ITMO Advanced OS 2024

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

Lesson 2024.09.25

Contents

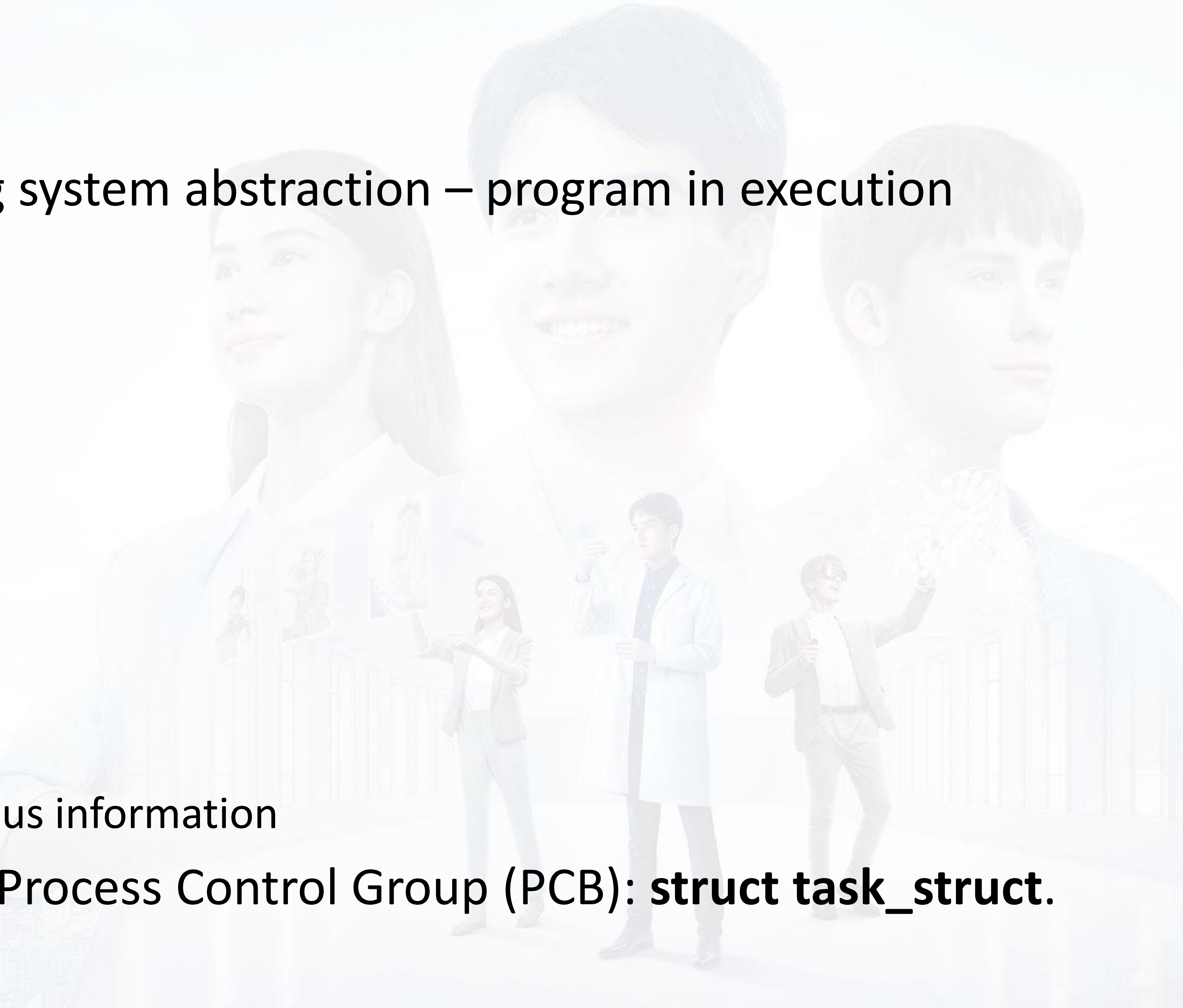
- Processes and threads
- Interrupts and exceptions
- Symmetric Multi-Processing

Processes and threads topics

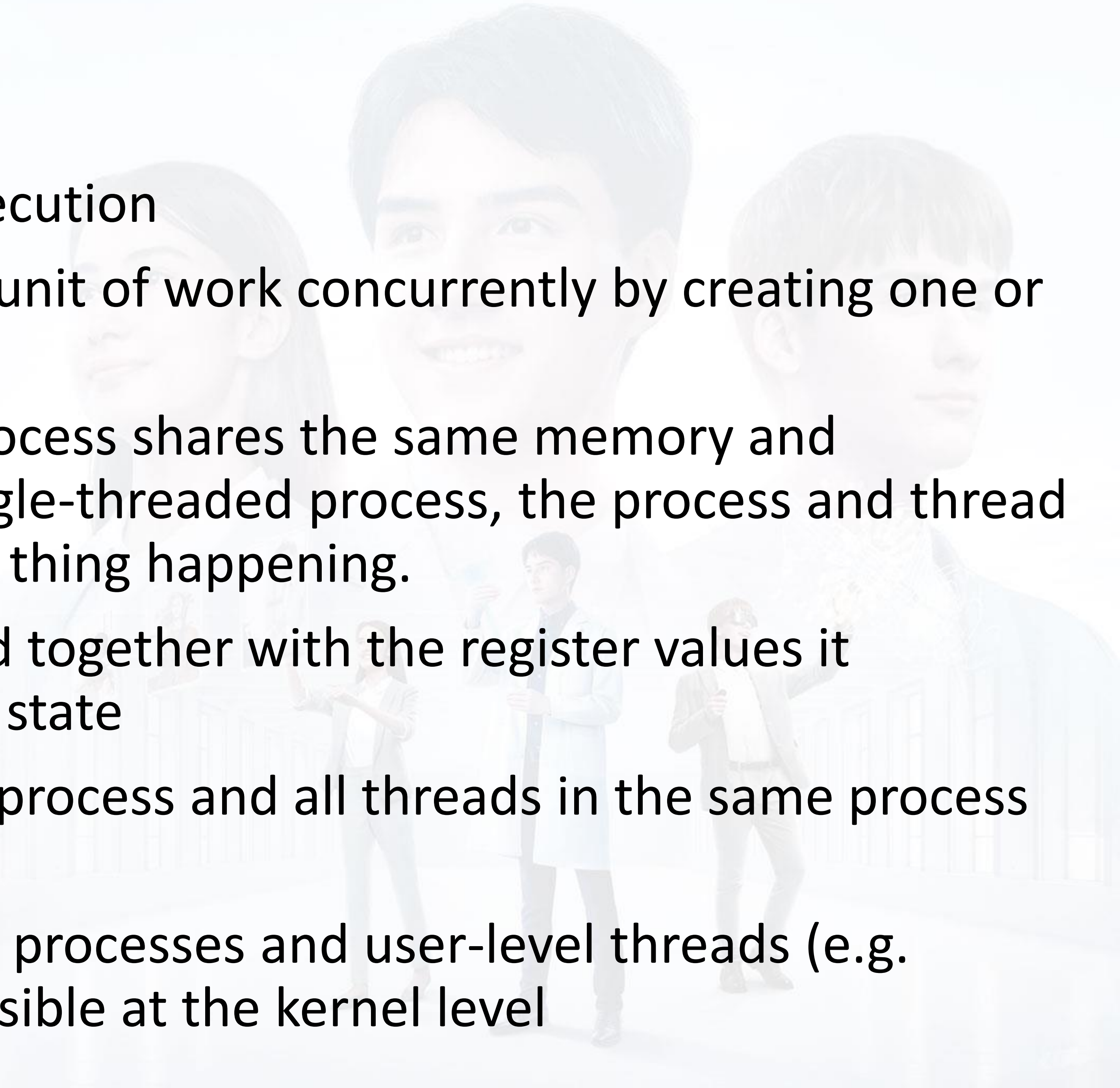
- Processes
- Threads
- Context switching
- Blocking and waking up
- Process context

Processes

- A process is an operating system abstraction – program in execution
- Process is set of:
  - An address space
  - One or more threads
  - Opened files
  - Sockets
  - Semaphores
  - Shared memory regions
  - Timers
  - Signal handlers
  - Other resources and status information
- They are grouped in the Process Control Group (PCB): **struct task_struct**.

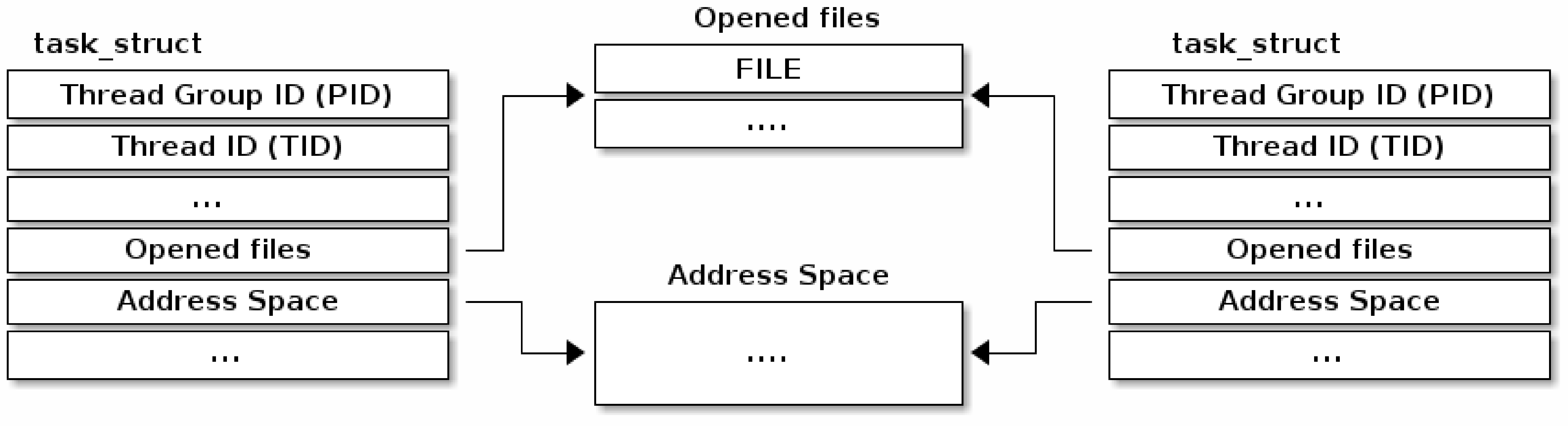


Threads



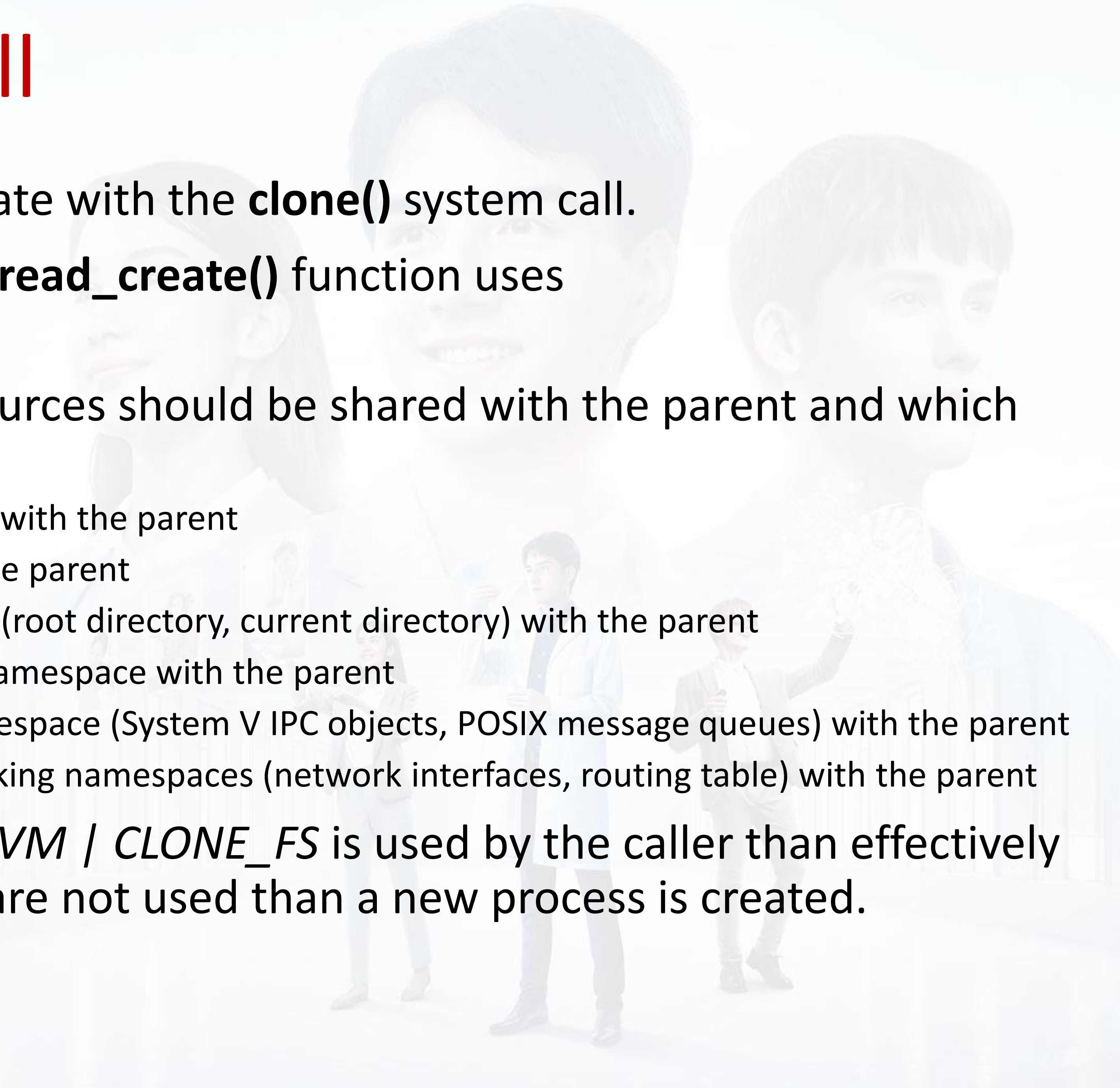
- Thread is (part of) program in execution
- A process can do more than one unit of work concurrently by creating one or more threads
- Any thread created within the process shares the same memory and resources of the process. In a single-threaded process, the process and thread are the same, as there’s only one thing happening.
- Each thread has its own stack and together with the register values it determines the thread execution state
- A thread runs in the context of a process and all threads in the same process share the resources
- The kernel schedules threads not processes and user-level threads (e.g. fibers, coroutines, etc.) are not visible at the kernel level

