Handling of Tasks in Operating Systems Supporting Multiprogramming

Tamás Kovácsházy, PhD

2nd topic, Scheduling

Méréstechnika és Információs Rendszerek Tanszék

History

- Batch systems...
- Spooling...
- Multiprogramming
  - 1 processor (execution unit), M tasks
  - Task types
    - System tasks
    - Batch tasks
    - On-line tasks (users)
      - A user may have multiple independent or dependent tasks
      - A single task may be divided to subtasks
  - Task/Job pool (combination of tasks the system handles)
  - To execute tasks resources are needed

The notion of task

- In case of multiprogramming the aim is to execute the task pool in an optimal way
- What does optimal mean?
  - Application specific
  - Optimal is quite different for a notebook, for a smartphone, for a server, or a hard real-time system such as ESP or ABS (in road vehiceles)
  - It is a fundamental question, we are going to talk about it a lot...
- In the default case tasks should know nothing about the other tasks:
  - They run in a virtual machine
    - They have their on CPU and memory virtually
  - They share the physical resources of the machine, and they need to interoperate (synchronize, communicate, etc.)
  - Tasks cannot solve these themselves, they need to use the operating system (its services) to provide these functionalities
    - The OS is the “The great machinator”, or the “Enlightened Dictator” in the system

Example from another fields 1.

- A department in a hospital
- Execution units
  - Doctors, nurses, support staff (heterogeneous multiprocessing system)
  - They are the most valuable resources of the system too!
- Resources (memory, peripherals etc.)
  - Rooms, special equipments, etc.
    - They are available in limited numbers
    - They must be available to tasks according to rules
  - Consumables (energy, stb.)
    - Medicine, cleaning stuff such as detergents, etc.
    - They must be consumed only when necessary
  - To access resources you need time

Example from another fields 2.

- Task: Caring for patients and operating the department
- Scheduling, scheduler:
  - The head of the department and the leading nurse distribute work to staff
    - Who does what and when
  - What if the task pool changes (fact of life)?
    - A task finishes, or the task changes (e.g. the status of a patient deteriorates)
    - Tasks are rescheduled
  - How execution units share resources?
- Luckily, it is drastically simpler for computers
- Generally speaking it is project management!

Operating systems

- We need to discuss the fundamental concepts
- It will be tiresome, but we cannot avoid it

Event

- Something happens during the operation of the system
- Internal event
  - Software interrupt (system call) or exception
- External event
  - Hardware interrupt
- Modern operating systems are interrupt driven!
- It is also called “event driven”...

Task 1.

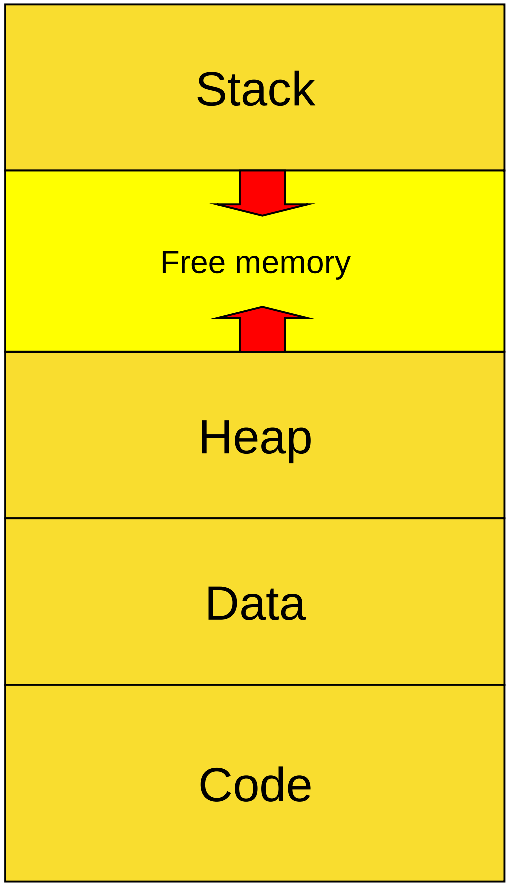
- The concept of task is used in an abstract way here
  - Later we are going to introduce the concept of process and thread to specify the fine details of operation
  - We may also call it a job
  - In some cases the terminology “process” is used, but that is somewhat misleading
  - Even OS API calls are not really consequent from this aspect
    - E.g. some OS API have a function called CreateTask()
      - They do very different things, some create a thread by definition, some other a process
    - Never use them based on their name, you have to read the documentation, and understand what the actual API call does!

Task 2.

- The task is a fundamental notion in multiprogramming
- Later we will realize the task as processes or a set of collaborating processes (implementation)
- A task is the execution of operations in a given order
  - It starts, executes instructions sequentially, it ends
    - This definition is going to be extended by the notion of threads (be prepared!)
  - A task is a program under execution:
    - Loading, execution, and ending programs are the most important function of an operating system, however, we do not deal with that, it is a hidden detail implemented by the development system (how to make a file that stores an executable program) on the OS (how to load the file)

Task 3.

- A task is more than a program:
  - It is active, not passive! (Virtual CPU)
  - It has a state as a state machine:
    - Minimalistic states: Run, Waiting, Ready
    - In an OS these three states can be extended by new OS specific ones
    - In case of UNIX and uCOS we will detail these states
  - There are data structures filled with data (based on the life of task):
    - Virtual memory (OS + HW)
    - Data (global data 💣)
    - Stack
    - Heap
    - Simplified figure on the right side



Task states 1.

- Simplified state diagram is shown.
- Typically, on OS has more, OS specific states.

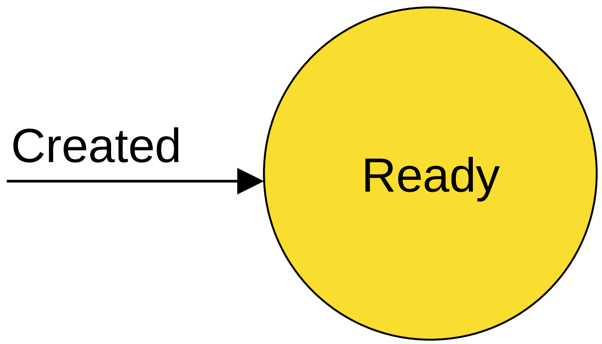
Méréstechnika és Információs Rendszerek Tanszék

Task states 2.

- All tasks must be created first...

