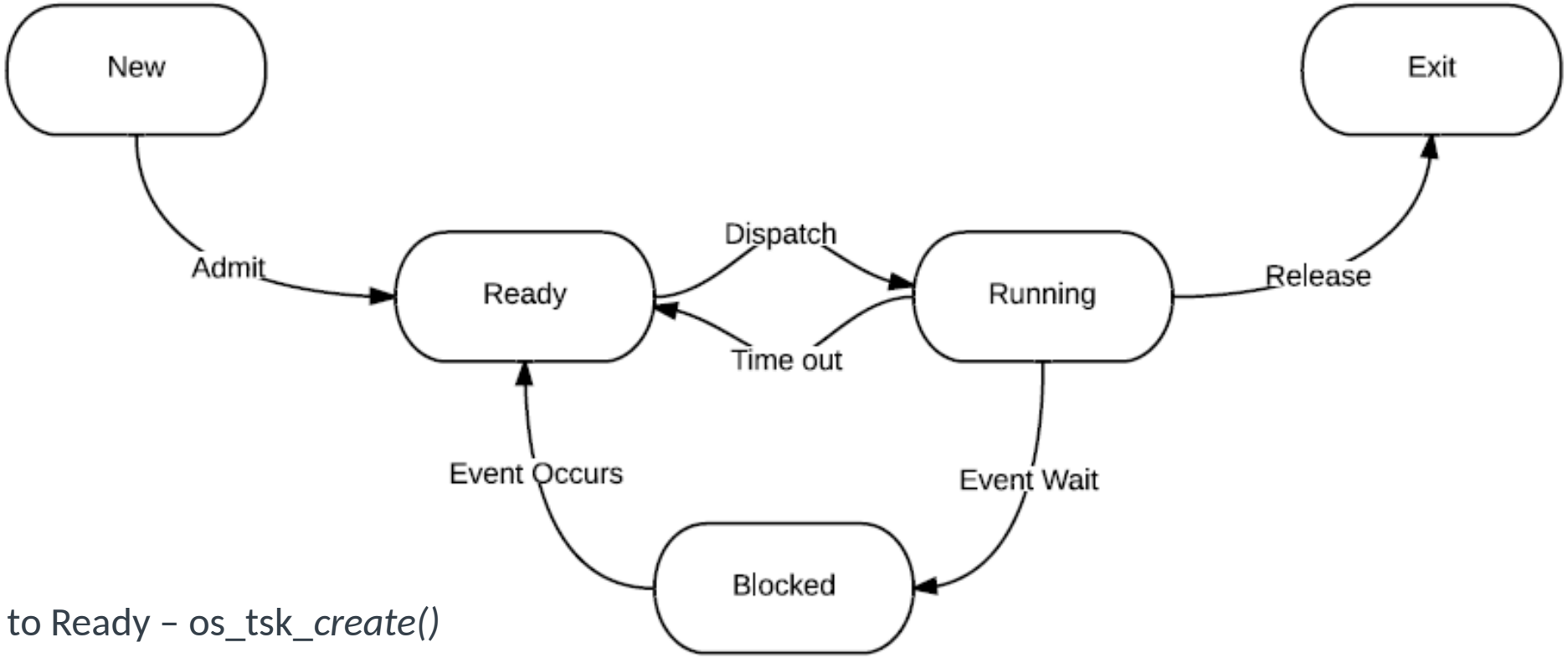
Previous

- Process
- Lab
  - Getting started with hardware using C and assembly
  - Context switching at assembly level
  - Basic RTX task operations

Current Module

- Levels of Scheduling
- Scheduling Algorithms
  - Non-pre-emptive
  - Pre-emptive
- Quantum Size
- Priority
- How to evaluate Performance
- Real Time Scheduling
- Aperiodic RT scheduling

Process State Model



- New to Ready – os_tsk_*create()*
- Running to Exit – os_tsk_*delete()*
- Ready to Running – *rt_dispatch()*
- Running to Ready – *os_tsk_pass()*
- When and How - the job of the scheduler
- To be practiced in lab session

Levels of Scheduling

- Long-term
  - Which processes to add to the ready queue
  - Historically the job of the operator
  - This scheduling level is unfrequently invoked compared to others
  - Establishes the degree of concurrency
  - Large scale computing systems
- Medium-term
  - Available for OSs that support swapping
  - Decides which process to swap out or suspend.
  - Moves tasks from the main memory to the secondary memory or vice versa
  - Uses the seven states model
- Short-term
  - Which ready process(es) to dispatch next
  - Performs context switching

- In this set of material, we will focus on the short-term scheduling. Whenever the term scheduling is used, unless otherwise specified, it refers to short-term scheduling.

