Operating Systems Internals – Task scheduling

Hussein Al-Rikabi
https://www.mit.bme.hu/eng/general/phd/rhussein

rhussein@mit.bme.hu

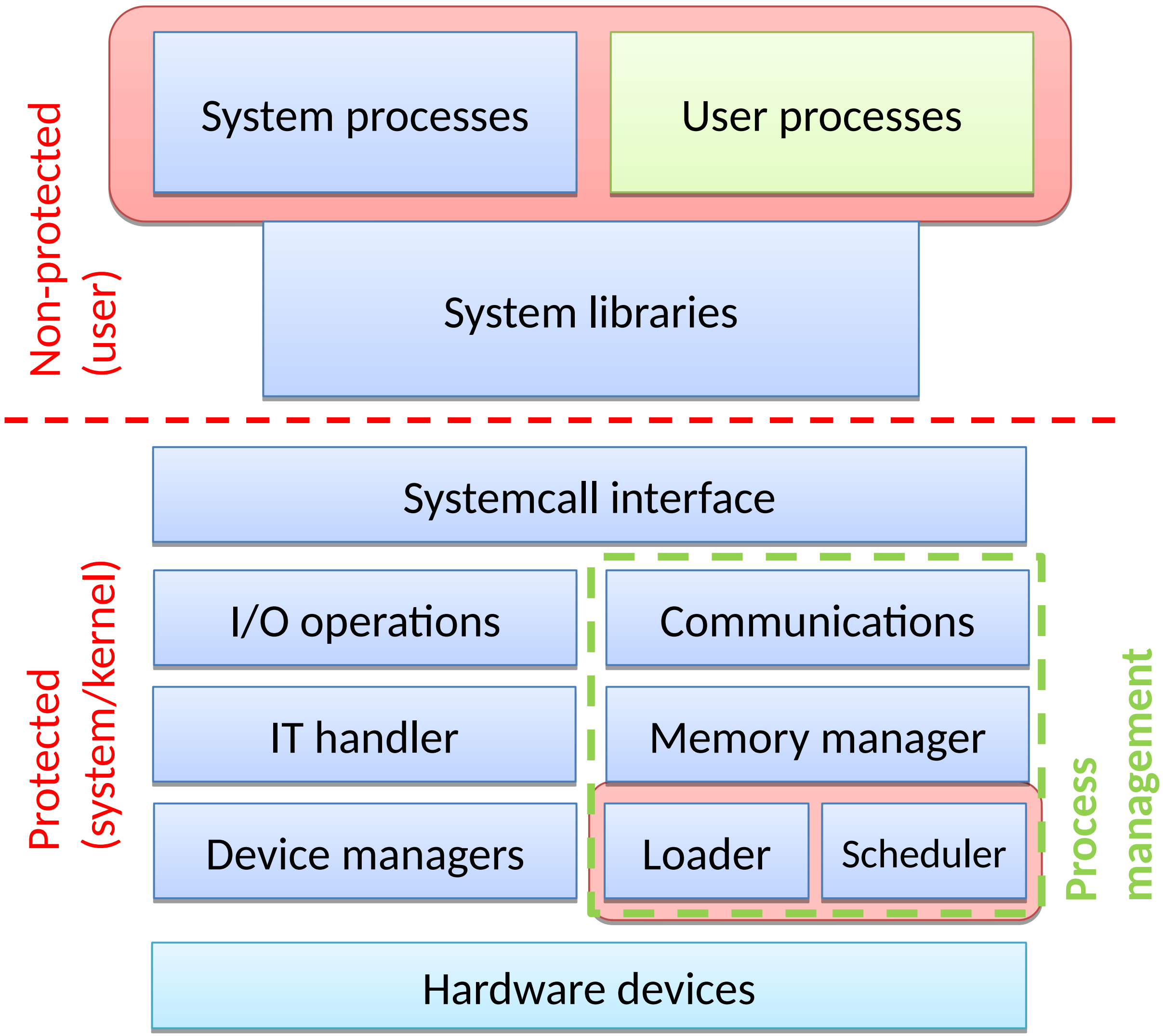
Budapest University of Technology and Economics (BME)
Department of Measurement and Information Systems (MIT)

The slides of the latest lecture will be on the course page. (https://www.mit.bme.hu/eng/oktatas/targyak/vimiab00)

The basics of task scheduling (recap)

- The scheduler chooses the next task to run
  - Short term (we learned about this), medium and long term
  - Basic properties
    - Data structure
    - Considered task properties
    - Decision algorithm
    - Complexity and overhead
- Simple schedulers
  - FCFS: simple, but it may perform badly (Convoy Effect)
  - RR: it is widely used, good response time, moderate overhead
  - SJF and SRTF: decision based on the task’s CPU burst, optimal waiting time
  - Priority: importance shown by a number
- Complex schedulers
  - Multilevel queues
    - It can use multiple algorithms (which is suited for the tasks)

The main blocks of the OS and the kernel (recap)