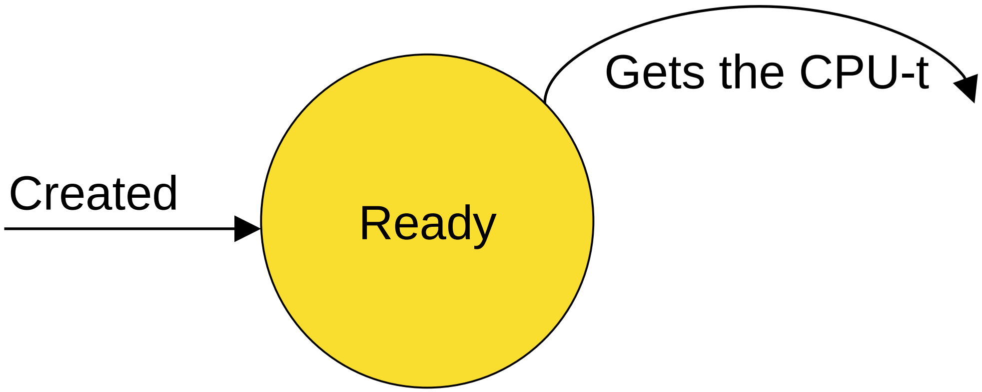
Task states 3.

- Most cases tasks are created in the Ready state
- Task in Ready state has all the required resources to run except the CPU



Task states 4.

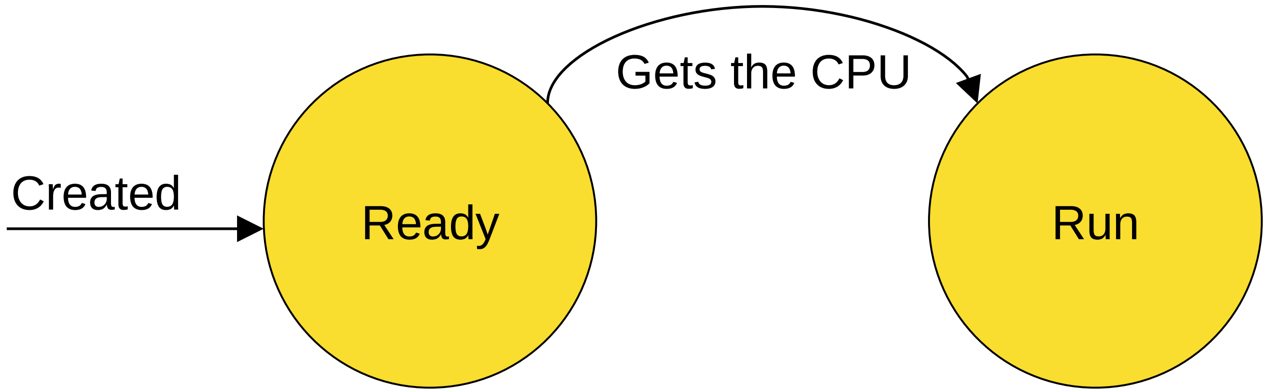
- If the CPU is freed, a task in Ready state can go to Run state
- What is the algorithm that decides which of the tasks in Ready state goes to Run state?
  - CPU scheduling is the solution



Méréstechnika és Információs Rendszerek Tanszék

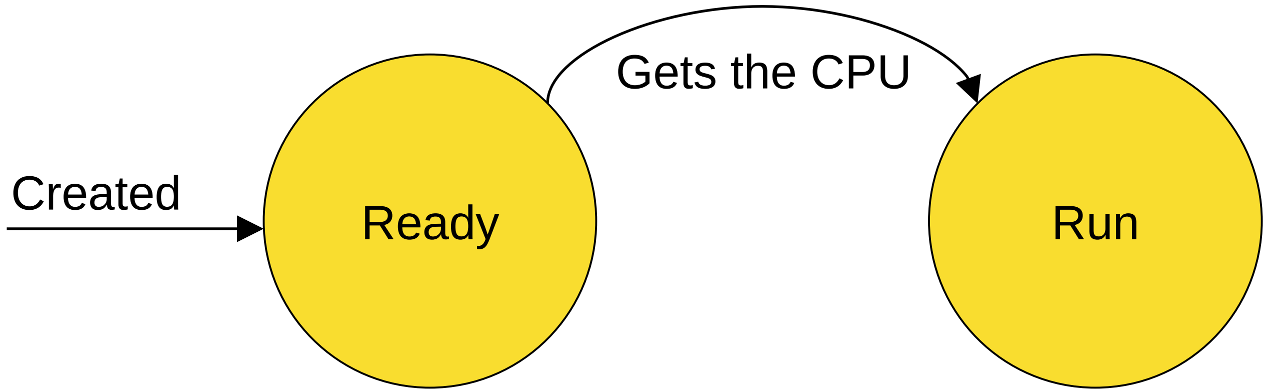
Task states 5.

- The task gets into the Run state, i.e., it runs on the CPU



Task states 5.

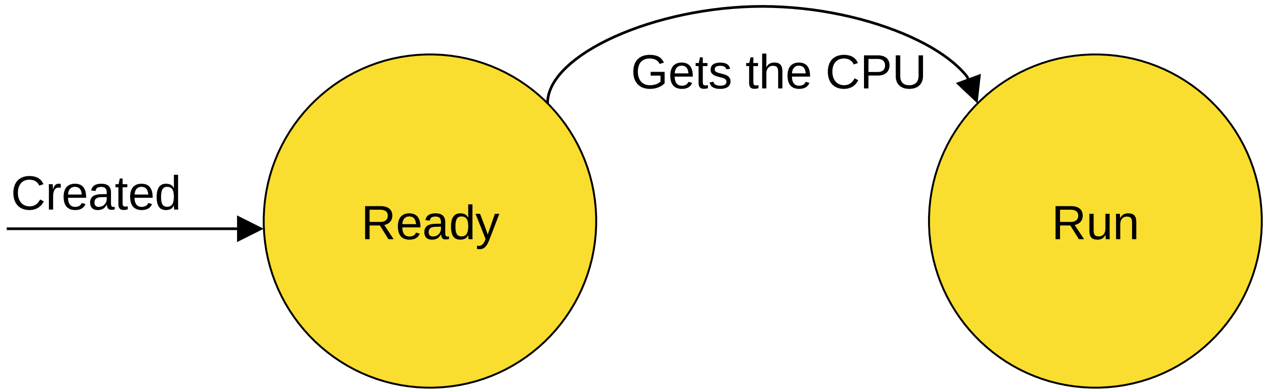
- The task gets into the Run state, i.e., it runs on the CPU



