ITMO Advanced OS 2024

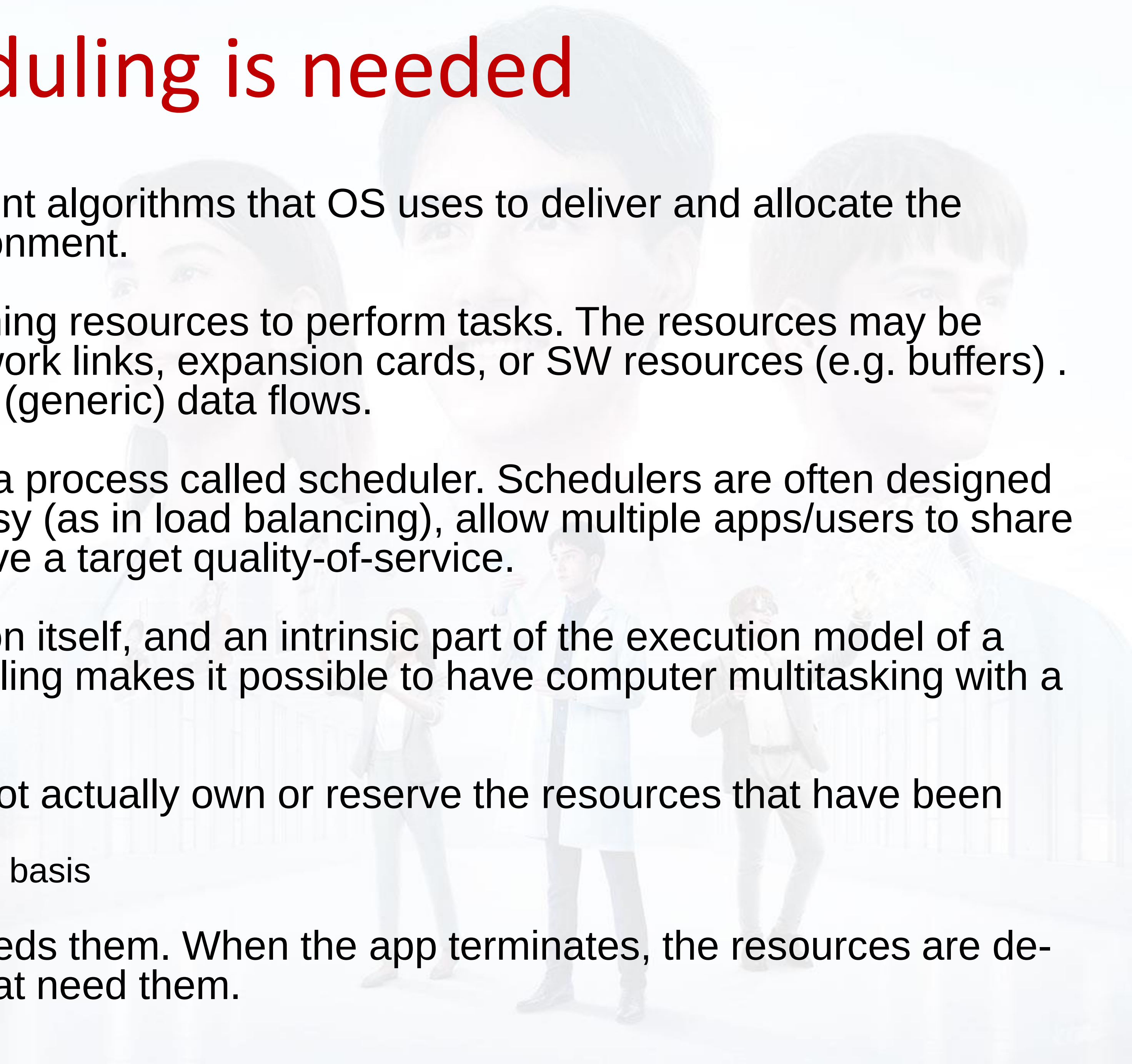
Aleksei Romanovskii, PhD,
SPb Research Center (CBG OS Lab)

Lesson 2024.11.27

Scheduling and resource management in OS

- CPU scheduling
- IO scheduling
- Network scheduling
- Memory scheduling
- Job scheduling (generic term, related to mainframes, historical)

Why resource scheduling is needed



- Resource scheduling refers to the different algorithms that OS uses to deliver and allocate the different resources in a computing environment.

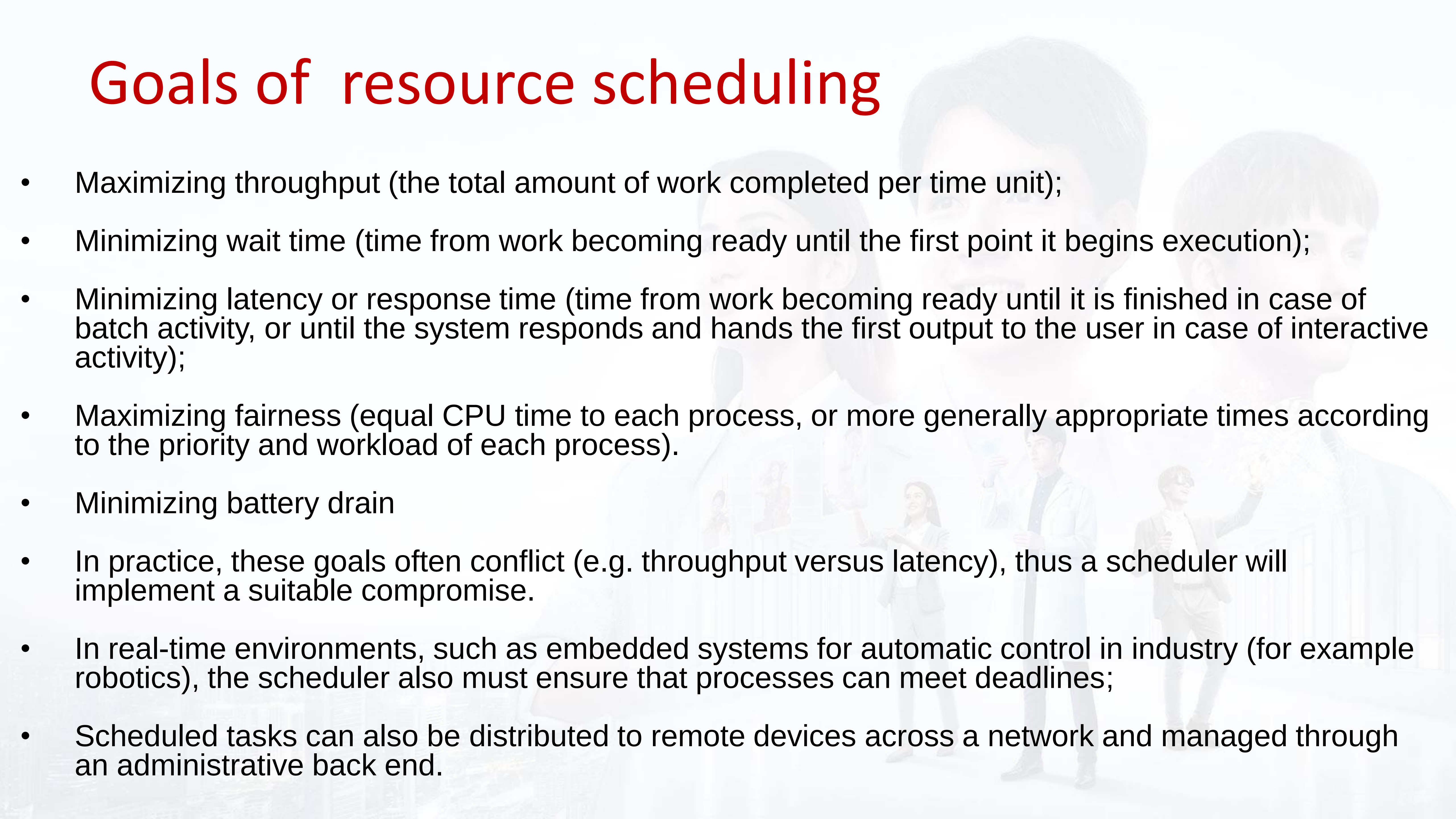
- Scheduling proper is the action of assigning resources to perform tasks. The resources may be processors, RAM, external storage, network links, expansion cards, or SW resources (e.g. buffers) . The tasks may be threads, processes or (generic) data flows.

- The scheduling activity is carried out by a process called scheduler. Schedulers are often designed so as to keep all computer resources busy (as in load balancing), allow multiple apps/users to share system resources effectively, or to achieve a target quality-of-service.

- Scheduling is fundamental to computation itself, and an intrinsic part of the execution model of a computer system; the concept of scheduling makes it possible to have computer multitasking with a single central processing unit (CPU).