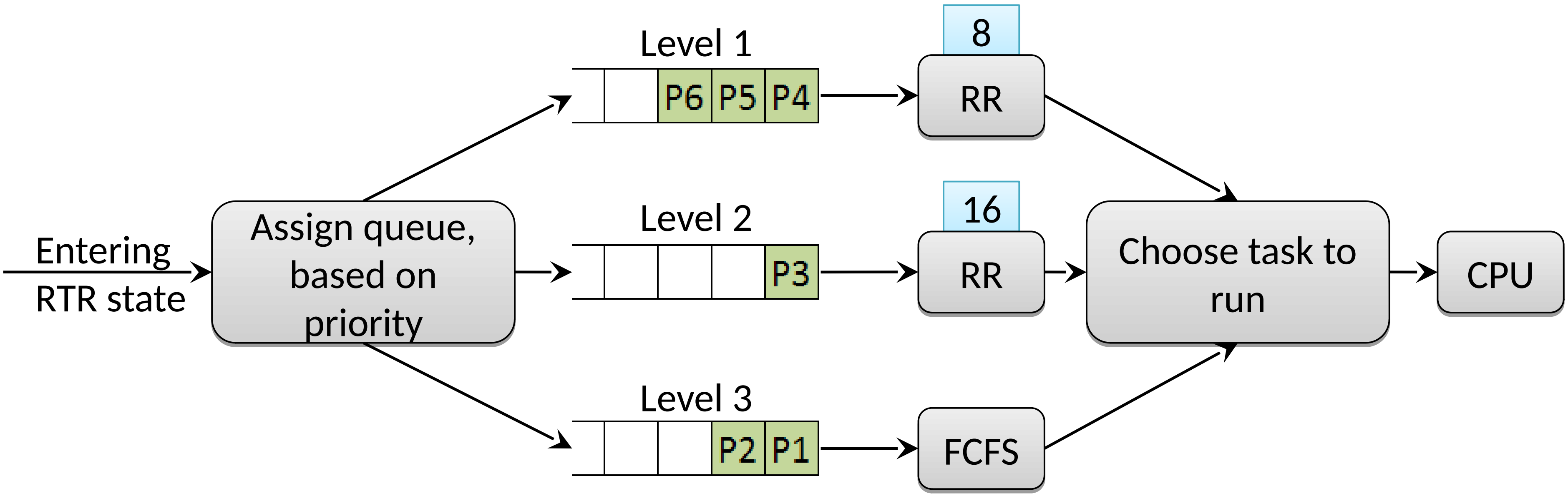
Multilevel scheduling

- Problems with the previous schedulers
  - The description capability of the priority is constrained
  - Not much information can be „crammed” in one number
  - The expectations for tasks can be different, one scheduler cannot fulfill all of them
  - The different schedulers can be optimal for different types of tasks
- Solution: Multilevel scheduling
  - If tasks can be categorized, they can be ordered in different queues. Every queue can have it’s own scheduling algorithm, which is the most appropriate for the tasks in the queue.
- The scheduling queues should also be scheduled
  - Which queue we choose the next task from?
  - Every queue may have a time slice (RR)
    - The more important level may have a longer time slice
  - Priorities can be assigned to the scheduling queues
    - Starvation may appear
  - Starvation can be avoided if the tasks are allowed to change the current scheduling queue
    - More complex: an algorithm is needed for stepping up and down the tasks

Static multilevel queues

- The tasks are assigned to a queue in a static way
  - There's no changing between queues (static priority)
  - The assignment is based on the priorities of the task
  - The priority stays the same till the completion of the task
- The different queues are defined by the nature of the tasks
  - Real-time operation
  - Serving system tasks
  - Providing interactive operation (user session in foreground)
  - Batch processing (long CPU burst, but non time critical tasks)
  - System statistics, logs, other tasks with low importance
- Advantages
  - Different levels, can be managed by different (appropriate) scheduling algorithms
  - The levels are managed in a simple way (no level changing)
- Disadvantages
  - Due to static priorities the starvation appears
  - The „nature changes” of the tasks are unmanageable