What runs when no tasks are ready?
Something needs to run, the CPU is on!
•Idle task
  •Low priority background task (if the system is priority based)
  •The main functionality of the idle task is power management

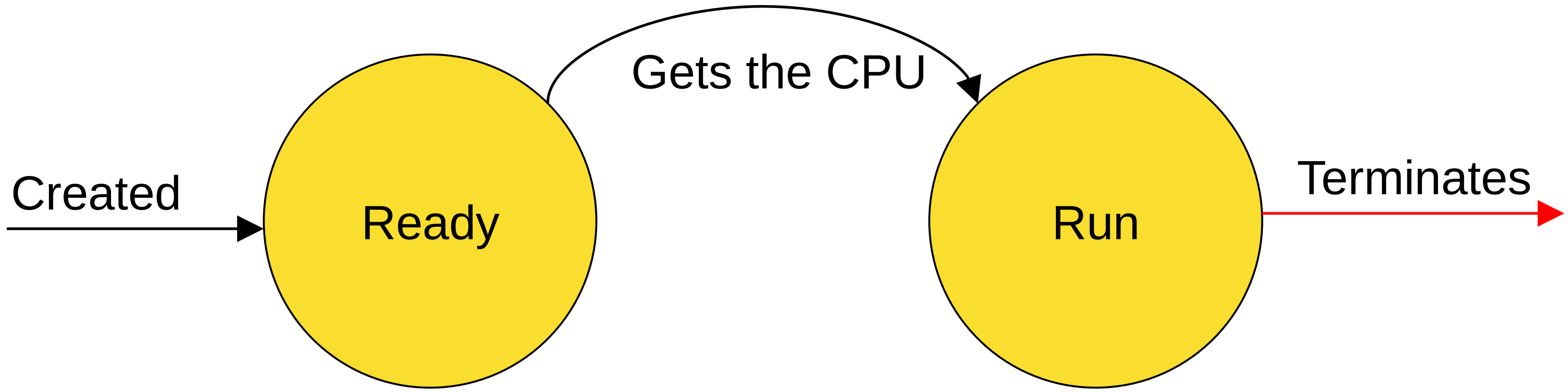
Task states 5.

- The task gets into the Run state, i.e., it runs on the CPU
- What can get the CPU away? Let's examine the possibilities!



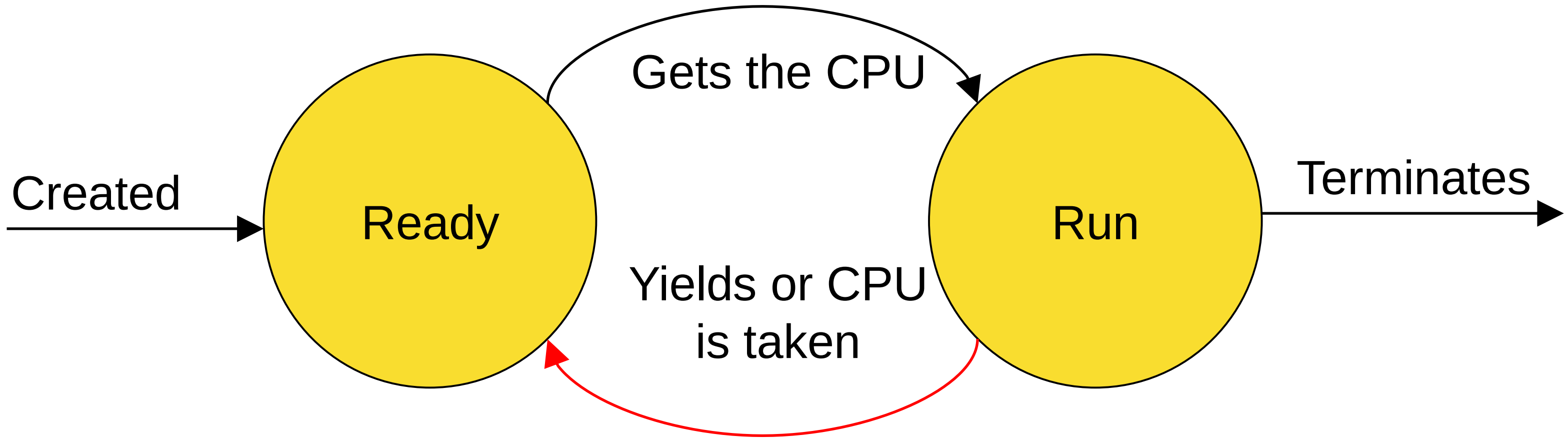
Task states 6a.

- The task terminates (by calling a syscall or making an exception)
- For error handling another temporary state may be needed…



Task states 6b.

- The task yields or the CPU is taken
- There is a yield() syscall in nearly all OSs.
- The CPU is taken, preemptive scheduling, we are goung to talk about it...



How the CPU can be taken?

•The task runs. What can take the CPU?

