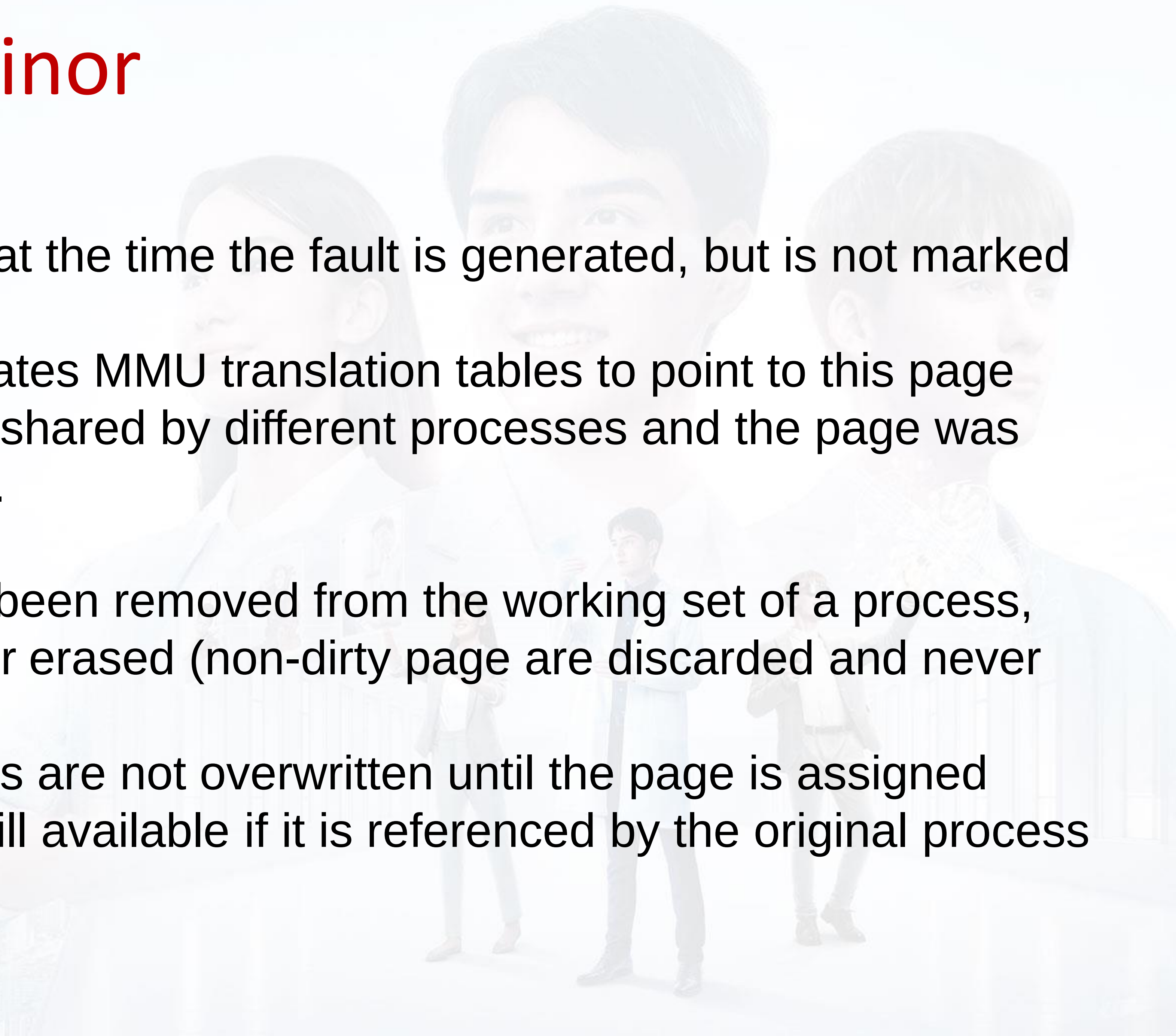
ITMO Advanced OS 2024

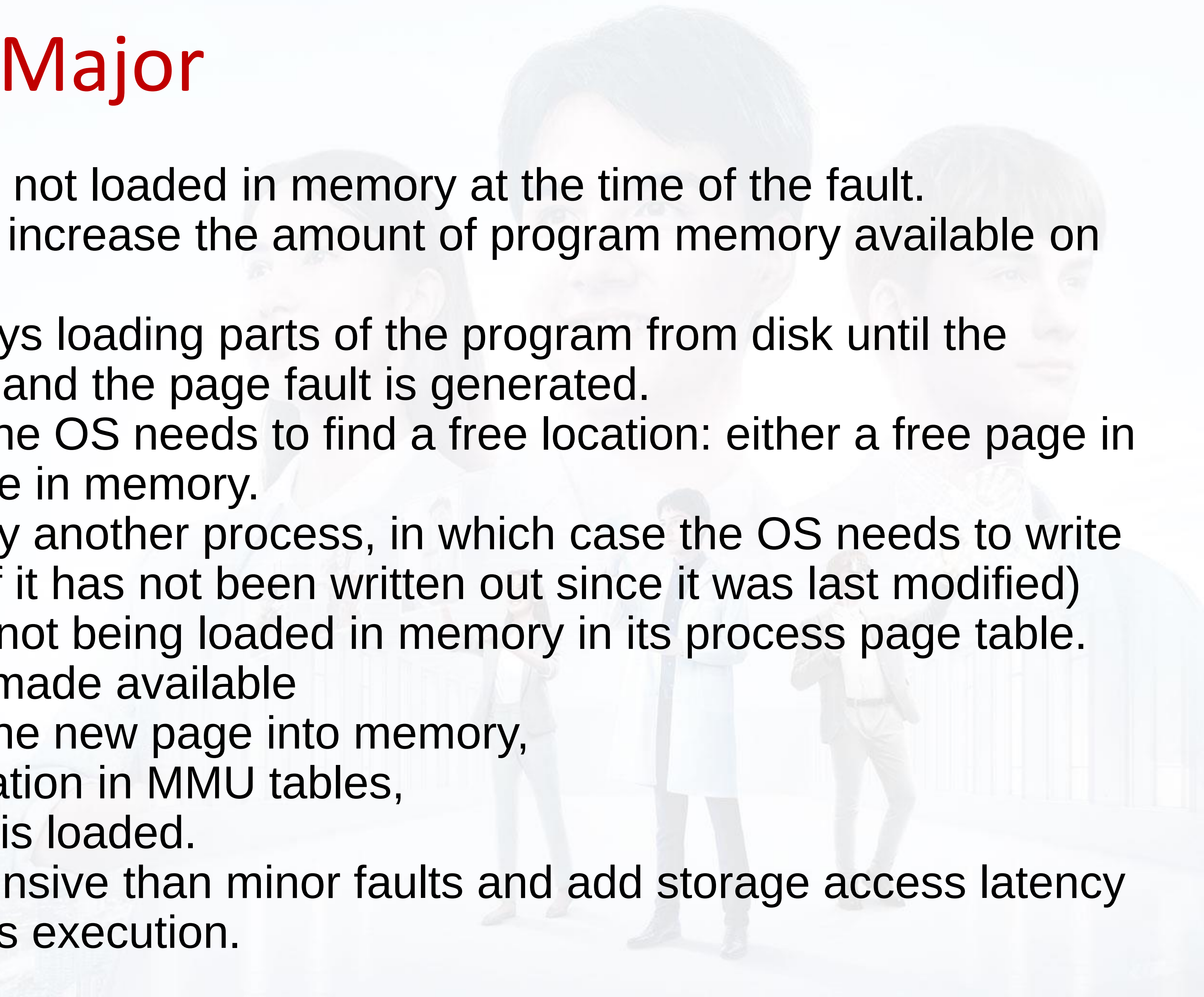
Aleksei Romanovskii, PhD,
SPb Research Center (CBG OS Lab)

Lesson 2024.10.30

Page fault types: Minor

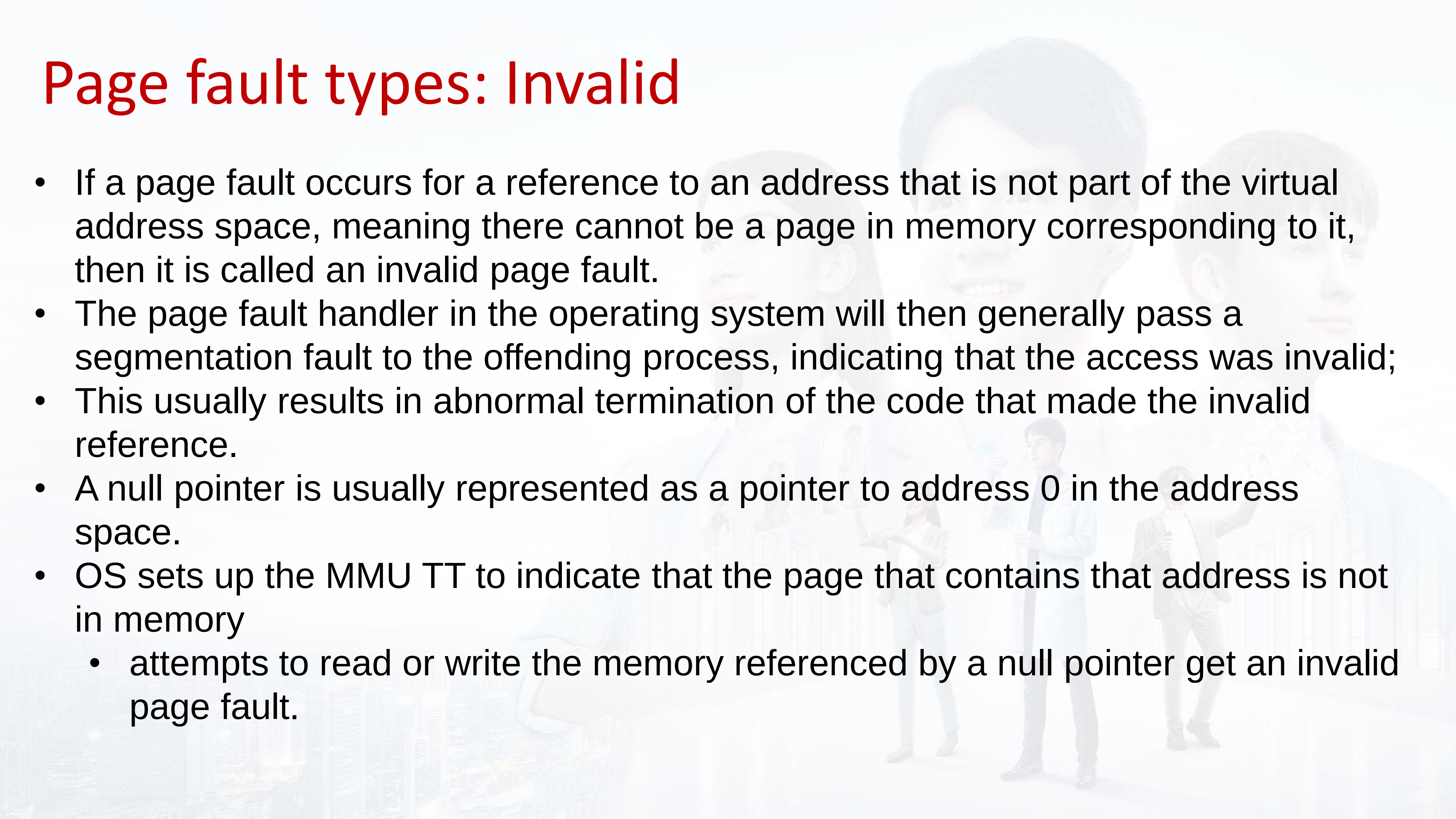


- case 1
  - page is loaded in memory at the time the fault is generated, but is not marked in the MMU as loaded.
  - OS page fault handler updates MMU translation tables to point to this page
  - this happens if the page is shared by different processes and the page was loaded by other processes.
- 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

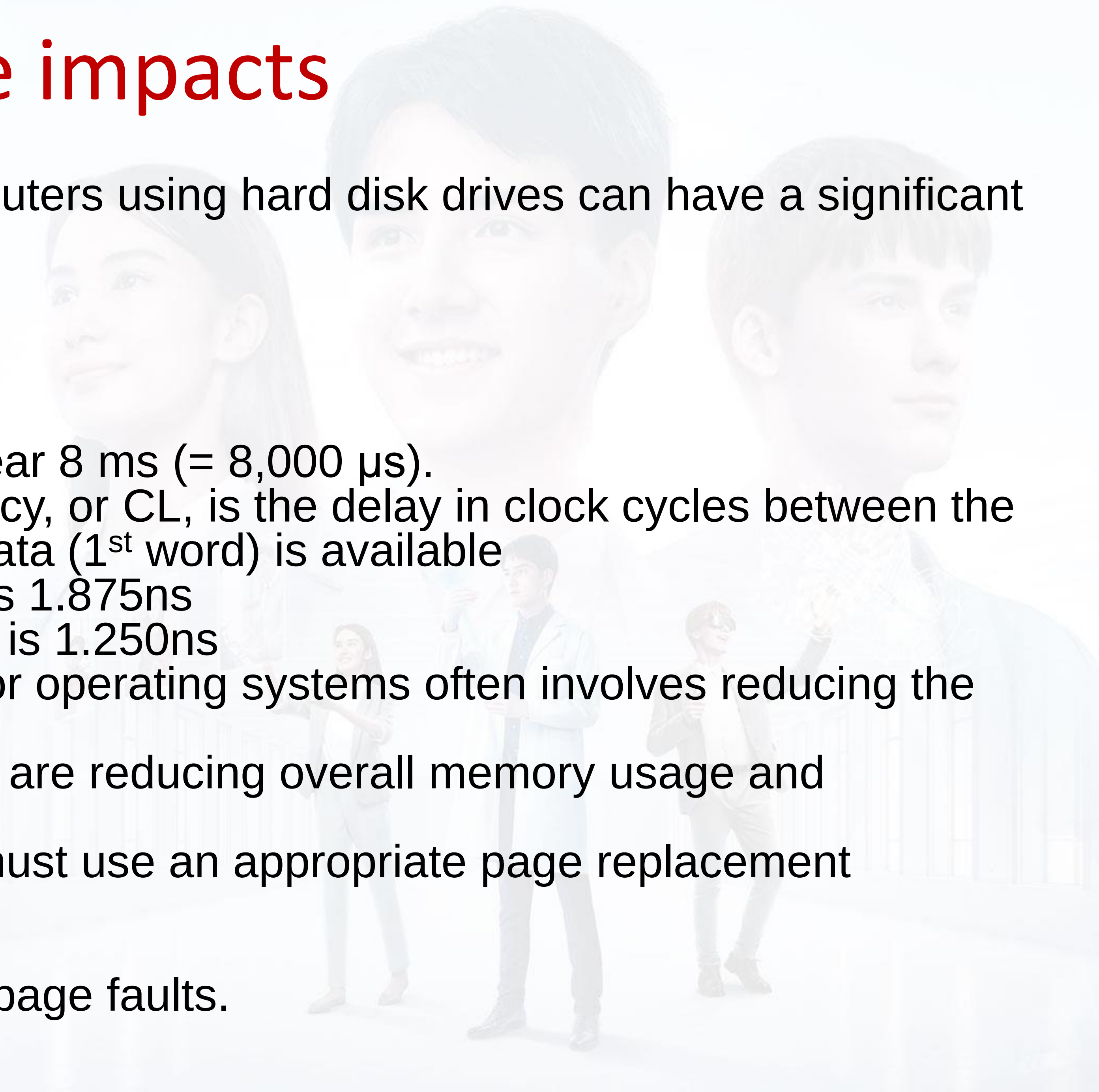
- Happens when the page is not loaded in memory at the time of the fault.
- Mechanism used by OS to increase the amount of program memory available on demand.
- The operating system delays loading parts of the program from disk until the program attempts to use it and the page fault is generated.
- The 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.

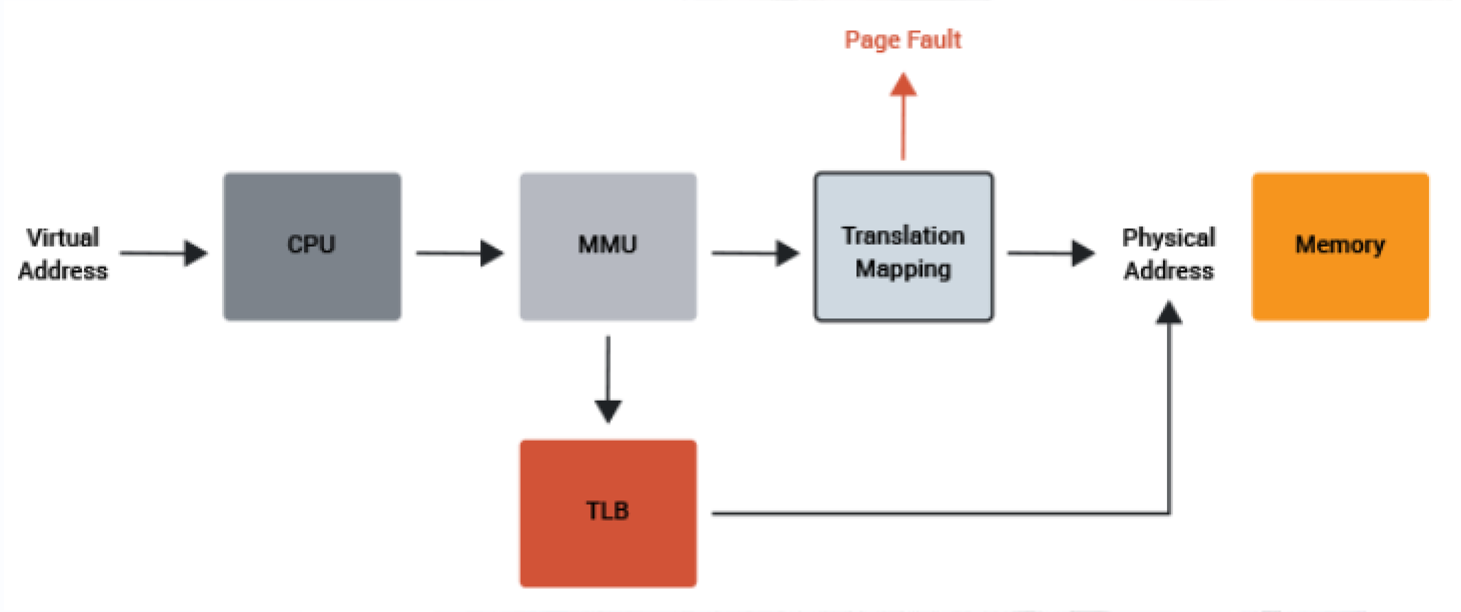
Page fault performance impacts



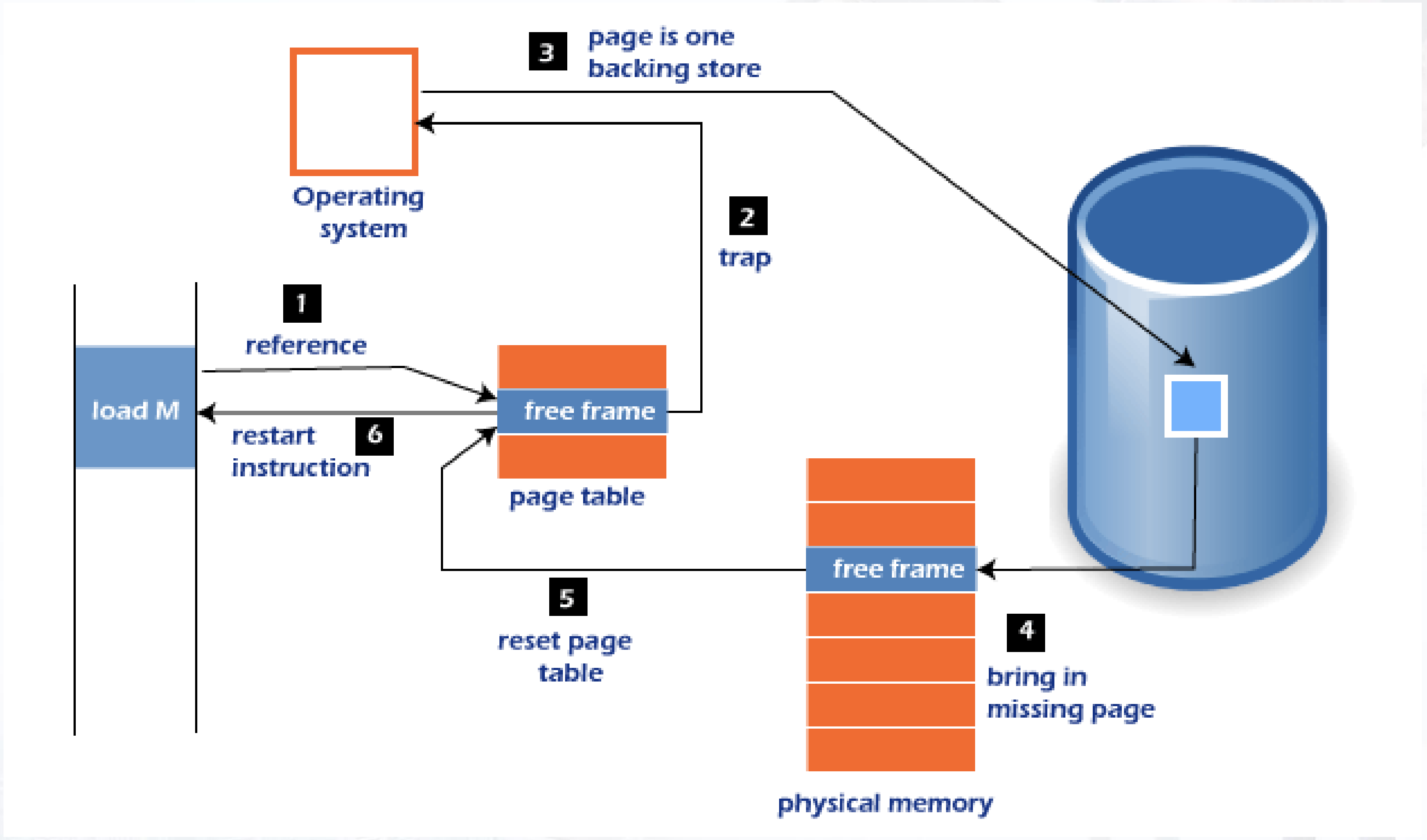
- Major page faults on conventional computers using hard disk drives can have a significant impact on their performance
- Average hard disk drive has
  - average rotational latency of 3 ms
  - seek time of 5 ms
  - transfer time of 0.05 ms/page
- Therefore, the total time for paging is near 8 ms (= 8,000 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