Static multilevel scheduler



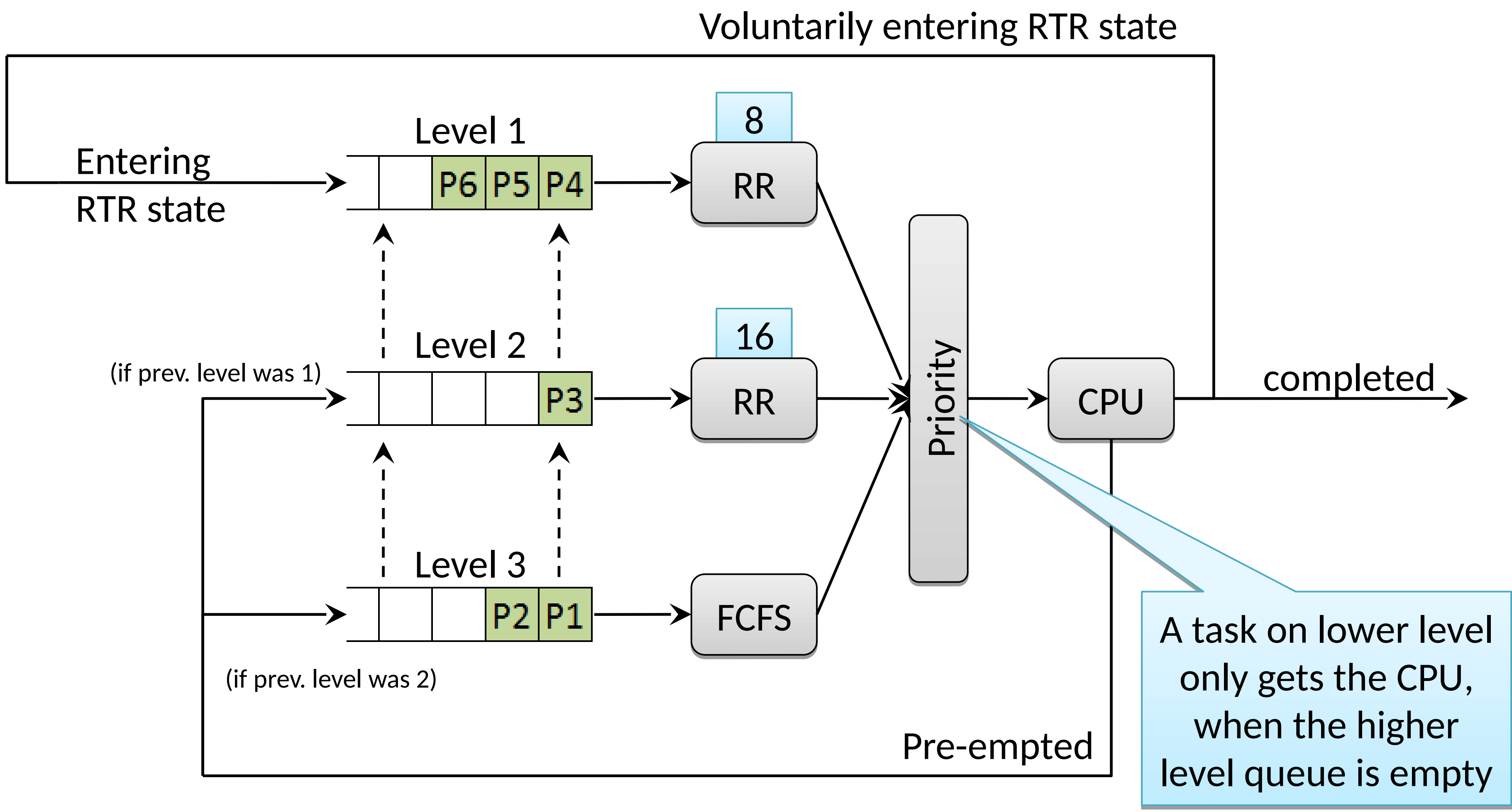
Dynamic multilevel scheduling

- The tasks assignment to queues is dynamic
  - The task's priority can change dynamically
    - Therefore the queue assignment is dynamic (All tasks at creation, estimated as short burst)
  - The task can change queues
    - Upgrade: changing to a higher priority level
    - Downgrade: changing to a lower priority level
  - Beside the above, the operation is the same as static multilevel queues
- Advantages
  - Like in static multilevel queues
    - Different levels, can be managed by different (appropriate) scheduling algorithms
    - The levels are managed in a simple way: ordering the tasks by priority
  - Aging mechanism can be used to avoid starvation
  - The changing nature of the tasks can result different scheduling
- Disadvantages
  - Upgrading and downgrading makes the algorithm more complex
  - More calculations because the dynamic priorities
  - Therefore higher overhead

Multilevel Feedback Queue (MFQ)

- A basic implementation of a dynamic multilevel scheduler
  - The tasks are ordered by the estimated CPU-burst
- The basic idea: learning from the past
  - The more a task uses the CPU, the lower priority level it will get
  - If a task uses less CPU its priority gets higher and it will be upgraded to a higher level
- The scheduling algorithms
  - On the lowest level: FCFS
  - On higher levels: RR with decreasing time-slice
  - This is a globally preemptive scheduler with priorities
- Moving between levels
  - The tasks are entering on the highest level
  - The CPU intensive (using more CPU time) tasks are getting to lower priority levels through time
  - The I/O intensive (using less CPU time) tasks are stays on the higher priority levels
  - The recent CPU time of the starving task are decreasing with time, therefore their priority will rise (like aging)
- Many current scheduler based on MFQ (UNIX, Windows NT kernel)

Multilevel Feedback Queue (MFQ)



Let’s design a scheduler!

- Further information and expectations for schedulers
  - Kernel mode
    - The kernel’s code is running: short CPU bursts, long I/O bursts (loading stuff from disks)
    - The CPU burst is known in advance
    - No CPU intensive tasks, no convoy effect expected
    - Goal: the smallest possible overhead

      No preempting the kernel task. (inconsistence state)
  - User mode
    - Application’s code is running: not known in advance
      - There are resources to wait
      - CPU intensive, I/O intensive, or changing nature tasks
      - Try to estimate the CPU burst, and schedule them accordingly
    - Convoy effect may appear (we have to manage it)
    - There may be priorities between tasks (they are not equally important)
    - It is expected: the tasks on the same priority should get equal chance to get the CPU
- The global properties of the schedulers
  - Priority (there are different importance tasks)
  - Multilevel (the kernel and user mode needs different scheduling)
  - Dynamic (the tasks nature can change, e.g. changing to kernel mode or back)

