ITMO Advanced OS 2024

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

Lesson 2024.10.16

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

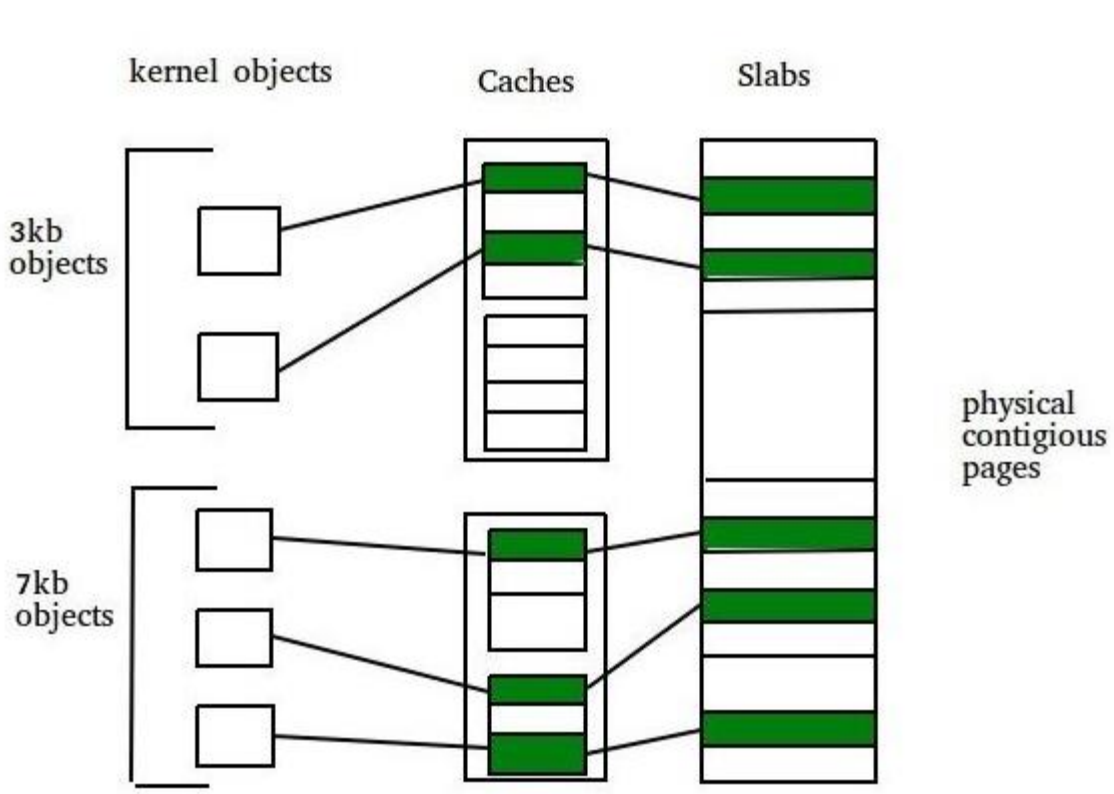
- Reduces fragmentation caused by allocations and deallocations.
- Used for retaining allocated memory containing a data object of a certain type for reuse upon subsequent allocations of objects of the same type.
- It is analogous to an object pool, but only applies to memory, not other resources.
- 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.

