University of Glasgow | School of Computing Science

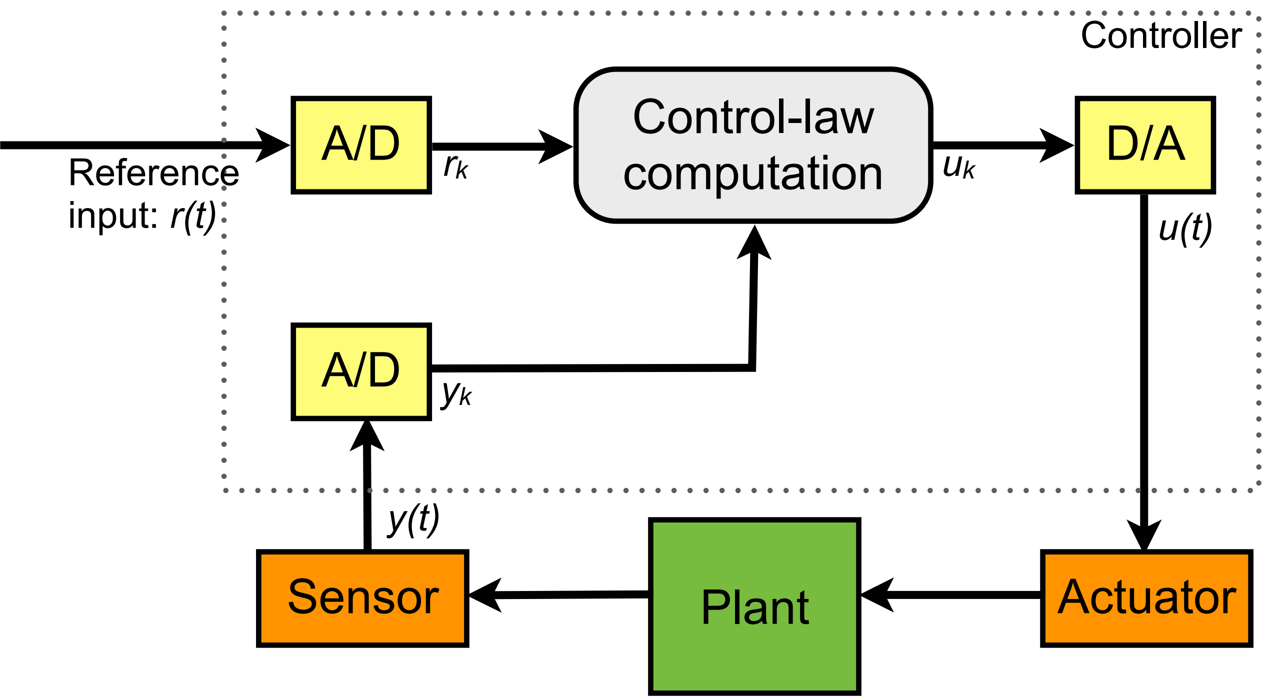
Introduction to Real-time Systems

Advanced Operating Systems (M)
Lecture 2

Introduction to Real-time Systems

- Real-time systems deliver services while meeting some timing constraints
  - Not necessarily fast, but must meet some timing deadline
  - Many real-time systems are embedded as part of some larger device or system
    - Washing machine, photocopier, mobile phone, car, aircraft, industrial plant, etc.
    - Representative classes: digital process control; telephony and multimedia
- Frequent requirement to validate for correctness
  - Many embedded real-time systems are safety critical: if they do not complete in a timely and correct basis, serious consequences result
  - Bugs in embedded real-time systems can often be difficult or expensive to fix: you can't just run "software update" on a car!

