Analysis for Scheduling

- some type of performance (i.e. hard-time constraints, spatial precision) are critical and a scheduling algorithm must *guarantee* that these conditions be met.
- analytical techniques compute worst-case performance with which to characterize processes - mostly computational processes.
- empirical techniques observe a statistically relevant number of samples in order to bound the probability that performance will not meet constraints - mostly physical processes.
- off-line analysis is often used to check for circumstances which violate constraints.
- a system is *unschedulable* if it cannot schedule tasks in order to meet all constraints.
- **necessary condition:** scheduler answers "no" when a situation exists for which deadlines cannot be guaranteed.
- **sufficient condition:** scheduler answers "yes" when all constraints can be satisfied.
- an **exact** analysis meets both necessary and sufficient conditions.

Analysis for Scheduling - (cont.)

...how much time does a process require to complete

Empirical Approaches

- the process is executed with an attached measuring device — independent, dedicated process that logs performance
- the process is observed over a *large* variety of operating contexts
- we assume that the performance bounds observed are, in fact, the worst-case performance
- **no guarantees** — finite sampling implies that we may only decrease the probability that our performance estimates are incorrect
- arguably the only way to handle performance constraints uniformly across computational and physical processes

Analyzing Performance

- automatically compute worst-case performance characteristics (temporal, spatial, functional)
- model-based
- can be overly **pessimistic** — average performance can be significantly better than worst-case
- only way to produce unequivocal guarantees

Example: Temporal Analysis for Scheduling

Measuring Execution Times

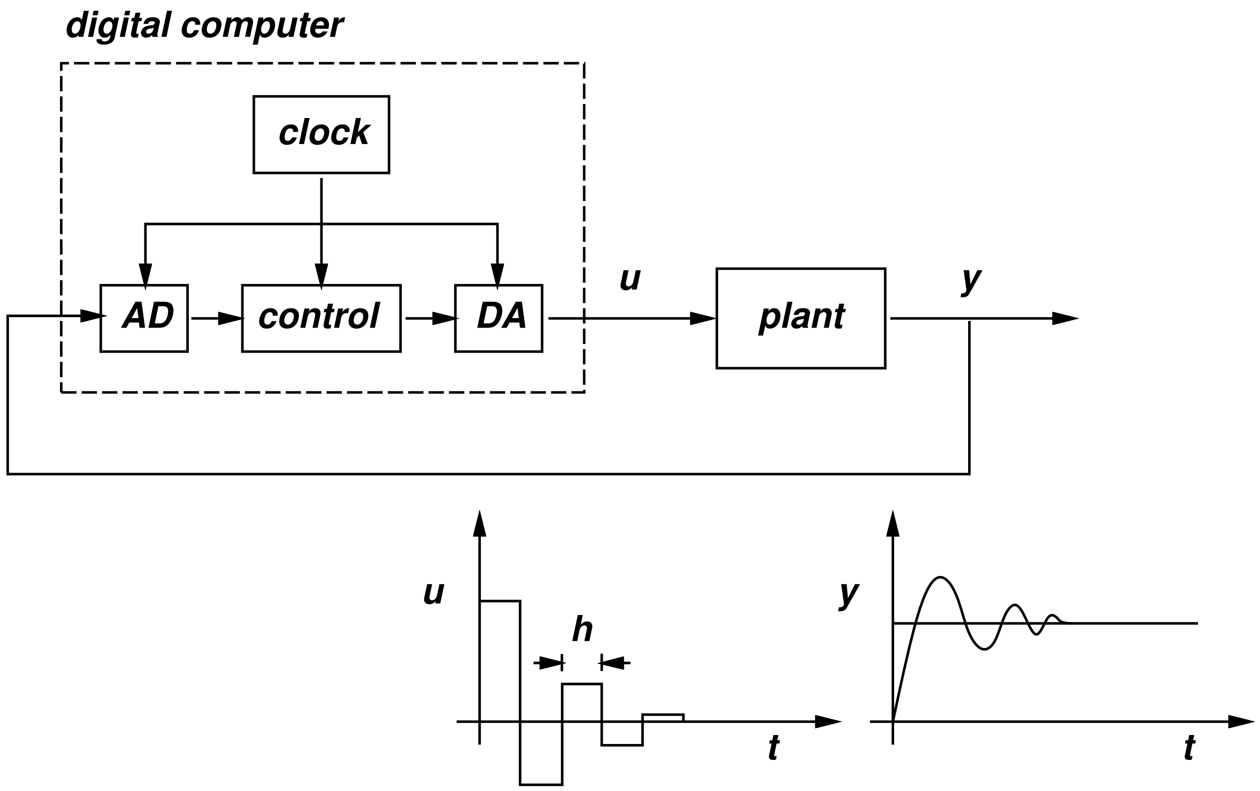
- execution time is data dependent
- cacheing — leads to non-determinism that is overcome only by disabling the cache
- pipelining — leads to non-determinism that is overcome by disabling pipelining
- shared memory multi-processors compete for memory on the bus — execution times depend on global context

Closed-form Execution Times

- use machine code instruction time tables from manufacturer
- compiler/architecture dependent
- branching statements — worst case
- loops annotated with maximum number of loops permitted
- recursion disallowed — difficult to predict maximum depth of recursion required
- dynamic memory allocation difficult to predict
- "goto" data flows difficult to work out

