Real-Time Systems

A real-time system (RTS) is any information processing system which has to respond to externally generated input stimuli within a finite and specified period.
Correctness: requires both correct result and timeliness of delivery.
Types of RTS: **hard** (deadlines must be met or disastrous), **soft** (alright to miss deadlines occasionally), **firm** (soft, but results are not useful if deadlines missed). A system may have multiple types of RTS and different cost functions of them missing deadlines.
Some RTS examples: fluid control system, grain roasting plant, process control system, production control system, embedded systems, car control networks, multi-media.

RTS Terminologies

Time-aware: systems work to explicit time schedules; example: alarm clock.
Reactive: something triggers the system, then it must produce a result within a deadline, must constrain **time variability** and control **jitter** for I/O; example: control systems.
Time-triggered: computation is triggered by the passage of time, such as do something at 9 am or every 25 ms (**periodic**).
Event-triggered: computation is triggered by an external or internal event, types: **sporadic** (has minimum arrival interval) and **aperiodic** (vice versa).

RTS Characteristics

Guaranteed response time: confident prediction of **worst-case response times** for systems, need efficiency but must have predictability.
Concurrent control of system components: concurrent entities in the program must operate in parallel in real-world;
Interaction with special purpose hardwares: able to program devices in a reliable and abstract way;
Numerical computation support: able to support the discrete/continuous computation necessary for control system feed-back and feed-forward algorithms;
Also large and complex, and require extreme reliability and safety due to criticality.

Control Loop

A software structure repeatedly read data from sensors, update internal states, produce outputs and write the outputs to actuators (effectors), attempting to timely and correctly respond to time-triggered events.
Environmental rate of change determines the required sampling rate. The bound on variability of sampling timings is the input jitter.
Period of the control loop (itself a **periodic activity**) is usually represented by *T*, and the **input jitter** determines the bound that the period can float (e.g, 20ms +/- 2ms)
Deadline of the control loop is usually represented by *D*, the time elapsed from the start of the loop within which the control loop should finish the operations.

Output jitter constraints the system from producing output too early, producing a lower bound of response time.

Worst-case execution time (WCET) of the control loop is usually represented by C, $C \leq T$ and $C \leq D$. C is determined by the hardware platform, unlike D and T (properties of the application).

Concurrency issues: a control loop may need to access resources shared with other loops, which may involve synchronisation control. To implement mutual exclusion over shared resources, entities such as monitors or semaphores need to be used.

Event-Triggered Control Loops

Event triggered control loops response to events, for which the computations must be completed before deadlines, and the computations may involve reading from inputs and writing to outputs.

A bound (period) on how often the event can be fired will be determined, although this may not always be up to the application designer. Types of bounds:

Sporadic events: event sources with a minimum arrival interval guaranteed by a property of the environment or enforcement by the implementation, easier to analyse.

Aperiodic events: we do not know the minimum arrival interval, but other constraints should apply to the arrival interval for it to be analysable.

Events with both types of bounds could potentially be scheduled, to be described later.

Ravenscar Model

A model of system with the ability to specify periods and deadlines of periodic and sporadic events, as well as protected shared states; along with a cool name.

Each activity is embedded in a **task**/thread, and shared states are embedded in monitors. Not sufficient to model aperiodic events.

Deadline classification: **implicit** ($D = T$), **constrained** ($D < T$) or **unconstrained** ($D > T$ is possible, also described as **arbitrary**).

Cyclic Executives

A method of implementing systems with mostly periodic events. It has a concurrent design with proceduralised code. It is **fully deterministic**.

Procedures are mapped onto a set of **minor cycles** (with *minimum* cycle times) to make up the **major cycle** (with a *maximum* cycle time).

Essentially, tasks are bundled as parts into each minor cycle, which may involve splitting long period tasks. Depending on cycle times and task periods, occurrences of each task in minor cycles may vary, for example: *abc, (*interrupted*), abde, (*interrupted*), abc.*

Advantages: no tasks required at run time, as everything is bundled into a main procedure; sub-procedures share a common address space to allow easy data sharing, and no mutual exclusion required due to the singular structure.

Disadvantages: very inflexible. The minor cycle time must be a common factor of all task periods, and the major cycle time is the absolute maximum period a task can have. Sporadic events are also nearly impossible to incorporate, and the scheduling is expensive and NP-hard. In addition, splitting long tasks may be difficult in programming.