What Makes a Good Scheduling Algorithm

- Scheduling criteria
  - Utilisation: computing resources, I/O, network
  - Throughput: completions of processes per time unit
  - Turnaround: the time it takes from New to Exit
  - Response: less delay in the interaction with users
  - Deadline: soft or hard
  - Priority: some processes are more important than the others
  - Fairness: despite the existence of privilege
  - Predictability: less variation in the performance

- A good scheduling algorithm is also application-specific

Behaviour Patterns of Processes

- Processes can be
  - CPU bound - more CPU computations than I/O interactions
  - IO bound – more IO interactions than CPU computations
- Dependency between processes
- Understanding the nature of the processes helps designing an optimized scheduling algorithm

Non-Preemptive Algorithms

Non-preemptive: a process keeps running until it voluntarily gives up or blocks.

Let's consider the single-CPU case

- **First-come, first-served (FCFS)**
  - Also known as first-in-first-out (FIFO)
  - Used for long-term scheduling by old batch systems
  - The ready queue needs to keep track of the task order.
    - `p_first = p_CB->p_lnk;`
    - `p_CB->p_lnk = p_first->p_lnk;`
    - Not really suitable for interactive systems:
      - Poor responsiveness and fairness
      - Unpredictable
      - Tends to starve short I/O-bound processes
      - Convoy Effect
    - Could be used as a sub-component in the queue

Non-Preemptive Algorithms

- **Shortest job first (SJF)**
  - Also known as shortest process next (SPN) or shortest job next (SJN)
  - Dispatches process with shortest execution time
  - To counteract the Convoy Effect, favours short processes
  - Minimizes the average waiting time
  - What if short processes keep coming?
    - Starves long processes
    - Also, how do you predict the execution time of the process?
    - Usually you can't, you can estimate
    - Exponentially weighted averaging

Exponentially Weighted Averaging

- You need to estimate the execution time of the process for some algorithms
- An iterative approach:
  - t_n: measure the n^{th} execution time
  - e_n: the estimate n^{th} execution time
  - w: the exponential weight (0<w<1)

  - $e_{n+1=}\ wt_n + (1-w)e_n$
  - $e_{n+1=}\ wt_n + (1-w)wt_{n-1} + (1-w)^2wt_{n-2} + \ldots (1-w)^iwt_{n-i} \ldots + (1-w)^{n+1}e_0$

  - To simplify the implementation, take only the most recent n history executions into account
  - Causes unpredictability

Preemptive Algorithms

Preemptive: processes can be interrupted by either processes with a higher priority or the task scheduler

- **Round-robin (RR)**
  - A preemptive FCFS variant
  - Needs to keep track of time
  - Once a process runs out of its time slice (or quantum), put it to the end of the Ready queue and dispatch the head of the queue
  - The Ready queue: a circular FCFS queue
  - Starvation-free, time-sharing
  - However, the average waiting time is increased, also frequent context switching results in higher overheads

Quantum Size

- How to pick the quantum size?
  - Large quantum size may starve small processes (degenerate to the FCFS)
  - Small quantum size introduces heavy context switching overhead

- Choose the size based on the needs of interaction
  - How frequently does the user interact with the OS?
  - The quantum size should provide the OS with good responsiveness to the user input

- Choose the size based on the performance (throughput)
  - Theoretical performance modelling, simulation
  - Practical experience
  - A rule of thumb: 80% of CPU bursts should be smaller than the quantum*.

* Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 6

Preemptive Algorithms

- **Shortest Remaining Time First (SRT)**
  - A preemptive SJF variant
  - Dispatch the process with shortest execution time once the current running process is finished or if any new task with shorter execution time is ready
  - You need to estimate the execution time as well

How does priority help?

