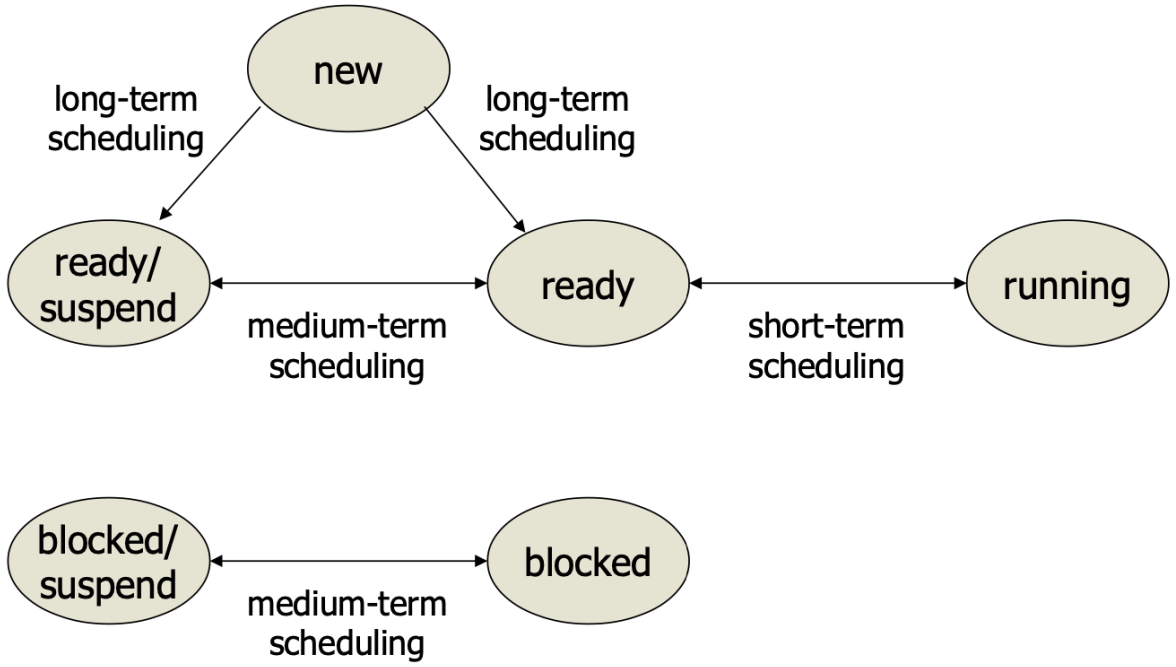
General Scheduling

Scheduling → attività di selezionare quale processo deve andare ad eseguire in ogni istante di tempo

- **Long term scheduling →** evito di creare un processo se non è necessario, deve passare un ADMISSION TEST (ammette un task nel sistema)
- **Medium term scheduling →** decide se un processo deve essere swappato in o out.
- **Short term scheduling →** quale processo viene eseguito dopo. Ci focalizziamo qua.
  - **Non-Preemtive** → un task continua fino a che non ha finito, ma causa bloccaggi ad altri task magari con meno priorità ma più corti. (First In First Out)
  - **Preemptive** → Round Robin



Criteri di Scheduling

- **User-oriented →** da parte dell'utente conta il tempo di risposta
- **System-oriented →** dal punto di vista sistemistico conta il throughput(quanto lavoro riesce ad eseguire il sistema in un intervallo di tempo)

CPU-Bound → task che ha molto poco da comunicare con l'esterno

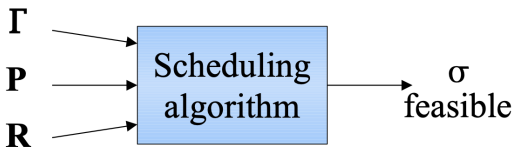
I/O bound → la maggior parte del tempo aspettano un input esterno (operazionin su disco)

CPU-Bound and I/O bound

Uno schedule σ è definito ***fattibile*** se tutti i task sono in grado di finire il loro lavoro rispettando tutti i vincoli che gli impongo

Un set di task è ***schedulabile*** se esiste un particolare algoritmo di scheduling che mi permette di meterlo in esecuzione in modo tale che i task rispettino le deadline.