Alternative: we do not actually need determinism which cyclic executives provide, but just predictability. Therefore, having tasks at run-time would be more useful, bringing us to **task-based scheduling**.

Task-Based Scheduling

As a scheduling scheme, **task-based scheduling** provides an algorithm for ordering the use of system resources (CPU in our context), and the means to predict worst-case behaviour of the system under the scheduling algorithm.

In this scheme, tasks exist at run-time, supported by the OS or software architecture.

Each task will have the followed states: **runnable** (ready-to-run or running), **suspended waiting for a timing events**, and **suspended waiting for a non-timing event.** Shared data must be protected.

There are several task-based scheduling approaches discussed during this module: **Fixed-Priority Scheduling** (FPS), **Earliest Deadline First** (EDF), **Least Laxity** (LL) and **Value-Based Scheduling** (VBS).

Characteristics of tests: **sufficient** (pass this test and deadlines will be definitely met), **necessary** (fail this test and deadlines will definitely *not* be met), **exact** (both sufficient and necessary) and **sustainable** (system stays **schedulable** if conditions "improve").

Fixed Priority Scheduling (FPS)

The most widely used approach.

Each task has a **fixed** and **static priority** which is precomputed, and based on the task's temporal requirements (not its significance to the system).

Runnable tasks are executed in priority order.

Earliest Deadline First (EDF)

Execute runnable tasks in the order of absolute deadlines (*D*), the task with the closest deadline is the next task to execute. As the absolute deadlines are computed at runtime, the approach uses **dynamic priority**.

Least Laxity (LL)

Execute the task in the order of laxity, which means that tasks with the least work to do will always be executed first. This approach computes laxities at runtime, therefore it also uses dynamic priority.

Value-Based Scheduling (VBS)

An adaptive scheme which assigns a value to each task, and employ a live value-based scheduling algorithm to determine the next task to execute.

Preemption

For any scheduling approaches with priority assignments, a high priority task may require to be executed during the execution of a low priority task. Therefore, the schedule approach/scheme may be **preemptive** (switch to high-priority task immediately, usually preferred) or **non-preemptive** (finish the low-priority task first); alternatively, a half-way approach such as **cooperative dispatching** (deferred preemption) may be used.

Simple Task Model

The model assumes that an application has a fixed set of periodic tasks with known periods, which are completely independent of each other. We ignore all overheads and context-switching times. Each task has a deadline equal to its period, with a fixed worst-case *C* (WCET*)*. All tasks are executed on a single processor with equal criticality.

Priority assignment**: Rate Monotonic Priority Assignment (RMPA)**: Assign each task with a unique priority based on its period — the shorter the period, the higher the priority, with priority 1 being the lowest priority. It is optimal — if any task set can be scheduled using preemptive priority-based scheduling with fixed priorities (such as preemptive FPS), then the task set can also be scheduled with a RMPA.

$$\prod_{i=1}^{N}\left(\frac{C_i}{T_i}+1\right) \le 2$$

<u>Priority assignment in simpler words</u>: rank tasks by their period, start from the longest task, give it priority 1, and for each immediately shorter task, increment the priority by 1.

$$U \equiv \sum_{i=1}^{N}\frac{C_i}{T_i} \le N(2^{1/N}-1)$$

Utilisation-Based Analysis: sufficient but not necessary (fail this test and the system might still be schedulable. For each task, divide computation time *C* by period *T*, and check that the sum of these quotients from division are smaller than or equal to the RHS. Asymptotic: U -> 0.69 as N -> Infinity. This test is not exact, not general, and <u>sufficient but not necessary</u>, but can be done in O(n).

We can also test families of tasks or use an alternative formula as shown, but they have the same limitations as stated above.

Critical Instant: the starting condition that leads to the worst case behaviour — when all tasks are released at time 0 (or together in the general sense).

Response-Time Analysis

We calculate each task's **worst case response time** *R* in the order of lowering priority, and check each *R* against its deadline *D*. If at any point a task has an *R* that is longer than its deadline *D*, then the entire system is said to be **unschedulable**. This test is both <u>sufficient and necessary</u>.

$$w_i^{n+1} = C_i + \sum_{j\in hp(i)}\left\lceil \frac{w_i^n}{T_j}\right\rceil C_j$$

In practise, first calculate the weight of the highest priority task, which suffers no interference so has a converged weight equal to its computation time *C*;

Then, we move on to each of the next highest priority tasks. Its weight is the sum of its own *C* and interference from each task with a higher priority.