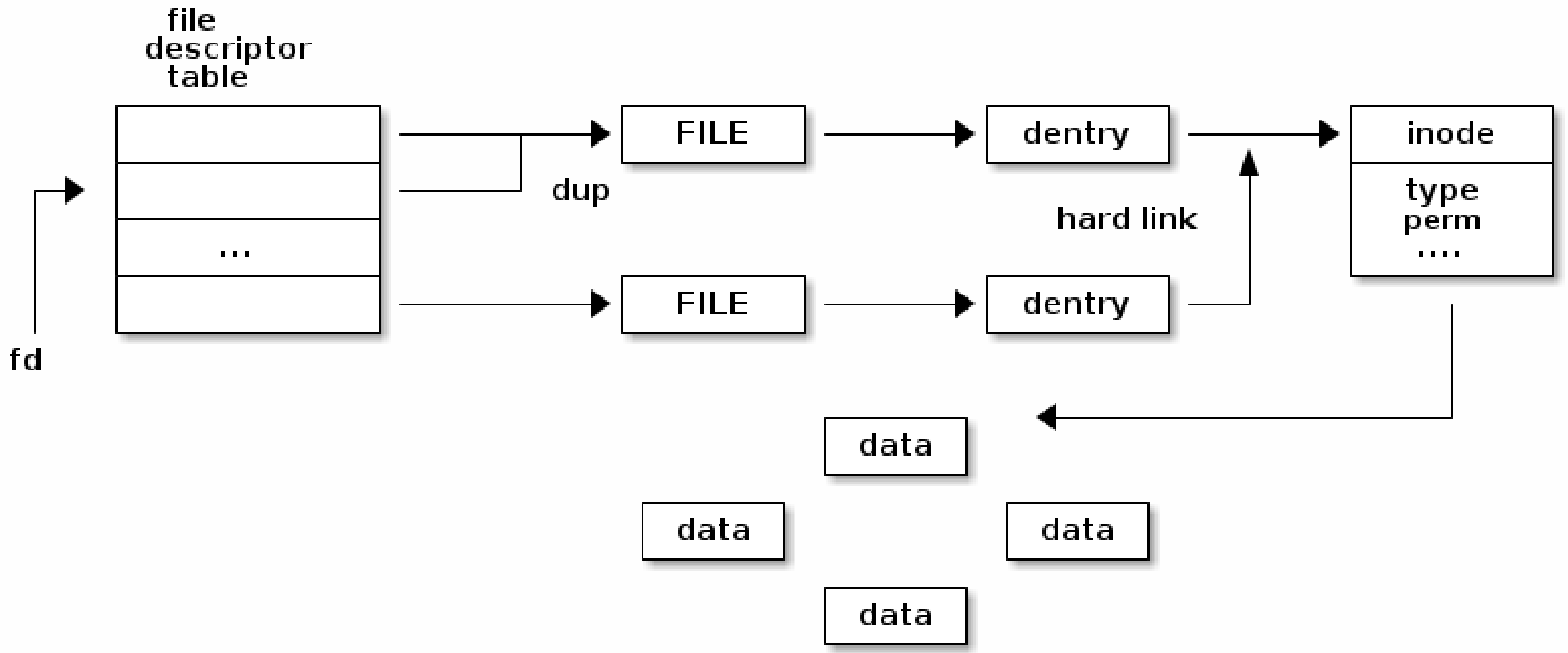
Page fault from process execution PoV



File system abstractions

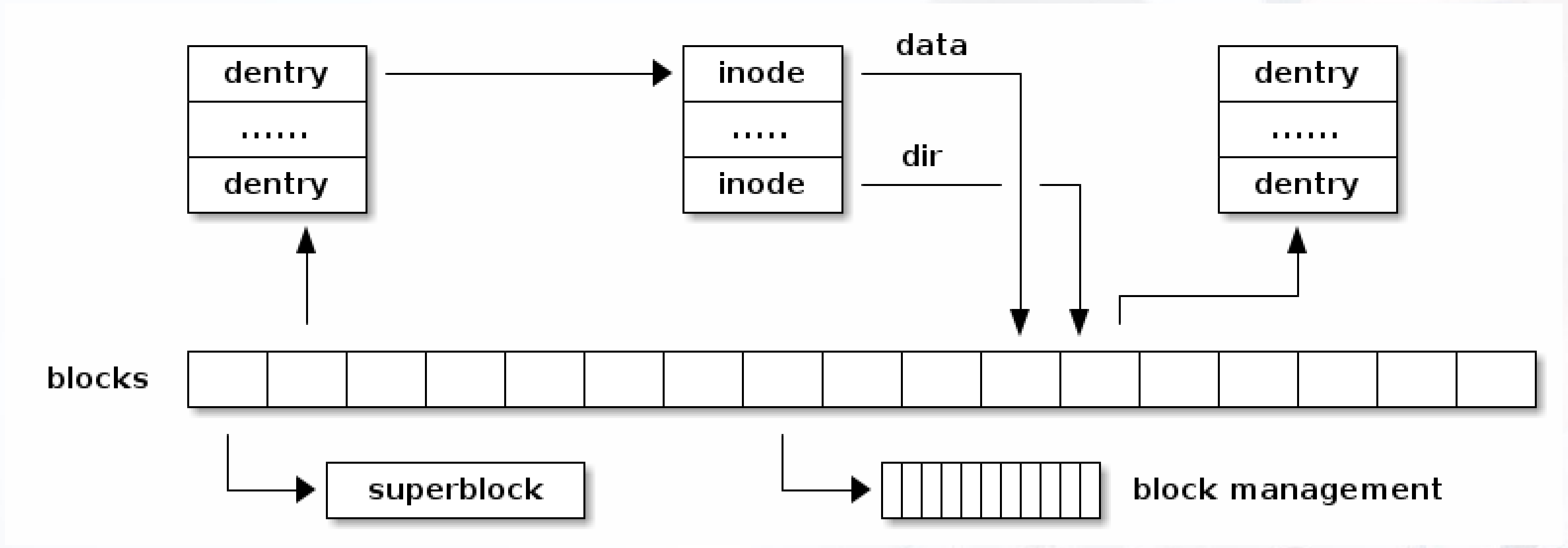
- The *superblock* abstraction contains information about the filesystem instance such as the block size, the root inode, filesystem size. It is present both on storage and in memory (for caching purposes).
- The *file* abstraction contains information about an opened file such as the current file pointer. 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



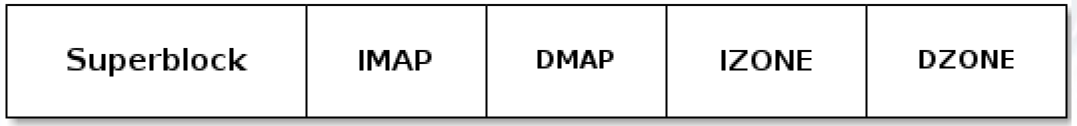
- Multiple file descriptors can point to the same file because we can use the dup() system call to duplicate a file descriptor.
- Multiple file abstractions can point to the same dentry if we use the same path multiple times.
- Multiple dentries can point to the same inode when hard links are used.

File system abstractions on storage (disk)



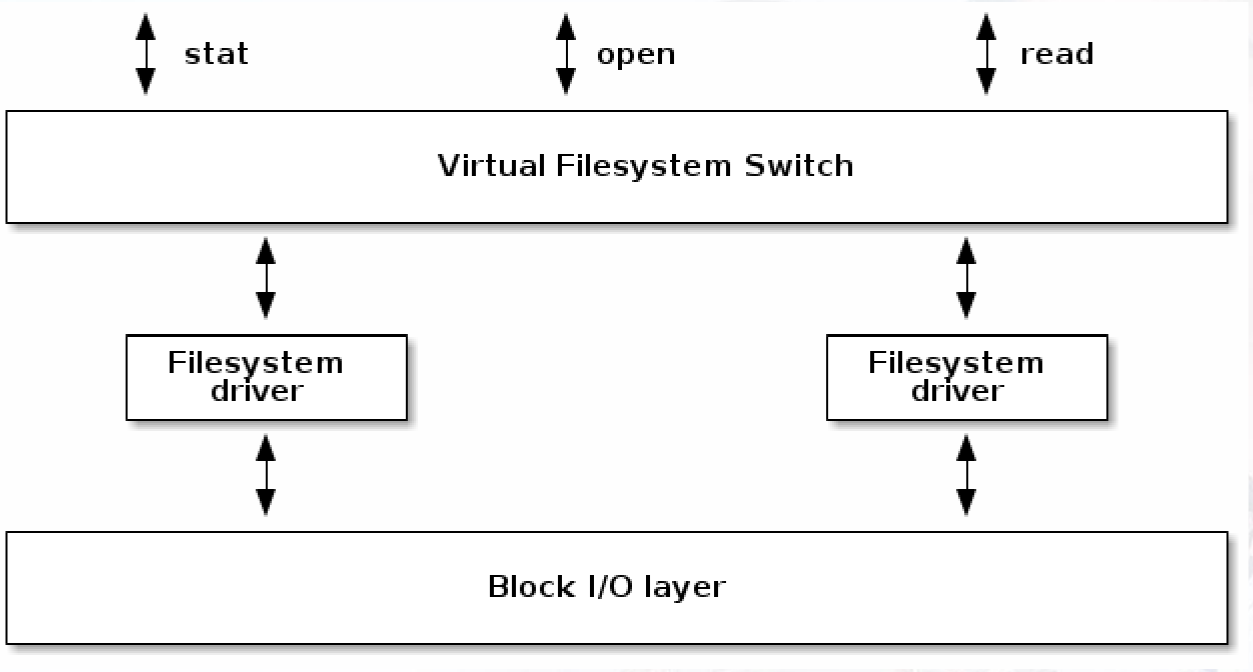
- *superblock* is typically stored at the beginning of the filesystem
- Various blocks are used with different purposes:
  - some to store dentries,
  - some to store inodes and some to store user data blocks.
  - Some blocks used to manage the available free blocks (e.g. bitmaps for the simple filesystems).

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

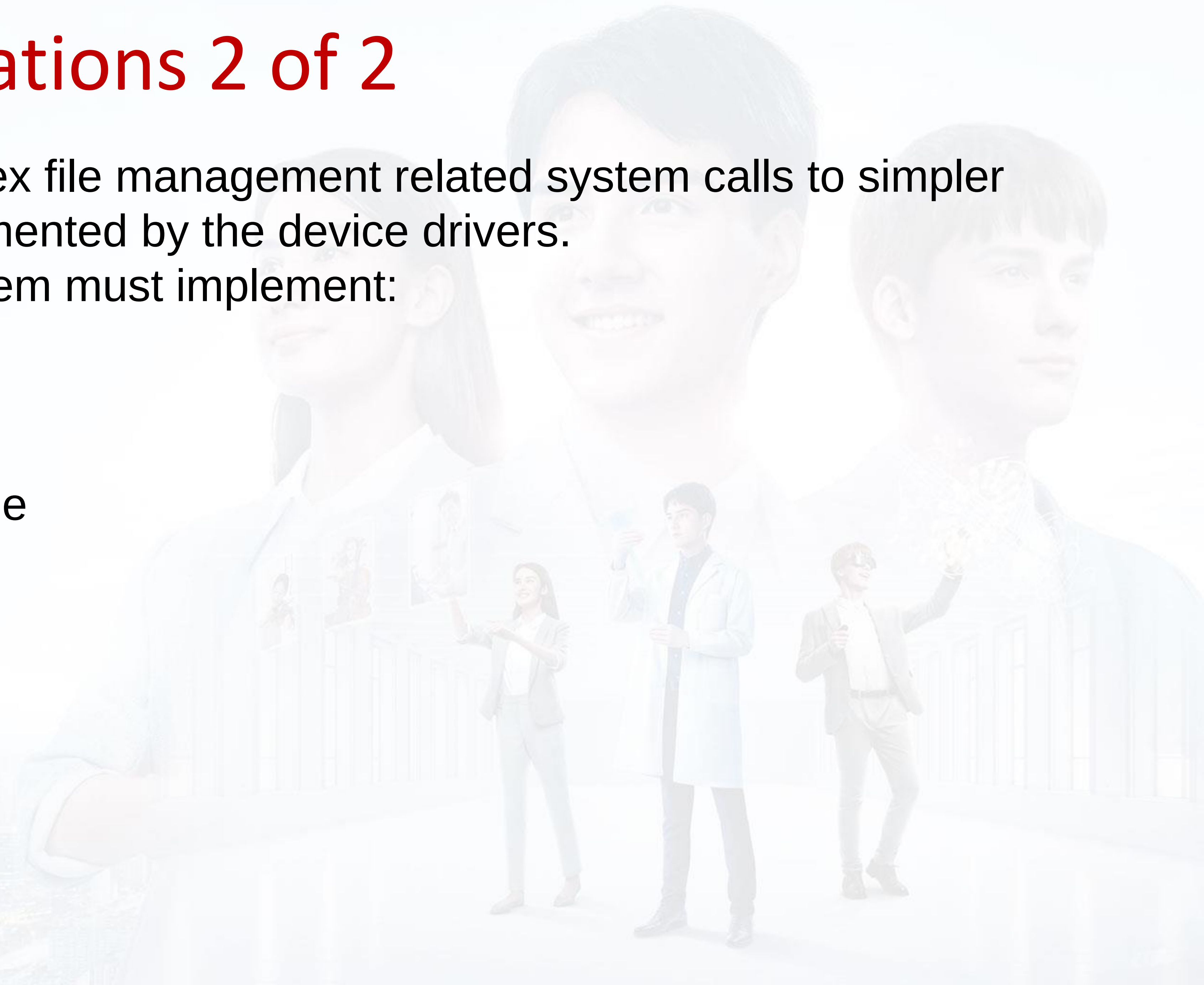
File system operations 1 of 2



- 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).

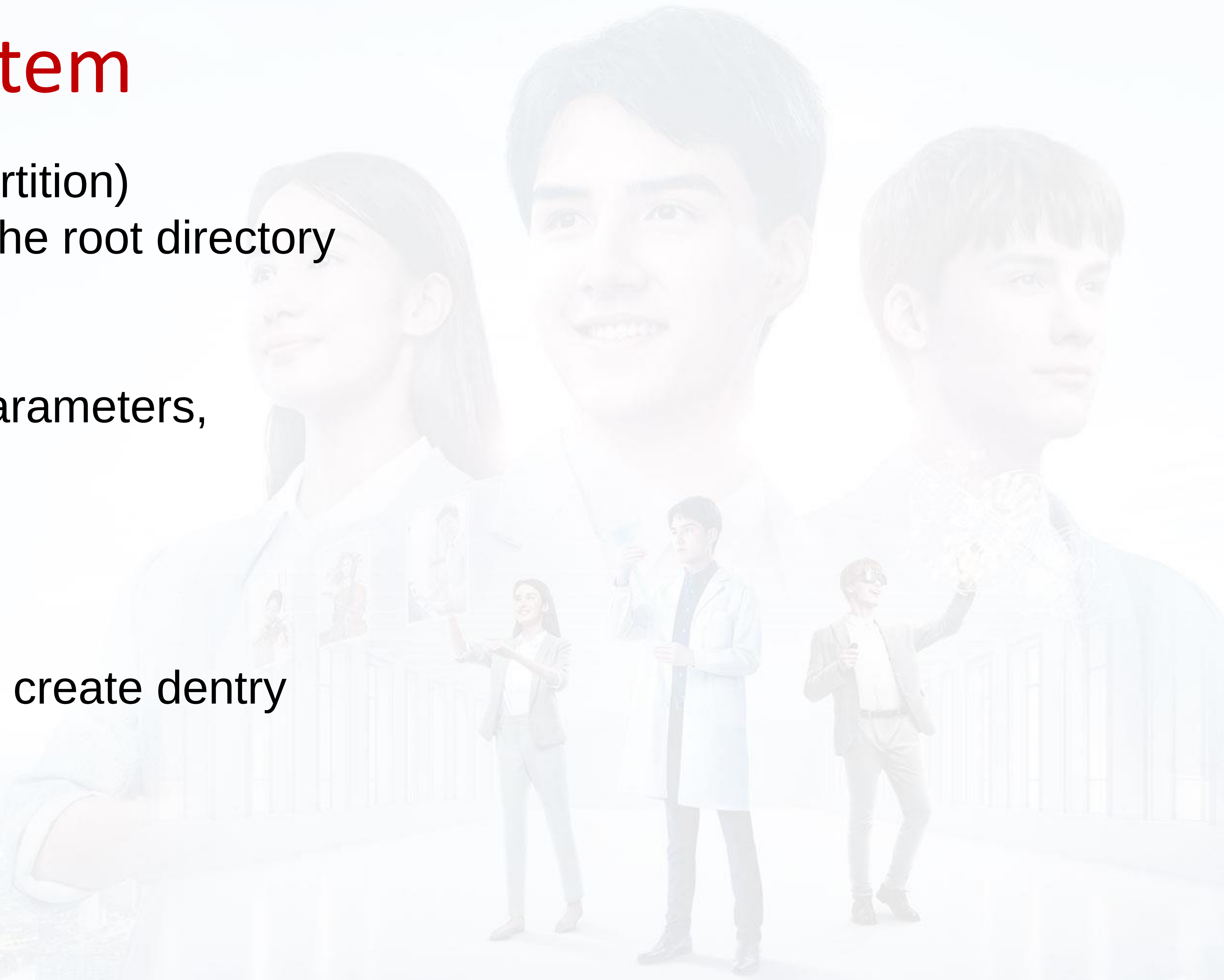
File system operations 2 of 2

- 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.



