SYSC 5701
Operating System Methods for Real-Time Applications

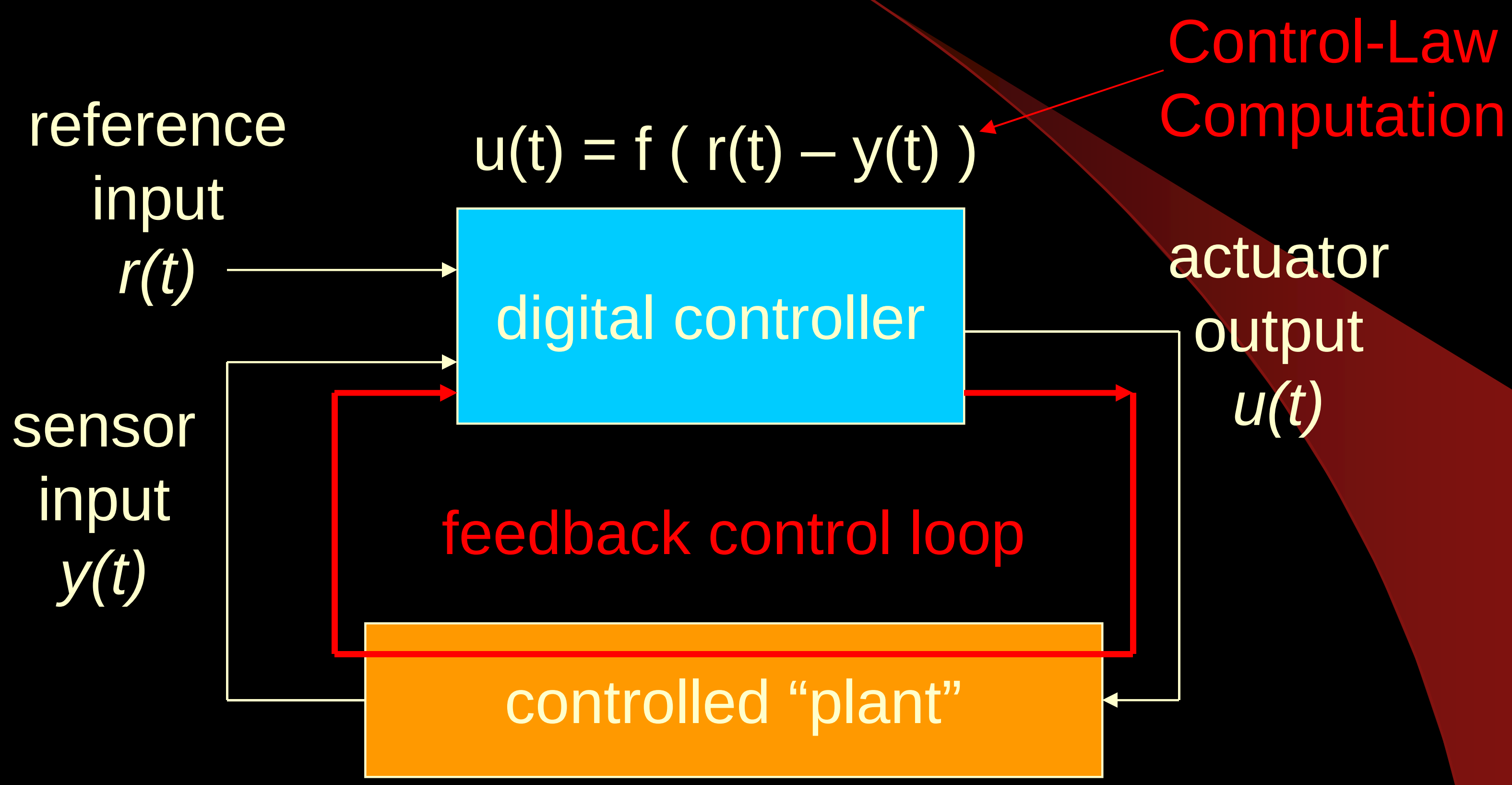
Real-Time (RT) Systems

Winter 2014

for those following Liu's text:

- **Ch. 1**: Typical RT Applications
  - digital controllers ("motivation")
- **Ch. 2**: Hard vs. Soft RT Systems
  - jobs, processors, timing
- **Ch.3**: Reference Model of RT Systems
  - basis for subsequent chapters

