Operating Systems Internals – Task scheduling

Lecturer: András Millinghoffer
milli@mit.bme.hu

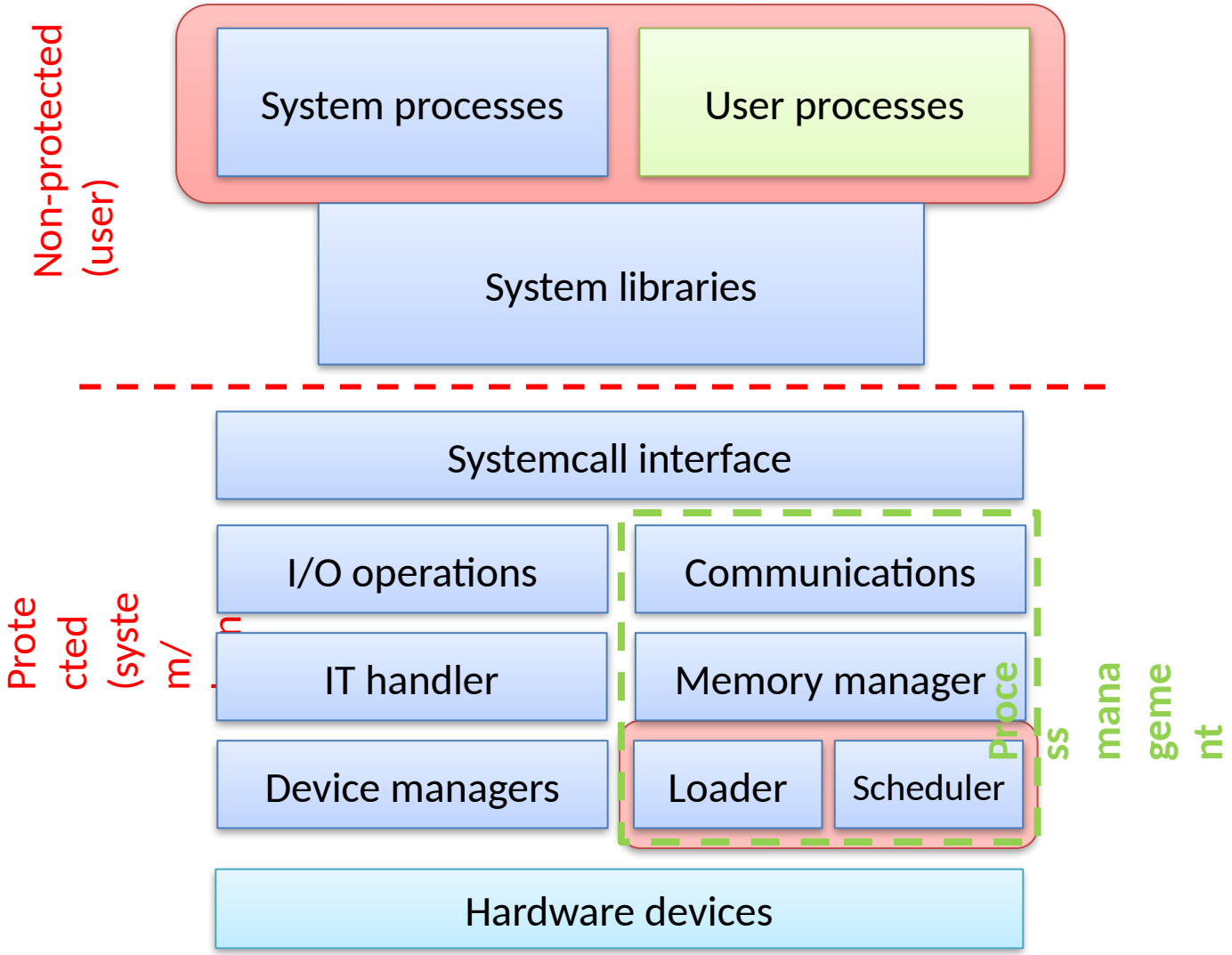
Budapest University of Technology and Economics (BME)
Department of Artificial Intelligence and Systems Engineering (MIT)