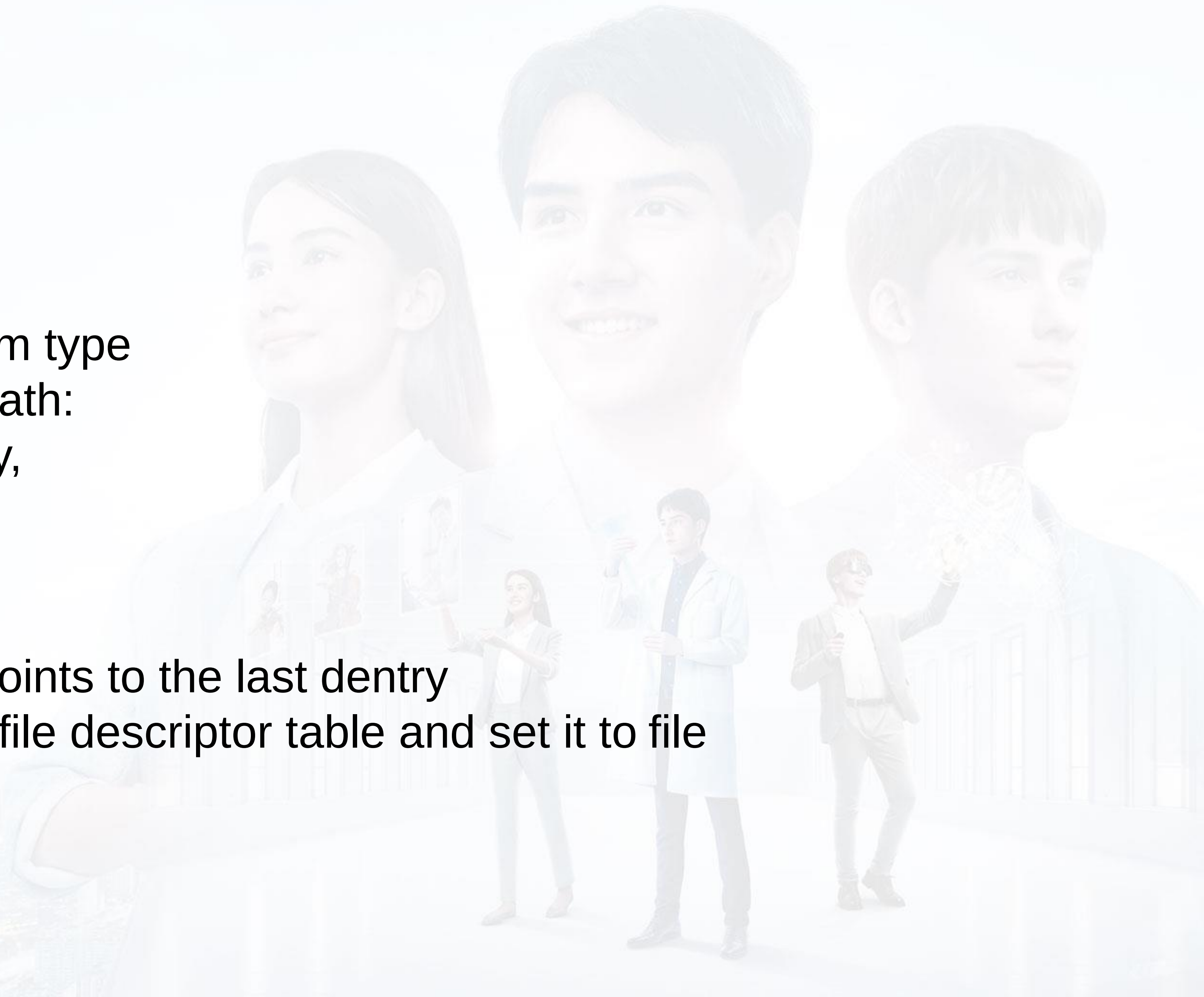
Mounting file system

- 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



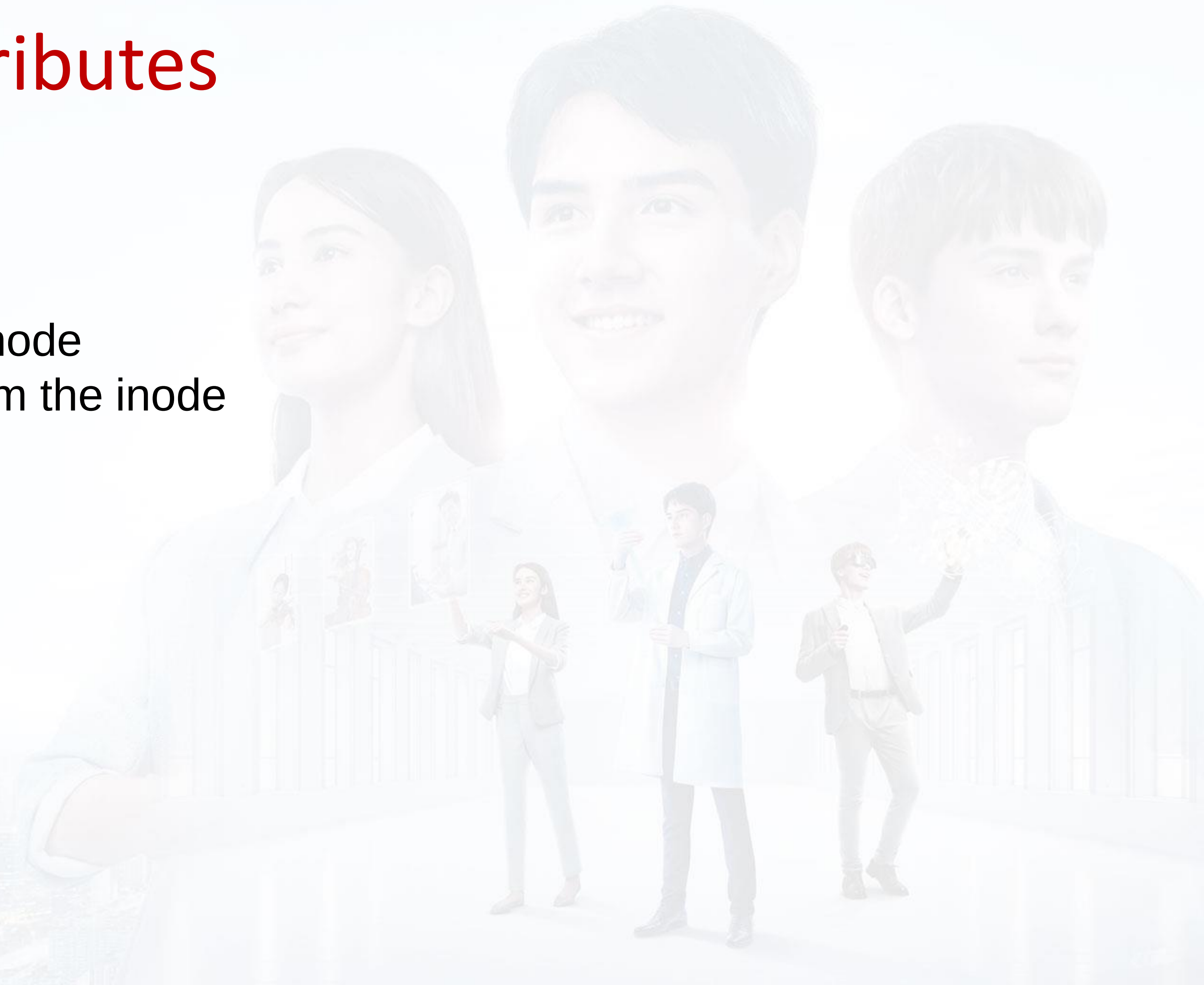
Opening a file

- 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



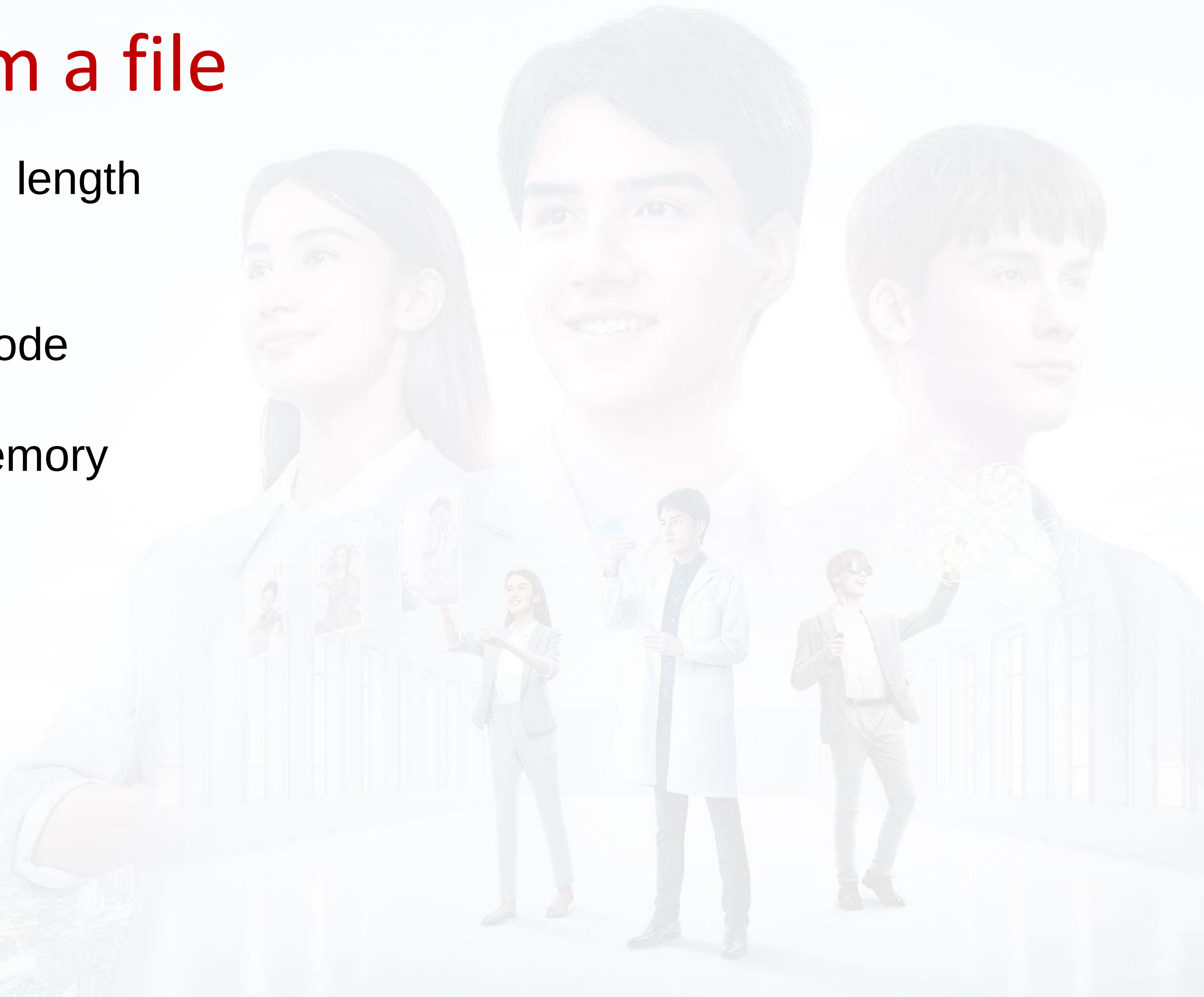
Querying file attributes

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



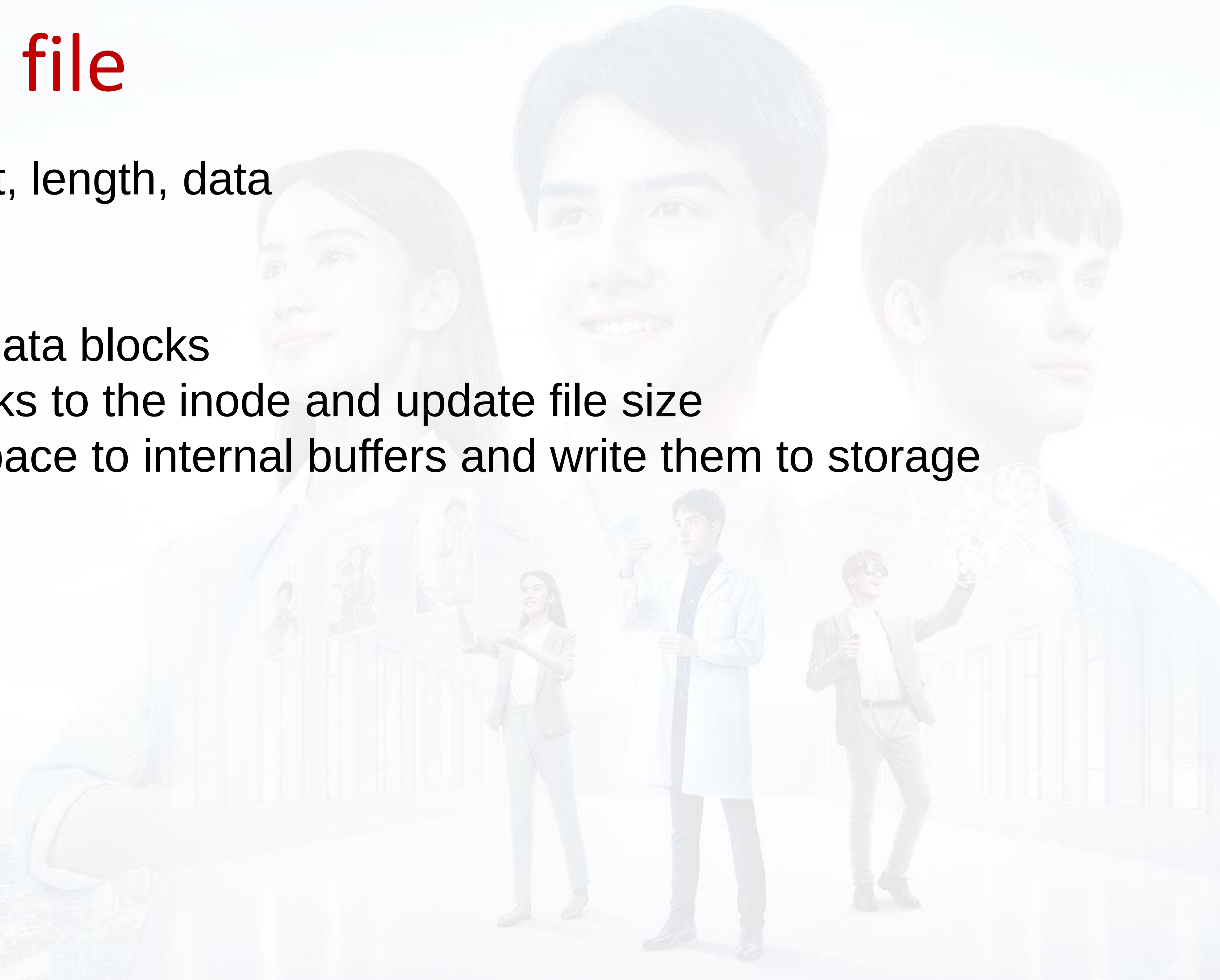
Reading data from a file

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



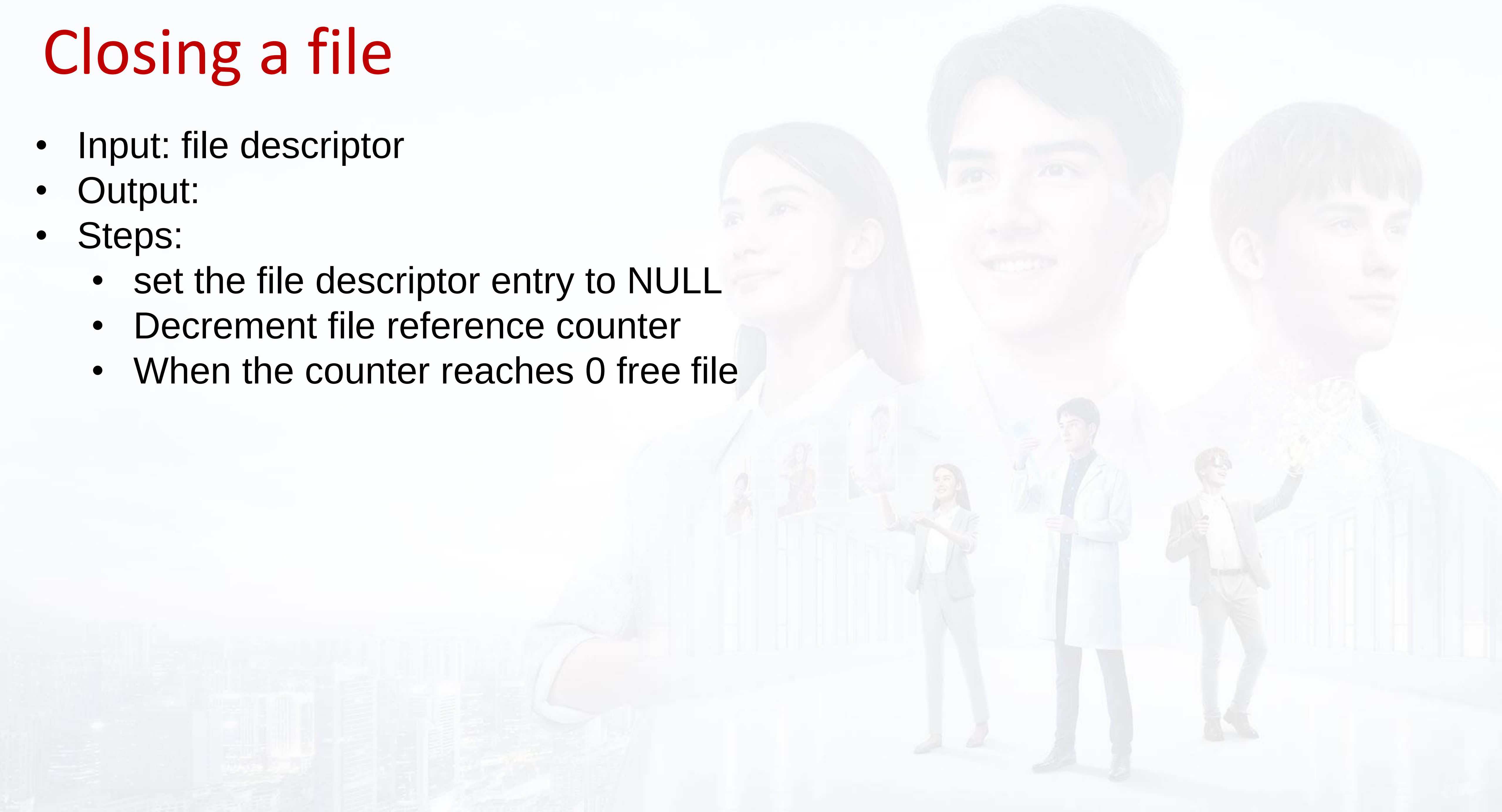
Writing data to a file

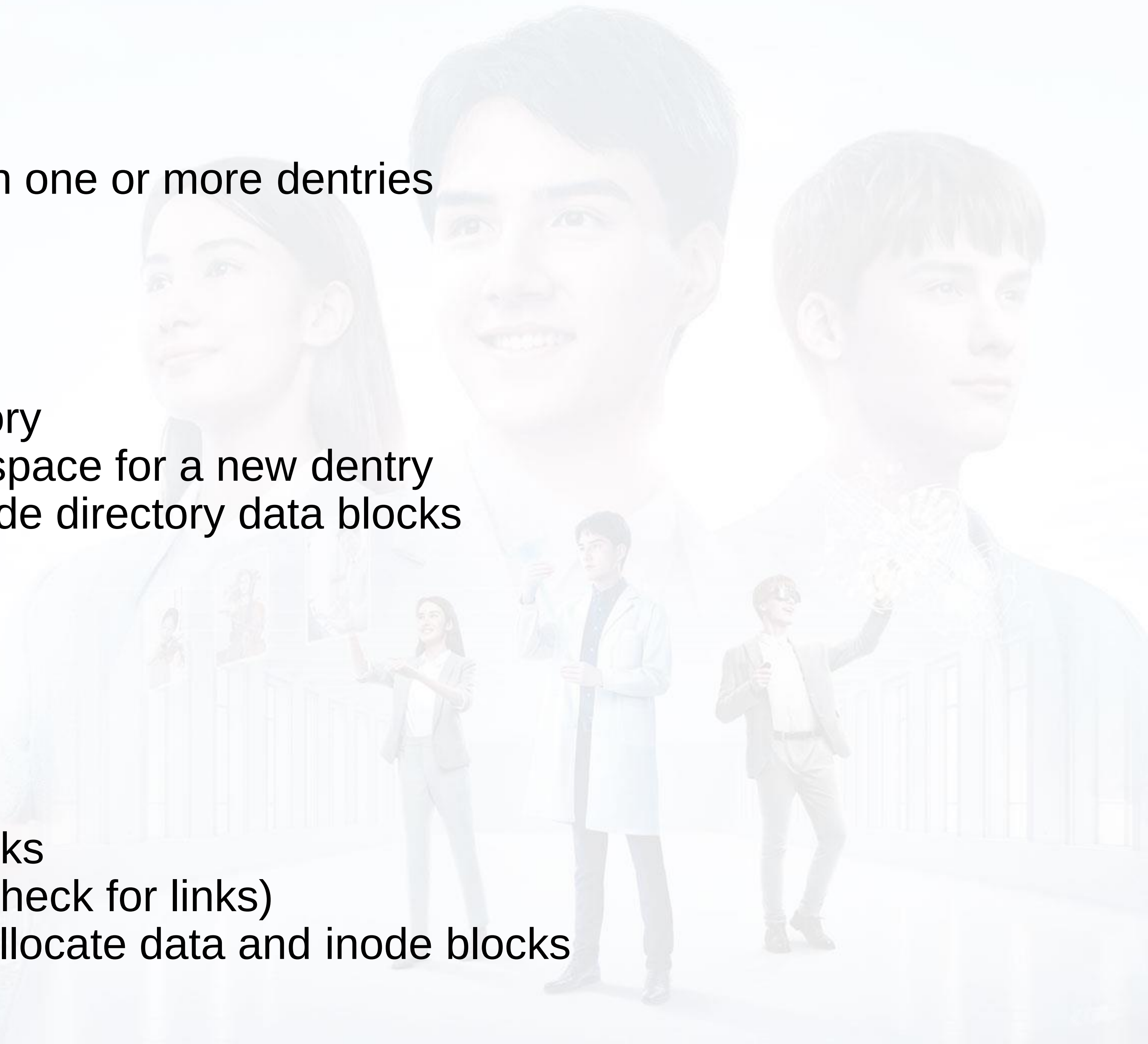
- Input: file descriptor, offset, length, data
- Output:
- Steps:
  - Allocate one or more data blocks
  - Add the allocated blocks to the inode and update file size
  - Copy data from userspace to internal buffers and write them to storage



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

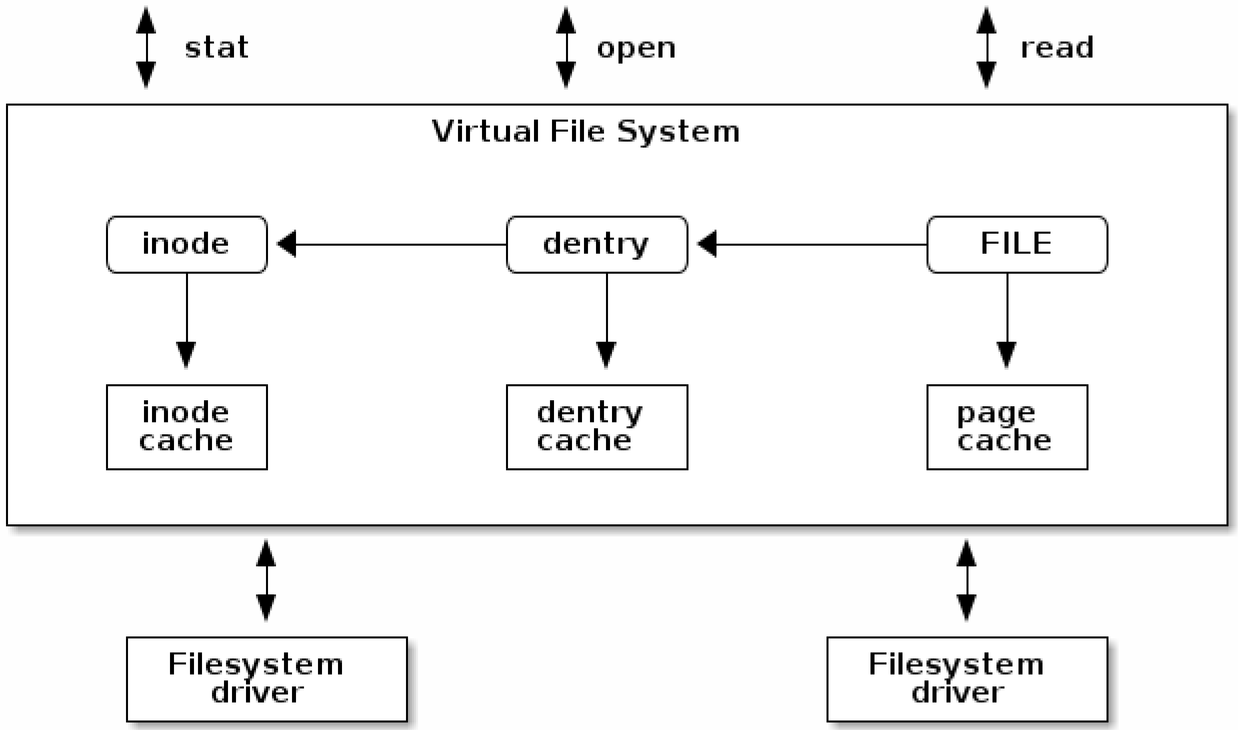
- They are special files which contain one or more dentries
- Creating a file:
  - Input: path
  - Output:
  - Steps:
    - Determine the inode directory
    - Read data blocks and find space for a new dentry
    - Write back the modified inode directory data blocks

- 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.