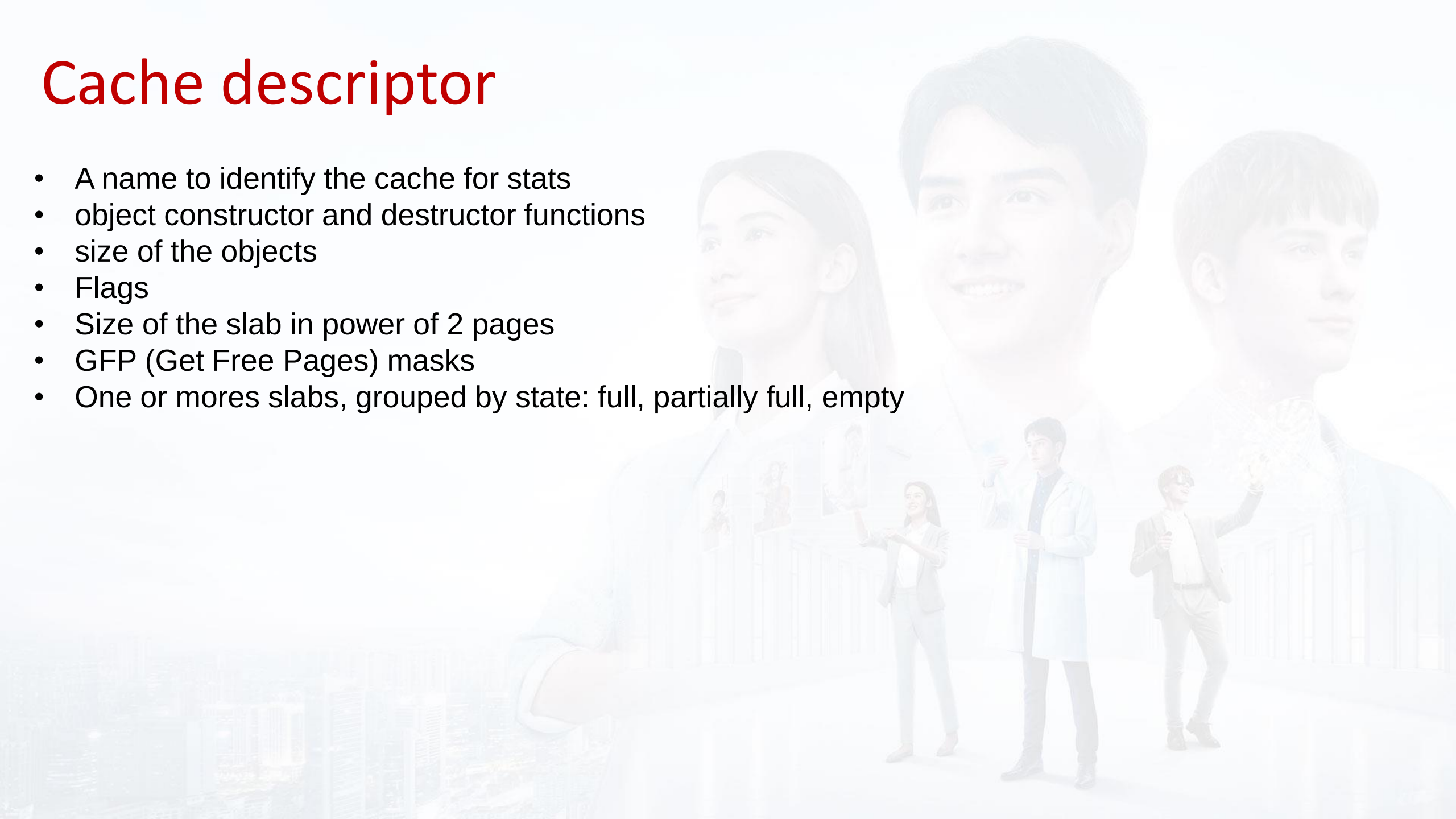


SLAB allocator implementation

- 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

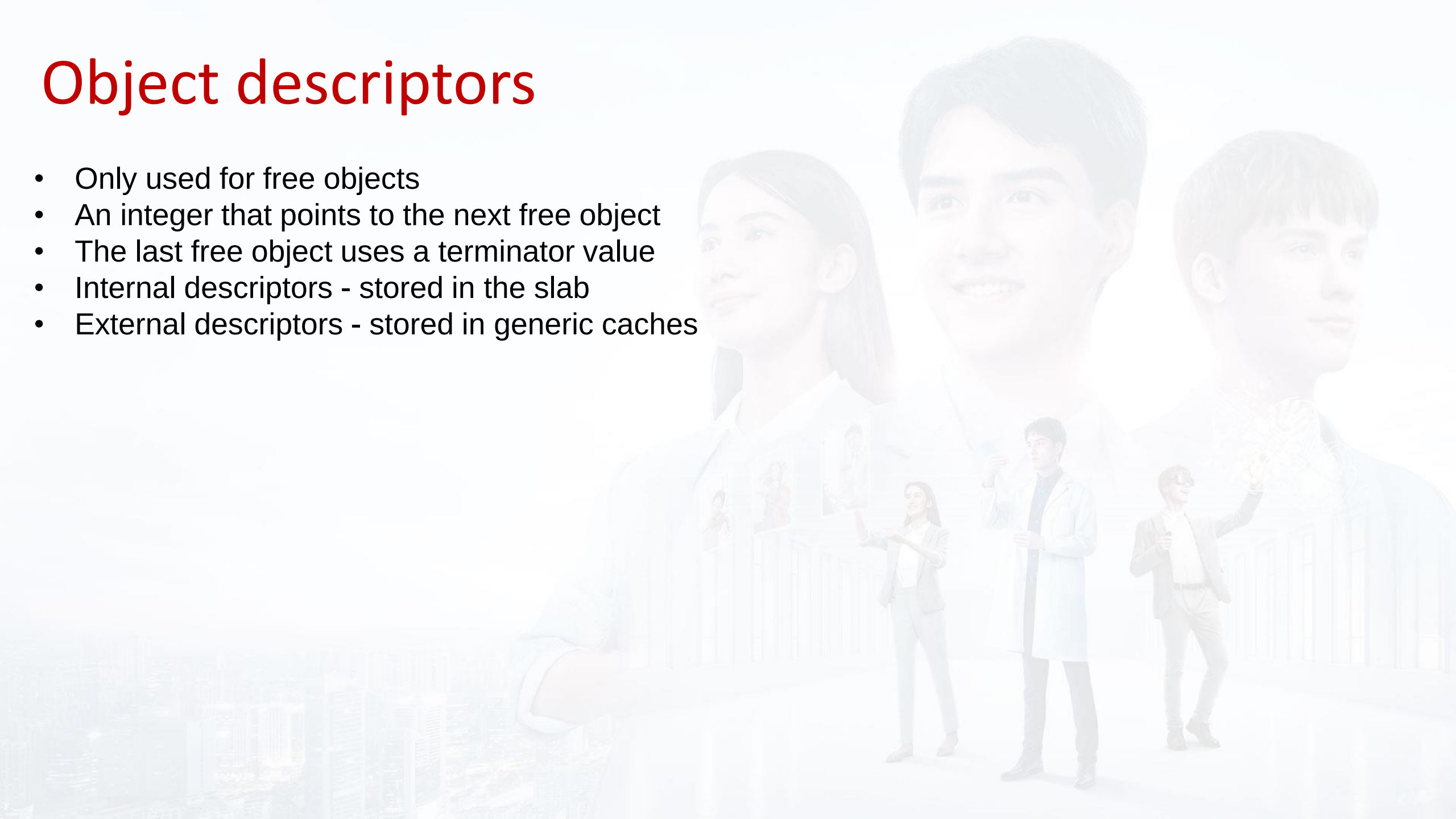


Slab descriptors and generic caches

- 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

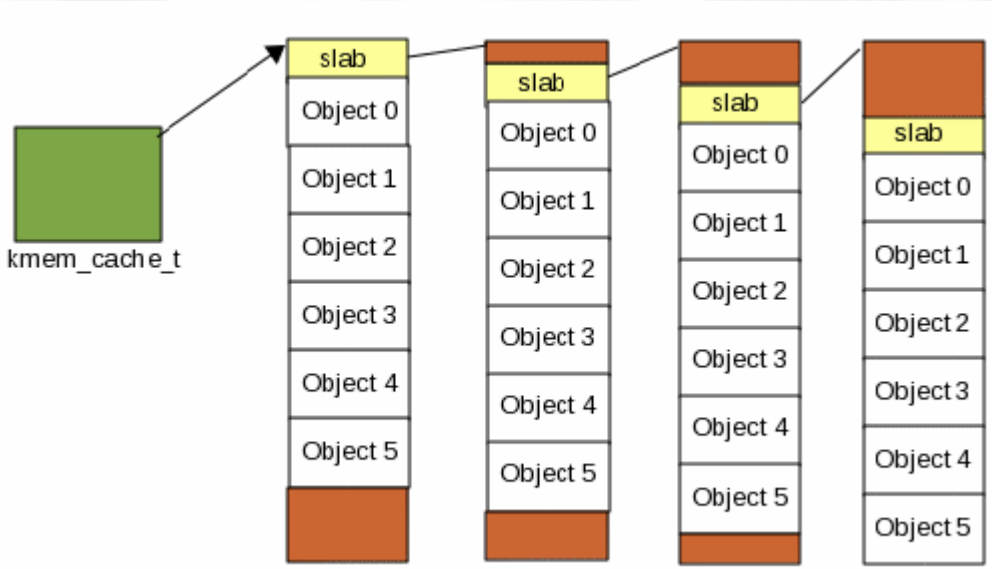
Object descriptors

- 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.

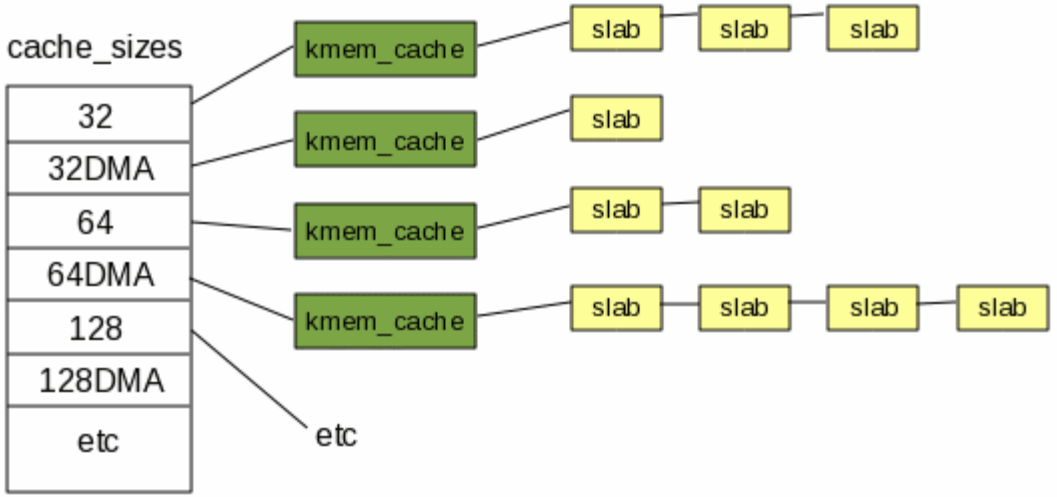


kmalloc() interface

- 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.

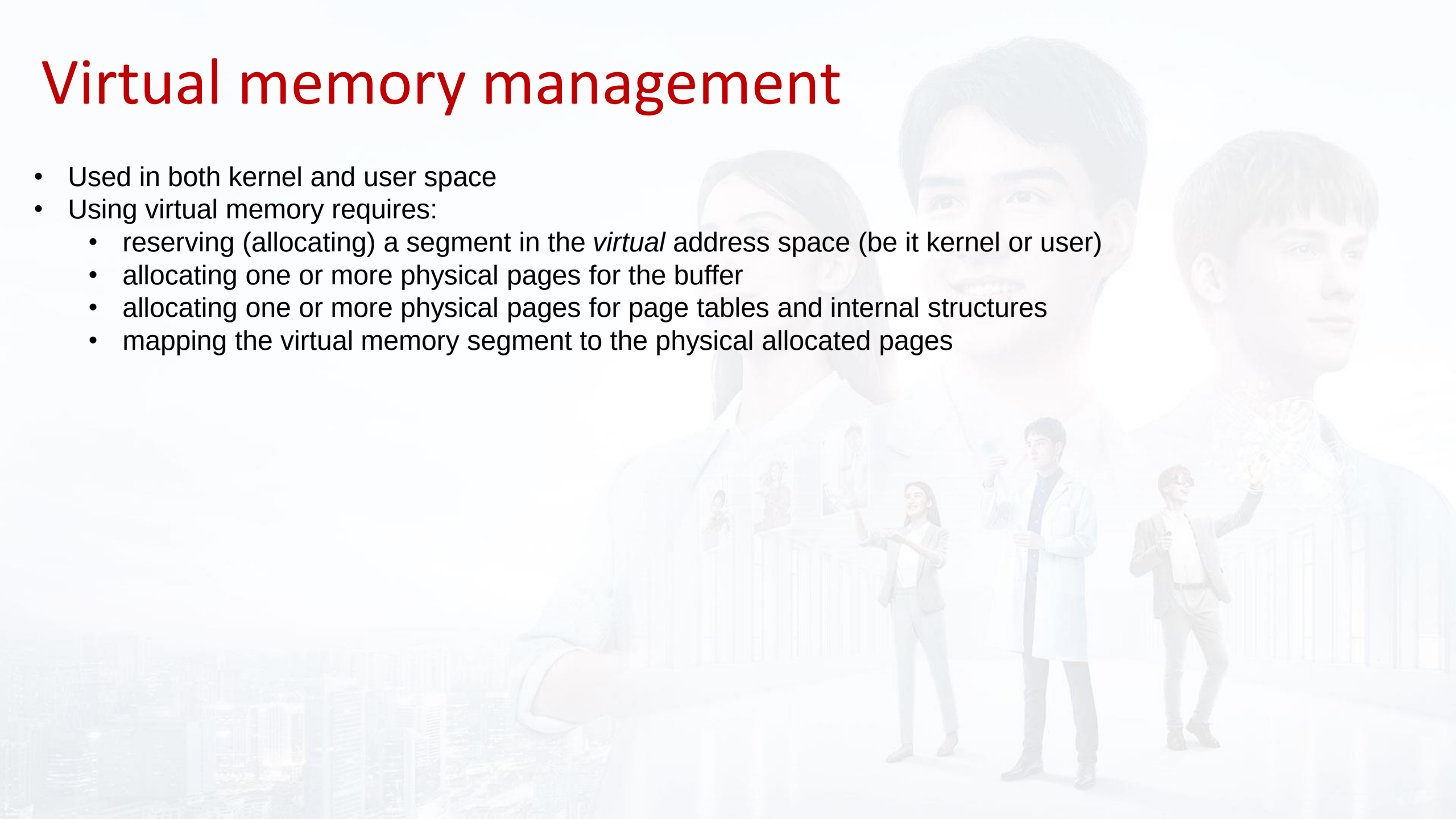
- For those instances drivers and kernel modules use the kmalloc() and kfree() routines.
- The Linux kernel ties these calls into the slab allocator too.

- 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.