- Is the shortest process always the most important?
  - On the contrary, usually trivial one
  - The user sets the importance of a process by assigning a priority to it
  - Can also use various types of scheduling algorithms
  - Static priority (remains unchanged throughout the lifecycle), dynamic priority(can be adjusted at runtime)

  - Examples:
    - Multiple queues with different priorities, within the queue you may use FCFS or RR
    - In RR, higher priority processes may have larger quantum

Preemptive Algorithms

- Feedback: dynamic priorities
  - Most modern OSs use a variant of such algorithms
  - For multiple queues (presumably with different priorities), user can configure the following:
    - Number of queues
    - Which queue the process belongs to
    - When a process can be upgraded to a queue with a higher priority or downgraded to a queue with a lower priority
    - Scheduling algorithms for each queue
    - Scheduling algorithms for choosing a queue, e.g. a priority round-robin, 3 time slices for the highest, 2 time slices for normal and 1 for the lowest queue each round
  - Flexible and dynamic

A Feedback Example

- Multiple RR queues
- New processes join the queue with highest priority
- Each time a process is preempted, it will be downgraded and join the end of the queue with the next lower priority (except the lowest queue)
- Scheduler always dispatches the first process in the highest queue that is not empty.

- Problems?
  - Starvation in lower queues.
    - Lower priority has longer quantum

A Practical Problem: How To Evaluate The Performance

- Impossible to make definite comparisons in reality

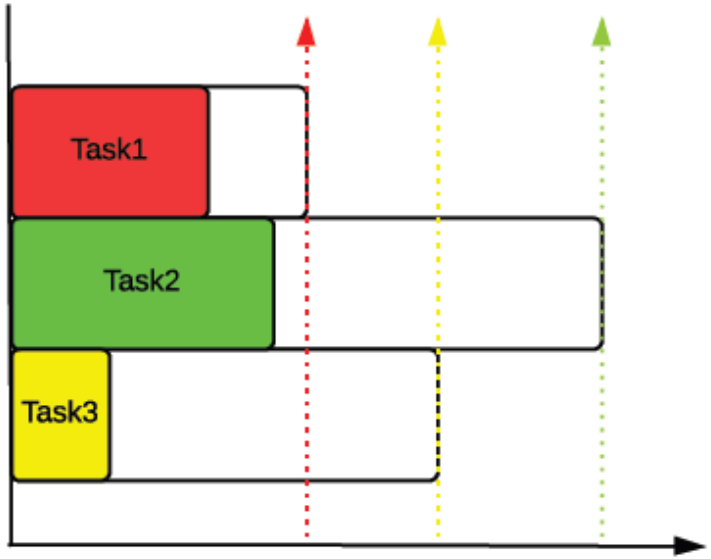
- Either theoretical analysis or some empirical measurements
  - Theoretical analysis – Queuing network, stochastic petri net, process algebra or other modelling tools
  - Empirical measurements – Simulation

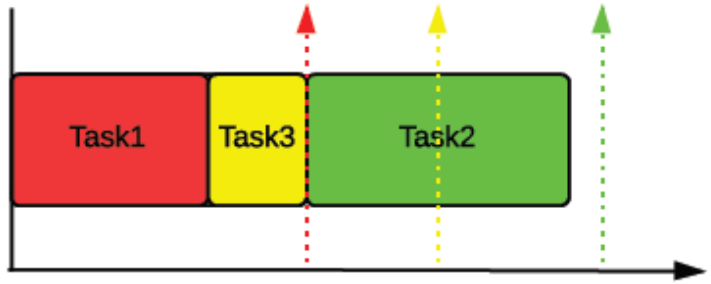
Real-Time Scheduling

- In a RTOS, meeting the deadline is more important than throughput and turn-around time
- Deadline: the maximal computational time allowed
- Needs special scheduling algorithms
- Preemption is usually a standard in RTOS: otherwise scheduler will have to be idle from time to time in case it misses tasks with earlier deadlines that arrive late.
- Two basic categories: aperiodic and periodic
- Also involves the priority and concurrency problems
- Here we look at simple aperiodic cases first
- All the plots in the following sections ignore the context switching time