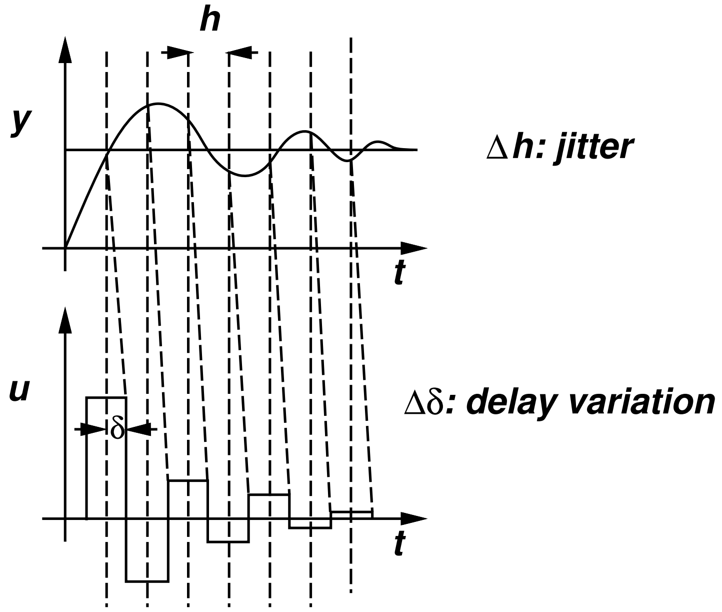
Periodic Tasks Revisited

A feedback control process is the canonical periodic task with a fixed period — the sampling rate.



between sampling intervals, the system is ballistic

Periodic Tasks Revisited



in practive, the control delay is not constant — variations are due to:

- preemptions from higher priority tasks
- variation in the execution times (branching, data dependencies)
- communication delays in distributed controllers

jitter can result from:

- preemptions from higher priority tasks
- improper useage of timing primitives in the RT kernel

Periodic Tasks Revisited

```
LOOP
   wait for clk interrupt
   read analog in
   compute control
   update control parameters
   write analog out
END
```

control latency

the control latency can be modeled as a pure delay — performance is improved if the delay is small (better servo rate) and constant (minimizes *jitter*)

```
LOOP
   wait for clk interrupt
   read analog in
   compute control
   write analog out
END
update control parameters
```

control latency

Periodic Tasks Revisited

variation in control delay and jitter can be handled in two ways:

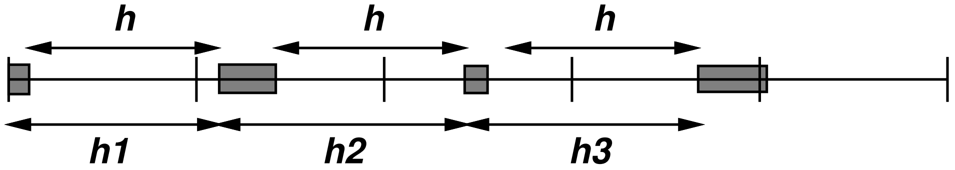
- Control Engineering:
  - robust control design - performance is proven acceptable for bounded delay and jitter *a priori*
- Computer Engineering:
  - design RT system to be deterministic

Example: Constant Sampling Rate h

Alternative #1:

```
LOOP
   regulate
   wait_time(h)
END
```

won't work
sampling rate $> h$ and varying



Periodic Tasks Revisited

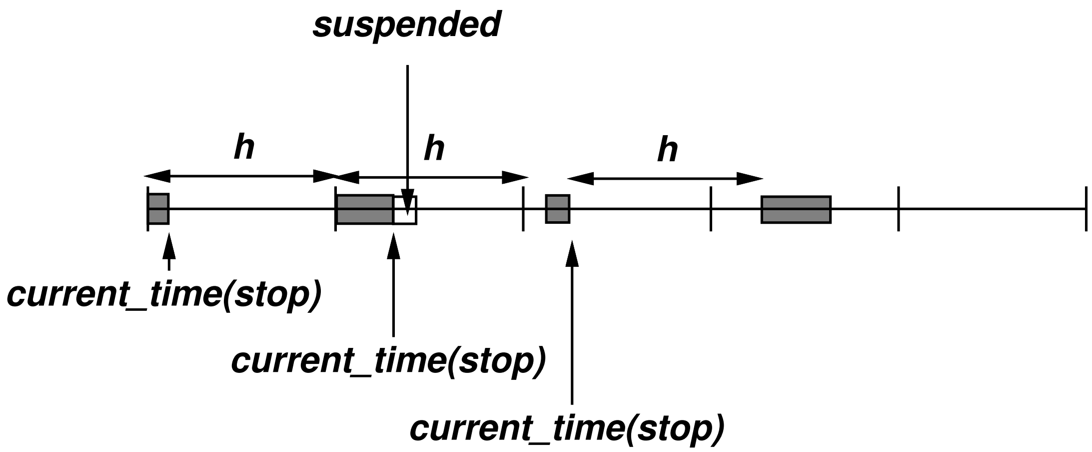
Example: Constant Sampling Rate h

Alternative #2:

```
LOOP
   current_time(start)
   regulate
   current_time(stop)
   wait_time(h - difftime(stop, start))
END
```

won't work

- an interrupt can occur between current_time() and wait_time()
- diff_time() not (typically) implemented in kernel, and is itself pre-emptable



Periodic Tasks Revisited

Example: Constant Sampling Rate h

Alternative #3:

```
current_time(t)
LOOP
   regulate
   inc_time(t, h)
   wait_until(t)
END
```

works!