Let’s design a scheduler – choosing algorithms

- Multilevel (kernel/user mode), dynamic priority scheduler
- Kernel mode
  - Small bursts ▭ let’s use a cooperative scheduler
  - In a non-preemptive case static priorities are suitable
  - Because it’s non-preemptive the protection of the data structures are simple
  - In summary we get small overhead

  - *How and when should the static priority determined?*

- User mode
  - Because the convoy effect, a preemptive scheduler is needed
  - The **optimal** solution would be the preemptive SJF (**SRTF**)
  - **But** Because the user priorities, the simple SRTF is not suitable
  - The tasks with same priority should get equal chance ▭ RR scheduler better

  - *How to combine SRTF and RR schedulers?*
  - *How and when should be the dynamic priority calculated?*
  - *How to manage starvation (with aging, but how)?*

Determination of static priorities in kernel mode

- The priority doesn't depend on
  - the task's priority in user mode (we are on a different level)
  - how much CPU time the task used in the past (no SJF)

- The kernel mode priority is based on
  - **What resource** the task is waiting for?
  - This called: sleep priority
  - For example:
    - Waiting for 20 I/O operations
    - Waiting for 10 input from the character terminal

- When calculate it?
  - After waking up from waiting, it will get the resource's sleeping priority

  - Even if the kernel tasks are preemptive , they can go to waiting.

Determination of priorities in user mode

- Scheduling variables for the tasks
  - `p_pri` – the current priority of the task (smaller ⇒ higher priority, =>0)
  - `p_cpu` – the CPU usage in the past
  - `p_nice` – the priority modifier value, given by the user (integer, =>0)

- The CPU usage in the past is used to estimate the CPU burst
  - The `p_cpu` is incremented in every clock cycle when the task is running

- Calculation of the priority

```
p_pri = P_USER + p_cpu / 4 + 2 * p_nice
```

  - `P_USER = 50` (the kernel priorities are below 50)
  - `p_nice = 10` by default
    - The user may increase it ⇒ priority will drop
    - The root user can decrease to 0 ⇒ priority will increase

The mechanism of aging

- The p_cpu is incremented in every clock cycle when the task is running

  p_cpu++; (getting less priority)
- The p_cpu value should be also „aged" with time
  - p_cpu = p_cpu * CF (correction factor < 1)
- Determination of CF
  - For example: CF = ½ (simple operation, right shift)
  - There are problems with it!

- How to create a better CF? What should it depend on?
  - If there are no RTR tasks → no starvation is possible
    - In this case the value p_cpu can be cleared
  - If there are few RTR tasks, the RR scheduler gives CPU to them in a short period of time
    - p_cpu we can be aged quickly (to bound the priorities)
  - If there are many RTR tasks, the waiting time is higher in RR scheduler
    - p_cpu should aged slowly (to ensure lower priorities for the task which are already used the CPU in the past)
  - Use the average number of the RTR tasks: load_avg

```
CF = 2 * load_avg / (2 * load_avg + 1)
```

Scheduling in user mode

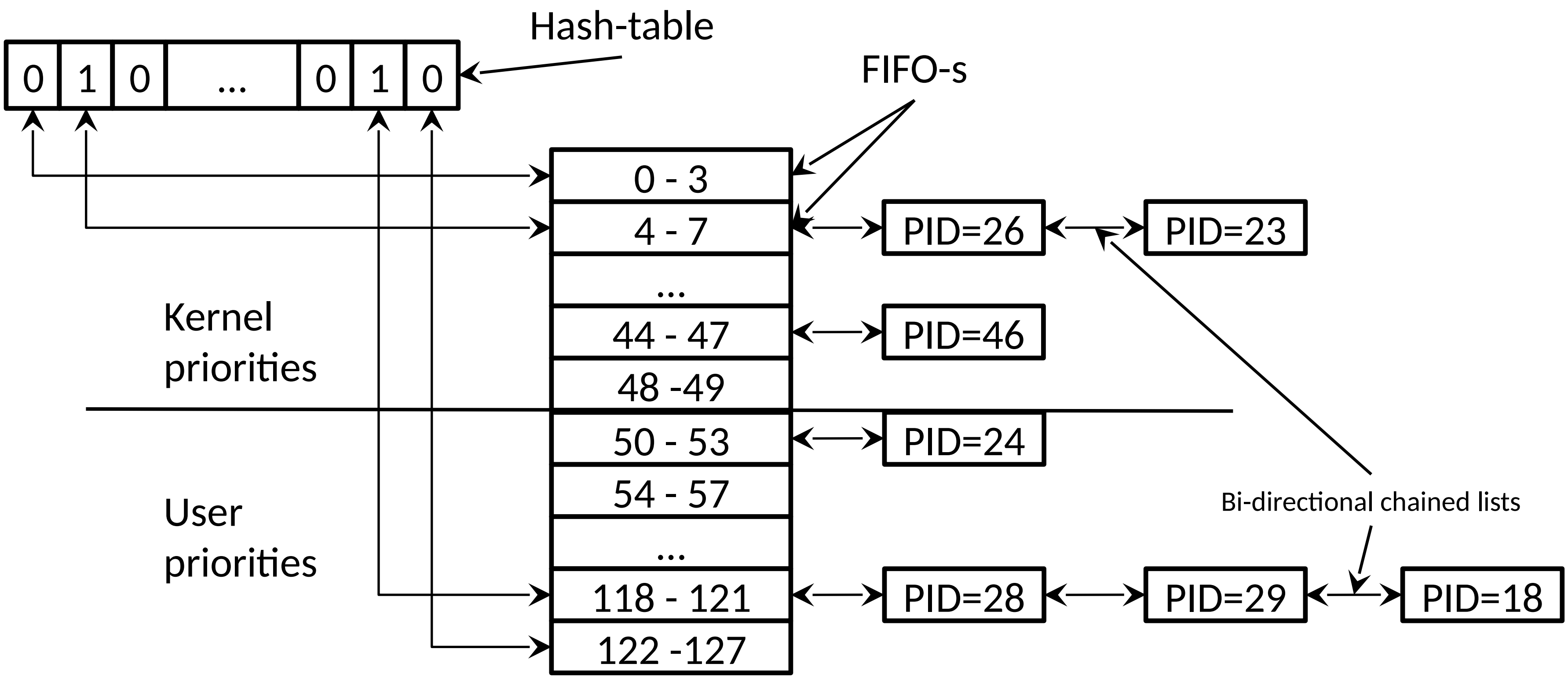
- How to combine SRTF and RR schedulers?
  - SRTF orders the tasks by their priority (optimal)
  - Priority calculation is based on the CPU-burst estimation with the `p_nice`
  - Tasks on the same priority level are scheduled with RR (time-sharing)