Aperiodic Real-Time Scheduling

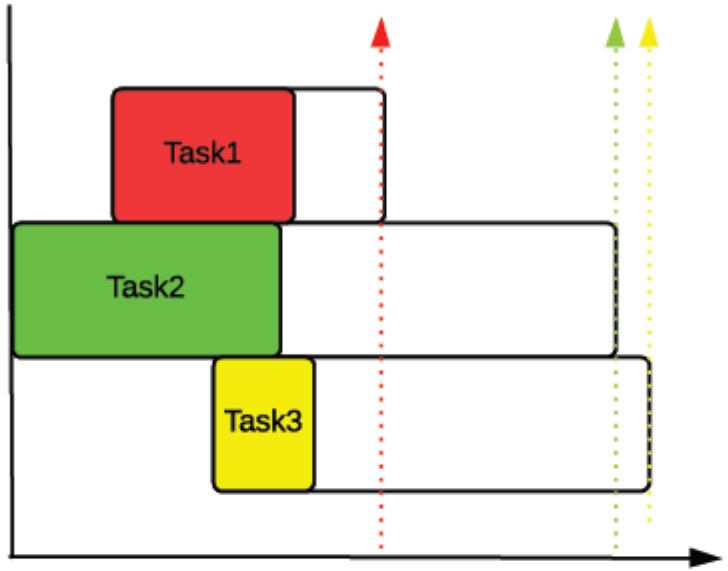
- **Earliest Due Date (EDD)**
  - All tasks arrive at the same time
  - Known deadline and execution time for all tasks
  - Start the task with the earliest due date
  - Non-preemptive
  - Sorting algorithms
  - Optimal

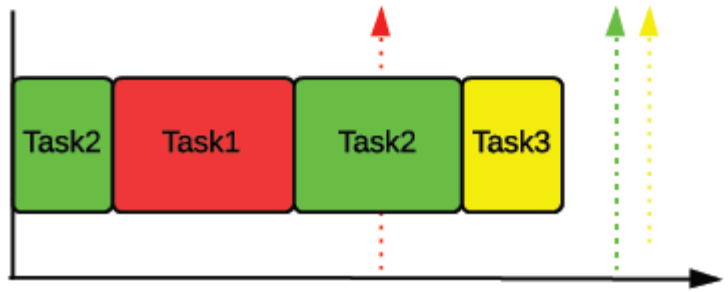




Aperiodic Real-Time Scheduling

- **Earliest Deadline First (EDF)**
  - **Different arrival times**
  - Known deadline and execution time for all tasks
  - Start the task with earliest deadline
  - Preemptive (so overhead!)
  - Sorting algorithms
  - Optimal with respect to minimizing maximum lateness





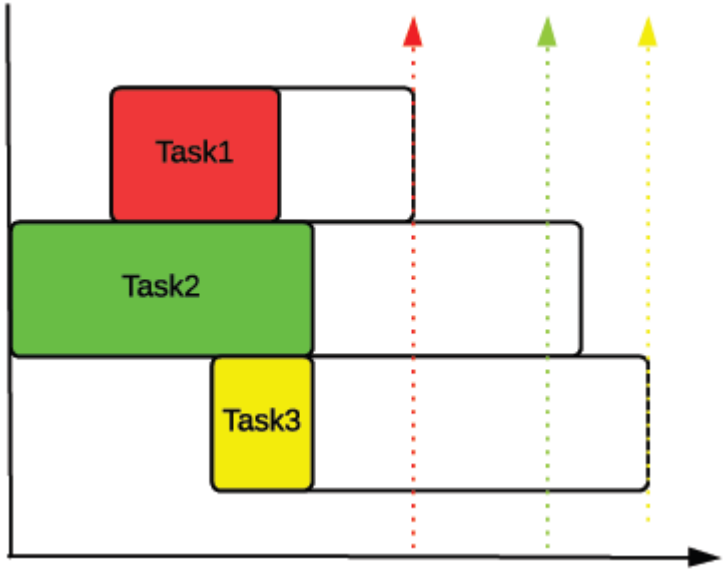
Aperiodic Real-Time Scheduling

- **Least Laxity (LL)**
  - **Different arrival times**
  - Laxity/slack: time to deadline minus (remaining) execution time
  - Known deadline and execution time for all tasks
  - Start the task with least laxity
  - Preemptive (more frequent)
  - Sorting algorithms
  - Dynamic priority: if not dispatched, then the priority will be raised; if it is executing, the priority maintains

