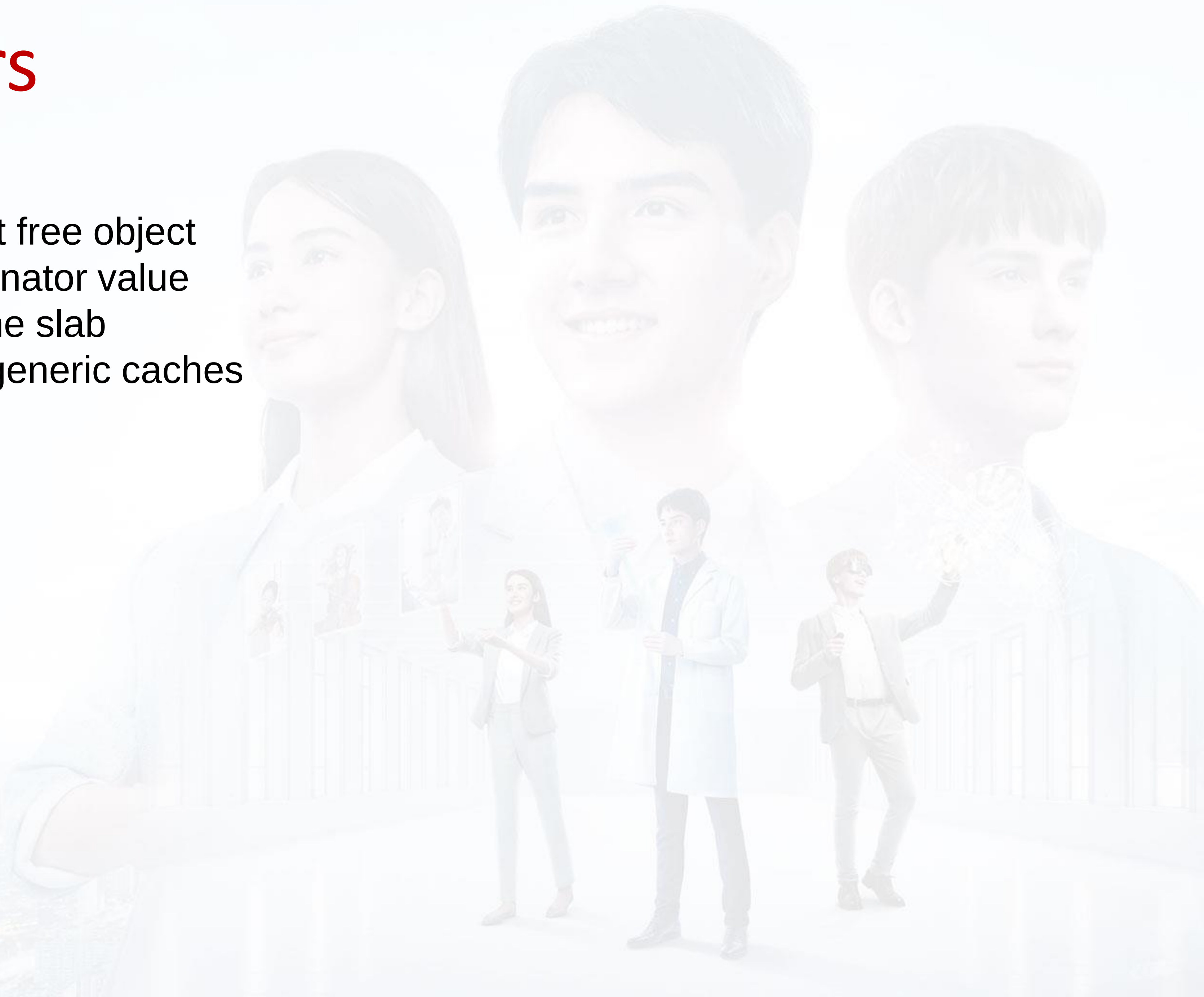
Slab descriptors and generic caches



- Slab descriptor:
  - Number of objects
  - Memory region where the objects are stored
  - Pointer to the first free object
  - Descriptor are stored either in
    - the SLAB itself (if the object size is lower the 512 or if internal fragmentation leaves enough space for the SLAB descriptor)
    - in generic caches internally used by the SLAB allocator
- Generic caches are used internally by the slab allocator
  - allocating memory for cache and slab descriptors
- They are also used to implement kmalloc() by implementing 20 caches with object sizes geometrically distributed between 32bytes and 4MB
- There are specific caches, created on demand by kernel subsystems