For the second highest, the new weight (RHS) is its 'own *C*' plus the *C* of the one higher priority task interfering, multiplied by the celling of previous weight (calculated from the highest priority task) divided by the period of that interfering task. Repeat this for the second highest priority task with the new weight, until we get the same weight to the previous (convergence), and that weight *W* becomes this second highest priority task's *R*.

The third highest priority task will have two interfering tasks added to its *C*, and so on; until we reach the lowest priority task.

If each task has an *R* < *D*, then the system is schedulable; otherwise unschedulable.

Advanced Task Model

While the simple task model represents an ideal scenario, advanced task model is more realistic, in which we consider **sporadic** and **aperiodic tasks**, **overheads**, **release jitter**, **offsets**, **task criticality** / **value** and so on, as explained in the rest of the sections. However, the same assertion is that <u>all tasks have a fixed worst-case execution time</u>.

Sporadic Tasks

These tasks represent sporadic events (as introduced previously) in a task model, which means that they all have a minimum inter-arrival time, and have *D* < *T*. The response-time analysis algorithm above works for all *D* < *T* sporadic tasks, with a stopping criteria of $W_i^{n+1} > D_i$ instead of convergence, and with any priority ordering.

<u>Use of worst case response times (WCRT)</u>: due to sporadic WCRT's needs of coping with burst interrupts and abnormal sensor readings, WCRT figures may be much higher than average response times. In an average case, scheduling solely based on WCRT may result in low processor utilisations. Therefore, it may be advantageous to consider *hard* and *soft* sporadic tasks separately.

Scheduling all tasks with average execution time and arrival rates may result in tasks missing deadlines (known as **transient overload**), while scheduling all tasks with worst case time and rates may result in unacceptably low processor utilisations — which may also indicate overly conservative estimation of WCRT. A balance between worst case and average case scheduling could be reached for hard tasks and soft tasks separately.

Aperiodic Tasks

Representing aperiodic events, these tasks do not have a minimum inter-arrival times, therefore no guarantee of schedulability possible. A solution may be to run aperiodic tasks at a priority lower than hard periodic and sporadic tasks, to prevent interference of the latter's schedulability, which however may cause soft tasks to miss their deadlines. A better solution is to use an **execution-time server**.

Execution-Time Servers

Operating at run-time, these servers provide a capacity/budget *C* to their client tasks (usually aperiodic tasks), which is used when a client task runs. A replenishment policy refills *C* in a certain way, and if *C* is temporarily depleted, the client task cannot run so that it would not affect the execution of other tasks.

Types of execution-time servers:

Periodic server (PS): starting at a point such as time = 0, at the start of each fixed replenishment period *T* (e.g. 0 to *T*, *T* to 2*T*), a task has budget *C* and can run while *C* is not depleted. *C* is always replenished in full at the start of every *T*, and idles away if not used.

Deferrable server (DS): similar to PS, but a client task does not have to use up its budget at the start of a replenishment period *T*. A fixed budget *C* is available for the task to run for up to *C* amount of time during *T*. Comparing with PS, DS is **bandwidth preserving**, allowing C to be retained for as long as possible.

Sporadic server (SS): originally used to enforce minimum separation for sporadic tasks. Each SS client also has a budget *C* and a replenishment period *T*, but with a different suspension criteria: execution requests when current consumption below *C* is always accepted, and requests when consumption between *C* and 2*C* are fulfilled in two stages, with *C* amount of time provided immediately, and the rest provided at replenishment time *t*+*T*. SS is also bandwidth preserving.

Choice of Priority Ordering and Deadline Monotonic Priority Ordering

Rate Monotonic Priority Ordering (RMPO), based on higher priorities for shorter periods — as introduced earlier — is optimal when a task set has *D* = *T*.

When *D* < *T* however, a task set has optimal ordering with **Deadline Monotonic Priority Ordering** (DMPO), which is defined as $D_i < D_j \Rightarrow P_i > P_j$, i.e. the shorter the deadline a task has, the higher the priority it is assigned.

