Definitions

A schedule $\sigma$ is said to be feasible if it satisfies a set of constraints.

A task set $\Gamma$ is said to be feasible, if there exists an algorithm that generates a feasible schedule for $\Gamma$.

A task set $\Gamma$ is said to be schedulable with an algorithm $A$, if $A$ generates a feasible schedule.

Examples of constraints

- Timing constraints: activation, period, deadline, jitter.
- Precedence: order of execution between tasks.
- Resources: synchronization for mutual exclusion.

Feasibility vs. schedulability

The scheduling problem

Given a set $\Gamma$ of $n$ tasks, a set $P$ of $p$ processors, and a set $R$ of $r$ resources, find an assignment of $P$ and $R$ to $\Gamma$ that produces a feasible schedule under a set of constraints.

Complexity

- In 1975, Garey and Johnson showed that the general scheduling problem is NP hard.

In practice, it means that the time for finding a feasible schedule grows exponentially with the number of tasks.

Fortunately, polynomial time algorithms can be found under particular conditions.

Why do we care about complexity?

- Let's consider an application with $n = 30$ tasks on a processor in which the elementary step takes 1 $\mu$s.

- Consider 3 algorithms with the following complexity:
  
  - $A_1$: $O(n)$
  - $A_2$: $O(n^8)$
  - $A_3$: $O(8^n)$

  $30 \mu$s  
  182 hours  
  40,000 billion years
### Simplifying assumptions
- Single processor
- Homogeneous task sets
- Fully preemptive tasks
- Simultaneous activations
- No precedence constraints
- No resource constraints

### Algorithm taxonomy
- Preemptive vs. Non Preemptive
- Static vs. dynamic
- On line vs. Off line
- Optimal vs. Heuristic

### Static vs. Dynamic
**Static**

scheduling decisions are taken based on fixed parameters, statically assigned to tasks before activation.

**Dynamic**

scheduling decisions are taken based on parameters that can change with time.

### Off-line vs. On-line
**Off-line**

all scheduling decisions are taken before task activation; the schedule is stored in a table (table-driven scheduling).

**On-line**

scheduling decisions are taken at run time on the set of active tasks.

### Optimal vs. Heuristic
**Optimal**

They generate a schedule that minimizes a cost function, defined based on an optimality criterion.

**Heuristic**

They generate a schedule according to a heuristic function that tries to satisfy an optimality criterion, but there is no guarantee of success.

### Optimality criteria
- **Feasibility:** Find a feasible schedule if there exists one.
- Minimize the maximum lateness
- Minimize the number of deadline miss
- Assign a value to each task, then maximize the cumulative value of the feasible tasks
Task set assumptions

We consider algorithms for different types of tasks:

- Single-job tasks (one shot)
  tasks with a single activation (not recurrent)
- Periodic tasks
  recurrent tasks regularly activated by a timer (each task potentially generates infinite jobs)
- Aperiodic/Sporadic tasks
  recurrent tasks irregularly activated by events (each task potentially generates infinite jobs)
- Mixed task sets

Graham’s Notation

\[ \alpha | \beta | \gamma \]

- \( \alpha \) denotes the number of processors
- \( \beta \) denotes the constraints on tasks
- \( \gamma \) denotes the optimality criterion

Examples:

1. \( \text{preem.} \ R_{\text{avg}} \) Uniprocessor algorithm for preemptive tasks that minimizes the average response time.
2. \( \text{sync.} \ L_{\text{max}} \) Dual-core algorithm for synchronous tasks that minimizes the maximum lateness.
3. \( \text{preem.} \ L_{\text{max}} \) Quad-core algorithm for preemptive tasks that minimizes the maximum lateness.