Example: Digital Process Control

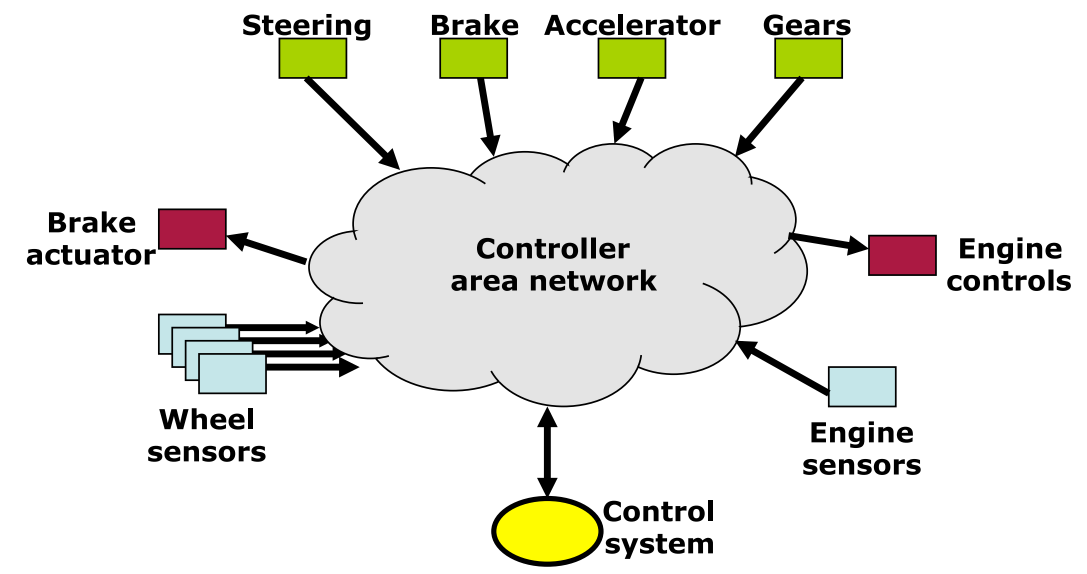



```
set timer to interrupt periodically with period T;
at each timer interrupt, do
  do analogue-to-digital conversion of y(t) to get yₖ;
  compute control output uₖ based on reference rₖ and yₖ;
  do digital-to-analogue conversion of uₖ to get u(t);
end do;
```

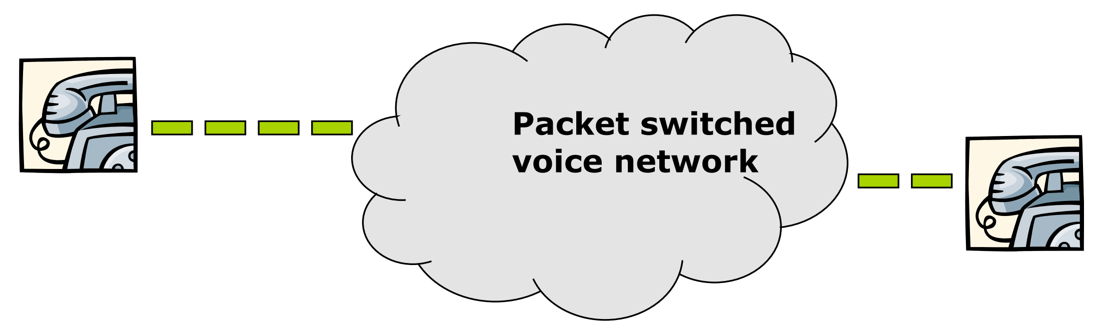
- Controlling some device ("the plant") using an actuator, based on sampled sensor data
- Effective control depends on correct control law computation, reference input, and accuracy of measurements
- Time between measurements of $y(t)$, $r(t)$ is the sampling period, T
  - Small T better approximates analogue control but large T needs less processor time; if T is too large, oscillation will result as the system fails to keep up with changes in the input
- A simple control loop is conceptually simple to implement
  - Complexity comes from multiple control loops running at different rates, and from systems that must switch between different modes of operation

Examples: Drive-by-Wire and Telephony

- Real-time systems are increasingly built as distributed systems
- The components of the system are connected via some communications network
  - E.g., a sensor that sends data to the controller process over a local area network, perhaps as part of a drive by wire car
  - E.g., a voice-over-IP telephony system, where real-time speech data is transferred over a wide area IP network such as the Internet
- These systems not only need to run a control law under time constraints, but must also schedule communications according to deadlines



