➢ Commonly Used Approaches to Real-Time Scheduling

CLOCK-DRIVEN:

- Primarily used for hard real-time systems where all properties of all jobs are known at design time, such that offline scheduling techniques can be used

WEIGHTED ROUND-ROBIN:

- Primarily used for scheduling real-time traffic in high-speed, switched networks

PRIORITY-DRIVEN:

- Primarily used for more dynamic real-time systems with a mix of time based and event-based activities, where the system must adapt to changing conditions and events.

Decisions about what jobs execute when are made at specific time Instants-

– These instants are chosen before the system begins execution
– Usually regularly spaced, implemented using a periodic timer interrupt
• Scheduler awakes after each interrupt, schedules the job to execute for the next period, then blocks itself until the next interrupt
• E.g. the helicopter example with an interrupt every 1/180th of a second
• E.g. the furnace control example, with an interrupt every 100ms

CLOCK-DRIVEN APPROACH:

- when scheduling is clock-driven (also called time-driven), decisions on what jobs execute at what times are made at **specific time instants**.
- These instants are chosen a **priori** before the system begins execution.
- Typically, in a system that uses clock-driven scheduling, **all the parameters** of hard real-time jobs are **fixed and known.**
- A schedule of the jobs is computed offline and is stored for use at run time.
- The scheduler schedules the jobs according to this schedule at each scheduling decision time.
- In this way, scheduling overhead during run-time can be minimized.
- A frequently adopted choice is to make scheduling decisions at regularly spaced time instants.
  - ➢ One way to implement a scheduler that makes scheduling decisions periodically is to use a hardware timer.
  - ➢ The timer is set to expire periodically without the intervention of the scheduler. When the system is initialized, the scheduler selects and schedules the job(s) that will execute until the next scheduling decision time and then blocks itself waiting for the expiration of the timer. When the timer expires, the scheduler awakes and repeats these actions.

Typically in clock-driven systems:

– All parameters of the real-time jobs are fixed and known
– A schedule of the jobs is computed off-line and is stored for use at runtime; as a result, scheduling overhead at run-time can be minimized
– Simple and straight-forward, not flexible

Regular round-robin scheduling is commonly used for scheduling Time-shared applications:

– Every job joins a FIFO queue when it is ready for execution
– When the scheduler runs, it schedules the job at the head of the queue to execute for at most onetime slice

- Sometimes called a quantum – typically O (*tens of ms)*

– If the job has not completed by the end of its quantum, it is preempted and placed at the end of the queue
– When there are *n* ready jobs in the queue, each job gets one slice every *n* time slices (*n* time slices is called a round)
– Only limited use in real-time systems

➢ Weighted Round-Robin Scheduling

❖ **In *weighted round robin* each job *J* i is assigned a weight *w* i; the job will receive *w* i consecutive time slices each round, and the duration of a round is** $\sum_{i=1}^{n} w_i$

– Equivalent to regular round robin if all weights equal 1
– Simple to implement, since it doesn't require a sorted priority queue

• Partitions capacity between jobs according to some ratio
• Offers throughput guarantees

– Each job makes a certain amount of progress each round

• By giving each job a fixed fraction of the processor time, a round robin scheduler may delay the completion of every job

– A precedence constrained job may be assigned processor time, even while it waits for its predecessor to complete; a job can't take the time assigned to its successor to finish earlier

➢ Priority-Driven Scheduling

• Assign priorities to jobs, based on some algorithm
• Make scheduling decisions based on the priorities, when events such as releases and job completions occur

– Priority scheduling algorithms are *event-driven.*
– Jobs are placed in one or more queues; at each event, the ready job with the highest priority is executed.

– The assignment of jobs to priority queues, along with rules such a whether preemption is allowed, completely defines a priority scheduling algorithm

• **Priority-driven algorithms make *locally optimal* decisions about which job to run**

– Locally optimal scheduling decisions are often *not globally optimal*

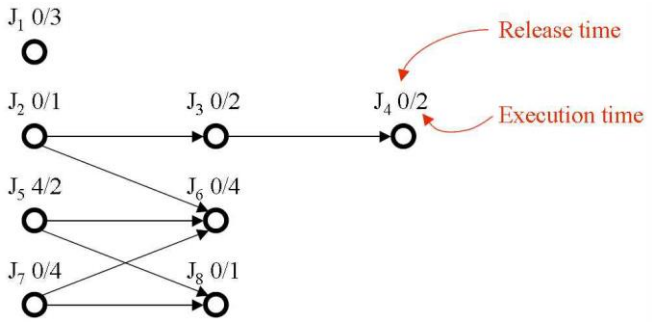
– Priority-driven algorithms *never* intentionally leave any resource idle

• Leaving a resource idle is not locally optimal

Example:

Consider the following task:

– Jobs *J* 1, *J* 2, …, *J* 8, where *J* i had higher priority than *J* k if $i < k$



– Jobs are scheduled on two processors *P* 1 *and P* 2

– Jobs communicate via shared memory, so communication cost is negligible

– The schedulers keep one common priority queue of ready jobs

• **Schedulers**

– Preemptable – scheduling decisions are made whenever some job becomes ready for execution or a job completes

– Non-preemptable – scheduling decision are delayed until a running job completes.