- Resources are very limited so apps do not actually own or reserve the resources that have been allocated to them.
  - Resources are given to apps on temporary basis

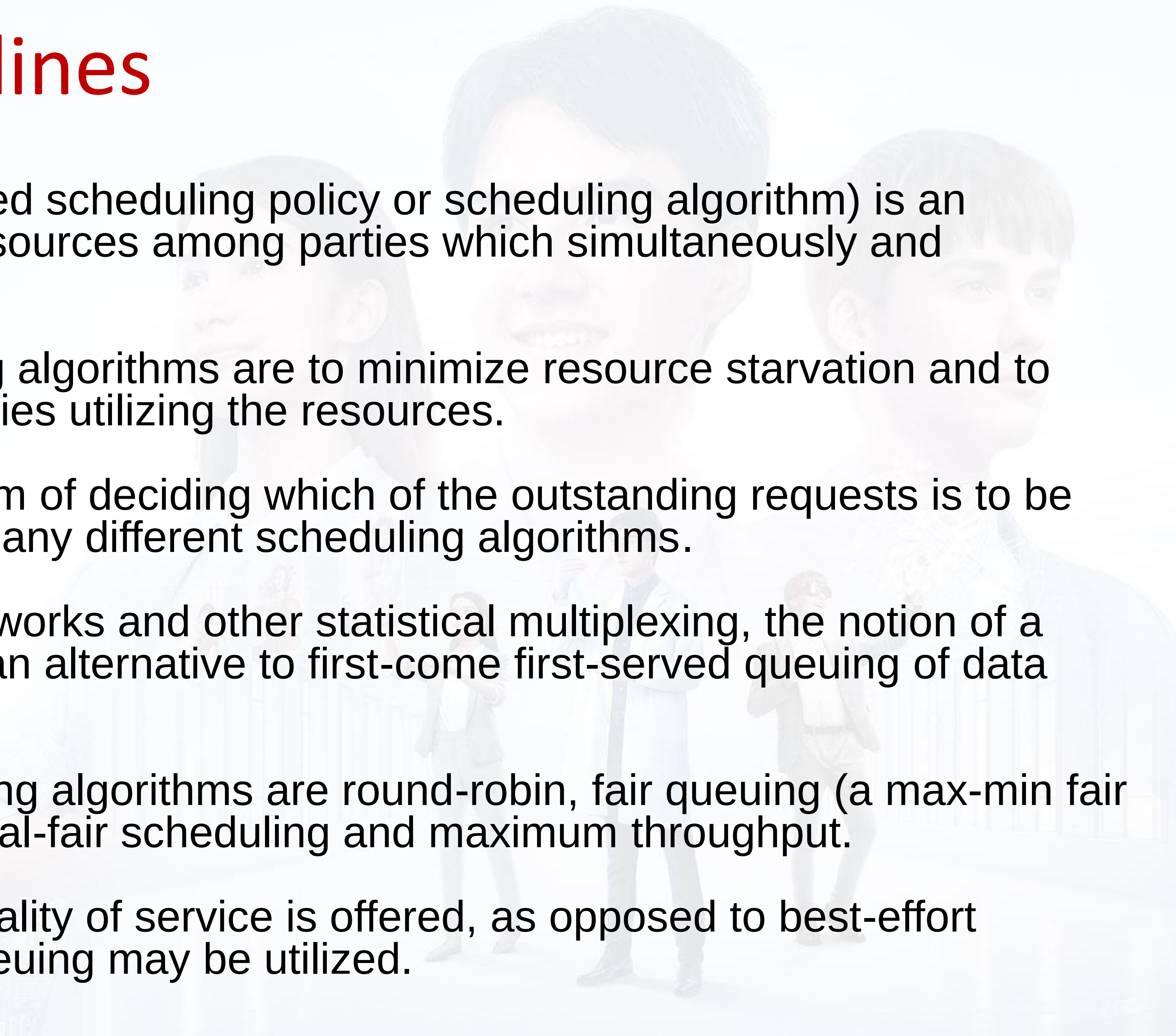
- OS allocates resources when an app needs them. When the app terminates, the resources are de-allocated, and allocated to other apps that need them.



Goals of resource scheduling

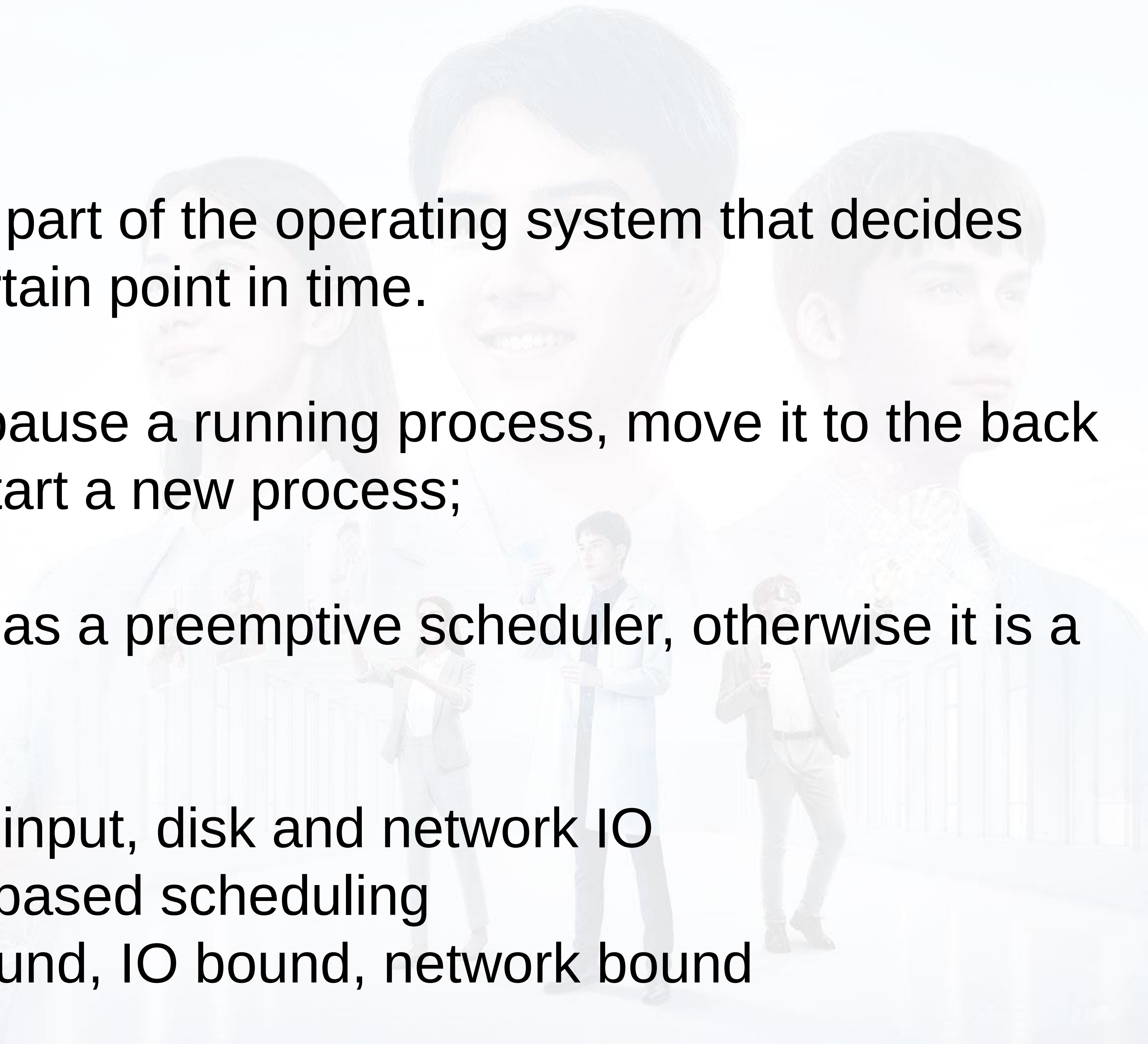
- Maximizing throughput (the total amount of work completed per time unit);
- Minimizing wait time (time from work becoming ready until the first point it begins execution);
- Minimizing latency or response time (time from work becoming ready until it is finished in case of batch activity, or until the system responds and hands the first output to the user in case of interactive activity);
- Maximizing fairness (equal CPU time to each process, or more generally appropriate times according to the priority and workload of each process).
- Minimizing battery drain
- In practice, these goals often conflict (e.g. throughput versus latency), thus a scheduler will implement a suitable compromise.
- In real-time environments, such as embedded systems for automatic control in industry (for example robotics), the scheduler also must ensure that processes can meet deadlines;
- Scheduled tasks can also be distributed to remote devices across a network and managed through an administrative back end.

Scheduling disciplines



- A scheduling discipline (also called scheduling policy or scheduling algorithm) is an algorithm used for distributing resources among parties which simultaneously and asynchronously request them.
- The main purposes of scheduling algorithms are to minimize resource starvation and to ensure fairness amongst the parties utilizing the resources.
- Scheduling deals with the problem of deciding which of the outstanding requests is to be allocated resources. There are many different scheduling algorithms.
- In packet-switched computer networks and other statistical multiplexing, the notion of a scheduling algorithm is used as an alternative to first-come first-served queuing of data packets.
- The simplest best-effort scheduling algorithms are round-robin, fair queuing (a max-min fair scheduling algorithm), proportional-fair scheduling and maximum throughput.
- If differentiated or guaranteed quality of service is offered, as opposed to best-effort communication, weighted fair queuing may be utilized.

Process scheduler



- The process scheduler is a part of the operating system that decides which process runs at a certain point in time.
-
- It usually has the ability to pause a running process, move it to the back of the running queue and start a new process;

- Such a scheduler is known as a preemptive scheduler, otherwise it is a cooperative scheduler.