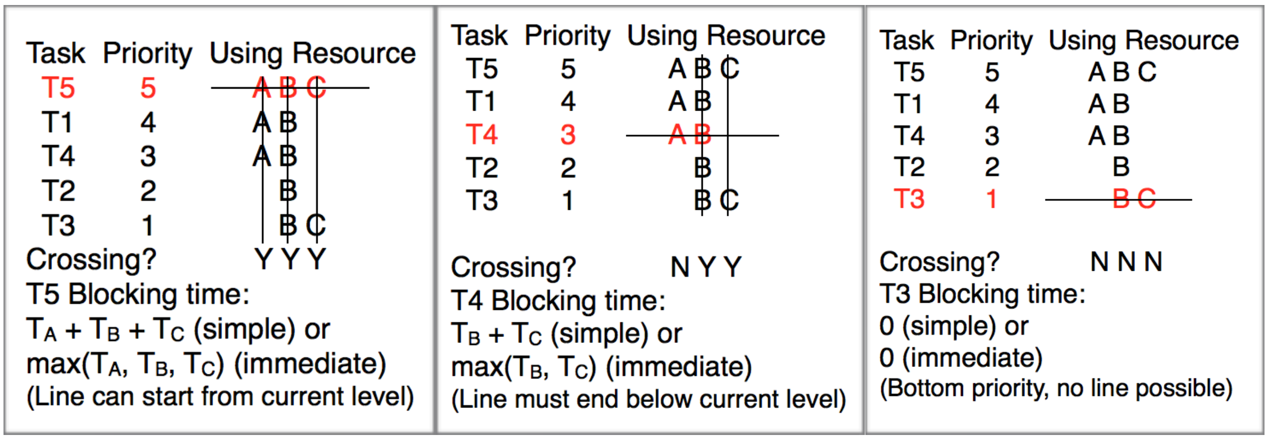
To prove that DMPO is optimal for all task sets with *D* < *T*, we need to be able to transform any priority scheme for a task set into DMPO, showing that the ordering by

DMPO is schedulable in all cases. This requires each step of the transformation to preserve schedulability.
In plain words, for two adjacent-priority tasks in a task set with a currently schedulable priority ordering, if the higher priority task has a later deadline than the lower priority task, we want to swap the two tasks for them to be under DMPO.
If we swap the two tasks' priorities, other tasks with higher or lower priorities than these two will obviously not be affected, due to no change in interference received. The swapped task that now has the higher priority would obviously not be affected as well, since it now receives less interference, which leaves us the other swapped task, which now has a lower priority.
We now need to show that this lower priority task will also always be schedulable after swapping. In the ordering scheme before our swap, the now higher priority task had an earlier deadline than the now lower-priority task, which is why we needed to swap them for DMPO. Since the original task set was schedulable, both tasks have $R < D$ and $D \leq T$, and the previously higher priority task could only have been released once during a response of the previously lower priority task (otherwise unschedulable considering other adjacent tasks), causing at most one interference. In the new DMPO ordering, this implies that the original WCRT R of the now higher priority task becomes the R of the now lower priority task (as it can run without interfering the now higher priority task), still $R < D$, and the combined response time of two remains the same. The system remains schedulable.
The same process can be repeated to move the priorities to a DMPO ordering.

Priority Inversion
In a simple RMPO or DMPO priority ordering, an problem may arise in which a higher priority task is suspended waiting (**blocked**) for a long-running lower-priority task to finish, potentially preventing it from being scheduled. This problem is described as **priority inversion.**
We are able to determine the response time of tasks under priority inversion, which implies that a resource required by both higher and lower priority tasks is blocked by the lower priority task. The calculation is similar to that used for simple response-time analysis.

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

To mitigate, **priority inheritance** may be used (higher priority copied to the blocking task), which may also involve a **priority celling protocol** (PCO). Two PCOs have been described: OCPP and ICPP, which are described below. All PCOs on single processors guarantee mutual exclusion and prevent deadlocks and transitive blocking. However, a good practise is to keep all critical sections small.
Note: for the blocking time B_i in the formula, make a table of tasks ordered by their priority (most important first), write out the resources they require, as explained below:



Task	Priority	Using Resource
T5	5	~~A B C~~
T1	4	A B
T4	3	A B
T2	2	B
T3	1	B C

Crossing? Y Y Y
T5 Blocking time:
$T_A + T_B + T_C$ (simple) or
$\max(T_A, T_B, T_C)$ (immediate)
(Line can start from current level)

Task	Priority	Using Resource
T5	5	A B C
T1	4	A B
T4	3	~~A B~~
T2	2	B
T3	1	B C

Crossing? N Y Y
T4 Blocking time:
$T_B + T_C$ (simple) or
$\max(T_B, T_C)$ (immediate)
(Line must end below current level)