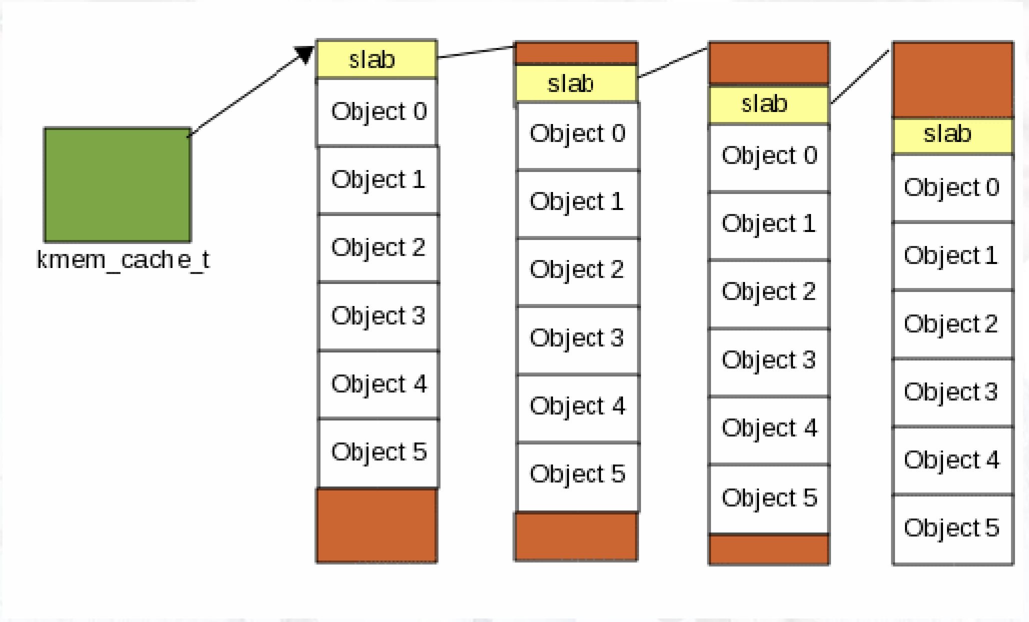
Object descriptors

- Only used for free objects
- An integer that points to the next free object
- The last free object uses a terminator value
- Internal descriptors - stored in the slab
- External descriptors - stored in generic caches



Cache coloring of slabs

- Cache Coloring is a method to ensure that access to the slabs in kernel memory make the best use of the processor L1 cache.
- This is a performance tweak to try to ensure that we take as few cache hits as possible.
- Since slabs begin on page boundaries, it is likely that the objects within several different slab pages map to the same cache line, called 'false sharing'.
- This leads to less than optimal hardware cache performance.
- By offsetting each beginning of the first object within each slab by some fragment of the hardware cache line size, processor cache hits are reduced.

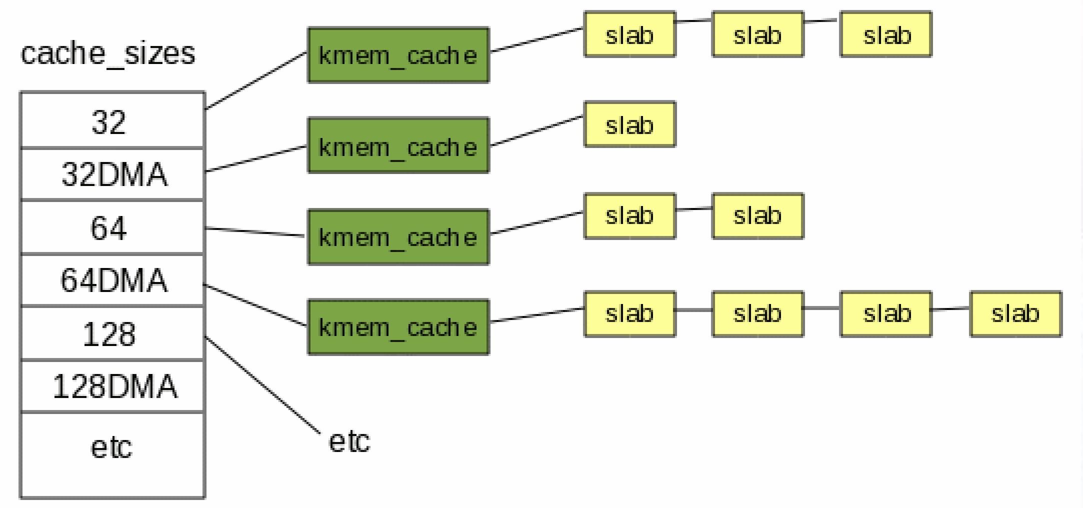


kmalloc() interface

- When a kernel module or driver needs to allocate memory for an object that doesn't fit one of the uniform types of the other caches, for example string buffers, one-off structures, temporary storage, etc.