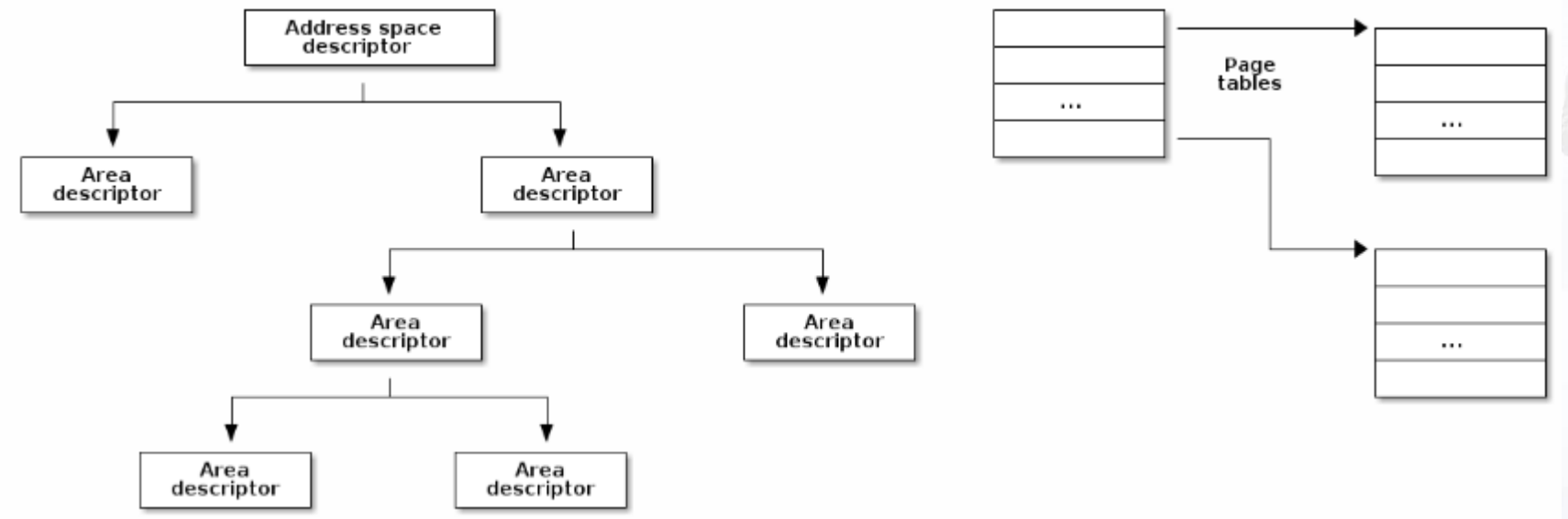
Virtual memory management

- 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



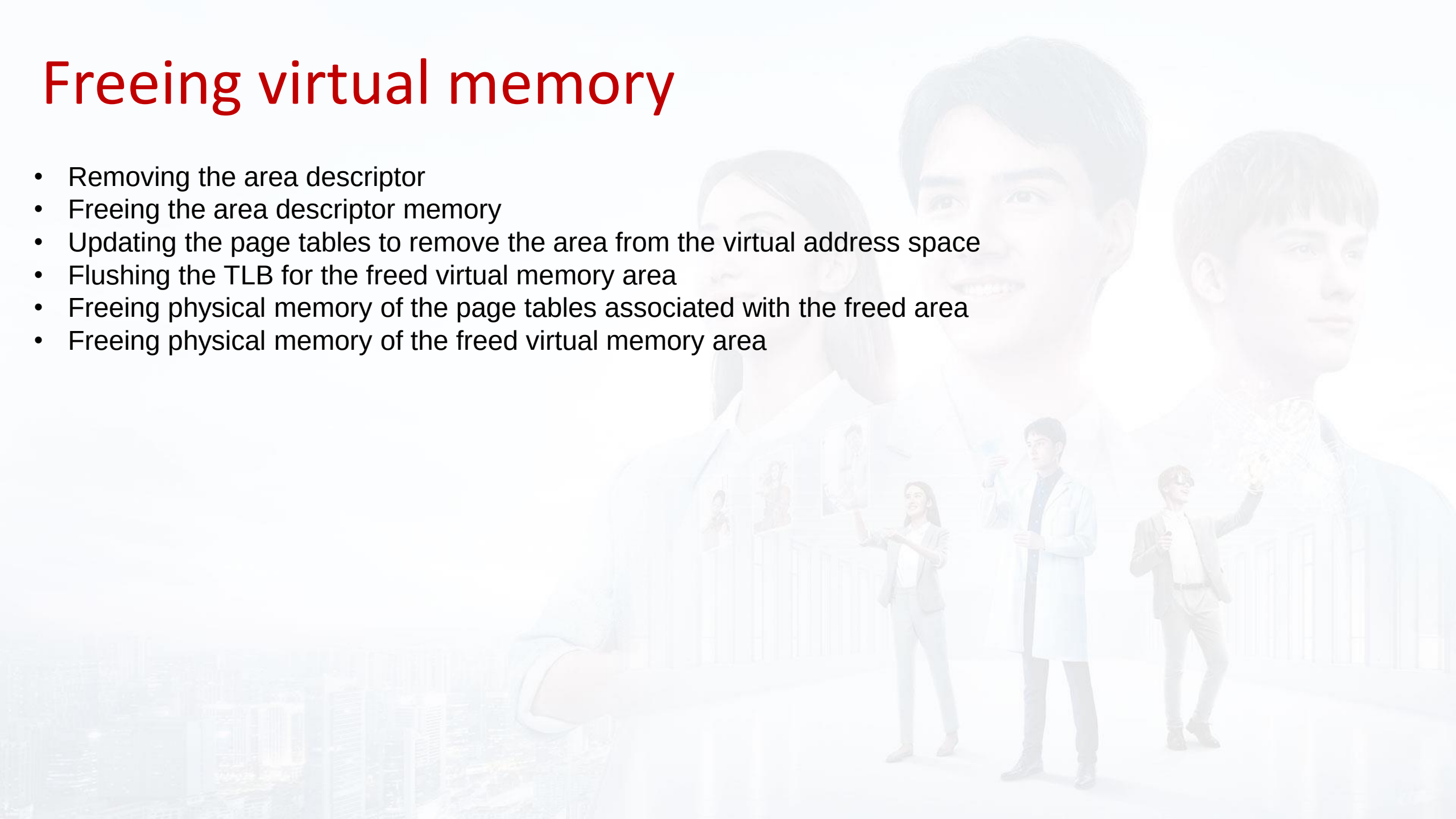
Address space descriptors and VM allocation

- 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.

Summary, bits and pieces 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.

Summary, bits and pieces 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.

What is page fault

- It is an exception that the memory management unit (MMU) raises when a process accesses a (virtual) memory page without mapping to RAM physical page
  - Write: VA is allocated but no PA mapped
  - Read: VA is mapped to page in swap space (backing store)
  - Illegal (invalid VA): Process terminated
- Write requires a mapping to be added to the process's virtual address space.
  - OS invokes the *OOM killer* if no free pages available
  - 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.
- Read requires actual page contents to be loaded from a backing store
- The MMU detects the page fault, but the operating system's kernel handles the exception by making the required page accessible in the physical memory or denying an illegal memory access.

Page fault data structures

- MMU TLBs (caches with frequently used VA->PA mappings)
- MMU translation tables
- Process working set (PWS)
  - group of physical memory pages currently dedicated to a specific process
  - OS may write modified pages to a dedicated area on a mass storage device (usually known as swapping or paging space)
  - OS marks unmodified pages as being free (there is no need to write these pages out to disk as they have not changed)
  - A LRU algorithm determines which pages are eligible for removal from PWS
- Per-CPU (core) list of free pages
  - fast path – no synchronization (almost)
  - in the zone structure used to describe a memory-management zone
  - When MM system is under memory pressure these lists are taken (become empty)
  - each CPU now has *two* sets of lists to hold free pages, one of which is in use at any given time, while the other is kept in reserve
  - Atomic compare-and-swap is used to switch the two lists to be taken by MM system
- Shared list of free pages
  - slow path, used when per-core list is empty
- Kernel same-page merging (KSM) deduplicates VM pages

Page fault types: Minor

- case 1
  - page is loaded in memory at the time the fault is generated, but is not marked in the MMU as loaded.
  - OS page fault handler updates MMU translation tables to point to this page
  - this happens if the page is shared by different processes and the page was loaded by other processes.
- case 2
  - The page could also have been removed from the working set of a process, but not yet written to disk or erased (non-dirty page are discarded and never written to disk)
  - However, the page contents are not overwritten until the page is assigned elsewhere, meaning it is still available if it is referenced by the original process before being allocated.

Page fault types: Major

- Happens when the page is not loaded in memory at the time of the fault.
- Mechanism used by OS to increase the amount of program memory available on demand.
- The operating system delays loading parts of the program from disk until the program attempts to use it and the page fault is generated.
- The page fault handler in the OS needs to find a free location: either a free page in memory, or a non-free page in memory.
- This latter might be used by another process, in which case the OS needs to write out the data in that page (if it has not been written out since it was last modified)
  - and mark that page as not being loaded in memory in its process page table.
- Once free page has been made available
  - OS reads the data for the new page into memory,
  - adds an entry to its location in MMU tables,
  - indicates that the page is loaded.
- Major faults are more expensive than minor faults and add storage access latency to the interrupted program's execution.

Page fault types: Invalid

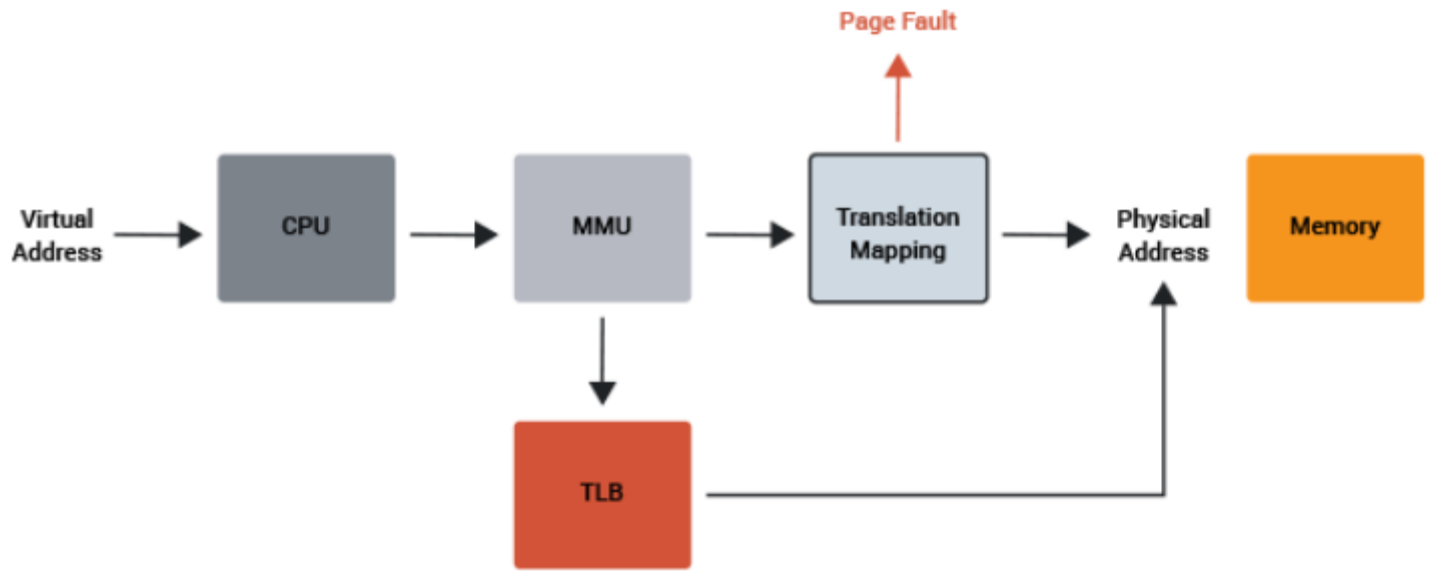
- If a page fault occurs for a reference to an address that is not part of the virtual address space, meaning there cannot be a page in memory corresponding to it, then it is called an invalid page fault.
- The page fault handler in the operating system will then generally pass a segmentation fault to the offending process, indicating that the access was invalid;
- This usually results in abnormal termination of the code that made the invalid reference.
- A null pointer is usually represented as a pointer to address 0 in the address space.
- OS sets up the MMU TT to indicate that the page that contains that address is not in memory
  - attempts to read or write the memory referenced by a null pointer get an invalid page fault.