Simple Digital Controller

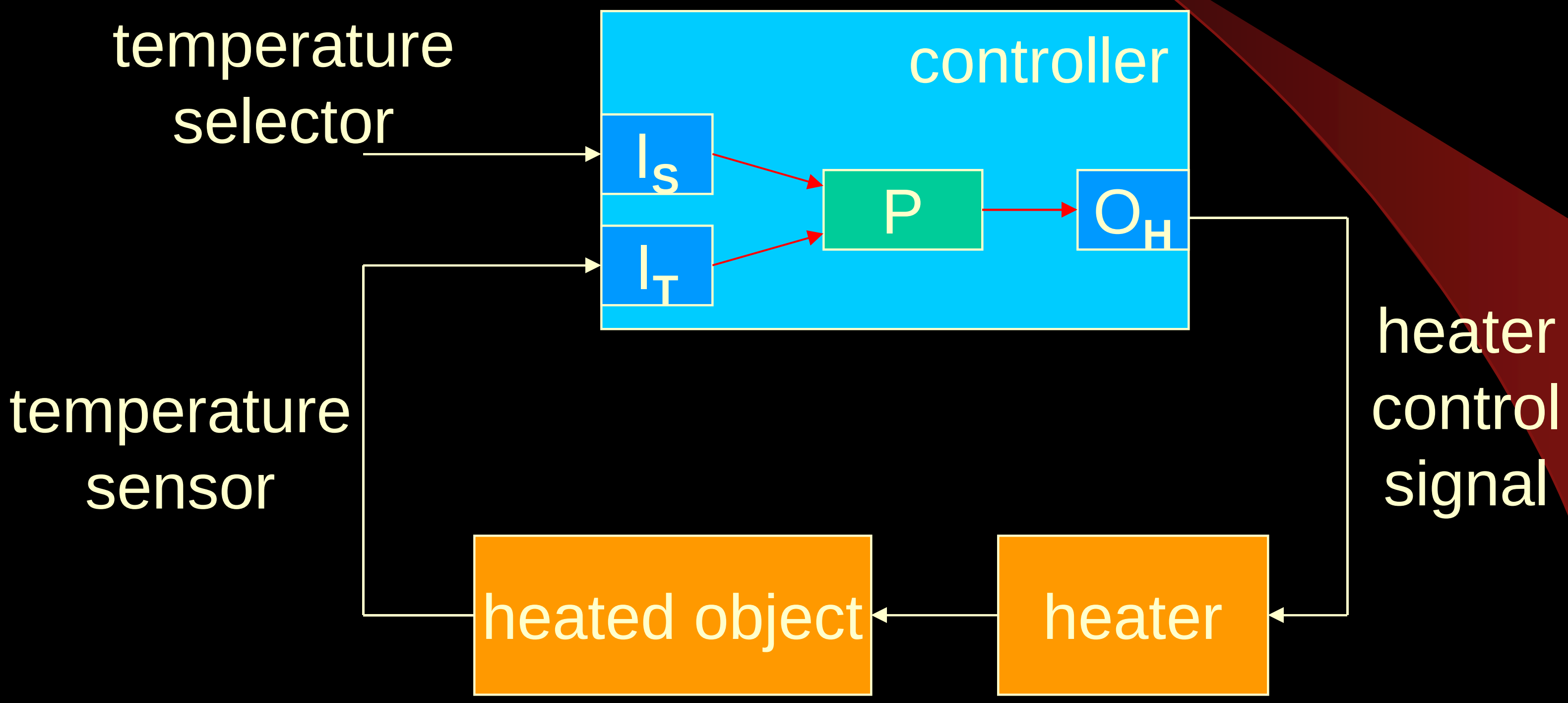


Digital Control

repeatedly:

1. sample inputs $r(t_i)$ and $y(t_i)$
   – requires input hardware (e.g. A to D)
2. calculate control-law computation → $u(t_i)$
   – requires processor
3. generate output $u(t_i)$
   – requires output hardware (e.g. D to A)

Temperature Controller Example



Temperature Controller Activities

active components

1. sample inputs

   sample I_S sample I_T ← I/O hardware

2. calculate control-law computation

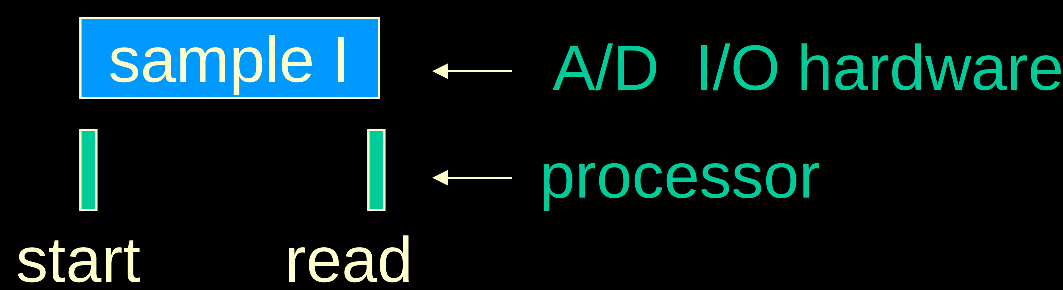
   calculate ← processor

3. generate output

   generate O_H ← I/O hardware

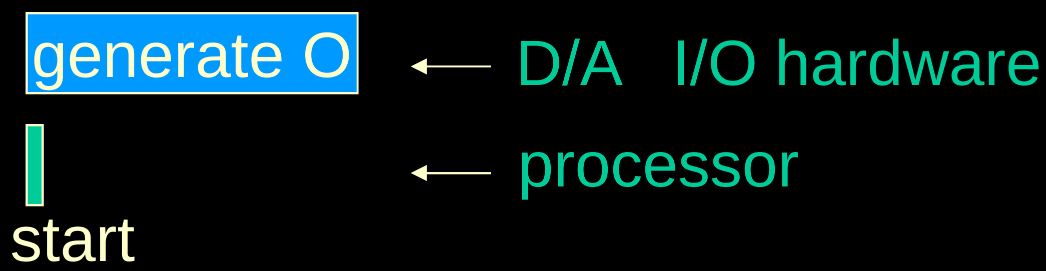
Processor in Sampling Input?