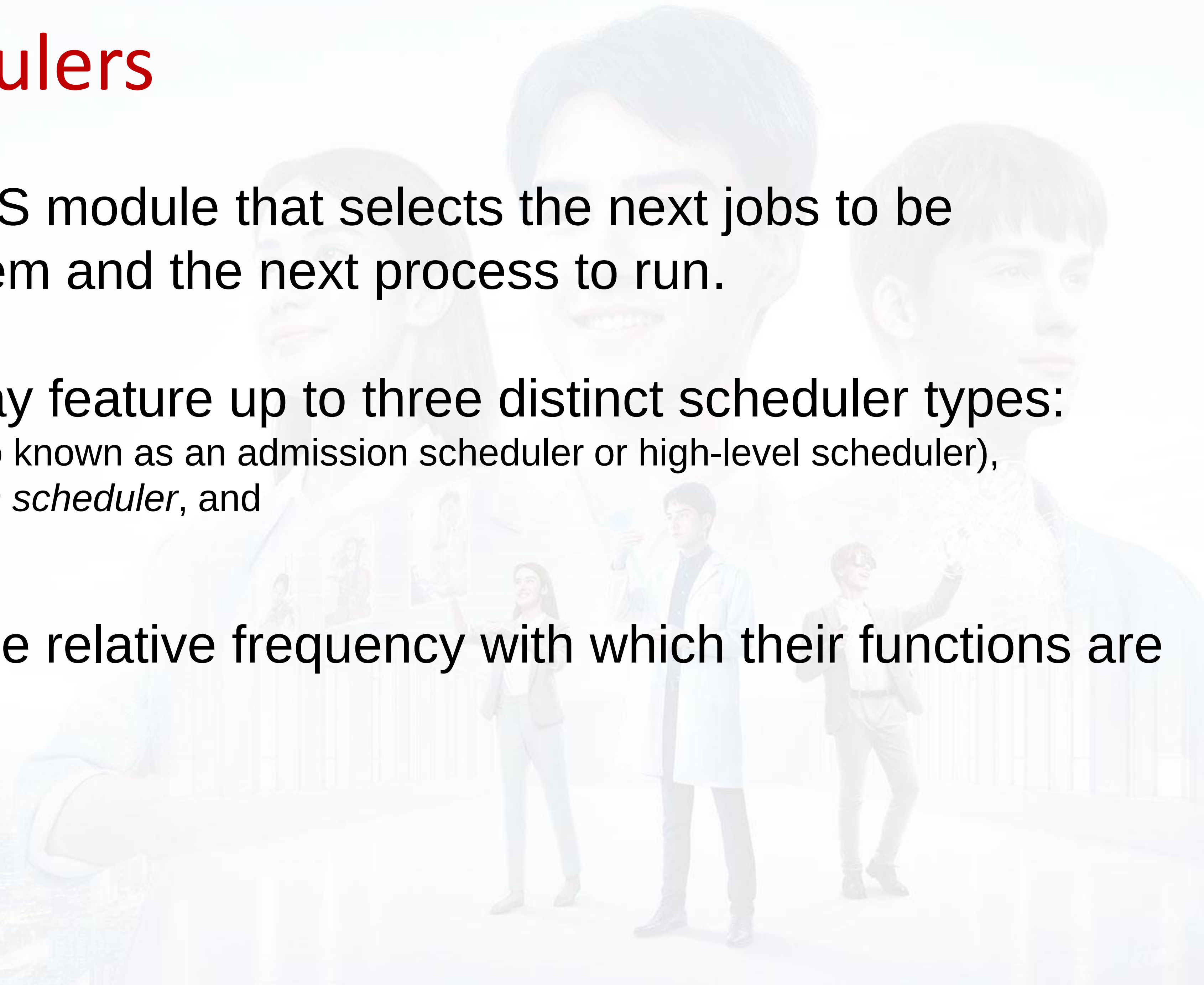
- Processes depend on user input, disk and network IO
  - statistics and prediction based scheduling
  - CPU bound, memory bound, IO bound, network bound

Types of schedulers

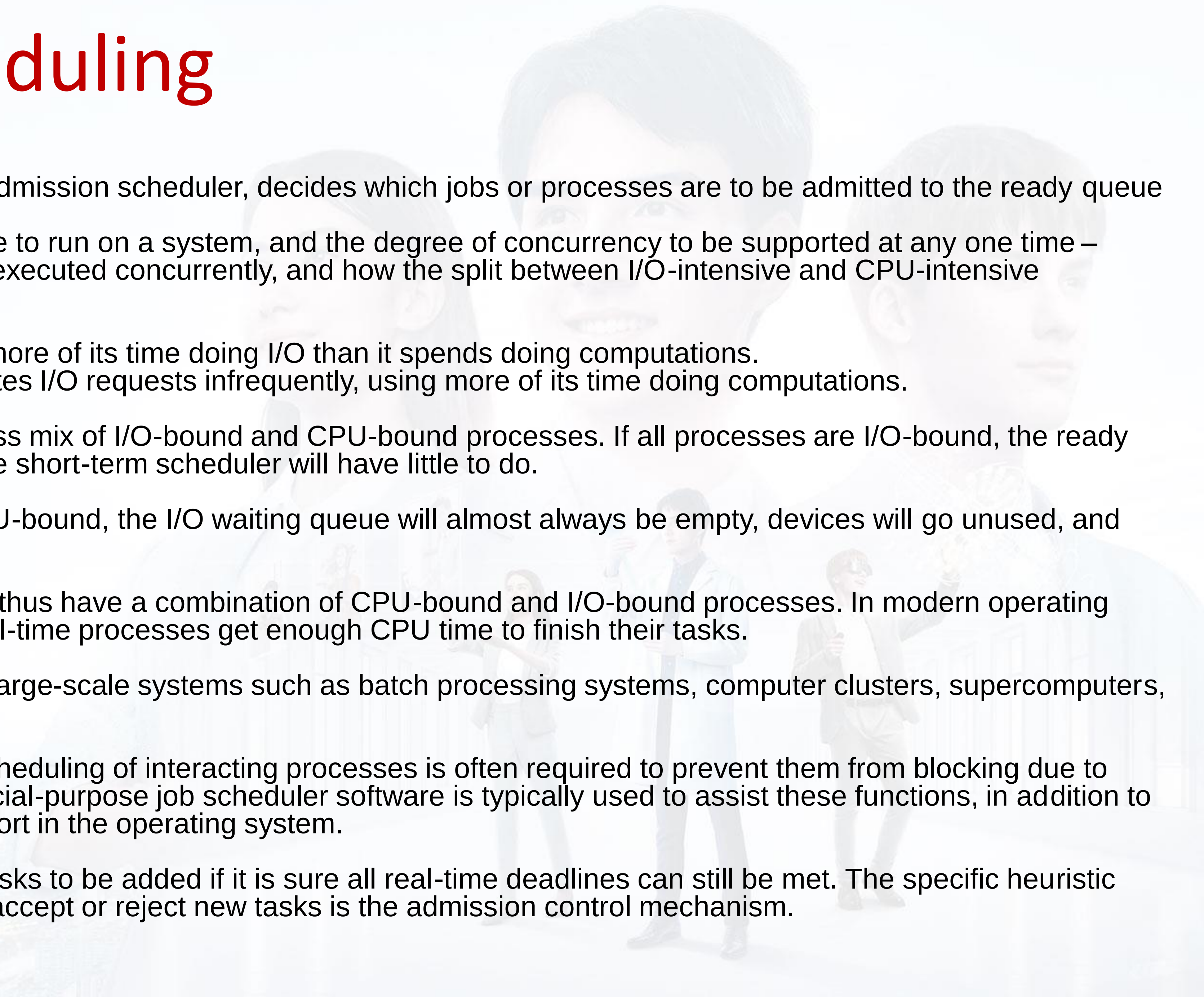
- The scheduler is an OS module that selects the next jobs to be admitted into the system and the next process to run.

- Operating systems may feature up to three distinct scheduler types:
  - a *long-term scheduler* (also known as an admission scheduler or high-level scheduler),
  - a *mid-term or medium-term scheduler*, and
  - a *short-term scheduler*.

- The names suggest the relative frequency with which their functions are performed.



Long-term scheduling



- The long-term scheduler (for servers), or admission scheduler, decides which jobs or processes are to be admitted to the ready queue

- This scheduler dictates what processes are to run on a system, and the degree of concurrency to be supported at any one time – whether many or few processes are to be executed concurrently, and how the split between I/O-intensive and CPU-intensive processes is to be handled.

- An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations.
- A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations.

- Long-term scheduler selects a good process mix of I/O-bound and CPU-bound processes. If all processes are I/O-bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do.

- On the other hand, if all processes are CPU-bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced.