Page fault performance impacts

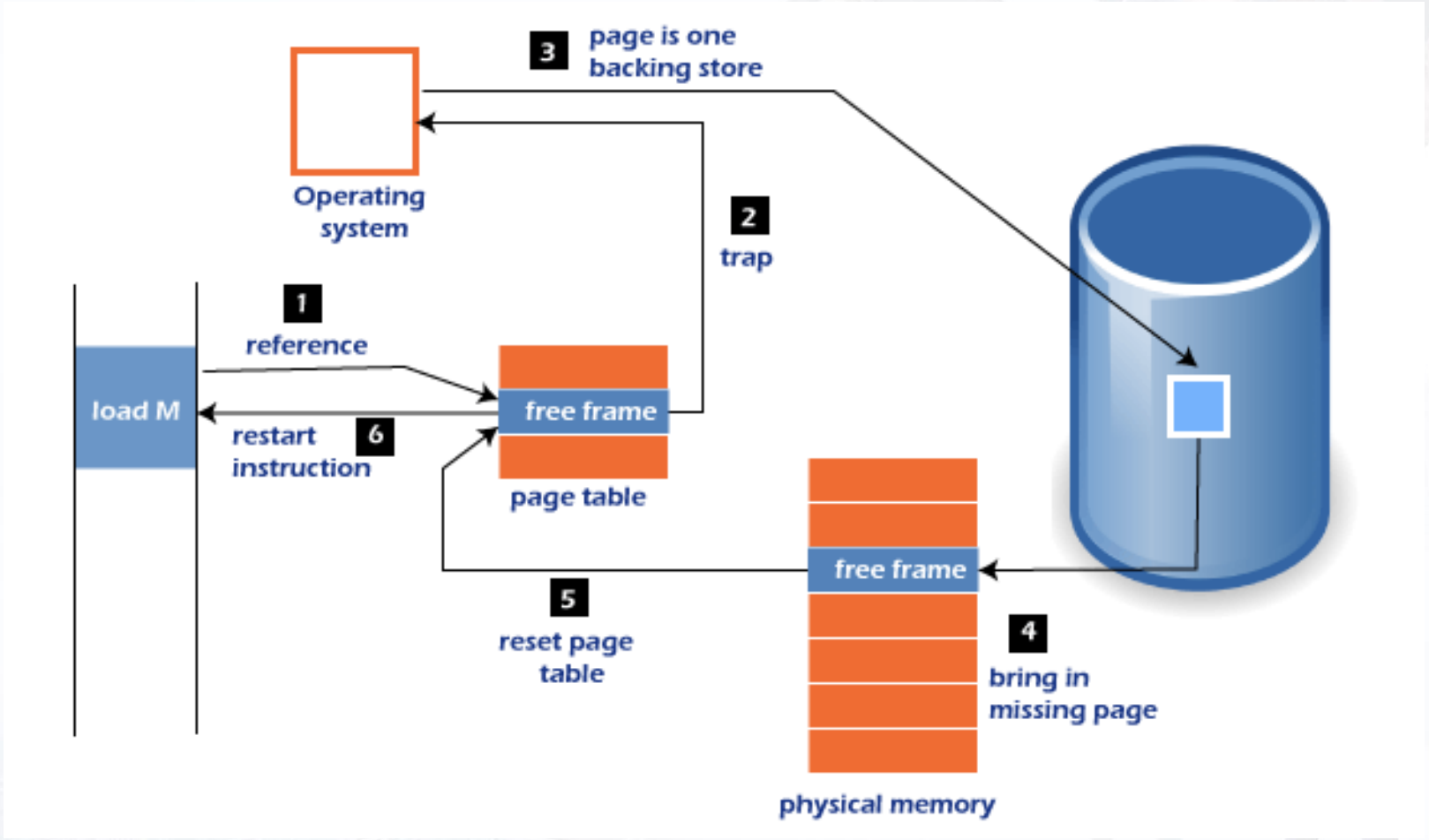
- Major page faults on conventional computers using hard disk drives can have a significant impact on their performance
- Average hard disk drive has
  - average rotational latency of 3 ms
  - seek time of 5 ms
  - transfer time of 0.05 ms/page
- Therefore, the total time for paging is near 8 ms (= 8,000 μs).
  - Column Address Strobe (CAS) latency, or CL, is the delay in clock cycles between the READ command and the moment data (1st word) is available
  - DDR3-1066 CL is 7 cycles, 1 cycle is 1.875ns
  - DDR4-1600 CL is 12 cycles, 1 cycle is 1.250ns
- Performance optimization of programs or operating systems often involves reducing the number of page faults.
- Two primary focuses of the optimization are reducing overall memory usage and improving memory locality.
- To reduce the page faults, developers must use an appropriate page replacement algorithm that maximizes the page hits.

- A larger physical memory also reduces page faults.

Summary of how page fault happens



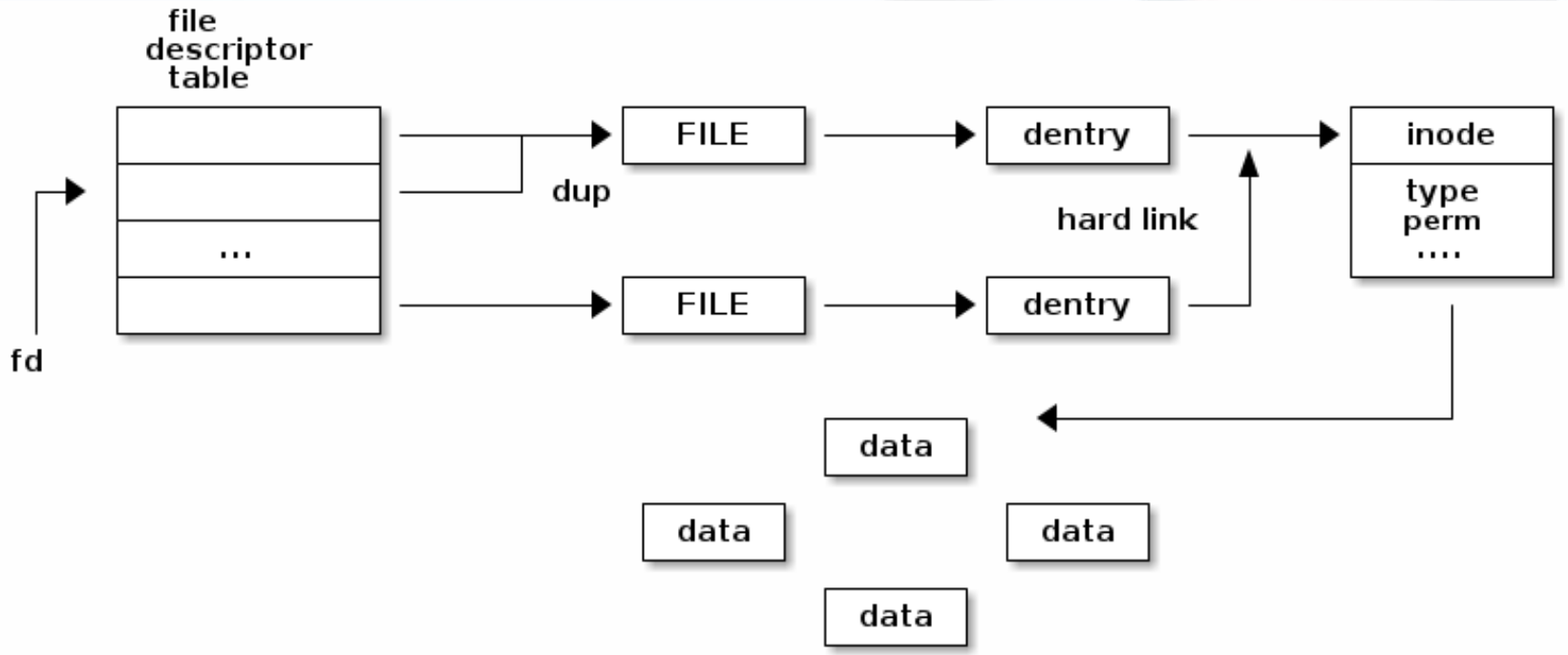
Page fault from process execution PoV



File system abstractions

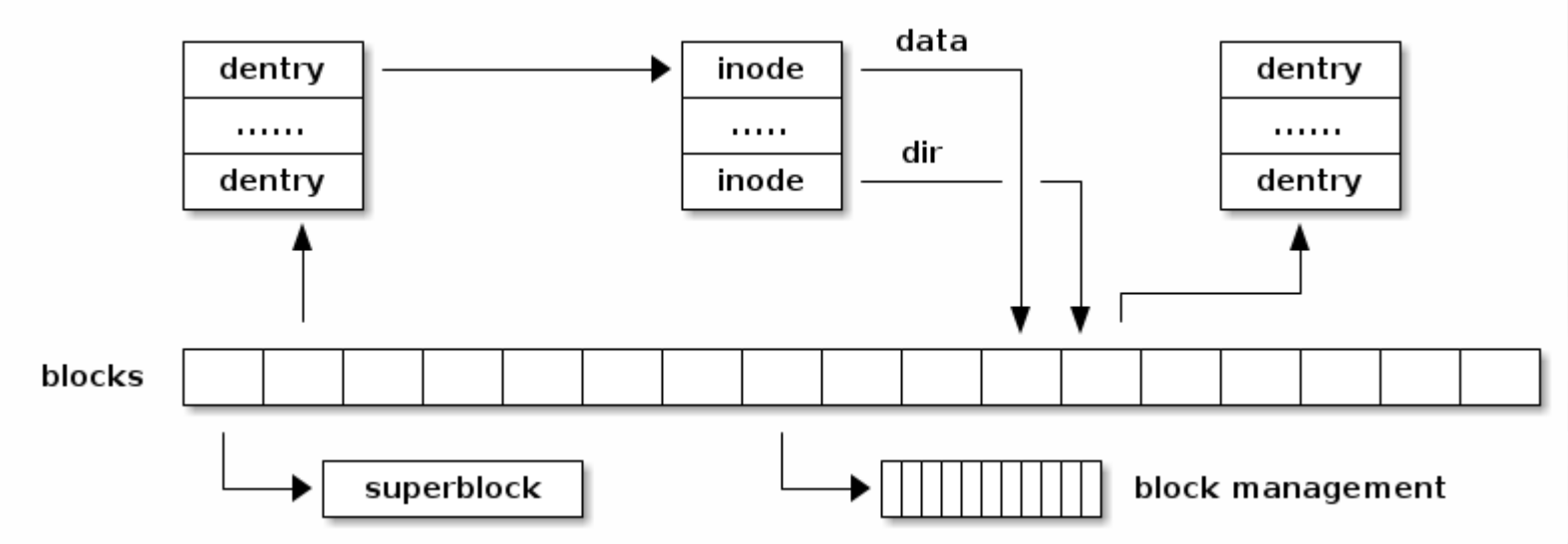
- The *superblock* abstraction contains information about the filesystem instance such as the block size, the root inode, filesystem size. It is present both on storage and in memory (for caching purposes).
- The *file* abstraction contains information about an opened file such as the current file pointer. It only exists in memory.
- The *inode* is identifying a file on disk. It exists both on storage and in memory (for caching purposes). An inode identifies a file in a unique way and has various properties such as the file size, access rights, file type, etc.
- The *dentry* associates a path/file name with an inode. It exists both on storage and in memory (for caching purposes).