- assume inputs via A/D converters
- processor must write *start* command to begin an A/D conversion
- processor must read digital value when conversion complete



Processor in Generating Output?

- assume output via D/A converter
- processor must be sure converter is not busy when starting a new conversion
- processor must write: data to begin a D/A conversion



Solution 1 <u>Sequential: Go Fast!</u>

do forever

{ poll I_S for selector input: start → poll → read

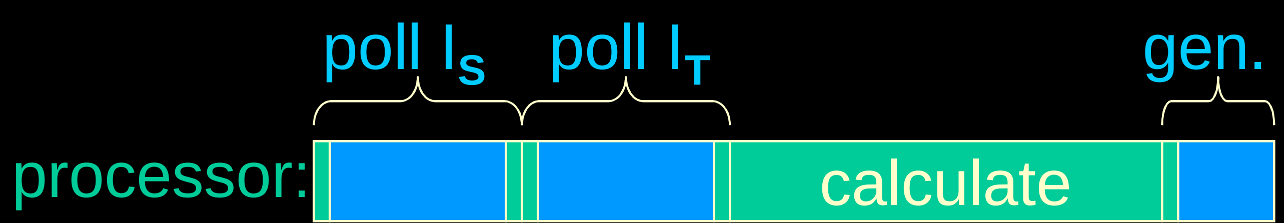
poll I_T for temperature input: (start poll read)

calculate control-law computation

wait (poll) for O_H hardware ready

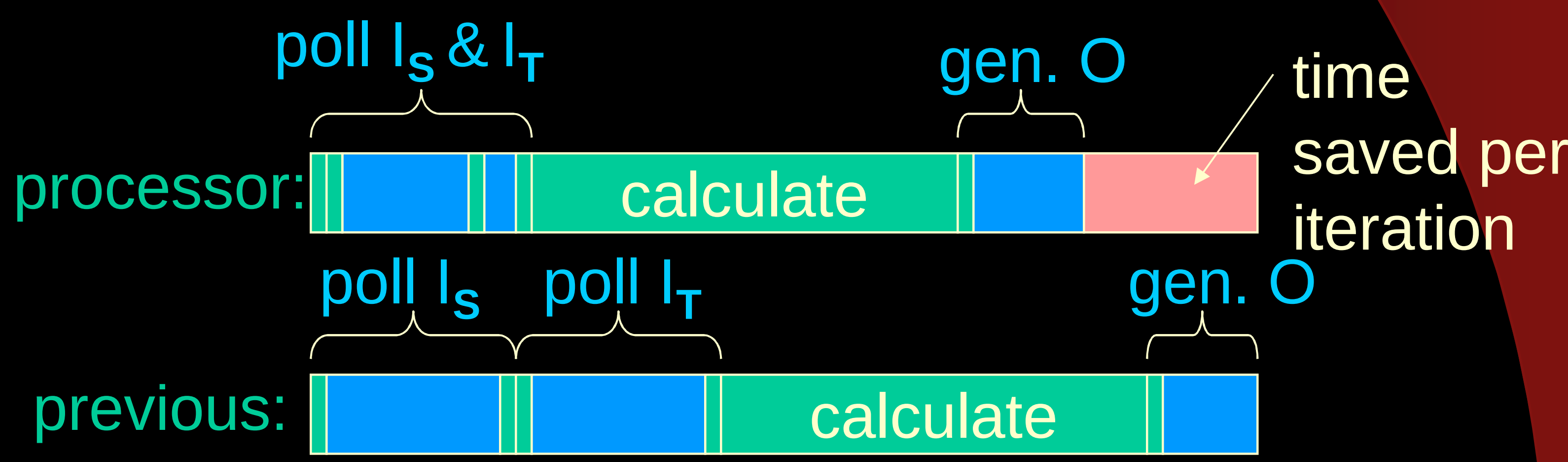
start O_H generating heater control signal

}



Analysis of Solution 1

- design approach:
  – no design, just GO as fast as possible
- could it go faster? (Solution 1a)
  – utilize <u>concurrency</u> of active input devices!



Solution 1: Faster Still?

- faster hardware? but … is faster necessary?
- engineering?
  - reduce cost and still meet requirements?
    - slower hardware is often less expensive
  - is behaviour predictable? analysis?
  - extension? processor available for more work?
  - are there redundant loop iterations? power?

Requirements Analysis

- **INPUT:** timing and magnitude of reference input changes? ← requirements!
- **OUTPUT:** how fast must output be adjusted to maintain acceptable plant state?
  - what is "acceptable"? ← requirements!
  - variation from "ideal"?
  - oscillation?

  } tolerances for engineering

Periodic Iteration?