- Given a set Γ of n tasks, a set P of m processors, and a set R of r resources, find an assignment of P and R to Γ which produces a feasible schedule.



Ci focalizziamo su sistemi a singolo processore, preemptive, ad attivazioni simultanee, nessun vincolo di risorse o di precedenza.

Off-line → se tutte le decisioni di scheduling sono state decise prima che i task partano (schedule table driven) ed è molto predicibile.

On-line → se tutte le decisioni di scheduling sono state prese quando i task sono già avviati

---

Statico → gli algoritmi di scheduling vengono scelti in base a parametri fissi (le priorità, i tempi non cambiano)

Dinamico → gli algoritmi di scheduling vengono scelti in base a parametri che possono cambiare nel tempo

---

Ottimali → un algoritmo è ottimale se quando il task-set è fattibile allora lo trova (complessità esponenziale)

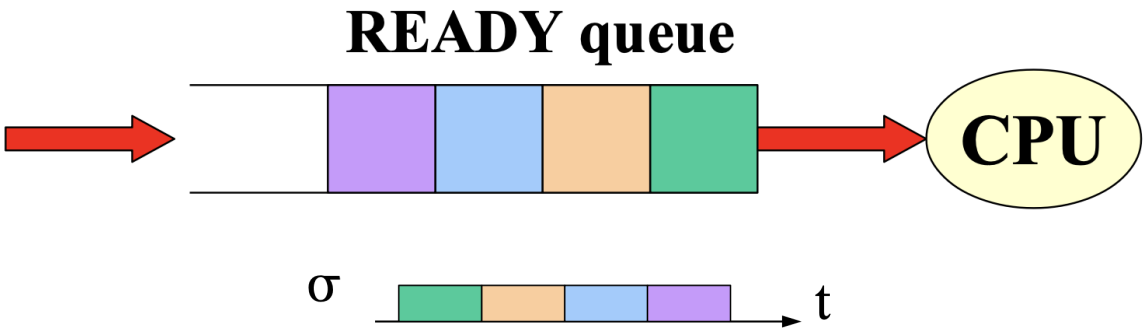
Best-effort → cerco più o meno un algoritmop ottimale, faccio il meglio che posso (complessità polinomiale)

Scheduling → quale algoritmo scegliere

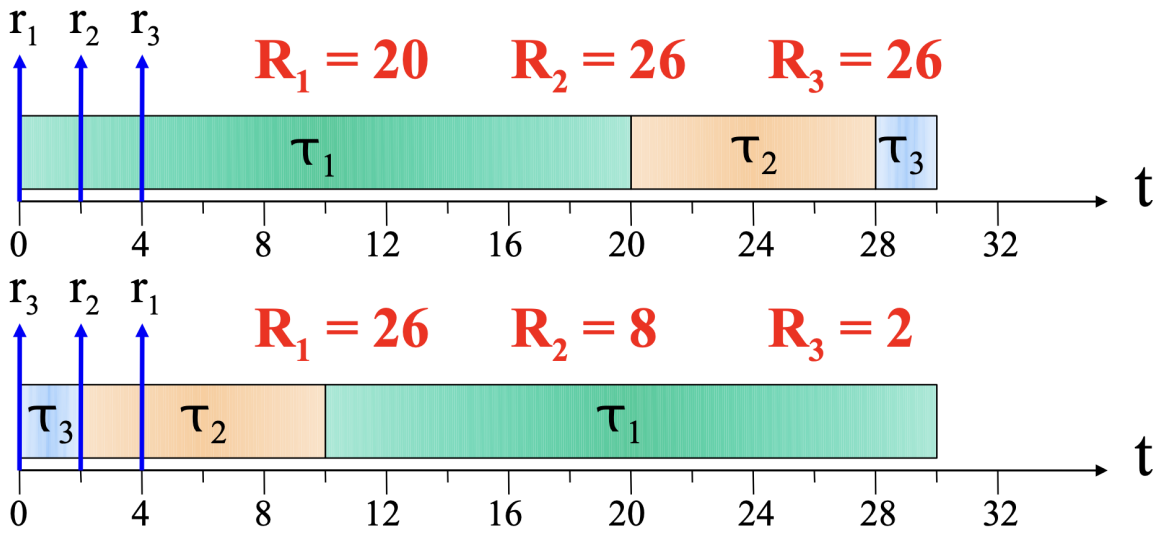
Schedulability → se le deadline vengono rispettate