Linux implementation of threads



The clone() system call

- In Linux a new thread or process is create with the **clone()** system call.
- Both the **fork()** system call and the **pthread_create()** function uses the **clone()** implementation.
- It allows the caller to decide what resources should be shared with the parent and which should be copied or isolated:
  - CLONE_FILES - shares the file descriptor table with the parent
  - CLONE_VM - shares the address space with the parent
  - CLONE_FS - shares the filesystem information (root directory, current directory) with the parent
  - CLONE_NEWNS - does not share the mount namespace with the parent
  - CLONE_NEWIPC - does not share the IPC namespace (System V IPC objects, POSIX message queues) with the parent
  - CLONE_NEWNET - does not share the networking namespaces (network interfaces, routing table) with the parent
- For example, if *CLONE_FILES | CLONE_VM | CLONE_FS* is used by the caller than effectively a new thread is created. If these flags are not used than a new process is created.



Namespaces and containers 1 of 3

- Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources.
- The feature works by having the same namespace for a set of resources and processes, but those namespaces refer to distinct resources.
- Resources may exist in multiple spaces. Examples of such resources are process IDs, hostnames, user IDs, file names, and some names associated with network access, and interprocess communication
- Namespaces are a fundamental aspect of containers on Linux.
- The term "namespace" is often used for a type of namespace (e.g. process ID) as well as for a particular space of names.
- A Linux system starts out with a single namespace of each type, used by all processes. Processes can create additional namespaces and join different namespaces.

Namespaces and containers 2 of 3

- Linux namespaces were inspired by the wider namespace functionality used heavily throughout Plan 9 from Bell Labs

- The Linux Namespaces originated in 2002 in the 2.4.19 kernel with work on the mount namespace kind. Additional namespaces were added beginning in 2006 and continuing into the future.

- Adequate containers support functionality was finished in kernel version 3.8 with the introduction of User namespaces

Namespaces and containers 3 of 3

- Containers are a form of lightweight virtual machines that share the same kernel instance, as opposed to normal virtualization where a hypervisor runs multiple VMs, each with its one kernel instance.

- Examples of container technologies are LXC - that allows running lightweight "VM" and docker - a specialized container for running a single application.

- Containers are built on top of a few kernel features, one of which is namespaces. They allow isolation of different resources that would otherwise be globally visible.

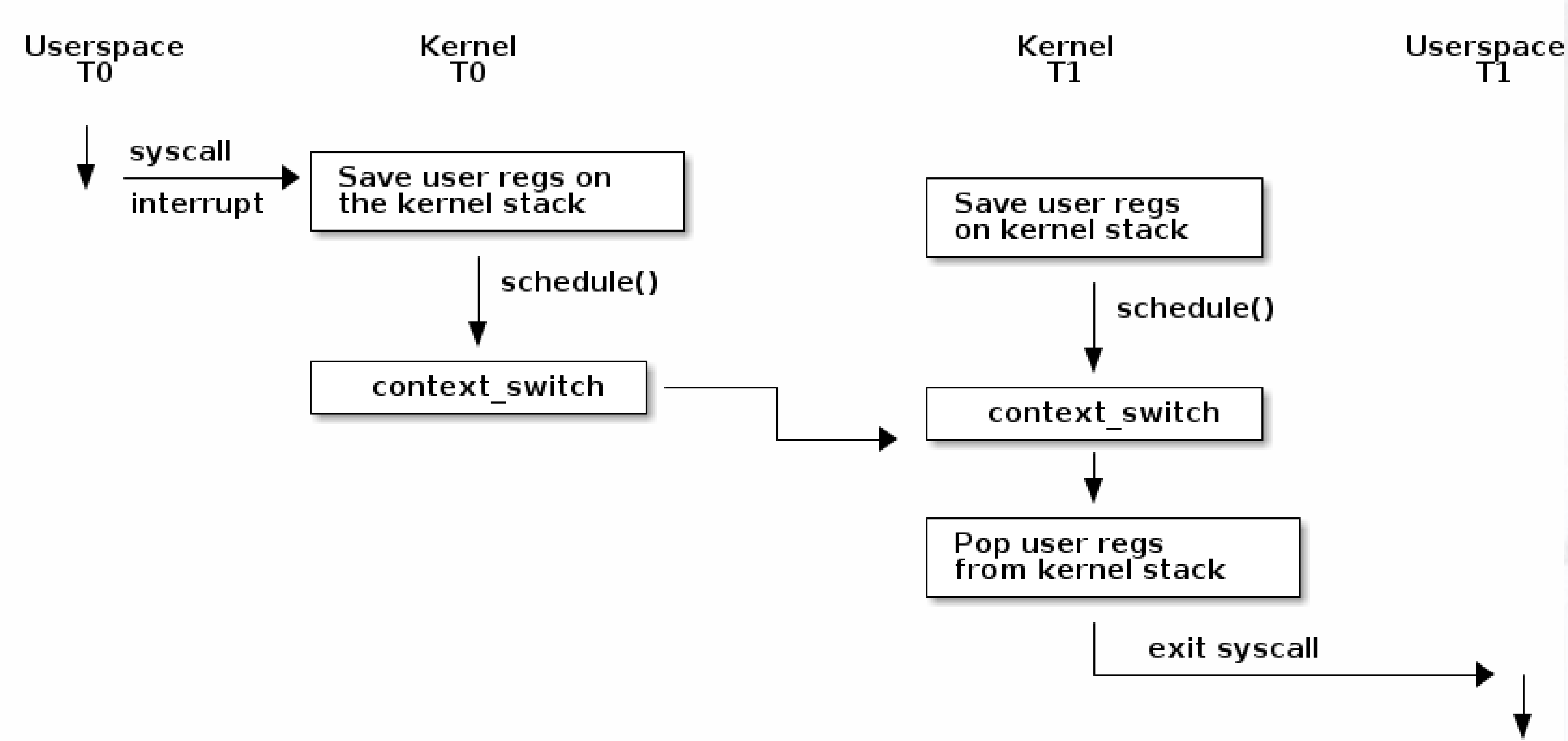
- To achieve this partitioning, the struct nsproxy structure is used to group types of resources that we want to partition. It currently supports IPC, networking, cgroup, mount, networking, PID, time namespaces.

- When a new namespace is created a new net namespace is created and then new processes can point to that new namespace instead of the default one.

Accessing the current process

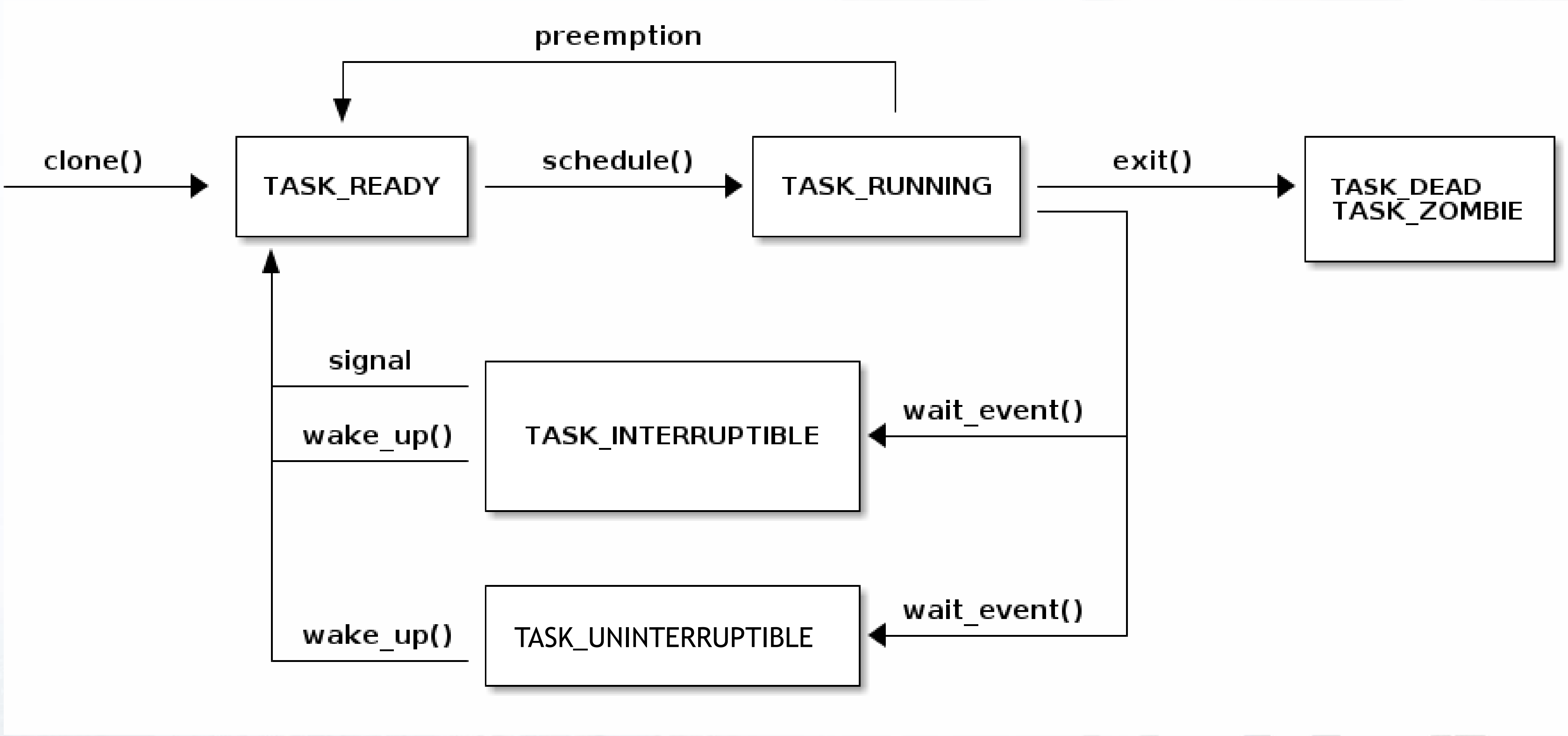
- Accessing the current process is a frequent operation
- opening a file needs access to **struct task_struct**'s file field
- mapping a new file needs access to **struct task_struct**'s mm field
- Over 90% of the system calls needs to access the current process structure so it needs to be fast
- The **current** macro is available to access to current process **struct task_struct**
- In order to support fast access in multi processor configurations a per CPU variable is used to store and retrieve the pointer to the current **struct task_struct**

Context switching (from T0 to T1)