Aperiodic Real-Time Scheduling

- An Example

Task	Starting at	Execution Time	Deadline
1	3	5	12
2	0	9	16
3	6	3	19

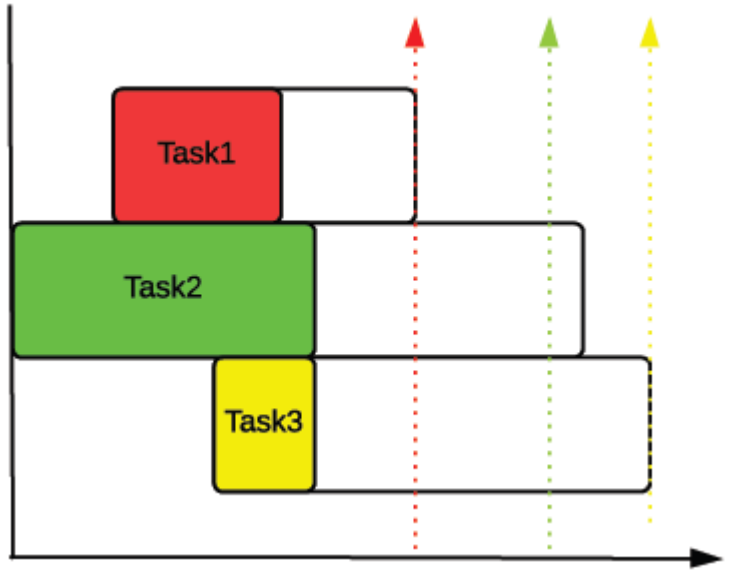


Aperiodic Real-Time Scheduling

- Try to fill in the laxity at each time step, dispatch the least

	0	1	2	3	4	5	6	7	8	9
1	-	-	-	4						
2	7	7	7	7						
3	-	-	-	-						

	10	11	12	13	14	15	16	17	18	19
1										
2										
3										



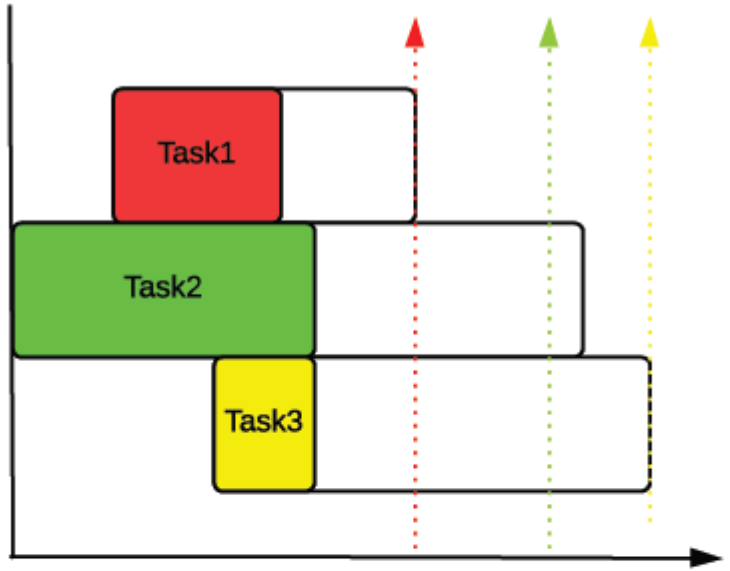
arm

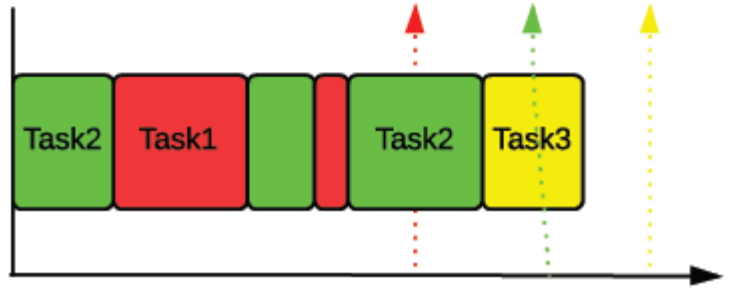
Aperiodic Real-Time Scheduling

- Underscore indicates the least laxity process

	0	1	2	3	4	5	6	7	8	9
1	-	-	-	4	4	4	4	4	3	2
2	7	7	7	7	6	5	4	3	3	3
3	-	-	-	-	-	-	10	9	8	7

	10	11	12	13	14	15	16	17	18	19
1	-	-	-	-	-	-	-	-	-	-
2	2	2	2	2	-	-	-	-	-	-
3	6	5	4	3	2	2	2	-	-	-





- Notice that at time step 7 and time step 9, preemptions happen due to change of priority (laxity)

arm

All that we have discussed so far…

- All of the above is based on an ideal condition where:
  - Context switching time ignored
  - Single CPU: scheduling on multi-core is more challenging
  - “Well-behaved” processes
  - No dependency between processes
  - No periodic processes

- Be aware of the real world…

Next

- Assignment
  - Quantitative analysis and comparisons of various schedulers
- Concurrency