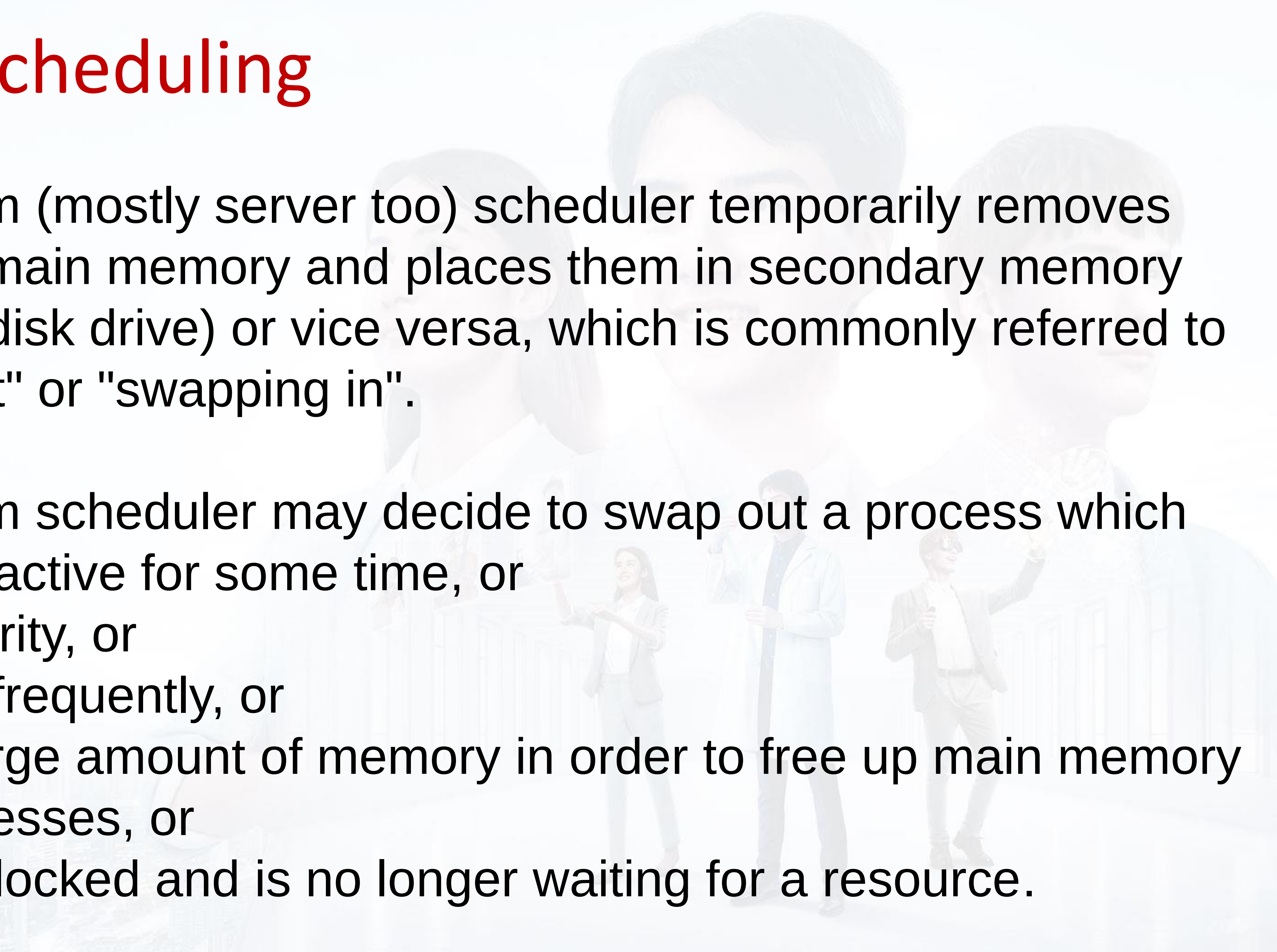
- The system with the best performance will thus have a combination of CPU-bound and I/O-bound processes. In modern operating systems, this is used to make sure that real-time processes get enough CPU time to finish their tasks.

- Long-term scheduling is also important in large-scale systems such as batch processing systems, computer clusters, supercomputers, and render farms.

- For example, in concurrent systems, co-scheduling of interacting processes is often required to prevent them from blocking due to waiting on each other. In these cases, special-purpose job scheduler software is typically used to assist these functions, in addition to any underlying admission scheduling support in the operating system.

- Some operating systems only allow new tasks to be added if it is sure all real-time deadlines can still be met. The specific heuristic algorithm used by an operating system to accept or reject new tasks is the admission control mechanism.

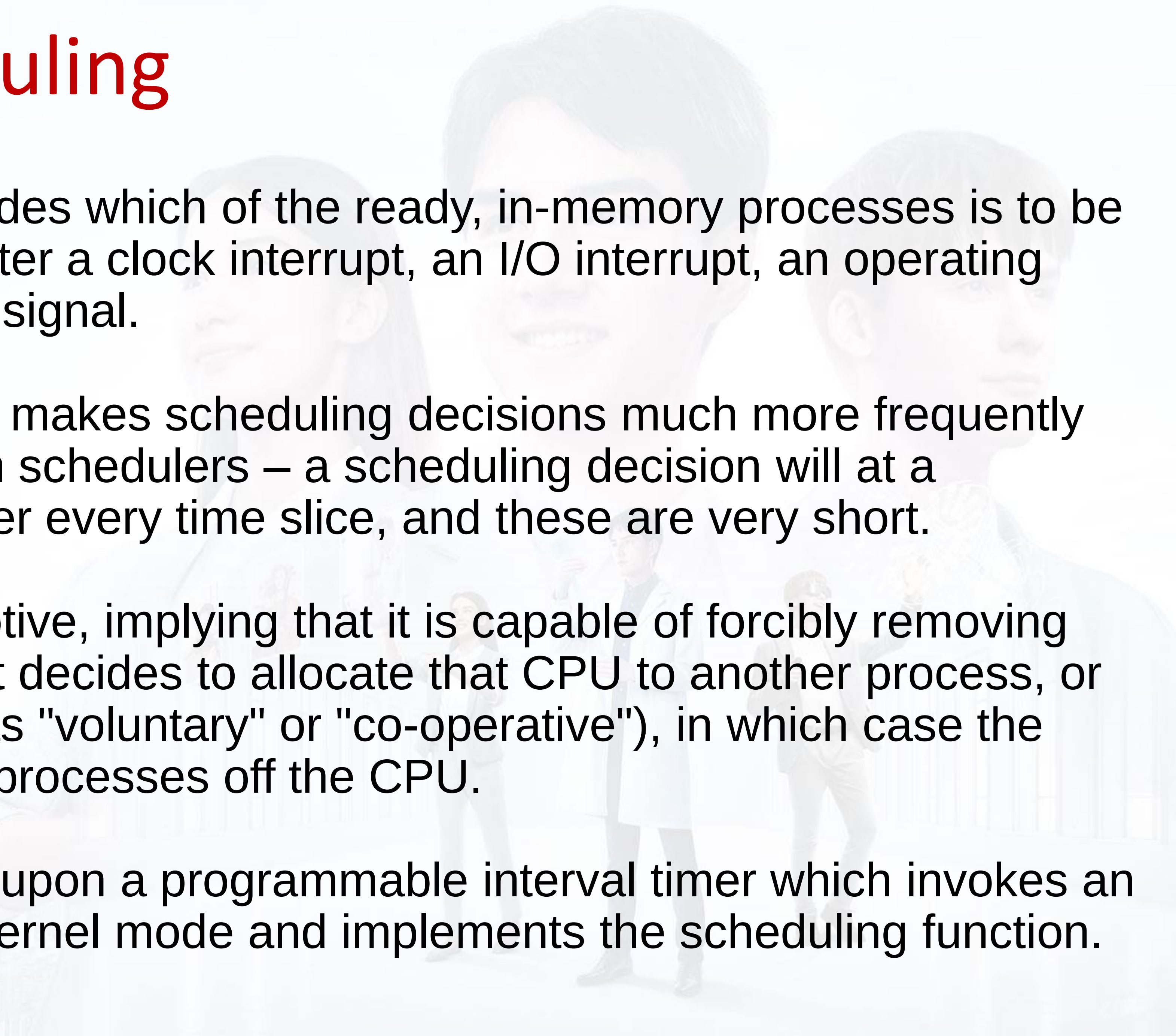
Mid-term scheduling



- The medium-term (mostly server too) scheduler temporarily removes processes from main memory and places them in secondary memory (such as a hard disk drive) or vice versa, which is commonly referred to as "swapping out" or "swapping in".

- The medium-term scheduler may decide to swap out a process which
  - has not been active for some time, or
  - has a low priority, or
  - page faulting frequently, or
  - taking up a large amount of memory in order to free up main memory for other processes, or
  - has been unblocked and is no longer waiting for a resource.

Short-term scheduling



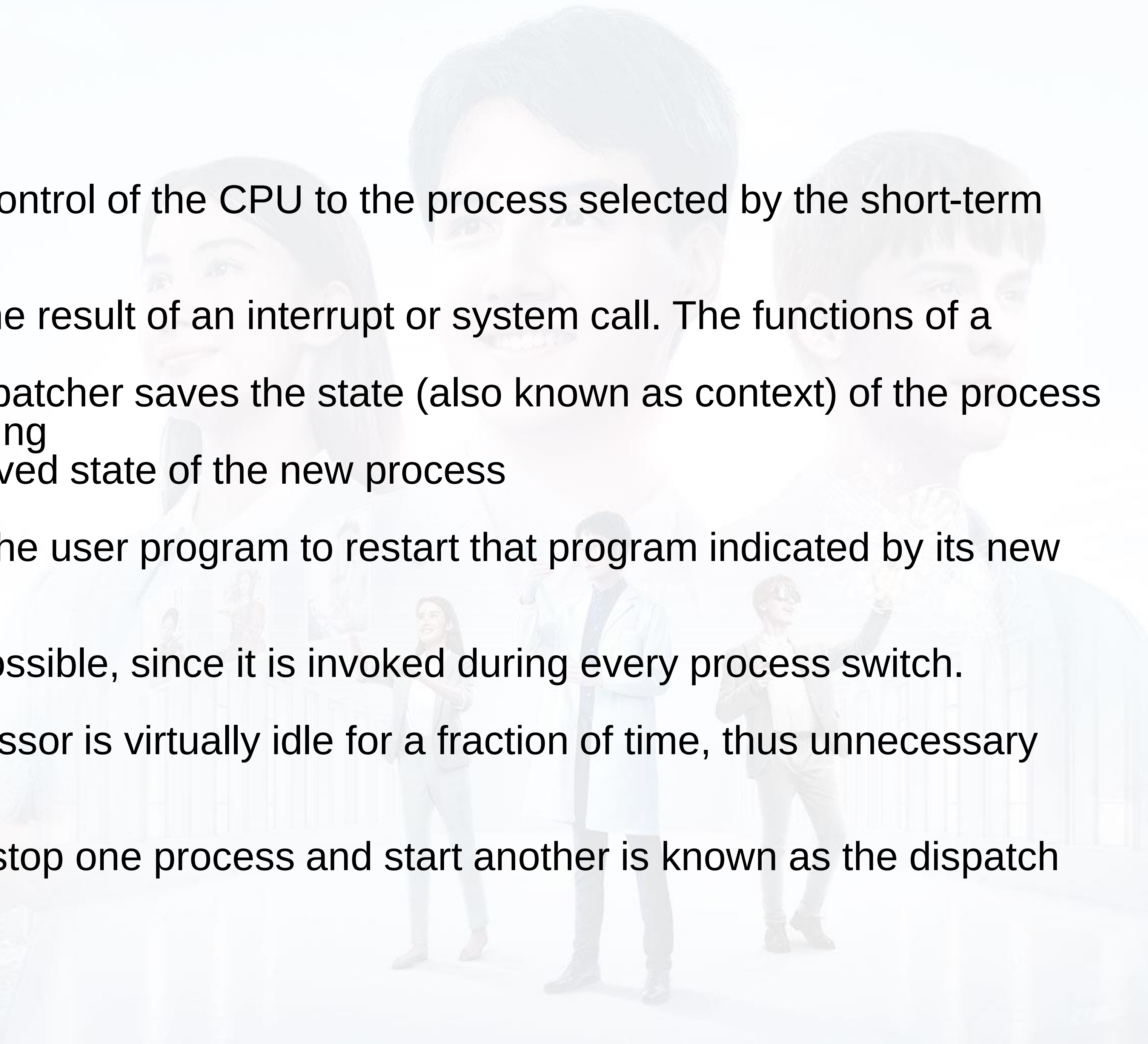
- The short-term scheduler decides which of the ready, in-memory processes is to be executed (allocated a CPU) after a clock interrupt, an I/O interrupt, an operating system call or another form of signal.

