ITMO Advanced OS 2024

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

Lesson 2024.09.10

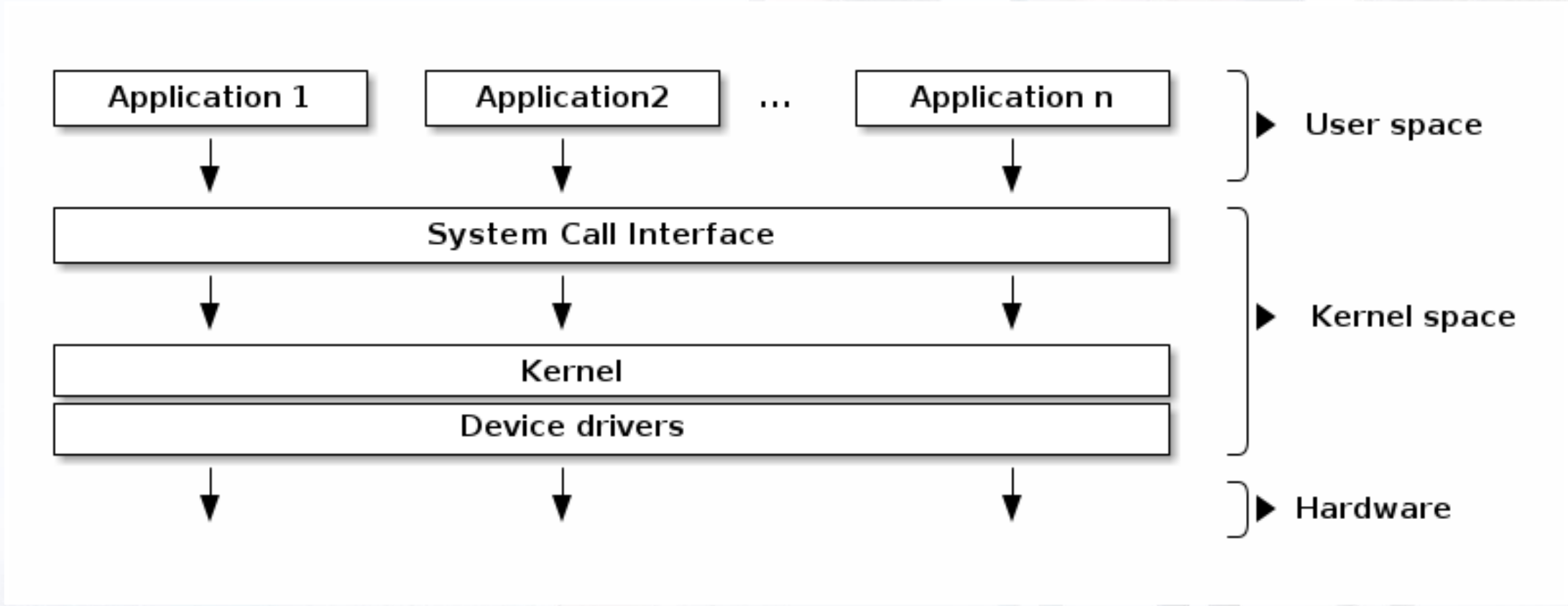
Contents

- Recommended literature
- Basic operating systems terms and concepts
- Very generic overview of the Linux (Harmony OS) kernel

Recommended literature

- Modern Operating Systems 4th Edition. By Herbert Bos, Andrew Stuart Tanenbaum
- OPERATING SYSTEMS INTERNALS AND DESIGN PRINCIPLES. By William Stallings
- Operating System Concepts, 10th Edition By Abraham Silberschatz et al.

General OS architecture 1 of 2



General OS architecture 2 of 2

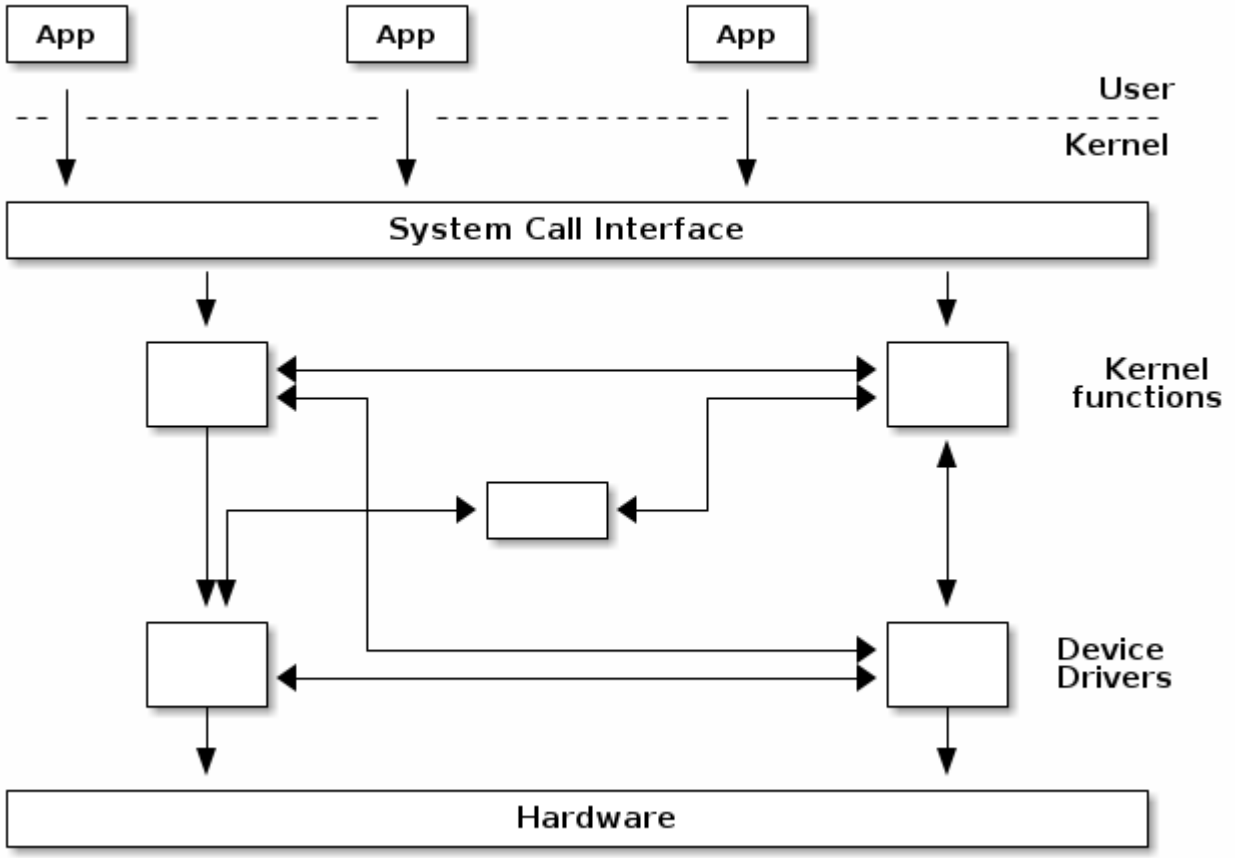
- System calls are part of kernel APIs. They are the boundary where execution mode switches from user mode to kernel mode
- System calls are rarely changed. Linux particularly enforces this (as opposed to in-kernel APIs that can change as needed)
- Kernel code itself can be logically separated in core kernel code and device drivers code. Device drivers code is responsible of accessing particular devices while the core kernel code is generic. Core kernel can be further divided into multiple logical subsystems (e.g. file access, networking, process management, etc.)