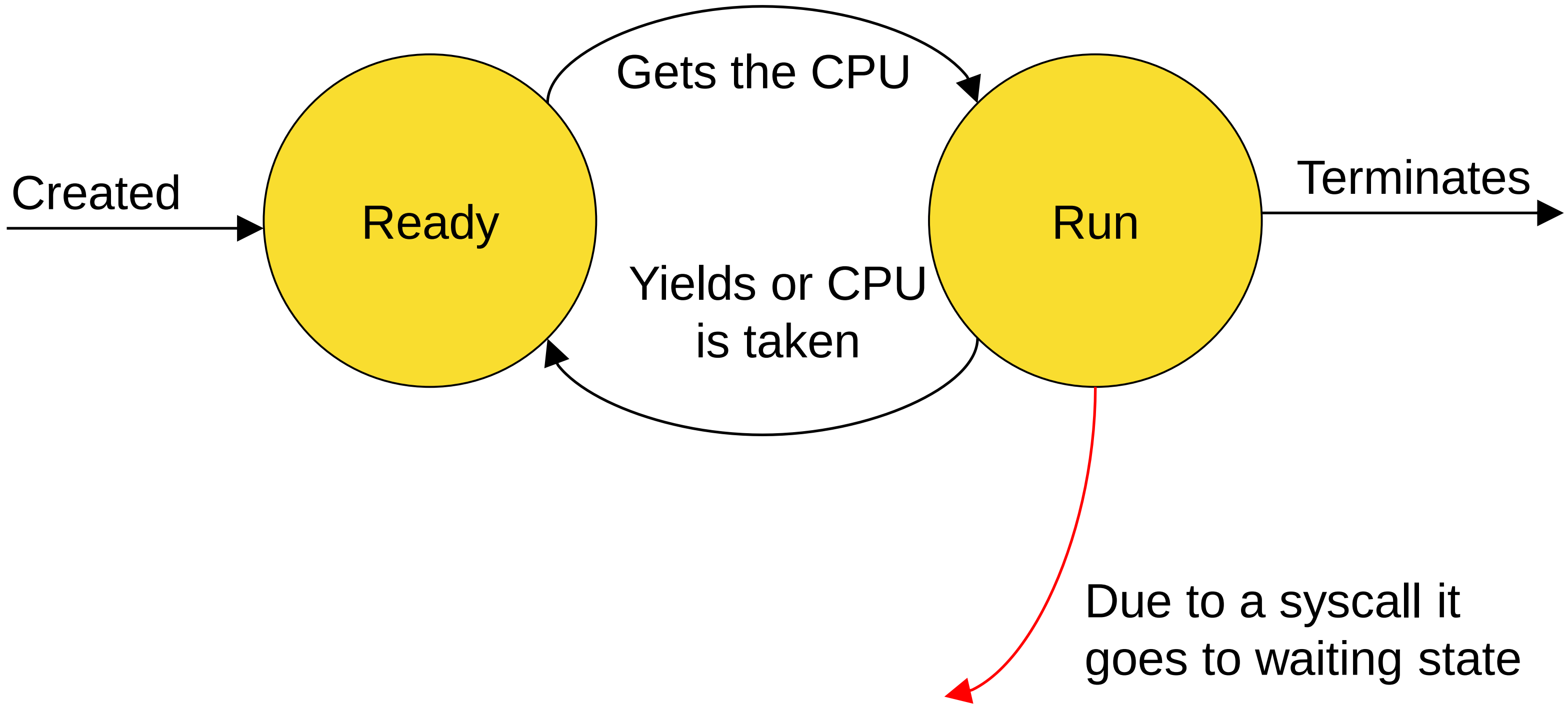
• Only the OS can do it. → It needs to run, but the task runs...

• An incoming interrupt may take the CPU away from the task...

• We need an interrupt (HW, SW, or exception)

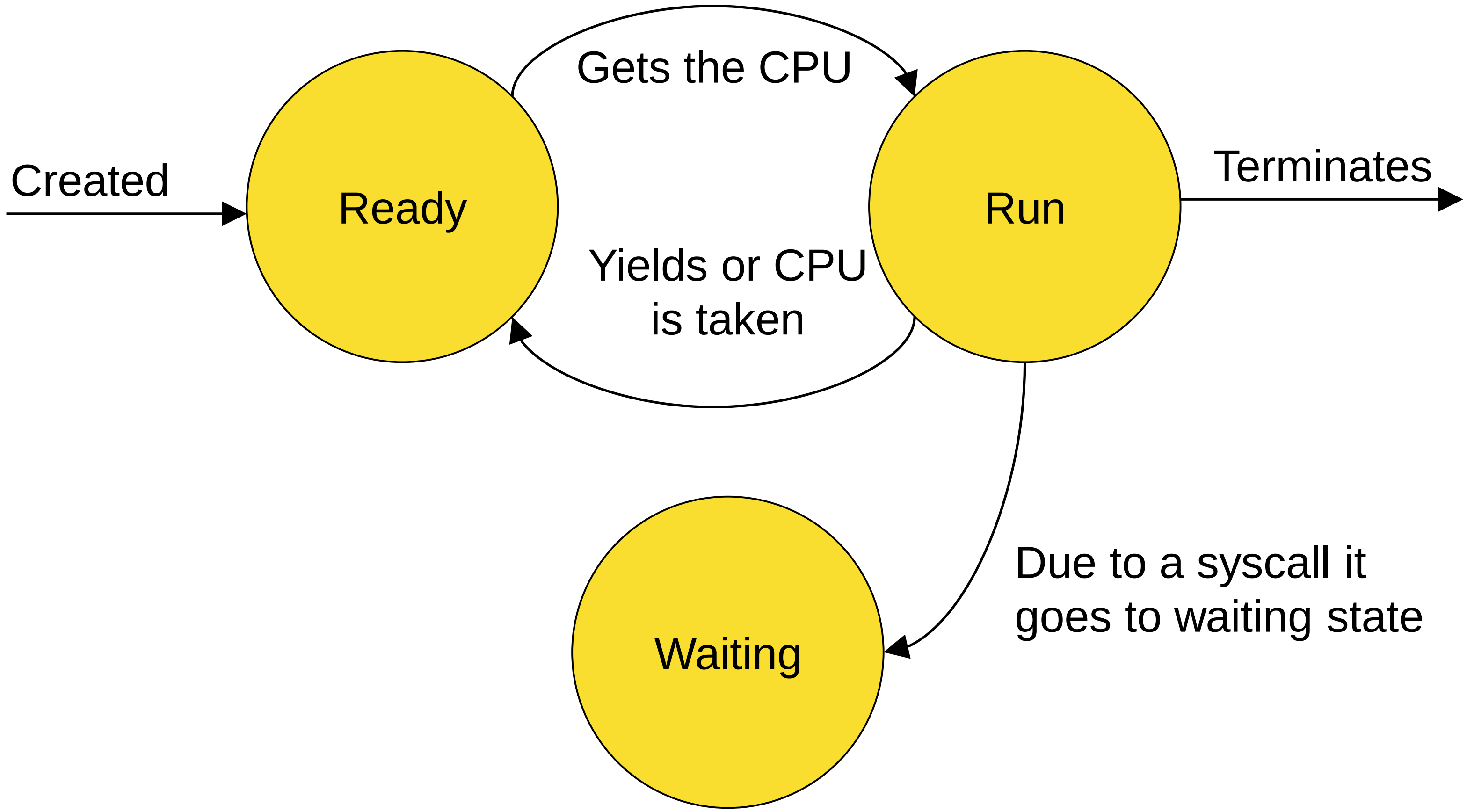
Task states 6c.