- Thus the short-term scheduler makes scheduling decisions much more frequently than the long-term or mid-term schedulers – a scheduling decision will at a minimum have to be made after every time slice, and these are very short.

- This scheduler can be preemptive, implying that it is capable of forcibly removing processes from a CPU when it decides to allocate that CPU to another process, or non-preemptive (also known as "voluntary" or "co-operative"), in which case the scheduler is unable to "force" processes off the CPU.

- A preemptive scheduler relies upon a programmable interval timer which invokes an interrupt handler that runs in kernel mode and implements the scheduling function.

Dispatcher



- Dispatcher is an algorithm that gives control of the CPU to the process selected by the short-term scheduler.

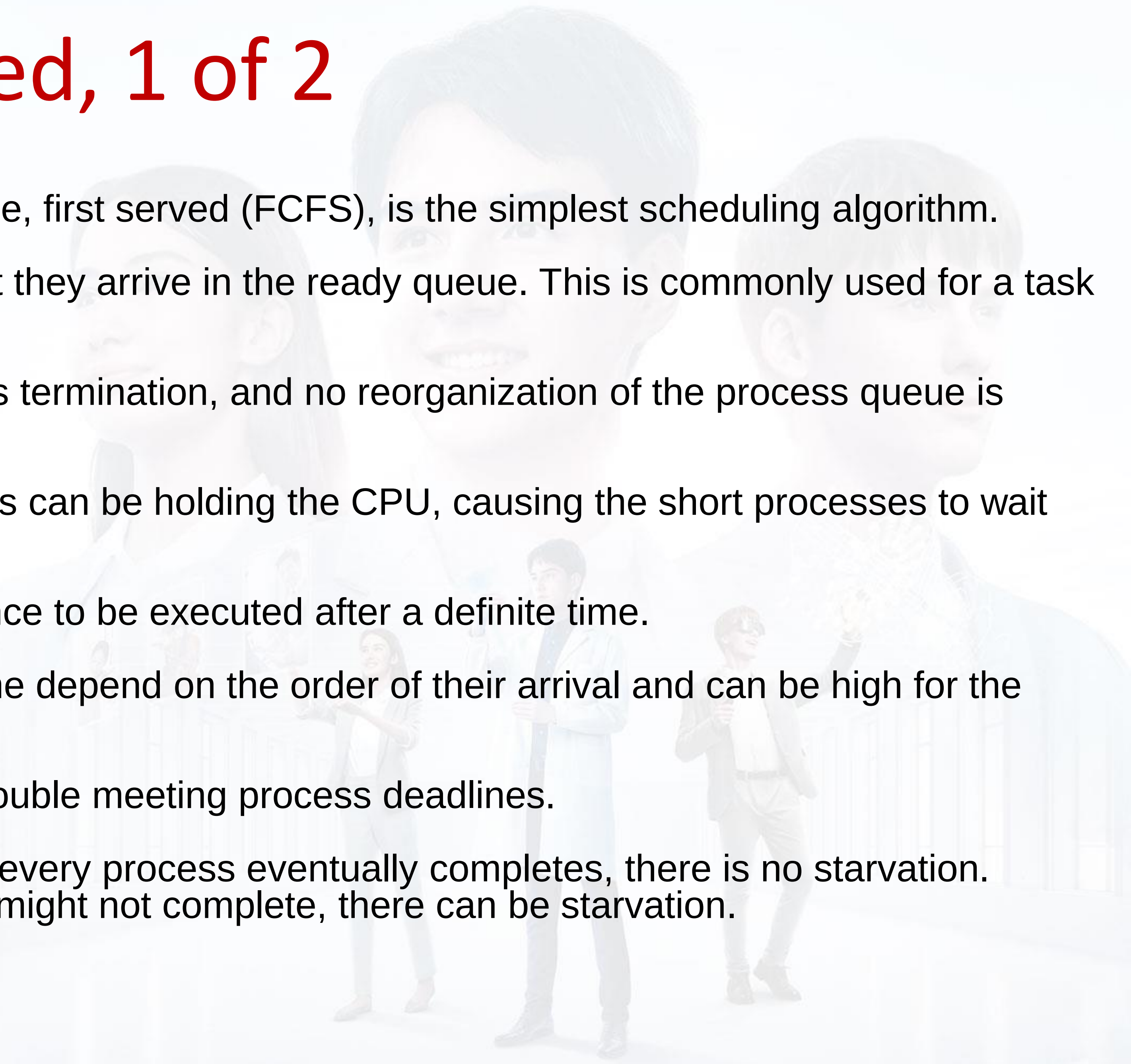
- It receives control in kernel mode as the result of an interrupt or system call. The functions of a dispatcher involve the following:
  - Context switches, in which the dispatcher saves the state (also known as context) of the process or thread that was previously running
  - Loading the initial or previously saved state of the new process
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program indicated by its new state

- The dispatcher should be as fast as possible, since it is invoked during every process switch.

- During the context switches, the processor is virtually idle for a fraction of time, thus unnecessary context switches should be avoided

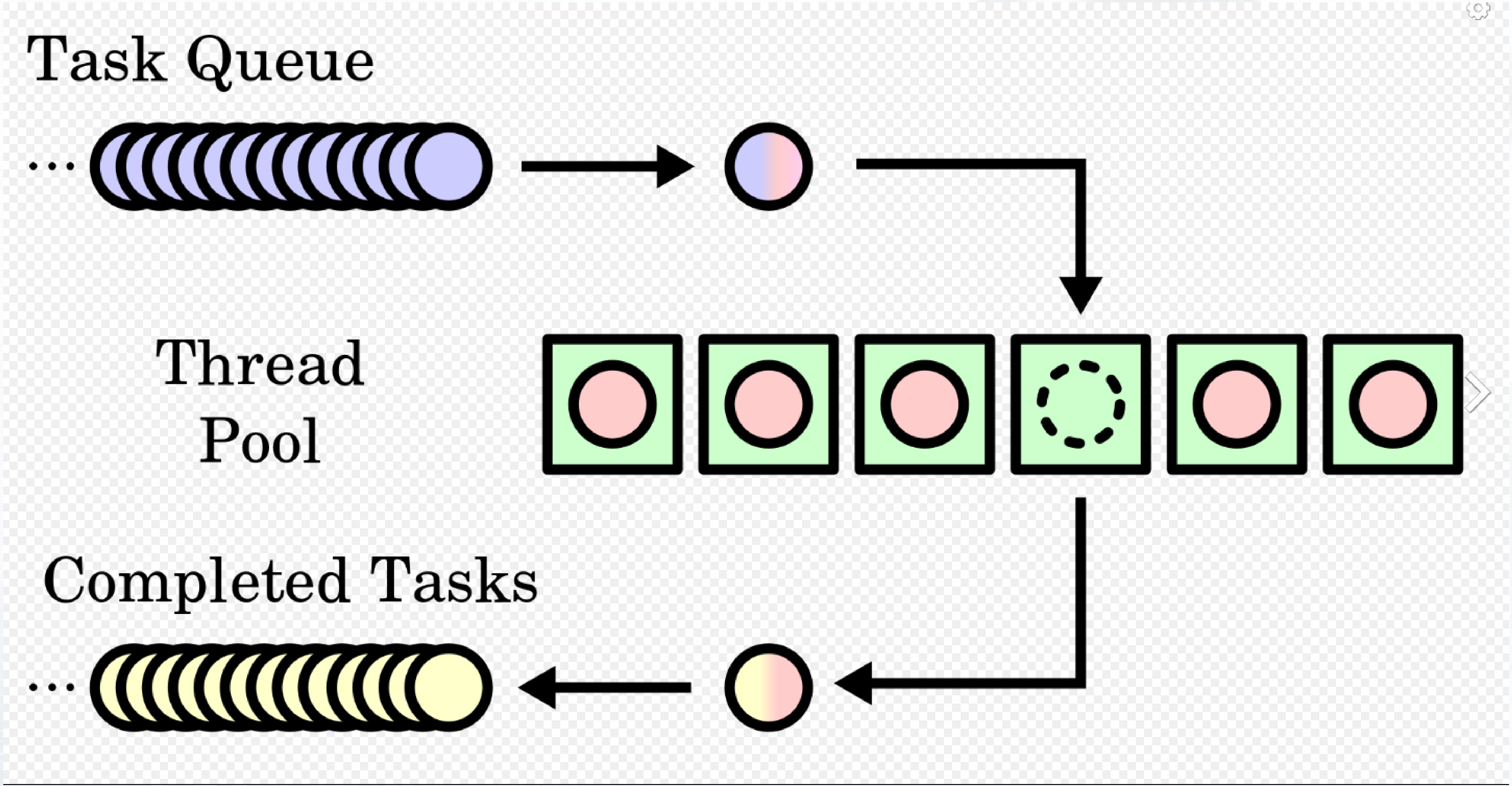
- The time it takes for the dispatcher to stop one process and start another is known as the dispatch latency