Non real-time scheduling algorithms

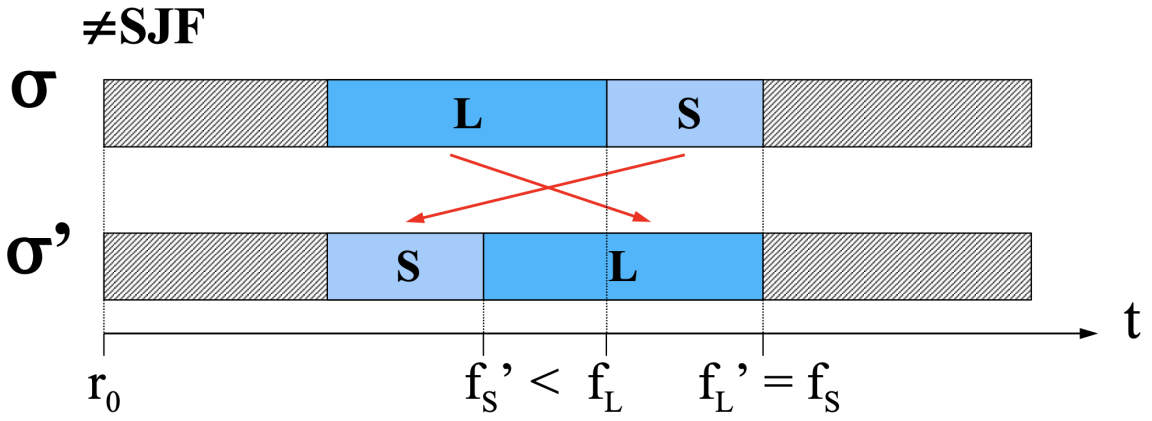
- **First Come First Served (FCFS)** → viene eseguito chi arriva prima, è non preemptive, dinamico, best-effort e on-line.



Non va bene perchè i tempi di risposta variano notevolmente a seconda dell'ordine di esecuzione.



- **Shortest Job First (SJF)** → dà priorita al task che ha il WCET più piccolo (C_i = **Worst Case Execution Time - WCET** → tempo di esecuzione di caso peggiore, se qualcosa andrà male dobbiamo considerare la configurazione che causa il ritardo maggiore) (magari l'ultima mail è quella del rettore che dice che mi ha licenziato)



$$f_S' + f_L' \leq f_S + f_L$$

$$\overline{R}(\sigma') = \frac{1}{n}\sum_{i=1}^{n}(f'_i - r_i) \leqslant \frac{1}{n}\sum_{i=1}^{n}(f_i - r_i) = \overline{R}(\sigma)$$

Tesi: SJF minimizza i tempi di risposta medi.

Per contraddizione: esiste un algoritmo σ diverso da SJF e che ha un tempo di risposta più piccolo.

E' diverso perchè prima o poi la decisione di come mettere in esecuzione i task è diversa da SJF, ovvero schedula prima un job lungo da uno corto.

Se li inverto? Così σ' è più simile a SJF di quanto lo sia σ, ma non ancora, dato che nello spazio grigio finale non sappiamo cosa accade.

$$f_s' \leq f_L \quad \& \quad f_L' = f_s$$

Aggiungo ad ambi membri f_L' e f_s.

$$f'_s + f'_L \leq f_s + f_L$$

Così dimostro che il tempo di risposta minimo di σ' è migliore di quello di σ.

Vado avanti all'infinito con questo ragionamento di scambio fino a che non arrivo a σ^*, ovvero l'algoritmo di scheduling che prende tutte le decisioni uguali ad SJF, cioè lo stesso SJF.