- could shift design approach to perform loop iterations at regular periodic intervals
- need h/w timer to gauge start of period
- period too large → slow

  → failure to meet system requirements

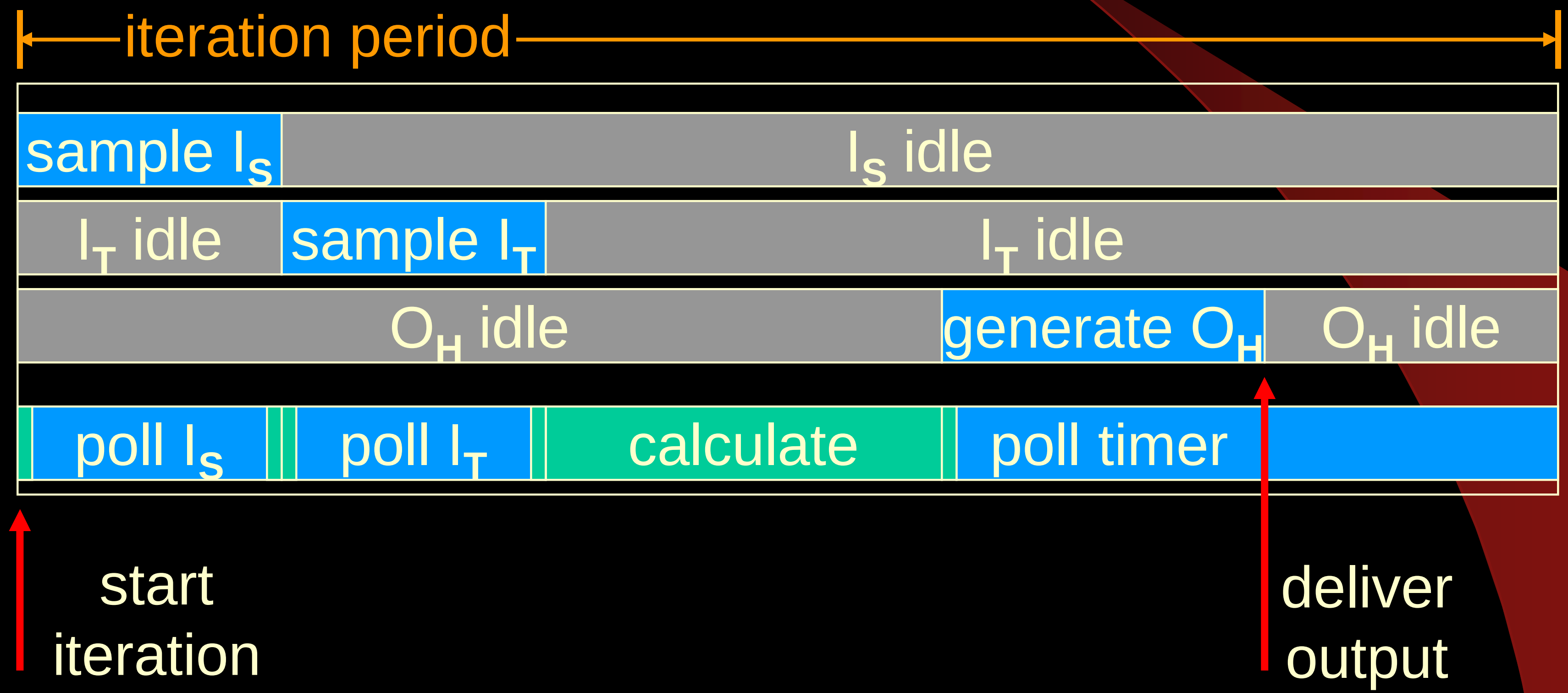
  – unacceptable from user's perspective
- period too small → fast

  → may have under-engineered product

  – not optimal from engineering perspective

Solution 2:

Sequential: Polled + Periodic



Solution 2: Timing

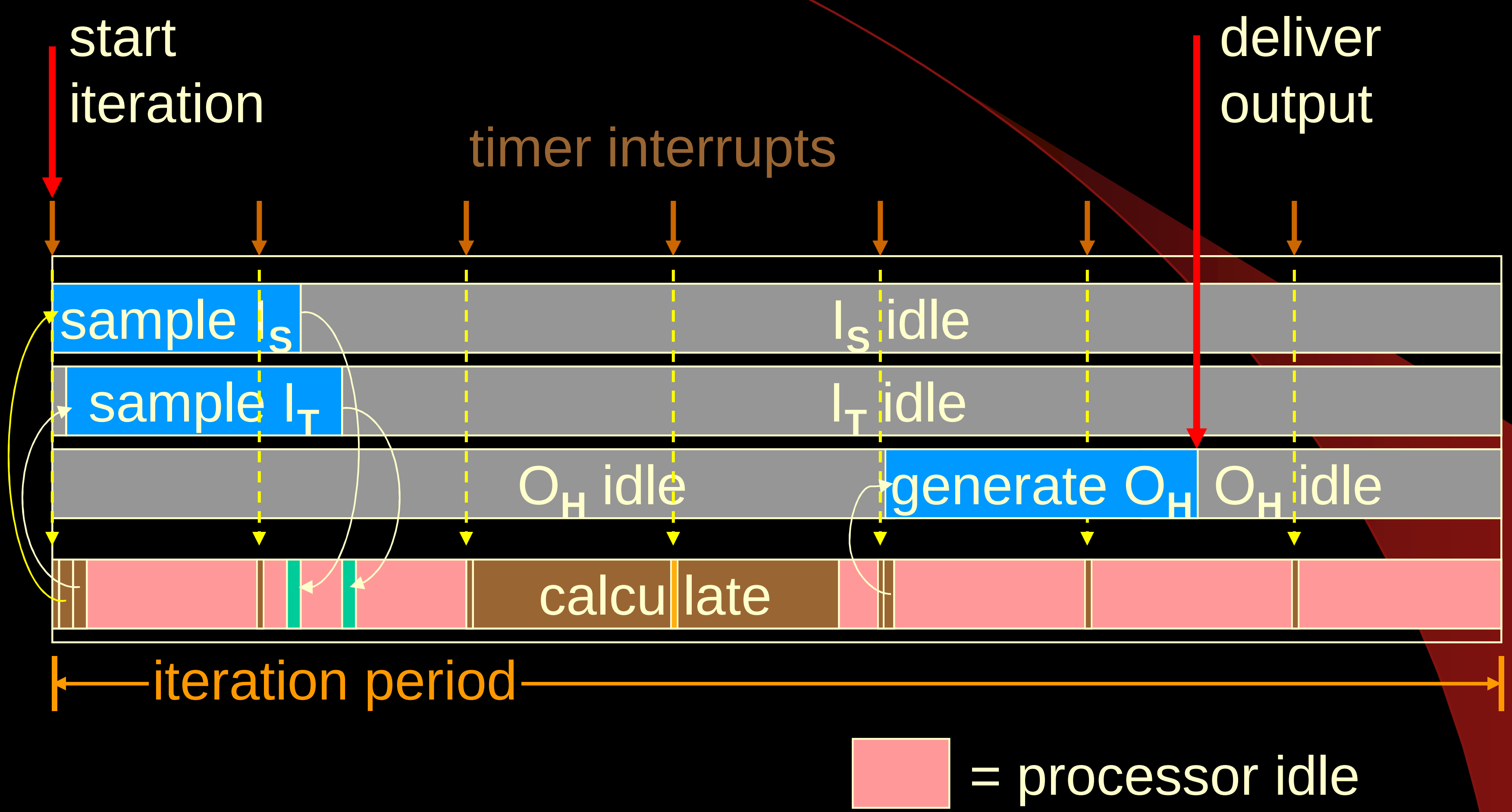
- processor has no idle time → **busy waiting** (poll)
- what factors influence the controller's timing behaviour? Are they predictable?
  - complexity of calculation
  - behaviour of I/O hardware
    - sampling inputs and generating outputs

Solution 3

Event-Driven: Interrupts

- periodic timer interrupt
- iteration period = integer multiple of timer period
- assume A/D input device generates interrupt when data ready to be read
- use **interrupts** to schedule activities
- use **ISRs** to execute activities on processor

Solution 3


start
iteration
deliver
output
timer interrupts
sample I_S
I_S idle
sample I_T
I_T idle
O_H idle
generate O_H
O_H idle
calcu late
iteration period
= processor idle

Solution 3: Processing

- all work done in ISRs → no polling!
- input ISRs: read values when ready
- timer ISR: regular tick plus
  - start input sampling
  - calculate output
  - start output generation
  - may require ability for timer interrupt to interrupt timer ISR!
    - tick in calculate!

OK for Toy Examples…but …

- multivariate, multirate systems
  - multiple degrees of freedom
  - different rates of control-law calculation
- more complex control-law computations
  - smooth the output trajectory
  - include estimation based on input history (state variables) and heuristics

What About Control Hierarchy?

- higher-level objectives
  - e.g. is temperature control part of a bigger manufacturing process?
- communication among hierarchy levels
- Liu text has more detailed examples!

Engineering vs. Art

- art: creation of a system using methods that are unique to artist and artist’s abilities
- engineering: specification, design and development of realistic systems using quantitative, systematic and repeatable methods known to “many”

Reference Model for RT Systems

- towards engineering RT systems
- terminology & taxonomy
  - application characteristics
  - scheduling, resource management
- generalize where possible
  - simplify discussion
  - assume general, unless specific reference

Jobs & Tasks

- **job** : a unit of work that is carried out by the system (J_i)
- **task** : a set of related jobs that provide some system function ($\tau_i = \{ J_{i,1}, J_{i,2}, \dots, J_{i,N} \}$)
- task → a generalization → a class of jobs
  - tasks are specified at design-time
- job $J_{i,k}$ → k^{th} instance of task i
  - jobs occur at run-time