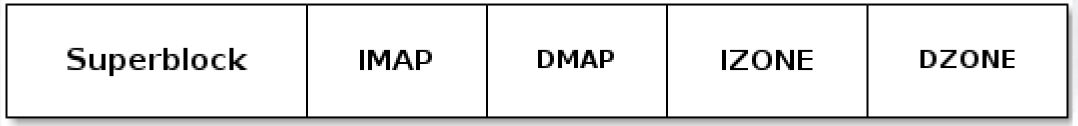
- In order to deal with multiple filesystem types, VFS introduced the common filesystem abstractions (presented above).
- Filesystem driver can also use its own particular fileystem abstractions in memory (e.g. ext4 inode or dentry) and that there might be a different abstractions on storage.
- Three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

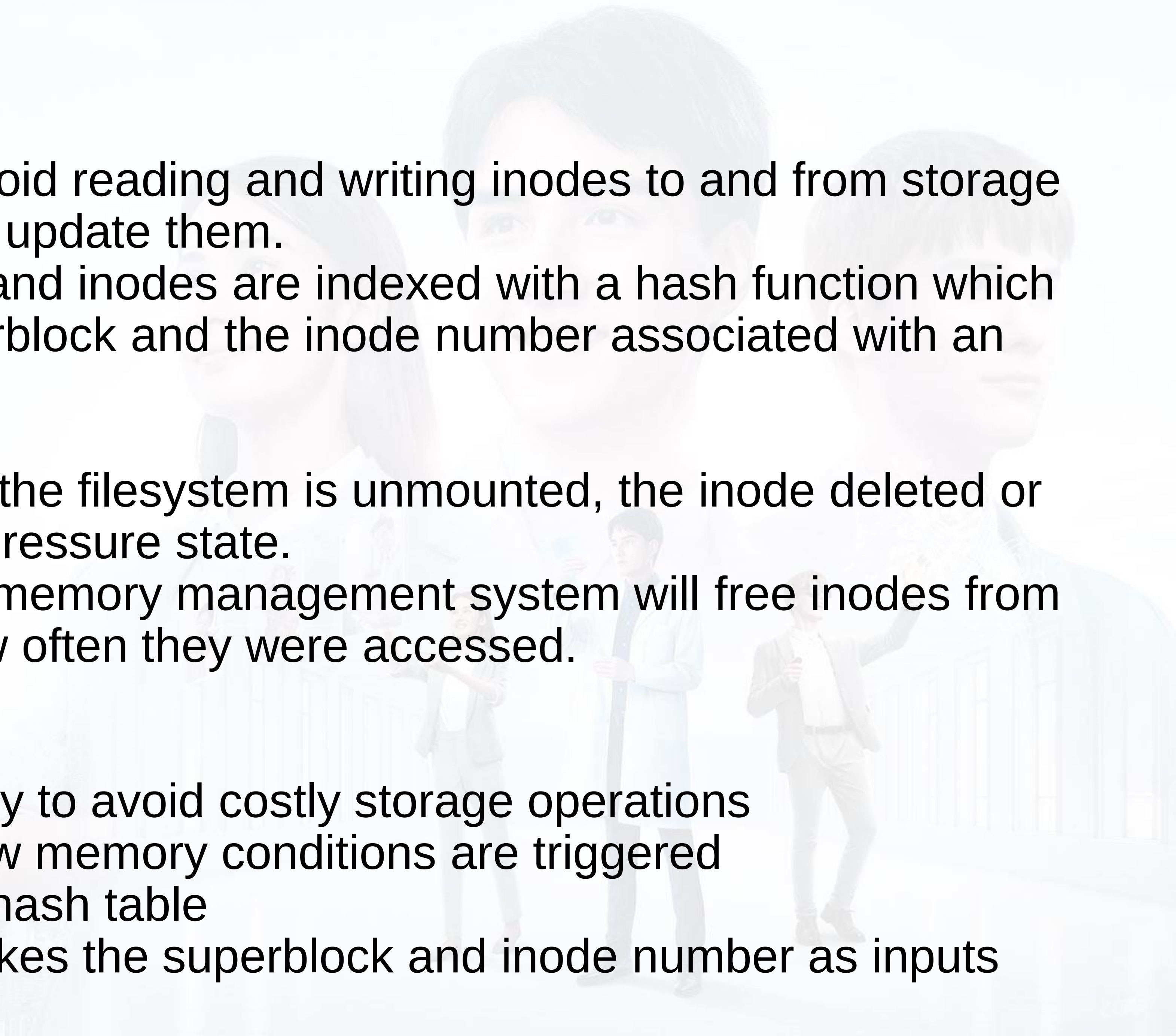
- create() file
- lookup() file name
- link() file
- unlink() or remove file
- symlink()
- mkdir() create directory
- rmdir()
- rename()
- readlink()
- follow_link()
- put_link()

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

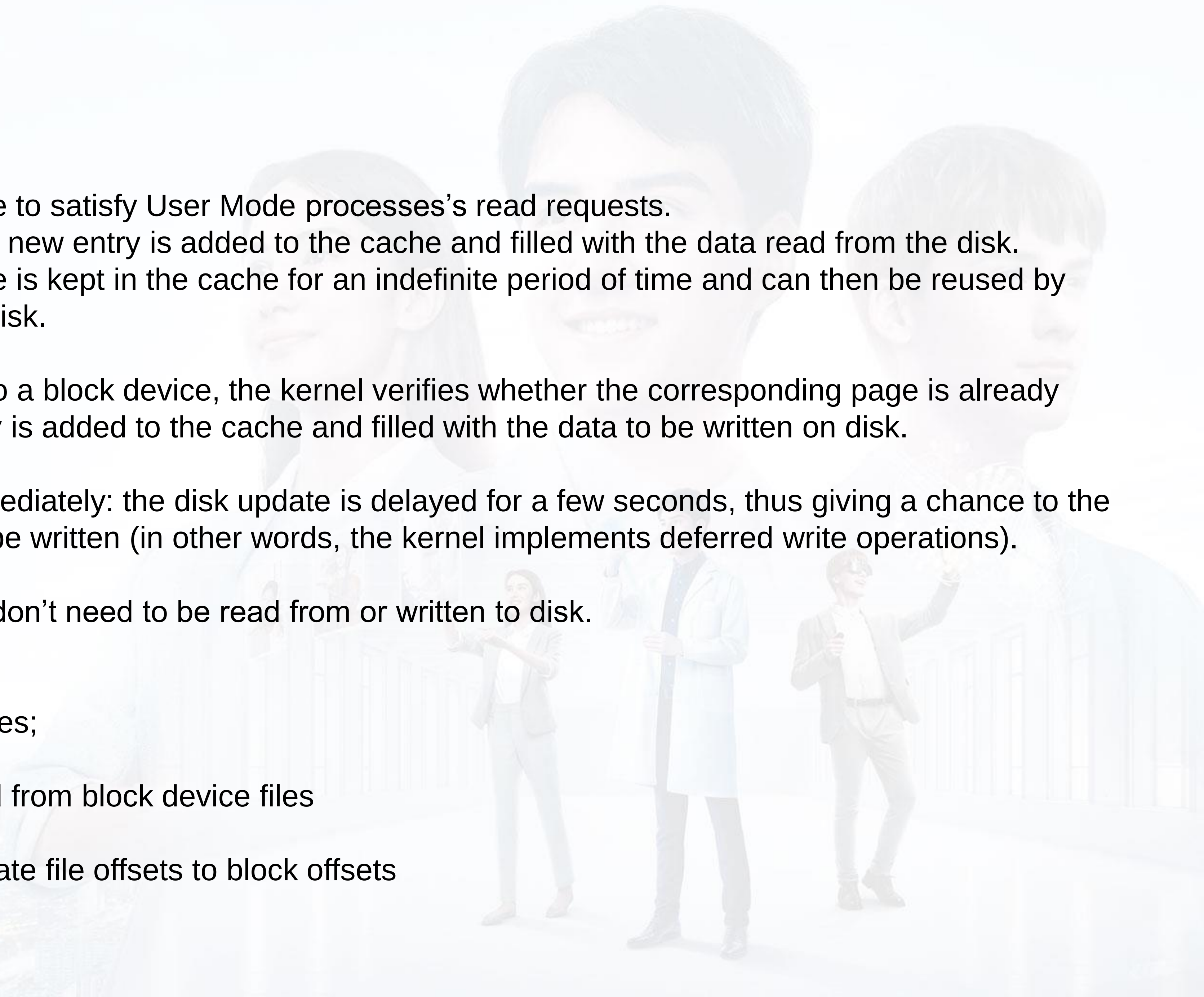


Dentry cache

- 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).

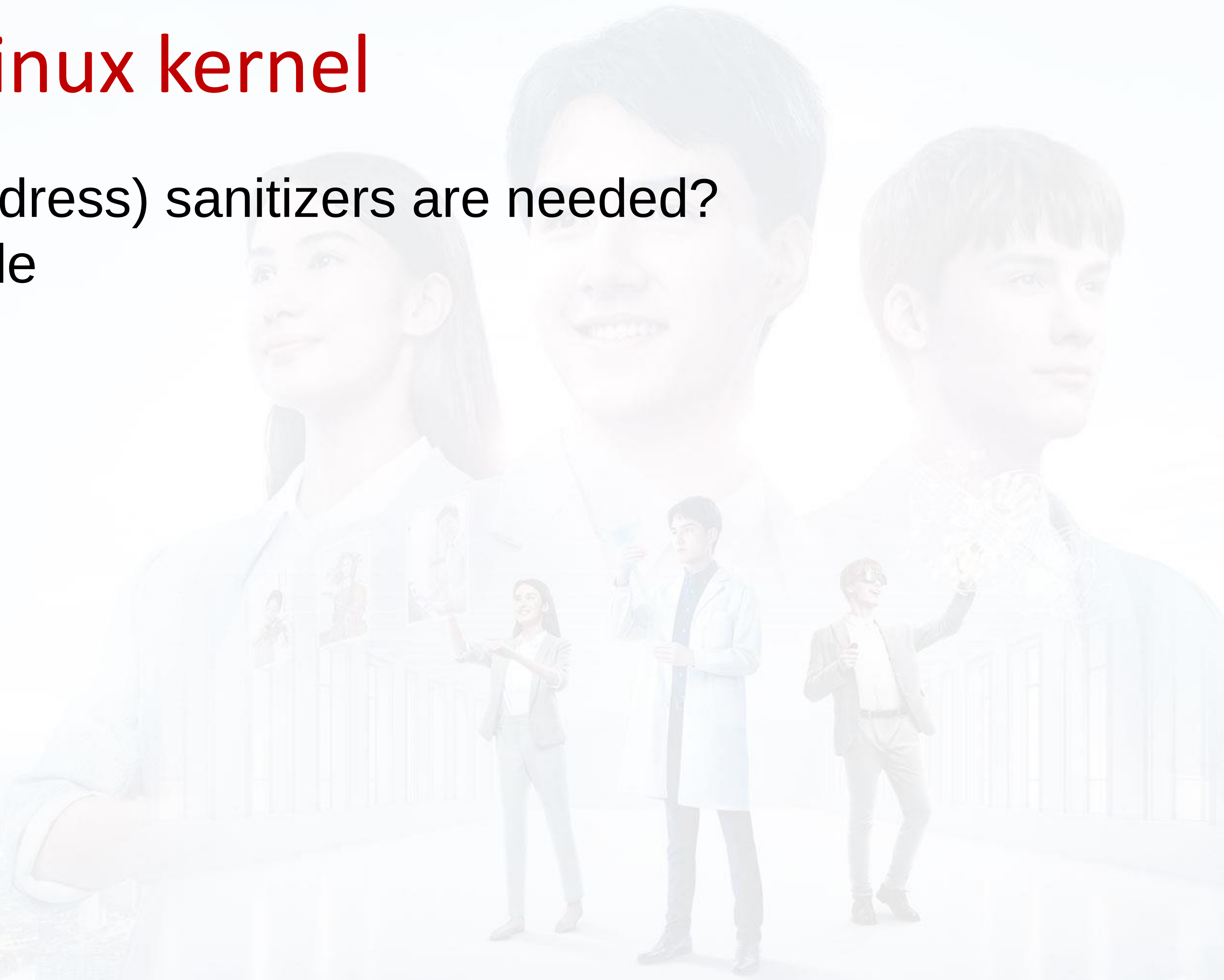
- Kernel code and kernel data structures don't need to be read from or written to disk.

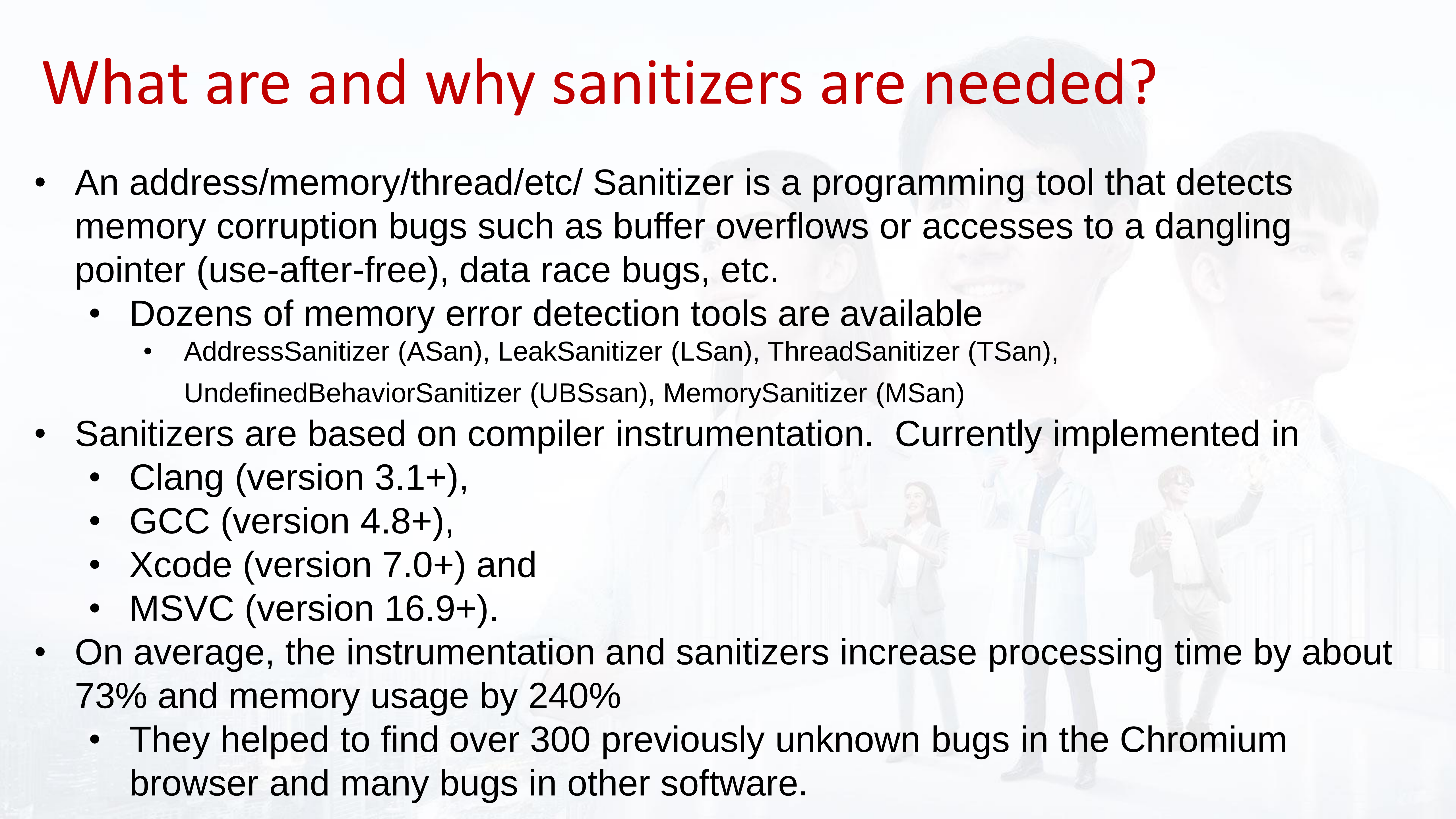
- Pages included in the page cache:
  - Pages containing data of regular files;
  - Pages containing directories;
  - Pages containing data directly read from block device files

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

Sanitizing the Linux kernel

- What are and why (address) sanitizers are needed?
  - Sanitizers by Google
- KFENCE
  - D.U.M.A
- KASAN

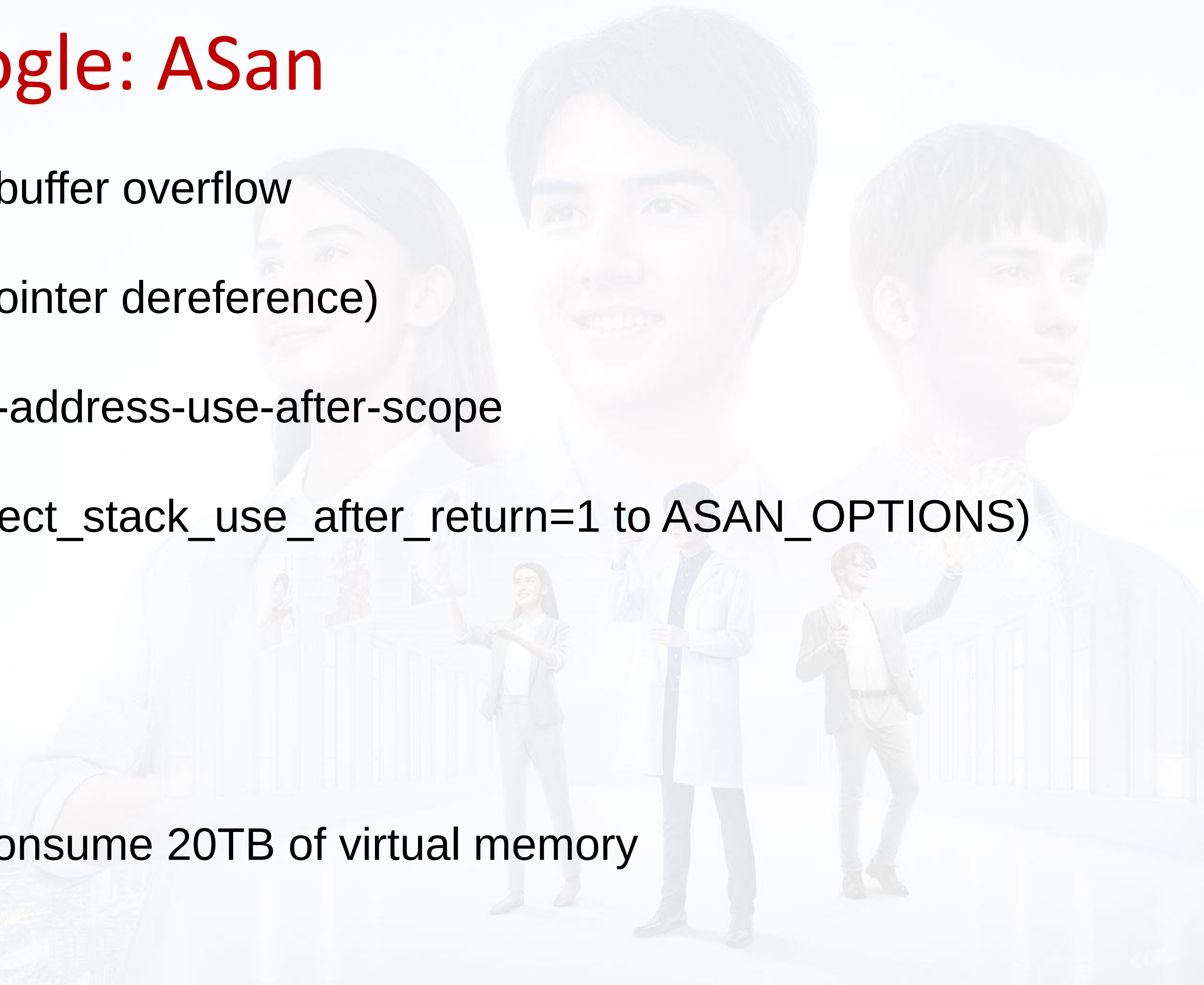




What are and why sanitizers are needed?