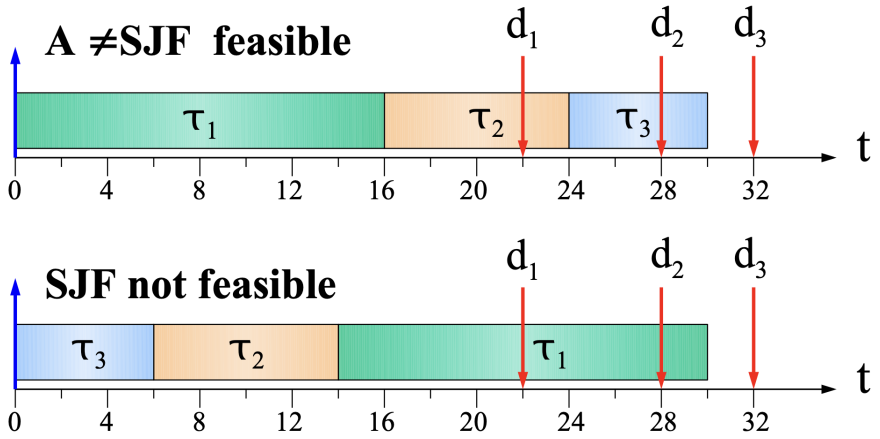
$$\sigma \longrightarrow \sigma' \longrightarrow \sigma'' \longrightarrow \ldots \longrightarrow \sigma^*$$

$$\overline{R}(\sigma) \geqslant \overline{R}(\sigma') \geqslant \overline{R}(\sigma'') \ldots \geqslant \overline{R}(\sigma^*)$$

$$\sigma^* = \sigma_{SJF}$$

$\overline{R}(\sigma_{SJF})$ is the minimum average response time achievable by any algorithm

Così dimostro che SJF è l'algoritmo ottimale in termini di risposta minima media! Ma non è comunque adatto ai sistemi real-time per lo stesso motivo del FIFO.



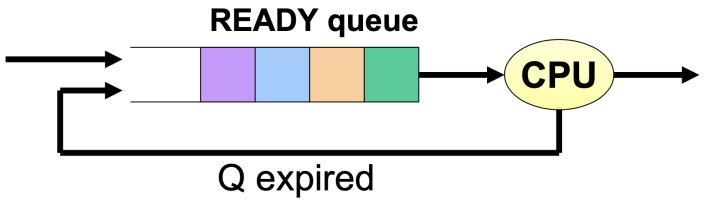
- **Priority Scheduling →** Assegnamo ad ogni task una priorità da 0 a 255 e tra quelli in coda eseguo sempre quelli con priorità di alta, mettendo in attesa gli altri bassi.

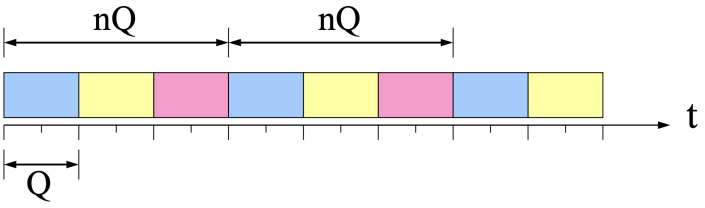
  Se più processi hanno la stessa priorità posso utilizzare il **FIFO - SCHED_FIFO** o **RR - SCHED_RR**.

  E' adatto a Real-Time? DIPENDE → va bene se assegno la priorità inversamente proporzionale al periodo del task $\frac{1}{T_i}$ o alla deadline relativa $\frac{1}{D_i}$.