- The task may execute a system call but does not gets back the CPU immediately
- The OS may decide that it needs time to handle the request, and for that time the CPU may be used by other tasks



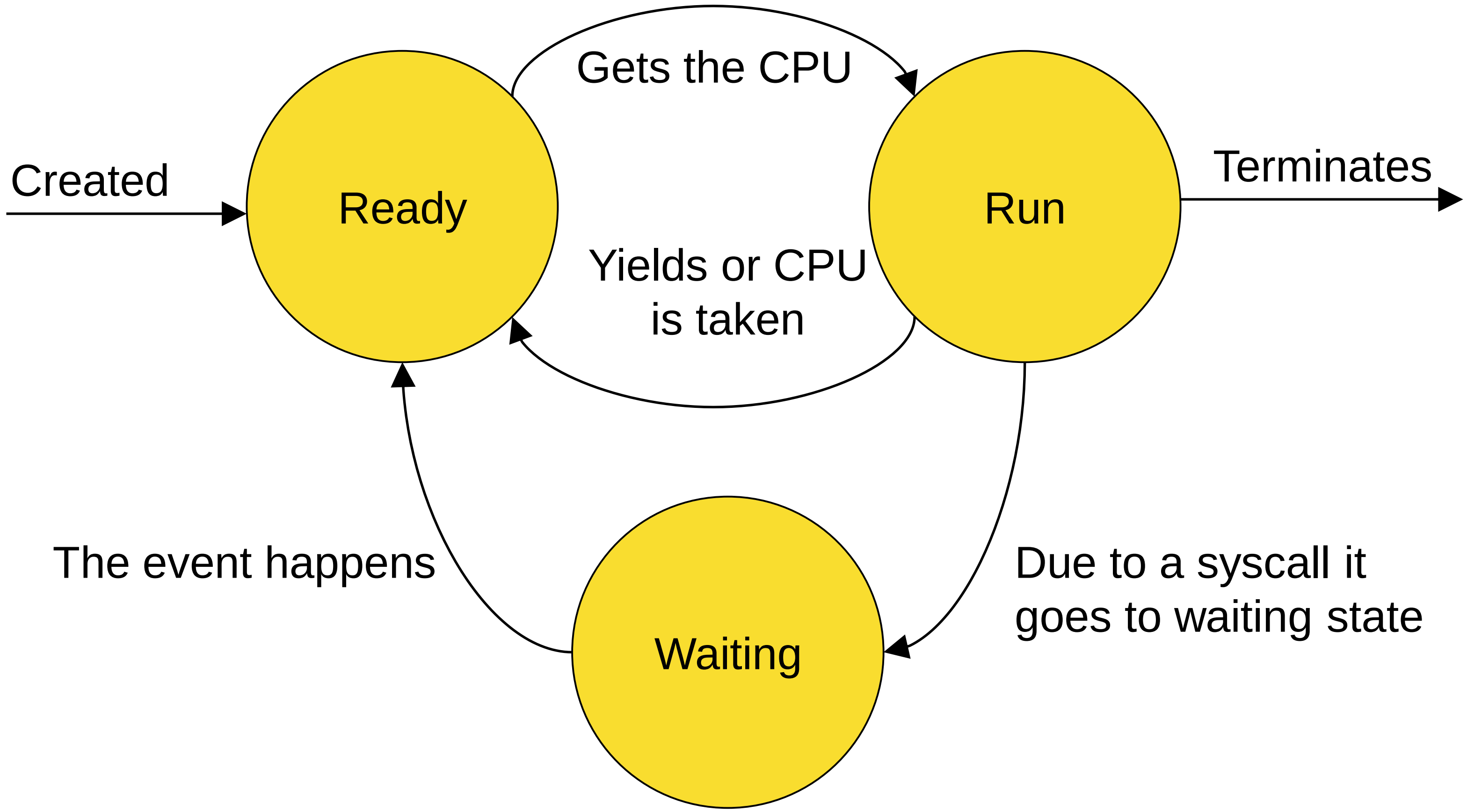
Task states 7.

- The task goes into Waiting state for that time
- The task waits in a passive state, it does not use CPU time



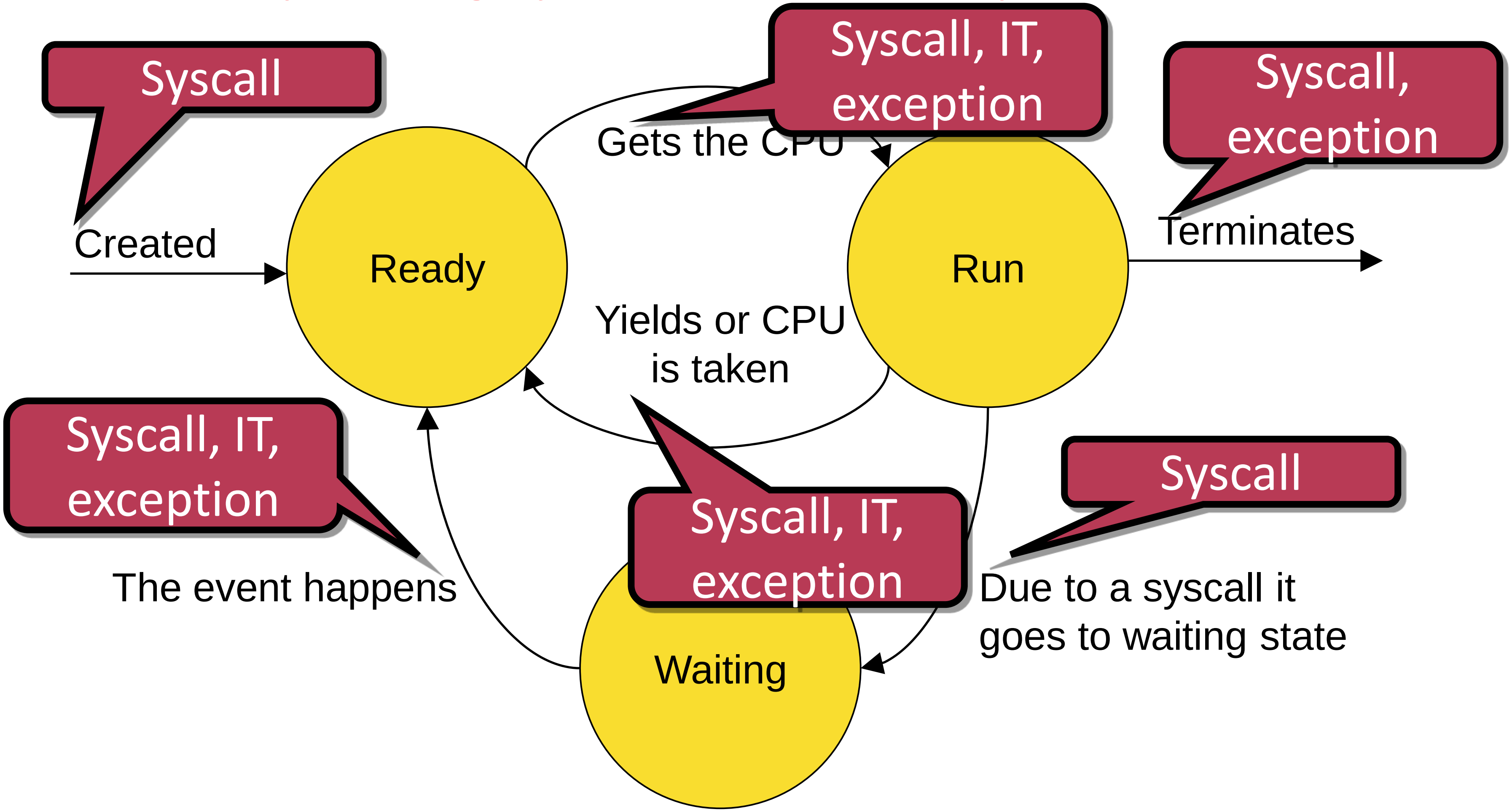
Task states 8.