▭ the user mode scheduler is also multilevel (Kernel is single)
  - Ordering tasks based on priority

- How is this scheduler has the attributes of SRTF?
  - `p_cpu` estimates the remaining CPU time
  - Priority is determined by `p_cpu`
  - The scheduler orders the tasks by their CPU bursts ▭ SRTF

Data structures of the scheduler

- Priority is an integer: 0 – 127
  - 0 is the highest, 127 is the lowest priority
  - 0 – 49 kernel levels, 50 – 127 user levels
- The scheduler put the tasks into 32 FIFO queues based on their priority
- A hash-table is used to determine which level there are tasks on



Operation of the designed scheduler

- Multilevel scheduler with dynamic priorities
  - In kernel mode: **cooperative**, **static priorities**
    - Priorities depends on the cause of waiting (faster device → higher priority)
  - In user mode: **preemptive**, **dynamic priorities, time-sharing**
    - Priorities depends on the estimated CPU-burst
- Event-based scheduling in kernel mode
  - If a task wakes up by an event, priority is set and the appropriate queue is selected
- Time-based scheduling in user mode
  - Every clock cycle
    - If there’s a task on higher level → preemption
  - At the end of every RR time-slice
    - If there’s a task in the same priority queue as the running task → preemption
    - The preempted task is put to the end of the queue
  - After every ex 100th time-slice
    - „Aging” `p_cpu` → recalculating `p_pri` → reordering queues