File system abstractions in memory



- Multiple file descriptors can point to the same file because we can use the dup() system call to duplicate a file descriptor.
- Multiple file abstractions can point to the same dentry if we use the same path multiple times.
- Multiple dentries can point to the same inode when hard links are used.

File system abstractions on storage (disk)

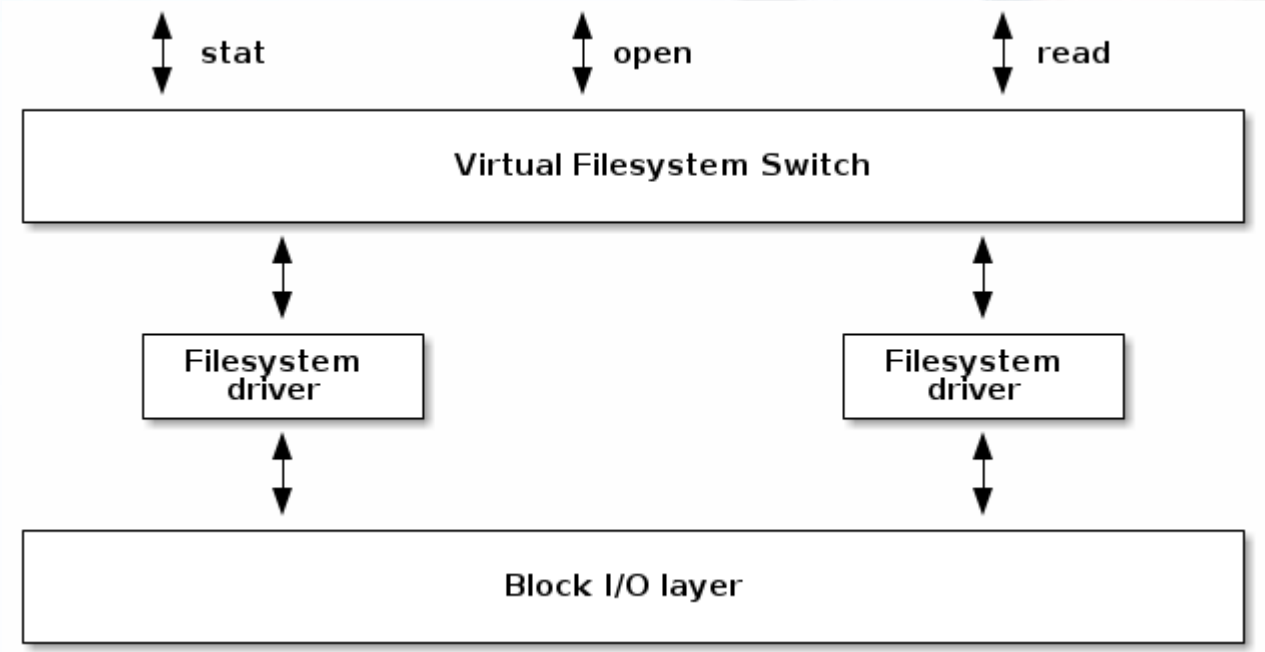


- *superblock* is typically stored at the beginning of the filesystem
- Various blocks are used with different purposes:
  - some to store dentries,
  - some to store inodes and some to store user data blocks.
  - Some blocks used to manage the available free blocks (e.g. bitmaps for the simple filesystems).

Simple file system example (disk)

- the superblock contains information about the block size as well as the IMAP, DMAP, IZONE and DZONE areas.
- IMAP area is comprised of multiple blocks which contains a bitmap for inode allocation
  - it maintains the allocated/free state for all inodes in the IZONE area
- DMAP area is comprised of multiple blocks which contains a bitmap for data blocks
  - it maintains the allocated/free state for all blocks the DZONE area

File system operations 1 of 2



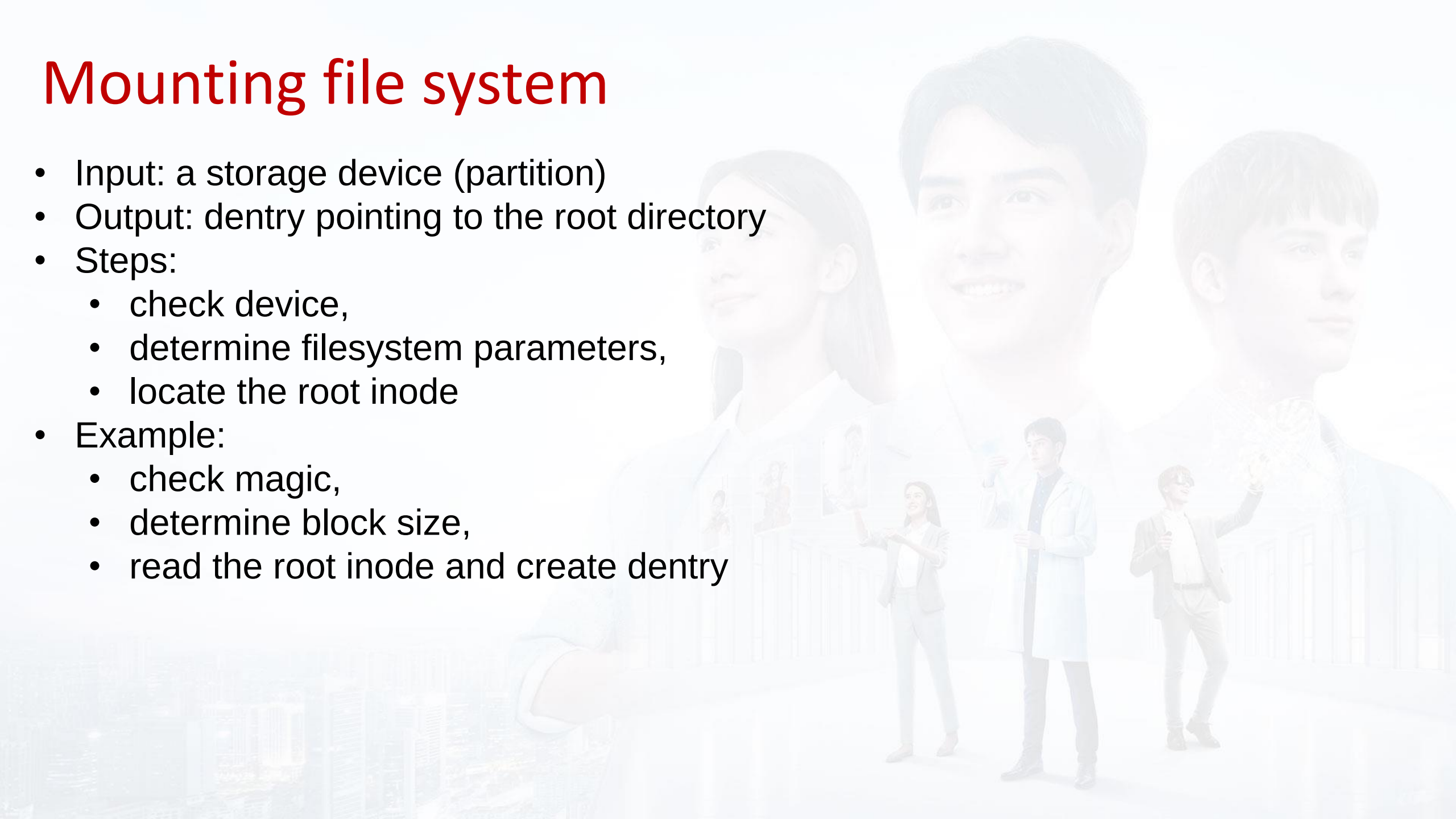
- high level overview of how the file system drivers interact with the rest of the file system "stack".
- In order to support multiple filesystem types and instances Linux implements a large and complex subsystem that deals with filesystem management.
- This is called Virtual File System/Switch (VFS).

File system operations 2 of 2

- VFS translates the complex file management related system calls to simpler operations that are implemented by the device drivers.
- Operations that a file system must implement:
  - Mount
  - Open a file
  - Querying file attributes
  - Reading data from a file
  - Writing file to a file
  - Creating a file
  - Deleting a file
  - Etc.