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 management (recap)

- The OS serves the user and system tasks
  - I/O intensive tasks
  - CPU intensive tasks
  - Memory intensive tasks
  - The OS don’t know the nature of the tasks in advance
- User expectations can be various
  - Waiting time, turnaround time, response time, throughput, CPU utilization %
  - Deterministic, small overhead
- Task: a program during execution
  - Life-cycle: the sequence of state transitions from the start
  - Abstract virtual machine: virtual CPU and memory for the tasks
  - Process: has it’s own memory range, contains threads
  - Thread: sequential task, may share memory with other threads
- The role of the scheduler
  - Controlling the transition: ready-to-run -> run
  - Event (interrupt) driven
  - Task change is performed very often in a current system
  - The overhead of the scheduling and the task (context) change should be minimal

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

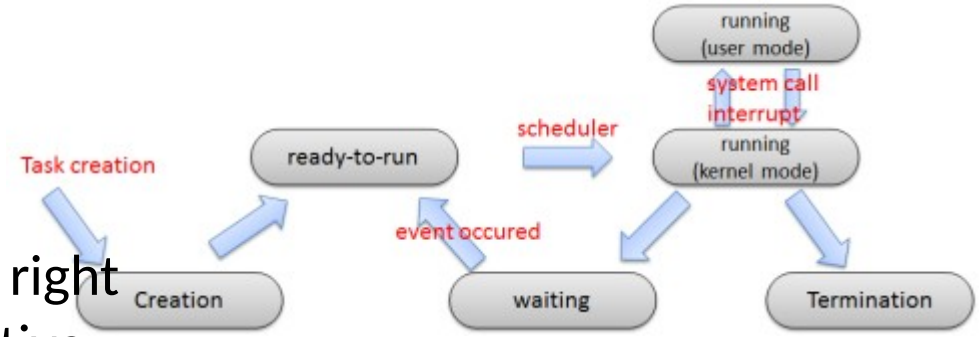


The goals and temporal properties of the scheduler

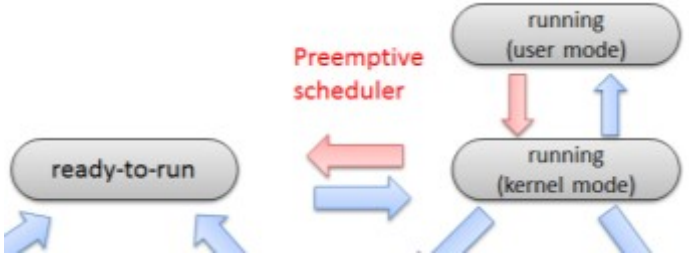
- The scheduler has to decide which task should run
  - Managing the ready-to-run and currently running tasks
  - Selects the next one which enters the run state (gets the CPU)
  - This decision is based on system and task properties
- Where the tasks are selected from?
  - Short term (or CPU scheduling)
    - Selects a task from the ready-to-run queue
    - The operation interval is usually in the range of 1-100 ms
    - We will examine this type of scheduler later
  - Long term
    - Starting and stopping jobs (tasks or task groups)
    - Usually in the user space (e.g. UNIX cron)
    - Operation interval: hours, weeks, months
  - Medium term
    - Some tasks are pulled out from the short term scheduler temporarily
    - It can enhance system performance
    - The user also can initiate it: pause specific tasks
    - Operation interval: minutes, hours

Short term scheduler in the kernel (recap)

- A modern OS is event driven



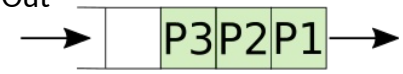
- When a task can only give up the right of running voluntarily -> Cooperative (non-preemptive) scheduler



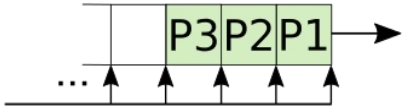
- A preemptive scheduler can take away the CPU from a running task