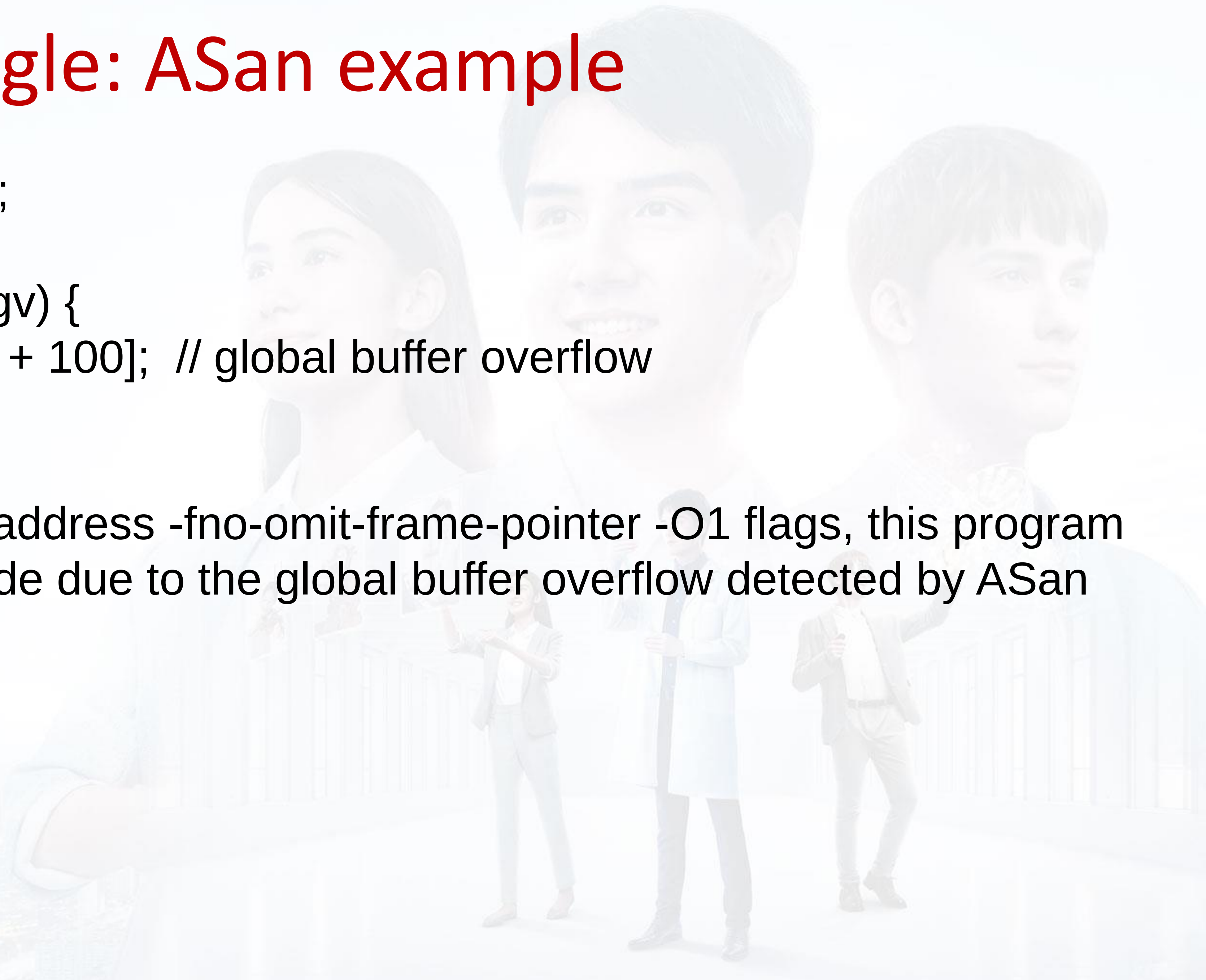
- An address/memory/thread/etc/ Sanitizer is a programming tool that detects memory corruption bugs such as buffer overflows or accesses to a dangling pointer (use-after-free), data race bugs, etc.
  - Dozens of memory error detection tools are available
    - AddressSanitizer (ASan), LeakSanitizer (LSan), ThreadSanitizer (TSan), UndefinedBehaviorSanitizer (UBSsan), MemorySanitizer (MSan)
- Sanitizers are based on compiler instrumentation. Currently implemented in
  - Clang (version 3.1+),
  - GCC (version 4.8+),
  - Xcode (version 7.0+) and
  - MSVC (version 16.9+).
- On average, the instrumentation and sanitizers increase processing time by about 73% and memory usage by 240%
  - They helped to find over 300 previously unknown bugs in the Chromium browser and many bugs in other software.

Sanitizers by Google: ASan



- Heap-, stack-, and global buffer overflow
- Use-after-free (dangling pointer dereference)
- Use-after-scope -fsanitize-address-use-after-scope
- Use-after-return (pass detect_stack_use_after_return=1 to ASAN_OPTIONS)
- Double free, invalid free
- Initialization order bugs
- ASan-ified binaries may consume 20TB of virtual memory

Sanitizers by Google: ASan example



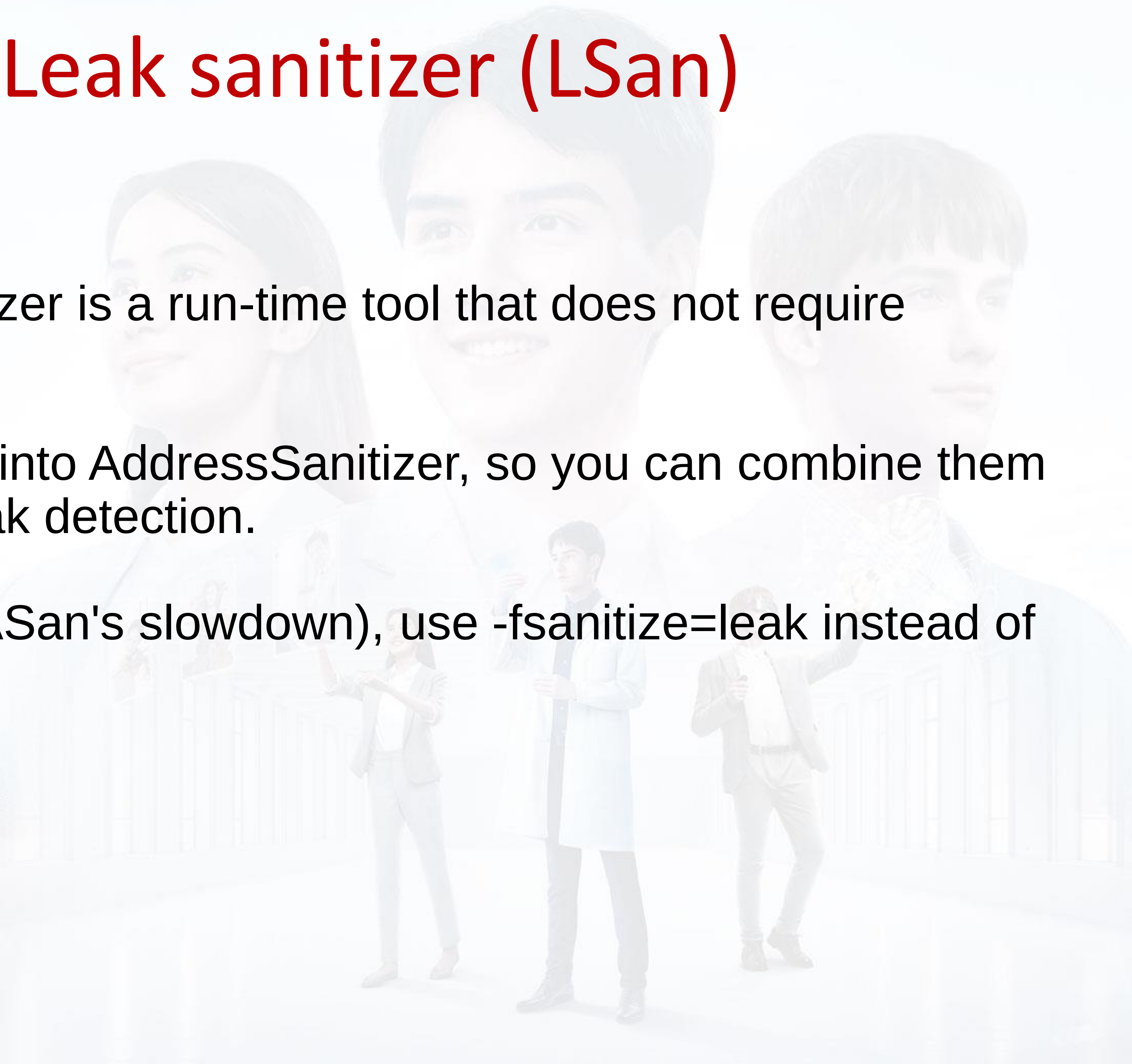
- int global_array[100] = {-1};

- int main(int argc, char **argv) {
-     return global_array[argc + 100]; // global buffer overflow
- }

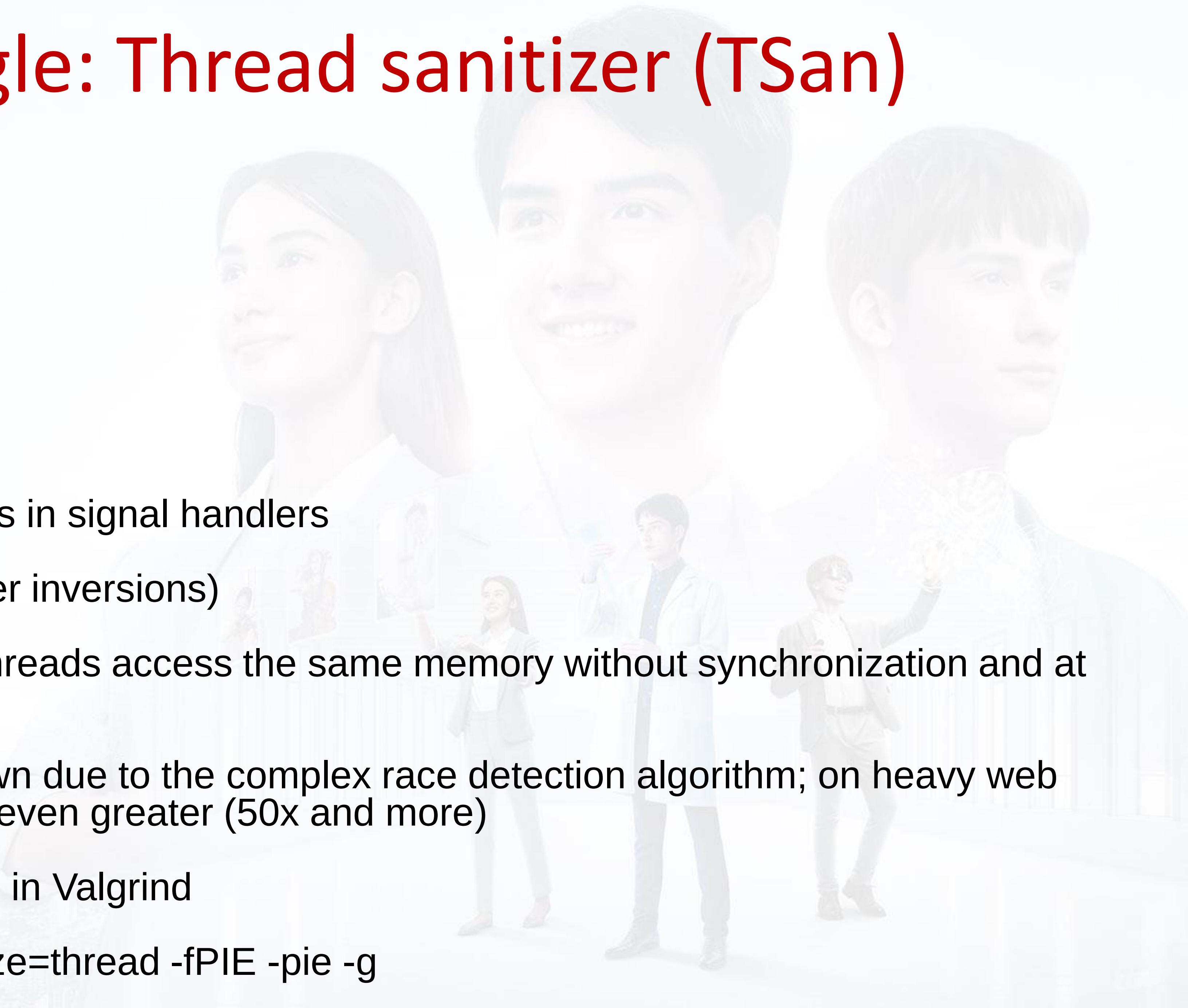
- When built with -fsanitize=address -fno-omit-frame-pointer -O1 flags, this program will exit with a non-zero code due to the global buffer overflow detected by ASan

Sanitizers by Google: Leak sanitizer (LSan)

- It is memory leak detector.
- In a stand-alone mode, this Sanitizer is a run-time tool that does not require compiler instrumentation.
- However, LSan is also integrated into AddressSanitizer, so you can combine them to get both memory errors and leak detection.
- To run LSan only (and avoid the ASan's slowdown), use -fsanitize=leak instead of -fsanitize=address
- int main(){
    - int *x = new int(10); // leaked
    - return 0;
- }



Sanitizers by Google: Thread sanitizer (TSan)



- It detects
  - Normal data races
  - Races on C++ object vptr
  - Use after free races
  - Races on mutexes
  - Races on file descriptors
  - Races on pthread_barrier_t
  - Destruction of a locked mutex
  - Leaked threads
  - Signal-unsafe malloc/free calls in signal handlers
  - Signal handler spoils errno
  - Potential deadlocks (lock order inversions)

- Data races occur when multiple threads access the same memory without synchronization and at least one access is a write.

- TSan in Valgrind: 5x–30x slowdown due to the complex race detection algorithm; on heavy web applications the slowdowns were even greater (50x and more)

- TSan in LLVM is much faster than in Valgrind

- To use TSan compile with -fsanitize=thread -fPIE -pie -g

Sanitizers by Google: TSan example

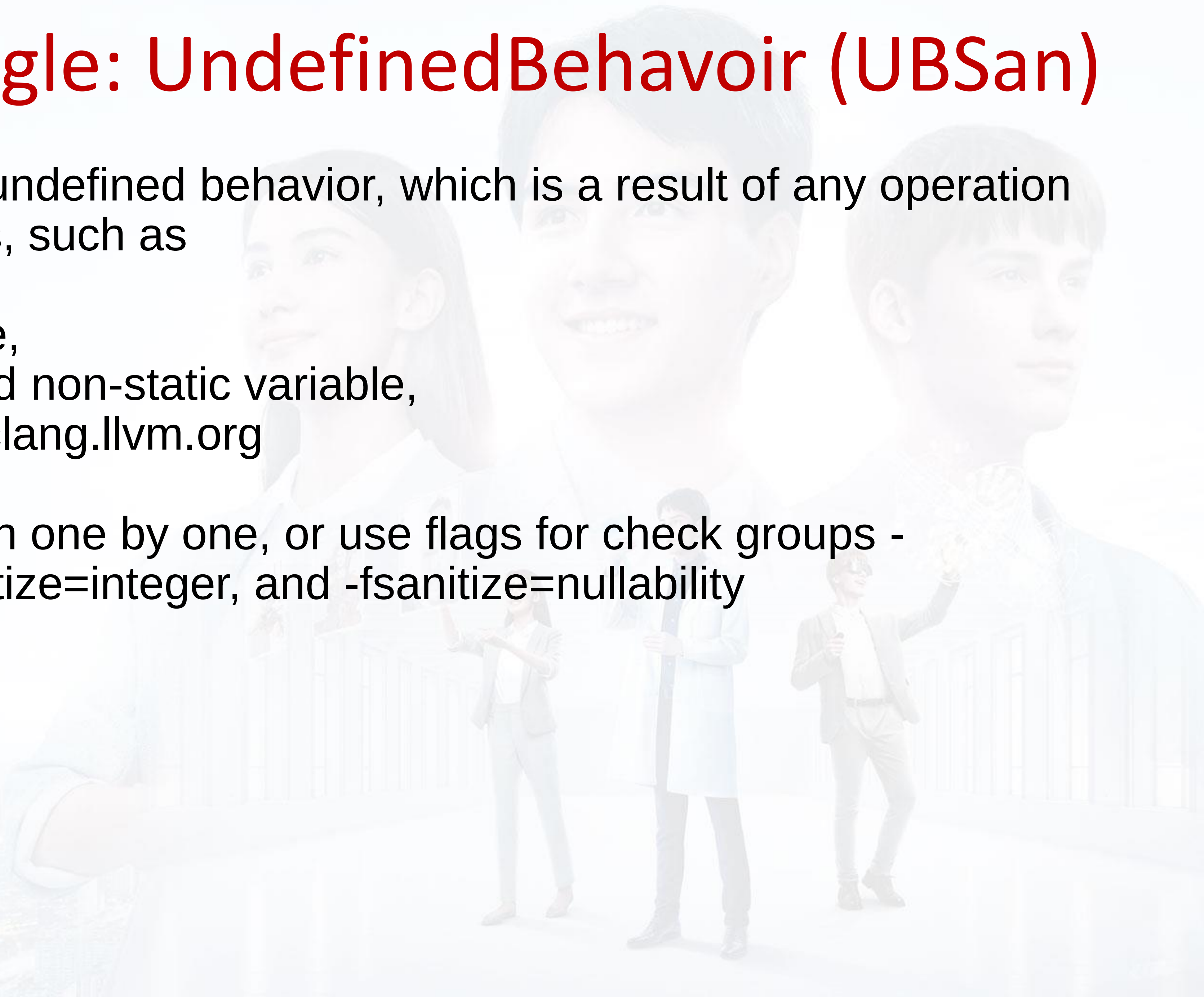
```
•   #include <pthread.h>
•   #include <stdio.h>

•   int Global;

•   void *Thread1(void *x) {
•       Global++;
•       return NULL;
•   }

•   void *Thread2(void *x) {
•       Global--;
•       return NULL;
•   }

•   int main() {
•       pthread_t t[2];
•       pthread_create(&t[0], NULL, Thread1, NULL);
•       pthread_create(&t[1], NULL, Thread2, NULL);
•       pthread_join(t[0], NULL);
•       pthread_join(t[1], NULL);
•   }
```



Sanitizers by Google: UndefinedBehavoir (UBSan)

- It is a runtime checker for undefined behavior, which is a result of any operation with unspecified semantics, such as
  - dividing by zero,
  - null pointer dereference,
  - usage of an uninitialized non-static variable,
  - etc., see the full list at clang.llvm.org

- One can turn the checks on one by one, or use flags for check groups -fsanitize=undefined, -fsanitize=integer, and -fsanitize=nullability

- int main() {
-     int i = 2048;
-     i <<= 28;
-     return 0;
- }

Sanitizers by Google: Memory Sanitizer (MSan)

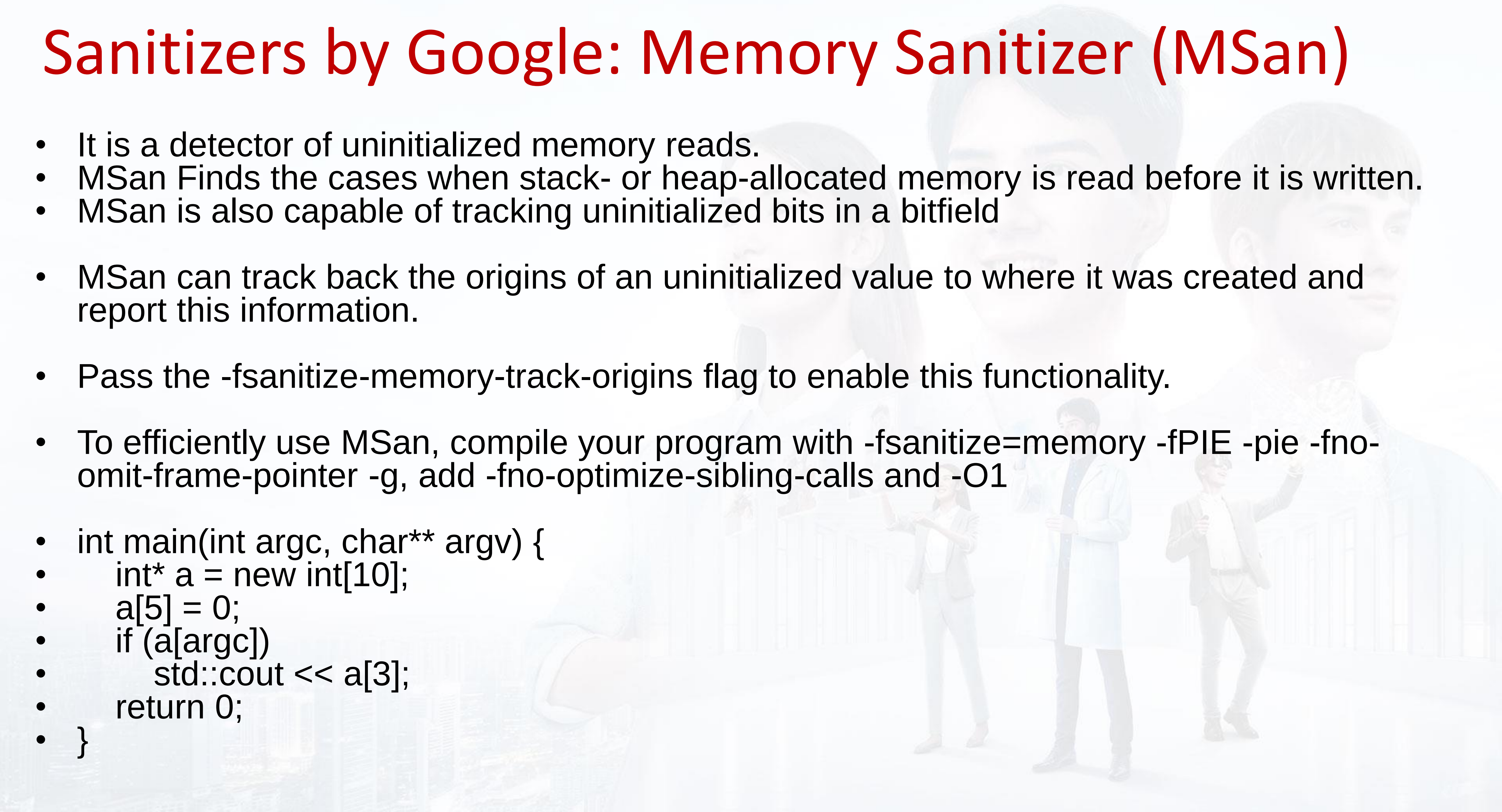
- It is a detector of uninitialized memory reads.
- MSan Finds the cases when stack- or heap-allocated memory is read before it is written.
- MSan is also capable of tracking uninitialized bits in a bitfield

- MSan can track back the origins of an uninitialized value to where it was created and report this information.