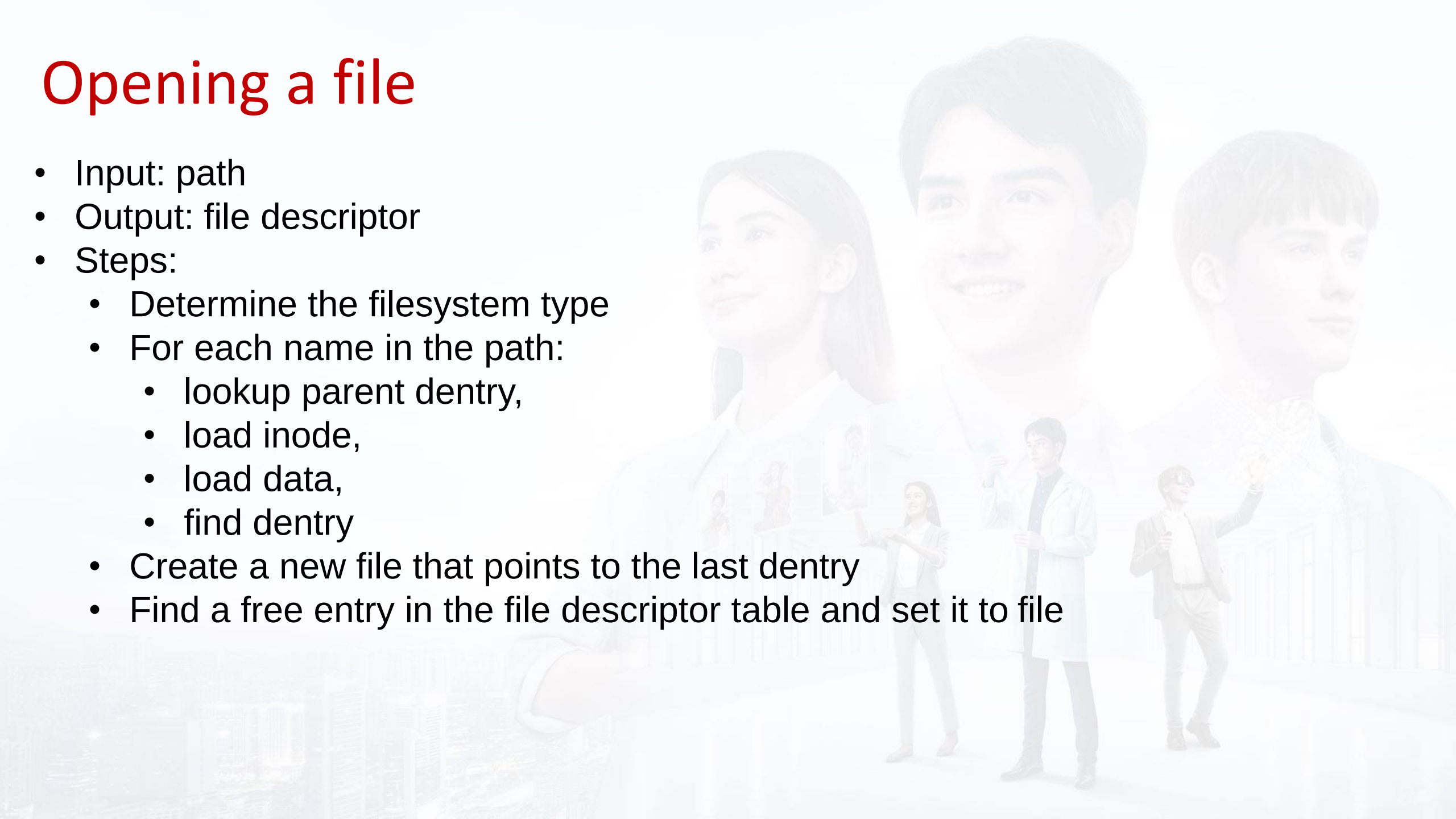
Mounting file system

- Input: a storage device (partition)
- Output: dentry pointing to the root directory
- Steps:
  - check device,
  - determine filesystem parameters,
  - locate the root inode
- Example:
  - check magic,
  - determine block size,
  - read the root inode and create dentry



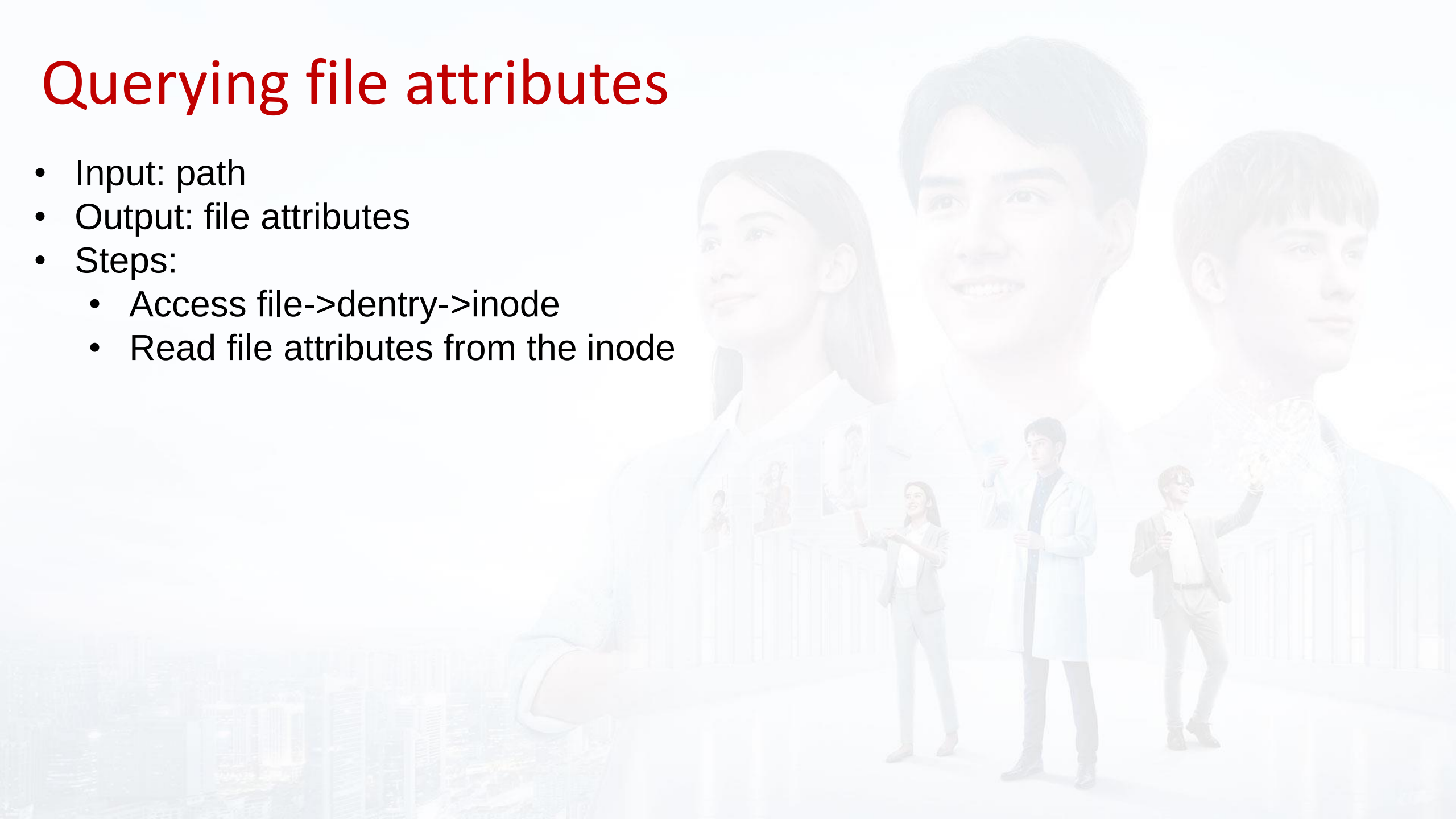
Opening a file

- Input: path
- Output: file descriptor
- Steps:
  - Determine the filesystem type
  - For each name in the path:
    - lookup parent dentry,
    - load inode,
    - load data,
    - find dentry
  - Create a new file that points to the last dentry
  - Find a free entry in the file descriptor table and set it to file



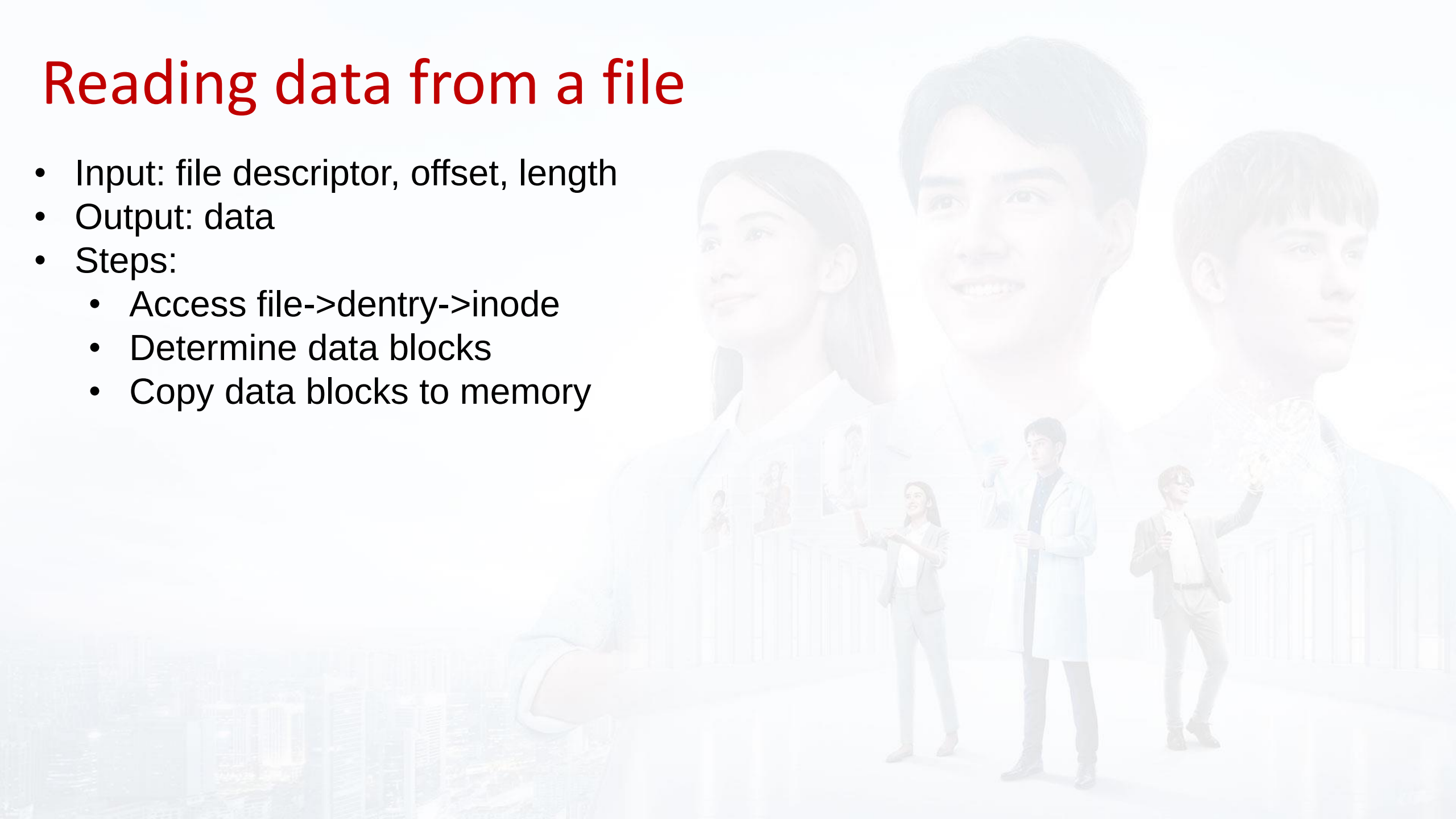
Querying file attributes

- Input: path
- Output: file attributes
- Steps:
  - Access file->dentry->inode
  - Read file attributes from the inode