Evaluation of the designed scheduler

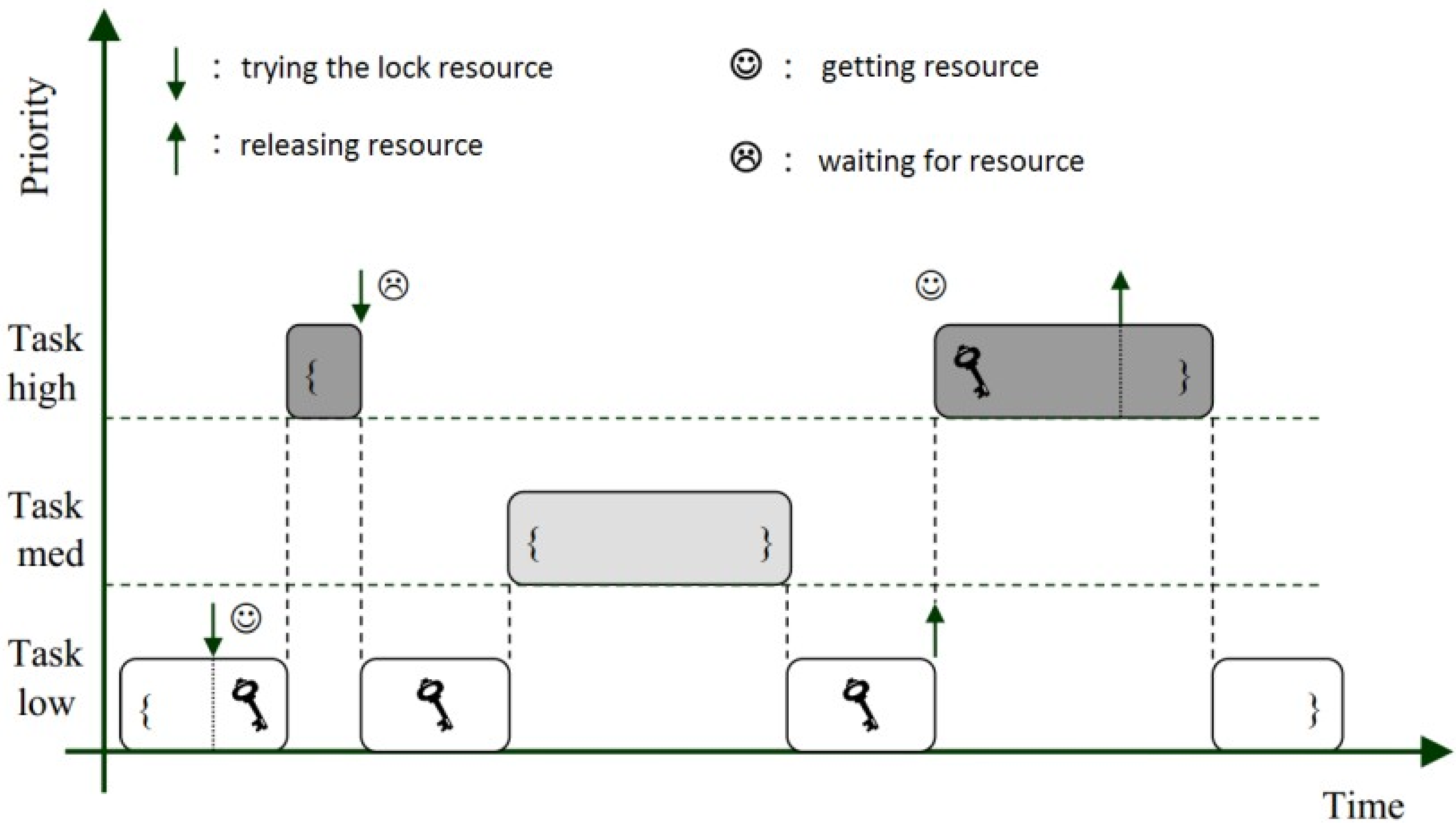
- This is the traditional UNIX scheduler
  - **Multilevel with priorities and time-sharing**
  - Designed for interactive systems
  - Works good also when batch and interactive tasks are in the system simultaneously
  - Provides good response time for interactive tasks while starvation of the background tasks are avoided
- Problems
  - High overhead, when the task count is very high
  - There’s no special tasks (e.g. real-time)
  - Many problems with the **cooperative** scheduler in **kernel** mode
    - Problems are caused by tasks with long CPU-bursts in kernel mode
    - Lower priority task can hold up tasks with higher priority (no preemption)
    - This is called: **priority inversion**
    - Because the single threaded kernel: more CPUs don’t solve the problem

    - How can we manage this?
    - Can we make a preemptive kernel mode scheduler?
    - How can we use more processors?