Example: drive-by-wire controls in a car



Example: voice-over-IP

Types of Real-Time System

- Purely cyclic
  - Every task executes periodically
  - Demands in resources (e.g., computing, communication, or storage) do not vary significantly from period to period
  - Example: most digital controllers and real-time monitors
- Mostly cyclic
  - Most tasks execute periodically
  - The system must also respond to some external events asynchronously (e.g., fault recovery and external commands)
  - Example: modern avionics and process control systems
- Asynchronous, mostly predictable
  - Most tasks are not periodic
  - The time between consecutive executions of a task may vary considerably, or the variations in resource utilisation in different periods may be large
  - These variations have either bounded ranges or known statistics
- Asynchronous, unpredictable
  - Applications that react to external events and/or have tasks with high and variable run-time complexity
  - Example: intelligent real-time control

Easier to reason about systems that are more cyclic, synchronous, and predictable

Implementation Considerations

- Some real-time embedded systems are complex, implemented on high-performance hardware
  - E.g., industrial plant control, avionics and flight control systems
- But, many are implemented on hardware that is low cost, low power, and low performance, but light-weight and robust
  - E.g., consumer goods
  - Often implemented in C or assembler, fitting within a few kilobytes of memory; correctness primary concern, efficiency a close second
- We are interested in proofs of correctness of the scheduling, and ways of raising the level of abstraction when programming such systems

Reference Model for Real-time Systems

- A reference model and consistent terminology let us reason about real-time systems
- Reference model is characterised by:
  - A model that describes the applications running on the system
  - A model that describes the resources available to those applications
  - Scheduling algorithms that define how the applications execute and use the resources