The data structures and complexity of scheduling

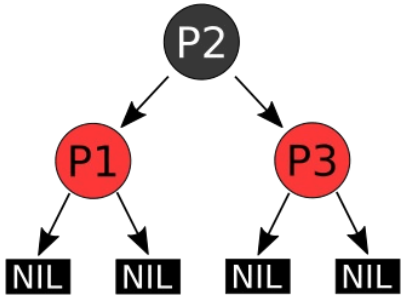
- Managing (registering) the tasks in different states can be done with:
  - Queues (typical)
    - The tasks are in a linear data structure
    - FIFO: First In First Out



    - List: insertion or taking out can be performed at any index



  - Tree structure (e.g.: Linux kernel 2.6.23+)
    - Iteration is faster than on a queue
    - E.g.: red-black tree (see Theory of Algorithms subject)



- The complexity of a scheduler
  - Important aspect (runs frequently)
  - The complexity of data manipulations worst case
    - Put to and get from FIFO: O(1) – constant
    - Seek from and put to chained-list: O(N) – linear
    - Seek from and put to red-black tree: O(log N) – logarithmic
  - The complexity of other parts of the scheduler: O(1) – constant
  - Theoretical and practical complexity usually not the same

Simple scheduling algorithms (introduction)

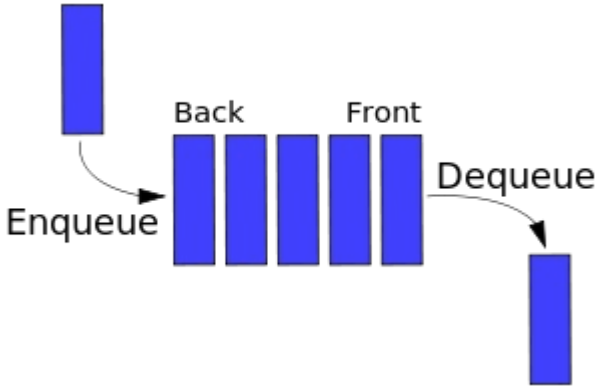
- Definitions
  - Tasks contains CPU and I/O operations
    - CPU-burst: sequence of instructions executed on the CPU by a task
    - I/O burst: the task waits for an I/O operation
    - See CPU/IO intensive tasks…
  - States of tasks
    - Running (R), ready-to-run (RTR) and waiting (W)
  - The schedulers task to manage the R ↔ RTR state transitions
  - For the first time, we examine the single processor scheduling
  - The operation of the scheduler can be visualized with a Gantt-chart:

A scheduling example

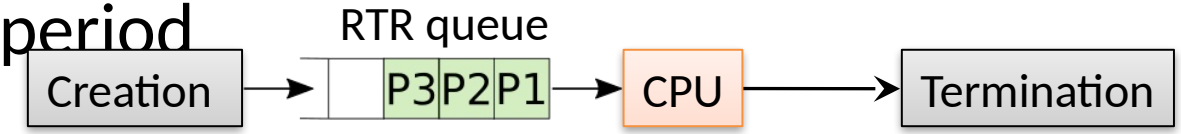
- Like in a strategy game
  - Task
    - Occupy/defend targets
    - Different importance (castle vs. mine)
  - The processing unit
    - Soldiers
  - The scheduler
    - Players
  - Temporal properties
    - Long term: when to play?
    - Medium term: break for dinner
    - Short term: where to go?
  - The attack/defense
    - Preparation (I/O burst)
    - Combat (CPU burst)

First Come First Served (FCFS)

- Data structures and operations
  - FIFO for the ready-to-run (RR) tasks
  - An entry can be enqueued at the end and dequeued on the front