Jobs & Task Example

Task i

process a packet

Instances

Job i,1 : process 1st packet

Job i,2 : process 2nd packet

…

Job i,k : process kth packet

…

design-time abstraction

run-time realization

Processors & Resources

- the available components in the system
  - design decisions!
- **processor** : an active h/w component involved in the execution of a job (P_i)
- **resource** : a passive (h/w or s/w) component required by a job

sometimes Liu text uses "resource" to encompass both processors and resources

Release Time & Deadline

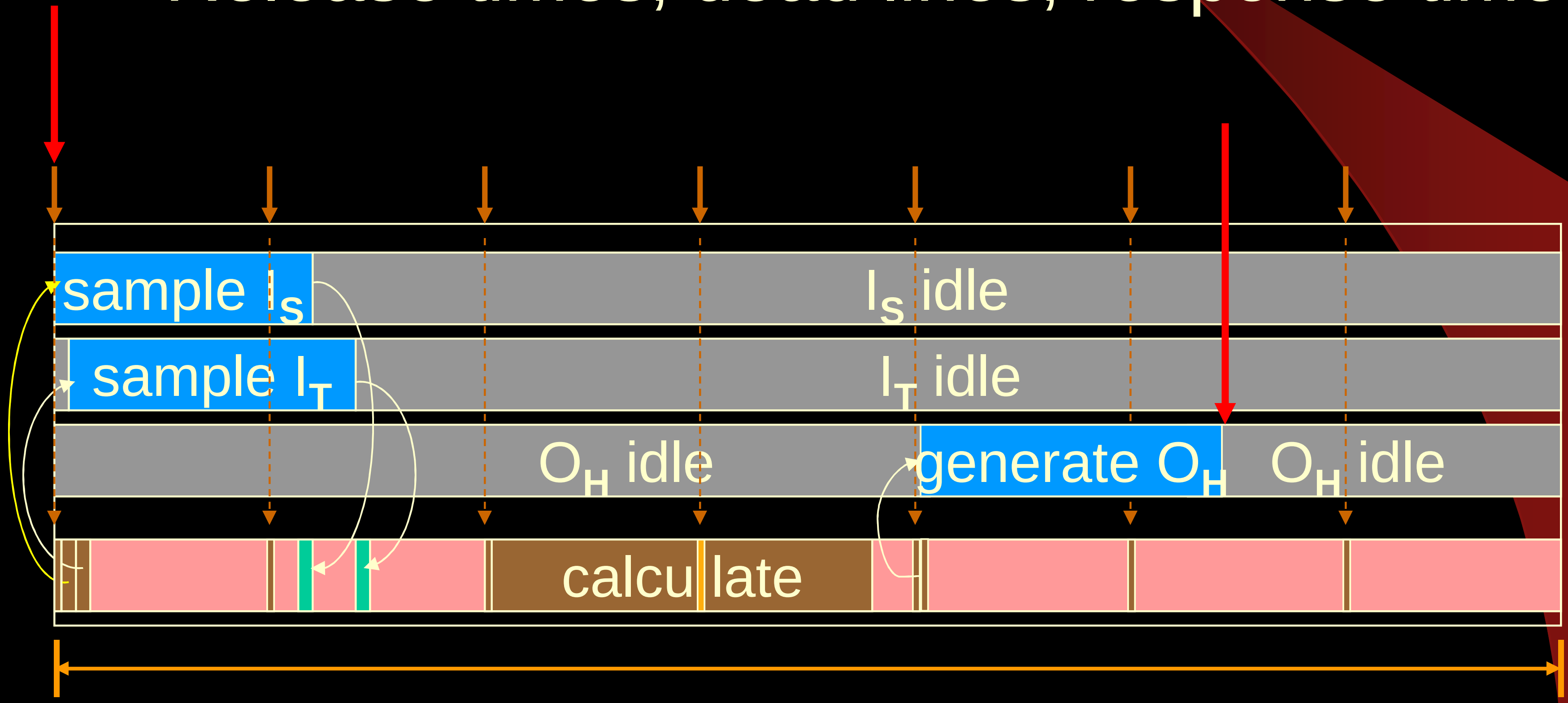
- **release time** (or **arrival time**) of a job: time at which the job becomes available for execution (r_i)
- **deadline** of a job: time at which the job must be completed
- **response time** of a job: length of time between the release time of the job and the time instant when it completes

Deadlines

- **relative deadline** of a job: maximum allowable response time of a job (D_i)
- **absolute deadline** of a job: time at which a job must be completed ($d_i = r_i + D_i$)
- **timing constraint**: a constraint imposed on the timing behaviour of a job
  - most often → the deadline of the job
  - others too e.g. jitter

Recall Temperature Control Example (Solution 3, slide 17)

- Tasks, jobs, processors, resources?
- Release times, dead lines, response times?