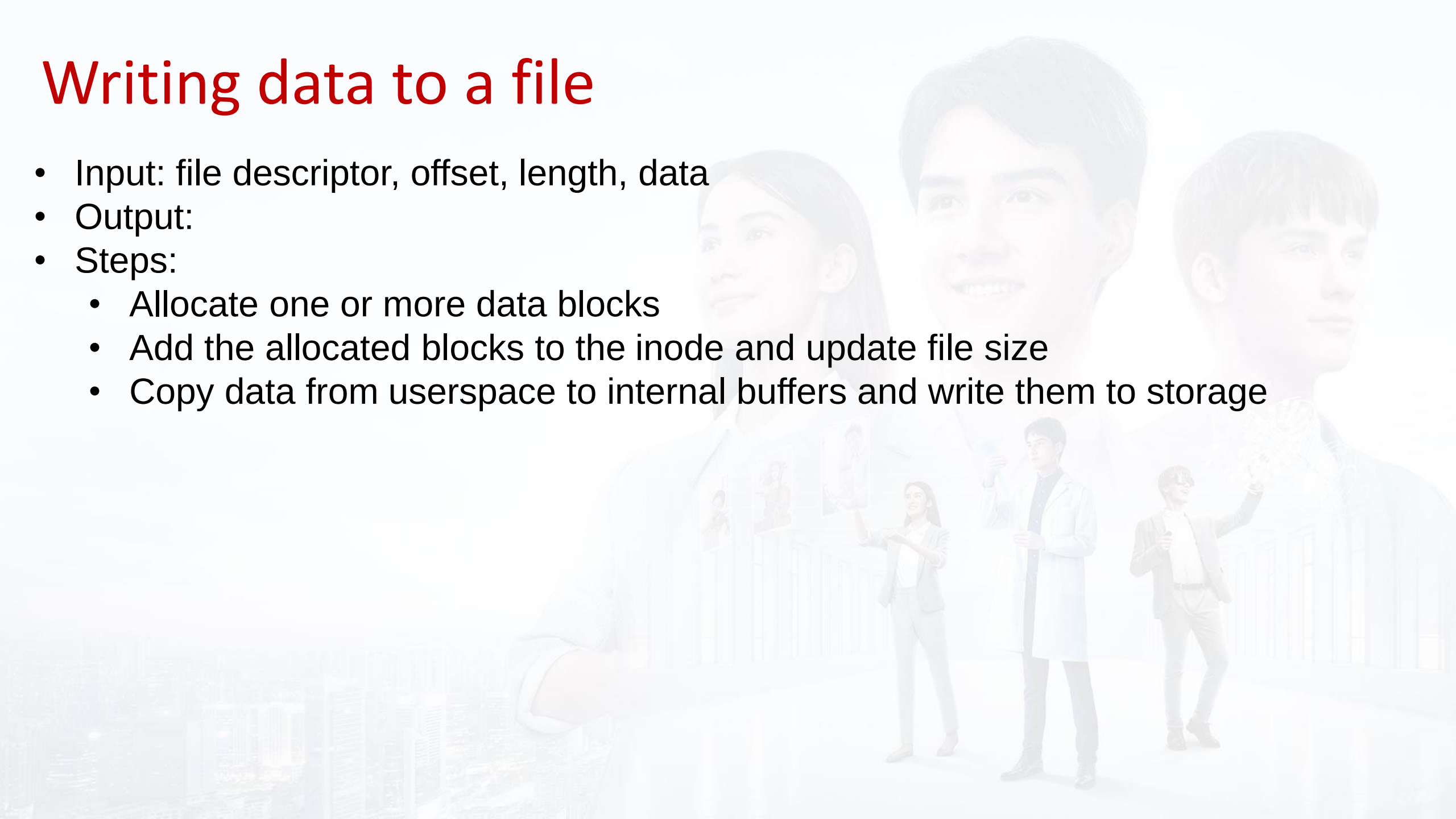
Reading data from a file

- Input: file descriptor, offset, length
- Output: data
- Steps:
  - Access file->dentry->inode
  - Determine data blocks
  - Copy data blocks to memory



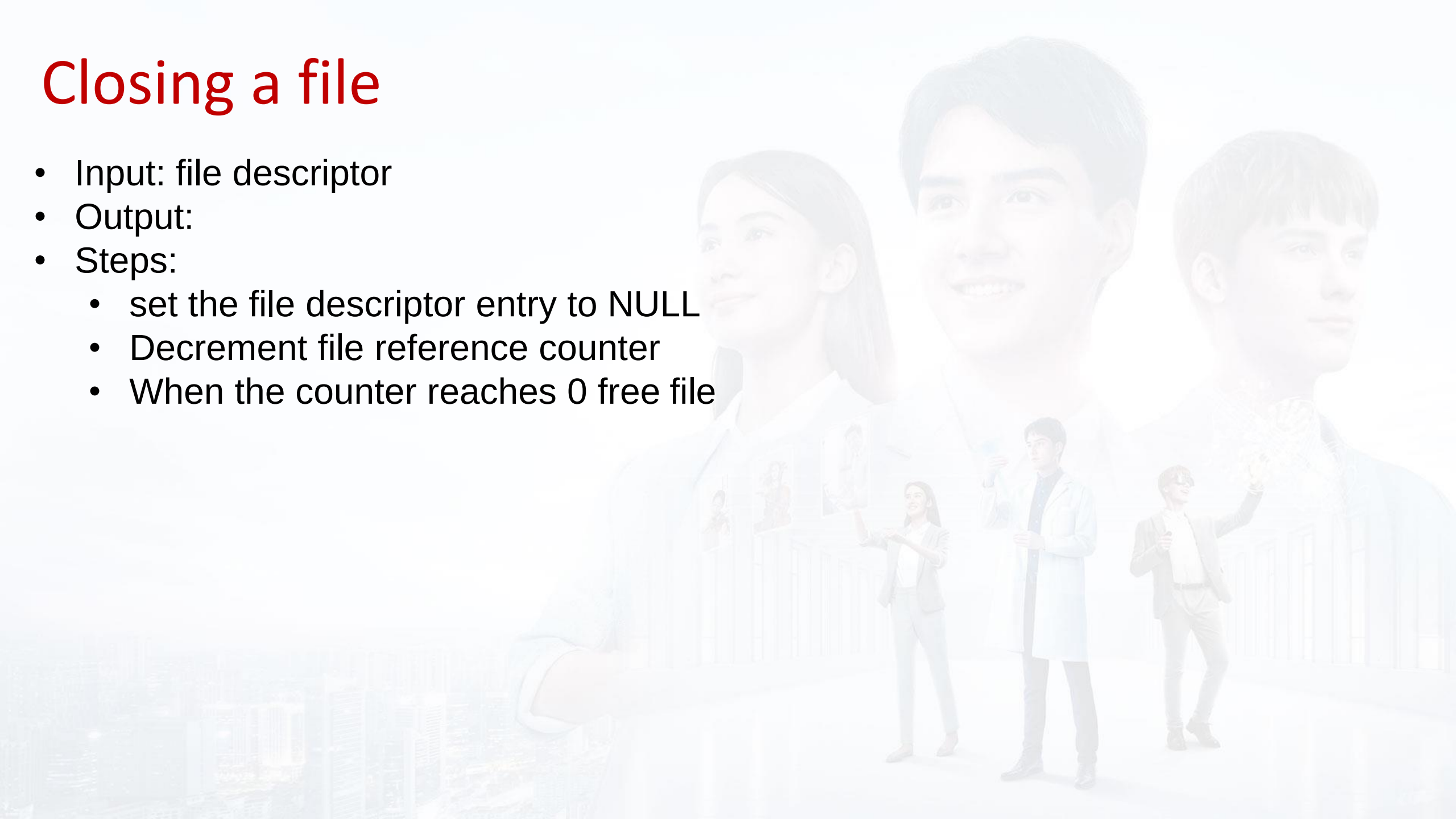
Writing data to a file

- Input: file descriptor, offset, length, data
- Output:
- Steps:
  - Allocate one or more data blocks
  - Add the allocated blocks to the inode and update file size
  - Copy data from userspace to internal buffers and write them to storage



Closing a file

- Input: file descriptor
- Output:
- Steps:
  - set the file descriptor entry to NULL
  - Decrement file reference counter
  - When the counter reaches 0 free file