Execution modes and memory protection

- CPU: hypervisor mode, kernel mode, user mode
  - Hypervisor (virtual machine monitor) mode to run VMs (OSes) inside OS
  - Kernel mode - to run code with higher privileges while user mode means running applications with lower privileges
  - Code running in kernel mode can fully control the CPU while code that runs in user mode has certain limitations. E.g., CPU interrupts can only be disabled or enabled while running in kernel mode. If interrupt is attempted in user mode an exception will be generated and the kernel will handle it
  - Besides HW provides restrictions (protection rings) on I/O ports and CPU instructions
- RAM: kernel space, user space
  - kernel space is access protected, user applications can not access it directly, while user space can be directly accessed from kernel mode code
  - Physical and virtual (paging/protected mode) memory

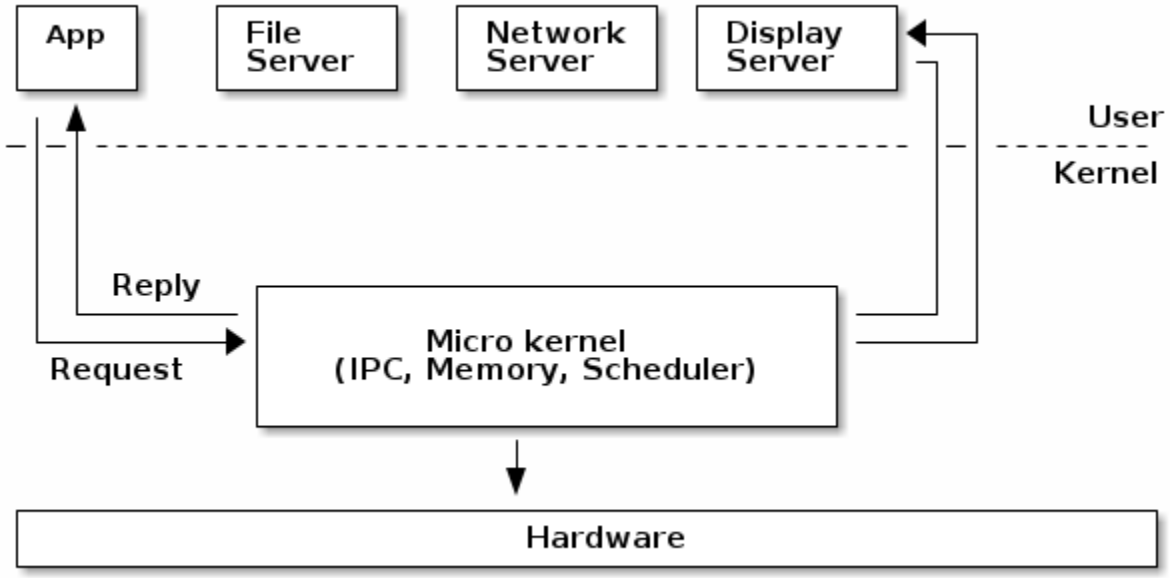
Monolithic kernel

- no access protection between the various kernel subsystems
- logical separation between subsystems, e.g. between the core kernel and device drivers with relatively strict APIs



Micro-kernel

- In micro-kernel large parts of the kernel are protected from each-other, usually running as services in user space
- kernel proper contains code for message passing between processes, and a few other things.
- Memory protection between services - but at a cost of performance



Monolithic kernels are modular

- Components can enabled or disabled at compile time
- Support of loadable kernel modules (at runtime)
- Organize the kernel in logical, independent subsystems
- Strict interfaces but with low performance overhead: macros, inline functions, function pointers

Monolitic vs. Micro. "Hybrid" kernels

- Monolitic is faster (therefore less battery drain)
- Micro is more reliable but slower (actually depends on…)
- Windows is claimed “Hybrid” (PR-ed), where major services run in monolithic kernel-mode
- 2007 LWN article on Monolitic vs. Micro https://lwn.net/Articles/220255/

CPU: OS multi-processing

- An OS that supports the parallel ("simultaneous“) execution of multiple processes
- Implemented by switching between running processes to allow process to do something useful
- Implementations may be:
  - Cooperative (non-preemptive) - processes cooperate to achieve multitasking. Process will run and relinquish CPU control back to the OS, which will then schedule another process
  - Preemptive - kernel will enforce time limits for each process, so that all processes have a chance. Each process is allowed to run a time slice (e.g. 100ms) after which, if still running, it is pre-empted and another process is scheduled

CPU: Non-preemprive and Preemptive kernels

- Preemptive multi-processing (multitasking) and preemptive kernels are different terms
- A kernel is preemptive if a process can be preempted (paused immediately) while running in kernel mode.
- However, non-preemptive kernels may support preemptive multitasking.

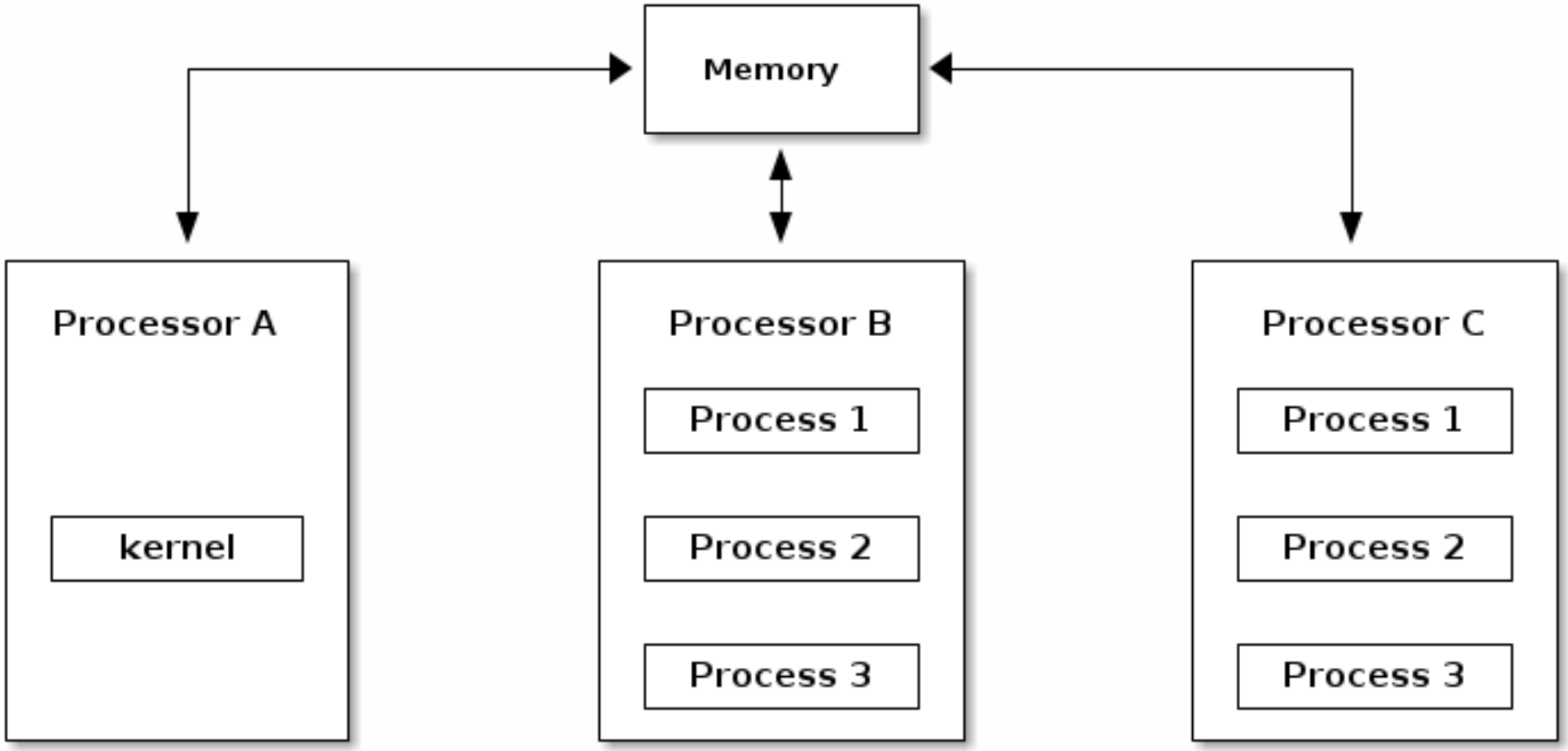
Non-preemptive kernels (Linux 2.4)

- Process continues to run until it finishes handling execution handler or voluntarily relinquishes CPU.
- It is less suitable for real-time programming as compared to preemptive kernel.
- Each and every task are explicitly given up CPU control.
- Generally does not allow preemption of process running in kernel mode.
- Response time is nondeterministic and is less responsive as compared to preemptive kernel.
- Higher priority task might have to wait for long time.
- Shared data generally requires semaphores.
- It can use non-reentrant code.
- It is less difficult to design non-preemptive kernels as compared to preemptive kernels
- It is less secure and less useful in real-world scenarios

