Real-Time Scheduling and Threads: Basics

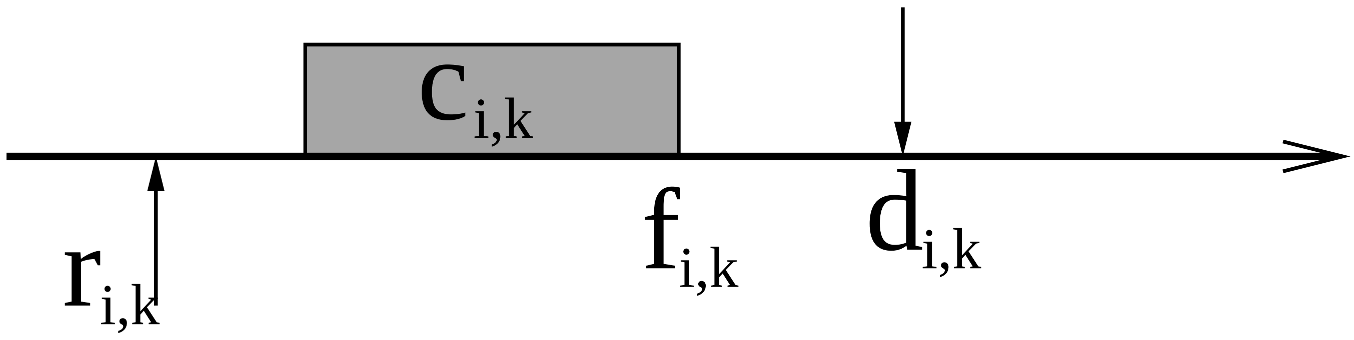
Luca Abeni
luca.abeni@unitn.it

- The time when a result is produced matters
  - A correct result produced *too late* is equivalent to a wrong result (or to no result)
- What does "*too late*" mean, here?
  - Applications characterised by temporal constraints that have to be respected
- Temporal constraints are modelled using the concept of *deadline*
  - Some activity has to finish before a specified time (deadline)
  - Some data has to be generated before a deadline
  - Some process/thread must terminate before a deadline
  - ...

Processes, Threads, and Tasks

- Algorithm $\rightarrow$ logical procedure used to solve a problem
- Program $\rightarrow$ formal description of an algorithm, using a *programming language*
- Process $\rightarrow$ instance of a program (program in execution)
  - Thread $\rightarrow$ flow of execution
  - Task $\rightarrow$ process or thread
- A task can be seen as a sequence of actions . . .
- . . . and a deadline must be associated to each one of them!
  - Some kind of formal model is needed to identify these “actions” and associate deadlines to them

- Real-Time task τ_i: stream of jobs (or instances) $J_{i,k}$
- Each job $J_{i,k} = (r_{i,k}, c_{i,k}, d_{i,k})$:
  - Arrives at time $r_{i,k}$ (activation time)
  - Executes for a time $c_{i,k}$
  - Finishes at time $f_{i,k}$
  - Should finish within an absolute deadline $d_{i,k}$



Mathematical Model of a Task - 2

- Summing up: a job is an abstraction used to associate deadlines (temporal constraints) to activities
  - $r_{i,k}$ is the time when job $J_{i,k}$ is *activated* (by an external event, a timer, an explicit activation, etc...)
  - $c_{i,k}$ is the computation time needed by job $J_{i,k}$ to complete
  - $d_{i,k}$ is the absolute time instant by which job $J_{i,k}$ must complete
    - job $J_{i,k}$ respects its deadline if $f_{i,k} \leq d_{i,k}$
- Response time of job $J_{i,k}$: $\rho_{i,k} = f_{i,k} - r_{i,k}$

Periodic and Sporadic Tasks

Periodic / Sporadic task $\tau_i = (C_i, D_i, T_i)$: stream of jobs $J_{i,k}$, with

$$
\begin{aligned}
r_{i,k+1} &= \text{(or } \geq \text{)} \quad r_{i,k} + T_i \\
d_{i,k} &= r_{i,k} + D_i \\
C_i &= \max_k \{c_{i,k}\}
\end{aligned}
$$

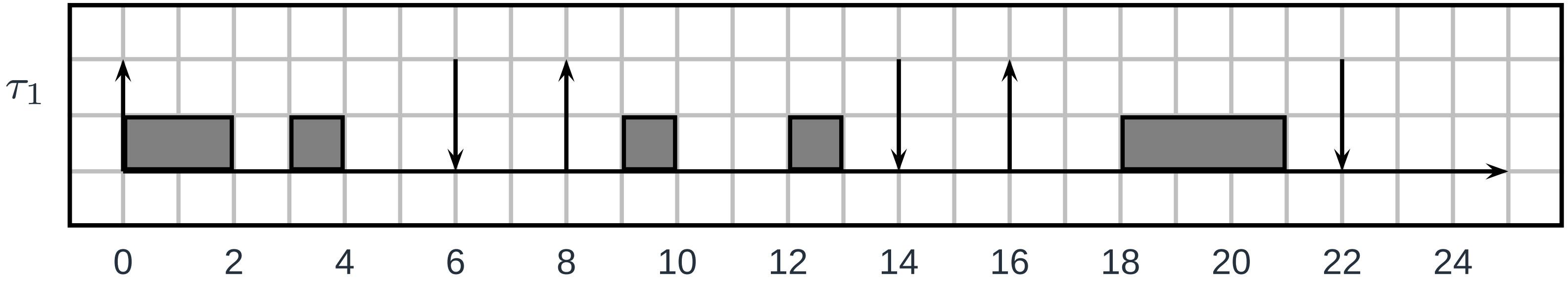
- T_i is the task *period* (or *Minimum Inter-arrival Time*)
- D_i is the task *relative deadline*
- C_i is the task worst-case execution time (WCET)
- R_i is the worst-case response time:
  $R_i = max_k\{\rho_{i,k}\} = max_k\{f_{i,k} - r_{i,k}\}$
  - for the task to be correctly scheduled, it must be $R_i \leq D_i$

Example: Periodic Task Model

- A periodic task has a regular structure (cycle):
  - activate periodically (period T_i)
  - execute a computation
  - suspend waiting for the next period

```
void *PeriodicTask(void *arg)
{
    <initialization>;
    <start periodic timer, period = T>;
    while (cond) {
        <do something...>;
        <wait next activation>;
    }
}
```

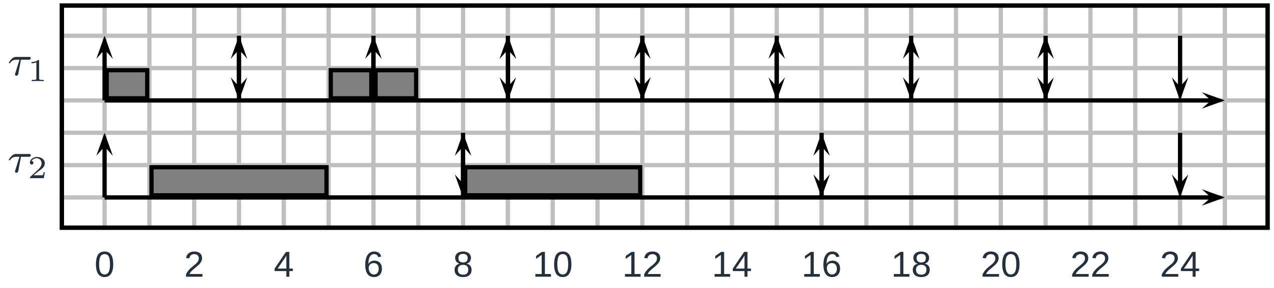
Tasks are graphically represented by using a scheduling diagram. For example, the following picture shows a schedule of a periodic task $\tau_1 = (3, 6, 8)$ (with $WCET_1 = 3$, $D_1 = 6$, $T_1 = 8$)



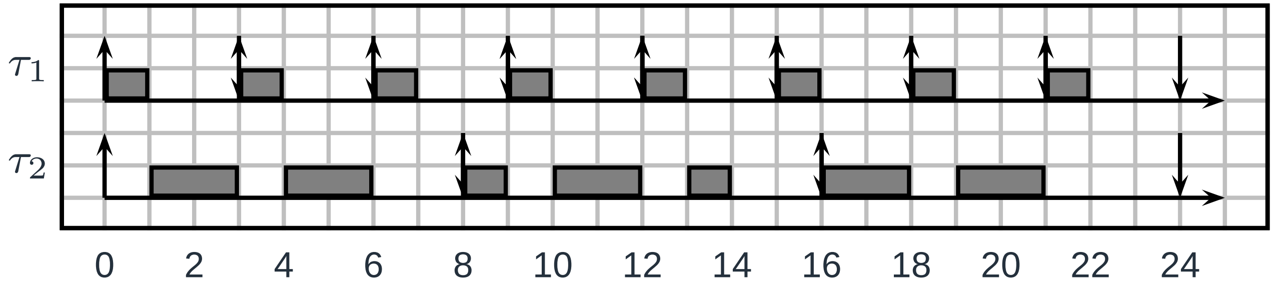
Notice that, while job $J_{1,1}$ and $J_{1,3}$ execute for 3 units of time (WCET), job $J_{1,2}$ executes for only 2 units of time.

- A real-time task $\tau_i = (C_i, D_i, T_i)$ (or $\tau_i = (C_i, T_i)$ if $D_i = T_i$) is properly served if all jobs respect their deadline...
- ...Appropriate scheduling is important!
  - The scheduler must somehow know the temporal constraints of the tasks...
  - ...In order to schedule them so that such temporal constraints are respected
- How should real-time tasks be scheduled? (scheduling algorithm?)
- Is it possible to schedule them so that all deadlines are respected?
- Does Linux provide an appropriate scheduling algorithm?

- The task set $\mathcal{T} = \{(1, 3), (4, 8)\}$ is not schedulable by FCFS



- $\mathcal{T} = \{(1, 3), (4, 8)\}$ is schedulable using fixed priorities



So... Are Fixed Priorities Enough?