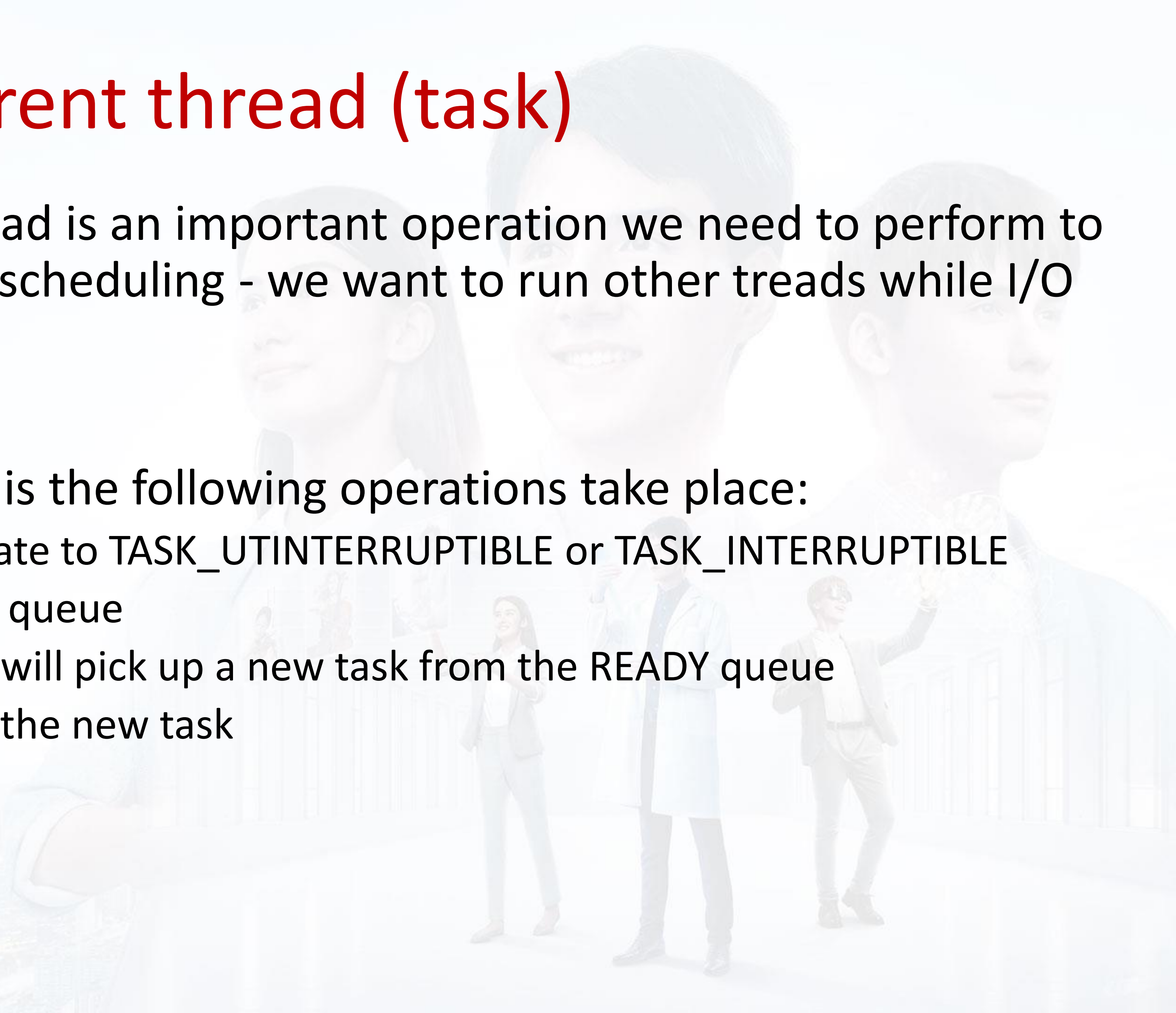


- Before a context switch can occur we must do a kernel transition, either with a system call or with an interrupt. At that point the user space registers are saved on the kernel stack.
- At some point the schedule() function will be called which can decide that a context switch must occur from T0 to T1 (e.g. because the current thread is blocking waiting for an I/O operation to complete or because it's allocated time slice has expired).

Blocking and waking up tasks



Blocking the current thread (task)

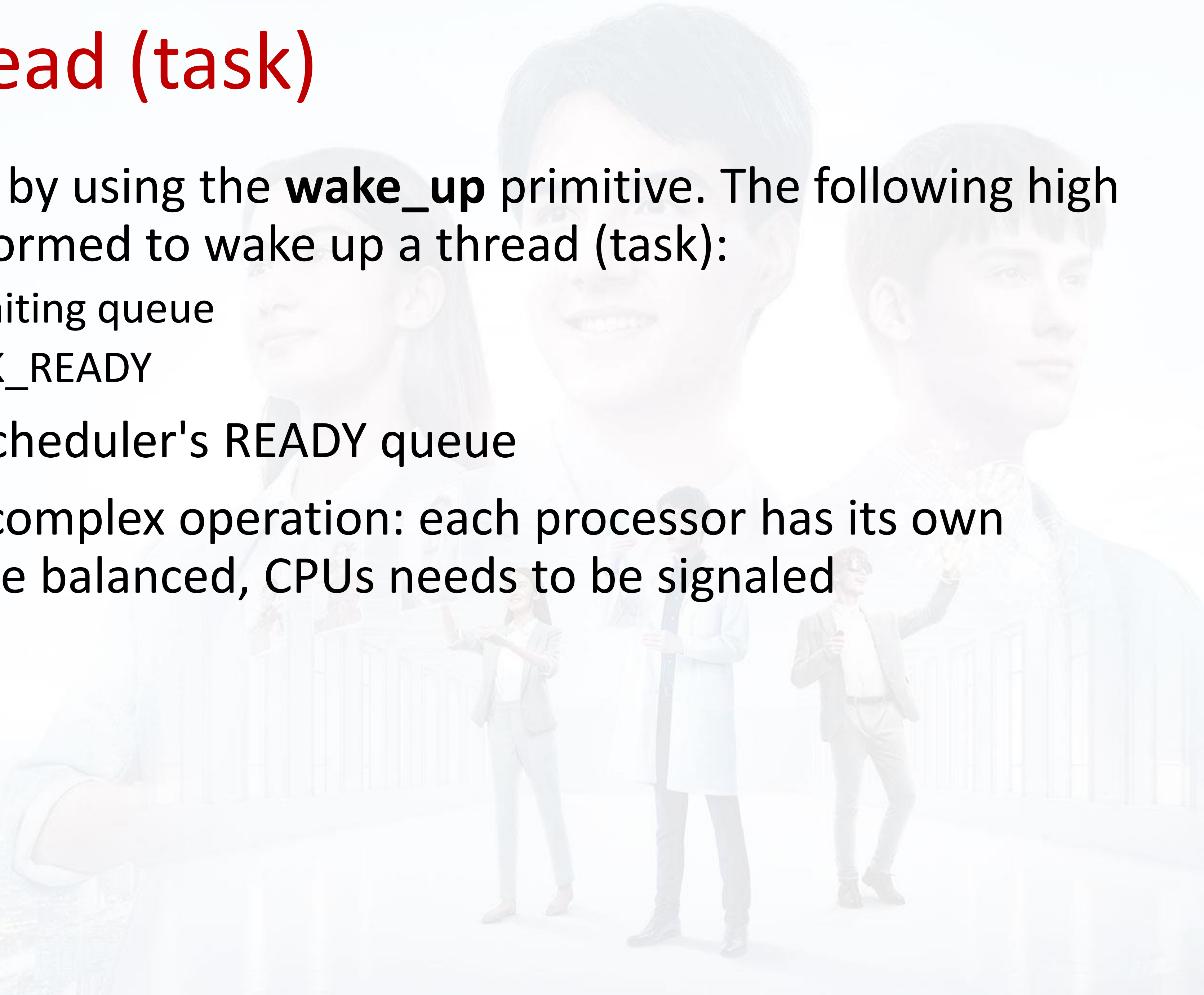


- Blocking the current thread is an important operation we need to perform to implement efficient task scheduling - we want to run other treads while I/O operations complete

- In order to accomplish this the following operations take place:
  - Set the current thread state to TASK_UTINTERRUPTIBLE or TASK_INTERRUPTIBLE
  - Add the task to a waiting queue
  - Call the scheduler which will pick up a new task from the READY queue
  - Do the context switch to the new task

Waking up a thread (task)

- We can wake-up threads by using the **wake_up** primitive. The following high level operations are performed to wake up a thread (task):
  - Select a task from the waiting queue
  - Set the task state to TASK_READY
- Insert the task into the scheduler's READY queue
- On SMP system this is a complex operation: each processor has its own queue, queues need to be balanced, CPUs needs to be signaled



Preempting tasks

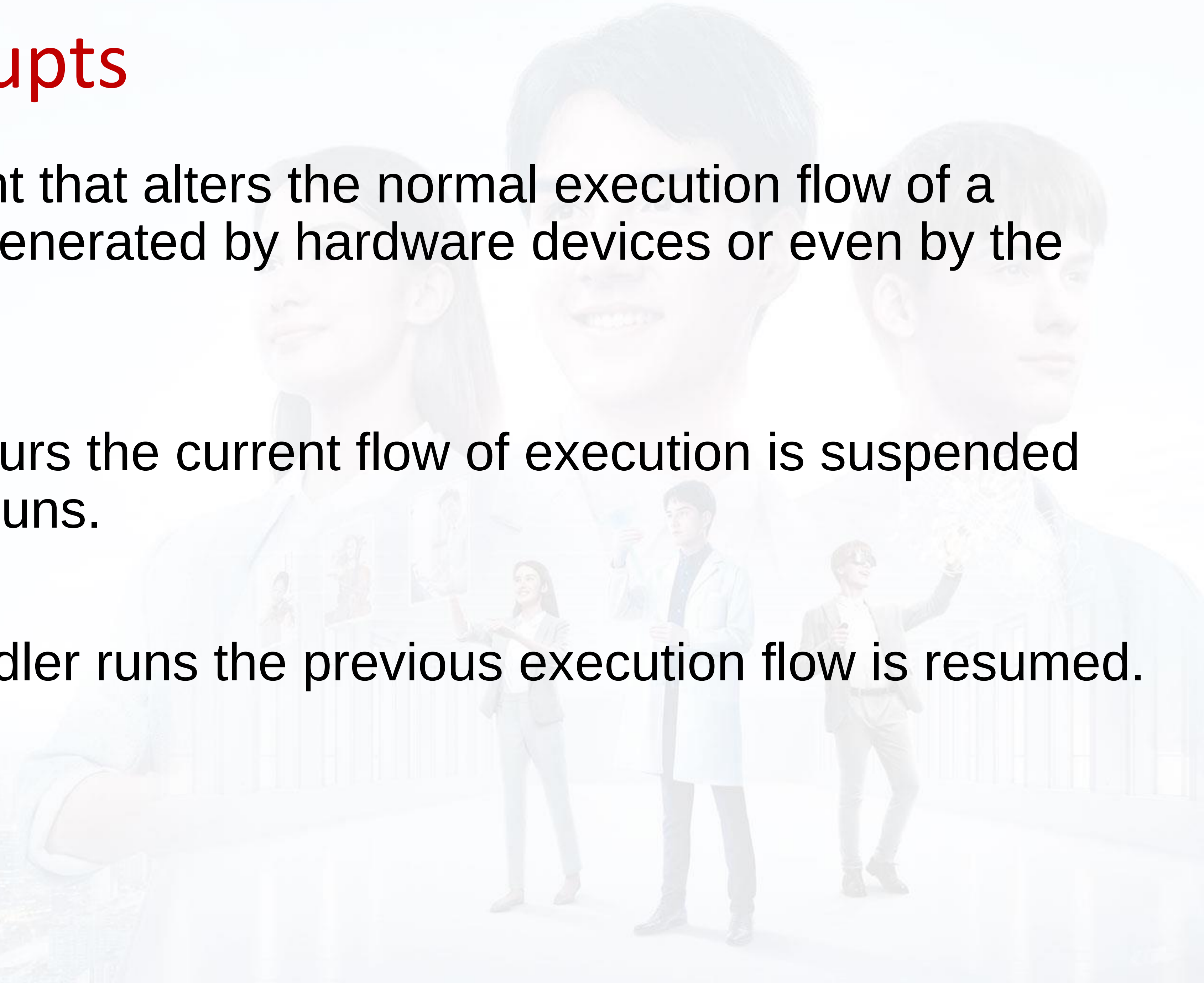
- We saw how context switches occurs voluntary between threads. Now - how preemption is handled?
- **Non preemptive kernel**
  - At every tick the kernel checks to see if the current process has its time slice consumed
  - If that happens a flag is set in interrupt context
  - Before returning to userspace the kernel checks this flag and calls **schedule()** if needed
  - In this case tasks are not preempted while running in kernel mode (e.g. system call) so there are no synchronization issues
- **Preemptive kernel**
  - In this case the current task can be preempted even if we are running in kernel mode and executing a system call. This requires using a special synchronization primitives: preempt_disable and preempt_enable.
  - In order to simplify handling for preemptive kernels and since synchronization primitives are needed for the SMP case anyway, preemption is disabled automatically when a spinlock is used.
  - If we run into a condition that requires the preemption of the current task (its time slices has expired) a flag is set. This flag is checked whenever the preemption is reactivated, e.g. when exiting a critical section through a spin_unlock() and if needed the scheduler is called to select a new task.