- The event the task is waiting for happens
- The task can go to Ready state, all resources are available to run except the CPU.



Task states 9.

- All transitions are done on interrupts!
- Modern operating systems are interrupt driven!



Task control block, TCB

- A data structure to store task specific data in the operating system
- The OS handles the tasks based on these data structures
  - The task may have access to these data structures (most cases it can be read but not written).
- What is in it?
  - Task ID (Process ID or Thread ID in real implementations)
  - State (Ready, Run, Waiting and other OS specific states)
  - Program Counter for non-running tasks (context switch)
  - CPU Registers (context switch)
  - CPU scheduling related task specific information
  - Memory related data (virtual memory)
    - MMU state (context switch)
  - Access rights (owner, access control list, i.e., ACL)
  - I/O status information (used resources and their states)

Context switch

- When a task goes into Run state it context must be restored to let it run on the CPU
  - PC and other CPU registers, MMU state, I/O...
- To do that, when a task leaves Run state its context must be saved (by this, later we can continue it)
- It can be done only by the OS, but the OS has also a context...
- The context switch is an overhead (CPU time is lost)
  - Some CPUs have special instructions to do it fast
  - Some CPUs have large number of registers allowing not saving registers, but assigning special registers for tasks
  - Hyperthreading (marketing name, nothing to do with threads)
    - There are multiple processors virtually
    - Architectural state is multiplied only (registers, stb.)
    - If there is some instruction level parallelism, then multiple arithmetic units may be utilizes in parallel (one fixed and one floating point)

Task scheduling

- Our aim: Select one task from the set of tasks in Ready state to put it into RUN state
- Tasks are stored in task queues (job queue) except in RUN state
  - In its simplest form it may be a FIFO, that stores pointers to TCB type structures/objects, i.e., most OSs are implemented in C/C++
  - I.e., there is a Ready queue, a Waiting queue, etc.
  - Queuing diagram of scheduling
  - Scheduling algorithms operate on this data structures (mostly on the Ready queue)
  - It is a search problem, and it is NP complete most of the cases
  - However, it must be executed in bounded time (hard real-time)
    - Overhead must be kept at minimum!

Time scales of scheduling 1.

- Short-term or CPU scheduling
  - Selecting the next running task from the tasks in Ready state
  - Typically it runs in every 10-20 ms or more frequently.
- Long-term scheduling (in batch systems most cases)
  - We have more tasks that we can handle in parallel
  - We let to enter as many tasks into the system we can handle efficiently (some tasks are submitted, but waits somewhere)
  - It runs in every minutes or so (for low overhead).
  - Most cases tasks can be set at specified times (UNIX: cron)
  - Example, copy of 3-4 large files on Windows from a disk to another one (i.e., from the internal one to an external one)
    - 1^{st} approach: parallel copy
    - 2^{nd} approach: sequential copy
    - Question: Which one is faster, the 1^{st} or the 2nd?

Time scales of scheduling 2.

- Medium-term scheduling
  - Swapping (we are going to deal with it later)
    - Tasks in the system have their memory written to the permanent storage (HDD)
    - More tasks can be stored in memory virtually
    - Or one tasks may see more memory physically
  - What if the required parts of memory of a task running is on the permanent storage?
    - It cannot run as long as all the required memory is back in physical memory
  - Medium-term scheduling controls swapping

Even more fundamental terminology

- We need to introduce even more fundamental terminology and definitions...
  - Sorry for that, this is necessary!

Preemptive scheduling

- It is a design decision it the CPU can be taken away from the task in Run state
  - Preemptive:The OS can take away the CPU from the running task
    - Typical in modern operating systems
    - The kernel or certain parts of it may not be preemptive, if it is the case real-time operation may not be guaranteed
      - The kernel of modern operating systems are also preemptive
  - Non-preemptive or cooperative
    - Last example of mainstream operating system: Windows 3.x, MAC OS 1-9
    - The running task must yield or execute I/O to let other tasks to run
    - The operation of the whole system depends on an application
      - Practically on the programmer of that application, not a good idea…
    - If the running task is faulty (infinite cycle) the whole system becomes faulty
      - Is it an OS at all? ☺
      - Some embedded systems use this model, it is simple, and these systems can be tested thoroughly

Metrics 1.

- Metric
  - Allow us to compare scheduling algorithms
  - Watching multiple metrics may make our decision
  - A metric has a unit also!
- CPU utilization
  - Unit: %
  - Time spent in useful work compared to the whole time
  - $t_{CPU}=t_{CPU,work}+t_{CPU,admin}+t_{CPU,idle}$, i.e., administration and idle time is lost
  - Something between 40-90 % is OK most cases

$$\frac{\sum t_{CPU,work}}{\sum t_{CPU}} * 100[\%]$$

Méréstechnika és Információs Rendszerek Tanszék

Metrics 2.

- Throughput
  - Unit: task/s, job/s, or 1/s
  - Task done in specified time
  - System tasks are not counted
    - They reduce throughput
  - Typical values depend on lot of factors
    - Standard benchmarks
    - E.g. TPC-X benchmarkok a Transaction Processing Performance Council-tól (http://www.tpc.org/)

$$\frac{\text{Finished tasks}}{\text{Time}}[1/s]$$

Metrics 3.