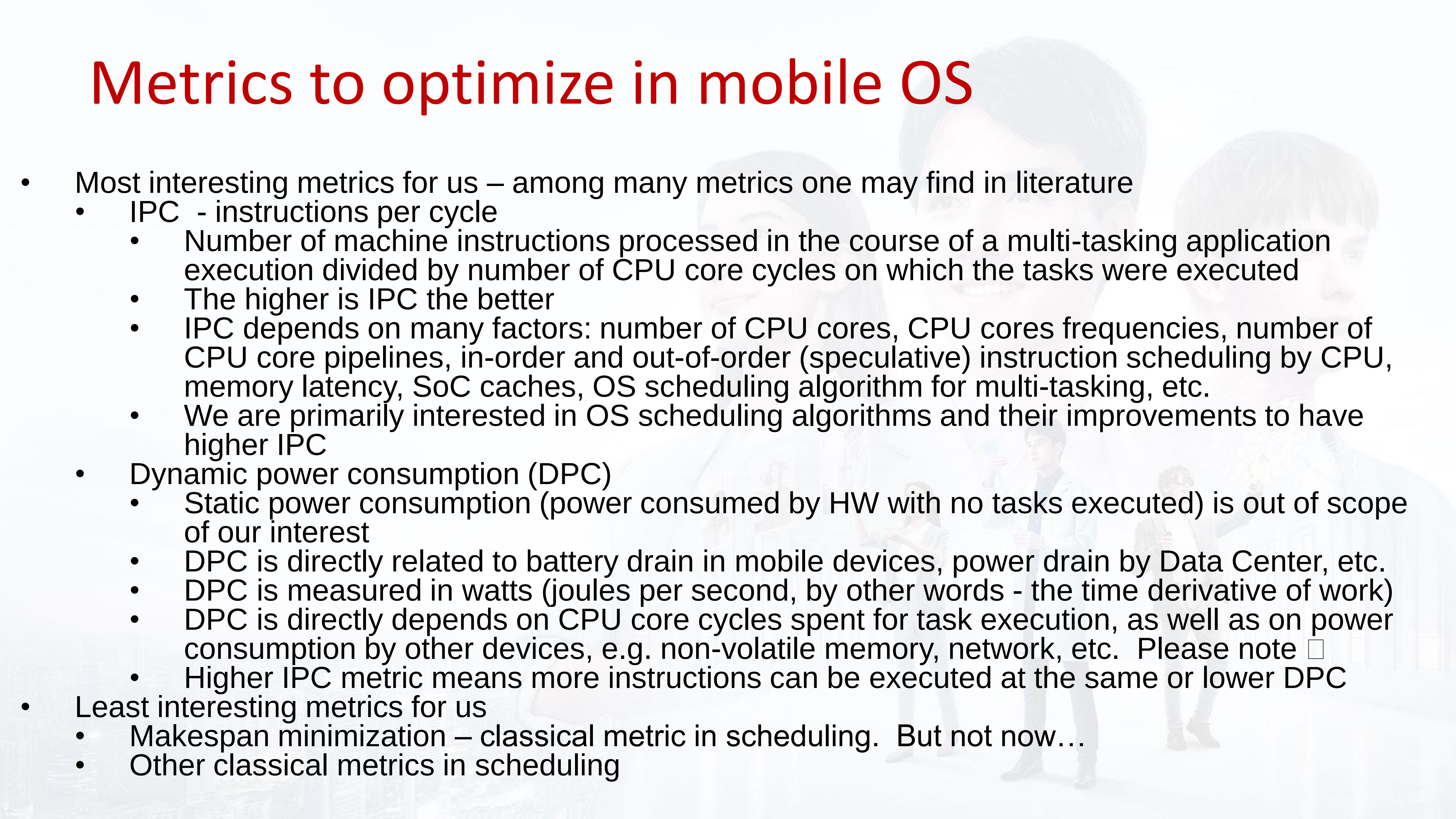
First come, first served, 1 of 2



- First in, first out (FIFO), also known as first come, first served (FCFS), is the simplest scheduling algorithm.
- FIFO simply queues processes in the order that they arrive in the ready queue. This is commonly used for a task queue (see next slide)
- Since context switches only occur upon process termination, and no reorganization of the process queue is required, scheduling overhead is minimal.
- Throughput can be low, because long processes can be holding the CPU, causing the short processes to wait for a long time (known as the convoy effect).
- No starvation, because each process gets chance to be executed after a definite time.
- Turnaround time, waiting time and response time depend on the order of their arrival and can be high for the same reasons above.
- No prioritization occurs, thus this system has trouble meeting process deadlines.
- The lack of prioritization means that as long as every process eventually completes, there is no starvation.
  - In an environment where some processes might not complete, there can be starvation.
- It is based on queuing.

First come, first served 2 of 2



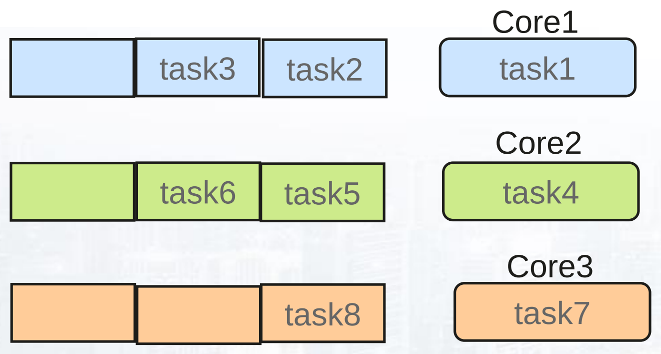


Metrics to optimize in mobile OS

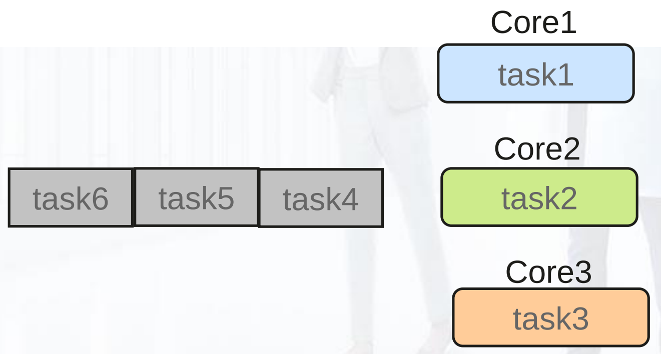
- Most interesting metrics for us – among many metrics one may find in literature
  - IPC - instructions per cycle
    - Number of machine instructions processed in the course of a multi-tasking application execution divided by number of CPU core cycles on which the tasks were executed
    - The higher is IPC the better
    - IPC depends on many factors: number of CPU cores, CPU cores frequencies, number of CPU core pipelines, in-order and out-of-order (speculative) instruction scheduling by CPU, memory latency, SoC caches, OS scheduling algorithm for multi-tasking, etc.
    - We are primarily interested in OS scheduling algorithms and their improvements to have higher IPC
  - Dynamic power consumption (DPC)
    - Static power consumption (power consumed by HW with no tasks executed) is out of scope of our interest
    - DPC is directly related to battery drain in mobile devices, power drain by Data Center, etc.
    - DPC is measured in watts (joules per second, by other words - the time derivative of work)
    - DPC is directly depends on CPU core cycles spent for task execution, as well as on power consumption by other devices, e.g. non-volatile memory, network, etc. Please note □
    - Higher IPC metric means more instructions can be executed at the same or lower DPC
- Least interesting metrics for us
  - Makespan minimization – classical metric in scheduling. But not now…
  - Other classical metrics in scheduling

Scheduling abstractions, 1 of 2

- Static and dynamic scheduling of applications (or task sets)
  - In static scheduling all application information is known and scheduling decisions are made at compile time (before execution of tasks). While it has low or no run-time overhead
  - In pure dynamic scheduling all scheduling decisions occur at run-time while the tasks are executed
- We are interested in dynamic scheduling that may use some historical (statistical) information about previous executions of applications
  - Fortunately in our use-cases repeating executions of same applications on the same device by the same user show a lot of correlations
- Preemptive and non-preemptive scheduling
  - We are interested in preemptive scheduling – when execution of an app is suspended (task is put in background) due to arrival of higher-priority app
  - One of our major concerns – what and how many apps to keep in background to have their fast resume. Historical information helps a lot.
- Hard real-time and soft (non-) real-time scheduling
  - Our case is non-real-time scheduling: deadline misses are OK while maintaining a certain throughput of the system
- Partitioned and global scheduling
  - Partitioned scheduling of cores with tasks clustering may be a possible direction of joint R&D