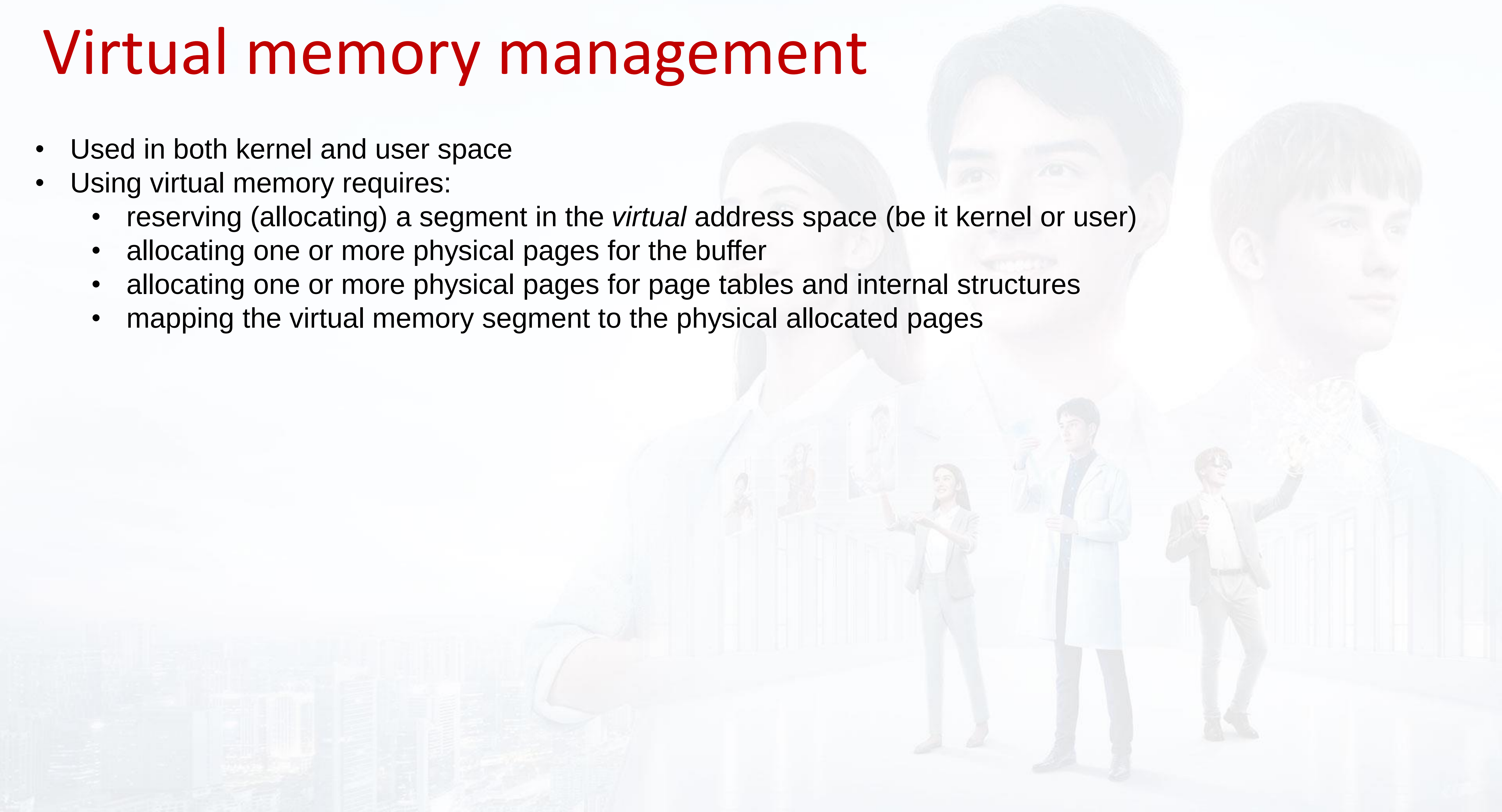
- For those instances drivers and kernel modules use the kmalloc() and kfree() routines.
- The Linux kernel ties these calls into the slab allocator too.

- On initialization, the kernel asks the slab allocator to create several caches of varying sizes for this purpose.
- Caches for generic objects of 32, 64, 128, 256, all the way to 131072 bytes are created for both the GFP_NORMAL and GFP_DMA zones of memory.
- When a kernel module or driver needs memory, the cache_sizes array is searched to find the cache with the size appropriate to fit the requested object.
- For example, if a driver requests 166 bytes of GFP_NORMAL memory through kmalloc(), an object from the 256 byte cache would be returned.



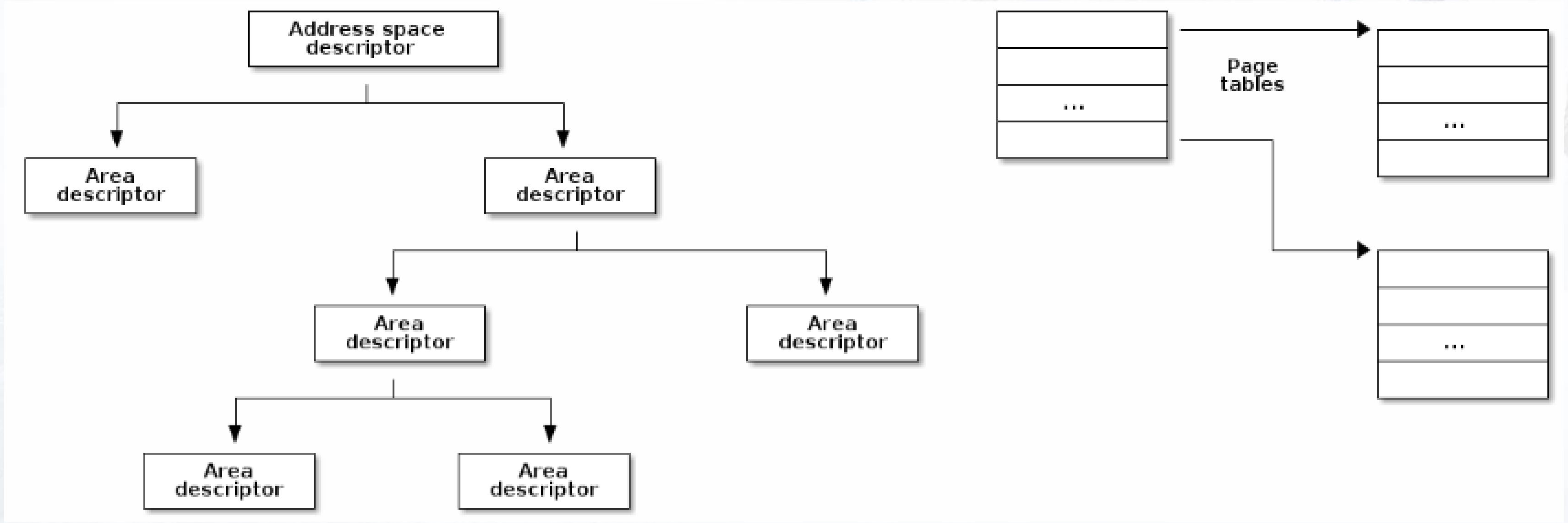
Virtual memory management

- Used in both kernel and user space
- Using virtual memory requires:
  - reserving (allocating) a segment in the *virtual* address space (be it kernel or user)
  - allocating one or more physical pages for the buffer
  - allocating one or more physical pages for page tables and internal structures
  - mapping the virtual memory segment to the physical allocated pages