- According to the previous example, a fixed-priority scheduler can be appropriate for scheduling real-time tasks...
- ...Is this true in general, or only for some (theoretical) examples?
  - Given a set of real-time tasks $\Gamma = \{\tau_i\}$, can a fixed priority scheduler allow to respect all the deadlines?
  - Is it possible to know in advance if some deadline will be missed?
  - How to assign the priorities?
- If fixed priorities are enough, the `SCHED_FIFO` and `SCHED_RR` policies can be used!

Optimal Priority Assignment

- Given a periodic task set T with all tasks having relative deadline D_i equal to the period T_i ($\forall i, \quad D_i = T_i$)
  - The best assignment is the *Rate Monotonic* (RM) assignment
  - Shorter period $\rightarrow$ higher priority
- Given a periodic task set with deadline different from periods:
  - The best assignment is the *Deadline Monotonic* assignment
  - Shorter relative deadline $\rightarrow$ higher priority
- For sporadic tasks, the same rules are valid as for periodic tasks

- Given a task set, is it possible to check if it is schedulable of not?
- In many cases it is useful to have a very simple test to see if the task set is schedulable.
- A sufficient test is based on the *Utilisation bound*:
  - The *utilisation least upper bound* for scheduling algorithm $\mathcal{A}$ is the smallest possible utilisation U_{lub} such that, for any task set $\mathcal{T}$, if the task set's utilisation U is not greater than U_{lub} ($U \leq U_{lub}$), then the task set is schedulable by algorithm $\mathcal{A}$

Utilisation

- Each task uses the processor for a fraction of time

$$U_i = \frac{C_i}{T_i}$$

- The total processor utilisation is

$$U = \sum_i \frac{C_i}{T_i}$$

- This is a measure of the processor's load

Necessary Condition

- If $U > 1$ the task set is surely not schedulable
- However, if $U < 1$ the task set may or may not be schedulable ...
- If $U < U_{lub}$, the task set **is schedulable**!!!
  - "Gray Area" between U_{lub} and 1
  - We would like to have U_{lub} near to 1
  - $U_{lub} = 1$ would be optimal!!!

Utilisation Bound for RM

- We consider n periodic (or sporadic) tasks with relative deadline equal to periods.
- Priorities are assigned with Rate Monotonic;
- $U_{lub} = n(2^{1/n} - 1)$
  - U_{lub} is a decreasing function of n;
  - For large n: $U_{lub} \approx 0.69$

n	**U**$_{lub}$	**n**	**U**$_{lub}$
2	0.828	7	0.728
3	0.779	8	0.724
4	0.756	9	0.720
5	0.743	10	0.717
6	0.734	11	...

Utilisation Bound for DM

- If relative deadlines are less than or equal to periods, instead of considering $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$, we can consider:

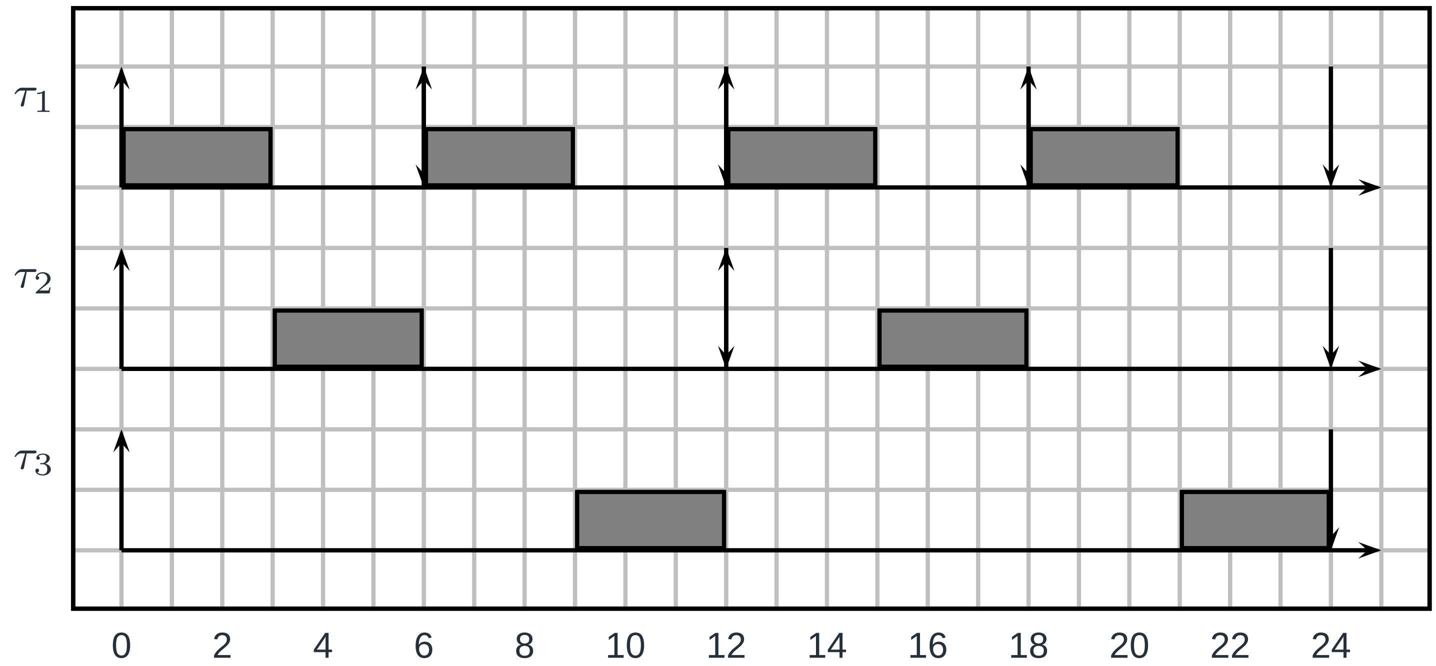
$$U' = \sum_{i=1}^{n} \frac{C_i}{D_i}$$

- Then the test is the same as the one for RM (or DM), except that we must use U' instead of U.
- Idea: $\tau = (C, D, T) \rightarrow \tau' = (C, D, D)$
  - τ' is a "worst case" for τ
  - If τ' can be guaranteed, τ can be guaranteed too

Pessimism of the Analysis: Example

$\tau_1 = (3, 6)$, $\tau_2 = (3, 12)$, $\tau_3 = (6, 24)$;

$U = 1$;



Dynamic Priorities - Earliest Deadline First

- RM and DM are optimal *fixed priority* assignments
- Maybe we can improve schedulability by using *dynamic priorities*?
  - Fixed priority scheduling: a task τ always has the same priority
  - Dynamic priority scheduling: τ's priority can change during time...
  - Let's assume that the priority changes from job to job (a job $J_{i,j}$ always has the same priority $p_{h,k}$)
- Simplest idea: give priority to tasks with the earliest absolute deadline: $d_{i,j} < d_{h,k} \Rightarrow p_{i,j} > p_{h,k}$
  - Earliest Deadline First (EDF)
  - DM $\rightarrow$ *relative* deadlines; EDF $\rightarrow$ *absolute* deadlines

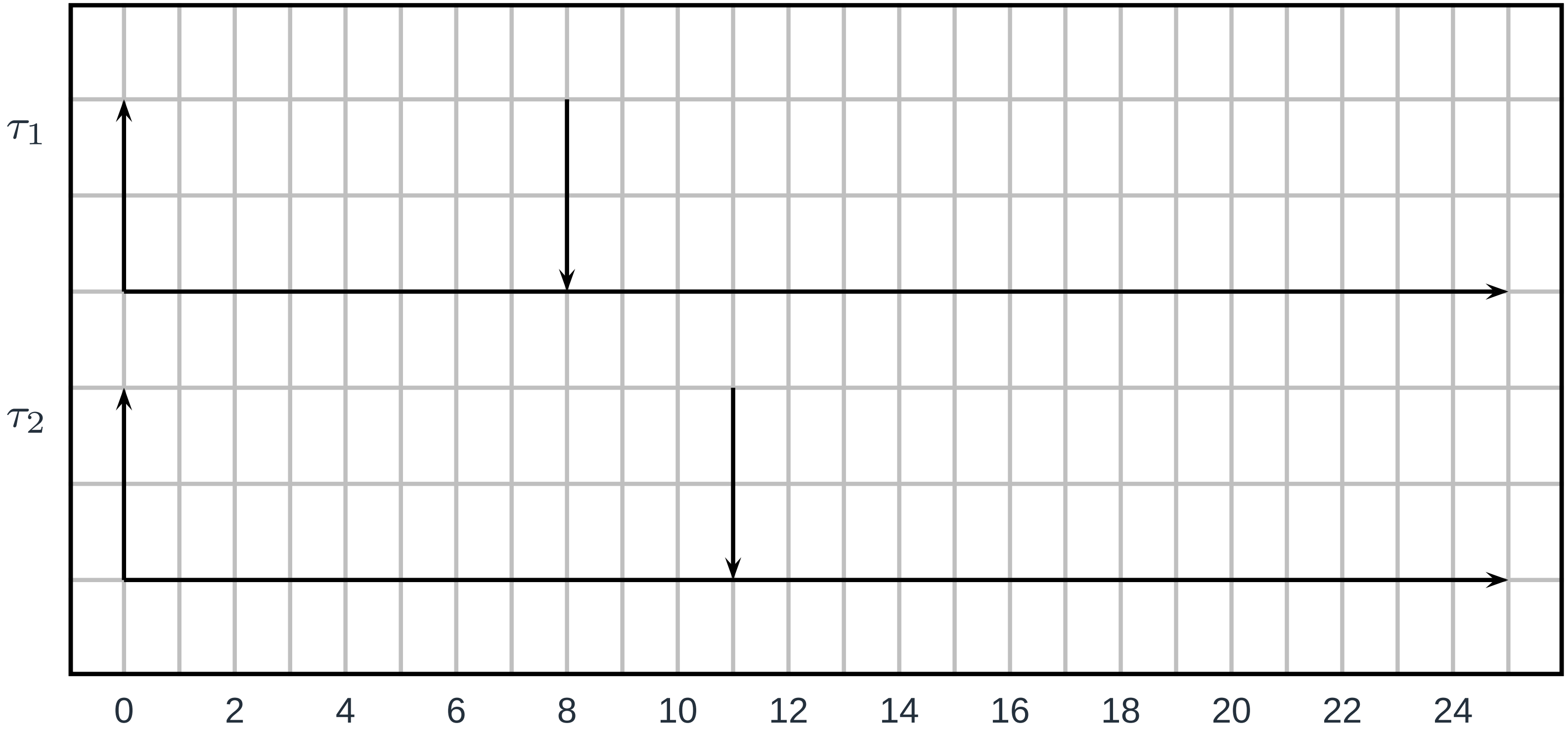
Can We Do any Better than RM?