Task	Priority	Using Resource
T5	5	A B C
T1	4	A B
T4	3	A B
T2	2	B
T3	1	~~B C~~

Crossing? N N N
T3 Blocking time:
0 (simple) or
0 (immediate)
(Bottom priority, no line possible)

Original Priority Celling Protocol (OCPP)
Each task is assigned a static priority (e.g. through DMPO) and a dynamic priority, which is determined at run time as the maximum of its static priority and the inherited priority from any higher priority task it blocks.
Each resource has a static celling value, which is the highest priority of all tasks using the resource. For all celling values at a point in runtime, excluding the celling values of resources that a task is currently locking, the task may only lock a resource if its dynamic priority is higher than the highest of these celling values.

Immediate Priority Celling Protocol (ICPP)
A simplified version of OCPP, ICPP also has a static and a dynamic priority assigned to each task, as well as a static celling value assigned in the same way to each resource. However, the dynamic priority of a task is instead the maximum of its static priority and celling values of resources it is already blocking. Therefore, we do not need to consider resources not currently blocked by a task in ICPP. The check criteria of a task's ability to lock a resource remains the same.
In ICPP, a task will only suffer a block at the very beginning of its execution. Once the task starts actually executing, all the resources it needs must be free; if they were not, then some task would have an equal or higher priority and the task's execution would be postponed.
Compared to OCPP, ICPP has an identical worst-case behaviour, but is easier to implement (no need to monitor blocking relationships), leads to fewer context switches as blocking always happen before start of a task's execution. However, ICPP also requires more priority movements.

Dealing with Uncertain Releases and Deadlines
Release Jitter: the release time of sporadic tasks may not be exact, which may affect the schedulability of the system. To take this into account, add the jitter of each task to its previous weight when calculating WCRT for each task. For a task that may release later but never early, under a deferrable server (DS) this jitter can represent the worst case behaviour of a task (releasing at the end of replenishment period and immediately released again when replenished, aperiodically ran for 2C at a time).
Busy period: an abstraction of worst case response-time analysis at each priority level. Starting at at time 0, the busy period is the length of the interval (starting at 0) during which the processor is busy executing tasks with priority *i* or more for each priority level *i*. During this, later-ran tasks are first blocked (insufficient dynamic priority for the celling value) then interfered (insufficient priority) before being allowed to execute.
Arbitrary deadline: for pipelined tasks, we need an assured deadline to schedule the consecutive tasks, which may require a buffer in between. However, if the deadline of a predecessor task is arbitrary, the task set may still be scheduled if that task's average response time is smaller than its period; but the task can interfere with itself in worst case. To take into account staggered release, for each release count *q*, calculate $R_i(q) = w_i^n(q) - qT_i$, and then $w_i^{n+1}(q)$, stopping when $R_i(q) \leq T_i$ is no longer true. If a release jitter applies, add a jitter to $R_i(q)$ and $W_i^{n+1}(q)$.

$$w_i^{n+1}(q) = B_i + (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j$$

Cooperative Scheduling

For safety-critical systems, arbitrary preemption may not be acceptable. An alternative is to split tasks into slots to avoid preemption. In addition to providing mutual exclusion, this also increases the schedulability of the system with lower *C* values. Response-time analysis will take into account the execution time of the final slot, F_i, and $R_i = w_i^n + F_i$ at convergence. The last slot of execution is also safe from interference. However, under certain circumstances this would affect the separation of busy periods, causing incorrect response-time analysis.

$$w_i^{n+1} = B_{MAX} + C_i - F_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

Fault Tolerance

Fault tolerance allows a certain level of faults to occur and the system to recover from these faults with extra computation time, while retaining schedulability. The level of faults tolerable is described by a **fault model**. If *F* is the max number of faults allowed for a level, then the product of *F* and the max fault recovery time of higher or equal priority tasks is added to the response-time analysis equation. If instead there is a minimum arrival interval of the fault, then the response time can be calculated as in this formula.

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} \left(\left\lceil \frac{R_i}{T_f} \right\rceil C_k^f \right)$$

Offsets and Non-Optimal Analysis

Tasks may not all start at a critical instant such as time $t = 0$, and may have differing constant offsets from the critical instant. Response-time analysis is unaffected as long as the task set can be appropriately transformed,

<u>In practise</u>: to replace two tasks with the same period and an offset in between with a single notional process with half the period, the higher priority, the shorter deadline and the longer computation time of the two, and carry on with the response-time analysis as normal. If the notional process is schedulable then its represented tasks are schedulable, and assert the same interferences on lower priority tasks.