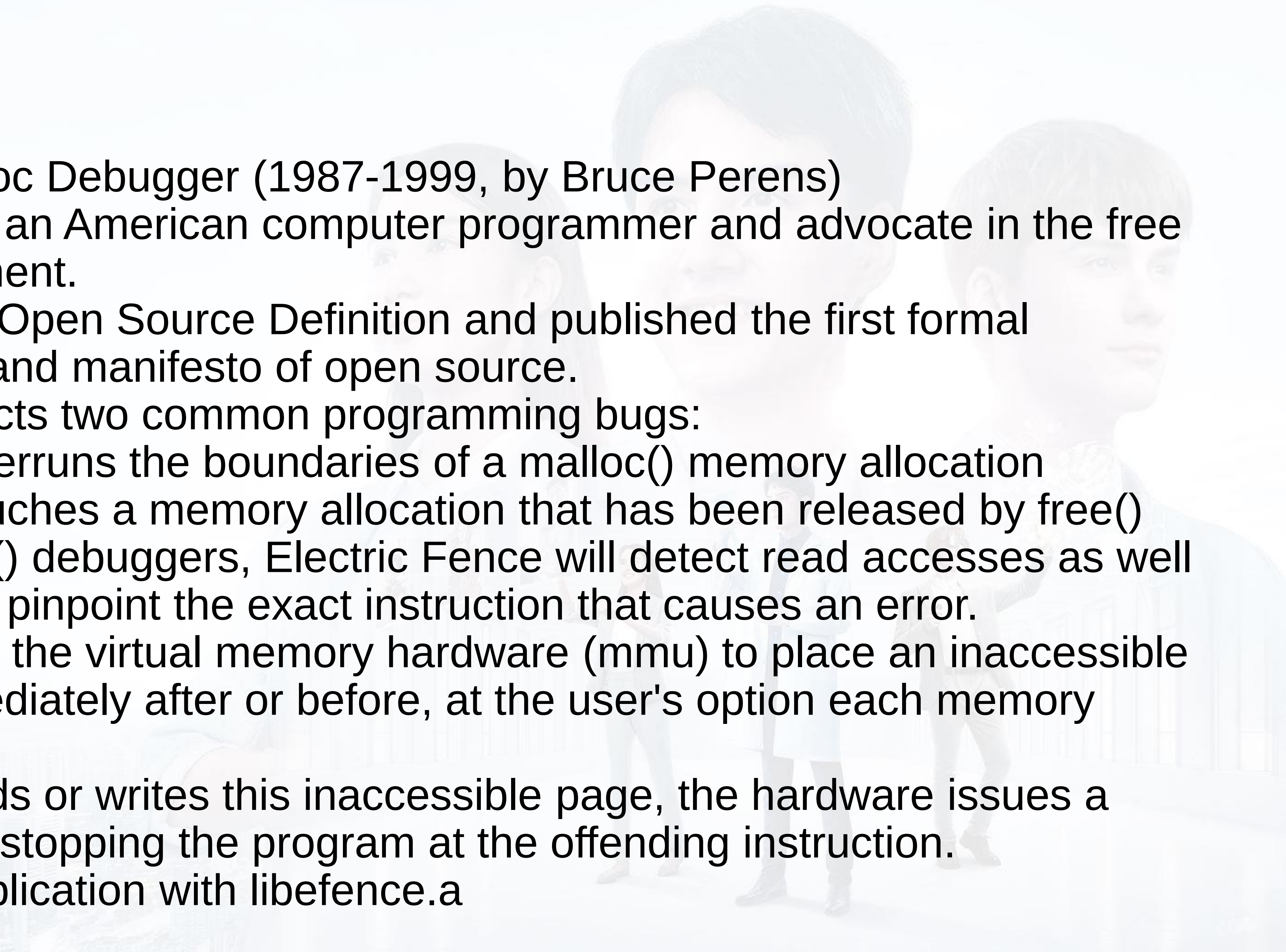
- Pass the -fsanitize-memory-track-origins flag to enable this functionality.

- To efficiently use MSan, compile your program with -fsanitize=memory -fPIE -pie -fno-omit-frame-pointer -g, add -fno-optimize-sibling-calls and -O1

```
• int main(int argc, char** argv) {
•     int* a = new int[10];
•     a[5] = 0;
•     if (a[argc])
•         std::cout << a[3];
•     return 0;
• }
```

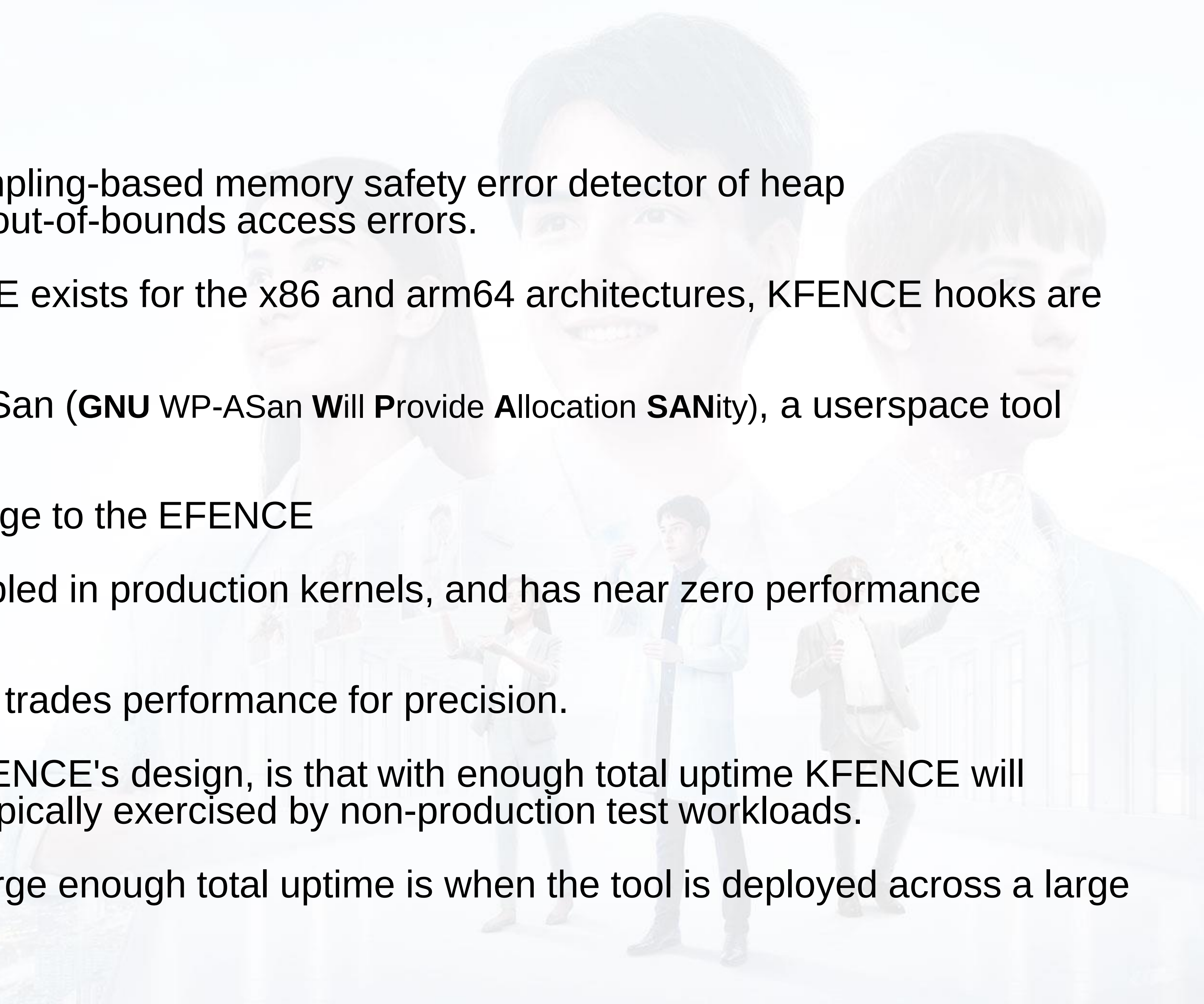


EFENCE



- Electric Fence Malloc Debugger (1987-1999, by Bruce Perens)
  - Bruce Perens is an American computer programmer and advocate in the free software movement.
  - He created The Open Source Definition and published the first formal announcement and manifesto of open source.
- Electric Fence detects two common programming bugs:
  - software that overruns the boundaries of a malloc() memory allocation
  - software that touches a memory allocation that has been released by free()
- Unlike other malloc() debuggers, Electric Fence will detect read accesses as well as writes, and it will pinpoint the exact instruction that causes an error.
- Electric Fence uses the virtual memory hardware (mmu) to place an inaccessible memory page immediately after or before, at the user's option each memory allocation.
- When software reads or writes this inaccessible page, the hardware issues a segmentation fault, stopping the program at the offending instruction.
- Simply link your application with libefence.a

KFENCE 1 of 4



- KFENCE is a low-overhead sampling-based memory safety error detector of heap
- use-after-free, invalid-free, and out-of-bounds access errors.

- Since Linux kernel 5.12 KFENCE exists for the x86 and arm64 architectures, KFENCE hooks are in SLAB and SLUB allocators.

- KFENCE is inspired by GWP-ASan (**GNU** WP-ASan **W**ill **P**rovide **A**llocation **SAN**ity), a userspace tool with similar properties.

- The name "KFENCE" is a homage to the EFENCE

- KFENCE is designed to be enabled in production kernels, and has near zero performance overhead.

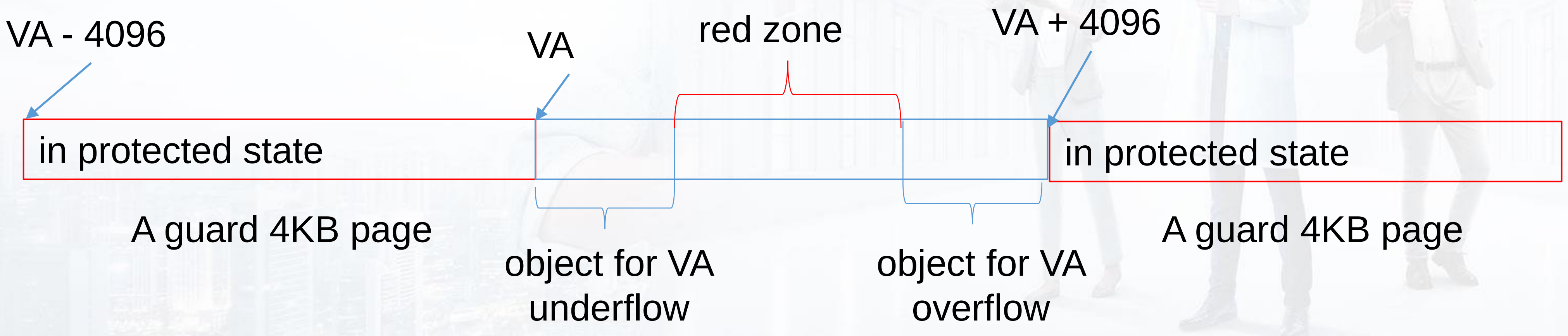
- Compared to KASAN, KFENCE trades performance for precision.

- The main motivation behind KFENCE's design, is that with enough total uptime KFENCE will detect bugs in code paths not typically exercised by non-production test workloads.

- One way to quickly achieve a large enough total uptime is when the tool is deployed across a large fleet of machines.

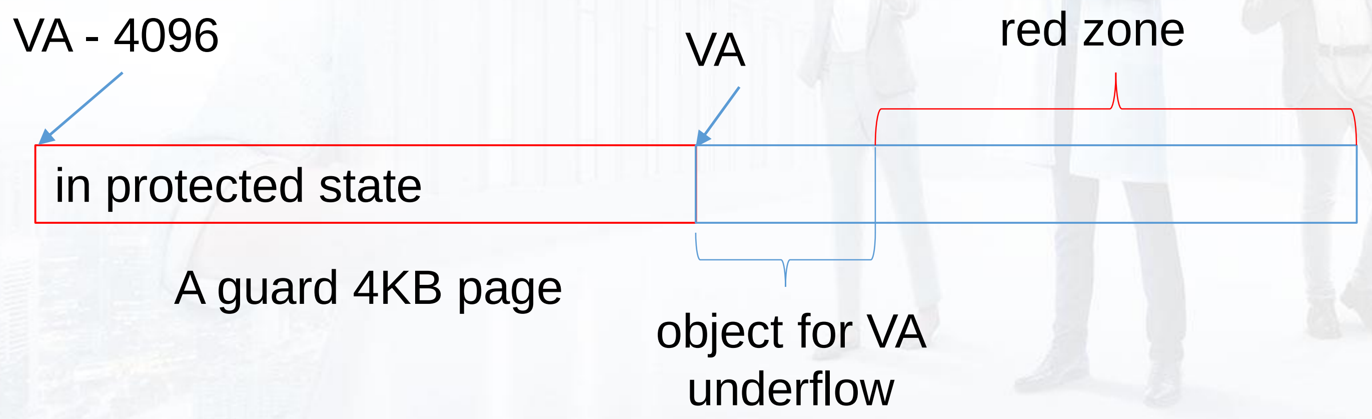
KFENCE 2 of 4

- KFENCE objects each reside on a dedicated page, at either the left or
- right page boundaries.
- The pages to the left and right of the object page are "guard pages", whose attributes are changed to a protected state, and cause page faults on any attempted access to them.
- Such page faults are then intercepted by KFENCE, which handles the fault
- gracefully by reporting a memory access error.
- Each object requires 2 pages, one for the object itself and the other one used as a guard page
- On architectures that support huge pages, KFENCE will ensure that the pool is using pages of size PAGE_SIZE. This will result in additional page tables being allocated.

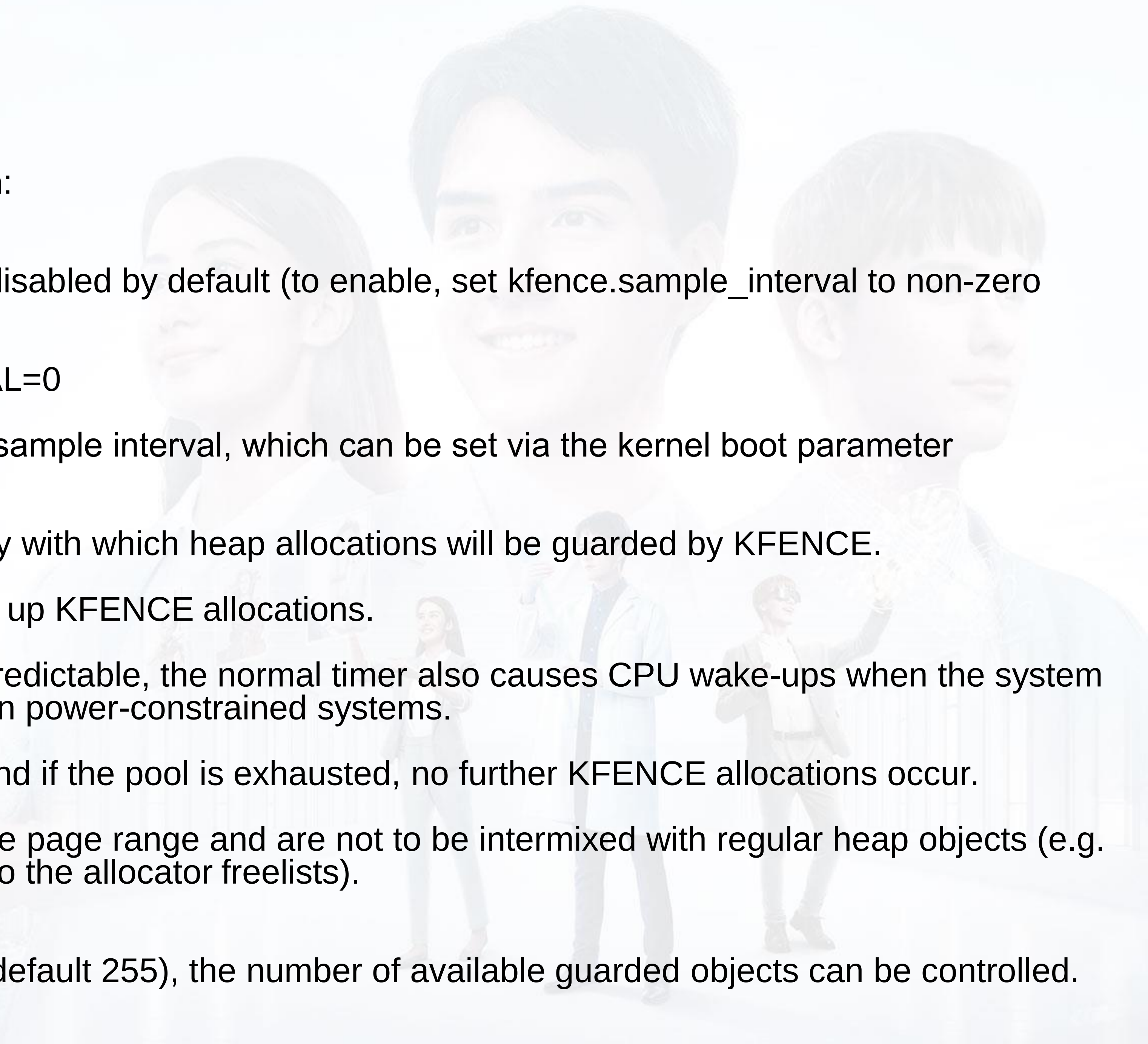


KFENCE 3 of 4

- To detect out-of-bounds writes to memory within the object's page itself, KFENCE also uses pattern-based redzones.
- For each object page, a redzone is set up for all non-object memory.
- For typical alignments, the redzone is only required on the unguarded side of an object.
- Because KFENCE must honor the cache's requested alignment, special alignments may result in unprotected gaps on either side of an object, all of which are redzoned.
- Upon deallocation of a KFENCE object, the object's page is again protected and the object is marked as freed.
- Any further access to the object causes a fault and KFENCE reports a use-after-free access.
- Freed objects are inserted at the tail of KFENCE's free list, so that the least recently freed objects are reused first, and the chances of detecting use-after-frees of recently freed objects is increased