- Yes (of course!): EDF can get full processor utilisation
- Consider a system of periodic tasks with relative deadline equal to the period.
- The system is schedulable by EDF if and only if

$$\sum_i \frac{C_i}{T_i} \leq 1$$

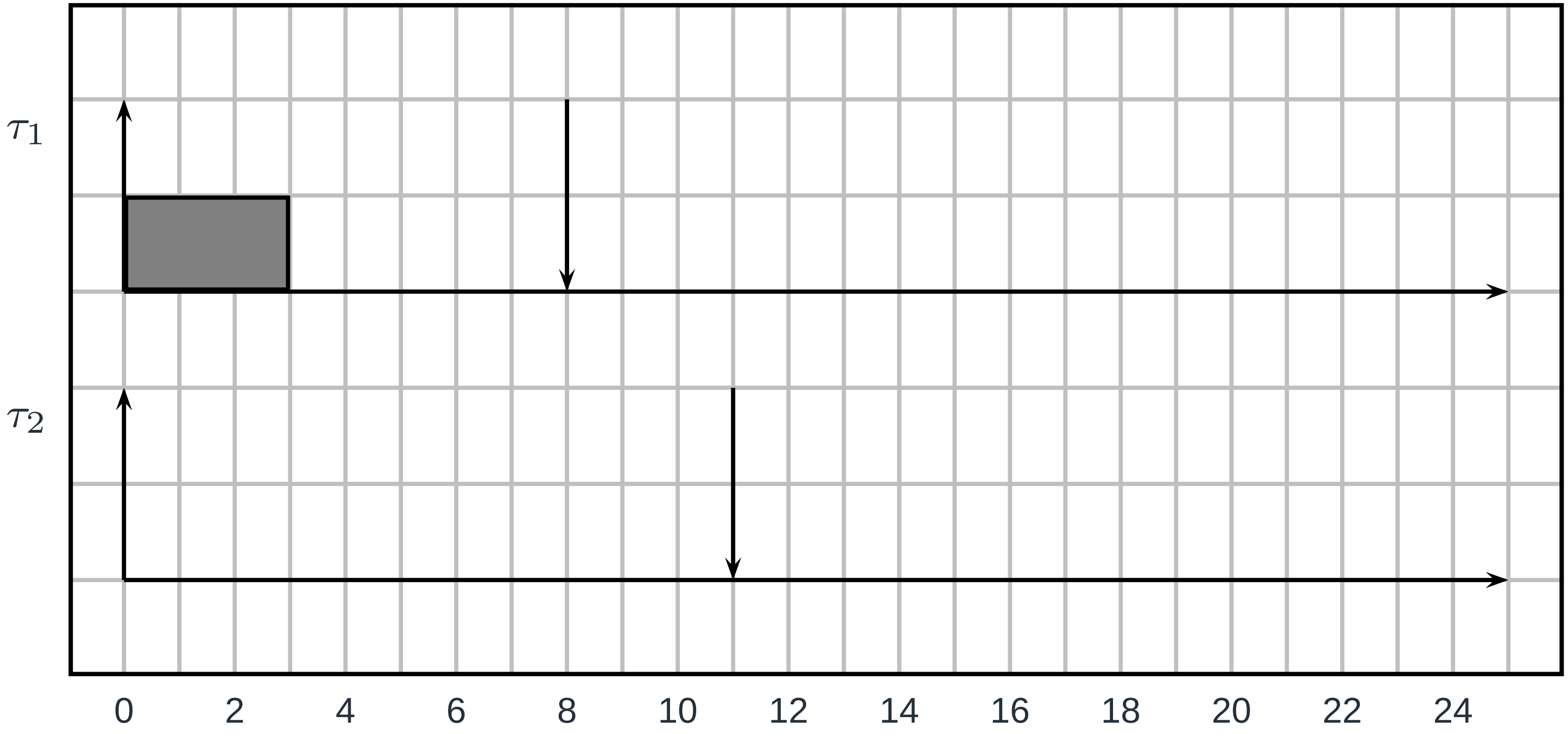
  - $U_{lub} = 1$!!!
- If $D_i \neq T_i$:
  - Processor demand approach or response time analysis can be applied to EDF too
  - But it is not obvious!