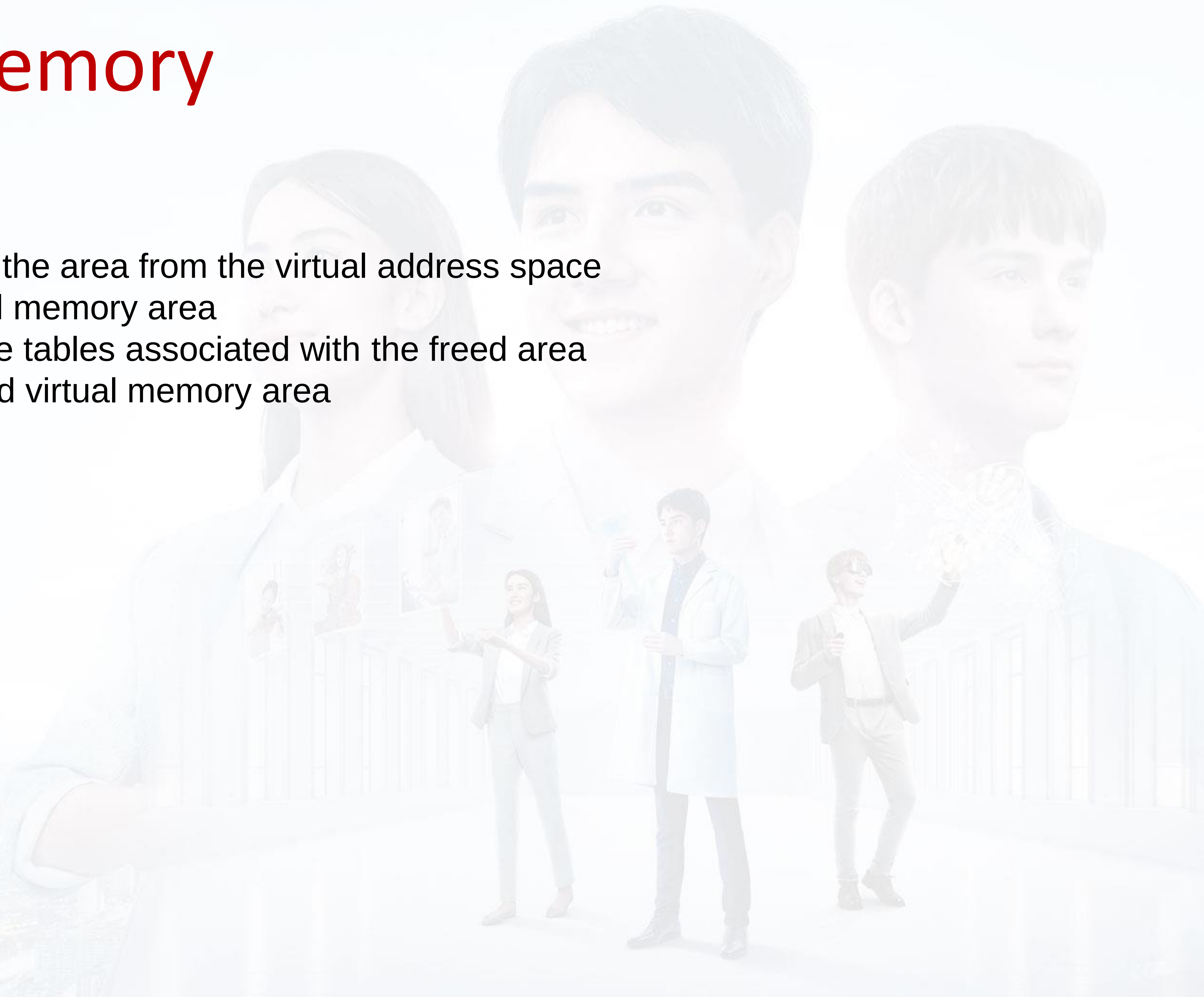
Address space descriptors and VM allocation

- The address space descriptor is used by the kernel to maintain high level information:
  - file and file offset (for mmap with files),
  - read-only segment,
  - copy-on-write segment, etc.
- Page table is used either by:
  - The CPU's MMU
  - The kernel to handle TLB exception
- Allocating VM
  - Search a free area in the address space descriptor
  - Allocate memory for a new area descriptor
  - Insert the new area descriptor in the address space descriptor
  - Allocate physical memory for one or more page tables
  - Setup the page tables for the newly allocated area in the virtual address space
  - Allocating (on demand) physical pages and map them in the virtual address space by updating the page tables