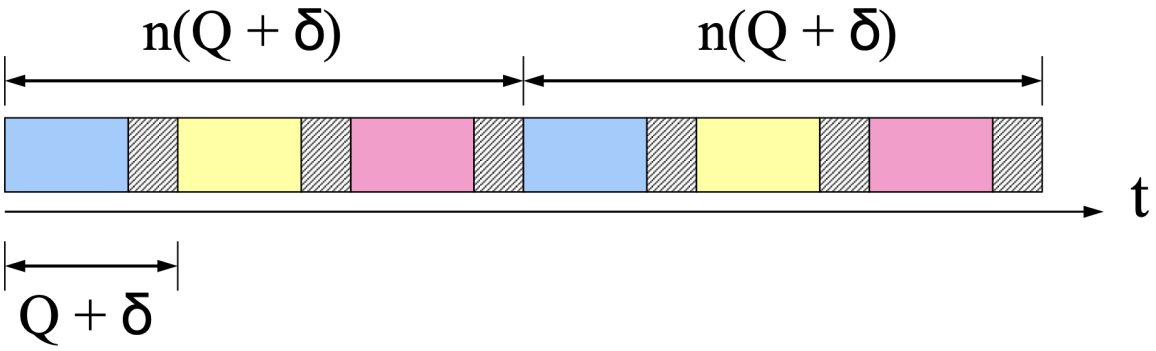
  Potrebbe incorrere in starvation → i task a bassa priorità non eseguono mai perchè ce ne sono troppi ad alta priorità. **Per risolvere** posso considerare l'**AGING**, ovvero aumentare la priorità di un task all'aumentare della sua età senza essere eseguito.

- **Round Robin (RR) →** accodo i task con politica FIFO, ma per eseguirli mi affido ad un TIME QUANTUM Q, tempo in base al quale metto in esecuzione un task, esegue per il tempo Q, se non ha finito lo rimetto in coda ed eseguo il prossimo task e così via

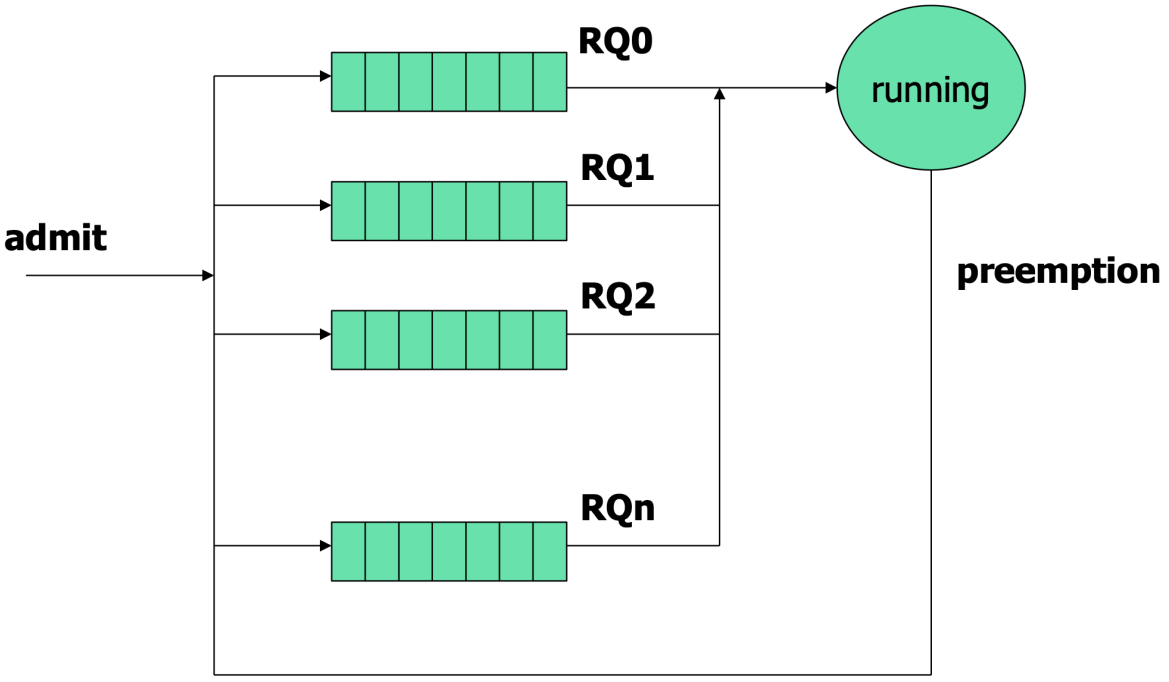




- immagina di avere n task nel sistema
- una volta finito il task azzurro rieseguo dopo $(n-1)Q$, quindi posso dire che ogni macrociclo è grande nQ
- il tempo di risposta è $R_i \simeq (nQ)\frac{C_i}{Q} = nC_i$