- $\tau_1 = (3, 8, 8), \tau_2 = (6, 11, 11) \Rightarrow U = 0.92$



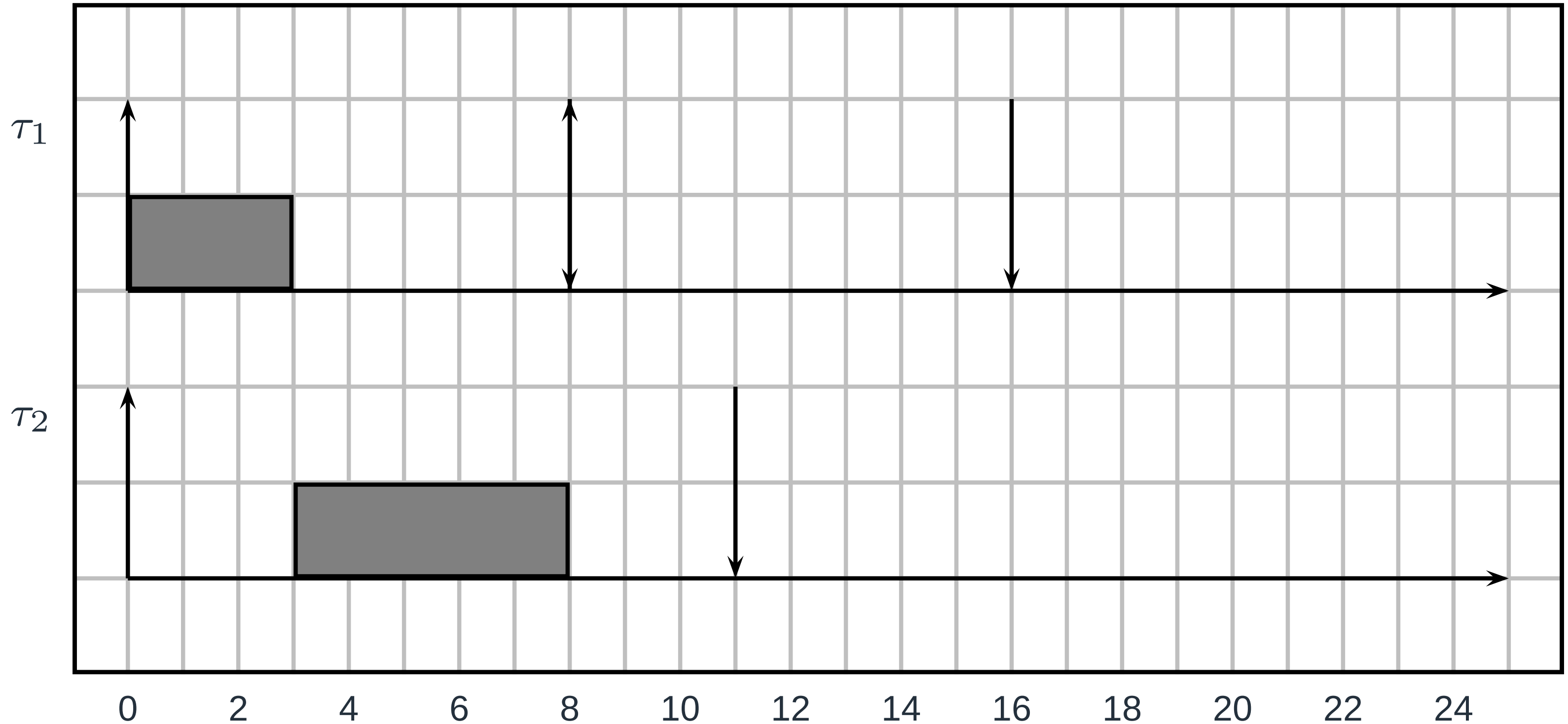
An Example – RM

- $\tau_1 = (3, 8, 8), \tau_2 = (6, 11, 11) \Rightarrow U = 0.92$



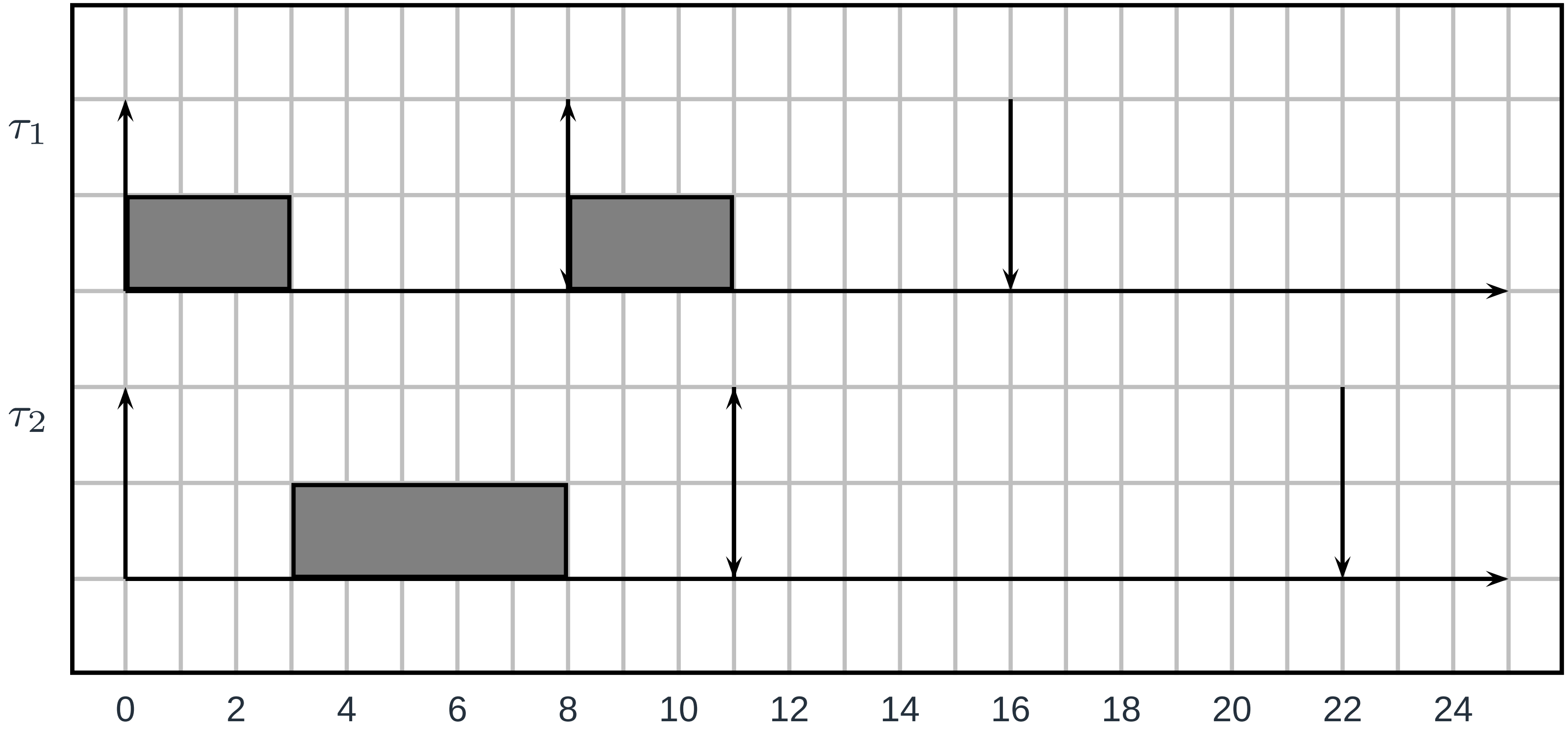
An Example – RM

- $\tau_1 = (3, 8, 8), \tau_2 = (6, 11, 11) \Rightarrow U = 0.92$

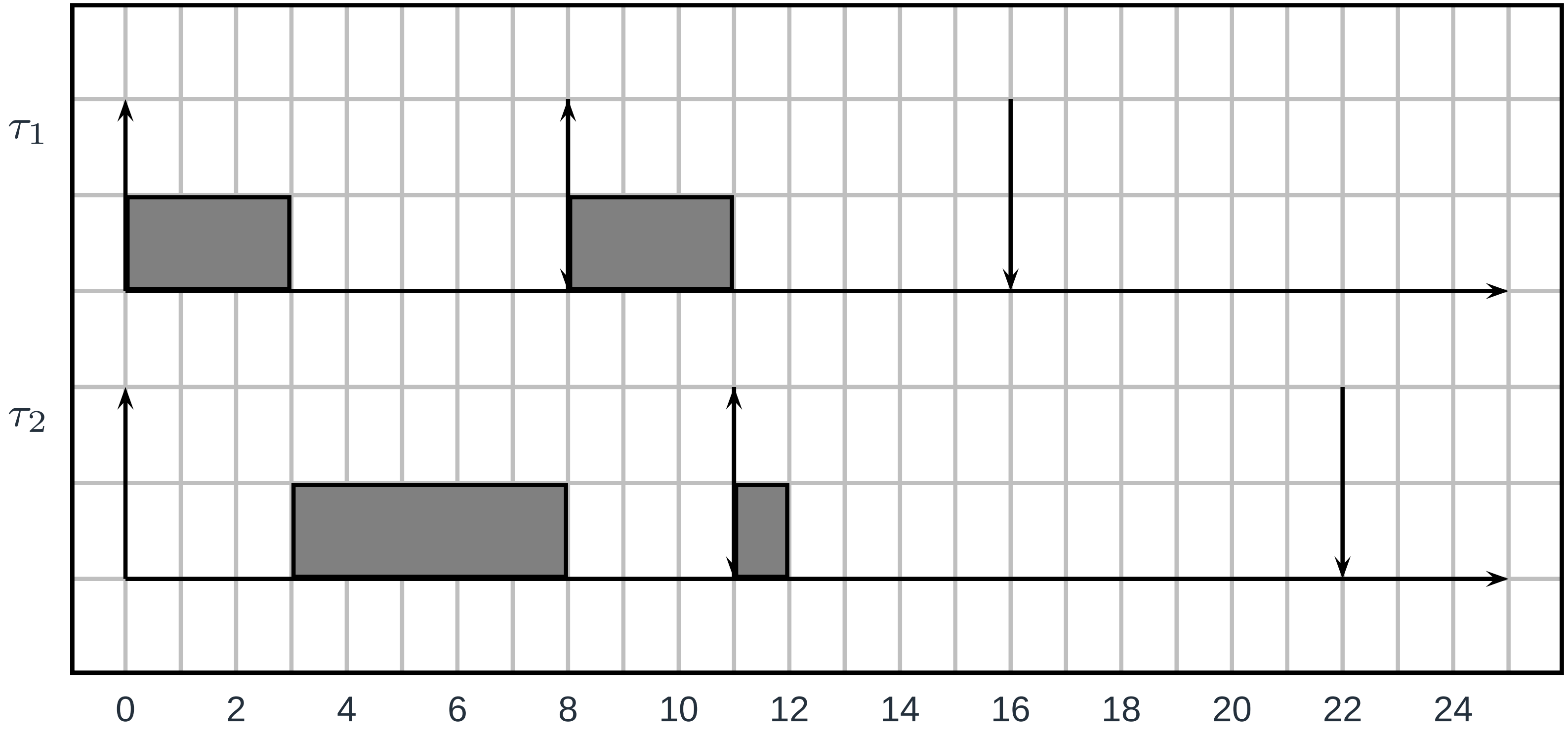


An Example – RM

- $\tau_1 = (3, 8, 8), \tau_2 = (6, 11, 11) \Rightarrow U = 0.92$

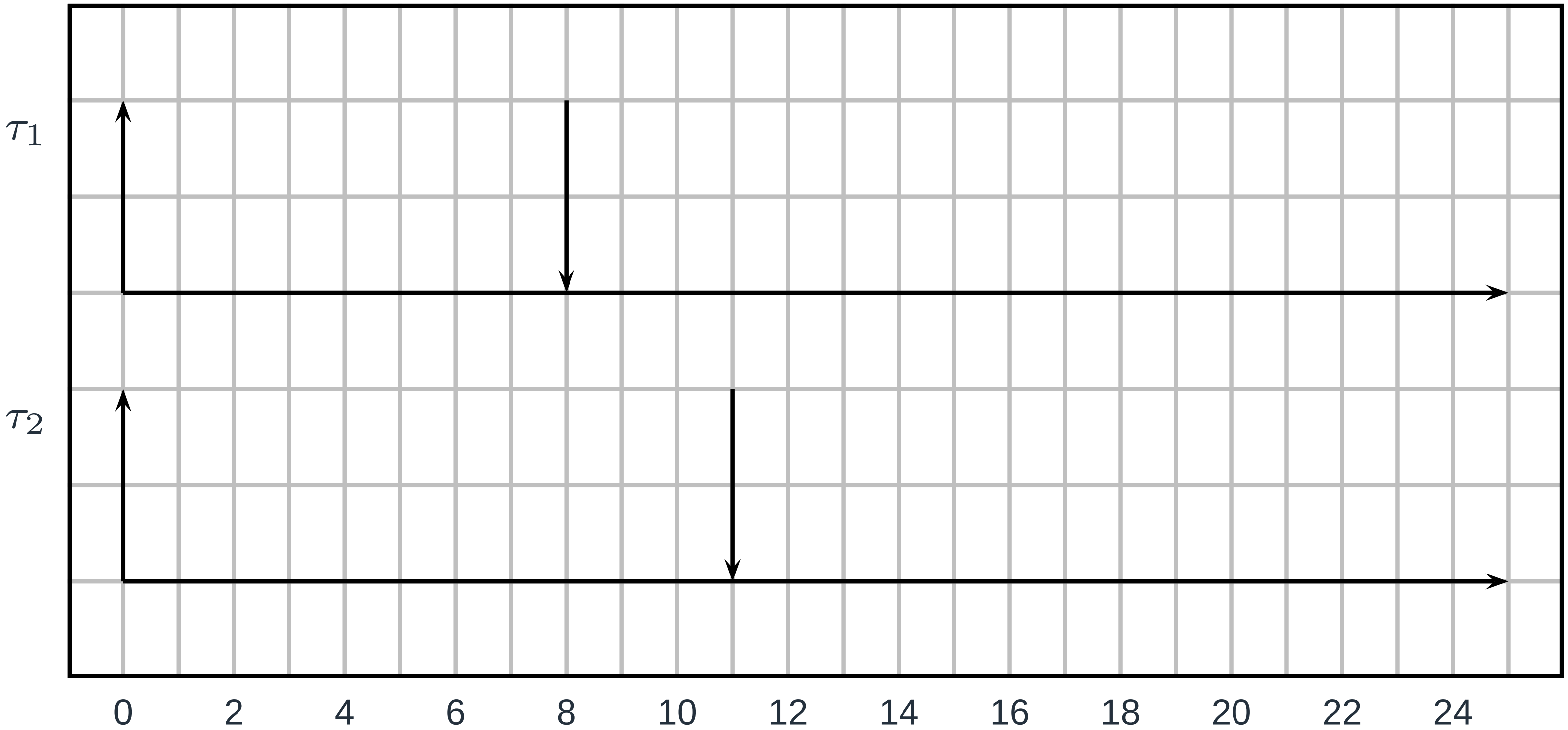


- $\tau_1 = (3, 8, 8), \tau_2 = (6, 11, 11) \Rightarrow U = 0.92$

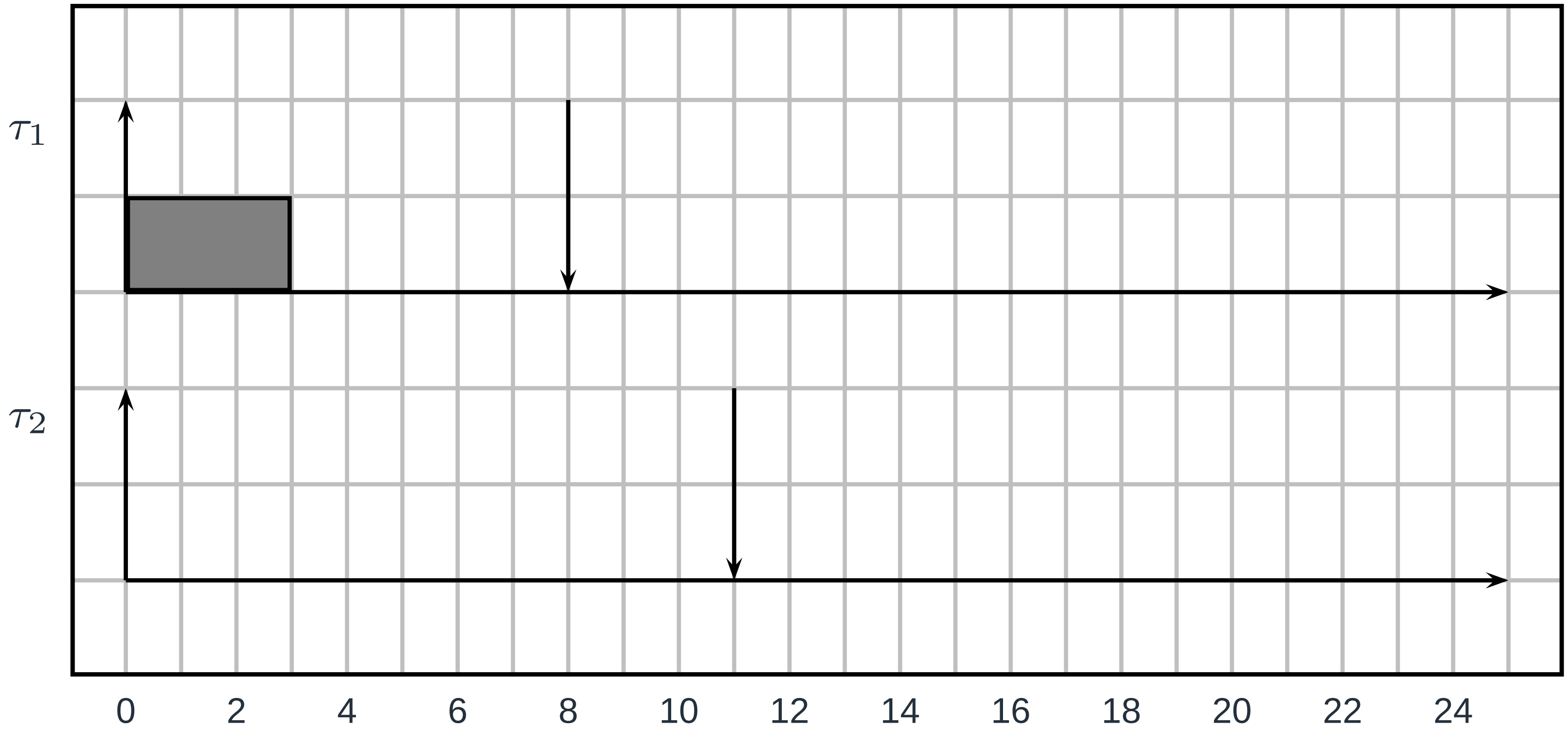


The Same Example – EDF

- $\tau_1 = (3, 8, 8), \tau_2 = (6, 11, 11) \Rightarrow U = 0.92$

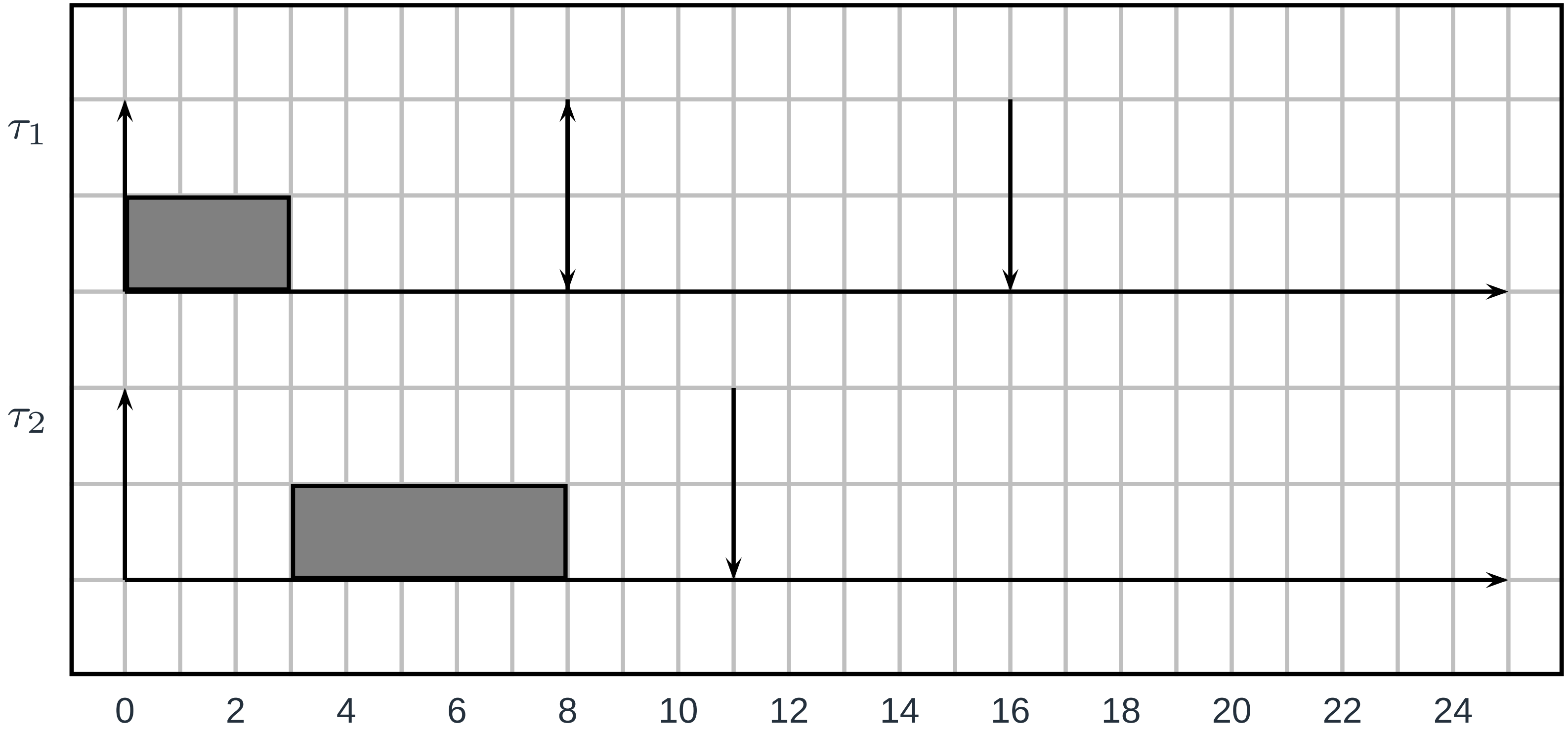


- $\tau_1 = (3, 8, 8), \tau_2 = (6, 11, 11) \Rightarrow U = 0.92$



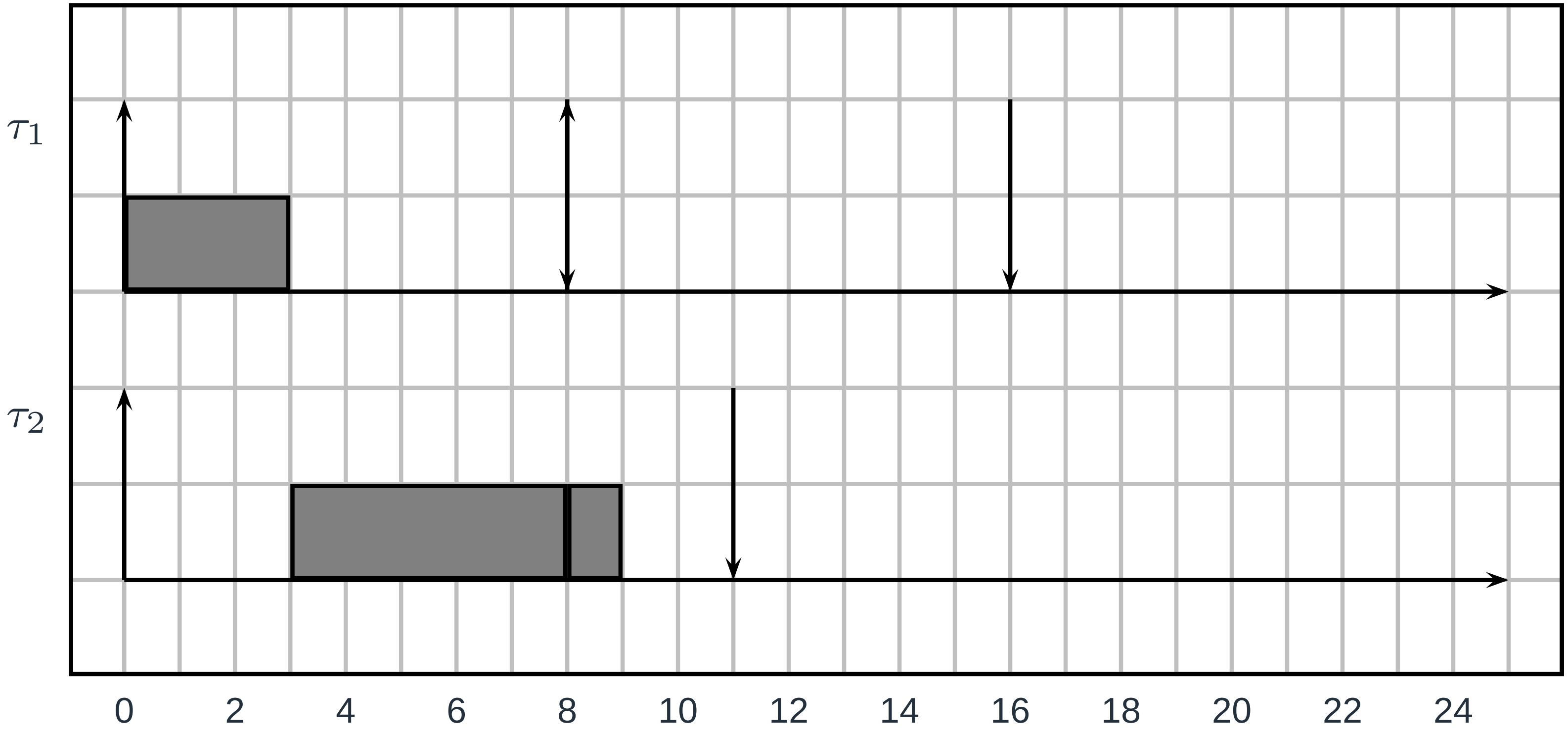
The Same Example – EDF

- $\tau_1 = (3, 8, 8), \tau_2 = (6, 11, 11) \Rightarrow U = 0.92$



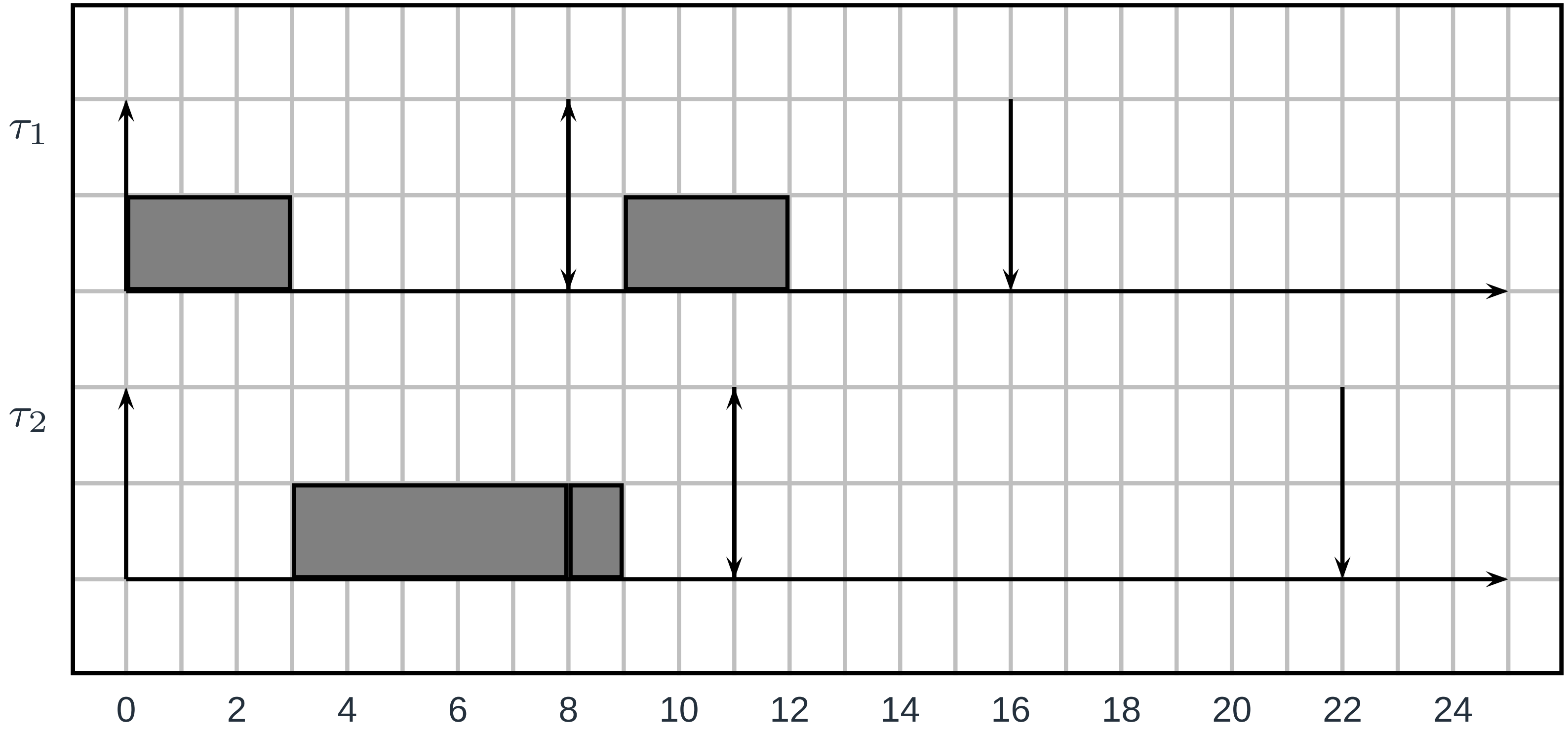
The Same Example – EDF