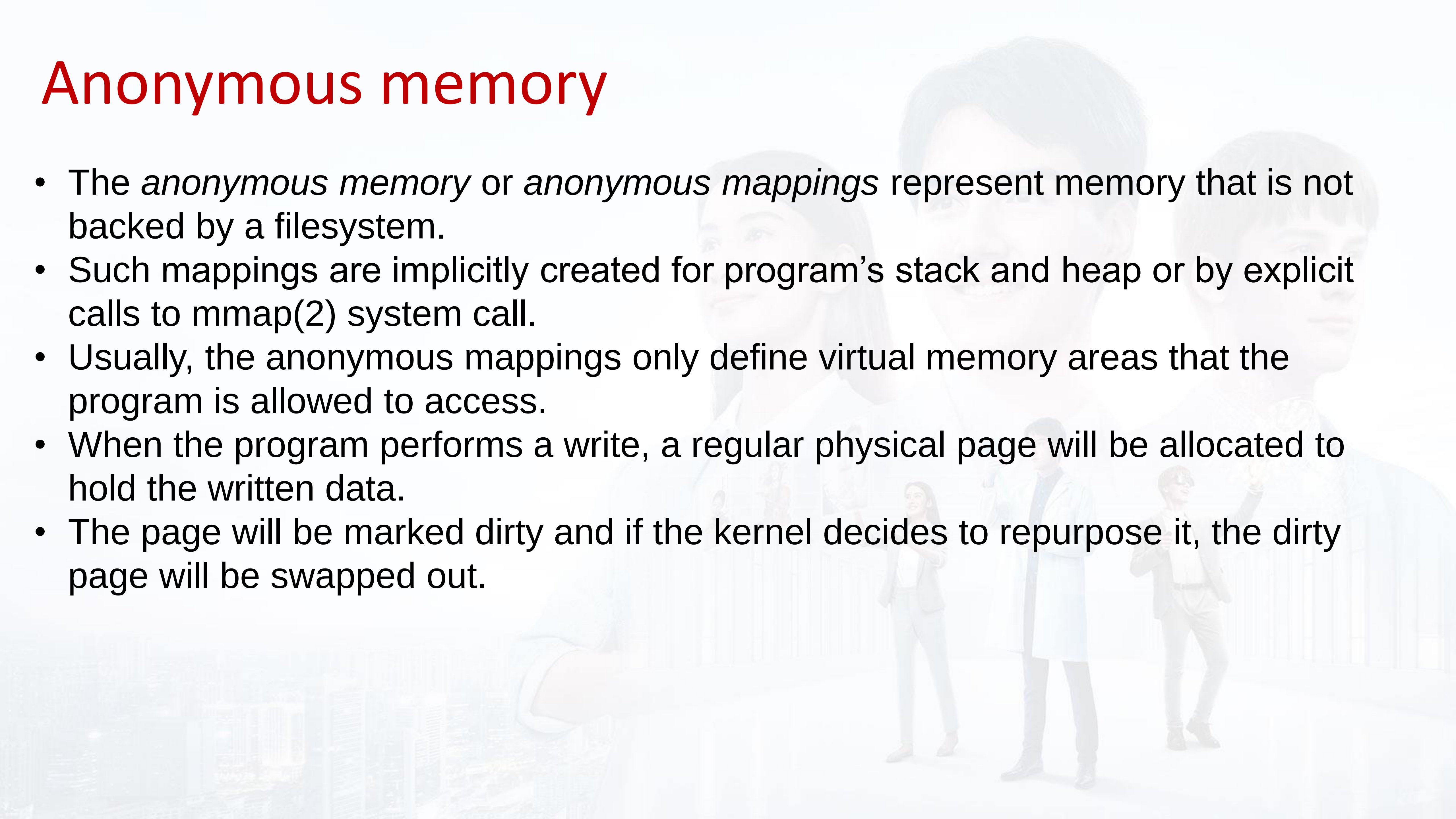
Freeing virtual memory

- Removing the area descriptor
- Freeing the area descriptor memory
- Updating the page tables to remove the area from the virtual address space
- Flushing the TLB for the freed virtual memory area
- Freeing physical memory of the page tables associated with the freed area
- Freeing physical memory of the freed virtual memory area