- Se tutti i miei task sono più piccoli $Q > max(C_i)$ → RR=FCFS
- Se $Q \simeq$ contect switch time δ (decisione di chi andrà in esecuzione dopo) allora:
  - Qui il macrociclo dura $n(Q+\delta)$ perchè δ non è più trascurabile
  - Il tempo di risposta è $R_i \simeq n(Q+\delta)\frac{C_i}{Q} = nC_i\frac{Q+\delta}{Q}$, il ritardo aumenta all'aumentare di Q

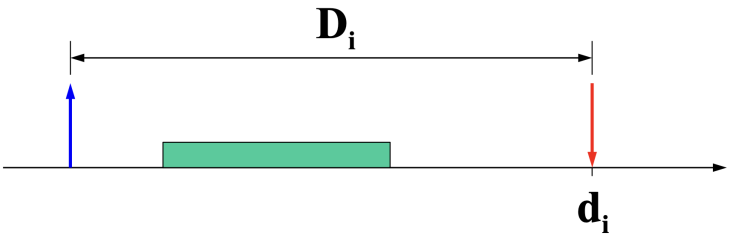


Scheduler a priorità

- Una coda per ogni priorità del sistema, gestita in FIFO o RR
- Con il meccanismo del **multiple-feedback queues**, via via che un task a bassa priorità veniva eseguito ma non riusciva a terminare data la presenza di molti task ad alta priorità e la sua conseguente preemtion frequente, lo preempto ma non nella stessa coda, ma in una a priorità più alta e così via fino al suo termine. (magari può anche arrivare ad avere priorità massima 0)

I task possono essere schedulati a seconda di

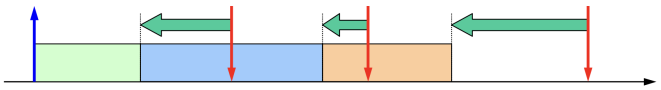
- Deadline relativa D_i → statica
- Deadline assoluta d_i → dinamica



Real-time algorithms

- **Earliest Due Date** → quando i task arrivano tutti insieme allo stesso tempo all'instante t e li schedula in base alla loro deadline ($d_i = D_i$, perchè l'istante di partenza è uguale per tutti), minimizza la lateness massima L_{max} (finishing time - deadline assoluta) (quanto è in ritardo il mio task rispetto alla deadline)

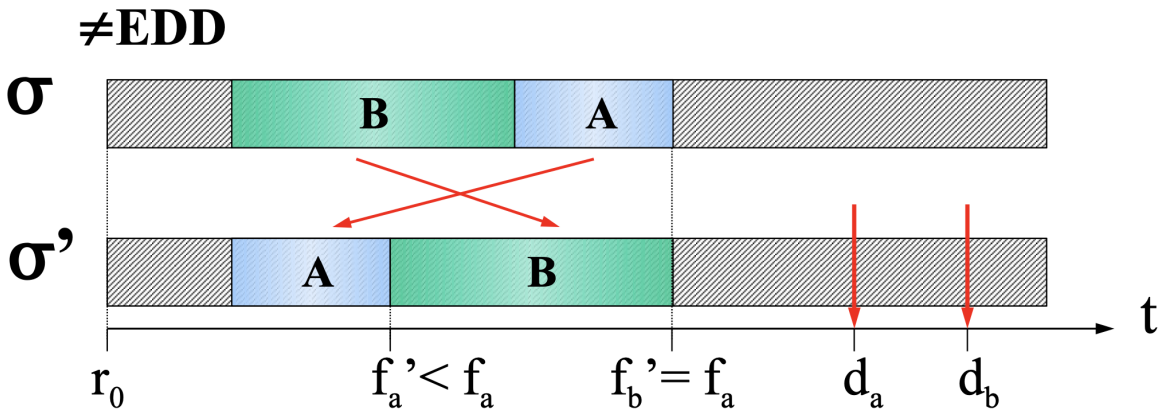
$$L_{max} = \max_i (L_i)$$



if $(L_{max} < 0)$ **then**

no task misses its deadline

Dimostrazione: vogliamo domostrare che EDD minimizza la lateness massima.