- Waiting time
  - Unit: s
  - The time the task spent waiting from entering the system until leaving it finished
  - Strongly depends on the scheduling algorithm...
  - There are very strong statistical variations:
    - We have to speak about average waiting time, dispersion of waiting time, etc.
    - Standard benchmarks are valuable here too

$$t_{waiting} = t_{ready} + t_{other,non-running} [s]$$

Méréstechnika és Információs Rendszerek Tanszék

Metrics 4.

- Turnaround time
  - Unit: s
  - The time spent in the system by a task
  - Strongly depends on the scheduling algorithm...
  - There are very strong statistical variations:
    - We have to speak about average turnaround time, dispersion of turnaround time, etc.
    - Standard benchmarks are valuable here too

$$t_{CPU,execution} + t_{waiting}$$

Metrics 5.

- Response time
  - Unit: s
  - For on-line, interactive tasks
  - Time spent from the insertion of the task into the system until the first outputs are produced
  - Strongly depends on the scheduling algorithm...
  - There are very strong statistical variations:
    - We have to speak about average response time, dispersion of response time, etc.
    - This is the most valuable metric for users because the user wants to see the output of his/her work ASAP, i.e., he or she does not need to idle, but can advance with the work
    - Standard benchmarks are valuable here too

Metrics 6.

- Energy metrics
  - Unit: for example Ws/task (TPC-Energy)
    - How much energy is used to finish a standard task?
  - Energy consumption is in the center of interest
  - Strongly influenced by the other metrics
    - Computing power, price, energy use?
    - E.g. Compare Intel ATOM to a C2D notebook manufactured? Which one is the better?
  - There are very strong statistical variations here also
  - Energy aware scheduling and benchmarks under development

Requirements for scheduling

- Quantitative properties
- Real-time operation of the scheduler
  - Low overhead…
- Optimization is done using a target function
  - The target function combines multiple metrics into a scalar which can be used to compare algorithms according to complex requirements
  - E.g., the weighted linear combination of metrics may be used
- For evaluation we may use mathematical models, simulation or measurements
  - Reproducible and typical load must be administered to the system (benchmark)
  - There will be statistical variations.
    - Primarily due to speculative execution in the system (cache, etc.)

Qualitative properties

- Expected properties (cannot be quantitative or very hard to make quantitative):
  - Fairness
  - No starvation
  - Predictability, determinism
  - Low overhead
  - Maximum throughput, low waiting times
  - Making decisions based on resource use
    - Using widely used resources (use and let others use it)
    - Using rarely used resources (free)

Requirements 2.

- Hard or soft real-time scheduling of tasks
- Priority
  - Priority ≠ Soft real-time (typical fault)
  - We are going to deal with it soon!
- Graceful degradation
  - If the load of the system reaches knee capacity the system should not collapse, but provide reduced services (due to system overheads).
  - System should loose functionality gradually in case of overload

Other aspects

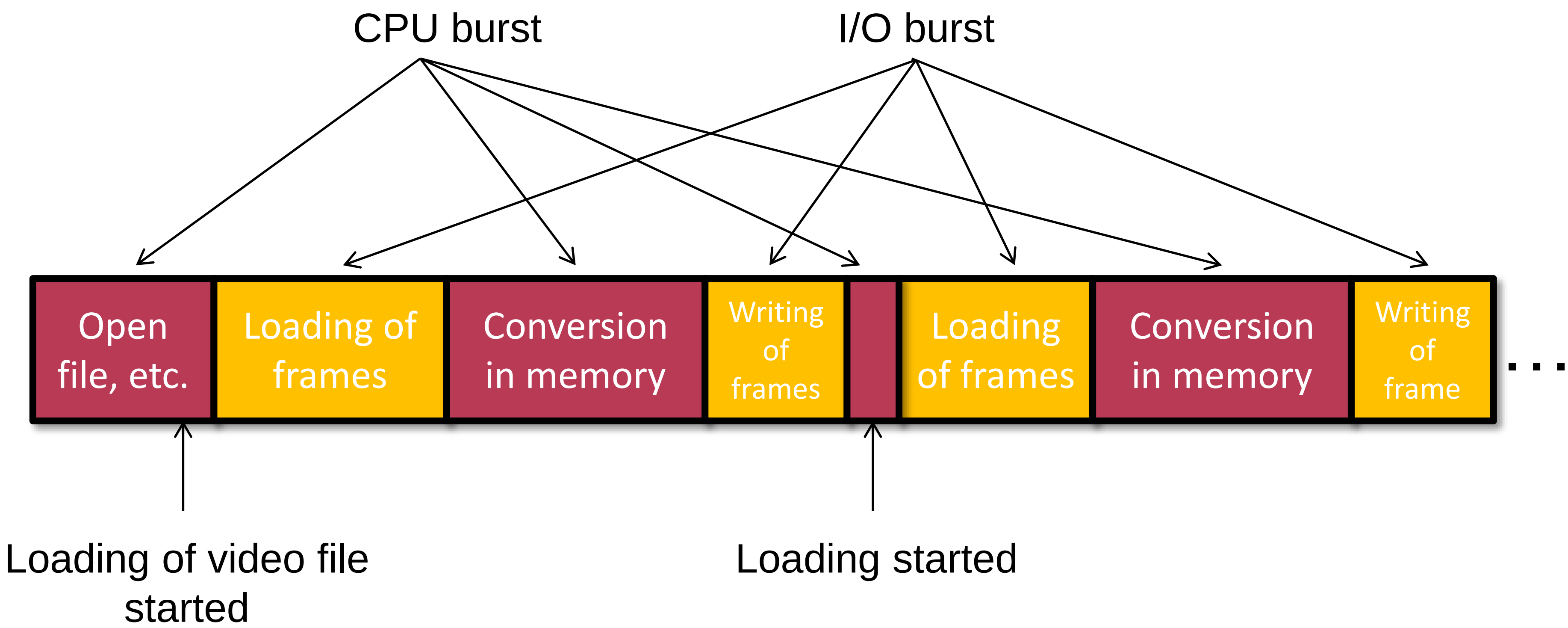
- Static or dynamic scheduling
  - Static:
    - All scheduling decisions are made design time (the task pool must be known in advance)
    - Designing for the worst case
    - We do not deal with it, it is used is mission critical embedded systems
  - Dynamic: The scheduling decision is made run-time
    - We use it typically today in modern operating systems
    - Dynamic resource sharing
    - They are hard to investigate in design-time

Assumptions (1 CPU)