Audsley's Algorithm

If task p is assigned the lowest priority and is feasible then, if a feasible priority ordering exists for the complete task set, an ordering exists with task p assigned the lowest priority. In other words, if a task is schedulable at the the lowest priority determined for the task set, we can assign the lowest priority to that task, and all other tasks can hence be assigned a priority. This algorithm also implies that orderings of other priority tasks are not significant when determining the response time of a task.

Before applying Audsley's algorithm to assign priority, we can initially order tasks by importance, or start with DMPO to reduce time of test.

Priority Insufficiency

If we do not have a sufficient number of priority levels, tasks must share priority. Two tasks sharing a priority are assumed to interfere with each other. To simplify this, we can pack tasks together for response-time analysis.

Power-Aware Systems

Sometimes we do not want to get every task done as soon as possible, but rather as late as possible while still meeting the deadlines. For example, a variable clock rate processor

that runs slower at lower load can extend the battery life of a mobile device. Task scheduling can be adapted for this purpose.
Sensitivity analysis will, for a schedulable system, enable the maximum scaling factor for all C values to be ascertained – with the system remaining schedulable.

Overheads in Task Model
In practise, task processing will generate overhead in three main areas: context switches (one per new task), interrupts (one per sporadic task release) and real-time clock overheads.
Each context switch generate a cost switching to the task and a cost switching away from the task; the cost of interrupt handling is added up from costs of individual interrupts; and clock interrupts to move tasks between queues with elapsed time will generate a cost. The full model is as shown below.

$$R_i = CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j) + \sum_{k \in \Gamma_s} \left\lceil \frac{R_i}{T_k} \right\rceil IH + \left\lceil \frac{R_i}{T_{clk}} \right\rceil CT_c + \sum_{g \in \Gamma_p} \left\lceil \frac{R_i}{T_g} \right\rceil CT_s$$

Earliest Deadline First (EDF) Scheduling
As described earlier, in EDF scheduling, the task with the earliest absolute deadline will always be run next.
Advantages compared to fixed priority scheduling (FPS): EDF has a simpler utilisation-based test: sum up the quotient of computation time with period of all tasks in the task set, and if the sum does not exceed 1, the system is schedulable. Period can also be replaced by deadline for $D < T$ task sets. For a single processor, EDF always schedules better and usually supports high utilisations.
Disadvantages compared to FPS: EDF is more difficult to implement, as it requires a run-time system to determine dynamic priorities for the task, generating significant overhead. EDF is also more susceptible to other influencing factors, such as switching overheads.
In overload situations, FPS is more predictable and the first tasks to miss deadlines would be lower priority ones; while EDF is susceptible to domino effects in missing deadlines.
Note: Least Laxity (LL) scheduling also has the same utilisation-test as EDF's, but suffers from high overheads.

Processor Demand Analysis (PDA)
PDA is a necessary and sufficient test for EDF, it checks whether the system load at all task deadline points is lower than the maximum schedulable load.
First calculate the L_b bound for PDA by converging the first formula; then for each deadline (from any task, one task may have multiple deadlines) before Lb, calculate the system load at that point with the second formula, the floor function of which has a minimum of 0 for $D > T$ tasks. If for all $t > 0$, $h(t) \leq t$ then the system is schedulable.
Note: there is also an upper bound L_a involved, but in all circumstances seen in this module, $L_b < L_a$ therefore L_a has not been used much, for details of L_a please see course materials.

$$w^0 = \sum_{i=1}^{N} C_i$$

$$w^{j+1} = \sum_{i=1}^{N} \left\lceil \frac{w^j}{T_i} \right\rceil C_i$$

$$h(t) = \sum_{i=1}^{N} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor_0 C_i$$

Quick Processor Demand Analysis (QPA)
A simplified and faster version of PDA. In QPA, we start at bound L (calculated in the same way as above), and work backwards towards 0. If at any time *t*, h(*t*) > *t* then the system is not schedulable. Otherwise, if h(*t*) = *t*, decrement *t*. If *t* is smaller than the lowest deadline, then the task set is schedulable. QPA is also necessary and sufficient.