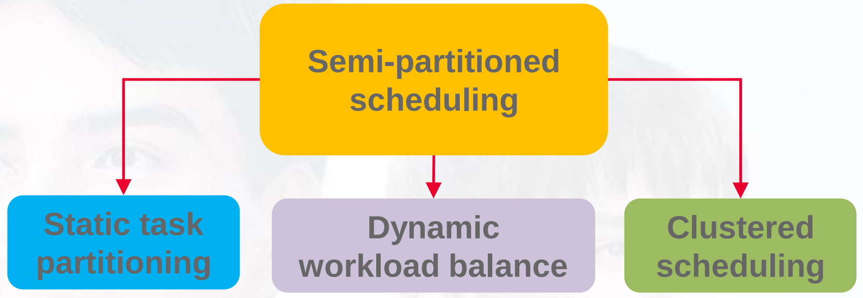
Partitioned scheduling

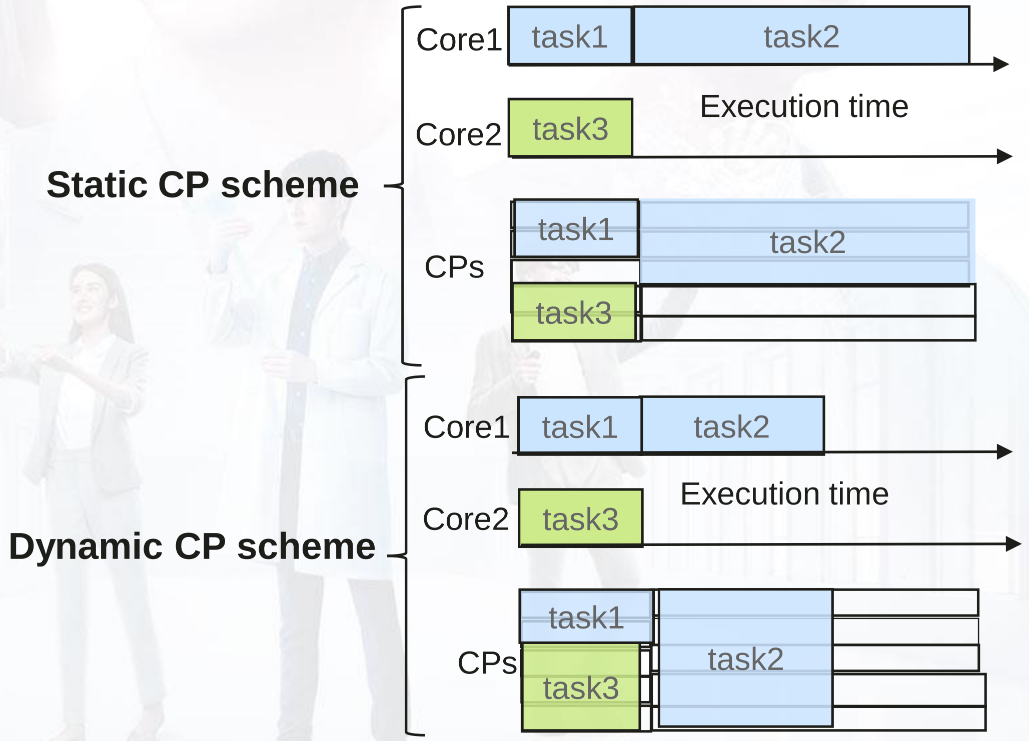


Global scheduling

Scheduling abstractions, 2 of 2

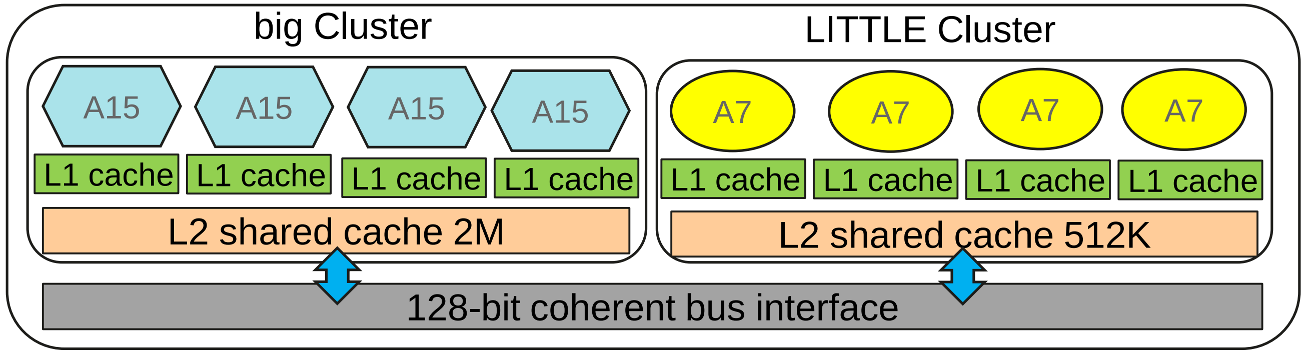
- Static task migrations – task-splitting is done by developers (before execution):
  - some tasks are allocated to only one core while the others are allocated to more than one core
  - parallel task models to break up the task into different components in order to reduce the execution time of the tasks (e.g. Grand Central Dispatcher)
- Dynamic workload balance – run-time migration of tasks
  - For semi-partitioned scheduling is done inside one partition of cores (cores are not necessary from the same cluster)
  - the migrating tasks may use the same data in LLC
- Clustered scheduling
  - The cores are divided in clusters
  - The taskset is partitioned across the clusters
  - Each cluster schedules its tasks independently
- Static and dynamic Cache Partitioning (CP)
  - Several unsuccessful attempts were done to apply dynamic CP to Linux kernel
  - There are a few publications on dynamic cache partitioning in RT systems though.





Device abstraction

- Clustered Heterogeneous Multicore Architecture (HMA)
  - O is set of |O| cores, indexed as o[1..|O|] : O = {o[1],…o[|O|]}
  - H is set of |H| core types, indexed as h[1..|H|]
  - K is set of |K| clusters, each comprises cores of the same core-type, indexed as k[1..|K|]
- Each core-type cluster operates on independent frequency, while cores in the same cluster operates in the same frequency
  - Frequency range for core-type is defined by a finite set F of |F| frequencies, indexed as f[1..|F|]
- Each cluster has an independent Last Level Cache (LLC) that is shared by all cores in the cluster
  - LLC can be partitioned into set A of |A| cache partitions, indexed as a[1..|A|]
  - Core based partitioning: each core in a cluster can be assigned a number of distinct a[i]
  - Alternative task based partitioning: a number of distinct a[i] can be assigned to each task (see Task model)
  - Set W of |W| cache partitions assigned to each core (task), indexed as w[1..|W|]
- SoC example in terms of the specified model:
  - |O| = 8 cores,
  - |H| = 2 (ARM big.LITTLE),
  - |K| = 2
  - Fbig = {0.2, 0.3, …2.0}GHz, Flittle = {0.2, …, 1.4}GHz
  - LLCbig = 2048kB, LLClittle = 512kB. |A| = 8



- RAM, disk, network (together with CPU cores - can be generalized as resources of different types)
  - MB is available RAM bandwidth, ML is RAM latency, MS is RAM size
  - DB is available disk bandwidth, DL is disk latency, DS is disk size
  - NB is available network bandwidth, NL is network latency (min, average, max)

Task abstraction, 1 of 2

- Open model of non-periodic online tasks
  - new tasks can be added at any time, old tasks deleted at any time
  - each task can be run at arbitrary time
  - each task may have requirements for responsiveness on external events (e.g. on UI)
  - Set T of |T| tasks, indexed as t[1..|T|]: T = {t[1],…t[|T|]}
- Each task t[i] is represented by <r[i], e[i], d[i], p[i], uc[i], um[i], ud[i], un[i]>, where
  - r[i] is release (earliest start) time of task t[i]
  - e[i] is Worst Case Execution Time (WCET) of task t[i]
  - d[i] is deadline (latest end) time of task t[i]
  - p[i] is task t[i] priority (may be dynamic)
  - uc[i] is task average CPU utilization (usually vector for HMA)
  - um[i] is task average memory bandwidth utilization
  - ud[i] is task average disk utilization
  - un[i] is task average network utilization
- Cores (core clusters) are scheduled/partitioned to tasks (or vice-versa). Caches are scheduled/partitioned to tasks
- RAM, disks(IO), network are scheduled/partitioned to tasks

Task abstraction, 2 of 2

- Task t[i] utilization uc[i] = e[i]/(d[i] – r[i]), uc[i] <= 1.
  - To ensure valid schedule the sum of all uc[i] allocated to each core (for a given period of time) must be <= 1