Process context

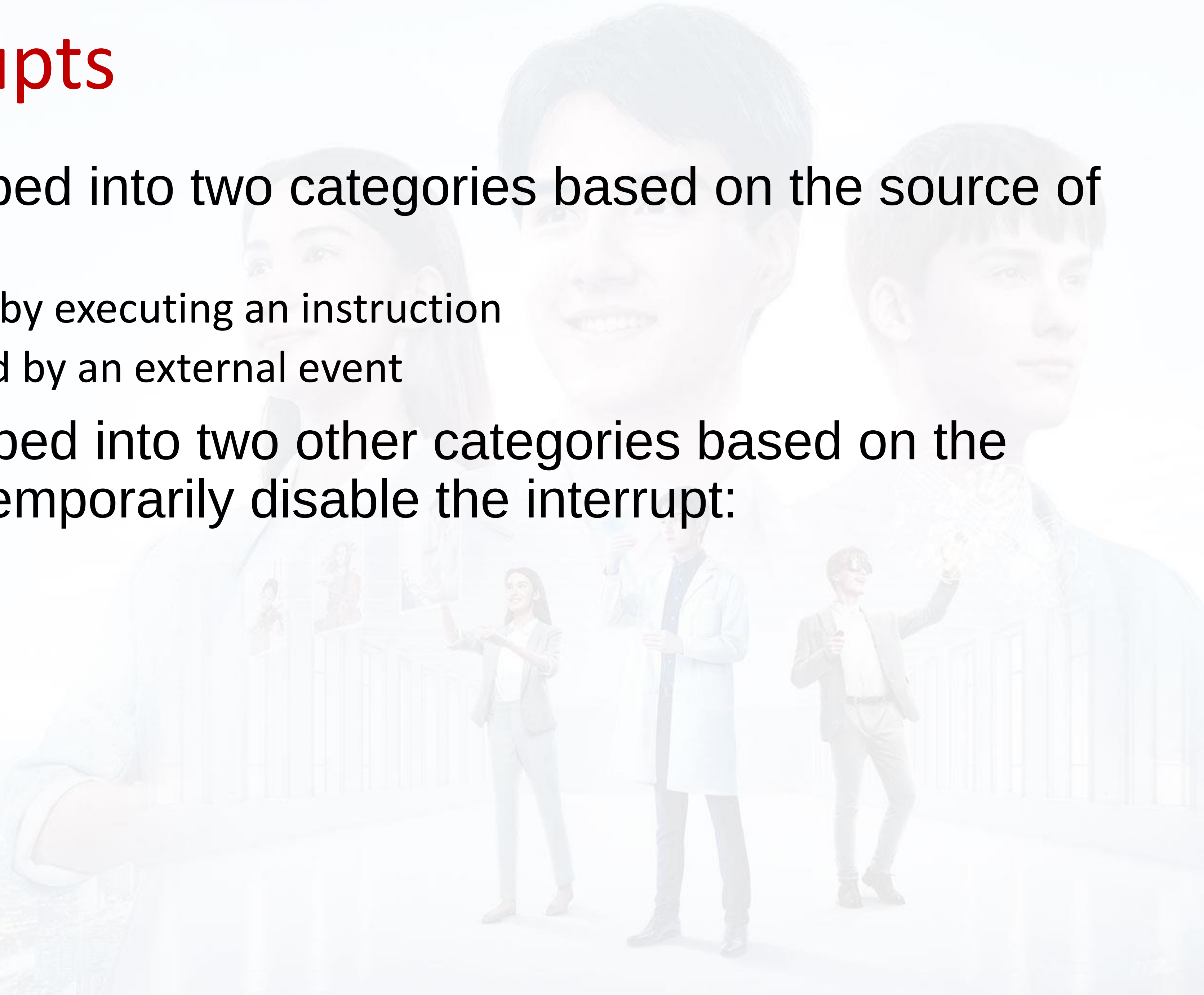
- The context of a process includes its address space, stack space, virtual address space, register set image (e.g. Program Counter (PC), Stack Pointer (SP), Instruction Register (IR), Program Status Word (PSW) and other general processor registers), etc.
- The kernel is executing in process context when it is running a system call.
- In process context we can access the current process data with **current**
- In process context we can sleep (wait on a condition).
- In process context we can access the user-space (unless we are running in a kernel thread context).

What are interrupts

- An interrupt is an event that alters the normal execution flow of a program and can be generated by hardware devices or even by the CPU itself.
- When an interrupt occurs the current flow of execution is suspended and interrupt handler runs.
- After the interrupt handler runs the previous execution flow is resumed.



What are interrupts



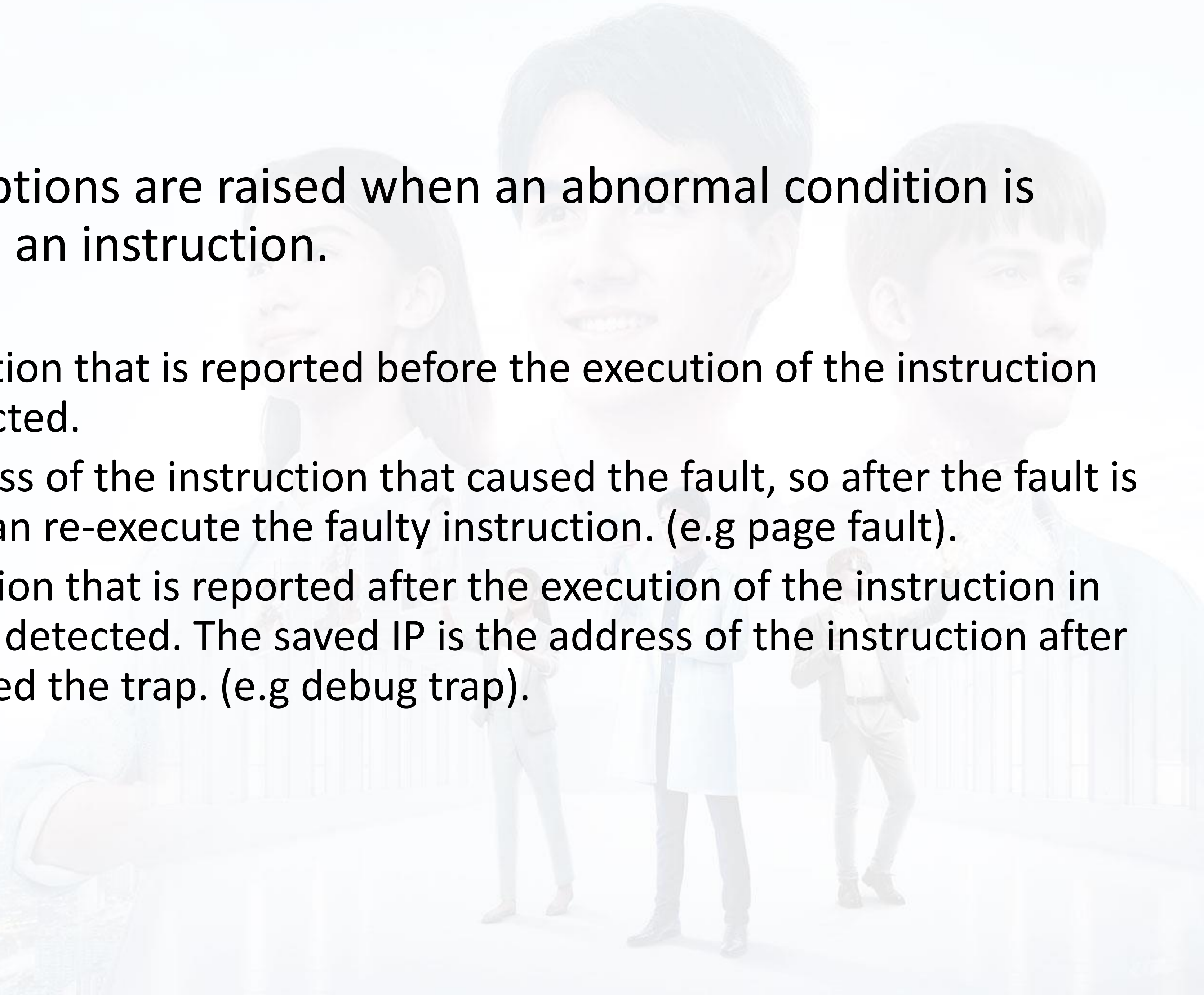
- Interrupts can be grouped into two categories based on the source of the interrupt.
  - **synchronous**, generated by executing an instruction
  - **asynchronous**, generated by an external event
- They can also be grouped into two other categories based on the ability to postpone or temporarily disable the interrupt:
  - **maskable**
    - can be ignored
    - signaled via INT pin
  - **non-maskable**
    - cannot be ignored
    - signaled via NMI pin

What are interrupts

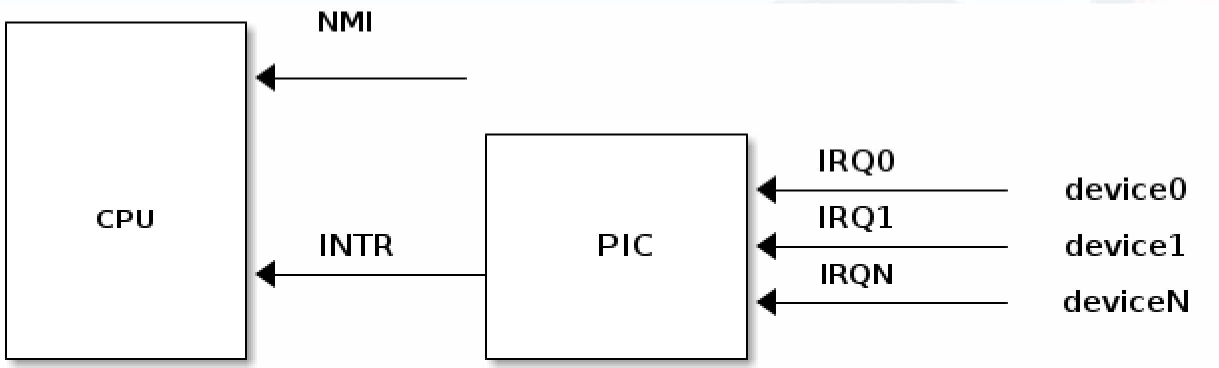
- Synchronous interrupts, usually named exceptions, handle conditions detected by the processor itself in the course of executing an instruction.
    - Divide by zero or a system call are examples of exceptions
- Asynchronous interrupts, usually named interrupts, are external events generated by I/O devices.
    - Network card generates an interrupts to signal that a packet has arrived
- Most interrupts are maskable, which means we can temporarily postpone running the interrupt handler when we disable the interrupt until the time the interrupt is re-enabled.
- However, there are a few critical interrupts that can not be disabled/postponed.