Hard RT System (Liu)

- recall previous definition → failure oriented
- a system is a hard real-time system if the requirements include the validation that the system always meets certain (hard) timing constraints
- validation: demonstration by a provably correct procedure, or by exhaustive simulation and testing
- guarantee vs. best effort

Specifying Hard Timing Constraints

- deterministic ← common (hard!)
  - specify constraints that must always be met
- probabilistic ← not as common (softer)
  - specify constraint and probability of meeting constraint
  - allows some (few) failures over many instances

Job & Task Parameters

- **temporal**: timing constraints and behaviour
- **interconnection**: dependencies among jobs (or among tasks)
- **resource**: active (processor) and passive (resource) components required

Temporal Parameters of Jobs

- includes r_i , d_i and D_i
- **feasible interval**: $(r_i , d_i]$
  - does not include r_i , includes d_i
  - includes execution time
- various forms of jitter → variations in timing behaviours of instances of jobs

Job Execution Time

- **execution time** : processing time required to complete work associated with job (e_i)
  - assumes that all required processors and resources are available
  - depends on complexity of job and speed of processors
- **execution jitter**: range of possible execution times [e_i^-, e_i^+]
  - best case and worst case execution times

Release Time Revisited

- **fixed** : know exact release time
- **jittered** : range of possible release times

  $[\, r_i^- , r_i^+ \,]$
- **sporadic / aperiodic** : released at random intervals e.g. key pressed on a keyboard
  - **sporadic** : specified minimum inter-arrival time
  - **aperiodic** : no spec'ed minimum inter-arrival time

Periodic Task Model

- deterministic workload model
  - applied at design-time
- lots of research
  - Liu & Layland, 1973
- basis for Rate Monotonic (RM) analysis
  - DoD requirement for hard RT systems

Periodic Task Model (2)

- **period** : time between successive releases of jobs in a task (p_i)
  - – typically have jitter → use minimum
    - pessimistic ? deterministic !
- **execution time** : maximum execution time of a job in the task (e_i)
    - pessimistic ? deterministic !
- **phase** : release time of first job in task (ϕ_i)

Notes About Model

- assumptions:
  - number of tasks, periods, execution times, phases are known
  - required components are always available
- pessimistic → always assumes worst cases
  - **NOTE:** accuracy (and applicability) of model decreases with increasing jitter

Hyperperiod

- **hyperperiod** : least common multiple of all task periods (H)
  - number of jobs for task i $= \frac{H}{p_i}$
- if n tasks, number N of jobs in hyperperiod:

$$N = \sum_{i=1}^{n} \frac{H}{p_i}$$

Processor Utilization

- **processor utilization by a task** : fraction of time the task keeps the processor busy (u_i)

$$u_i = \frac{e_i}{p_i}$$

- total utilization of processor by tasks (U)

$$U = \sum_{i=1}^{n} u_i = \sum_{i=1}^{n} \frac{e_i}{p_i}$$

How is Utilization Useful?

- $U \leq 1.0$ for each processor is a necessary, but not sufficient, condition for meeting deadlines
- must consider other related factors
  - deadlines
  - priority
  - sporadic tasks

Deadlines

- in general D_i not constrained relative to p_i
  - can be shorter, equal, or longer than p_i
- if $D_i < e_i$ then impossible to meet deadline
- throughput assumption: system always keeps up with work demanded
  - periodic task model: $D_i = p_i$

Back to: Job & Task Parameters

- **temporal**: timing constraints and behaviour

  ✓ → **periodic task model**

- **interconnection**: dependencies among jobs (or among tasks)
- **resource**: active (processor) and passive (resource) components required

Interconnection Parameters

- **precedence constraint** : jobs (tasks) must be performed in specified order
  - independent : order not constrained
- **precedence relation** : partial order that identifies precedence constraints
  - denote "**<**" (Lamport: "happens before")
  - $J_i < J_k$ indicates that J_i must complete before J_k can begin i.e. J_i happens before J_k
    - J_i is a predecessor of J_k

More on Precedence

- J_i is an immediate predecessor of J_k if
  - $J_i < J_k$ AND
  - no other job J_j such that $J_i < J_j < J_k$
- J_i is independent of J_k if neither

  $J_i < J_k$ nor $J_k < J_i$
- chain : a set of jobs in which no two jobs are independent
  - for all pairs, either $J_i < J_k$ or $J_k < J_i$

Job Precedence Graph

- embody precedence relation **<** over set of jobs ***J*** in a directed graph : **G = (*J*, <)**
- **vertices** : each job in ***J*** is a vertex
- **edges** : edge from J_i to J_k iff J_i is an immediate predecessor of J_k
- lattice (not necessarily a tree!)
- job may have multiple immediate predecessors
- may have more than one job with no predecessors

Resource Parameters

(Resource = Processors + Resources)

- all jobs require one or more processors
- resource parameters of a job:
  - type of processor(s) & number(s)
  - other resources required
  - time interval over which each is needed
- parameter of resource: preemptivity

Sharing Resources

- All jobs require resources
- Can jobs share resources?
  - Yes! Jobs often share a processor and memory.
  - Sharing I/O is less common … single “driver” task

Sharing complicates things!

Sharing requires management! → Scheduling!

Can Sharing Involve Preemption? (or run to completion)

- priority concern!
  - can a job be preempted by a higher-priority job ?
    - yes → job is preemptable
    - no → job is nonpreemptable

    Which might lead to more complicated scheduling?
- jobs often share a processor with preemption
- preempting shared memory access? a good idea?