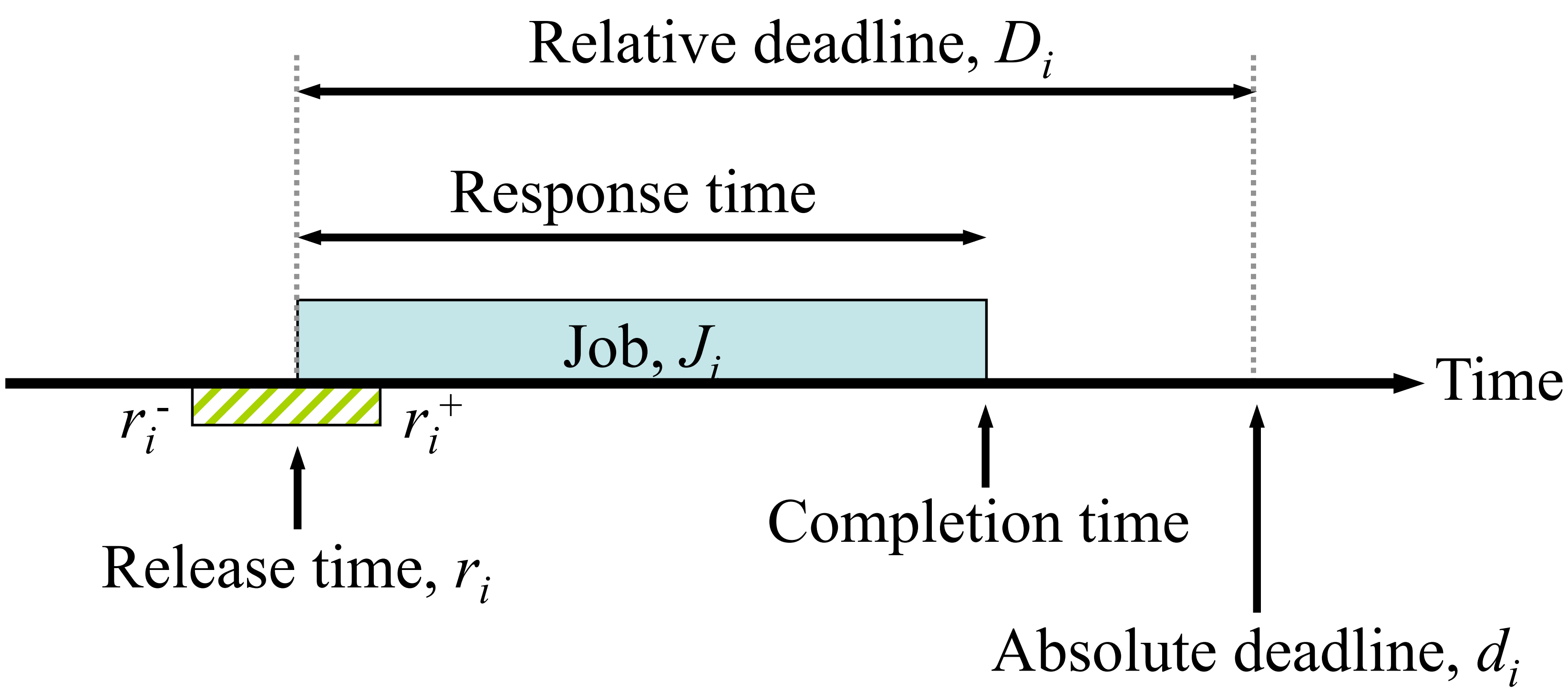
Jobs, Tasks, Processors, and Resources

- A *job* is a unit of work scheduled and executed by the system;
  - A task, T, is a set of related jobs, $J_1, J_2, \ldots, J_n$ that jointly provide some function
  - If jobs occur on a regular cycle, the task is termed periodic
  - if jobs are unpredictable, the task is termed aperiodic (or sporadic, if the jobs have deadlines once released)
- Jobs execute on a *processor* and may depend on some *resources*
- Processors are active devices on which jobs are scheduled
  - E.g., threads scheduled on a *CPU*, data scheduled on a *transmission link*
  - A processor has a *speed* attribute, that determines the rate of progress of jobs executing on that processor
- A resource, R, is a passive entity on which jobs may depend
  - E.g., system memory, a hardware device
  - Resources may have different types and sizes, but do not have a speed attribute
  - Resources are not consumed by usage, and can be reused multiple times
  - Jobs compete for access to resources, and may block if the resource is in use by another job
  - A resource is *plentiful* if there is enough of it that nothing blocks waiting access – such resources can't affect correctness, and so are generally ignored

Execution Time of Jobs

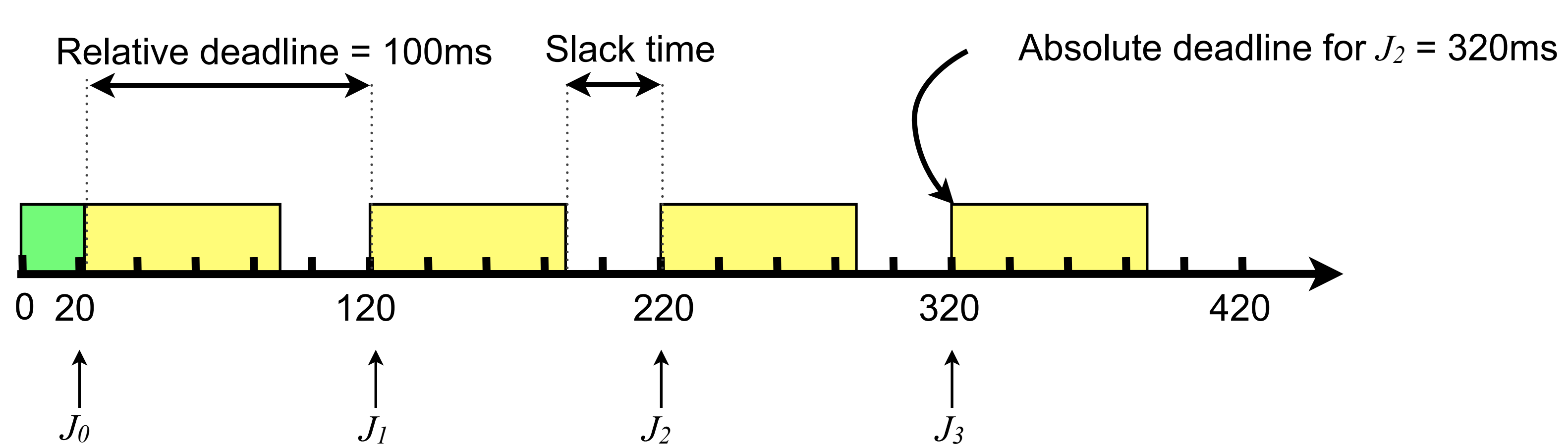
- A job J_i will execute for time e_i
  - This is the amount of time required to complete execution of J_i when it executes alone on the processor, and has all the resources it needs
  - The value of e_i depends on the complexity of the job and the speed of the processor; it may vary on a given processor due to conditional branches in the job, the effects of processor caches, etc.
    - Execution times therefore fall into an interval $[e_i^-, e_i^+]$; assume we know this interval for every real-time job, but not necessarily the actual e_i
    - Terminology: $(x, y]$ is an interval starting immediately after x, continuing up to and including y
    - Often, assume $e_i = e_i^+$ and validate using worst-case execution times: inefficient, but safe