Blocking and Servers for Aperiodic Tasks in EDF
In EDF, if we need to decide preemption level for access to shared resource or urgency for access to processor, a static assignment is used for preemption level, and dynamic priority is used for urgency. A task may preempt another if it has a closer deadline (more urgent) or greater resource preemption level. This also guarantees mutual exclusion.
Optimally, we use DMPO to order the preemption level of tasks.
Servers: in addition to PS, DS and SS that are available to FPS, EDF-specific servers are also available for EDF. Aperiodic tasks can be assigned a deadline for scheduling purposes through calculation of computation time request, aperiodic budget and replenishment period.

Analysis Method for Worst Case Execution Time (WCET)
WCET is known as worst case response time (WCRT) as previously mentioned.
Due to the substantial size of possible execution paths and limitations of test coverage, it is not always possible to measure WCET directly; but WCET can also be estimated through analysis, producing a close-to-reliable result. The result should be an upper-bound, application and hardware-dependent. The result must be safe and tight to be practically useful. A conservative prediction is preferred.
The real worst-case points are never seen in testing, and mostly never occurs in operation, therefore the determination of possible worst case is an approximate.
Basic block: a basic action which is part of possible sequences of actions of the task (execution path) in the given application, can be represented as a directed graph. The duration of each action occurrence on each possible path is also calculated.
Simple analysis: simply sum up the duration of each basic block to get computation time.
Complex analysis: in practise, more factors need to be taken into account, such as pipelining, caches and parallelism, making a full analysis difficult.
Subproblems: find description of possible execution paths in source code representation, then translate path information into machine language representation, finally conduct execution-time analysis, which include computations of WCET bound.
Cost of basic blocks may be dependent on each other, therefore program and machine analysis cannot be separated. Compilers may also alter paths and contents of basic blocks, causing discrepancies between two analysis stages.
It is necessary to know bounds for loops, and information such as input constraints from the user are helpful. It is impossible to automate analysis of the program alone.
Execution-time modelling (ETM): map a sequence of instructions to an execution time, which may vary due to different values in input parameters (requires determination of max) and internal state of the processor (requires foot printing of execution history). Other processor hardware features may also influence the execution time, such as branch prediction. ETM is typically done before WCET calculation in three phases: cache analysis (predictions of cache hit/miss), pipeline analysis (sequential combination of reservation tables) and path analysis.
Overall, WCET analysis is hard, and we need to combine techniques to be accurate.

Global Multiprocessor Scheduling

Scheduling tasks to be ran globally across multiple processing units present unique challenges. When a task consumes significantly more computing time on others, the task set may become unschedulable without splitting a task, known as **Dhall effect**. If each task is split into pieces to be ran on different processors, issues with priority ordering and access to shared resources may arise.

Priority ordering for multiprocessor: EDF and DMPO cannot achieve an optimal ordering on multiprocessor systems, there are better ordering schemes available, such as TkC, DkC under $D < T$. Other schemes include *pfair*, theoretically schedule up to a total utilisation of M (for M processors) but with excessive overhead; and EDZL (EDF until zero laxity, then execute task to completion) with a utilisation bound of $U \leq 0.63M$.

A combined approach with better average performance uses FPS to tasks with U greater than 0.5 and EDF for the rest, with $U \leq (M + 1) / 2$.

Response-time analysis for multiprocessor: not easy, as the most exact analysis does not have an optimal priority ordering, while for a less exact analysis we can use Audsley's algorithm to obtain an optimal priority ordering.

Overheads: costs of task migration and sharing run-queues are significant for multiprocessor systems.

An alternative approach of utilising multiple processors is via **partitioned systems**.

Partitioned Systems

A different method of utilising a multiprocessor system is to first allocate *entire* tasks to processors, then run single processor scheduling on individual processors. However, the allocation is reducible to the bin packing problem, the allocation is NP-hard. A good heuristic method is largest density first on a first-fit basis, attempting to decrease the computation time to deadline (C/D) ratio. Different first-fit utilisation bounds apply for EDF and other scheduling methods, when $T = D$.

Semi-partitioning: a variation of fully partitioned systems, allowing a small number of tasks to migrate at run time, also known as **task-splitting**.

Schemes exist for both FPS ($0.69M$ bound) and EDF ($>0.9M$ bound) task-split scheduling, both with a maximum of M-1 task migrations (e.g. 3 migrations for 4 processors). A task to be split has the highest non-preemptive priority for a limited time of its execution on one processor, then migrated to another processor to be finished before deadline, and returned to the original processor afterwords.