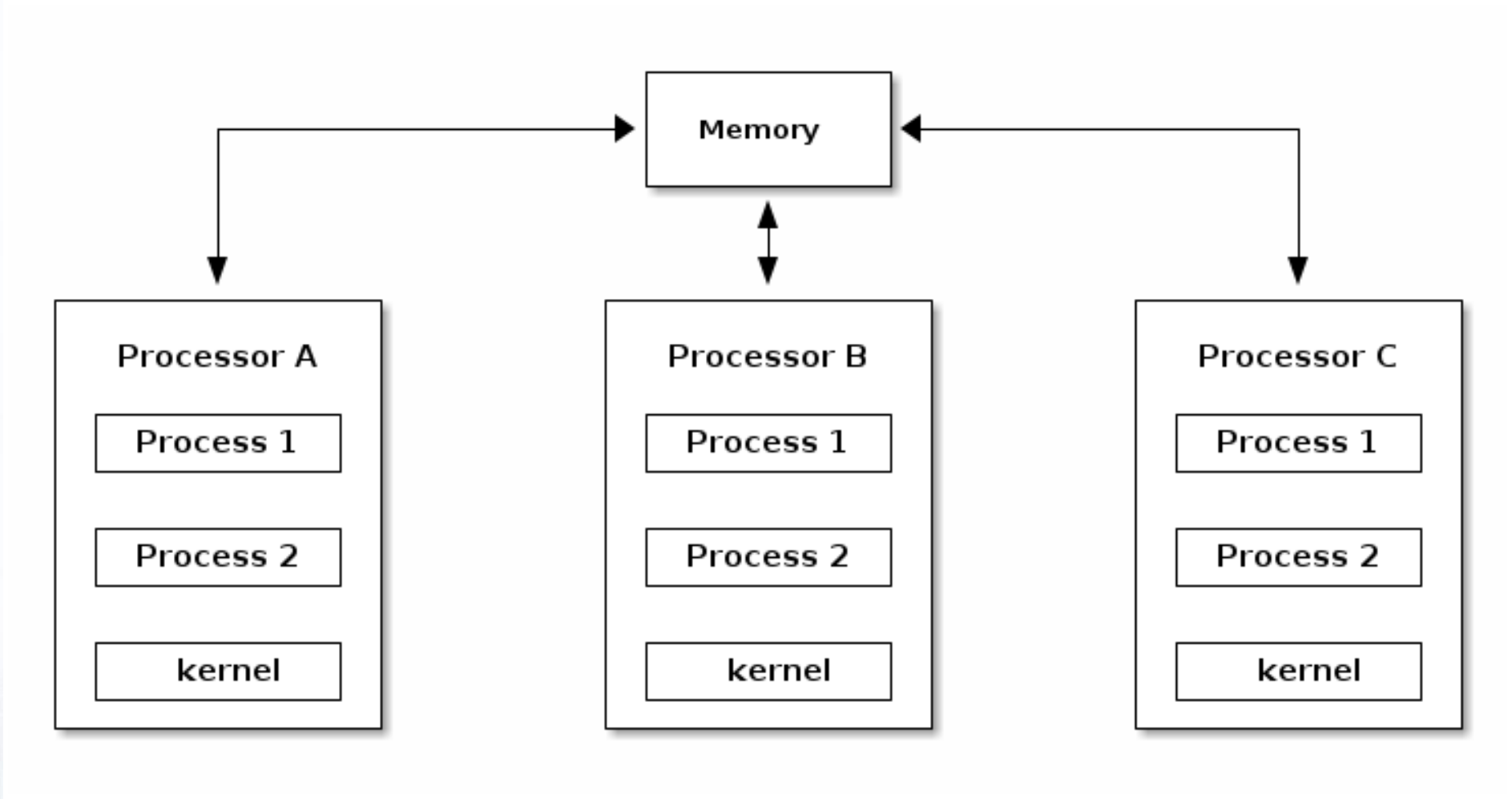
Preemptive kernels (Linux 2.6 and up)

- Any process that might be pre-empted (paused) immediately.
- More suitable for real time programming as compared to non-preemptive kernels.
- Higher priority task that are ready to run is given CPU control.
- Generally allows preemption even in kernel mode.
- Responsive time is deterministic and it is more responsive as compared to non-preemptive kernel.
- Higher priority task becomes ready, currently running task is suspended and moved to tail of ready queue.
- It does not require semaphores (wait, wake). Well, almost…
- It cannot use non-reentrant code.
- It is more difficult to design preemptive kernels as compared to non-preemptive kernel.
- It is more secure and more useful in real-world scenarios.

Asymmetric Multi-Processing (ASMP)

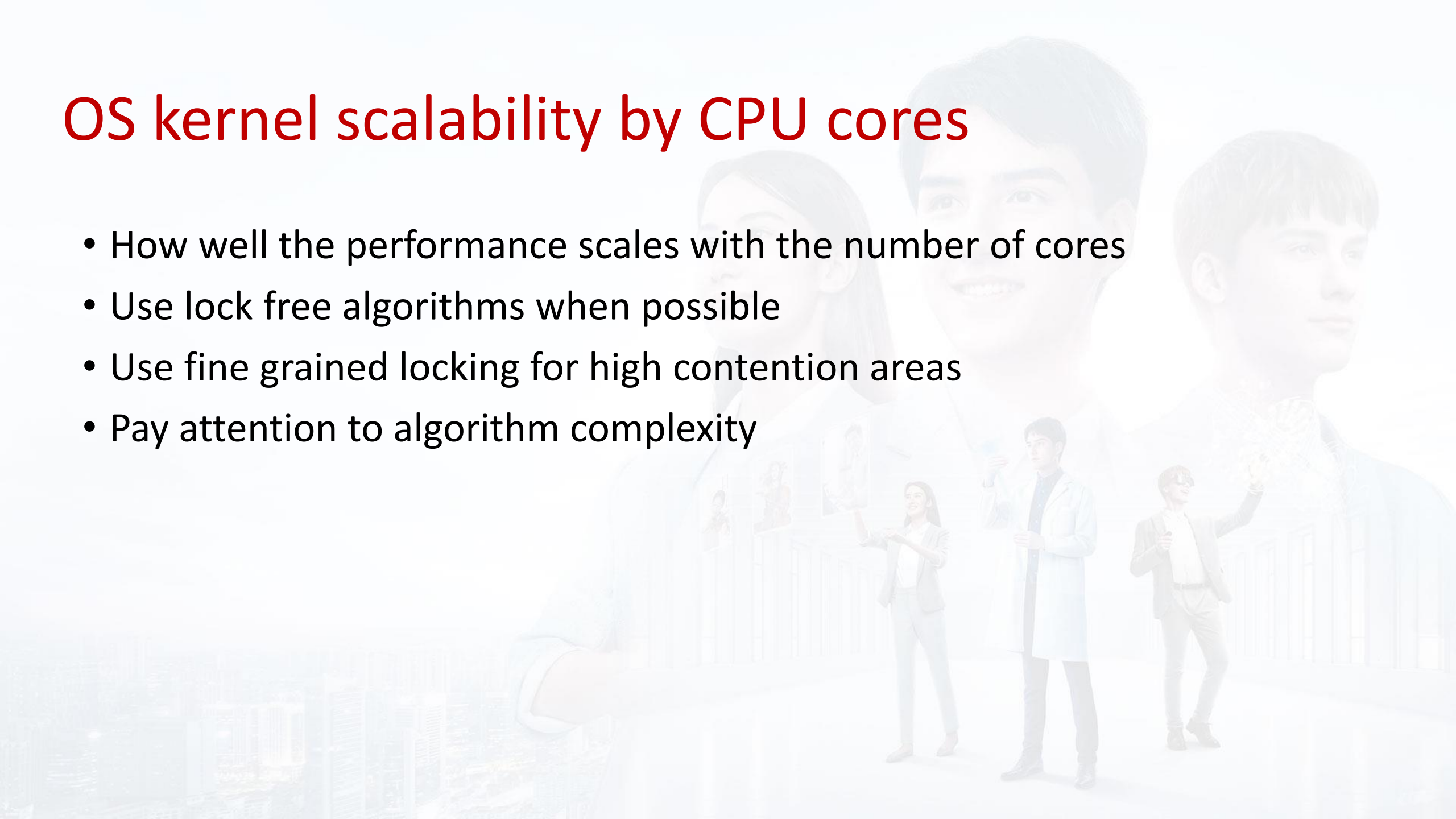


Symmetric Multi-Processing



OS kernel scalability by CPU cores

- How well the performance scales with the number of cores
- Use lock free algorithms when possible
- Use fine grained locking for high contention areas
- Pay attention to algorithm complexity