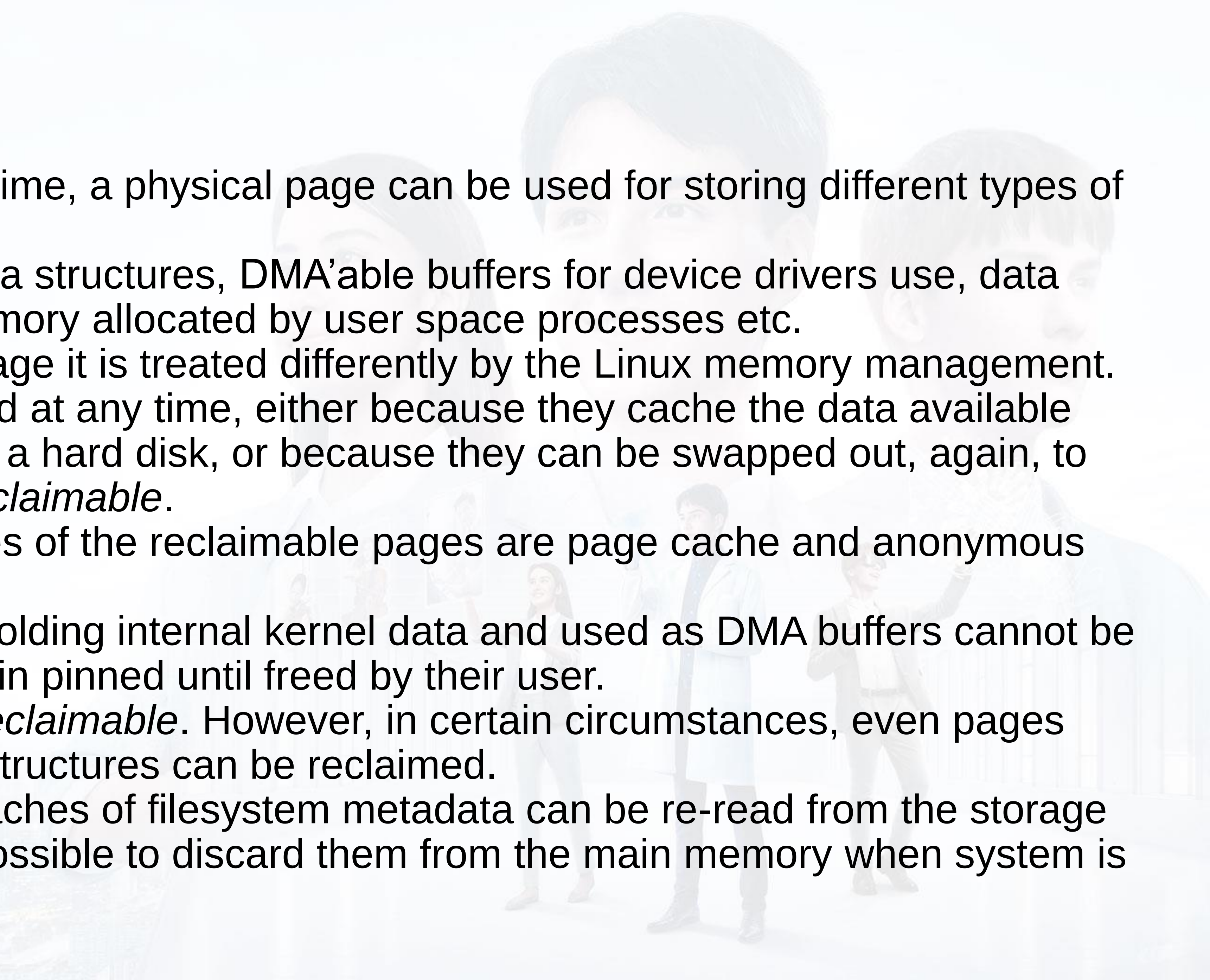


Anonymous memory

- The *anonymous memory* or *anonymous mappings* represent memory that is not backed by a filesystem.
- Such mappings are implicitly created for program’s stack and heap or by explicit calls to mmap(2) system call.
- Usually, the anonymous mappings only define virtual memory areas that the program is allowed to access.
- When the program performs a write, a regular physical page will be allocated to hold the written data.
- The page will be marked dirty and if the kernel decides to repurpose it, the dirty page will be swapped out.