Exceptions

- Processor detected exceptions are raised when an abnormal condition is detected while executing an instruction.
  - **Faults, traps, aborts**
  - A fault is a type of exception that is reported before the execution of the instruction and can be usually corrected.
  - The saved IP is the address of the instruction that caused the fault, so after the fault is corrected the program can re-execute the faulty instruction. (e.g page fault).
  - A trap is a type of exception that is reported after the execution of the instruction in which the exception was detected. The saved IP is the address of the instruction after the instruction that caused the trap. (e.g debug trap).
- Programmed exceptions
  - **int n**



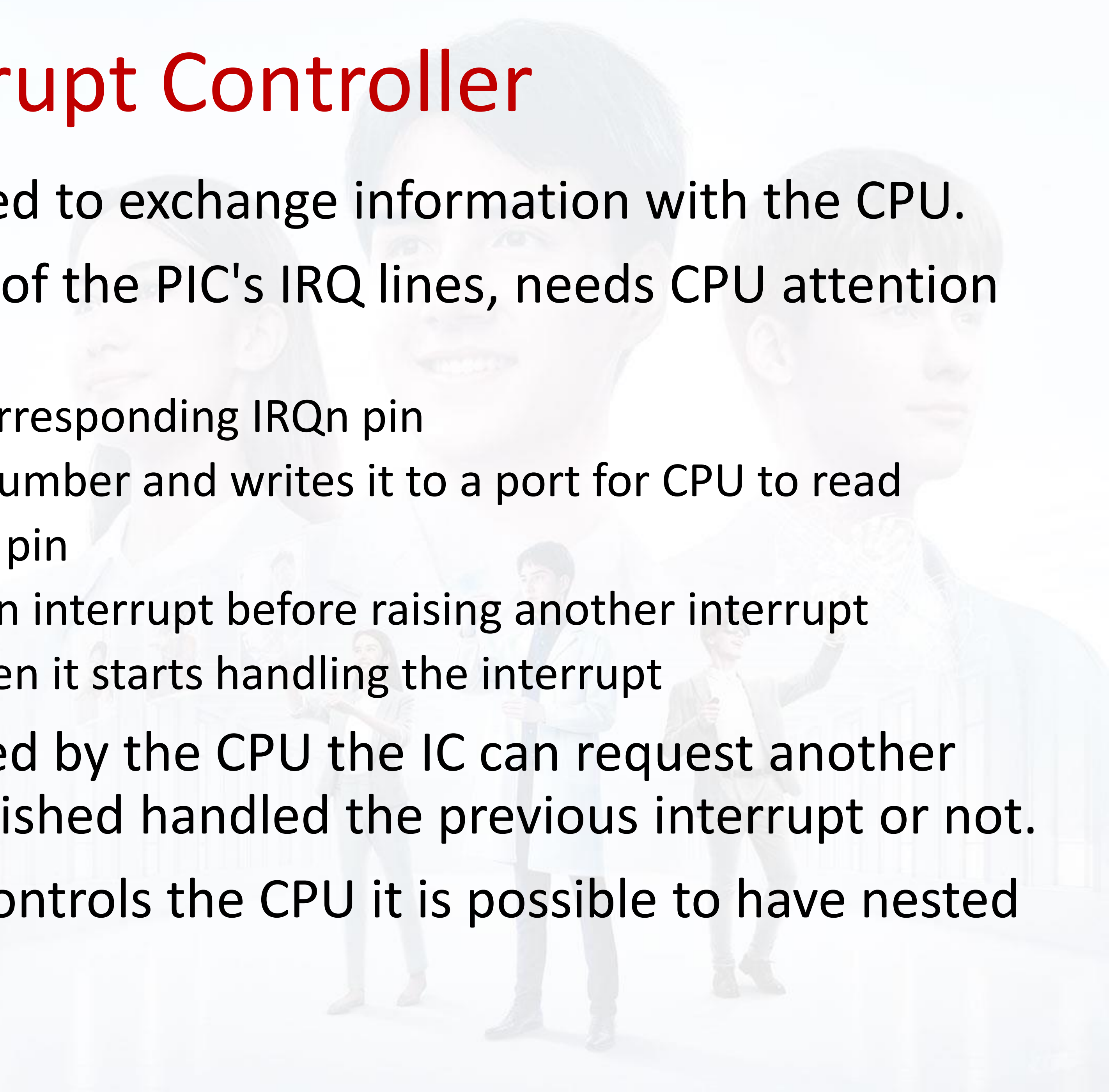
Hardware concepts



- A device supporting interrupts has an output pin used for signaling an Interrupt ReQuest.
- IRQ pins are connected to a device named Programmable Interrupt Controller (PIC) which is connected to CPU's INTR pin.

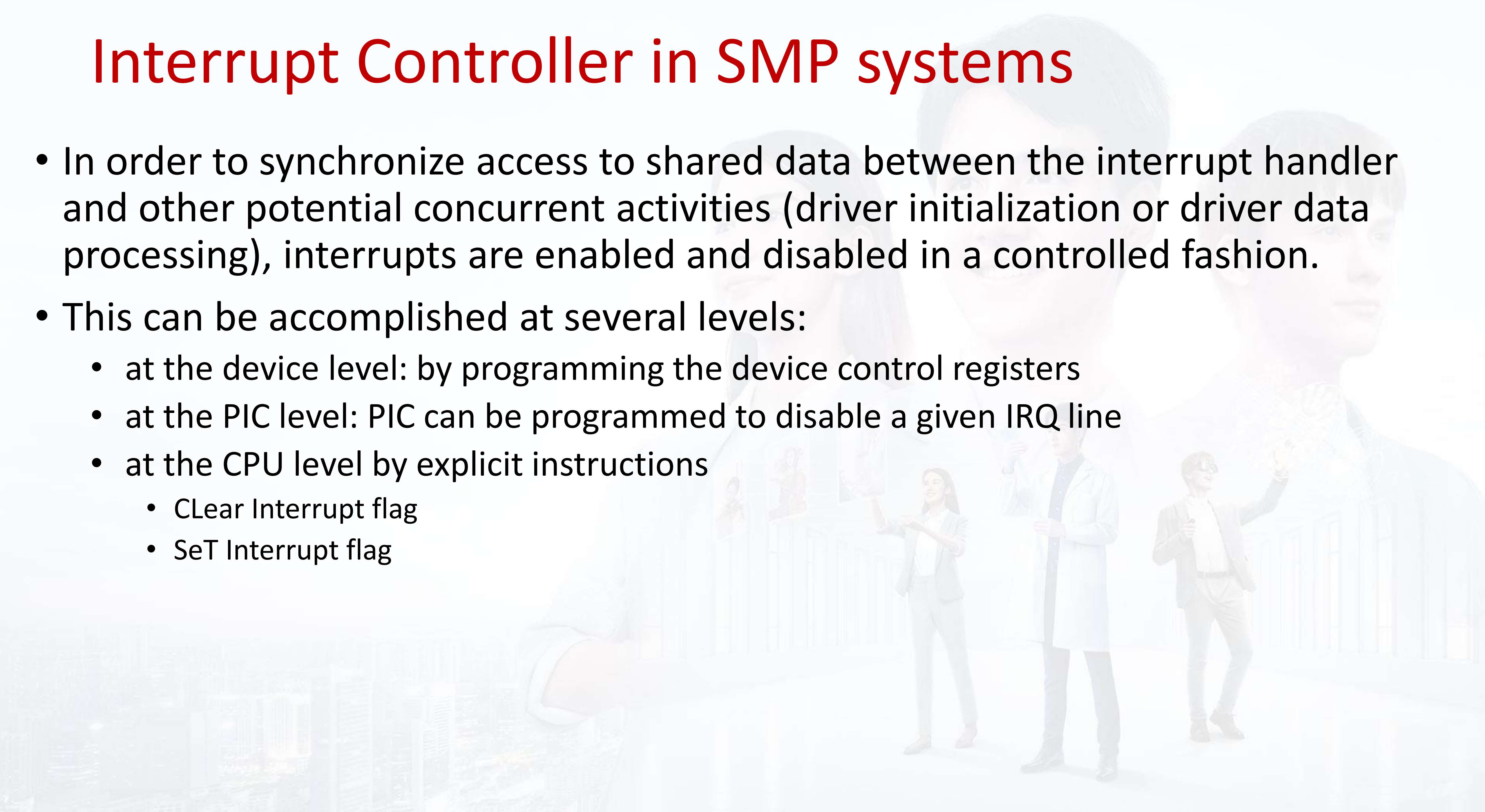
Programmable Interrupt Controller

- A PIC usually has a set of ports used to exchange information with the CPU.
- When a device, connected to one of the PIC's IRQ lines, needs CPU attention the following flow happens:
  - device raises an interrupt on the corresponding IRQn pin
  - PIC converts the IRQ into a vector number and writes it to a port for CPU to read
  - PIC raises an interrupt on CPU INTR pin
  - PIC waits for CPU to acknowledge an interrupt before raising another interrupt
  - CPU acknowledges the interrupt then it starts handling the interrupt
- Once the interrupt is acknowledged by the CPU the IC can request another interrupt, regardless if the CPU finished handled the previous interrupt or not.
- Thus, depending on how the OS controls the CPU it is possible to have nested interrupts.

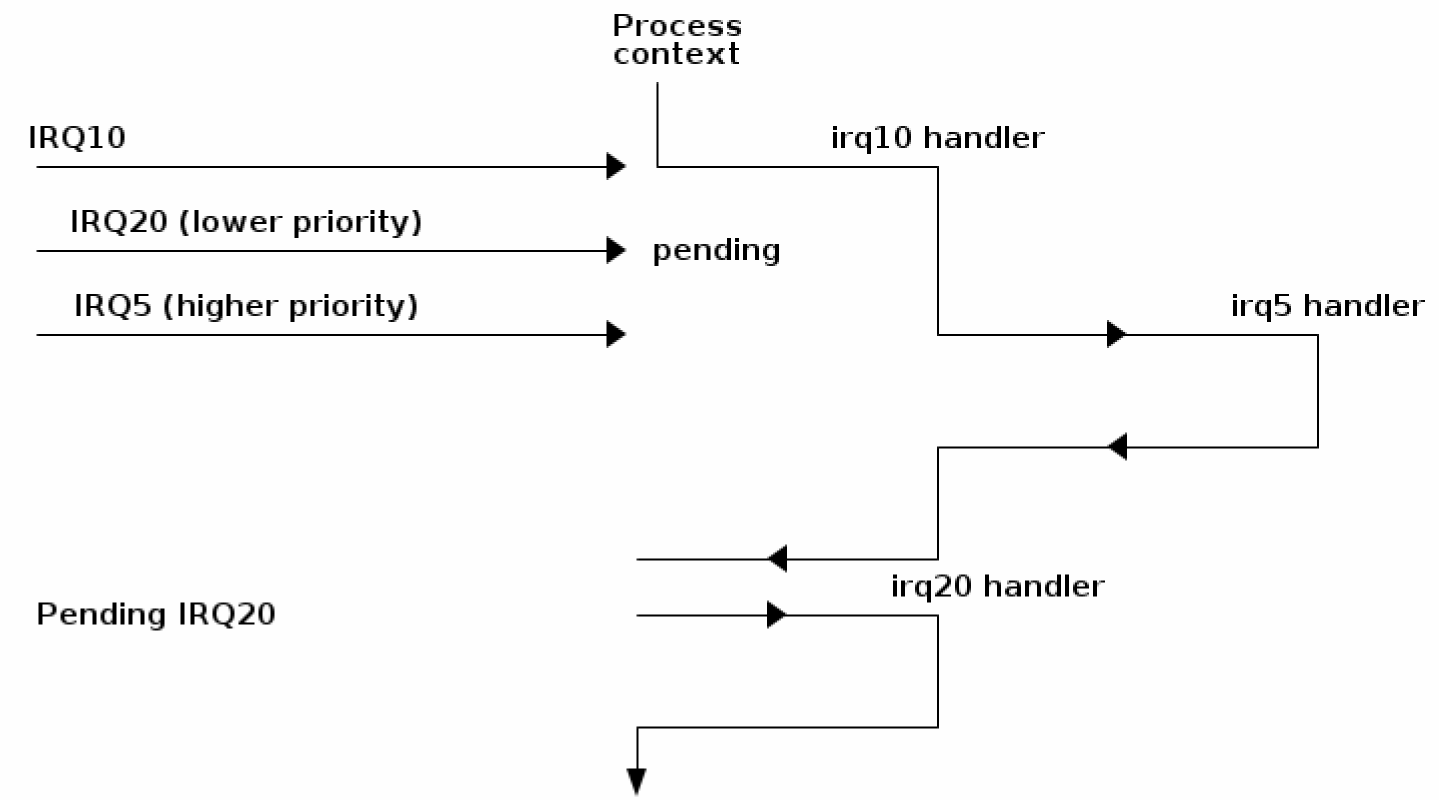


Interrupt Controller in SMP systems

- In order to synchronize access to shared data between the interrupt handler and other potential concurrent activities (driver initialization or driver data processing), interrupts are enabled and disabled in a controlled fashion.
- This can be accomplished at several levels:
  - at the device level: by programming the device control registers
  - at the PIC level: PIC can be programmed to disable a given IRQ line
  - at the CPU level by explicit instructions
    - CLear Interrupt flag
    - SeT Interrupt flag

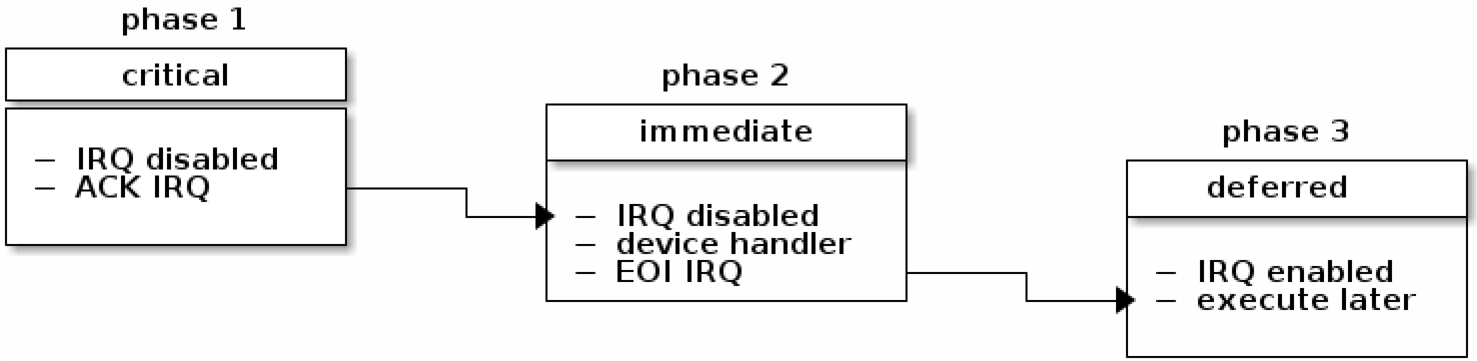


Interrupt priorities



- Most HW architectures support interrupt priorities, but not all OSes use them.
- When interrupt priority is enabled, it permits interrupt nesting only for those interrupts that have a higher priority than the current priority level.