Deadlines & Timing Constraints



Deadlines & Timing Constraints: Example

- A system to monitor and control a heating furnace
  - The system takes 20ms to initialise when turned on
  - After initialisation, every 100ms, the system:
    - Samples and reads the temperature sensor
    - Computes the control-law for the furnace to process the temperature readings, determine the correct flow rates of fuel, air, and coolant
    - Adjusts the flow rates to match the computed values
  - The system can be modelled as a task, T, comprising jobs $J_0, J_1, \ldots, J_k, \ldots$
    - The release time of J_k is 20 + ($k \times$ 100)ms
    - The relative deadline of J_k is 100ms; the absolute deadline is 20 + ((k + 1) $\times$ 100)ms



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
  - Makes no sense: derive 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

Hard vs. Soft Real-time Systems

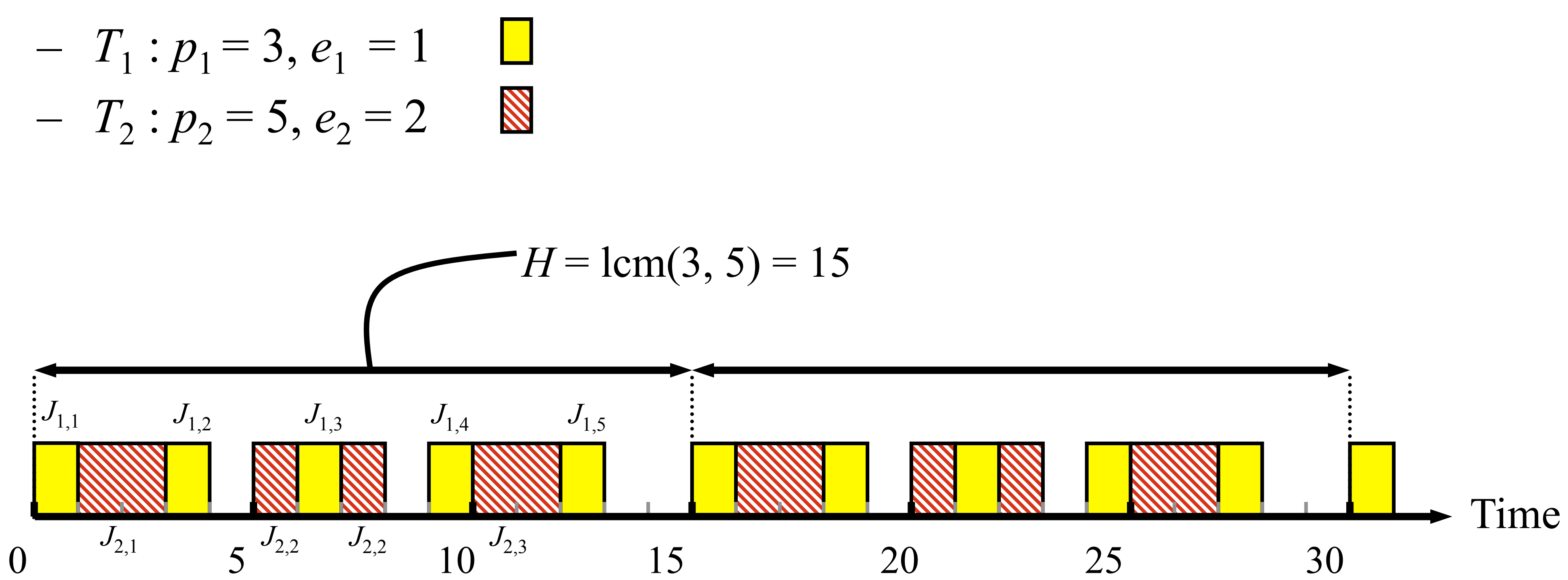
- The firmness of timing constraints affects how we engineer the system
  - If a job must never miss its deadline, the system is *hard real-time*
    - A timing constraint is hard is failure to meet it is considered a fatal error
    - A timing constraint is hard if the usefulness of the results falls off abruptly at the deadline
    - A timing constraint is hard if the user requires validation (formal proof or exhaustive simulation, potentially with legal penalties) that the system always meets the constraint
  - If some deadlines can be missed occasionally, with low probability, then the system is described as *soft real-time*
- Hard and soft real-time are two ends of a spectrum
  - In many practical systems, the constraints are probabilistic, and depend on the likelihood and consequences of failure
  - No system is guaranteed to *always* meet its deadlines: there is always some probability of failure