- $\tau_1 = (3, 8, 8), \tau_2 = (6, 11, 11) \Rightarrow U = 0.92$



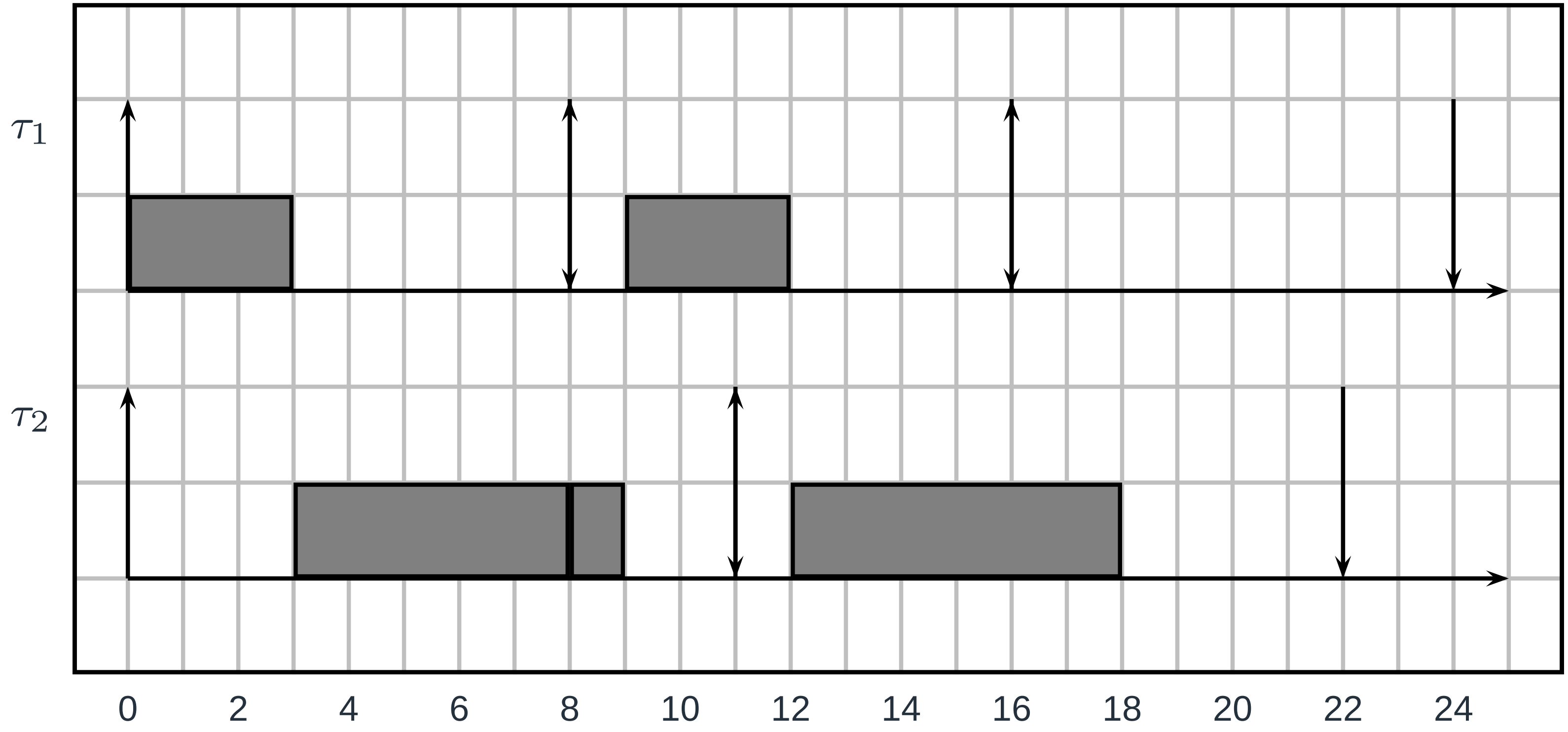
The Same Example – EDF

- $\tau_1 = (3, 8, 8), \tau_2 = (6, 11, 11) \Rightarrow U = 0.92$

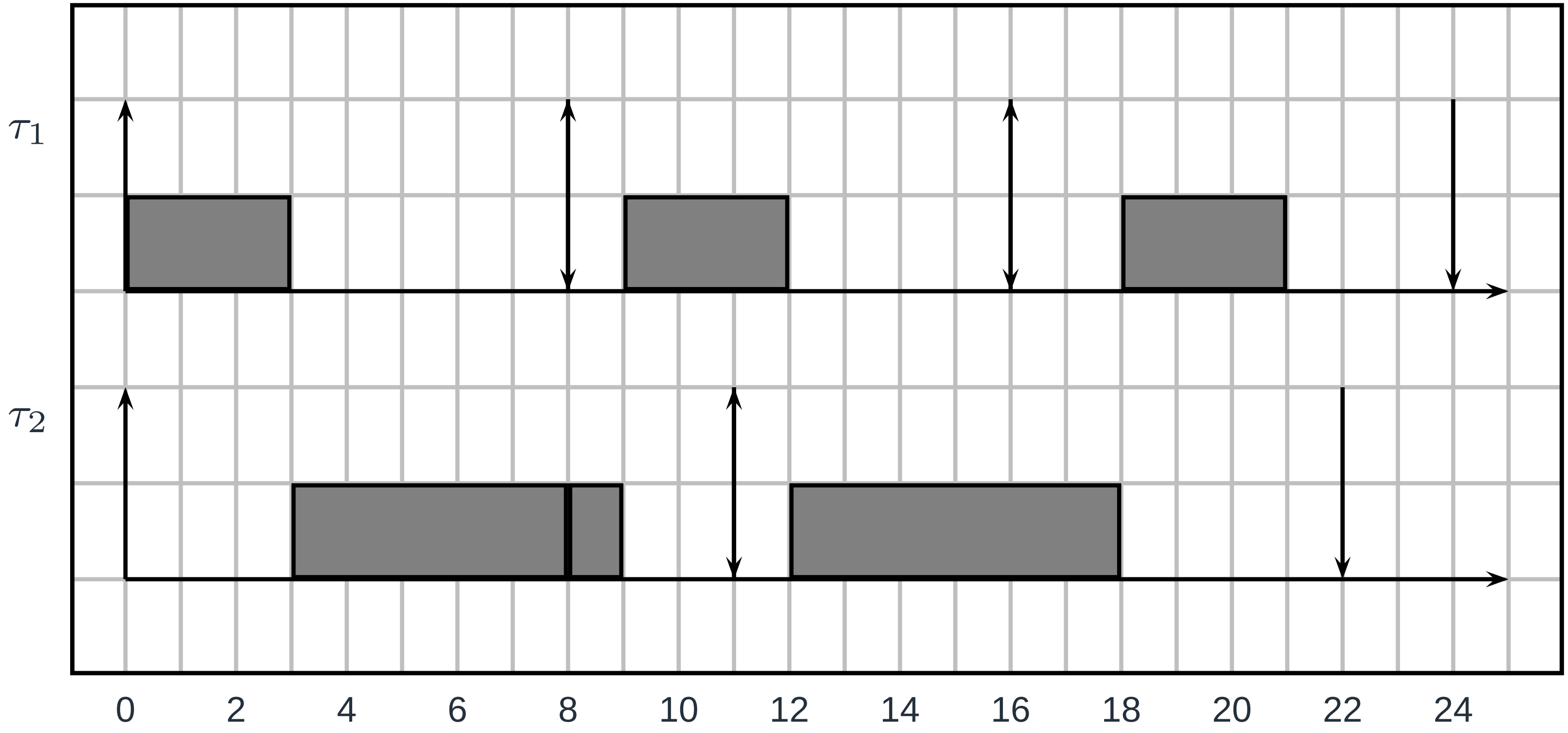


The Same Example – EDF

- $\tau_1 = (3, 8, 8), \tau_2 = (6, 11, 11) \Rightarrow U = 0.92$



- $\tau_1 = (3, 8, 8), \tau_2 = (6, 11, 11) \Rightarrow U = 0.92$



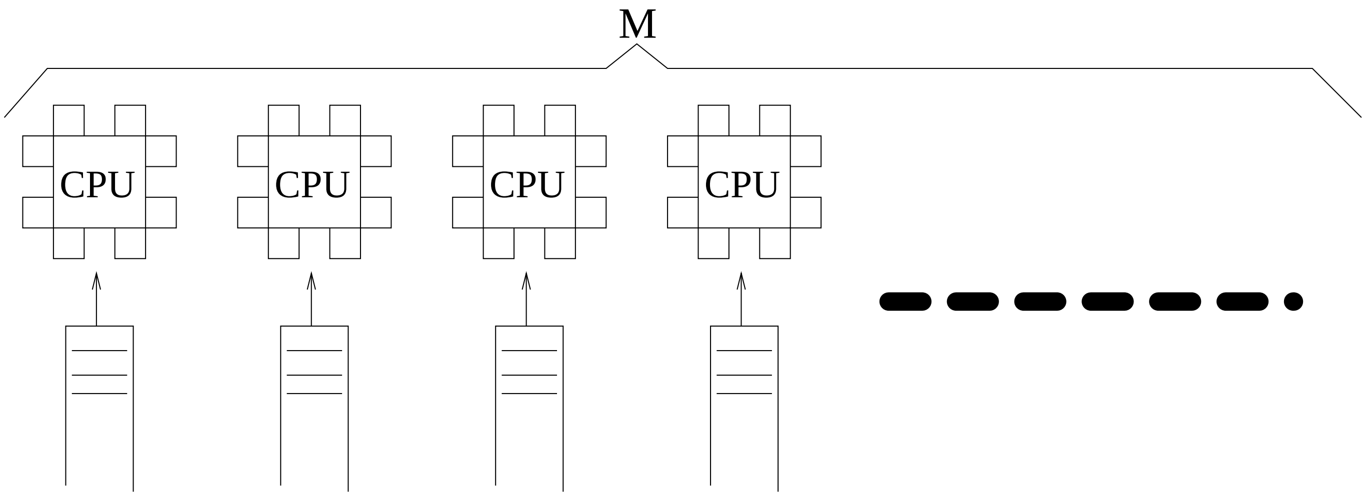
What About Multiple CPUs?

- UniProcessor Systems
  - A schedule $\sigma(t)$ is a function mapping time t into an executing task $\sigma : t \rightarrow \mathcal{T} \cup \{\tau_{idle}\}$ where $\mathcal{T}$ is the set of tasks running in the system
  - τ_{idle} is the *idle task*: when it is scheduled, the CPU becomes idle
- For a multiprocessor system with M CPUs, $\sigma(t)$ is extended to map t in vectors $\tau \in (\mathcal{T} \cup \{\tau_{idle}\})^M$
- How to implement a Real-Time scheduler for $M > 1$ processors?
  - Partitioned scheduling