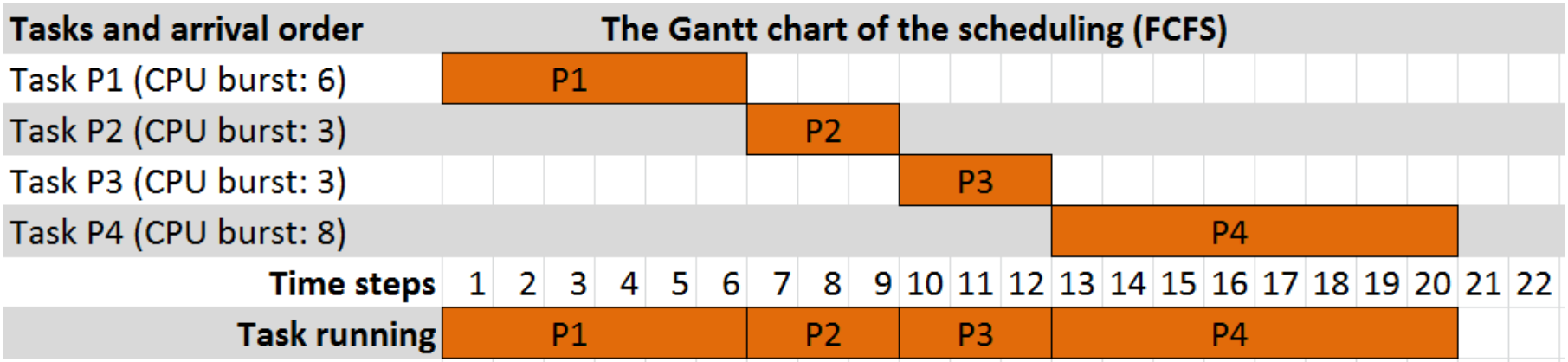


- Scheduling algorithm
  - When a running task gives up the right of running, the scheduler choose the task which is waiting for the longest period



Simple FCFS example

- The tasks are arriving sequentially into the state RTR
- The scheduler executes them in the order of arrival
- The scheduler don’t know the CPU-burst duration in advance



Evaluation of the FCFS scheduler

- Properties
  - Cooperative scheduler (non-preemptive)
  - Very simple data structure and algorithm
- Algorithmic complexity (insertion, seek)
  - O(1)
- Overhead
  - minimal
- Quality of service, problems
  - What happens when a long CPU burst task runs?
    - Convoy-effect: The RTR tasks are jammed behind the long CPU burst R task
    - There are some I/O intensive tasks, but they won't entering into waiting state
    - This method results higher average waiting time
- Practice example
  - Tasks arriving in order: P1…P3, CPU bursts: P1: 24, P2: 3, P3: 3
  - How long is the average waiting time?
- Further thoughts (for home)
  - How can we evaluate this scheduler when all tasks are I/O intensive?

Practice

Tasks and arrival order	The Gantt chart of the scheduling (FCFS)
Task P1 (CPU burst: 24)	P1
Task P2 (CPU burst: 3)	P2
Task P3 (CPU burst: 3)	P3

Time steps	1	2	3	.	.	.	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40

P1 waits:	0
P2 waits:	24
P3 waits:	27
AVG:	**17**

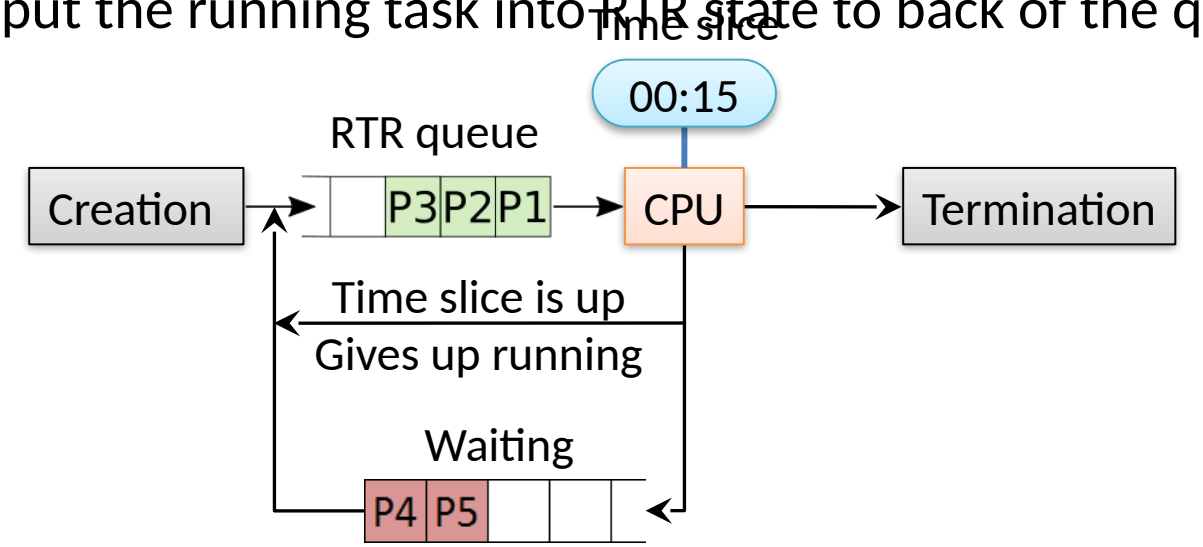
How to address these the problems of FCFS?