RAM: address spaces

- Physical address space
  - Physical RAM and peripheral memory
- Virtual address space
  - How the CPU sees the memory when (in protected/paging mode) executing processes
  - OS kernel is responsible of setting up a mapping to virtual address space where virtual pages are mapped to physical pages
  - Process address space is (part of) the virtual address space associated with a process. It is a continuous area that starts at zero
  - Kernel address space – RAM and peripherals as they are
  - Kernel and user share part of virtual space

VM Layout AArch64 Linux (4KB & 48 bit address)

Start	End	Size	Use
0000000000000000	0000ffffffffffff	256TB	user
ffff000000000000	ffff7fffffffffff	128TB	kernel logical memory map
[ffff600000000000	ffff7fffffffffff]	32TB	[kasan shadow region]
ffff800000000000	ffff800007ffffff	128MB	modules
ffff800008000000	fffffbffefffffff	124TB	vmalloc
fffffbfff0000000	fffffbfffdffffff	224MB	fixed mappings (top down)
fffffbfffe000000	fffffbfffe7fffff	8MB	[guard region]
fffffbfffe800000	fffffbffff7fffff	16MB	PCI I/O space
fffffbffff800000	fffffbffffffffff	8MB	[guard region]
fffffc0000000000	fffffdffffffffff	2TB	vmemmap
fffffe0000000000	ffffffffffffffff	2TB	[guard region]

- https://www.kernel.org/doc/html/latest/arm64/memory.html
- https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html

AARCH64 VM translation (4 levels & 48 bit VA)

- For 4kB page (granule) size the hardware can use a 4-level look up
- The 48-bit address has 9 address bits per translation level, that is 512 entries each, final 12 bits selecting a byte within the 4kB coming directly from the original address
- Bits 47:39 of the Virtual Address index into the 512 entry L0 table.
- Each of these table entries spans a 512 GB range and points to an L1 table.
- Within that 512 entry L1 table, bits 38:30 are used as index to select an entry and each entry points to either a 1GB block or an L2 table.
- Bits 29:21 index into a 512 entry L2 table and each entry points to a 2MB block or next table level.
- At the last level, bits 20:12 index into a 512 entry L3 table and each entry points to a 4kB page

Execution contexts

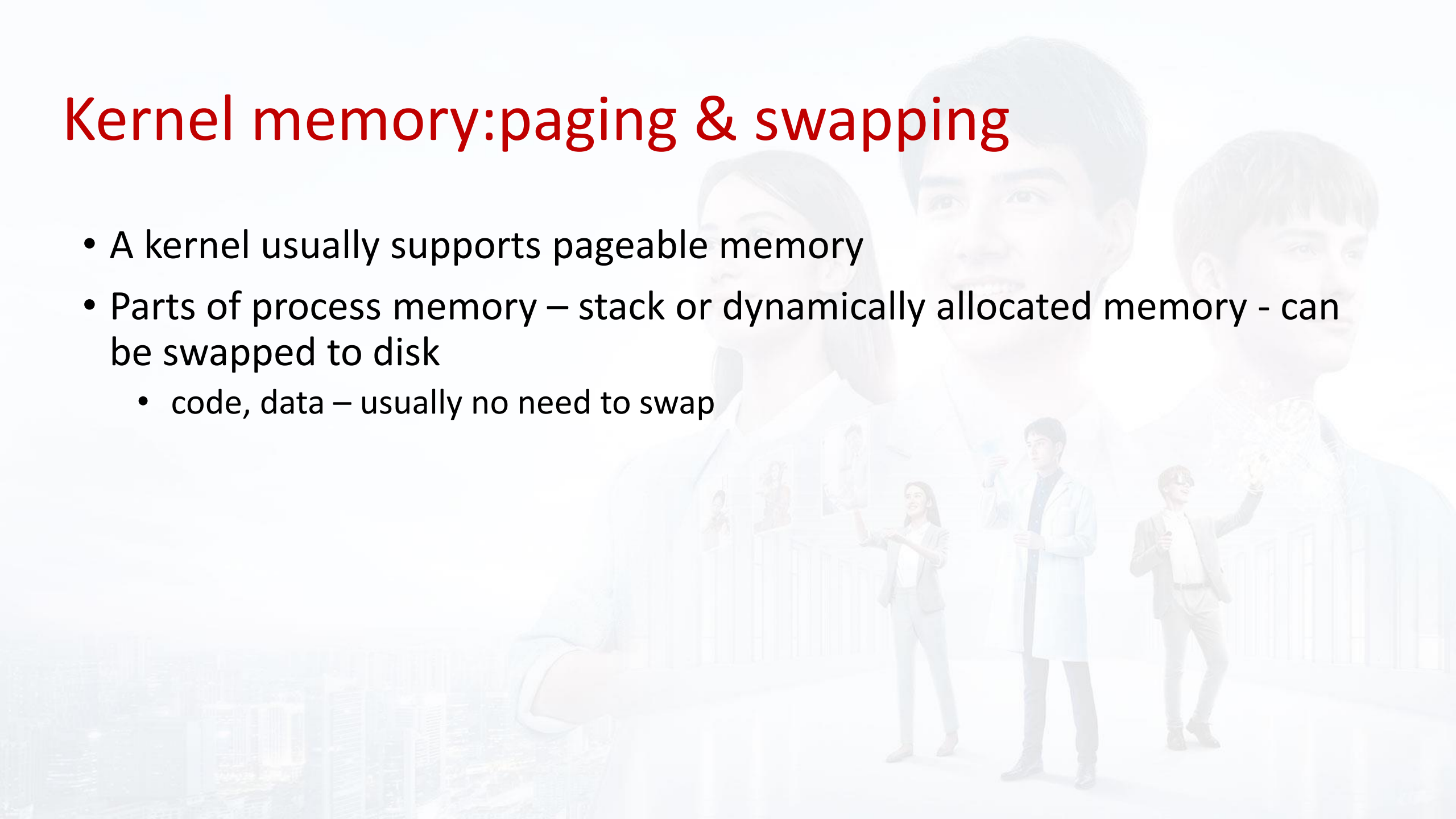
- Process context
  - Used by code that runs in user mode, part of a process
  - …and by code that runs in kernel mode, as a result of a system call issued by a process
  - Context switches
- Interrupt context
  - Used by code (Interrupt Handler, IH for short) that runs as a result of an interrupt
  - IH code always runs in kernel mode

User, Kernel and Interrupt stacks

- User stack grows downward to lower addresses, whereas dynamic allocations (heap) grow upwards to higher addresses. The user stack is only used while the process is running in user mode
- Each process has a kernel stack that is used to maintain the function call chain and local variables state while it is executing in kernel mode, as a result of a system call.
- The kernel stack is configured during compilation (e.g.2*4KB) kernel developer has to avoid allocating large structures on stack or recursive calls that are not properly bounded
- additional per-CPU interrupt stacks are used to process external interrupts