Periodic Tasks

- A set of jobs that are executed at regular time intervals can be modelled as a *periodic task* – many real-world systems fit this model
  - Each periodic task T_i is a sequence of jobs $J_{i,1}, J_{i,2}, \ldots, J_{i,n}$
  - The phase, φ_i, of task T_i is the release time $r_{i,1}$ of the first job $J_{i,1}$
  - The period, p_i, of task T_i is the minimum length of time between release times of consecutive jobs
  - The execution time, e_i, of task T_i is the maximum execution time of all jobs in the task
  - The utilisation of task T_i is $u_i = e_i / p_i$ and measures the fraction of time for which the task executes
  - The total utilisation of a system $U = \sum_i u_i$

Periodic Tasks: Example

- $T_1 : p_1 = 3, e_1 = 1$
- $T_2 : p_2 = 5, e_2 = 2$



A system of periodic tasks repeats after the hyper-period, $H = \text{lcm}(p_i)$ for $i = 1, 2, \ldots, n$

Aperiodic and Sporadic Tasks

- Many real-time systems are required to respond to unpredictable events
- These are modelled as *aperiodic* or *sporadic* jobs
  - An aperiodic job has no deadline; a sporadic job has a deadline once released
  - It is often possible to characterise the inter-arrival times for such jobs according to some probability distribution
- The presence of aperiodic and sporadic jobs greatly complicates reasoning about a system
  - Sporadic tasks make the design of a hard real-time system impossible, unless some bounds can be placed on their inter-arrival times and relative deadlines

Dynamic vs. Static Systems

- A multiprocessor system is *dynamic* if the jobs can migrate between processors; it is *static* if (sets of) jobs are bound to a single processor
- Expect static systems to have inferior performance (in terms of overall response job time) compared to dynamic systems
    - But it is possible to validate static systems, whereas this is not always true for dynamic systems; hence, most hard real time systems are static
    - Results demonstrated for uniprocessor systems are applicable to each processor of a static multiprocessor system; they are not necessarily applicable to dynamic multiprocessor systems

Overview of Real-time Scheduling

- Jobs are scheduled and allocated access to resources according to a scheduling algorithm and some resource access control protocol
- A *valid schedule* satisfies the following conditions:
  - Every processor is assigned at most one job at any time; every job is assigned to at most one processor at once
  - No job is scheduled before its release time
  - The total amount of processor time assigned to each job is equal to its maximum or actual execution time
  - All the precedence and resource usage constraints are satisfied
- A schedule is *feasible* if it's valid and every job meets its timing constraints
- A scheduling algorithm is *optimal* if it always produces a feasible schedule for a given set of jobs if a feasible schedule exists
  - There are some scheduling algorithms that will find some, but not all, feasible schedules, and so may fail to schedule a set of jobs that some other algorithm could schedule

Real-time Scheduling Algorithms

- Two main classes of algorithm for scheduling real-time tasks:
  - *Clock-driven* algorithms are used for mostly static systems, where all properties of all jobs are known at design time, such that offline scheduling techniques can be used
  - *Priority-driven* algorithms are used for more dynamic systems, with a mix of periodic tasks and event-driven (aperiodic and/or sporadic tasks), where the system must adapt to changing events and conditions

- Lecture 3: clock-driven scheduling
- Lectures 4-7: priority-driven scheduling

Further Reading

- Will focus on real-time scheduling in the next few lectures

- Recommended reading:
  - Jane W. S. Liu, "Real-Time Systems", Prentice Hall, 2000, ISBN 0130996513