Interrupt handling in Linux



- Three phases: critical, immediate and deferred.

1. The kernel will run the generic IH that determines the interrupt number, the IH for this particular interrupt and the IC. At this point any timing critical actions will also be performed (e.g. acknowledge the interrupt at the IC level). Local processor interrupts are disabled for the duration of this phase and continue to be disabled in the next phase.
2. All of the device driver's handlers associated with this interrupt will be executed. At the end of this phase, the IC's "end of interrupt" method is called to allow the IC to reassert this interrupt. The local processor interrupts are enabled at this point.
3. Interrupt context deferrable actions will be run. These are AKA "bottom half" of the interrupt (the upper half being the part of the interrupt handling that runs with interrupts disabled). At this point, interrupts are enabled on the local processor.

Interrupt Context

- While an interrupt is handled (from the time the CPU jumps to the interrupt handler until the interrupt handler returns - e.g. IRET is issued) it is said that code runs in "interrupt context".
- Code that runs in interrupt context has the following properties:
  - it runs as a result of an IRQ (not of an exception)
  - there is no well defined process context associated
  - not allowed to trigger a context switch (no sleep, schedule, or user memory access)

Deferrable actions

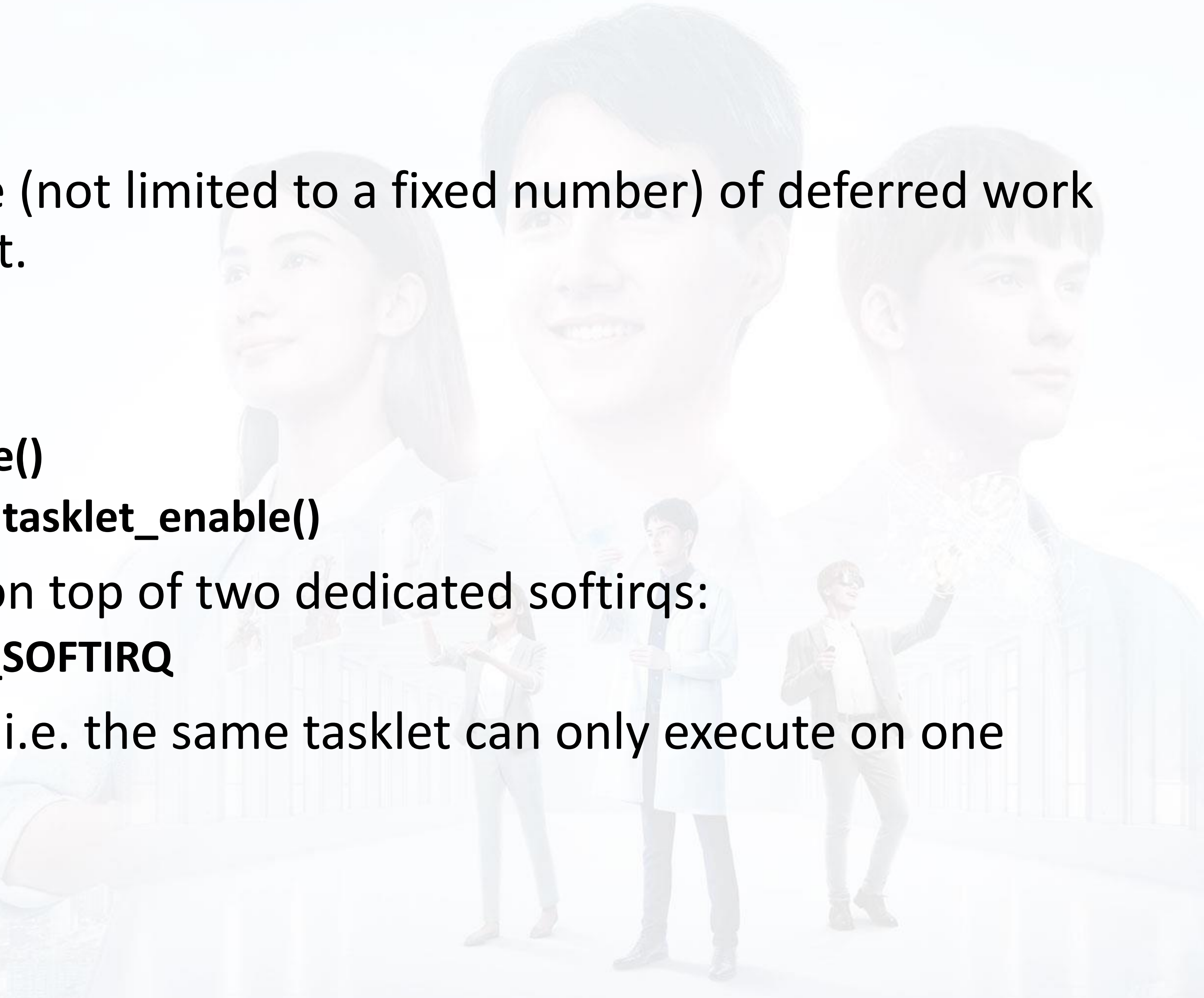
- Deferrable actions are used to run callback functions at a later time.
- If deferrable actions scheduled from an interrupt handler, the associated callback function will run after the interrupt handler has completed.
- There are two large categories of deferrable actions: those that run in interrupt context and those that run in process context.
- The purpose of interrupt context deferrable actions is to avoid doing too much work in the interrupt handler function.
- Running for too long with interrupts disabled can have undesired effects such as increased latency or poor system performance due to missing other interrupts (e.g. dropping network packets because the CPU did not react in time to dequeue packets from the network interface and the network card buffer is full).
- Deferrable actions have APIs to: **initialize** an instance, **activate** or **schedule** the action and **mask/disable** and **unmask/enable** the execution of the callback function. The latter is used for synchronization purposes between the callback function and other contexts.
- Typically the device driver will initialize the deferrable action structure during the device instance initialization and will activate / schedule the deferrable action from the interrupt handler.

Soft IRQs

- Soft IRQs is the term used for the low-level mechanism that implements deferring work from interrupt handlers but that still runs in interrupt context.
- Soft IRQ APIs:
  - initialize: **open_softirq()**
  - activation: **raise_softirq()**
  - masking: **local_bh_disable()**, **local_bh_enable()**
- Once activated, the callback function **do_softirq()** runs either:
  - after an interrupt handler or
  - from the ksoftirqd kernel thread
- Since softirqs can reschedule themselves or other interrupts can occur that reschedules them, they can potentially lead to (temporary) process starvation if checks are not put into place.
- Currently, the Linux kernel does not allow running soft irqs for more than **MAX_SOFTIRQ_TIME** or rescheduling for more than **MAX_SOFTIRQ_RESTART** consecutive times.
- Once these limits are reached a special kernel thread, **ksoftirqd** is wake-up and all of the rest of pending soft irqs will be run from the context of this kernel thread
  - minimum priority kernel thread
  - runs softirqs after certain limits are reached
  - tries to achieve good latency and avoid process starvation

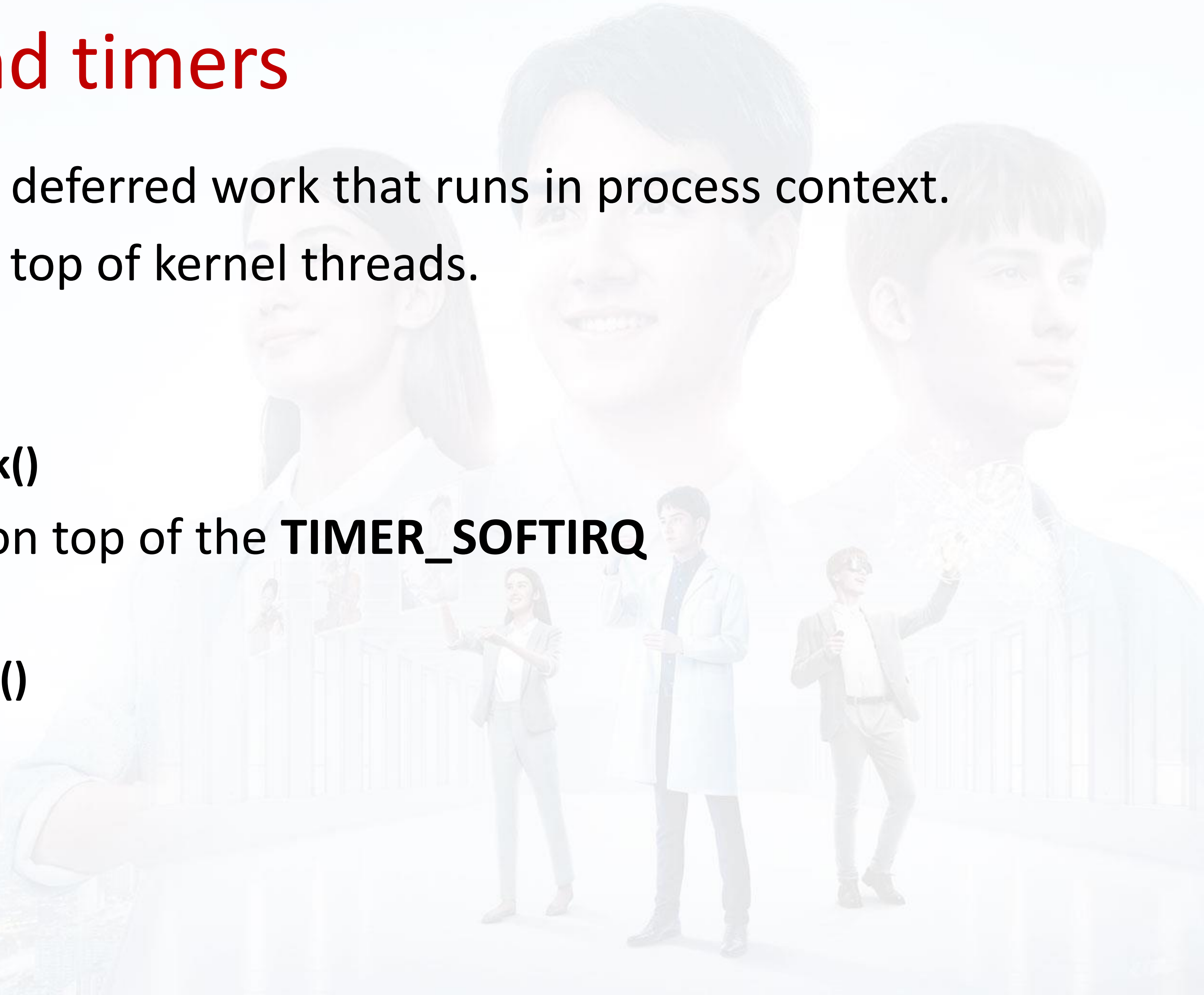
Tasklets

- Tasklets are a dynamic type (not limited to a fixed number) of deferred work running in interrupt context.
- Tasklets API:
  - initialization: **tasklet_init()**
  - activation: **tasklet_schedule()**
  - masking: **tasklet_disable()**, **tasklet_enable()**
- Tasklets are implemented on top of two dedicated softirqs:
  - **TASKLET_SOFITIRQ** and **HI_SOFTIRQ**
- Tasklets are also serialized, i.e. the same tasklet can only execute on one processor.