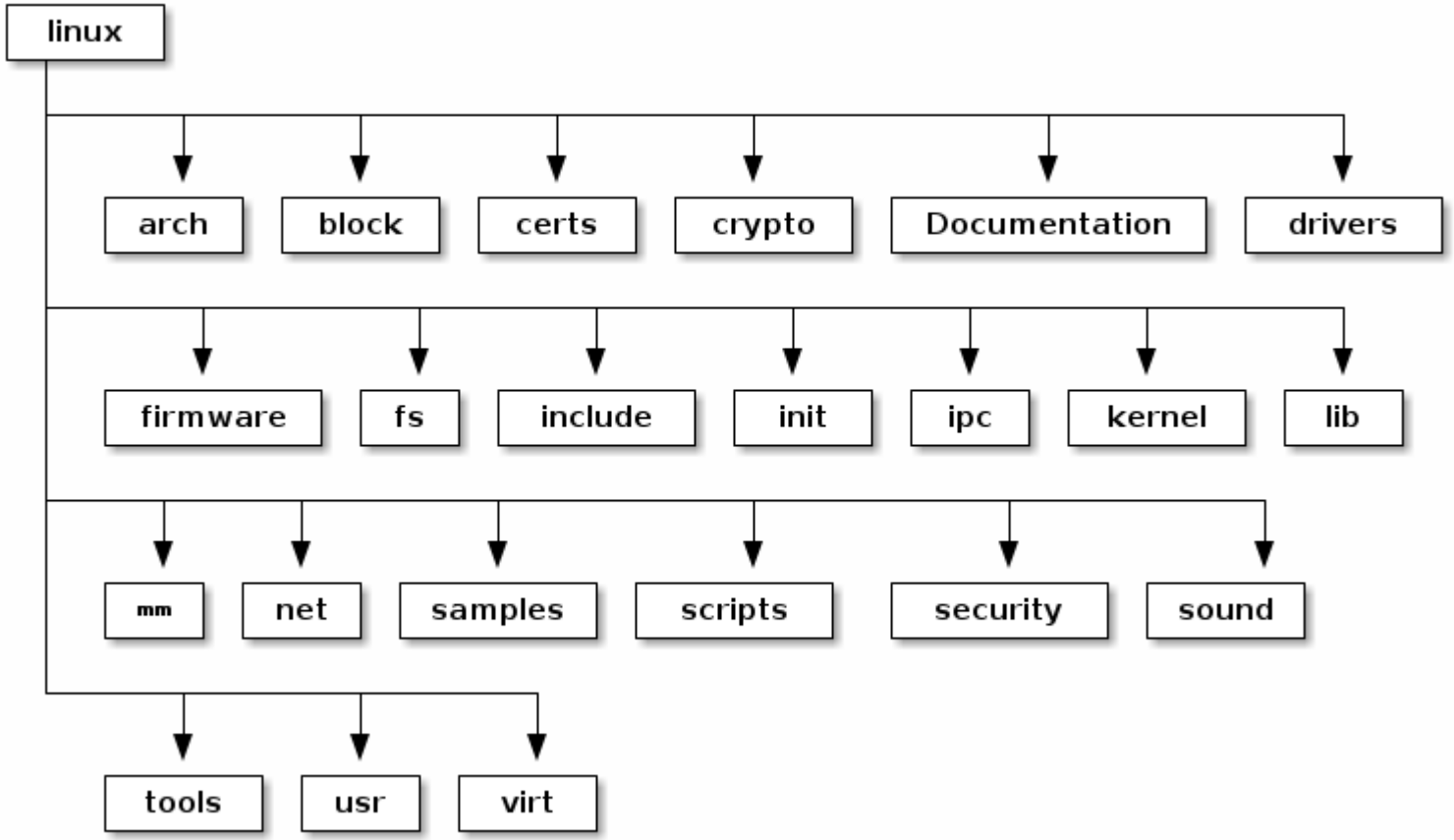
Kernel memory:paging & swapping

- A kernel usually supports pageable memory
- Parts of process memory – stack or dynamically allocated memory - can be swapped to disk
  - code, data – usually no need to swap

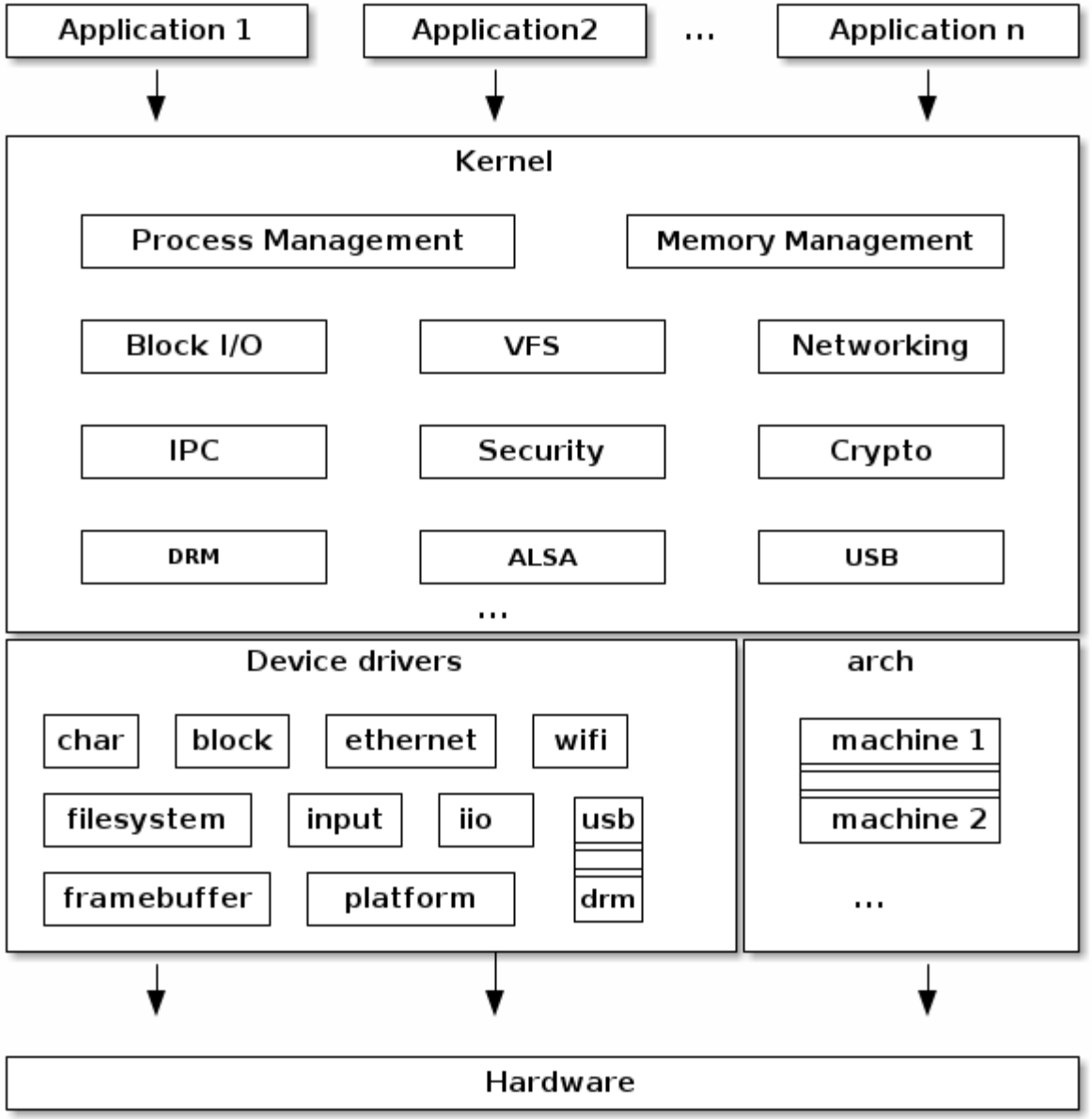


Linux kernel source code layout

- Architecture and machine specific code (C & ASM)
- Architecture independent code (C):
  - kernel core (further split in multiple subsystems)
  - etc
  - device drivers
  - etc



OS/Linux kernel architecture (flash back...)