- We assume the followings from now:
  - We have a single execution unit is the system
    - In case of a multiprocessor system it is too complex to handle
  - Only one task can run one time (1 CPU)
  - Ready tasks are stored in a queue (task/job queue)
    - In the special case it is a FIFO, but something else in most of the cases
  - Tasks consist CPU and I/O burst
    - Practical experience shows it, and we have measurements also
    - CPU burst are shoreter than 10 ms typically
    - In between CPU bursts there are I/O bursts to handle I/O
    - The actual distribution of CPU and I/O burst depend on the application

CPU and I/O burst example

- Video file conversion on a machine with 1 CPU and limited amount of memory…



First Input First Out (FIFO, FCFS)

- Simplest algorithm:
  - The queue is a FIFO (First Input First Output) queue storing references to TCBs
    - Using the put() function a task can be inserted to the Ready queue
    - Using the get() function a task can be picked to Run state
    - Also called FCFS (First Come First Serve)
- Non-preemptive
  - If the task does I/O, than a new task is picked (I/O burst schedules)
- The average waiting time can be large, and strongly depends on the CPU and I/O burst of the task
  - It also means long response times, so it is not a good choice for interactive systems
- Administrative overhead is very low

Properties of the FIFO algorithm

- More detailed investigation of the time domain properties:
- Pure CPU burst:
  - Tasks coming later needs to wait for tasks coming earlier
  - A long task can hold up task coming after the long task (convoy effect)
- Both CPU and I/O bursts:
  - If a task has long CPU bursts it can hold up tasks with short CPU bursts
    - The tasks with short CPU bursts will be finished fast
    - The task with the long CPU burst will arrive to the end of the queue fast, and when it is scheduled to run, it will hold up the other tasks
    - Convoy effect as in the previous simple case.

Round-robin algorithm

- It is invented for timesharing systems to correct the problems of FIFO scheduling
- Better for on-line interactive users
  - Better response time than FIFO
  - It reschedules tasks independently of the task (it does it work based on time)
- Preemptive:
  - Practically, it is a FIFO queue augmented with a timer interrupt maximizing the length of the Running state of the task
    - Quantum or time slice
    - 100-1 ms, typically 10-20 ms.
    - Primarily for providing acceptable on line response time for interactive users

Single-shot/one-shot timer

- Before running the task the timer is started, after the predefined time (time slice) it signals an interrupt and the OS can run and make a decision about the running task
  - If the task finishes or execute I/O operation (goes to Waiting state)
    - The OS reschedules the task and the time restarted
  - If the task does not finishes in its time slice
    - Interrupt comes in and the scheduler (OS) runs
    - The Running task goes to Ready state, scheduler runs, pick a new task for running, the timer restarted
- In practice, for the shake of simplicity and for time keeping purposes a periodic timer may be used
  - The implementation is simpler, but the analyses of algorithm is complex and its run-time properties are worse
  - The periodic timer interrupt can be used to implement the system clock, and software timers (that is a must in modern OSs)

Properties of the RR algorithms

- It’s properties depend on the size of the time slice, the statistical properties of the CPU bursts of the tasks, and the length of the time to do a context switch
- Time slice > Average CPU burst → It behaves like the FIFO
  - The length of the CPU bursts define the properties
- Time slice ≈ Average CPU burst → Normal operating region
  - Primarily the time slice defines how long a long task can run in a slot
- Rule of thumb: It is the best if 80% of the CPU bursts are smaller than the time slice
- Average CPU burst >> time slice, which is comparable to the length of context switch:
  - Large administrative overhead
  - Because the time slice is 10-20 ms (sometimes 1 ms), which is several decades bigger than the length of the context switch, it is not a problem today (Was a problem earlier times)

Priority based schedulers

- Scheduler family, there are large number of them
- Priority = urgency (0 is the smallest or the biggest?)
- A priority is assigned to tasks:
  - Internal/External priority:
    - External priority: An operator assigns the priority
    - Internal priority: The OS assigns the priority
  - Static/dynamic priorities:
    - Static priority: A priority is assigned to the task and that does not change during execution
    - Dynamic priority: Priority is changed run-time based on the performance or other properties of the task
  - The combination of it can be also used...

Dynamic priority assignment

- Assigning priority based on measurable task performance:
  - The user may assign an initial priority
  - And after that the OS computes the priority based on the following as example:
    - Time domain properties (response time, etc.),
    - Memory requirements (size, availability, etc.),
    - Used resources,
    - CPU and I/O bursts,
    - Run-time, planned run-time,
    - etc.

Properties of priority based schedulers

- Generic properties:
  - Typically preemptive, but can be non-preemptive also
- Typically not fair, we do not want to be it to be fair (we have priorities)...
- Starvation may happen (Is it an advantage or a disadvantage?):
  - We use it to consider urgance…
  - If starvation happen it is due to overload or design error
  - High priority tasks can be only a small portion of all tasks, that can be run with leaving CPU time for other tasks
  - All tasks cannot be important the same way
  - Tasks may be aged (e.g. increased priority with increasing waiting time)

Priority based schedulers 1.

- Simple priority based scheduler:
  - One task on one priority
  - Used in simple embedded operating systems
- Any number of tasks on a priority level:
  - With modifications it is used all over in modern OSs
    - UNIX, Windows, etc.
  - Modifications:
    - RR scheduler used if multiple tasks are Ready state on a priority level
    - Dynamic internal priority assignment.
- Fundamental problem with the standard priority bases schedulers:
  - Priority inversion, we are going to talk about it later

Priority based schedulers 2.

- Shortest Job First (SJF):
  - Non-preemptive, the task with the shortest estimated CPU burst is selected to Run.
    - How we can estimate future CPU bursts of a task?
    - Most cases it makes the decision based on previous CPU bursts (average of the previous last N bursts, moving average, etc.)
    - The users may know the algorithm, and try to influence it! ☺

Méréstechnika és Információs Rendszerek Tanszék

Priority based schedulers 3.

- Shortest Remaining Time First (SRTF):
  - Preemptive version of SJF
  - If a task becomes Ready the OS reschedules
  - The time of context switch must be taken into account
  - How we can estimate CPU burst?
    - The same problem as we faced at the SJF algorithm

Priority based schedulers 4.

- Highest Response Ratio (HRR):
  - It tries to minimize the possibility of starvation
  - It is based on the SJF algorithm
  - The priority takes into account the waiting time also
  - If a tasks waits more its priority increases...
  - k defines how waiting is taken into account

$$\frac{\text{Burst time} + k * \text{Waiting time}}{\text{Burst time}}$$