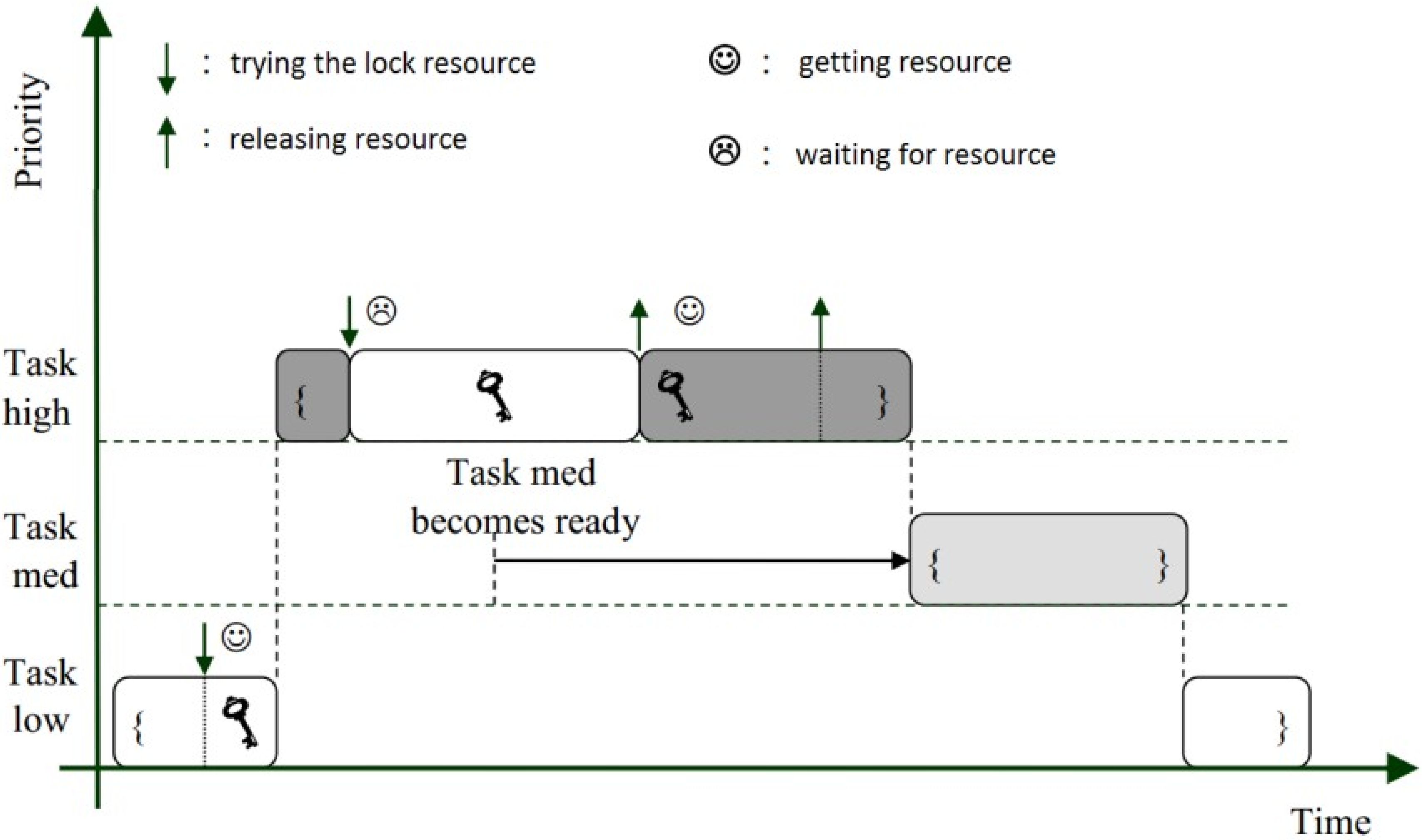
Priority inversion

- Definition: A task (A) with lower priority can hold up a task (B) with higher priority.
  - There is a dependency between the execution of A and B
  - For example: „A” is waiting for a resource in an uninterruptable way, and „B” also waits for that resource
  - It happens often! There can be dependencies between more than two tasks also
  - The result: the priority of „A” seems the same as the priority of „B”
- How it can be managed?
  - Increase the priority of „A” to the „B”’s level for a short time, to resolve the dependency
  - This is called **priority inheritance**
- The bounds of priority inheritance
  - It isn’t always known which task causing the problem
    - Complex dependency graphs, where we can’t find the cause
  - It can happen that more the one task is blocking „B”
    - Cannot increase the priority of many tasks → long-term effects (chain reaction)

Priority inversion



Priority inversion with priority inheritance



Priority inversion – other solutions

- Priority ceiling
  - The task is upgraded to the kernel level
- Priority inheritance
  - As seen before
- Random boosting
  - RTR tasks which are holding locks may randomly boosted to higher priority until they are exit the critical section
- Avoid blocking

How to make the kernel scheduler preemptive?

- Introducing preemption points
  - During certain points of the execution of the kernel code a task change is allowed (when no critical job is done) (no inconsistent state yeilds)
  - At this point it is checked if there's a higher priority task → if yes, preemption
  - The kernel memory consistence should be only ensured on these points
  - E.g. some UNIX System scheduler (some kernel tasks take long time)
    - Introduces real-time tasks, which requires preemption points
    - The scheduler checks, if there is a real-time task
- Fully preemptive scheduling for kernel level (Difficult)
  - This is the only way, if there are multiple processors
    - Current client and server OSs using preemptive kernel
  - All data structures have to be protected (later will be discussed)
    - Problems when
      - two tasks writes the same area
      - one task read from the same area, which modified by another task
  - Non preemptive code sections can be defined
    - Preemption can be disabled in critical section, if there's no other solution