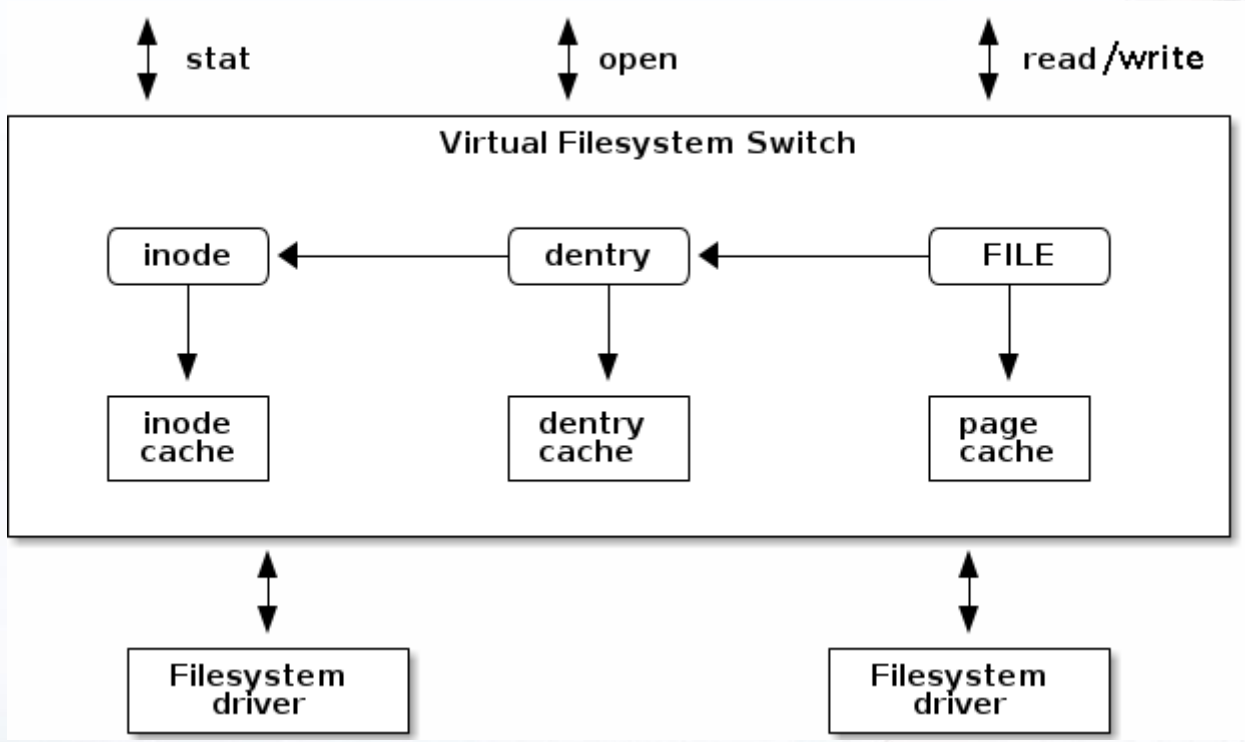
OS/Linux process management

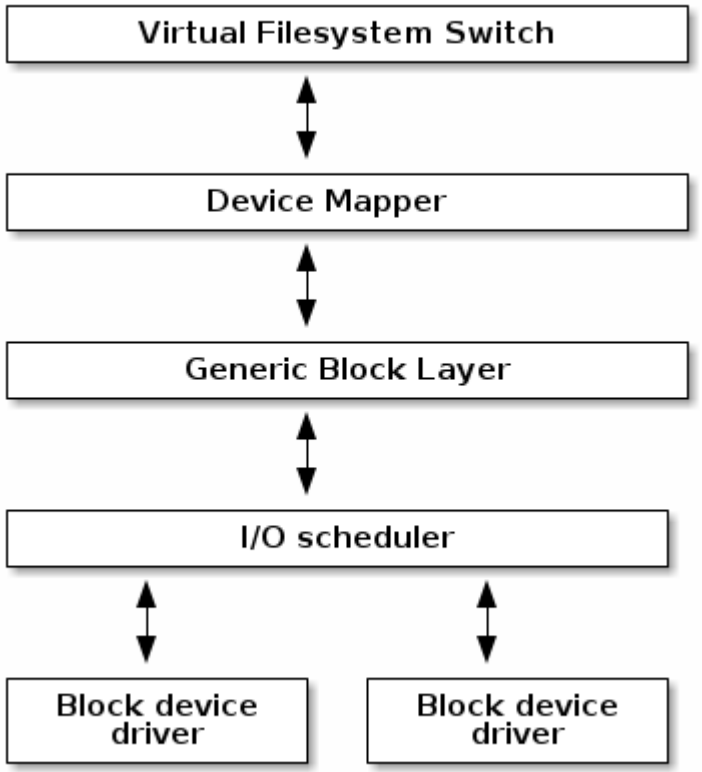
- Basic process management and POSIX threads support: fork(), exec(), wait(), pthreads, etc.
- Processes and threads are abstracted as tasks (struct task_struct)
  - task has pointers to resources, such as address space, file descriptors, IPC ids, etc.
  - resource pointers for tasks, that are part of the same process, point to the same resources
  - resources of tasks of different processes will point to different resources
- Operating system level virtualization
  - Namespaces: map a set of processes to a set of resources
  - Control groups: to organize processes hierarchically and distribute system resources along the hierarchy in a controlled and configurable manner
- Process/thread scheduling (large and important topic)

OS/Linux memory management

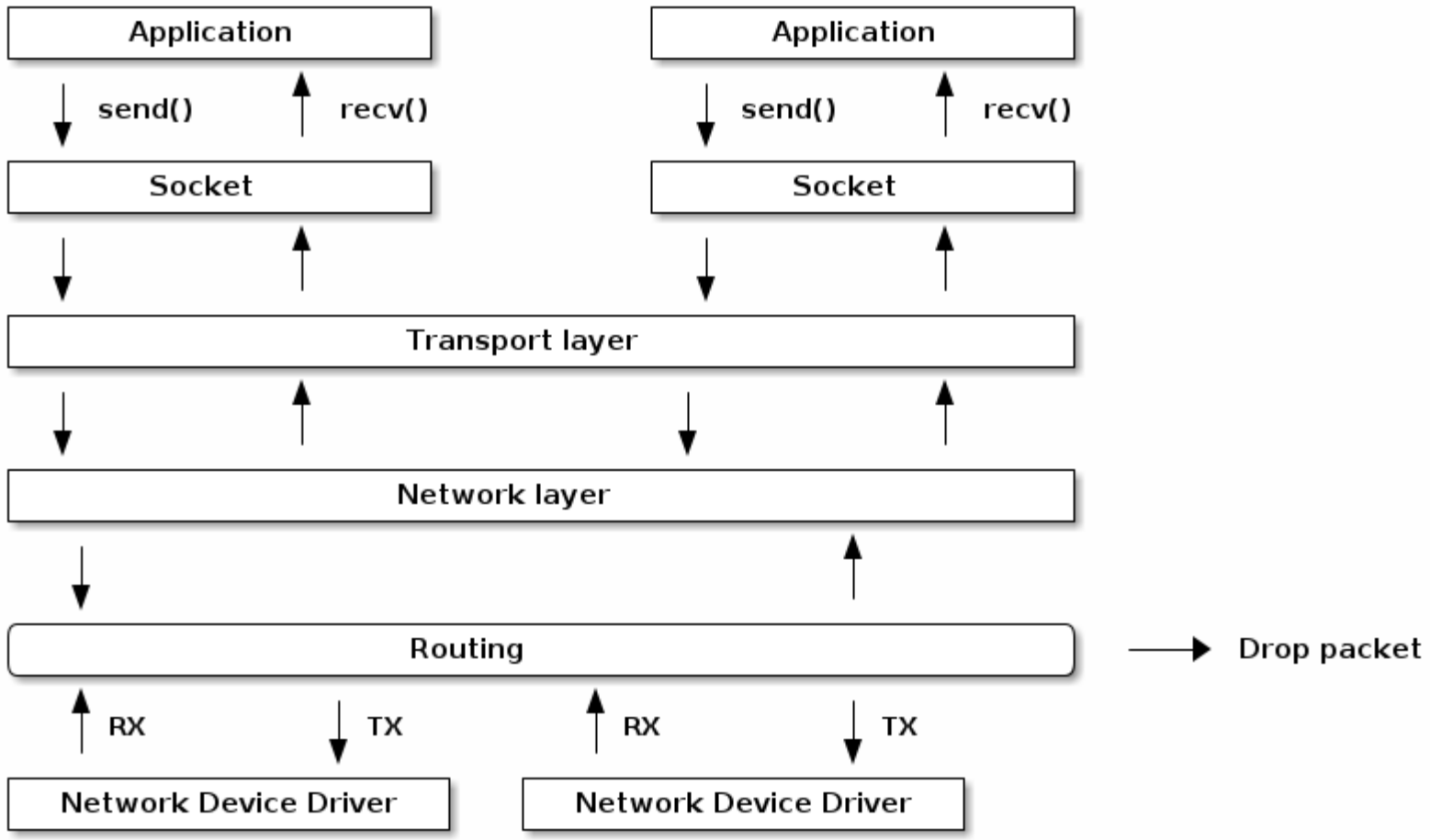
- Management of the physical memory: allocating and freeing memory
- Management of the virtual memory: paging, swapping, demand paging, copy on write
- User services: user address space management (e.g. mmap(), brk(), shared memory)
- Kernel services: SL*B allocators, vmalloc