Classical scheduling policies

- First Come First Served
- Shortest Job First
- Priority Scheduling
- Round Robin

Not suited for real-time systems

First Come First Served

It assigns the CPU to tasks based on their arrival times (intrinsically non-preemptive):

First Come First Served (SJF)

- Very unpredictable
  response times strongly depend on task arrivals:

Shortest Job First (SJF)

- Static \( (C_i \text{ is a constant parameter}) \)
- It can be used on line or off-line
- Can be preemptive or non-preemptive
- It minimizes the average response time
### SJF - Optimality

\[ \sigma \rightarrow \sigma' \rightarrow \sigma'' \rightarrow \ldots \rightarrow \sigma^* \]

\[ R(\sigma) \geq R(\sigma') \geq R(\sigma'') \ldots \geq R(\sigma^*) \]

\[ \sigma^* = \sigma_{SJF} \]

The minimum response time achievable by any algorithm

---

### SJF - Optimality

\[ f_s' + f_l' \leq f_s + f_l \]

\[ R(\sigma') = \frac{1}{n} \sum_{i=1}^{n} (f_i' - r_i) \leq \frac{1}{n} \sum_{i=1}^{n} (f_i - r_i) = R(\sigma) \]

---

### Priority Scheduling

- Each task has a priority \( P_i \), typically \( P_i \in [0, 255] \)
- The task with the highest priority is selected for execution.
- Tasks with the same priority are served FCFS

**NOTE:**

\[ P_i \propto \frac{1}{C_i} \Rightarrow SJF \]

\[ P_i \propto \frac{1}{a_i} \Rightarrow FCFS \]

---

### Is SJF suited for Real-Time?

- It is not optimal in the sense of feasibility

- **A \( \neq \) SJF feasible**
  - \( d_1 \)
  - \( d_2 \)
  - \( d_3 \)

- **SJF not feasible**
  - \( \tau_1 \)
  - \( \tau_2 \)
  - \( \tau_3 \)

---

### Round Robin

The ready queue is served with FCFS, but...

- Each task \( \tau_i \) cannot execute for more than \( Q \) time units (\( Q = \) time quantum).
- When \( Q \) expires, \( \tau_i \) is put back in the queue.

---

### Problem: starvation

Low priority tasks may experience long delays due to the preemption of high priority tasks.

### A possible solution: aging

Priority increases with waiting time.
Round Robin

\[ T = \frac{nQ}{Q} \]

**Time sharing**

Each task runs as if it was executing alone on a virtual processor \( n \) times slower than the real one.

\[ R_i \approx \frac{(nQ)C}{Q} = nC_i \]

Round Robin

- if \( Q > \text{max}(C_i) \) then \( RR = \text{FCFS} \)
- if \( Q \approx \text{context switch time (\( \delta \))} \) then

\[ R_i \approx \frac{n(Q + \delta)C}{Q} = nC_i \left( \frac{Q + \delta}{Q} \right) \]

Multi-Level Scheduling

**CPU**

- High priority
- Medium priority
- Low priority

**PRIORITY**

**RR**

**FCFS**

Real-Time Scheduling Algorithms

They can either schedule tasks by

- relative deadlines \( D_i \) (static)
- absolute deadlines \( d_i \) (dynamic)
**Earliest Due Date**

**Problem**
- 1 sync \( L_{\text{max}} \)

**Algorithm** [Jackson 55]
- Order the ready queue by increasing deadline.

**Assumptions**
- All the tasks are simultaneously activated
- Preemption is not needed.
- Static (\( D_i \) is fixed)

**Property**
- It minimizes the maximum lateness (\( L_{\text{max}} \))

**Lateness**

\[
L_i = f_i - d_i
\]

\[
L_i > 0
\]

- \( L_i < 0 \)

**Maximum Lateness**

\[
L_{\text{max}} = \max_i (L_i)
\]

- if \( L_{\text{max}} < 0 \) then
- no task exceeds its deadline

**EDD - Optimality**

\[
\sigma \rightarrow \sigma' \rightarrow \sigma'' \rightarrow \ldots \rightarrow \sigma^*
\]

\[
L_{\text{max}} (\sigma) \geq L_{\text{max}} (\sigma') \geq L_{\text{max}} (\sigma'') \geq \ldots \geq L_{\text{max}} (\sigma^*)
\]

\[
\sigma^* = \sigma_{\text{EDD}}
\]