Semi-partitioning in practise: when we split a task (e.g. a into a_1 and a_2), both a_1 and a_2 should keep a's period T, but have split execution time C and corresponding deadline D.

Resource protection in partitioned systems: mutually-exclusive access may be required on resources shared between tasks or split tasks (when semi-partitioned). An appropriate celling protocol such as adapted ICPP must be used, but unbounded blocking could occur. Therefore either a first-in first-out queue needs to be used, or resources accesses can be made non-preemptive — this could cause high priority tasks to miss deadlines.

Deadlocks in partitioned systems: on single processor, a priority celling protocol such as IPCP can prevent deadlock. Deadlocks can also occur on multiprocessor systems, requiring banning nested resources or having a static ordering on resources (also a solution to the dining philosopher problem).

Mixed Criticality Systems

In a more practical scenario, a hardware platform may host more than one system. In a safety-critical setting, these systems must not interfere with each other, and especially the ability to meet deadlines by more critical systems. **Criticality** defines this property for each system. An appropriate verification process is conducted for each system based on its criticality, which may involve formal proof of correctness for highest level (e.g.

conservative static analysis), extensive testing for middle level (e.g. static path analysis and basic block measurement) and adequate testing for low level (e.g. simple measurements with margins).

To take into account the criticality of a system in timing / schedulability analysis, we assign a criticality level *L* for each task, and each *L* carries its own worst-case execution time *C*(*L*). For the same task, the higher *L* is, the higher its *C*(*L*) will be. In a two-criticality (high and low) system, all high-criticality tasks must be schedulable when their *C*(high) values are used, and vice versa for low-criticality. In general, we have three approaches:

Method 1 (**Vestal**): For each task *i*, its response time *R* is its C value corresponding to its criticality level, plus the sum of all higher priority tasks' celling *R/T* values multiplied by their *C* values of *i*'s criticality level — not their own criticality levels. The theory of Method 1 is simple, but as it requires analysing high criticality *C* values for all low priority tasks, it is expensive to compute.

Method 2: Only allow a task to execute up to the *C* value corresponding to its criticality (e.g. *C*(low) time only for low-criticality tasks). This requires run-time support, but avoiding Method 1's drawback. In analysing higher priority tasks, we only use the C value corresponding to each one's criticality level.

Method 3: While Method 1's drawback means that we should not analyse high criticality *C* values for low priority tasks, it is not too expensive to analyse low-criticality *C* values for high criticality tasks, as they will be smaller than high criticality *C* values. Therefore, we can divide the system into two operation modes: *low-crit* mode (all execution times are at or below *C*(low) values) and *high-crit* mode (a task requires more than its *C*(low) to execute, must now abandon all low criticality tasks). In calculation, from the lowest criticality level mode to the highest criticality mode, we analyse R values individually; at each criticality mode, we use the C value that is either for the corresponding criticality level, or for the highest criticality used in that mode, whichever is lower. Also if the celling is going to be "busted" to the next integer, we do not proceed any further and use the *R* calculated at that point, since lower criticality tasks will be abandoned. If the celling will not be busted, we stop at convergence as normal. Finally we assign the *R* value of a task's corresponding criticality level as the final value.

It is worth nothing that **criticality is not priority**: under a common priority ordering scheme such as DMPO, a short deadline task on low criticality might have the highest priority, thus having no higher priority tasks and calculated first. In fact, DMPO might cause schedulable mixed criticality task sets to appear unschedulable in response time analysis, and since relative priorities of higher priority tasks is not important, Audley's algorithm can be used to assign optimal priorities.

Open Systems

In the case of dynamic soft real-time applications, we will not always know arrival patterns and computation times in advance. Therefore, most of the analyse must be done on-line. Hardware support is required, and sometimes we may need to abandon lower priority tasks.

Online analysis are efficient but potentially non-optimal analysis for scheduling schemes used to manage any overload that is likely to occur due to the dynamics of the system's environment. This may involve an admission control to limit the number of tasks competing for processor resources and an EDF dispatching routine for tasks admitted. This prevents EDF thrashing due to bad performance in transient overloads.

For online analysis, a **value** must be assigned to each task indicating their relative importance, which can be classified as static (always the same), dynamic (only computed at task release) or adaptive (can change during the task's execution). Assignment must be careful and use well-studied techniques to prevent issues.