- Interrupt the tasks with long CPU bursts
  - The convoy-effect disappears
  - It is expected: shorter average waiting time
  - The scheduler becomes preemptive
  - The algorithmic complexity stays the same
  - ⇒ **Round-robin scheduling (RR)**

- Execute the task first with shorter CPU burst
  - The convoy-effect disappears
  - The I/O intensive tasks enters quickly into waiting (W) state
  - From the CPU intensive tasks, the shorter burst tasks runs quicker
  - It is expected: shorter average waiting time
  - The scheduler stays cooperative (non-preemptive)
  - The algorithmic complexity is higher ⇒ more complex data structure
  - Higher overhead
  - ⇒ **Shortest job first scheduling (SJF)**

Round robin (RR) scheduling

- A task loses the right of running when a specified time slice is over
- Data structure
  - RTR tasks in a FIFO queue
- Algorithm
  - When a task ends or the time slice is up, choose the task with the longest waiting time from the FIFO (from the front of the queue) and put the running task into RTR state to back of the queue



Evaluation of the RR scheduler

- Properties
  - Preemptive scheduler (there are non-voluntarily task changes)
  - Very simple data structure and algorithm
- Algorithmic complexity (insertion, seek)
  - O(1)
- Overhead
  - Minimal, depends on the time slice specification
- Quality of service, problems
  - Better avg. waiting time (compared to FCFS), but non optimal
  - **How long the time slice should be? – It has great influence on the performance**
    - Too long ⇨ It becomes an FCFS scheduler
    - Too short ⇨ Too many context changes, higher overhead
- Practice example
  - Tasks arriving in order: P1…P3, CPU bursts: P1: 24, P2: 3, P3: 3
  - How long is the average waiting time if the time slice is 2, or 6?
- Further thoughts (for home)
  - Define the average waiting / turnaround time in the function of the time slice

Practice

Tasks and arrival order	The Gantt chart of the scheduling (RR, TS=2)
Task P1 (CPU burst: 24)	P1 P1 P1
Task P2 (CPU burst: 3)	P2 P2
Task P3 (CPU burst: 3)	P3 P3