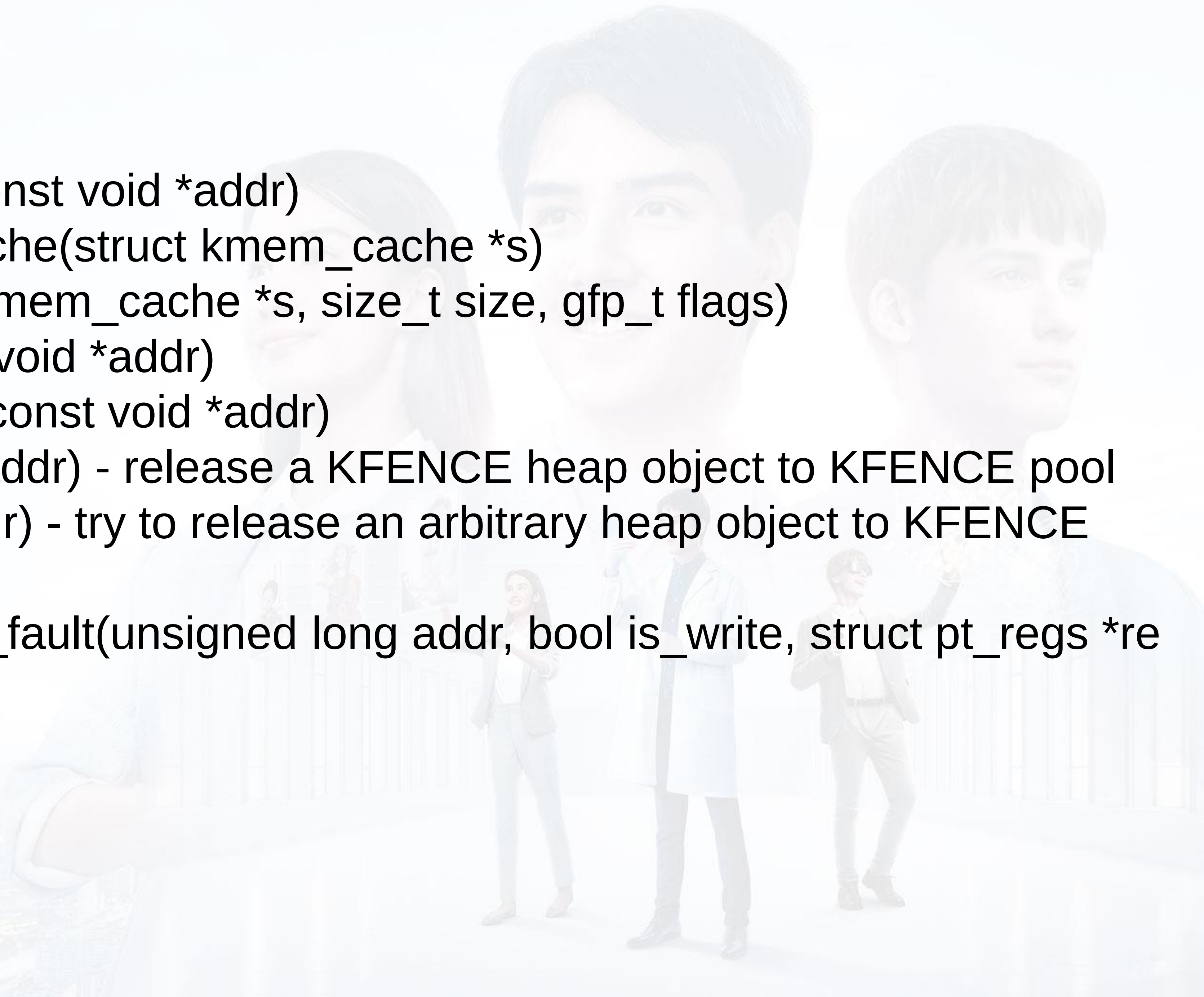
KFENCE 4 of 4

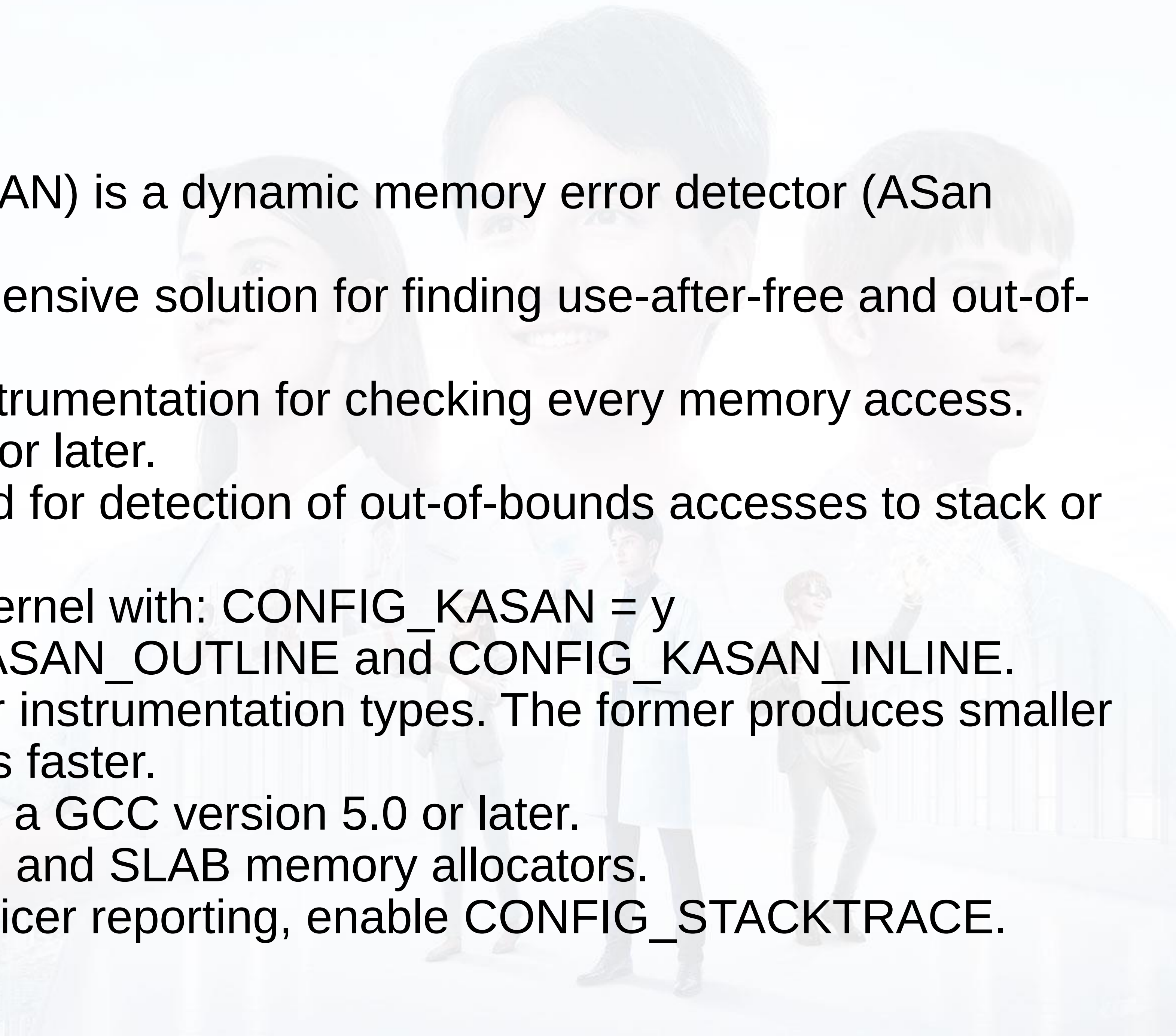


- To enable KFENCE, configure the kernel with:
  - CONFIG_KFENCE=y
- To build a kernel with KFENCE support, but disabled by default (to enable, set kfence.sample_interval to non-zero value), configure the kernel with:
  - CONFIG_KFENCE=y
  - CONFIG_KFENCE_SAMPLE_INTERVAL=0
- The most important parameter is KFENCE's sample interval, which can be set via the kernel boot parameter kfence.sample_interval in milliseconds.
- The sample interval determines the frequency with which heap allocations will be guarded by KFENCE.
- The sample interval controls a timer that sets up KFENCE allocations.
- By default, to keep the real sample interval predictable, the normal timer also causes CPU wake-ups when the system is completely idle. This may be undesirable on power-constrained systems.
- The KFENCE memory pool is of fixed size, and if the pool is exhausted, no further KFENCE allocations occur.
  - KFENCE objects/pages live in a separate page range and are not to be intermixed with regular heap objects (e.g. KFENCE objects must never be added to the allocator freelists).
- With CONFIG_KFENCE_NUM_OBJECTS (default 255), the number of available guarded objects can be controlled.

KFENCE API

- bool is_kfence_address(const void *addr)
- void kfence_shutdown_cache(struct kmem_cache *s)
- void *kfence_alloc(struct kmem_cache *s, size_t size, gfp_t flags)
- size_t kfence_ksize(const void *addr)
- void *kfence_object_start(const void *addr)
- void __kfence_free(void *addr) - release a KFENCE heap object to KFENCE pool
- bool kfence_free(void *addr) - try to release an arbitrary heap object to KFENCE pool
- bool kfence_handle_page_fault(unsigned long addr, bool is_write, struct pt_regs *regs)

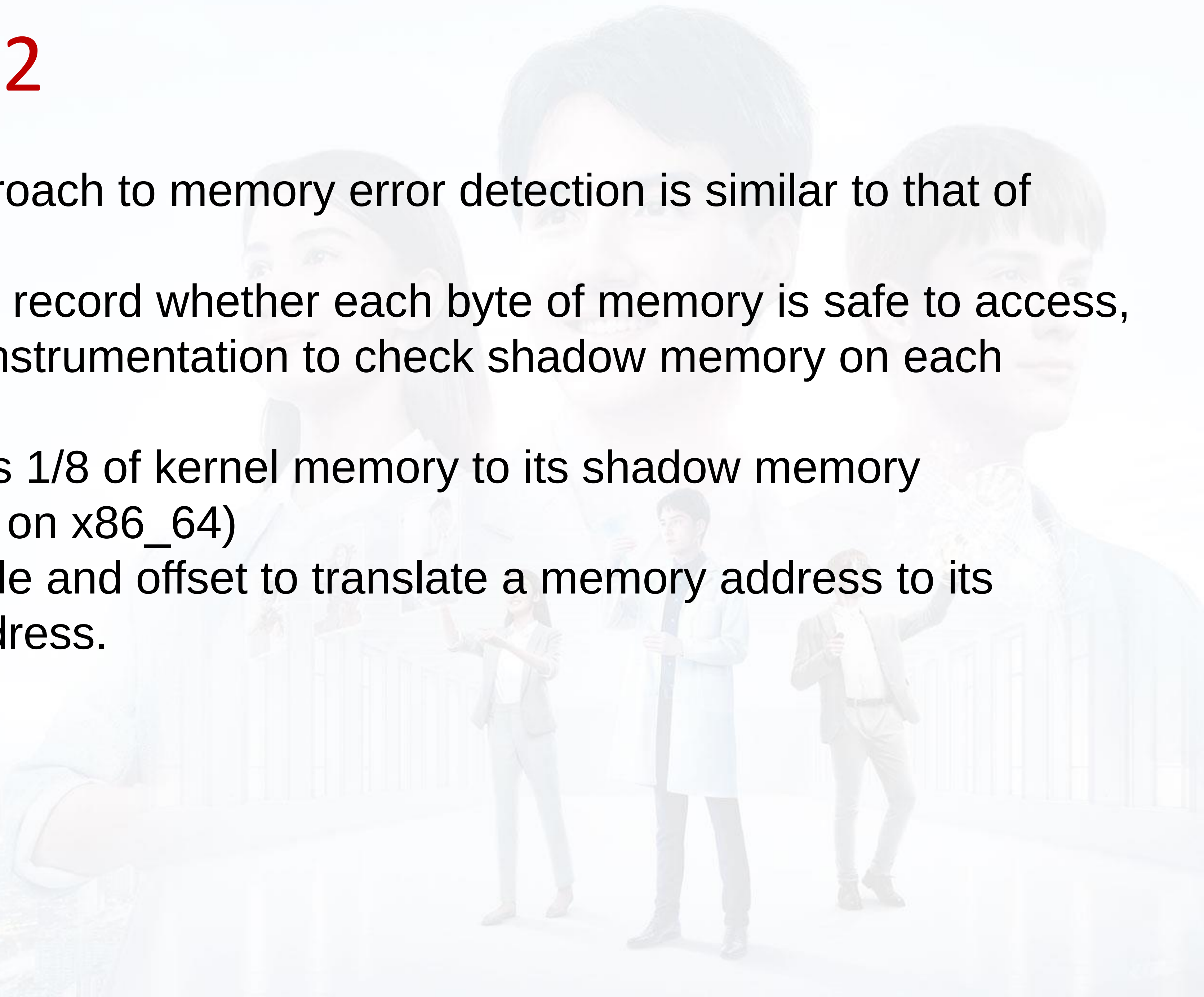




KASAN Intro 1 of 2

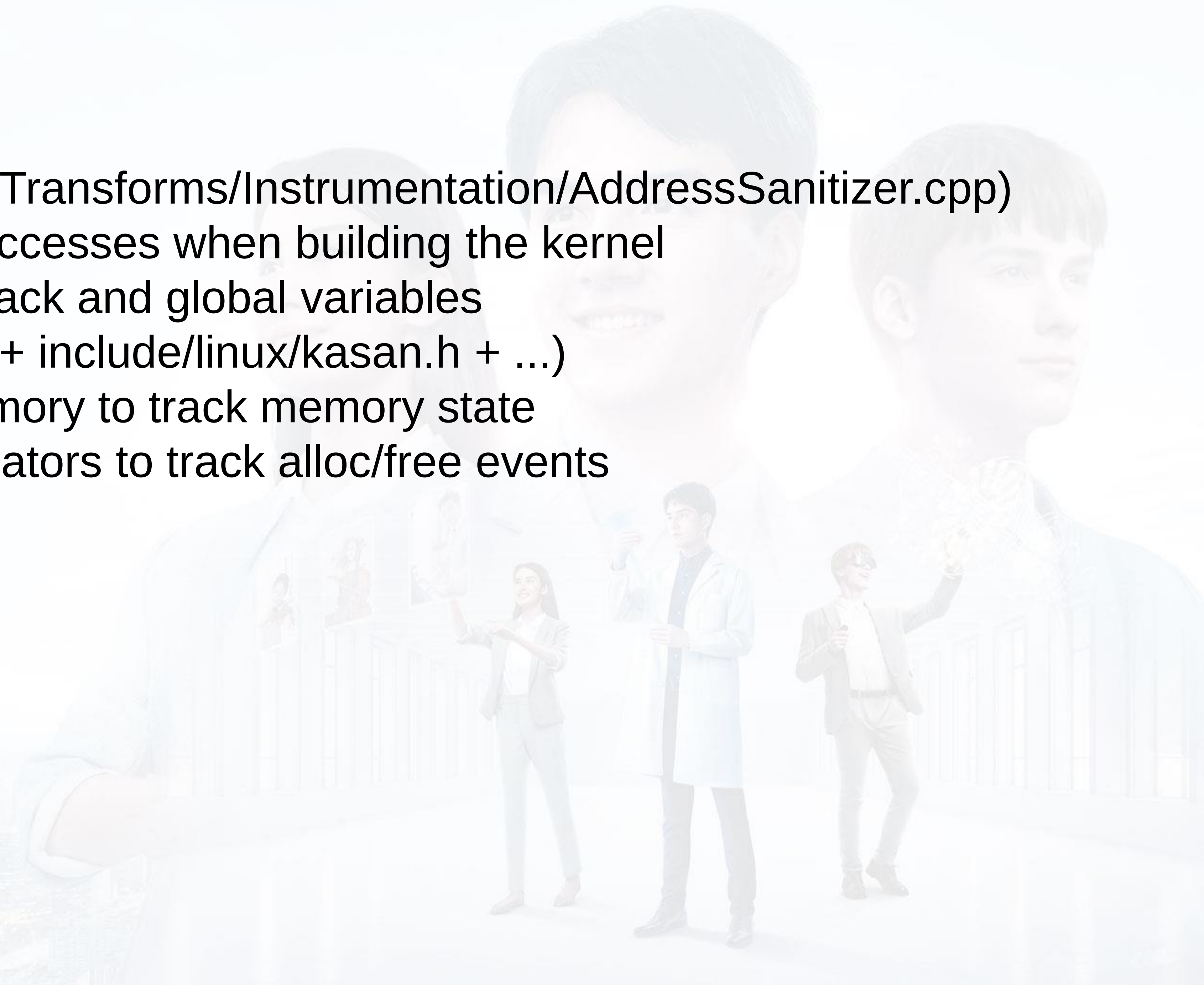
- KernelAddressSANitizer (KASAN) is a dynamic memory error detector (ASan ported to kernel)
- It provides a fast and comprehensive solution for finding use-after-free and out-of-bounds bugs.
- KASAN uses compile-time instrumentation for checking every memory access.
  - need a GCC version 4.9.2 or later.
  - GCC 5.0 or later is required for detection of out-of-bounds accesses to stack or global variables.
- To enable KASAN configure kernel with: CONFIG_KASAN = y
- Choose between CONFIG_KASAN_OUTLINE and CONFIG_KASAN_INLINE.
- Outline and inline are compiler instrumentation types. The former produces smaller binary the latter is 1.1 - 2 times faster.
- Inline instrumentation requires a GCC version 5.0 or later.
- KASAN works with both SLUB and SLAB memory allocators.
- For better bug detection and nicer reporting, enable CONFIG_STACKTRACE.

KASAN Intro 2 of 2



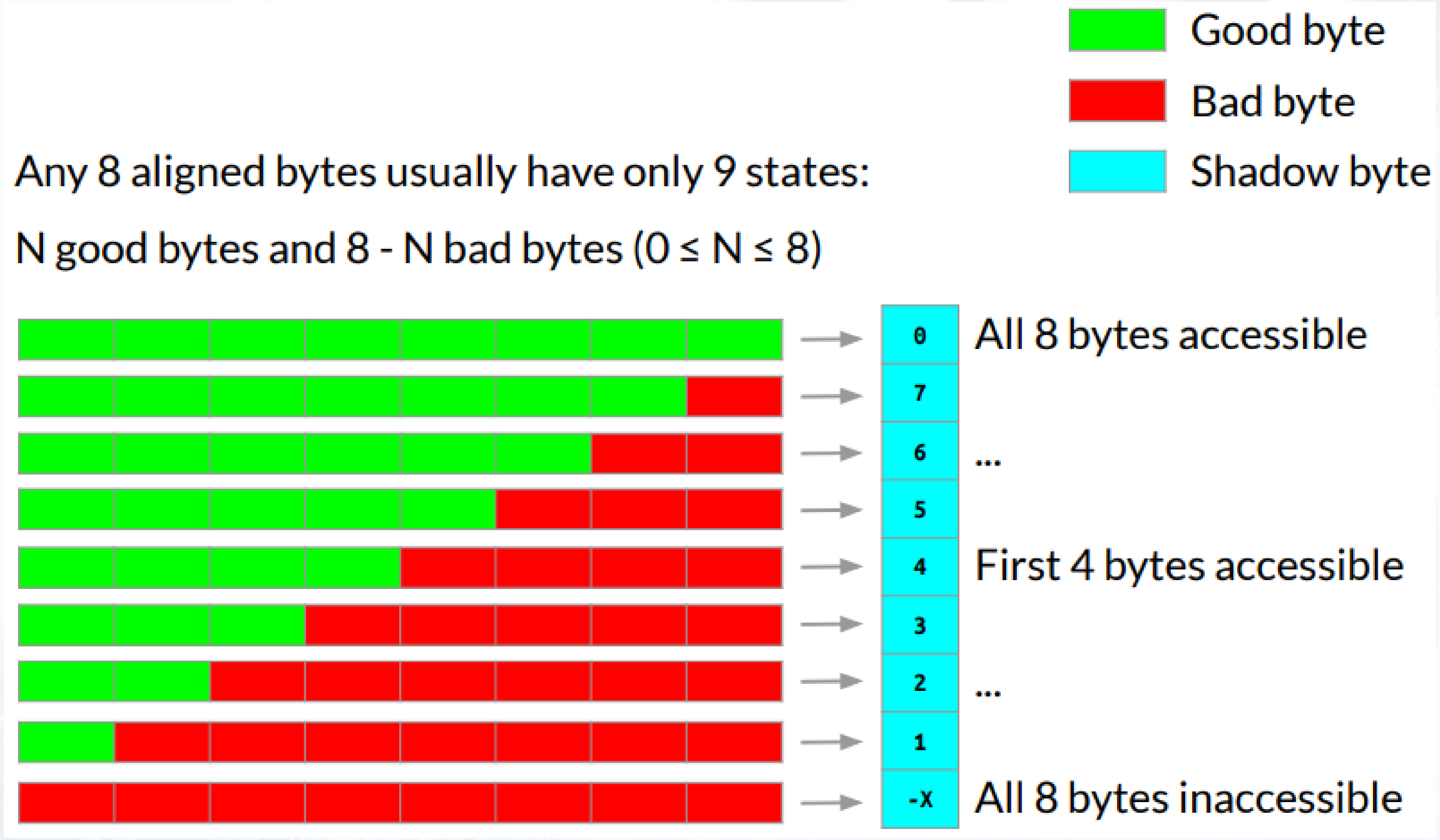
- From a high level, our approach to memory error detection is similar to that of kmemcheck:
  - use shadow memory to record whether each byte of memory is safe to access, and use compile-time instrumentation to check shadow memory on each memory access.
- AddressSanitizer dedicates 1/8 of kernel memory to its shadow memory
- (e.g. 16TB to cover 128TB on x86_64)
- It uses mapping with a scale and offset to translate a memory address to its corresponding shadow address.

KASAN parts

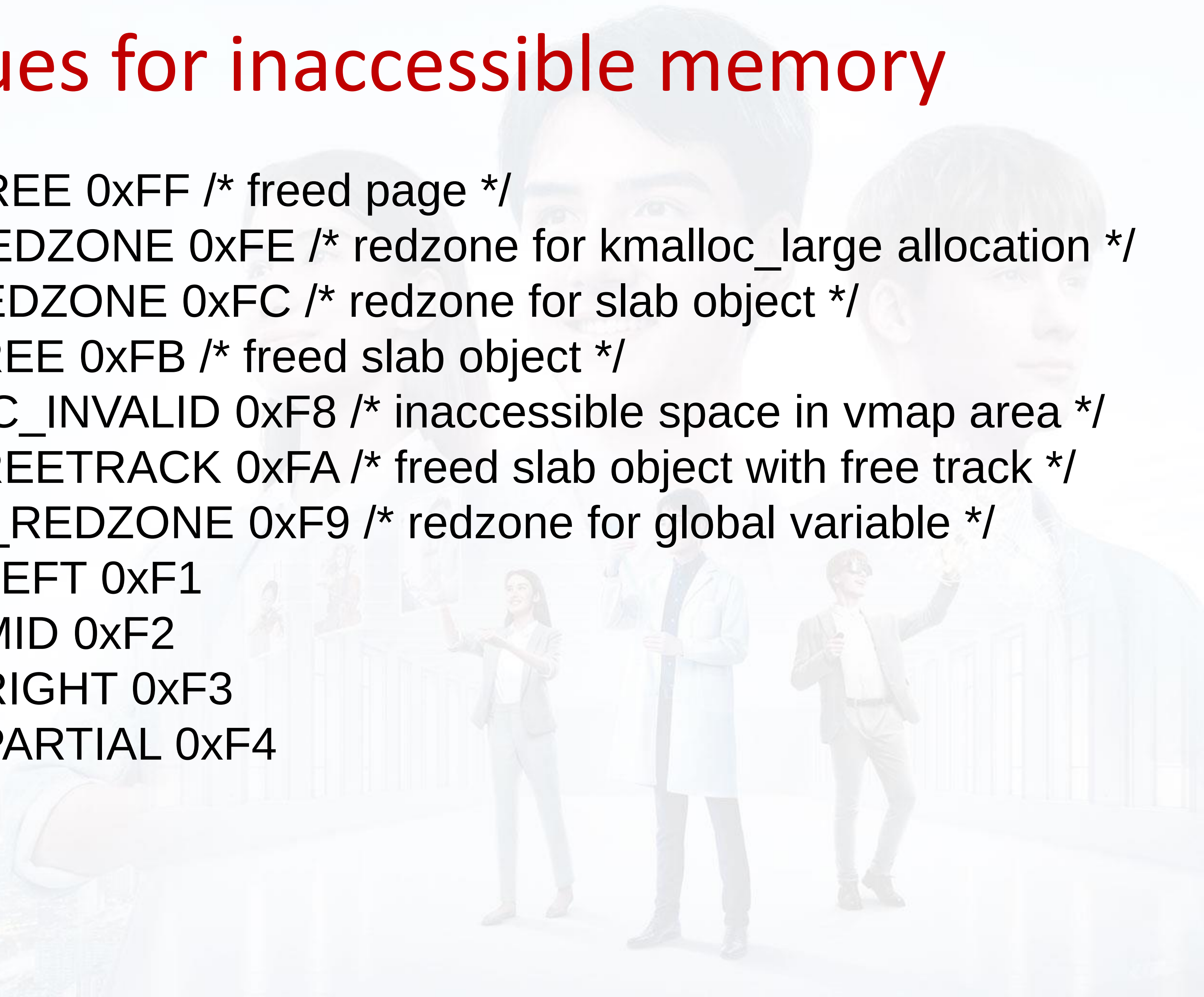


- Compiler module (llvm/lib/Transforms/Instrumentation/AddressSanitizer.cpp)
  - Instruments memory accesses when building the kernel
  - Inserts redzones for stack and global variables
- Runtime part (mm/kasan/ + include/linux/kasan.h + ...)
  - Maintains shadow memory to track memory state
  - Hooks into kernel allocators to track alloc/free events
  - Prints bug reports

KASAN shadow memory and shadow byte



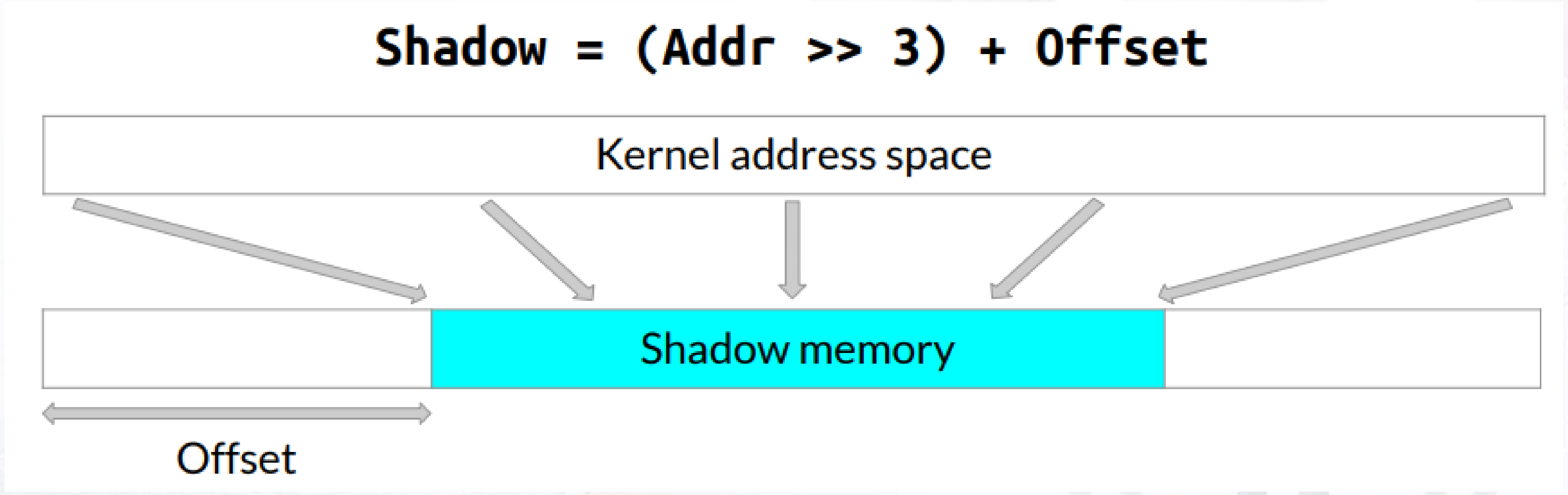
Shadow byte values for inaccessible memory