  - Global scheduling

- UP Scheduling:
  - N periodic tasks with $D_i = T_i$: (C_i, T_i, T_i)
  - Optimal scheduler: if $\sum \frac{C_i}{T_i} \leq 1$, then the task set is schedulable
  - EDF is optimal
- Multiprocessor scheduling:
  - Goal: schedule periodic task sets with $\sum \frac{C_i}{T_i} \leq M$
  - Is this possible?
  - Optimal algorithms

Partitioned Scheduling - 1

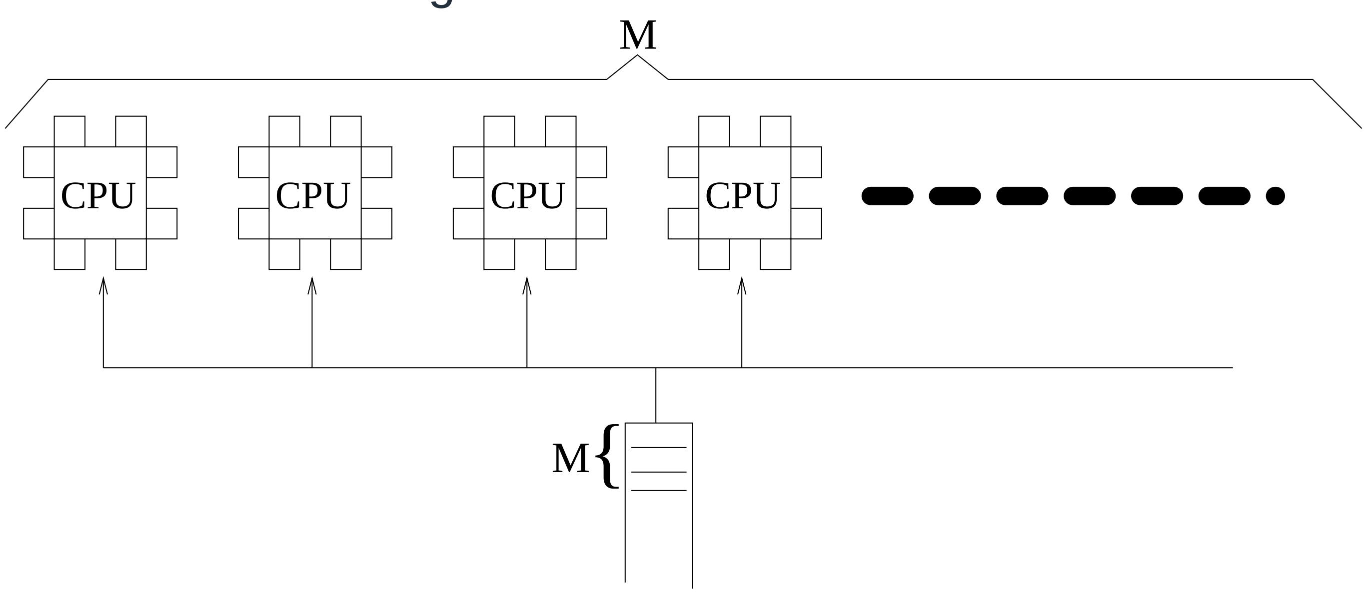
- Reduce $\sigma : t \rightarrow (\mathcal{T} \cup \{\tau_{idle}\})^M$ to M uniprocessor schedules $\sigma_p : t \rightarrow \mathcal{T} \cup \{\tau_{idle}\}, 0 \leq p < M$
  - Statically assign tasks to CPUs
  - Reduce the problem of scheduling on M CPUs to M instances of uniprocessor scheduling
  - Problem: system underutilisation



Partitioned Scheduling - 2

- Reduce an M CPUs scheduling problem to M single CPU scheduling problems and a bin-packing problem
- CPU schedulers: uni-processor, EDF can be used
- Bin-packing: assign tasks to CPUs so that every CPU has load ≤ 1
  - Is this possible?
- Think about 2 CPUs with $\{(6, 10, 10), (6, 10, 10), (6, 10, 10)\}$

- One single task queue, shared by M CPUs
  - The first M ready tasks from the queue are selected
  - What happens using fixed priorities (or EDF)?
  - Tasks are not bound to specific CPUs
  - Tasks can often migrate between different CPUs
- Problem: schedulers designed for UP do not work well



- Dhall's effect: U^{lub} for global multiprocessor scheduling can be quite low (for RM or EDF, converges to 1)
  - Pathological case: M CPUs, $M+1$ tasks. M tasks $(\epsilon, T-1, T-1)$, a task (T, T, T).
  - $U = M\frac{\epsilon}{T-1} + 1$. $\epsilon \to 0 \Rightarrow U \to 1$
- However, global EDF guarantees an upper bound for the *tardiness*!
  - Deadlines can be missed, but by a limited amount of time
- Global scheduling can cause a lot of useless migrations
  - Migrations are overhead!
  - Decrease in the throughput
  - Migrations are not accounted in admission tests...