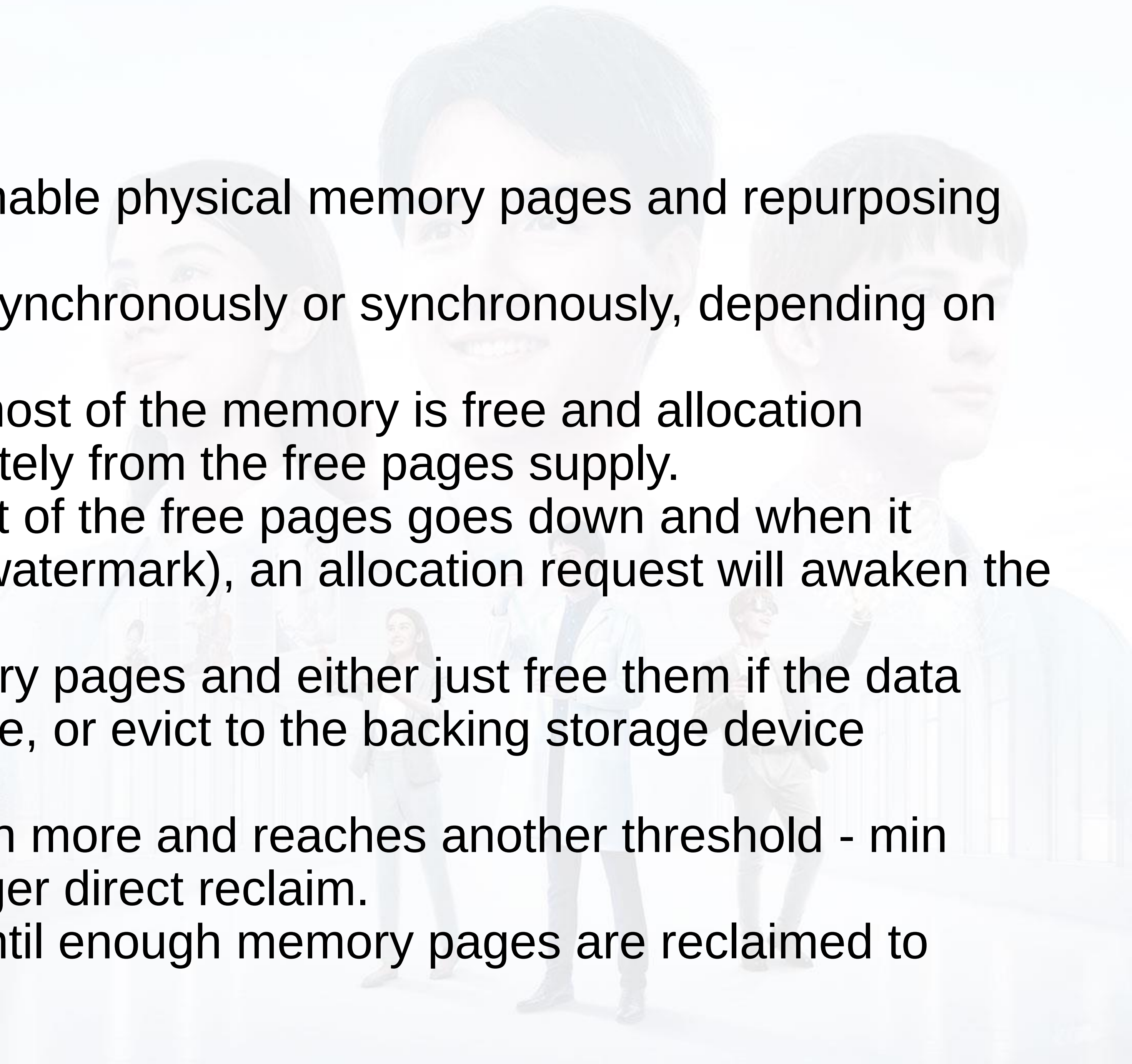


Reclaim 1 of 2



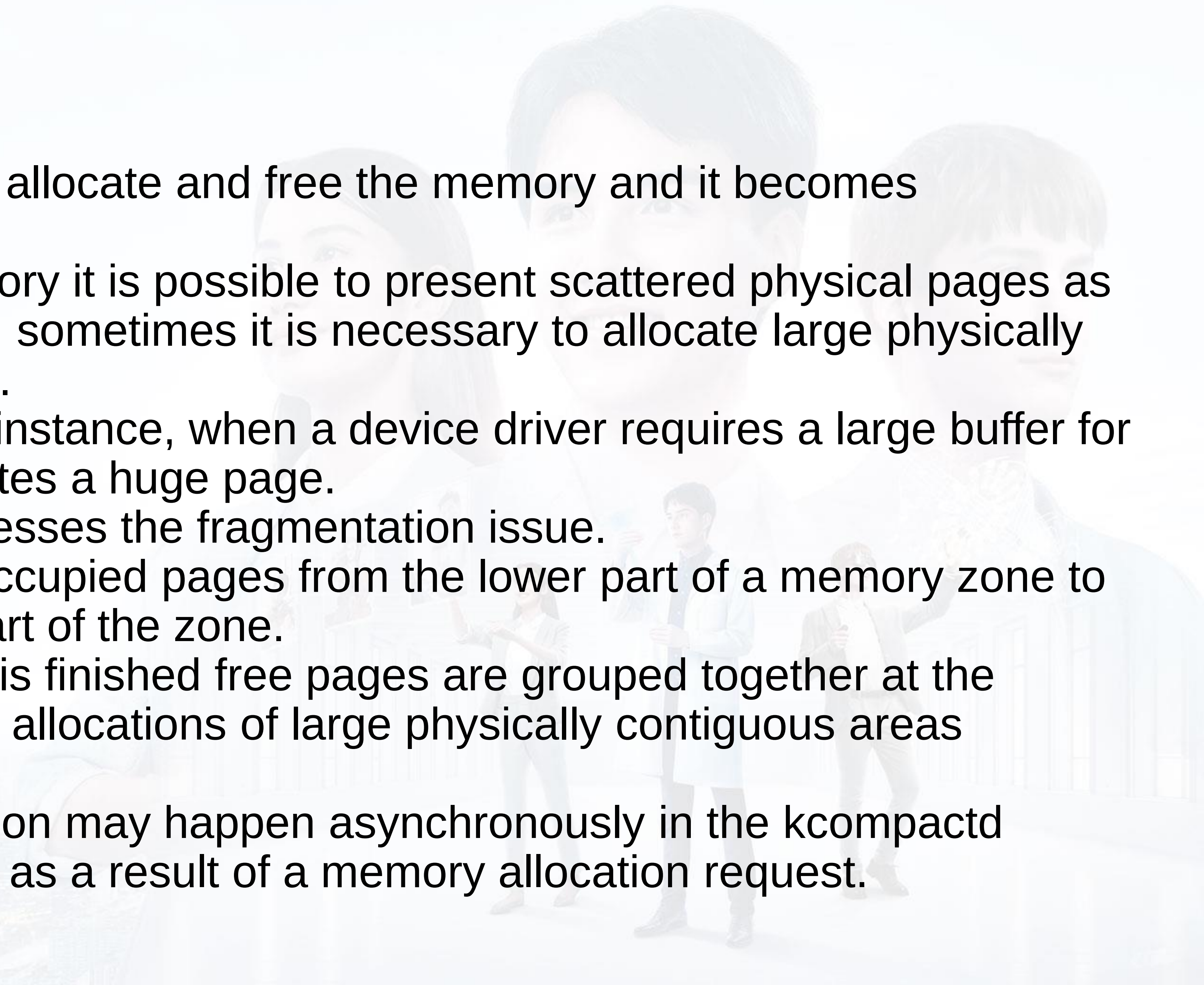
- Throughout the system lifetime, a physical page can be used for storing different types of data.
- It can be kernel internal data structures, DMA’able buffers for device drivers use, data read from a filesystem, memory allocated by user space processes etc.
- Depending on the page usage it is treated differently by the Linux memory management.
- The pages that can be freed at any time, either because they cache the data available elsewhere, for instance, on a hard disk, or because they can be swapped out, again, to the hard disk, are called *reclaimable*.
- The most notable categories of the reclaimable pages are page cache and anonymous memory.
- In most cases, the pages holding internal kernel data and used as DMA buffers cannot be repurposed, and they remain pinned until freed by their user.
- Such pages are called *unreclaimable*. However, in certain circumstances, even pages occupied with kernel data structures can be reclaimed.
- For instance, in-memory caches of filesystem metadata can be re-read from the storage device and therefore it is possible to discard them from the main memory when system is under memory pressure.