Time steps	1	2	3	4	5	6	7	8	9	10	11	12	.	.	.	29	30	31	32	33	34	35	36	37

P1 waits:	6
P2 waits:	6
P3 waits:	7
AVG:	**6,33**

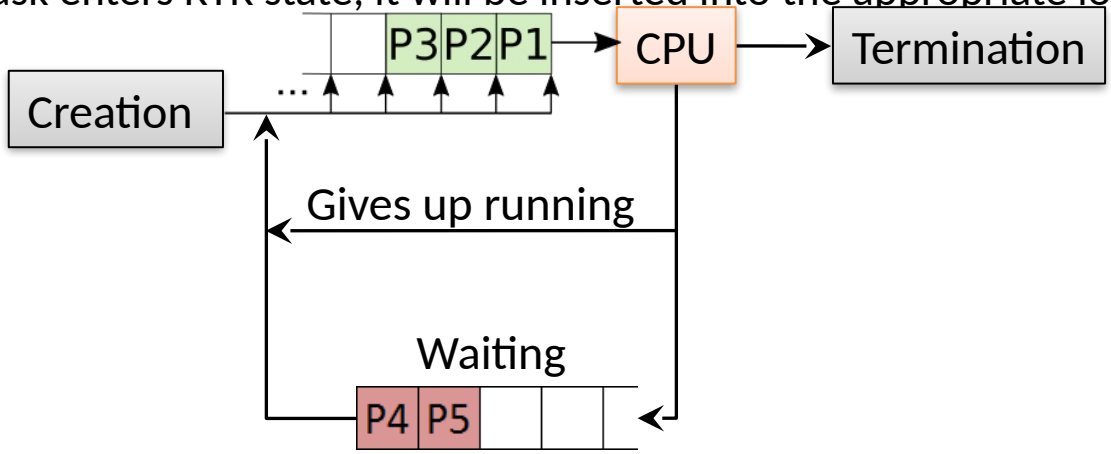
Tasks and arrival order	The Gantt chart of the scheduling (RR, TS=6)
Task P1 (CPU burst: 24)	P1 P1
Task P2 (CPU burst: 3)	P2
Task P3 (CPU burst: 3)	P3

Time steps	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	.	.	.	28	29	30	31	32	33

P1 waits:	6
P2 waits:	6
P3 waits:	9
AVG:	**7**

Shortest job first (SJF) scheduling

- It chooses the task with the shortest CPU burst
- Data structure
  - The RTR tasks are in an list, which is ordered by the CPU bursts
  - Insertion: any position according to the CPU burst
  - Acquiring: from the front of the list
- Algorithm
  - If a task ends or gives up the CPU, choose the task from the front of the list(this task has the shortest CPU burst)
  - It has a preemptive version: Shortest Remaining Time First (SRTF)
    - There is a time slice, like in RR
    - See later
  - When a task enters RTR state, it will be inserted into the appropriate location in the list



Evaluation of the SJF scheduler

- Properties
  - Cooperative scheduler
    - It has a preemptive version: SRTF
  - More complex data structure (a list ordered by CPU burst)
- Algorithmic complexity (insertion)
  - O(N) – we have to find the appropriate place in the list
- Overhead
  - More than FCFS and RR schedulers
- Quality of service, problems
  - Proven: it is optimal regarding to the waiting and turnaround time
  - Difficulties: How can we know the task's CPU burst?
    - We don't know: it is based on estimations
    - In embedded systems some task has predefined CPU burst
- Practice example
  - Tasks arriving in order: P1…P3, CPU bursts: P1: 24, P2: 3, P3: 3
  - How long is the average waiting time?
- Further thoughts (for home)
  - Calculate the waiting time with different CPU burst estimations!

Practice

Task and arrival order	**The GANTT chart of the scheduling (SJF)**																	
Task P1 (CPU burst: 24)							P1											
Task P2 (CPU burst: 3)	P2																	
Task P3 (CPU burst: 3)				P3														
Time steps:	1	2	3	4	5	6	7	8	9	.	.	.	28	29	30	31	32	
	P1 waits:			6														
	P2 waits:			0														
	P3 waits:			3														
	AVG:			**3,00**														

Further scheduling attributes

- Goal: enhance the global performance
  - The task are not the same (foreground vs. background tasks)
  - There are other aspects which should considered by the scheduler
  - We can introduce priorities for the tasks
- The previous schedulers ignores such properties of the tasks
- Solution: provide more information to the scheduler
  - Simple way: **priority**
  - A integral number assigned to each task