$$L_{max} = L_a = f_a - d_a$$

$$\left.\begin{aligned} L_a' &= f_a' - d_a < f_a - d_a \\ L_b' &= f_b' - d_b = f_a - d_b < f_a - d_a \end{aligned}\right\} \quad L'_{max} < L_{max}$$

Contraddizione: c'è un altro algoritmo, diverso da EDD, che mi ritorna una lateness ancora più piccola.

- In σ metto prima B rispetto ad A, variando rispetto a EDD dato che la deadline di A viene prima di quella di B, quindi avrei dovuto eseguire prima A.
- In σ' diventa più simile a EDD ed eseguo prima A poi B.

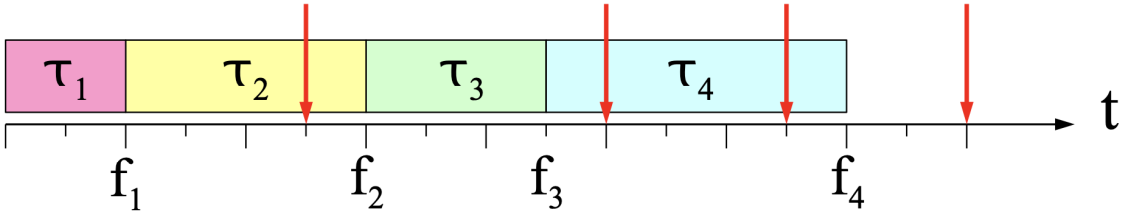
$$\sigma \longrightarrow \sigma' \longrightarrow \sigma'' \longrightarrow \ldots \longrightarrow \sigma^*$$

$$L_{max}(\sigma) \geqslant L_{max}(\sigma') \geqslant L_{max}(\sigma'') \ldots \geqslant L_{max}(\sigma^*)$$