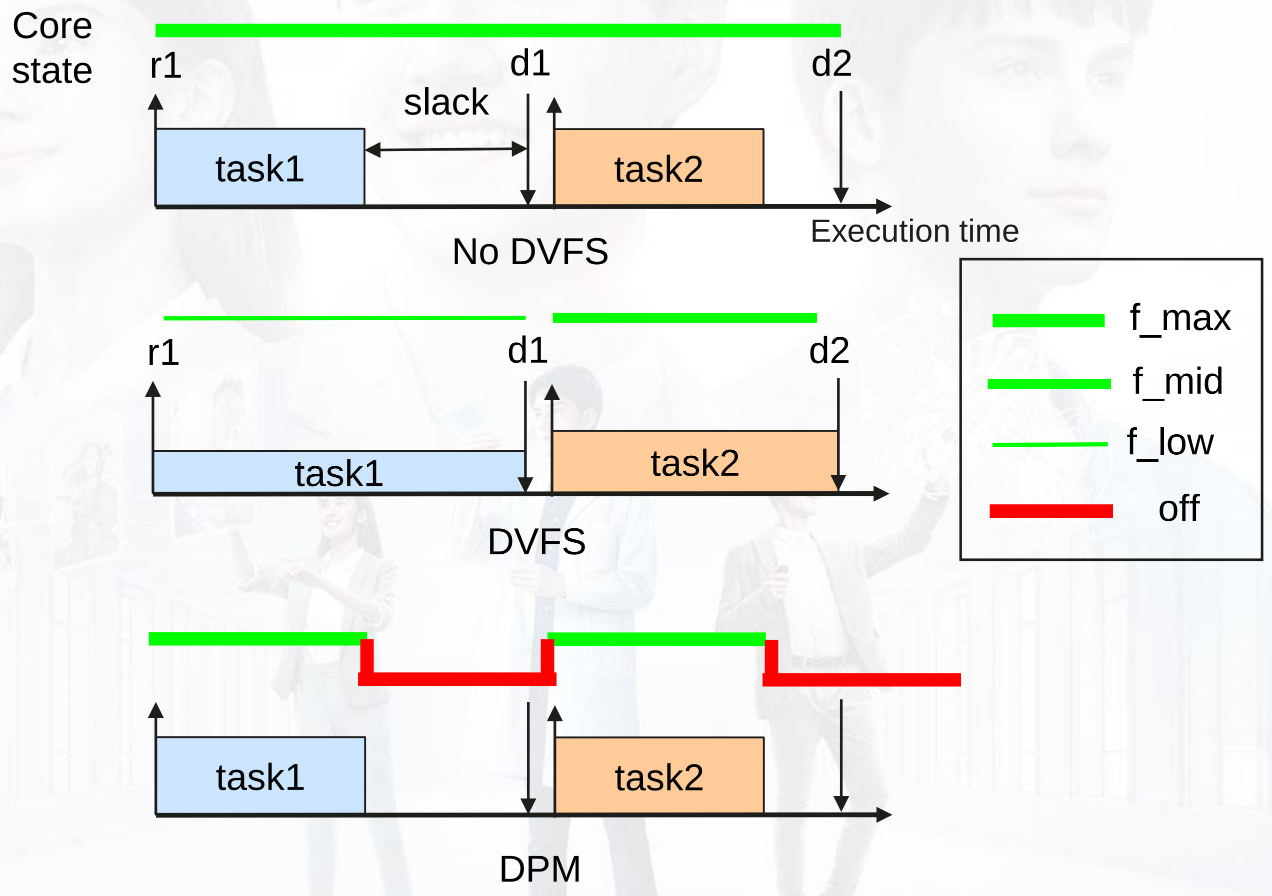
- Task t[i] execution cycles c[i] = cc[i] + mc_cache[i] + mc_ram[i] + dc[i] + nc[i], where
  - cc[i] are compute (CPU core) cycles, used by the task t[i]
  - mc_cache[i] are data access delays, measured in compute cycles, when data is in LLC
  - mc_ram[i] are data access delays, measured in compute cycles, when data is in RAM (either to read in LLC, or to write-through)
  - dc[i] are disk delays for data IO, measured in compute cycles
  - nc[i] are network delays for data IO, measured in compute cycles

- Task execution time e[i] expressed via CPU core frequency f[j], cache frequency f_cache and RAM frequency f_ram: e[i] = cc[i]/f[j] + mc_cache[i]/f_cache + mc_ram[i]/f_ram

- Task metrics that may be used by scheduling algorithm – to have higher IPC
  - compute intensity metric cim[i] = cc[i]/(cc_max[i] + mc_cache_max[i] + mc_ram_max[i] + dc_max[i] + nc_max[i]), where
    - cc_max[i] are compute cycles at max core frequency
    - mc_cache_max[i] are memory cycles at max cache frequency and max |W| of cache partitions allocated to task
    - mc_ram_max[i] max RAM frequency
    - etc.
  - …other metrics that will facilitate task allocation strategy to determine the best core, core-type, RAM amount, IO time slots, network time slots for each task – ultimately to have higher IPC

Power management abstractions, 1 of 2

- Dynamic Voltage and Frequency Scaling (DVFS)
    - clock-gating technique that varies the voltage and frequency of processor cores between minimum and maximum bounds
    - the processor core frequency has direct relation to voltage. Decrease in voltage (and frequency) result in reduction of DPC.
    - Frequency and voltage change at discrete steps. Usually done by programmable voltage regulator and clock generator
    - Inter-task DVFS and intra-task DVFS
- Dynamic Power Management (DPM) to manage power in core idle state - not in our area of interest



Power management abstractions, 2 of 2

- Dynamic Voltage and Frequency Scaling (DVFS), continued
  - Task execution time depends both on core cycle and on memory cycle latency, so it may happen if DVFS increases CPU core clock-cycle length then less memory cycles would suffice, therefore higher IPC could be achieved, see Fig.1
  - In turn Fig.2 illustrates dependency of task Dynamic Power Consumption on DVFS: lower frequency results in lower DPC

Fig.1

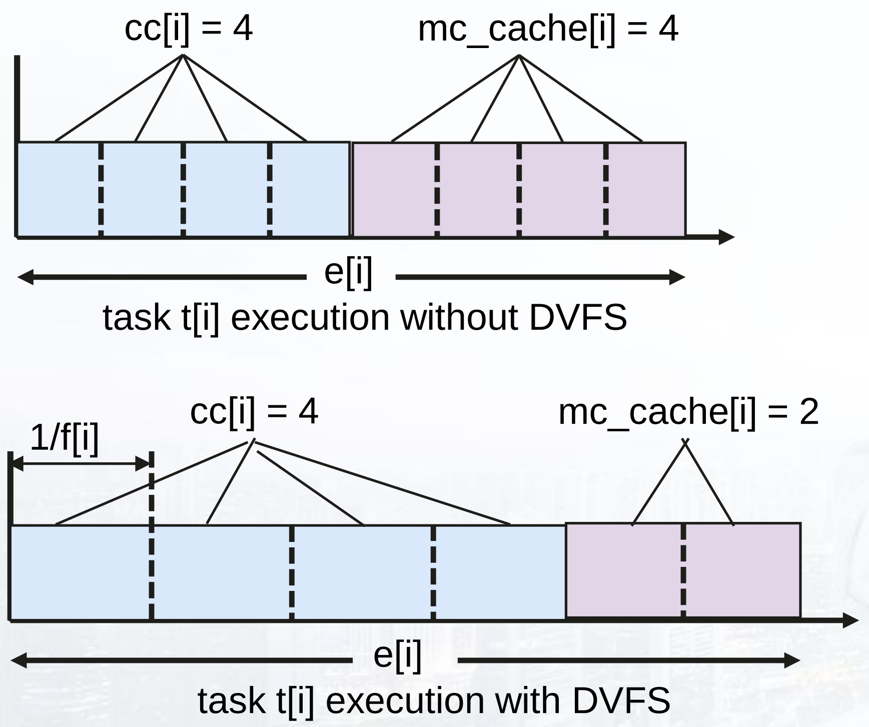
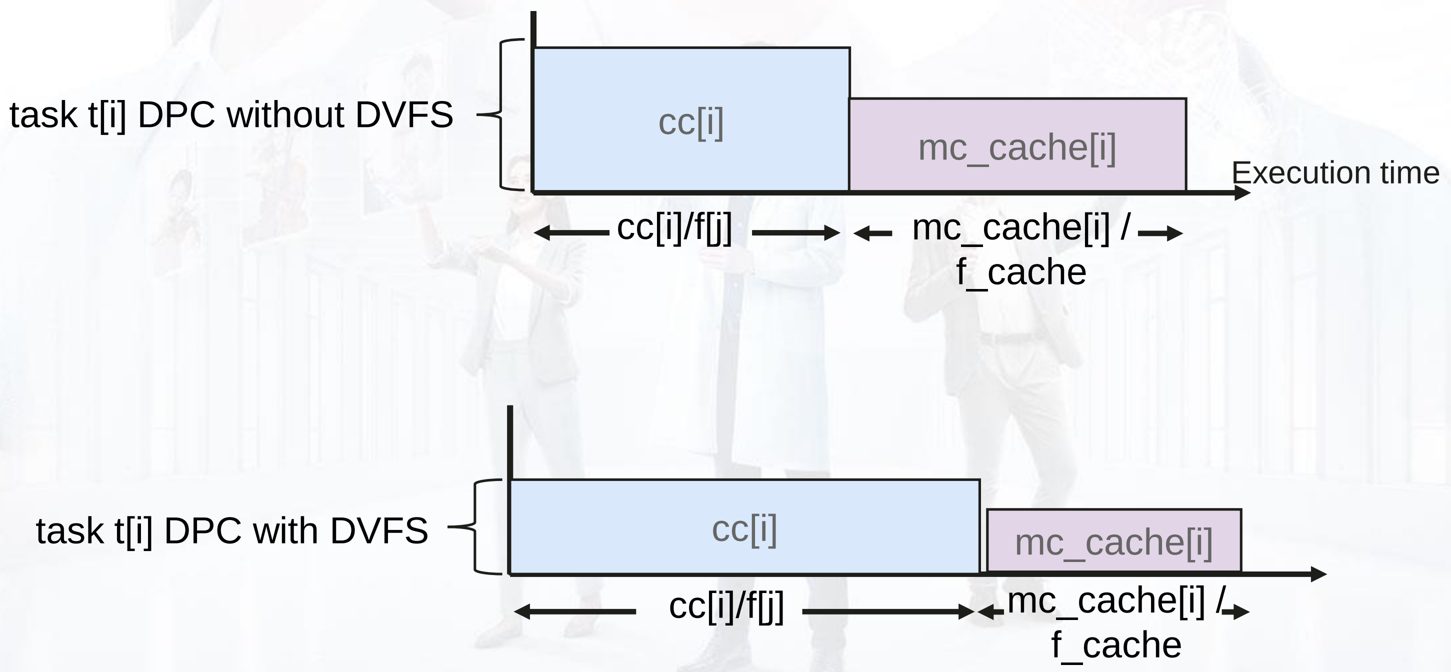


Fig.2



task t[i] DPC without DVFS
cc[i]
mc_cache[i]
Execution time
cc[i]/f[j]
mc_cache[i] / f_cache
task t[i] DPC with DVFS
cc[i]
mc_cache[i]
cc[i]/f[j]
mc_cache[i] / f_cache