- #define KASAN_PAGE_FREE 0xFF /* freed page */
- #define KASAN_PAGE_REDZONE 0xFE /* redzone for kmalloc_large allocation */
- #define KASAN_SLAB_REDZONE 0xFC /* redzone for slab object */
- #define KASAN_SLAB_FREE 0xFB /* freed slab object */
- #define KASAN_VMALLOC_INVALID 0xF8 /* inaccessible space in vmap area */
- #define KASAN_SLAB_FREETRACK 0xFA /* freed slab object with free track */
- #define KASAN_GLOBAL_REDZONE 0xF9 /* redzone for global variable */
- #define KASAN_STACK_LEFT 0xF1
- #define KASAN_STACK_MID 0xF2
- #define KASAN_STACK_RIGHT 0xF3
- #define KASAN_STACK_PARTIAL 0xF4

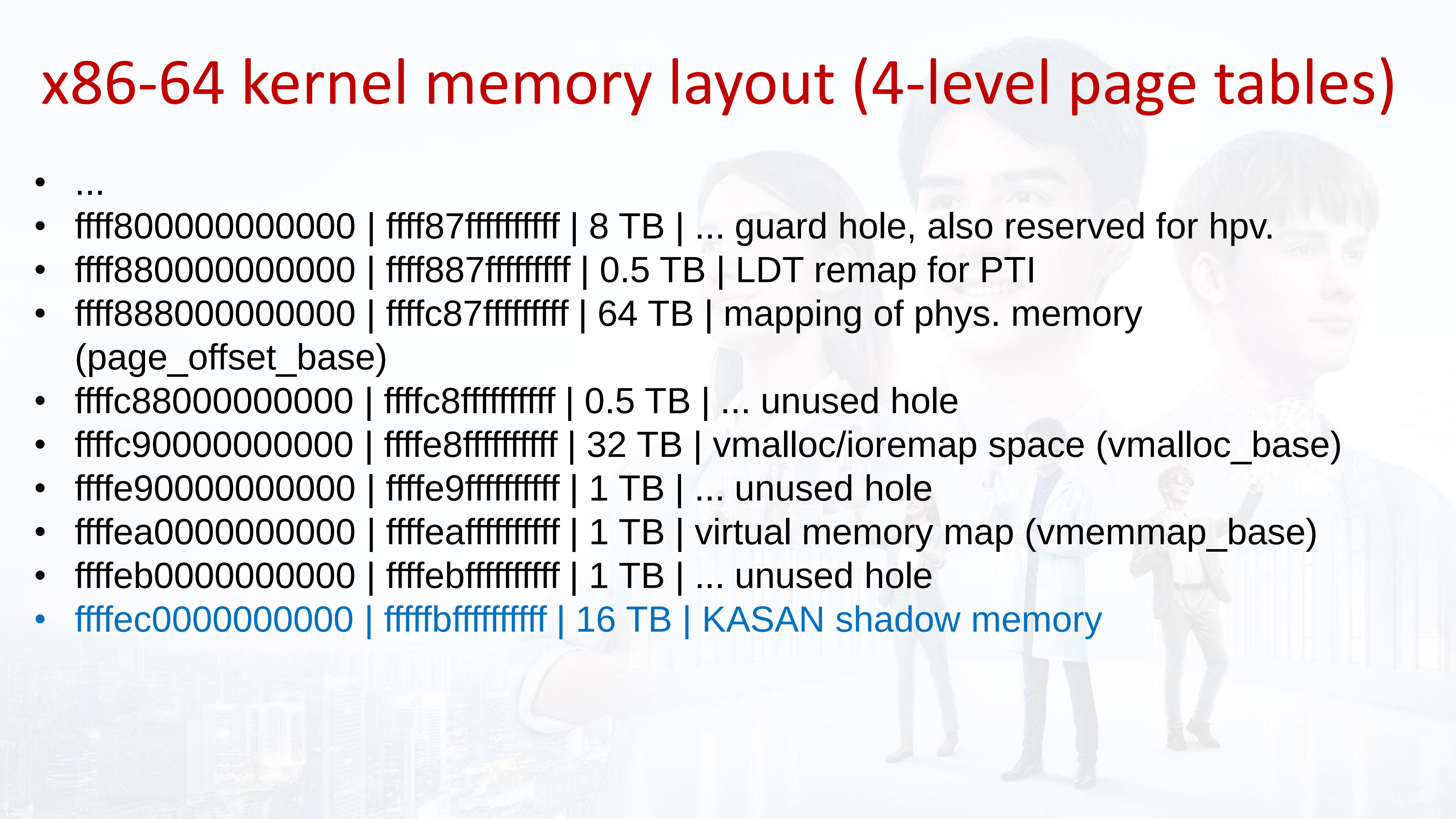
Shadow memory region

- Contains shadow bytes for each mapped region of kernel memory
- Memory-to-shadow mapping scheme:



x86-64 kernel memory layout (4-level page tables)

- ...
- ffff800000000000 | ffff87ffffffffff | 8 TB | ... guard hole, also reserved for hpv.
- ffff880000000000 | ffff887fffffffff | 0.5 TB | LDT remap for PTI
- ffff888000000000 | ffffc87fffffffff | 64 TB | mapping of phys. memory (page_offset_base)
- ffffc88000000000 | ffffc8ffffffffff | 0.5 TB | ... unused hole
- ffffc90000000000 | ffffe8ffffffffff | 32 TB | vmalloc/ioremap space (vmalloc_base)
- ffffe90000000000 | ffffe9ffffffffff | 1 TB | ... unused hole
- ffffea0000000000 | ffffeaffffffffff | 1 TB | virtual memory map (vmemmap_base)
- ffffeb0000000000 | ffffebffffffffff | 1 TB | ... unused hole
- ffffec0000000000 | fffffbffffffffff | 16 TB | KASAN shadow memory



Instrumentation of 8-byte access by compiler

```
*a = ...;
```

↓

```
char *shadow = (a >> 3) + Offset;
if (*shadow)
   kasan_report(a);
*a = ...;
```

Instrumentation of 1,2,4-byte access by compiler

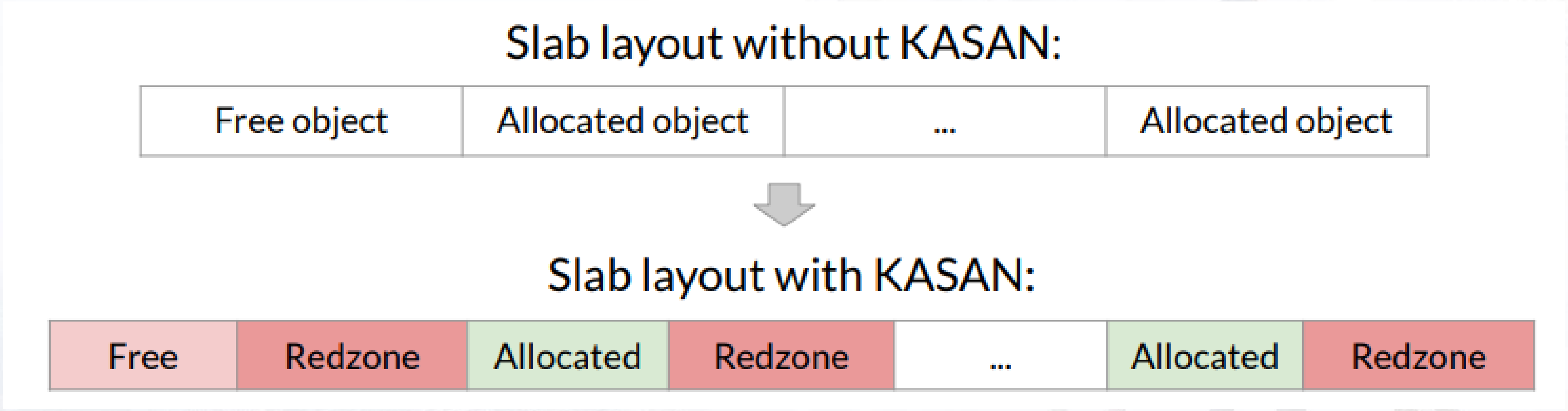
```
*a = ...;
```

$$\downarrow$$

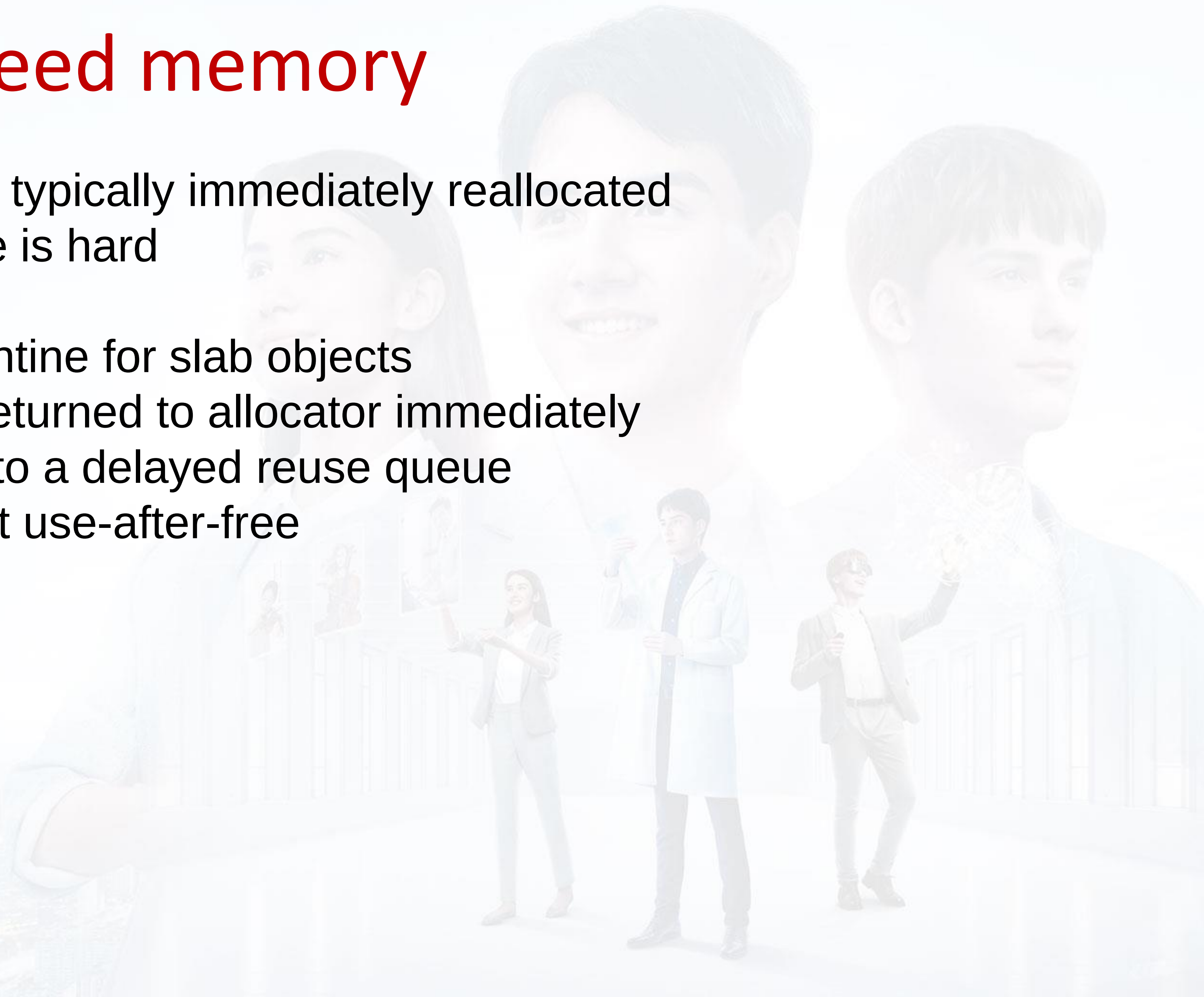
```
char *shadow = (a >> 3) + Offset;
if (*shadow && *shadow < (a & 7) + N)
    kasan_report(a);
*a = ...;
```

Allocation hooks

- KASAN need to keep shadow up-to-date
- This requires tracking of alloc/free events
- KASAN adds hooks to kernel allocators
  - SLUB/SLAB, page_alloc, vmalloc (grep code for "kasan_")



Quarantine for freed memory



- When memory is freed, it's typically immediately reallocated
  - Detecting use-after-free is hard

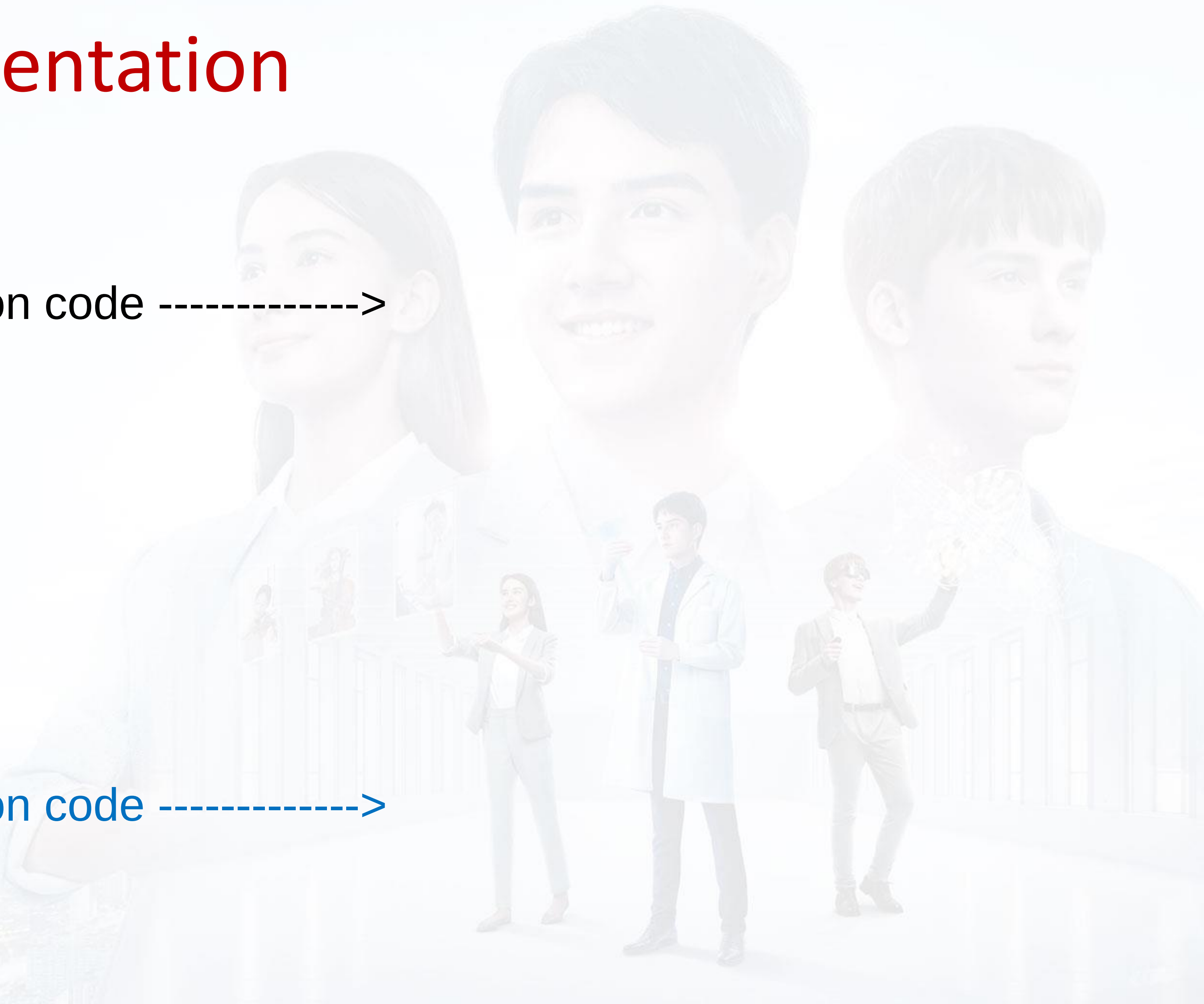
- KASAN implements quarantine for slab objects
  - Freed objects are not returned to allocator immediately
  - Instead, they are put into a delayed reuse queue
  - Higher chance to detect use-after-free

Compiler instrumentation

- void foo() {
- char x[10];
- <------------- Original function code ------------->
- }

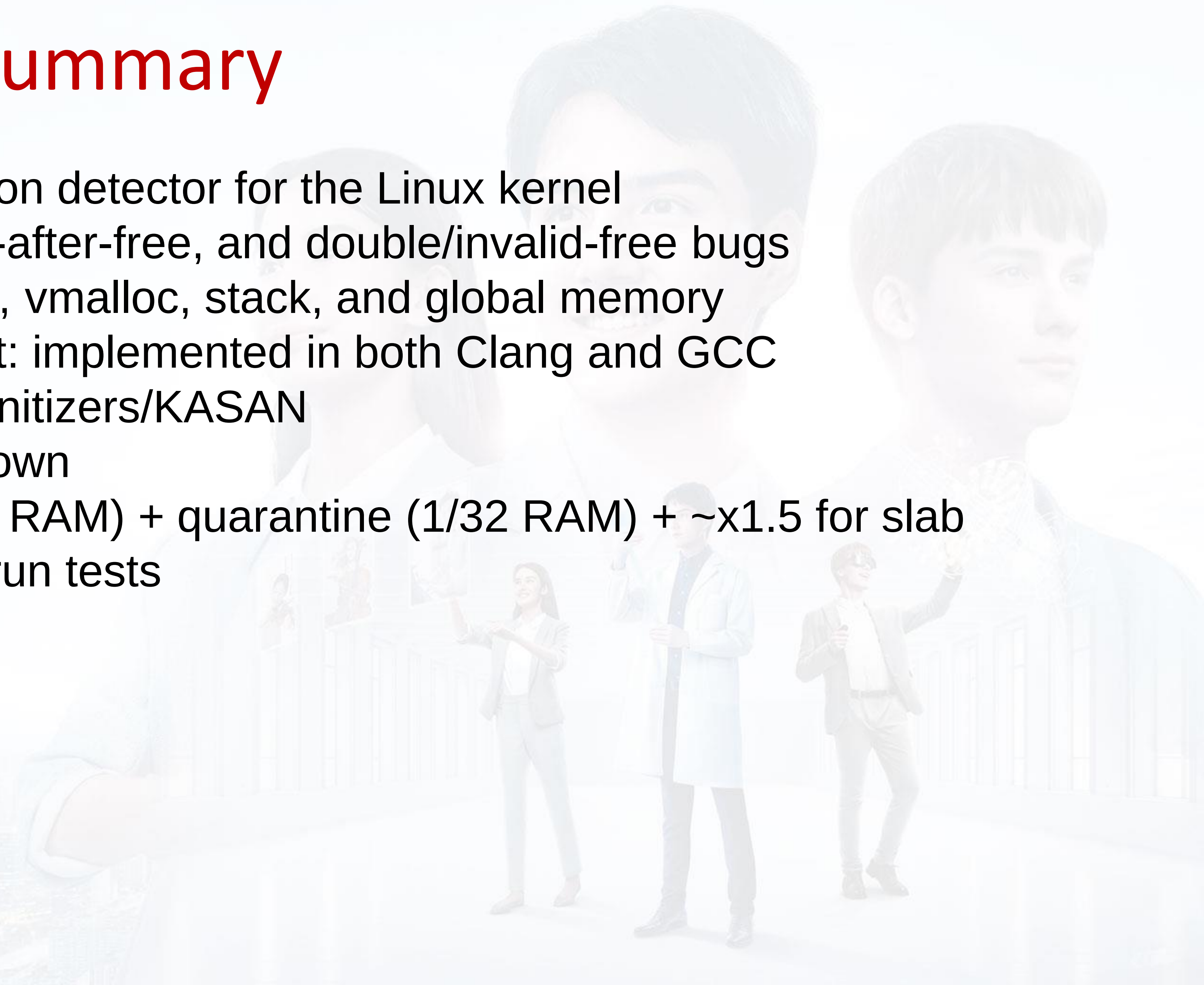
- To

- void foo() {
- char rz1[32];
- char x[10];
- char rz2[22];
- <------------- Original function code ------------->
- }



Generic KASAN summary

- Dynamic memory corruption detector for the Linux kernel
- Finds out-of-bounds, use-after-free, and double/invalid-free bugs
- Supports slab, page_alloc, vmalloc, stack, and global memory
- Requires compiler support: implemented in both Clang and GCC
- google.github.io/kernel-sanitizers/KASAN
- Relatively fast: ~x2 slowdown
- RAM impact: shadow (1/8 RAM) + quarantine (1/32 RAM) + ~x1.5 for slab
- Basic usage: enable and run tests



More sanitizers

- KMSAN — Kernel Memory Sanitizer (un-init mem)
- KCSAN — Kernel Concurrency Sanitizer
- UBSAN — [Kernel] Undefined Behavior Sanitizer