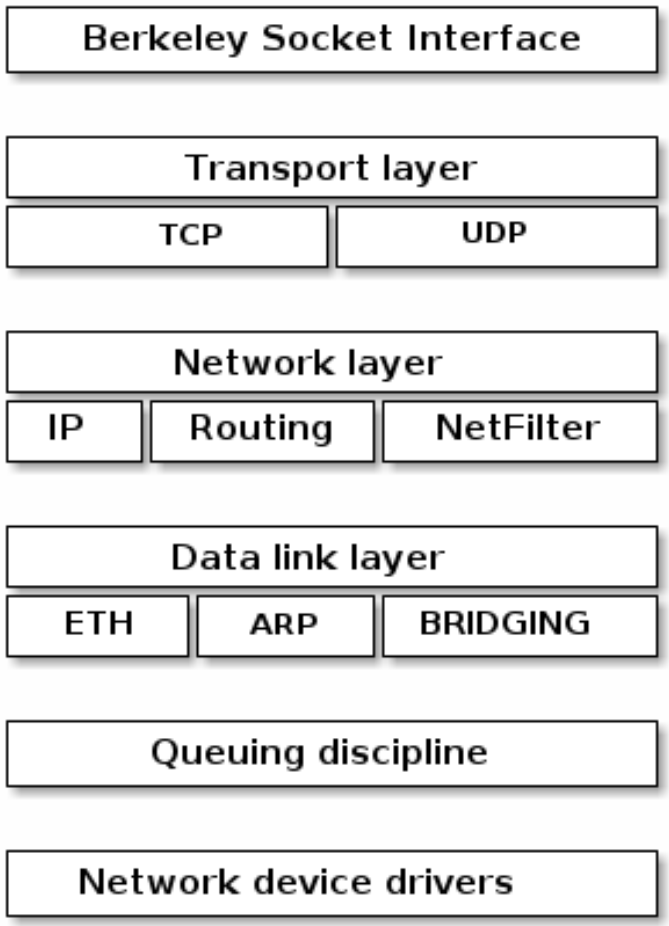
VFS and Block I/O management



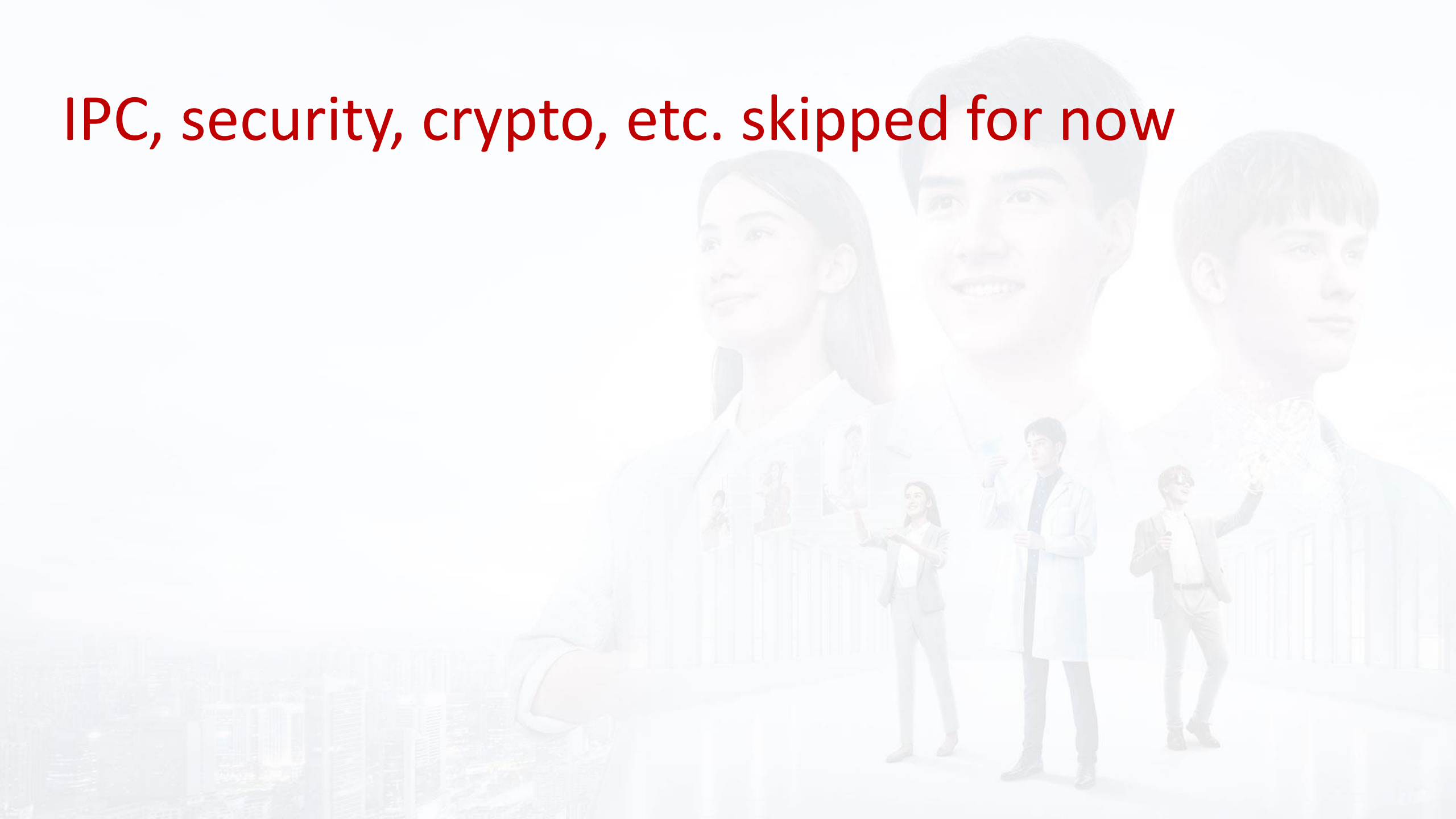


OS/Linux kernel network stack



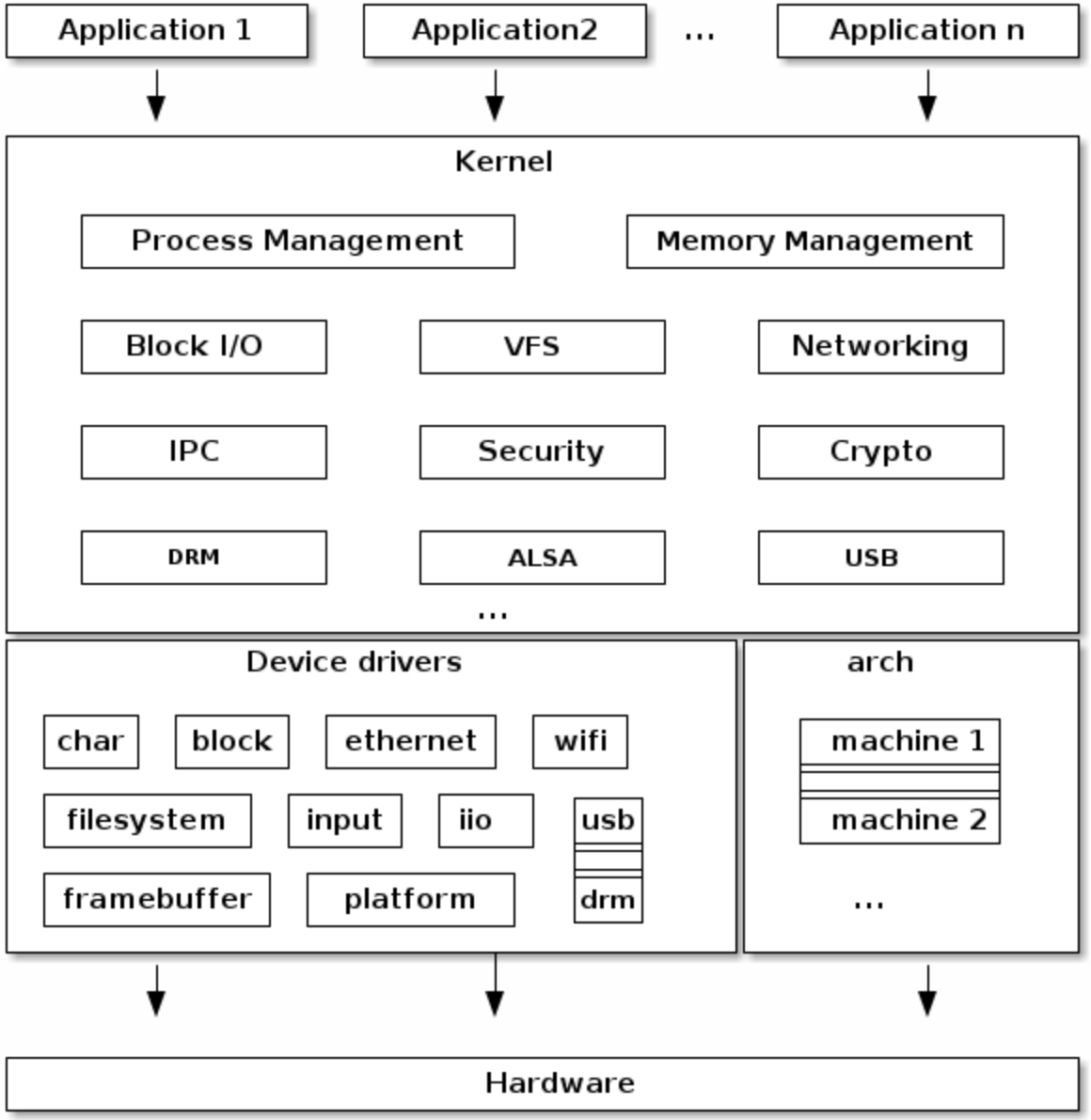


IPC, security, crypto, etc. skipped for now



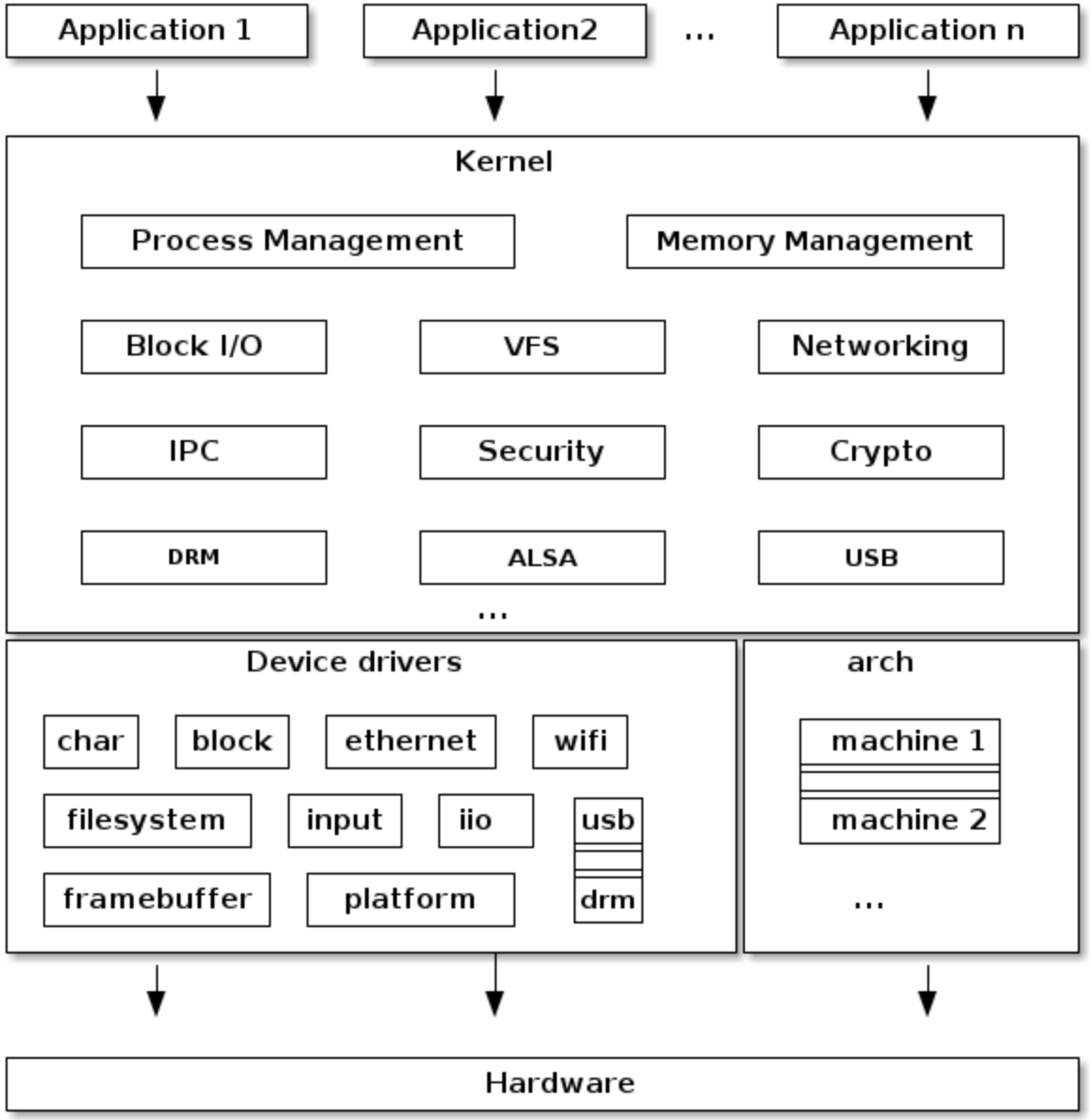
Device drivers

- Unified device model
- Each subsystem has its own specific driver interfaces
- Many device driver types (TTY, serial, SCSI, fileystem, ethernet, USB, framebuffer, input, sound, etc.)

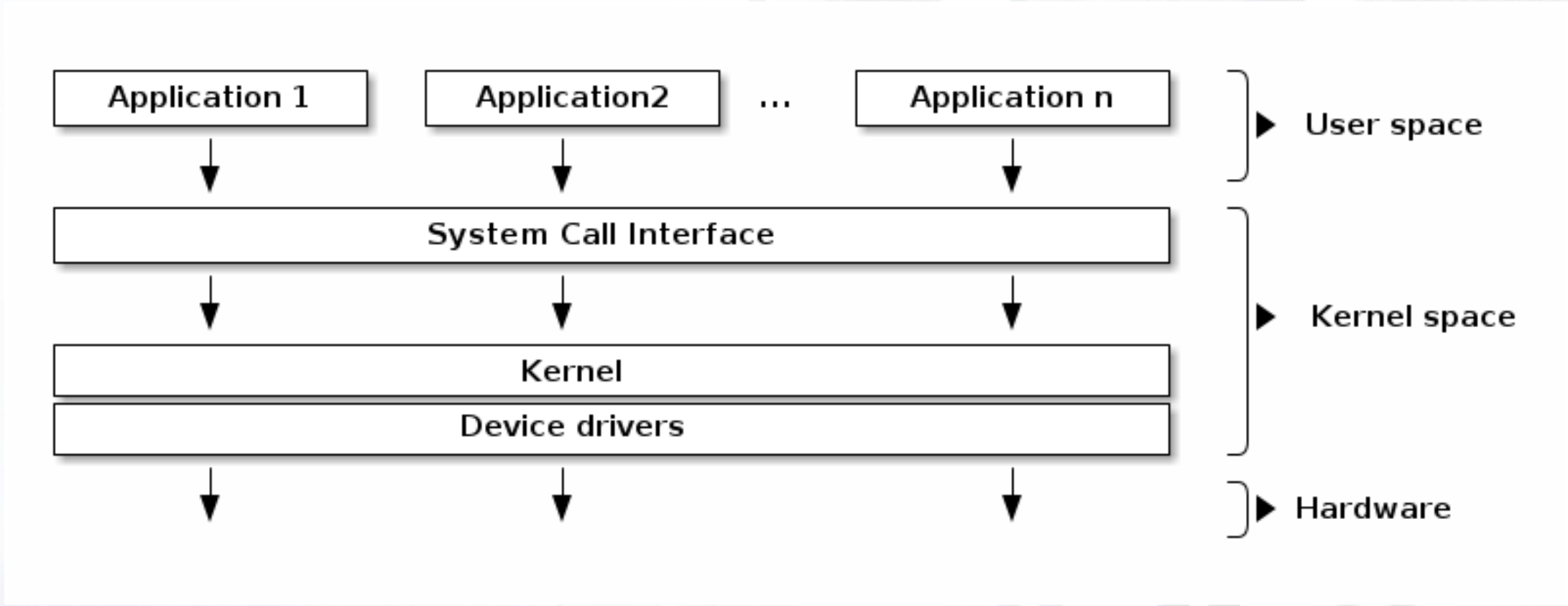


arch

- Architecture specific code
- Further sub-divided in machine specific code
- Interfacing with the boot loader and architecture specific initialization
- Access to various hardware bits that are architecture or machine specific such as interrupt controller, SMP controllers, BUS controllers, exceptions and interrupt setup, virtual memory handling
- Architecture optimized functions (e.g. memcpy, string operations, etc.)



General OS architecture



OS/Linux architecture: more details

- System calls are part of kernel APIs. They are the boundary where execution mode switches from user mode to kernel mode
- Kernel mode - to run code with higher privileges while user mode means running applications with lower privileges
- Kernel space is access protected, user applications can not access it directly, while user space can be directly accessed from kernel mode code
- Kernel and user share part of virtual space
- arch: architecture and machine specific code (C & AARCH64 ASM)