Using Fixed Priorities in Linux

- SCHED_FIFO and SCHED_RR use fixed priorities
  - They can be used for real-time tasks, to implement RM and DM
  - Real-time tasks have priority over non real-time (SCHED_OTHER) tasks
- The difference between the two policies is visible when more tasks have the same priority
  - In real-time applications, try to avoid multiple tasks with the same priority

Setting the Scheduling Policy


```
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);

int sched_setscheduler(pid_t pid, int policy,
                        const struct sched_param *param);
int sched_setparam(pid_t pid,
                    const struct sched_param *param);
```

- If `pid == 0`, then the parameters of the running task are changed
- The only meaningful field of `struct sched_param` is `sched_priority`

Problems with Real-Time Priorities

- In general, "regular" (SCHED_OTHER) tasks are scheduled in background respect to real-time ones
- Real-time tasks can preempt / starve other applications
- Example: the following task scheduled at high priority can make a CPU / core unusable

```
void bad_bad_task()
{
    while(1);
}
```

  - Real-time computation have to be limited (use real-time priorities only when **really needed**!)
  - On sane systems, running applications with real-time priorities requires root privileges (or part of them!)

Real-Time Throttling

- A “bad” high-priority task can make a CPU / core unusable...
- ...Linux provides the *real-time throttling* mechanism to address this problem
  - How does real-time throttling interfere with real-time guarantees?
  - Given a priority assignment, a taskset is guaranteed all the deadlines if no throttling mechanism is used...
  - ...But, what happens in case of throttling?
- Very useful idea, but something more “theoretically founded” might be needed...

- Can EDF (or something similar) be supported in Linux?
- Problem: the kernel is (was?) not aware of tasks deadlines...
- ...But deadlines are needed in order to schedule the tasks
  - EDF assigns dynamic priorities based on absolute deadlines
- So, a more advanced API for the scheduler is needed...
  - Assign at least a relative deadline D_i to the task...
  - We will see that we need a *runtime* and a *period* too
- Moreover, $d_{i,j} = r_{i,j} + D_i$...
  - ...However, how can the scheduler know $r_{i,j}$?
  - The scheduler is not aware of jobs...

Tasks and Jobs... And Scheduling Deadlines!

- To use EDF, the scheduler must know when a job starts / finishes
  - Applications must be modified to signal the beginning / end of a job (some kind of `startjob()` / `endjob()` system call)...
  - ...Or the scheduler can assume that a new job arrives each time a task wakes up!
- Or, some other algorithm can be used to assign dynamic *scheduling deadlines* to tasks
  - Scheduling deadline d_i^s: assigned by the kernel to task τ_i
  - If the scheduling deadline d_i^s matches the absolute deadline $d_{i,j}$ of a job, then the scheduler can respect $d_{i,j}$!!!

- Constant Bandwidth Server (CBS): algorithm used to assign a dynamic scheduling deadline d_i^s to a task τ_i
- Based on the *Resource Reservation* paradigm
  - Task τ_i is periodically reserved a *maximum runtime* Q_i every *reservation period* P_i
- Temporal isolation between tasks
  - The worst case finishing time for a task does not depend on the other tasks running in the system...
  - ...Because the task is guaranteed to receive its reserved time
- Solves the issue with "bad tasks" trying to consume too much execution time

CBS: Some More Details

- Based on CPU reservations (Q_i, P_i)
  - If τ_i tries to execute for more than Q_i every P_i, the algorithm decreases its priority, or throttles it
  - τ_i consumes the same amount of CPU time consumed by a periodic task with WCET Q_i and period P_i
- Q_i/P_i: fraction of CPU time reserved to τ_i
- If EDF is used (based on the scheduling deadlines assigned by the CBS), then τ_i is guaranteed to receive Q_i time units every P_i if $\sum_j Q_j/P_j \leq 1$!!!
  - Only on uni-processor / partitioned systems...
  - M CPUs/cores with global scheduling: if $\sum_j Q_j/P_j \leq M$ each task is guaranteed to receive Q_i every P_i with a maximum delay

CBS vs Other Reservation Algorithms

- The CBS is based on EDF
  - Assigns scheduling deadlines d_i^s
  - EDF on $d_i^s \Rightarrow$ good CPU utilisation (optimal on UP!)
- The CBS allows to serve *non periodic tasks*
  - Some reservation-based schedulers have problems with aperiodic job arrivals - due to the (in)famous "deferrable server problem"
  - The CBS explicitly supports aperiodic arrivals (see the rule for assigning deadlines when a task wakes up)
- Allows to support "self-suspending" tasks
  - No need to strictly respect the Liu&Layland task model
  - No need to explicitly signal job arrivals / terminations

- Each task τ_i is associated a scheduling deadline d_i^s and a current runtime q_i
  - Both initialised to 0 when the task is created
- When a job arrives:
  - If the previous job is not finished yet, queue the activation
  - Otherwise, check if the current scheduling deadline can be used ($d_i^s > t$ and $q_i/(d_i^s - t) < Q_i/P_i$)
    - If not, $d_i^s = t + P_i$, $q_i = Q_i$
- When τ_i executes for a time δ, $q_i = q_i - \delta$
- When $q_i = 0$, τ_i cannot be scheduled (until time d_i^s)
  - At time d_i^s, $d_i^s = d_i^s + P_i$ and $q_i = q_i + Q_i$

- New `SCHED_DEADLINE` scheduling policy
  - Foreground respect to all of the other policies
- Uses the CBS to assign scheduling deadline to `SCHED_DEADLINE` tasks
  - Assign a (maximum) runtime Q_i and a (reservation) period P_i to `SCHED_DEADLINE` tasks
  - Additional parameter: relative deadline D_i
  - The "check if the current scheduling deadline can be used" rule is used at task wake-up
- Then uses EDF to schedule them
  - Both global EDF and partitioned EDF are possible
  - Configurable through the `cpuset` mechanism

- Juri will talk about the API, but...
- ...How to dimension the scheduling parameters?
  - (Maximum) runtime Q_i
  - (Reservation) period P_i
  - `SCHED_DEADLINE` also provides a (relative) deadline D_i
- Obviously, it must be

$$\sum_i \frac{Q_i}{P_i} \leq M$$

  - The kernel can do this admission control
  - Better to use a limit smaller than M (so that other tasks are not starved!)

- Temporal isolation
  - Each task can be guaranteed independently from the others
- Hard Schedulability property
  - If $Q_i \geq C_i$ and $P_i \leq T_i$ (maximum runtime larger than WCET, and server period smaller than task period)...
  - ...Then the scheduling deadlines are equal to the jobs' deadlines!!!
  - All deadlines are guaranteed to be respected (on UP / partitioned systems), or an upper bound for the tardiness is provided (if global scheduling is used)!!!
- So, `SCHED_DEADLINE` can be used to serve hard real-time tasks!

What About Soft Real-Time?

- What happens if $Q_i < C_i$, or $P_i > T_i$?
  - $\frac{Q_i}{P_i}$ must be larger than the ratio between average execution time $\overline{c_i}$ and average inter-arrival time $\overline{t_i}$...
  - ...Otherwise, $d_i^s \rightarrow \infty$ and there will be no control on the task's response times
- Possible to do some stochastic analysis (Markov chains, etc...)
  - Given $\overline{c_i} < Q_i < C_i, T_i = nP_i$, and the probability distributions of execution and inter-arrival times...
  - ...It is possible to find the probability distribution of the response times (and the probability to miss a deadline)!

Changing Parameters...

- Tasks' parameters (execution and inter-arrival times) can change during the tasks lifetime... So, how to dimension Q_i and P_i?
- Short-term variations: CPU reclaiming mechanisms (GRUB, ...)
  - If a job does not consume all of the runtime Q_i, maybe the residual runtime can be used by other tasks...
- Long-term variations: adaptive reservations
  - Generally "slower", can be implemented by a user-space daemon
  - Monitor the difference between d_i^s and $d_{i,j}$
    - If $d_i^s - d_{i,j}$ increases, Q_i needs to be increased
    - If $d_i^s - d_{i,j} \leq 0$, Q_i can be decreased
- Lot of literature for both of these approaches

Things I did not Mention...

- What about interacting tasks (shared resources, IPC, ...)?
  - Inheritance (priority inheritance, deadline inheritance, BandWidth Inheritance)
  - Juri will probably say something about this...
- Is the kernel able to respect the theoretical schedule?
  - What happens if a task is scheduled later than expected?
  - Kernel Latency!!!
  - This is what Preempt-RT is for...
    - Preempt-RT and `SCHED_DEADLINE`: two orthogonal approaches that can (and must) be combined
- Optimality with multiple CPUs?