Most scheduling algorithms used in *non real-time systems are* priority-driven

– First-In-First-Out → Assign priority based on release time

– Last-In-First-Out

– Shortest-Execution-Time-First → Assign priority based on execution time

– Longest-Execution-Time-First

• **Real-time priority scheduling assigns priorities based on deadline or some other *timing constraint:***

– Earliest deadline first

– Least slack time first

– Etc.

Earliest deadline first (EDF)

– Assign priority to jobs based on deadline

– Earlier the deadline, higher the priority

– Simple, just requires knowledge of deadlines

• **Least Slack Time first (LST)**

– A job *J* i has deadline *d* i, execution time *e* i, and was released at time *r* i

– At time *t* < *d* i*:*

• Remaining execution time *t* rem = *e* i - (*t* – *r* i)

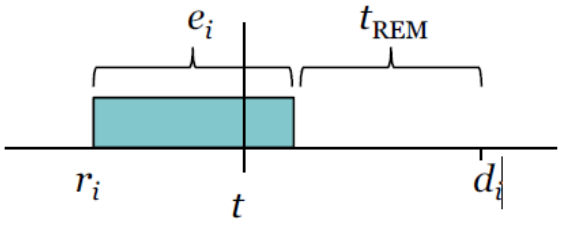
• Slack time *t* slack = *d* i – *t* – *t* rem

– Assign priority to jobs based on slack time, *t* slack

– The smaller the slack time, the higher the priority

– More complex, requires knowledge of execution times and deadlines

• Knowing the actual execution time is often difficult a priori, since it depends on the data, need to use worst case estimates (⇒ poor performance)



These algorithms are optimal

– i.e. they will always produce a feasible schedule if one exists

– Constraints: on a single processor, as long as preemption is allowed and jobs do not contend for resources.

➢ Dynamic vs. Static Systems

If jobs are scheduled on multiple processors, and a job can be dispatched from the priority run queue to any of the processors, the system is *dynamic*

• A job *migrates* if it starts execution on one processor and is resumed on a different processor

• If jobs are partitioned into subsystems, and each subsystem is bound statically to a processor, we have a *static* system

• Expect static systems to have inferior performance (in terms of overall response time of the jobs) relative to dynamic systems

– But it is possible to validate static systems, whereas this is not always true for dynamic systems

– For this reason, most *hard* real time systems are static

➢ Effective Release Times and Deadlines

• Sometimes the release time of a job may be later than that of its successors, or its deadline may be earlier than that specified for its predecessors

• This makes no sense: derive an *effective release time* or *effective deadline* consistent with all precedence constraints, and schedule using that

– Effective release time

• If a job has no predecessors, its effective release time is its release time

• If it has predecessors, its effective release time is the maximum of its release time and the effective release times of its predecessors

– Effective deadline

• If a job has no successors, its effective deadline is its deadline

• It if has successors, its effective deadline is the minimum of its deadline and the effective deadline of its successors

OFF-LINE VERSUS ON-LINE SCHEDULING:

➢ A clock-driven scheduler typically makes use of a precomputed schedule of all hard real-time jobs. This schedule is computed **off-line** before the system begins to execute, and the computation is based on the knowledge of the release times and processor-time/resource requirements of all the jobs for all times.

➢ It is the classical scheduling setting where we have a set of jobs and we need to define a proper scheduling of these jobs on a machine environment in order to optimize a certain objective function. So an offline algorithm is analyzed on its capacity to obtain optimal solutions.

➢ When the operation mode of the system changes, the new schedule specifying when each job in the new mode executes is also precomputed and stored for use. In this case, we say that scheduling is (done) off-line, and the precomputed schedules are off-line schedules.

➢ **Disadvantage** of off-line scheduling is inflexibility. This approach is possible only when the system is deterministic, meaning that the system provides some fixed set(s) of functions and that the release times and processor-time/resource demands of all its jobs are known and do not vary or vary only slightly.

➢ we use an **on-line scheduling** algorithm, if the scheduler makes each scheduling decision without knowledge about the jobs that will be released in the future; the parameters of each job become known to the on-line scheduler only after the job is released.

➢ The priority-driven algorithms described earlier are on-line algorithms.

➢ Obviously, on-line scheduling is the only option in a system whose future workload is unpredictable. An on-line scheduler can accommodate dynamic variations in user demands and resource availability. The price of the flexibility and adaptability is a reduced ability for the scheduler to make the best use of system resources. Without prior knowledge about future jobs, the scheduler cannot make optimal scheduling decisions.