Schedulers with priority

- Common problem in previous schedulers
  - Cannot distinguish between task according to their importance
  - Common expectations are cannot fulfilled
    - Kernel tasks (e.g.: interrupt handler task) are more important
    - The user tasks are not equal (multimedia, real-time, background task,...)
- Assign priorities
  - Priority is a number according to the expected running order of the RTR tasks
  - No standard for the value range, usually divided into ranges
    - UNIX: 0-49 kernel tasks, 50-127 user tasks
    - Windows: 16-31 static priority tasks, 1-15 dynamic priority tasks
  - Priority can be changed by the task and the user (in a constrained way)
    - It can be increased or decreased according to the actual permissions of the task owner
  - Priority determination
    - External priority: determined by the user or the task
    - Internal priority: determined by the kernel (e.g. scheduler, resource mgmnt.)
  - Priorities can be static (constant during the task's life-cycle) and dynamic

Starvation: common problem of schedulers with priority

- If the tasks aren't equal
  - The one with higher priority comes ahead of lower priorities
  - If there are many tasks with higher priority: the task never gets the CPU
    - This is called starvation or indefinite blocking
  - If this is allowed, there may tasks which are never gets the CPU

- How can we avoid starvation?
  - It cannot avoided with static priorities
  - We should use dynamic priorities
  - The priority of a starving task should be increased slowly
  - This called **aging**
    - The tasks in RTR state has a constantly increasing priority, proportional to the time spent in the queue (waiting time)
    - Sooner or later it will come ahead the higher priority tasks and gets the CPU

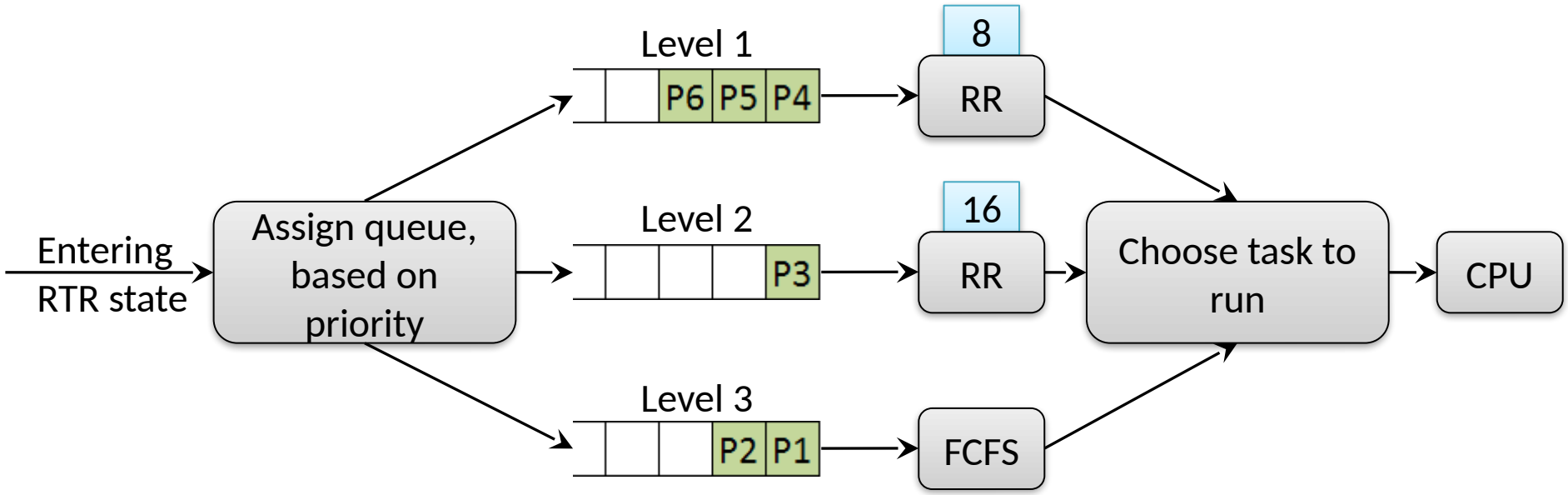
Multilevel scheduling

- 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
  - E.g.: a batch job may become interactive for a short time (Asks something from the user)

Static multilevel scheduler



Next time: dynamic multilevel queues...

Summary

- 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
  - 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)