Reclaim 2 of 2



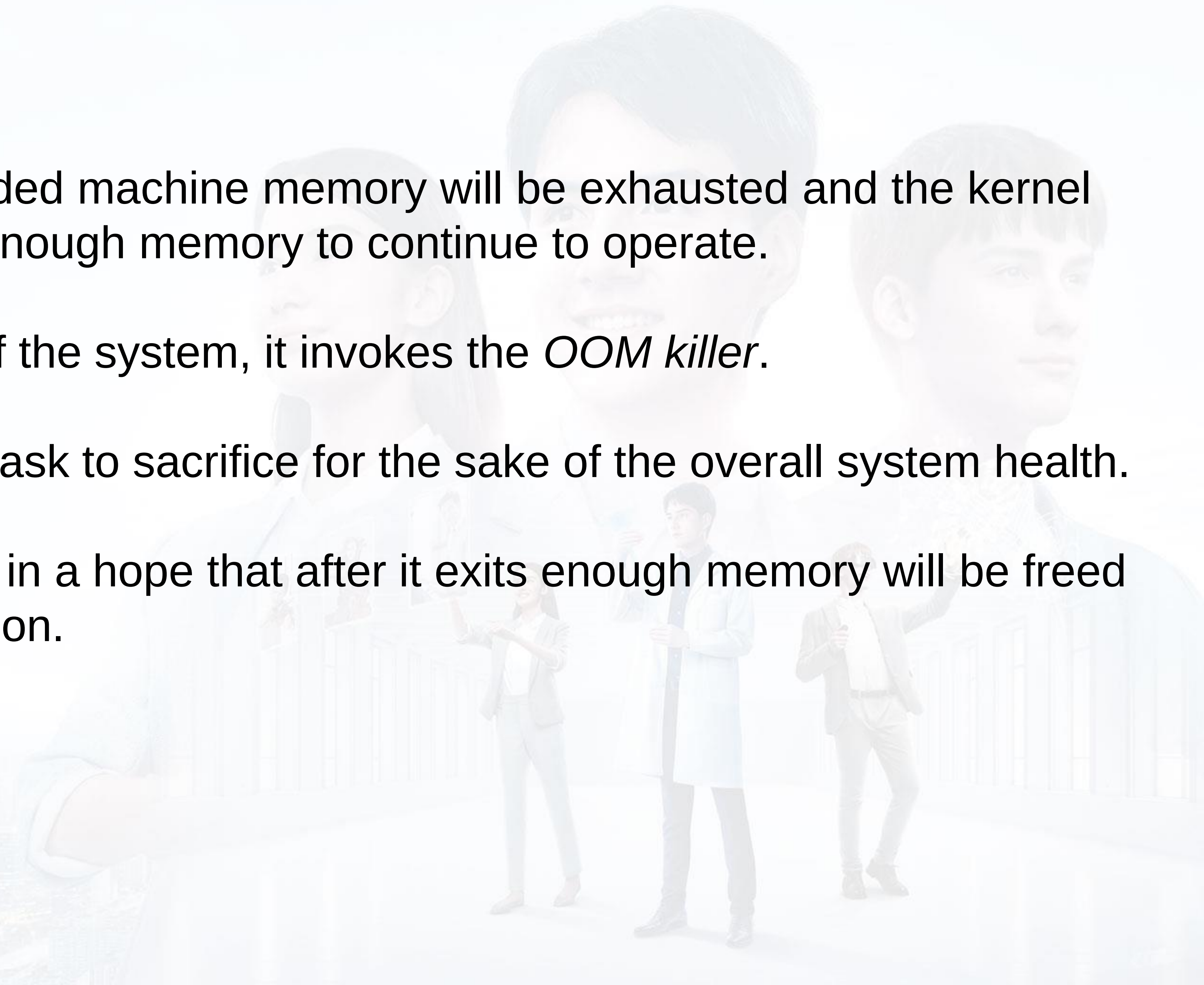
- The process of freeing the reclaimable physical memory pages and repurposing them is called (surprise!) reclaim.
- Linux can reclaim pages either asynchronously or synchronously, depending on the state of the system.
- When the system is not loaded, most of the memory is free and allocation requests will be satisfied immediately from the free pages supply.
- As the load increases, the amount of the free pages goes down and when it reaches a certain threshold (low watermark), an allocation request will awaken the kswapd daemon.
- It will asynchronously scan memory pages and either just free them if the data they contain is available elsewhere, or evict to the backing storage device (remember those dirty pages?).
- As memory usage increases even more and reaches another threshold - min watermark - an allocation will trigger direct reclaim.
- In this case allocation is stalled until enough memory pages are reclaimed to satisfy the request.

Compaction



- As the system runs, tasks allocate and free the memory and it becomes fragmented.
- Although with virtual memory it is possible to present scattered physical pages as virtually contiguous range, sometimes it is necessary to allocate large physically contiguous memory areas.
- Such need may arise, for instance, when a device driver requires a large buffer for DMA, or when THP allocates a huge page.
- Memory *compaction* addresses the fragmentation issue.
- This mechanism moves occupied pages from the lower part of a memory zone to free pages in the upper part of the zone.
- When a compaction scan is finished free pages are grouped together at the beginning of the zone and allocations of large physically contiguous areas become possible.
- Like reclaim, the compaction may happen asynchronously in the kcompactd daemon or synchronously as a result of a memory allocation request.

OOM killer



- It is possible that on a loaded machine memory will be exhausted and the kernel will be unable to reclaim enough memory to continue to operate.
- In order to save the rest of the system, it invokes the *OOM killer*.
- The *OOM killer* selects a task to sacrifice for the sake of the overall system health.
- The selected task is killed in a hope that after it exits enough memory will be freed to continue normal operation.