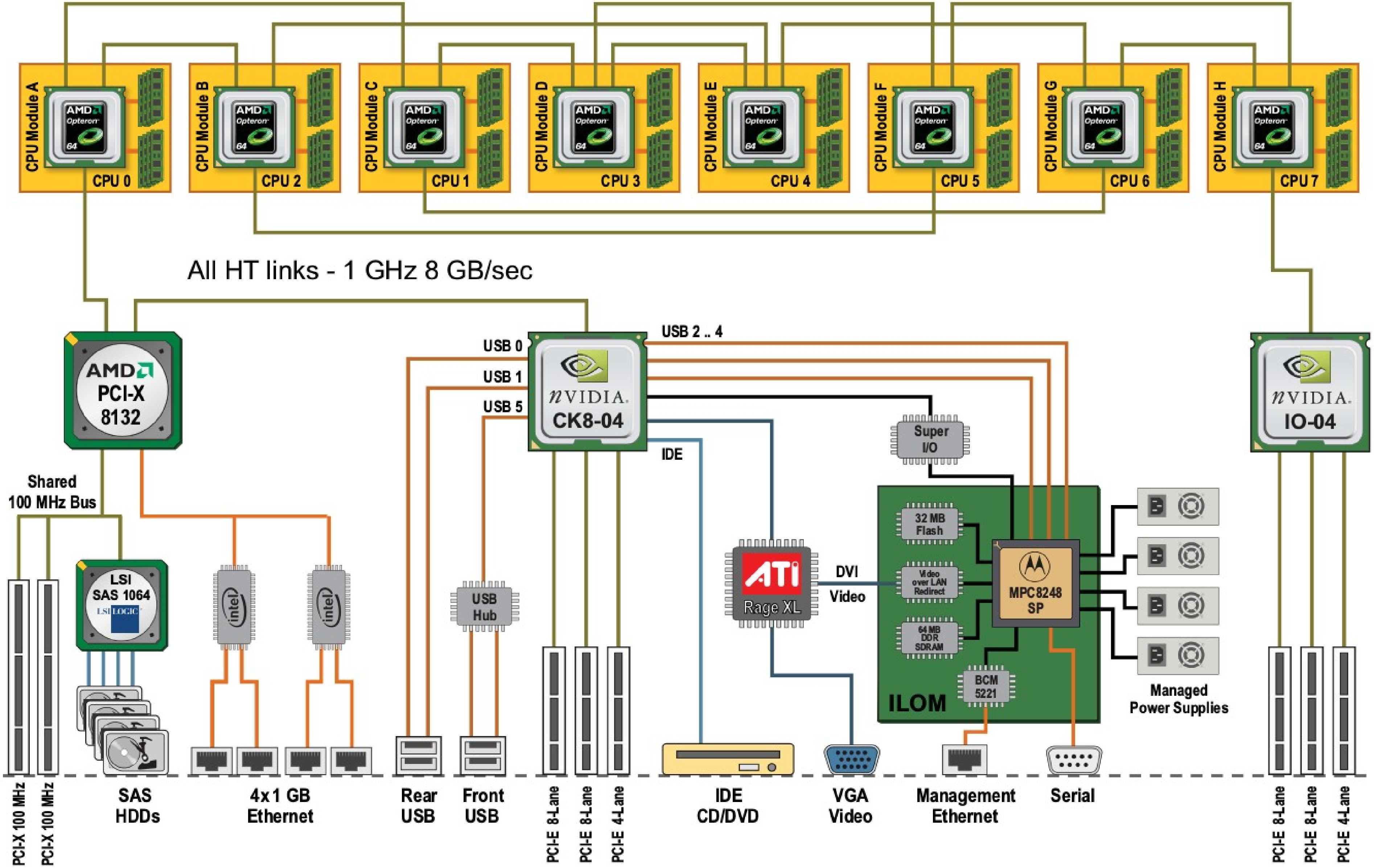
Multiprocessor scheduling

Basic questions of multiprocessor scheduling

- Until now we assumed only one processor unit
  - Current HWs provide multiple CPUs, also with different capabilities
- Multiple tasks may run in kernel mode
  - Preemptive kernel is required with protected data structures
- Managing „remainders” during context changes
  - There are **remaining data** when R → (W →) RTR → R state transition happen
    - In CPU registers, cache memories
    - E.g. handling an interrupt means minimal context change
  - A task should get back to the last processor
    - The scheduler should manage this
- Resource allocation for a process group
  - Processors and process groups (Threds) can be bind together to use same CPU or CPU cluster, because they are sharing same memory
- Load balancing
  - A suitable processor is chosen to each task

Basic variants of multiprocessor scheduling

- Asymmetrical systems
  - One of the units (One CPU) serves the kernel task(s)
  - The user tasks are running on the other units (CPUs)
  - Advantages
    - Easy implementation based on the single processor codes
    - The kernel can be one task → simple to implement
  - Disadvantages
    - The utilization of the CPU assigned to kernel will be low **GPUs**
  - **Rarely used, may be a good solution in heterogeneous systems**
- Symmetrical systems (better)
  - Every processor has its own scheduler
  - The RTR tasks can be in a shared queue, or in separate queues assigned to each CPU
  - Better CPU utilization
  - Risks: requires careful software development
  - **Current systems using this method**


CPU Module A
CPU 0
CPU Module B
CPU 2
CPU Module C
CPU 1
CPU Module D
CPU 3
CPU Module E
CPU 4
CPU Module F
CPU 5
CPU Module G
CPU 6
CPU Module H
CPU 7
All HT links - 1 GHz 8 GB/sec
AMD PCI-X 8132
USB 0
USB 1
USB 5
nVIDIA CK8-04
USB 2 .. 4
IDE
nVIDIA IO-04
Super I/O
Shared 100 MHz Bus
LSI SAS 1064
USB Hub
Rage XL
DVI Video
32 MB Flash
Video over LAN Redirect
64 MB DDR SDRAM
MPC8248 SP
BCM 5221
ILOM
Managed Power Supplies
PCI-X 100 MHz
PCI-X 100 MHz
SAS HDDs
4x1 GB Ethernet
Rear USB
Front USB
PCI-E 8-Lane
PCI-E 8-Lane
PCI-E 4-Lane
IDE CD/DVD
VGA Video
Management Ethernet
Serial
PCI-E 8-Lane
PCI-E 8-Lane
PCI-E 4-Lane

Processor affinity and its types

- The scheduler has to adapt to the HW
  - The increased amount of context changes should not cause more overhead than the benefit of using multiple processors
  - A preempted task should get back to the same CPU
    - It is not possible in every case: high load & high number of tasks
    - Priority of the tasks may influence this behavior
- Processor affinity
  - Binding processes to a specific CPU
  - This bond should be maintained
    - It is hard to maintain for tasks with large working set
  - Soft affinity
    - The OS tries to maintain the bond, but no guarantees
    - This is the default behavior in current operating systems
  - Hard affinity
    - allows a **process** to **specify** a subset of processors on which it may run

Thank You

Next Weeks Till Mid-Term

- Scheduling practice – 2 Okt
- Communication – 3 Okt
- IPC – 9 Okt
- Synchronization – 16 Okt
- Synchronization practice
- Mid-term