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

**EDD guarantee test** (off line)

A task set \( \Gamma \) is feasible iff

\[
\forall i \quad f_i \leq d_i
\]

\[
f_i = \sum_{k=1}^{i} C_k \quad \forall i \quad \sum_{k=1}^{i} C_k \leq D_i
\]
**Earliest Deadline First**

**Problem**

1  \( |\text{preem.}| \mid L_{\text{max}} \)

**Algorithm** [Horn 74]
- Order the ready queue by increasing absolute deadline

**Assumptions**
- Tasks can be activated dynamically
- Dynamic algorithm (\( d_i \) depends on \( a_i \))
- Tasks can be preempted at any time

**Property**
- It minimizes the maximum lateness (\( L_{\text{max}} \))

---

**EDF Guarantee test (on line)**

\[
\forall i, \sum_{k=1}^{i} c_k(t) \leq d_i - t
\]

---

**Complexity**

**EDD**
- Scheduler (queue ordering): \( O(n \log n) \)
- Feasibility Test (guarantee test): \( O(n) \)

**EDF**
- Scheduler (insertion in the queue): \( O(n) \)
- Feasibility Test (guarantee single task): \( O(n) \)

---

**EDF optimality**

⇒ In the sense of feasibility [Dertouzos 1974]

An algorithm A is **optimal** in the sense of feasibility if it generates a feasible schedule, if there exists one.

**Demonstration method**

It is sufficient to prove that, given an arbitrary feasible schedule, the schedule generated by EDF is also feasible.

---

**Dertouzos Transformation**

\[
\begin{align*}
\sigma(t) &= \text{task executing at time } t \\
E(t) &= \text{task with min } d \text{ at time } t \\
t_E &= \text{time at which } E \text{ is executed}
\end{align*}
\]

for \( t = 0 \) to \( D_{\text{max}} - 1 \)

\[
\text{if } (\sigma(t) = E(t)) \{
\begin{align*}
\sigma_{\text{DF}}(t) &= \sigma(t) \\
\sigma(0) &= E(t)
\end{align*}
\}
\]
Dertouzos Transformation

\[ \sigma(t) = \text{task executing at time } t \]
\[ E(t) = \text{task with min } d \text{ at time } t \]
\[ t_ε = \text{time at which } E \text{ is executed} \]

For \( t = 0 \) to \( D_{max} - 1 \)
\[
\begin{align*}
 &\text{if } (\sigma(t) \neq E(t)) \\
 &\quad \sigma(t_ε) = \sigma(t) \\
 &\quad \sigma(t) = E(t)
\end{align*}
\]

Dertouzos Transformation

\[ \sigma(t) = \text{task executing at time } t \]
\[ E(t) = \text{task with min } d \text{ at time } t \]
\[ t_ε = \text{time at which } E \text{ is executed} \]

For \( t = 0 \) to \( D_{max} - 1 \)
\[
\begin{align*}
 &\text{if } (\sigma(t) \neq E(t)) \\
 &\quad \sigma(t_ε) = \sigma(t) \\
 &\quad \sigma(t) = E(t)
\end{align*}
\]

A property of optimal algorithms

If a task set \( \Gamma \) is not schedulable by an optimal algorithm, then \( \Gamma \) cannot be scheduled by any other algorithm.

If an algorithm A minimizes \( L_{max} \) then A is also optimal in the sense of feasibility. The opposite is not true.

Non Preemptive Scheduling

Under non preemptive execution, EDF is not optimal:

Feasible schedule

EDF
Non Preemptive Scheduling

To achieve optimality, an algorithm should be clairvoyant, and decide to leave the CPU idle in the presence of ready tasks:

If we forbid leaving the CPU idle in the presence of ready tasks, then EDF is optimal. We say that:

NP-EDF is optimal among non-idle scheduling algorithms

Bratley's Algorithm [Bratley 71]

\[ ( 1 \mid \text{no-preem} \mid \text{Lmax} ) \]

Reduces the average complexity by a pruning rule:

Do not expand unless the partial schedule is found to be strongly feasible.

A partial schedule is said to be strongly feasible if adding any of the remaining nodes it remains feasible.