Workqueues and timers

- Workqueues are a type of deferred work that runs in process context.
- They are implemented on top of kernel threads.
- Workqueues API:
  - init: **INIT_WORK**
  - activation: **schedule_work()**
- Timers are implemented on top of the **TIMER_SOFTIRQ**
- Timer API:
  - initialization: **setup_timer()**
  - activation: **mod_timer()**



Deferrable actions summary

- softIRQ
  - runs in interrupt context
  - statically allocated
  - same handler may run in parallel on multiple cores
- tasklet
  - runs in interrupt context
  - can be dynamically allocated
  - same handler runs are serialized
- workqueues
  - run in process context

Symmetric Multi-Processing

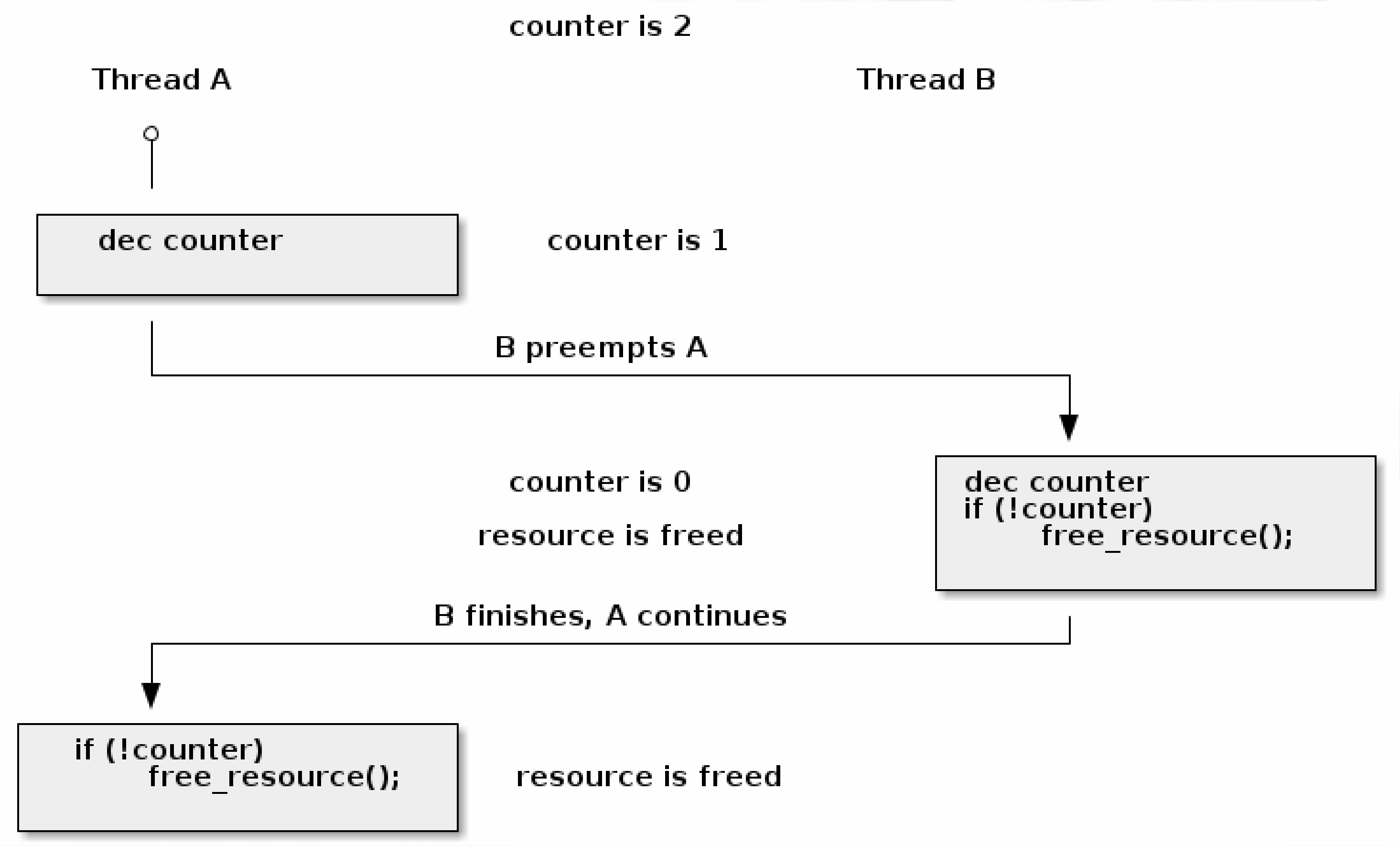
- Kernel concurrency
- Atomic operations
- Spinlocks
- Cache trashing
- Process and Interrupt context synchronization
- Mutexes
- Per CPU data
- Memory ordering and barriers
- Read-Copy update

Synchronization basics

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

Classic race condition example:

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



counter is 2

Thread A

Thread B

```
dec counter
```

counter is 1

B preempts A

```
dec counter
if (!counter)
      free_resource();
```

counter is 0

resource is freed

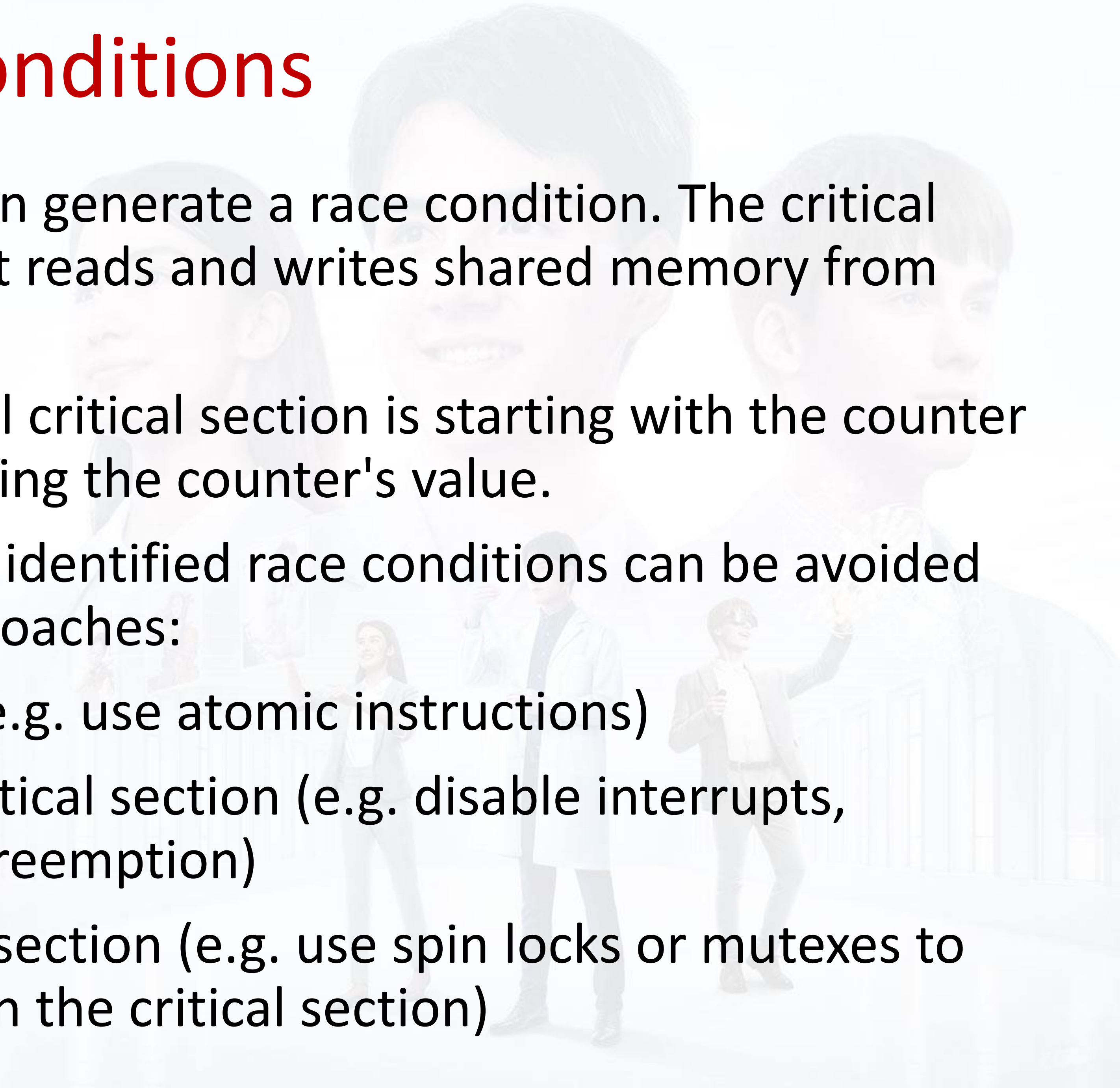
B finishes, A continues

```
if (!counter)
      free_resource();
```