$$\sigma^* = \sigma_{EDD}$$

$L_{max}(\sigma_{EDD})$ is the minimum value achievable by any algorithm

Test di schedulabilità → se dato un task γ esso è schedulabile



a task set Γ is feasible if $\forall\ i \quad f_i \leqslant d_i$

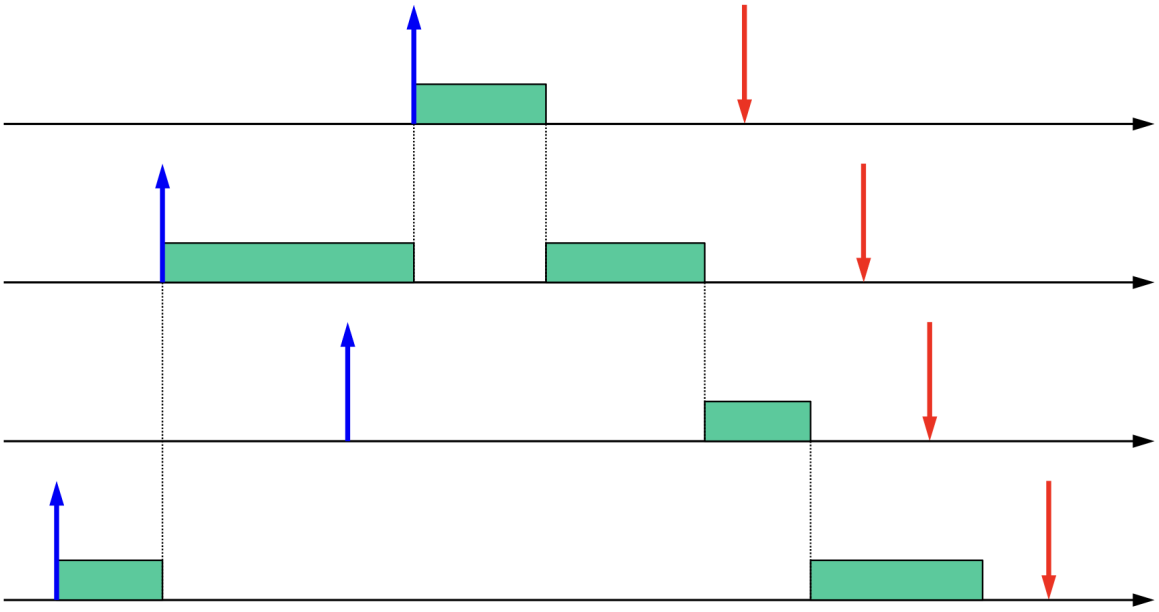
$$f_i = \sum_{k=1}^{i} C_k$$

$$\forall\ i \quad \sum_{k=1}^{i} C_k \leqslant D_i$$

Devp verificare il tempo di risposta di ogni task sia inferiore alla loro deadline, dove il tempo di risposta di ogni task corrisponde alla sommatoria qui sopra.

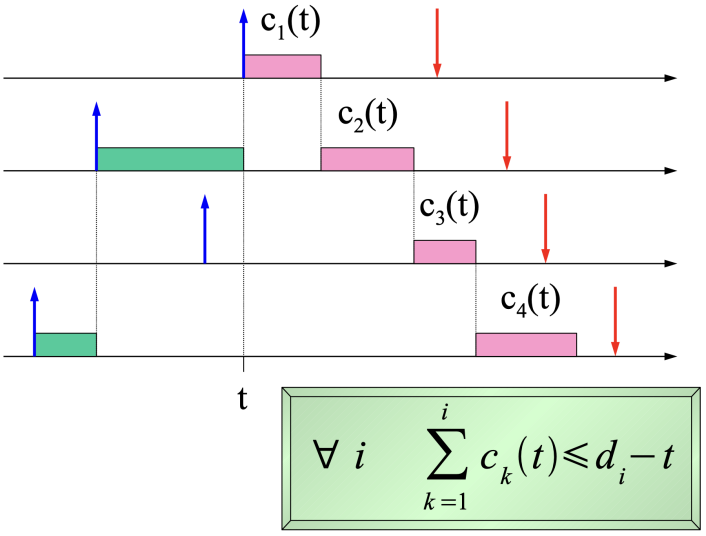
- **Earliest Deadline First** → stessa cosa di EDD, ma lo schedule dipende dalla schedule assoluta ($d_i = r_i + D_i$). Ci interessa la sua versione **preemptive** → se mi arriva un task che ha una deadline più emminente rispetto a quello che ho ora in esecuzione, lo eseguo preemptando quello in esecuzione.

  Anche lui minimizza la leteness massima. (stessa domostrazione di EDD).



- Il task che arriva prima sta nell'ultima fila e così a salire.
- Avvengono 5 cambi di contesto, quindi 5 possibili overhead.

Ma voglio trovare un modo per dire a priori se le deadline verranno rispettate.



- Per ogni task devo effettuare questo test:
  - Per J1 guardo il tempo che gli rimane da eseguire $c_1(t)$ sommato a tutti quelli che hanno priorità più alta di lui (in questi caso nessuno) sia più piccolo del **TTD - Time To Deadline** ($d_i - t$).

    Ho abbastanza spazio di esecuzione per eseguire quello che devo fare io più tutti quelli che hanno più alta priorità di me? In questo caso si
  - Per J2 sommo $c_1(t)$ e $c_2(t)$, questo deve essere minore di $d_2 - t$. In questo caso si.
  - Per J3 sommo $c_1(t), c_2(t), c_3(t)$, questo deve essere minore di $d_3 - t$. In questo caso si.
  - Per J4 sommo $c_1(t), c_2(t), c_3(t), c_4(t)$, questo deve essere minore di $d_4 - t$. In questo caso si.

Problemi di complessità

- EDD → $O(n\ logn)$ per ordinare i task e $O(n)$ per la garanzia.
- EDF → $O(n)$ per inserire un task nella coda e $O(n)$ per la garanzia di un nuovo task.