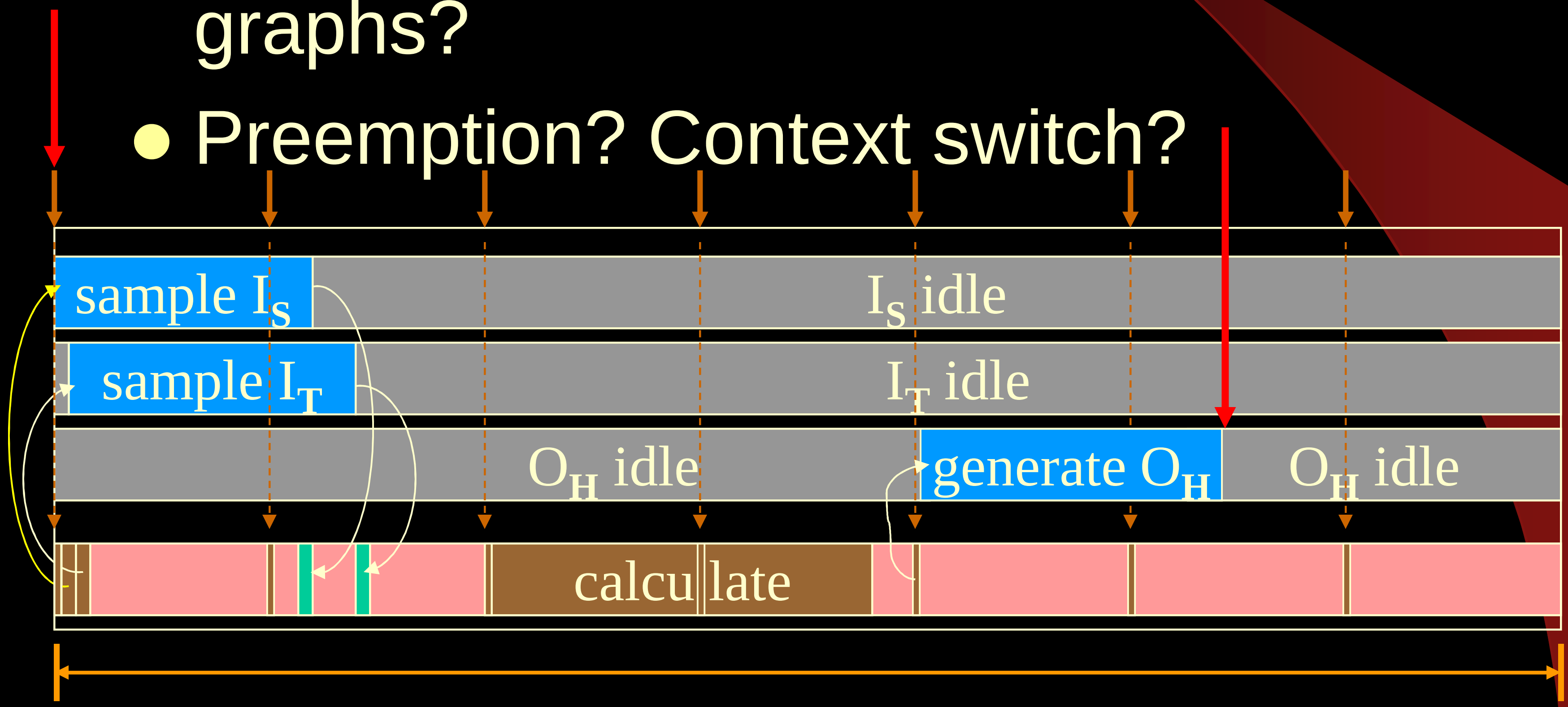
Implementing Preemption

- **context switch**:
  1. pause executing job
  2. save job/resource state at time of pausing
  3. install another job/resource state
- context switch back to preempted job (i.e. resume the job) at a future point in time

We'll see this in more detail later!

Recall Example (slide 17)

- Periods, execution times, jitter (?)
- Processor utilization, precedence graphs?
- Preemption? Context switch?



Scheduling Theory

- **ideal goal** : all jobs are always allocated required resources to complete execution within their feasible regions (r, d]
- **scheduling algorithm** : decides the order in which jobs are allocated resources
- **scheduler** : a module that implements a scheduling algorithm
- **scheduling decision point**: point in time when scheduler decides which job to execute next

Schedule

- **schedule** : assignment of all jobs (over time) to available resources
- **feasible schedule :** every job starts at or after its release time and completes by its deadline
  - Could be more than one feasible schedule!
- **optimal scheduling algorithm** : always produces a feasible schedule if at least one feasible schedule exists

Common Approaches For Real-Time Scheduling (Liu Ch. 4)

- **Clock-Driven (Time-Driven) :** scheduling decision points are specified *a priori* (static)
  - E.G. the temperature control example. **More Later!**
- **Weighted Round-Robin :** weighted jobs join a FIFO queue – weight determines amount of processor time allocated to the job ☹
- **Priority-Driven (Event-Driven)** : scheduling decisions are made as events occur (dynamic)
  - schedule ready job with highest priority

Priority-Driven Scheduling

- A major topic! But first …
- lets look at an **event-driven process model** in more detail