Heuristic search

Spring algorithm [Stankovic & Ramamritham 87]

1. The schedule for a set of N tasks is constructed in N steps
2. The search is driven by a heuristic function H
3. At each step the algorithm selects the task that minimizes the heuristic function

Backtracking is possible

Heuristic functions

Spring algorithm [Stankovic & Ramamritham 87]

Example of heuristic functions:

1. \( H = t_i \) \( \Rightarrow \) FCFS
2. \( H = C_i \) \( \Rightarrow \) SJF
3. \( H = D_i \) \( \Rightarrow \) DM
4. \( H = d_i \) \( \Rightarrow \) EDF

Composite heuristic functions:

\[ H = w_1 t_i + w_2 D_i \]
\[ H = w_1 C_i + w_2 d_i \]
\[ H = w_1 V_i + w_2 d_i \]
Heuristic functions

Spring algorithm [Stankovic & Ramamritham 87]

Possibility to handle precedence constraints:

Eligibility

\[ E_i = \infty \quad \text{or} \quad E_i = 1 \]

Heuristic functions:

\[ H = E_i (w_1 r_i + w_2 D_i) \]
\[ H = E_i (w_1 C_i + w_2 d_i) \]

Heuristic algorithm

Spring algorithm [Stankovic & Ramamritham 87]

Complexity:

- Exhaustive search: \( O(N!N!) \)
- Heuristic search: \( O(N^2) \)
- Heuristic w. k btracks: \( O(kN^2) \)

Handling precedence constraints

1 | prec, sync | \( L_{\max} \)

Latest Deadline First (LDF) [Lawler 73]

Given a precedence graph, it constructs the schedule from the tail: among the nodes with no successors, LDF selects the task with the latest deadline:

![LDF Diagram]

Handling precedence constraints

EDF* [Chetto & Chetto 89]

- Assumes that arrival times are known a priori;
- Transforms precedence constraints into timing constraints by modifying arrival times and deadlines based on the precedence graph;
- Applies EDF to the modified task set.

Handling precedence constraints

EDF* [Chetto & Chetto 89] 1 | prec, preem | \( L_{\max} \)

The idea is to:
- postpone the arrival time of a successor
- advance the deadline of a predecessor
Handling precedence constraints

**EDF* [Chetto & Chetto 89]** 1 | prec, preem | $L_{\text{max}}$

The idea is to:

- Postpone the arrival time of a successor: $r^*_B = r_A + C_A$
- Advance the deadline of a predecessor: $d^*_A = d_B - C_B$

Arrival time modification

1. For all root nodes, set $r^*_i = r_i$.  
2. Select a task $\tau_i$ such that all its immediate predecessors have been modified, else exit.
3. Set $r^*_i = \max \{ r_i, \max (r^*_k + C_k) \}$.
4. Repeat from line 2.

Deadline modification

1. For all leaves, set $d^*_i = d_i$.  
2. Select a task $\tau_i$ such that all its immediate successors have been modified, else exit.
3. Set $d^*_i = \min \{ d_i, \min (d^*_k - C_k) \}$.
4. Repeat from line 2.

Summary

<table>
<thead>
<tr>
<th>activ</th>
<th>prec</th>
<th>preem</th>
<th>algorithm</th>
<th>authors</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync</td>
<td>N</td>
<td>*</td>
<td>EDD</td>
<td>Jackson '55</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Asyn</td>
<td>N</td>
<td>Y</td>
<td>EDF</td>
<td>Horn '74</td>
<td>$O(n) \land \tau$</td>
</tr>
<tr>
<td>Asyn</td>
<td>N</td>
<td>N</td>
<td>NP-EDF</td>
<td>Jeffay '91</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Asyn</td>
<td>Y</td>
<td>N</td>
<td>Tree search</td>
<td>Bratley '71</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Asyn</td>
<td>Y</td>
<td>N</td>
<td>Spring</td>
<td>Stankovic '87</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sync</td>
<td>Y</td>
<td>*</td>
<td>LDF</td>
<td>Lawler '73</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Asyn</td>
<td>Y</td>
<td>Y</td>
<td>EDF*</td>
<td>Chetto '89</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>