Directories

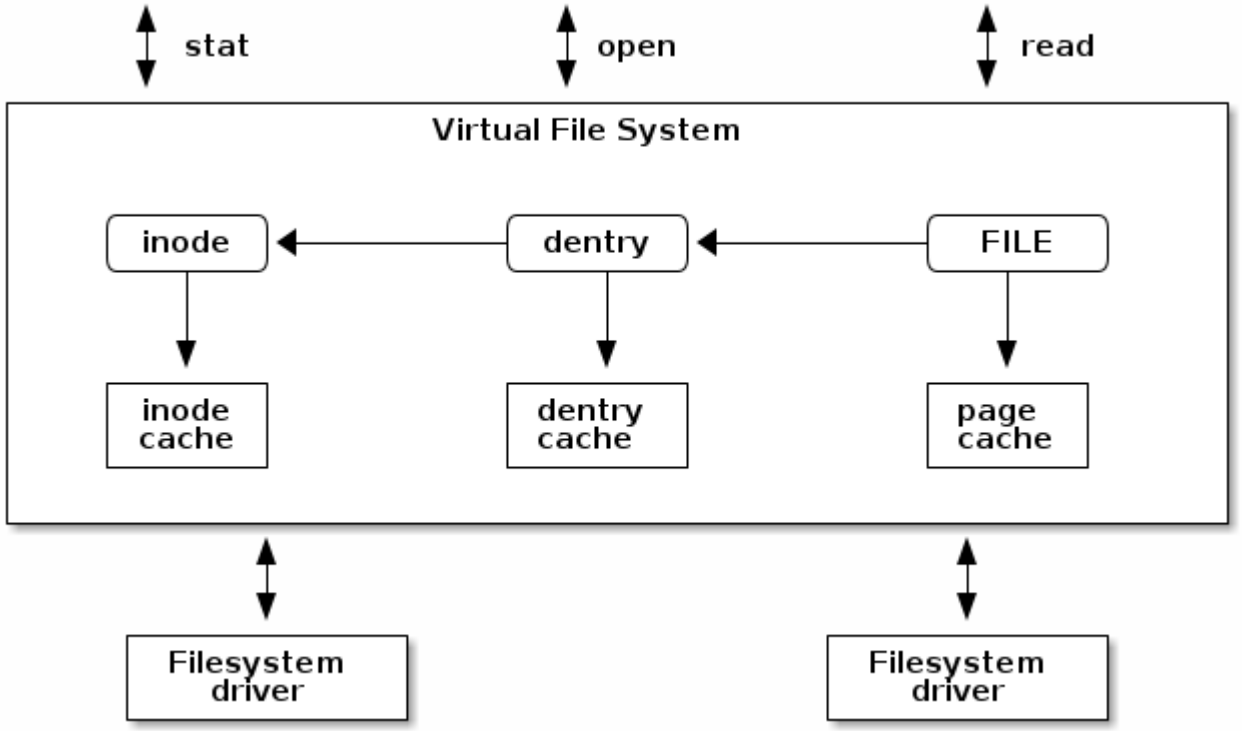
- They are special files which contain one or more dentries
- Creating a file:
  - Input: path
  - Output:
  - Steps:
    - Determine the inode directory
    - Read data blocks and find space for a new dentry
    - Write back the modified inode directory data blocks

- Deleting a file:
  - Input: path
  - Output:
  - Steps:
    - determine the parent inode
    - read parent inode data blocks
    - find and erase the dentry (check for links)
    - when last file is closed: deallocate data and inode blocks

Virtual File System

- Main purpose for the original introduction of VFS was to support multiple filesystem types and instances
- Side effect was that it simplified fileystem device driver development since command parts are now implement in the VFS.
- Almost all of the caching and buffer management is dealt with VFS, leaving just efficient data storage management to the filesystem device driver.

- In order to deal with multiple filesystem types, VFS introduced the common filesystem abstractions (presented above).
- Filesystem driver can also use its own particular fileystem abstractions in memory (e.g. ext4 inode or dentry) and that there might be a different abstractions on storage.
- Three slightly different filesystem abstractions:
  - one for VFS - always in memory
  - for a particular filesystem - in memory used by the filesystem driver
  - and one on storage.



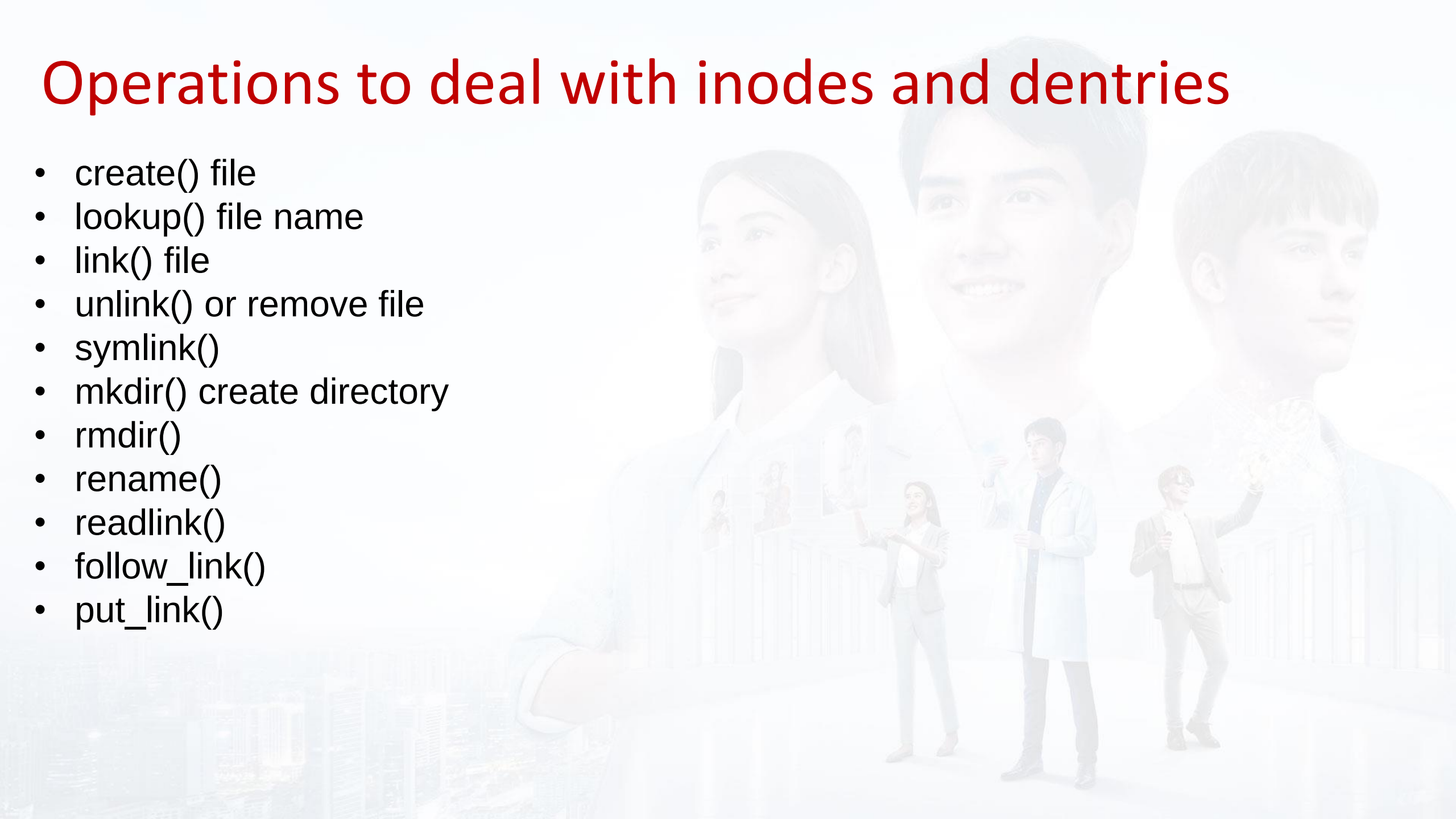
Superblock operations

- VFS requires that all filesystem implement a set of "superblock operations“
  - fill_super()) - reads the filesystem statistics (e.g. total number of inode, free number of inodes, total number of blocks, free number of blocks)
  - write_super() - updates the superblock information on storage (e.g. updating the number of free inode or data blocks)
  - put_super() - free any data associated with the filsystem instance, called when unmounting a filesystem

- Extras to manipulate fileystem inodes - they receive VFS inodes as parameters:
  - read_inode
  - write_inode
  - evict_inode
  - statfs
  - remount_fs

Operations to deal with inodes and dentries

- create() file
- lookup() file name
- link() file
- unlink() or remove file
- symlink()
- mkdir() create directory
- rmdir()
- rename()
- readlink()
- follow_link()
- put_link()



Inode cache

- The inode cache is used to avoid reading and writing inodes to and from storage every time we need to read or update them.
- The cache uses a hash table and inodes are indexed with a hash function which takes as parameters the superblock and the inode number associated with an inode.

- inodes are cached until either the filesystem is unmounted, the inode deleted or the system enters a memory pressure state.
- When this happens the Linux memory management system will free inodes from the inode cache based on how often they were accessed.

- Functions:
  - Caches inodes into memory to avoid costly storage operations
  - An inode is cached until low memory conditions are triggered
  - inodes are indexed with a hash table
  - The inode hash function takes the superblock and inode number as inputs

Dentry cache

- A set of dentry objects in the in-use, unused, or negative state + hash table to derive the dentry object associated with a given filename and a given directory quickly.
- If the required object is not included in the dentry cache, the hashing function returns a null value.
- The dentry cache also acts as a controller for an inode cache.
- The inodes in kernel memory that are associated with unused dentries are not discarded, since the dentry cache is still using them.
- Thus, the inode objects are kept in RAM and can be quickly referenced by means of the corresponding dentries.

- Dentry cache
  - List of used dentries (dentry->d_state == used)
  - List of the most recent used dentries (sorted by access time)
  - Hash table to avoid searching the tree
- State:
  - Used – d_inode is valid and the dentry object is in use
  - Unused – d_inode is valid but the dentry object is not in use
  - Negative – d_inode is not valid; the inode was not yet loaded or the file was erased

Page cache

- It is disk IO cache used by Linux kernel.
- New pages are added to the page cache to satisfy User Mode processes's read requests.
- If the page is not already in the cache, a new entry is added to the cache and filled with the data read from the disk.
- If there is enough free memory, the page is kept in the cache for an indefinite period of time and can then be reused by other processes without accessing the disk.

- Similarly, before writing a page of data to a block device, the kernel verifies whether the corresponding page is already included in the cache; if not, a new entry is added to the cache and filled with the data to be written on disk.

- The I/O data transfer does not start immediately: the disk update is delayed for a few seconds, thus giving a chance to the processes to further modify the data to be written (in other words, the kernel implements deferred write operations).

- Kernel code and kernel data structures don't need to be read from or written to disk.

- Pages included in the page cache:
  - Pages containing data of regular files;
  - Pages containing directories;
  - Pages containing data directly read from block device files

- Uses the struct address_space to translate file offsets to block offsets
- Used for both read / write and mmap
- Uses a radix tree