resource is freed

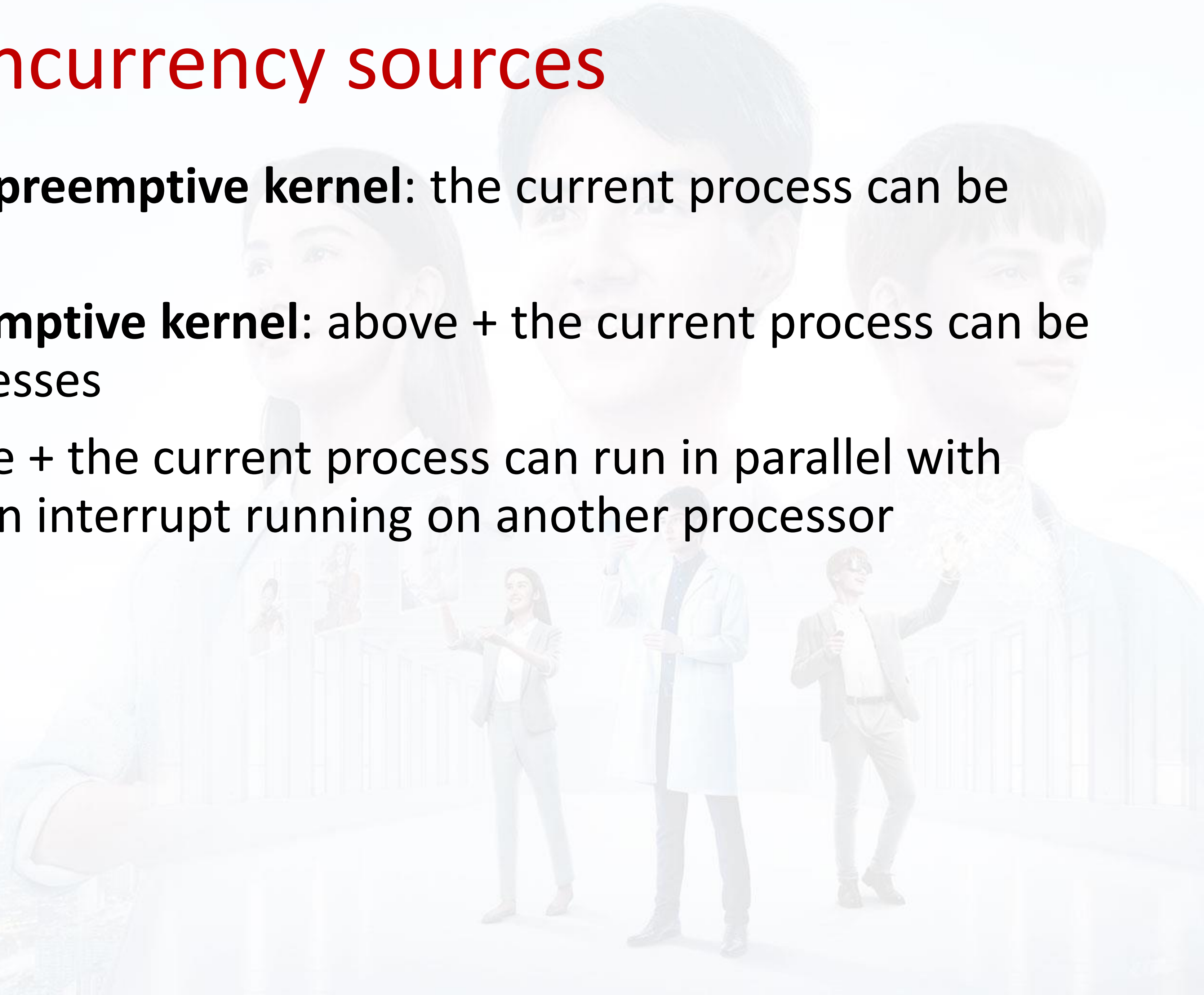
How to avoid race conditions

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



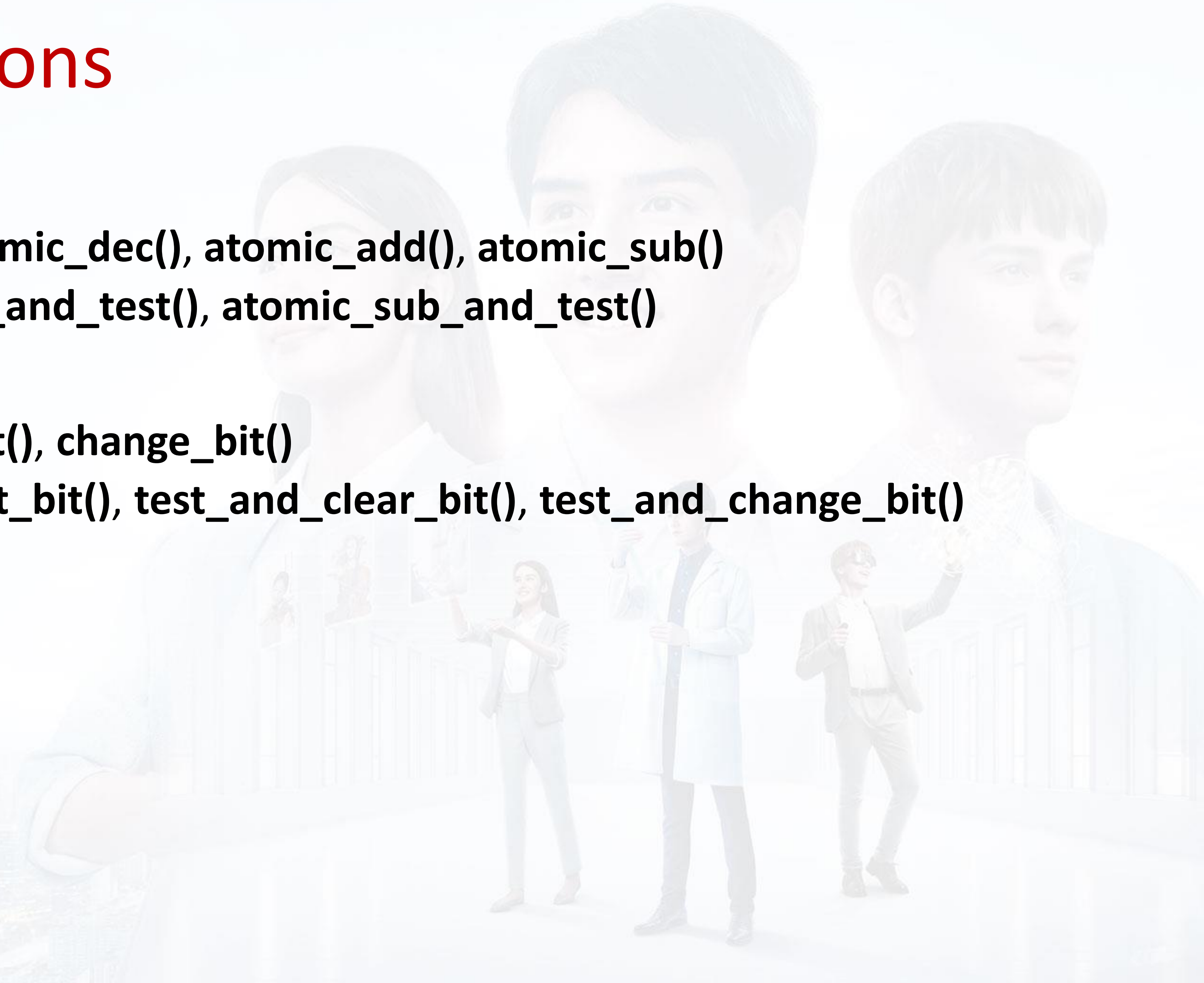
Linux kernel concurrency sources

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



Atomic operations

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

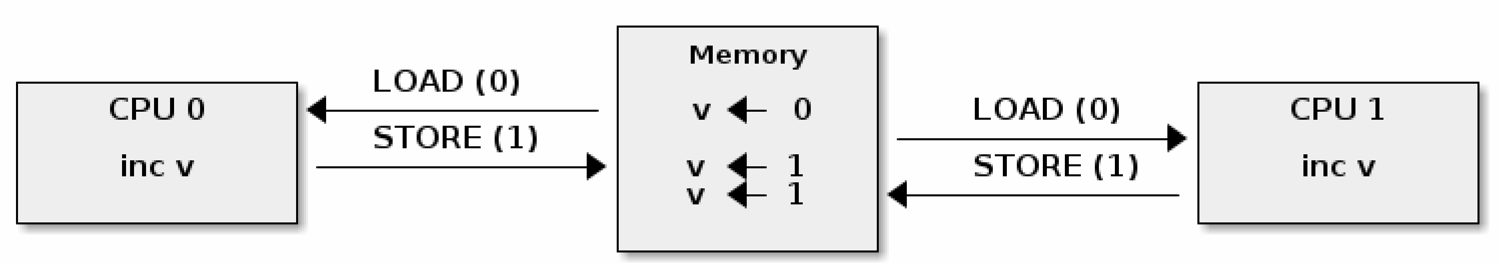


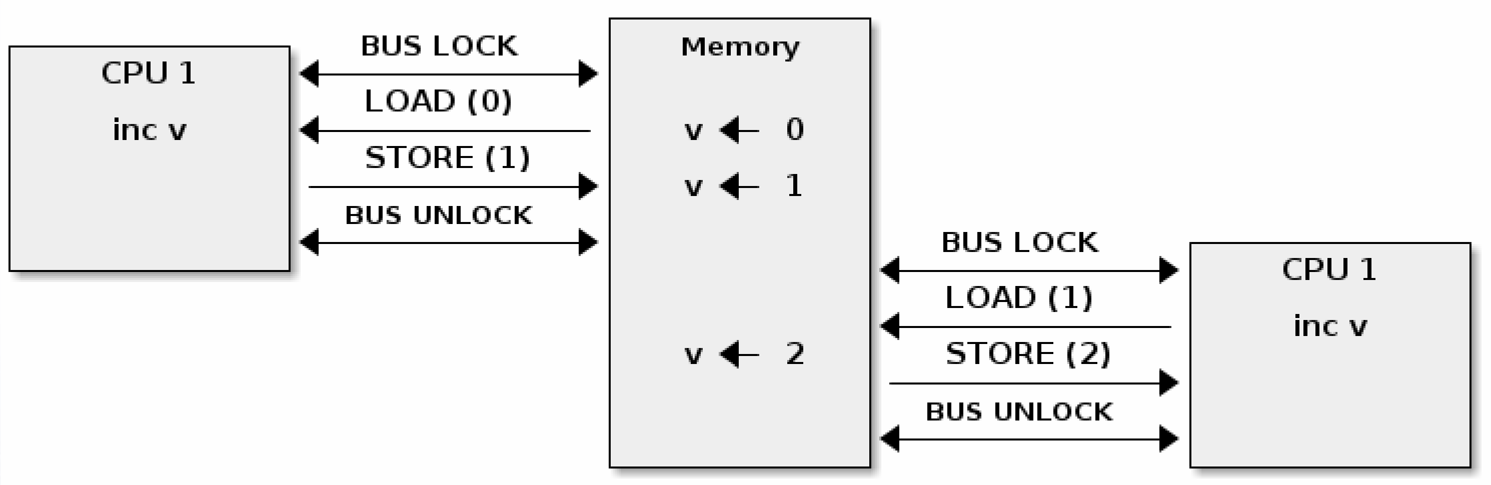
Using atomic_dec_and_test()

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

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

Atomic_ may be not atomic in SMP



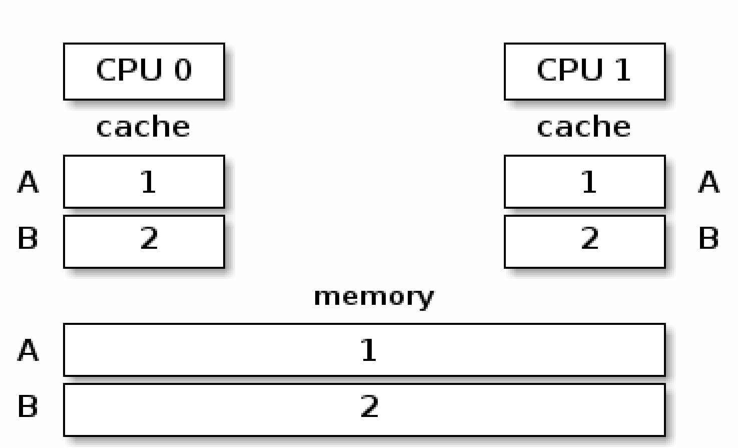


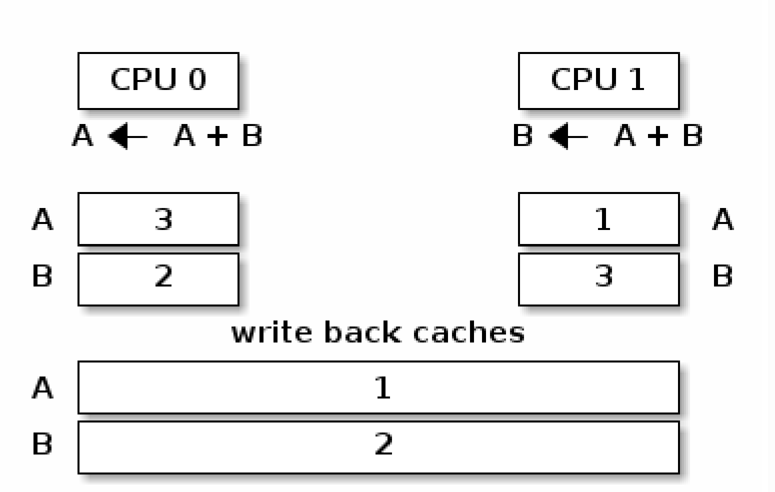
Spin locks

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

Cache coherency in multi-processor systems

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





Cache coherency protocols

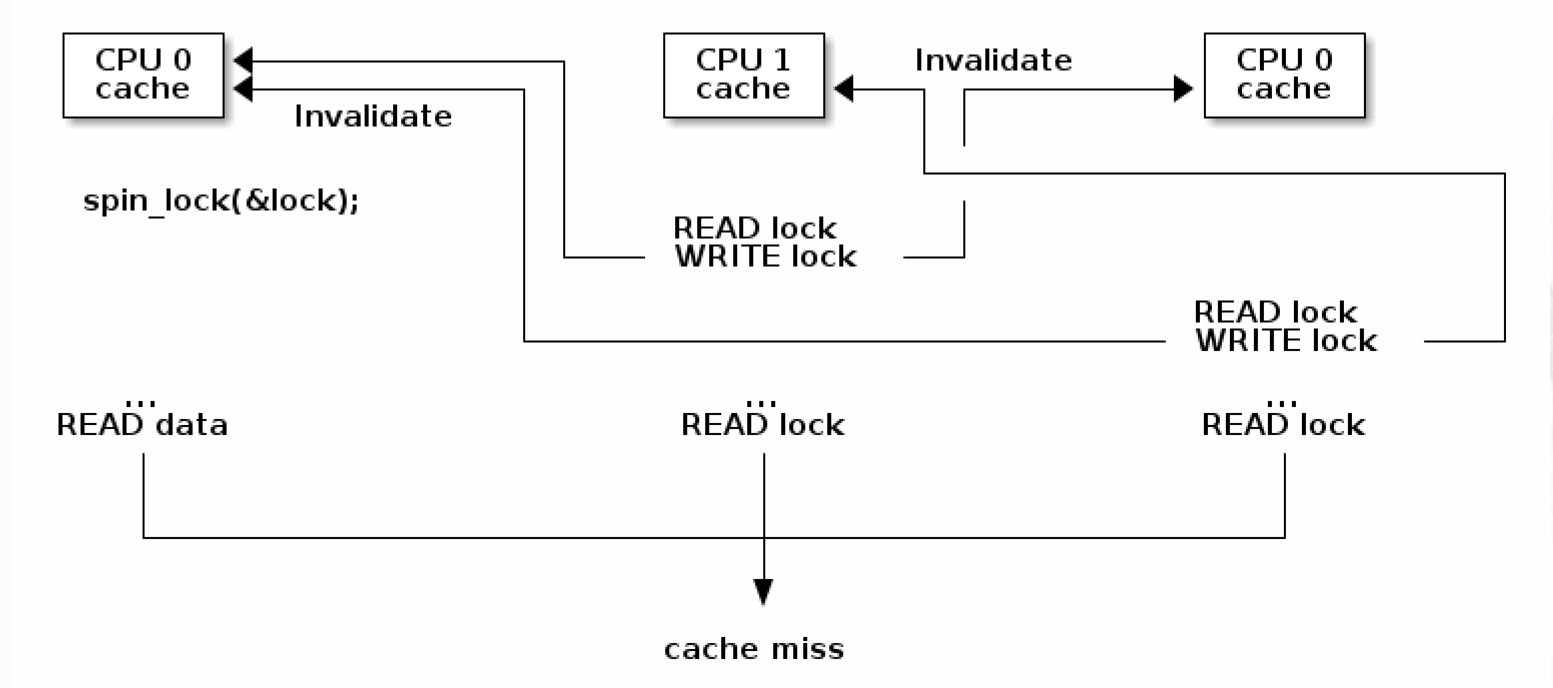
- Multi-processor systems use cache coherency protocols. There are two main types of cache coherency protocols:
- Bus snooping (sniffing) based: memory bus transactions are monitored by caches and they take actions to preserve coherency
- Directory based: there is a separate entity (directory) that maintains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

MESI cache coherency protocol

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

Cache trashing

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



Process and Interrupt Context Synchronization

- TBD at the next lesson