University of Glasgow | School of Computing Science

# Principles of Real-time Systems

## Advanced Operating Systems
## Lecture 1

# Lecture Outline

- Introduction and course administration
  - Aims, rationale, intended learning outcomes
  - Timetable
  - Assessment and examination
- Principles of real-time systems

# Introduction and Course Administration

# Resources and Contact Details

- Lecture slides and other materials are on Moodle
  - Also http://csperkins.org/teaching/adv-os/
  - Printed lecture handouts will not be provided – learning is enhanced by taking your own notes during lectures and tutorials

- Course coordinator:
  - Dr Colin Perkins, Room 405, Sir Alwyn Williams Building
  - Email: colin.perkins@glasgow.ac.uk
  - No assigned office hours – email to make appointment if needed

# Rationale

- Radical changes to computing landscape;
  - Desktop PC becoming irrelevant
  - Heterogeneous, multicore, mobile, and real-time systems – smart phones, tablets – now ubiquitous
- Not reflected by corresponding change in operating system design practice
- This course will...
  - review research on systems programming techniques and operating systems design;
  - discuss the limitations of deployed systems; and
  - show how the operating system infrastructure might evolve to address the challenges of supporting modern computing systems.

# Aims and Objectives

- To explore programming language and operating system facilities essential to implement real-time, reactive, and embedded systems
- To discuss limitations of widely-used operating systems, introduce new design approaches to address challenges of security, robustness, and concurrency
- To give an understanding of practical engineering issues in real-time and concurrent systems; and suggest appropriate implementation techniques

# Intended Learning Outcomes (1)

- At the end of this course, you should be able to:
  - clearly differentiate the issues that arise in designing real-time systems; analyse a variety of real-time scheduling techniques, prove correctness of the resulting schedule; implement basic scheduling algorithms;
  - apply real-time scheduling theory to the design and implementation of a real-world system using the POSIX real-time extensions; demonstrate how to manage resource access in such a system;
  - describe how embedded systems are constructed, and discuss the limitations and advantages of C as a systems programming language; understand how managed code and advanced type systems might be used in the design and implementation of future operating systems;
  - discuss the advantages and disadvantages of integrating garbage collection with the operating system/runtime; understand the operation of popular garbage collection algorithms; know when it might be appropriate to apply garbage collection and managed runtimes to real-time systems;
    …

# Intended Learning Outcomes (2)

…

- understand the impact of heterogeneous multicore systems on operating systems; compare and evaluate different programming models for concurrent systems, their implementation, and their impact on operating systems;
- construct simple concurrent programs using transactional memory and message passing to understand trade-offs and implementation decisions.

# Course Outline

- Real-time operating systems
  - Real-time scheduling
  - Resource allocation
  - Programming model
- Garbage collection
- Implications of multicore systems
  - Message passing
  - Transactions
  - General purpose GPU programming models

# Timetable (1)

| Week | Lecture | Subject |
|---|---|---|
| 1 | Lecture 1 | Principles of Real-time Systems |
| | Lecture 2 | Real-time Scheduling of Periodic Tasks (1) |
| | Lecture 3 | Real-time Scheduling of Periodic Tasks (2) |
| 2 | Tutorial 1 | Real-time Scheduling of Periodic Tasks |
| | Lecture 4 | Real-time Scheduling of Aperiodic and Sporadic Tasks (1) |
| | Lecture 5 | Real-time Scheduling of Aperiodic and Sporadic Tasks (2) |
| 3 | Tutorial 2 | Real-time Scheduling of Aperiodic and Sporadic Tasks |
| | Lecture 6 | Resource Management |
| | Lecture 7 | Real-time & Embedded Systems Programming |
| 4 | Tutorial 3 | Resource Management/Systems Programming |
| | Lecture 8 | Garbage Collection (1) |
| | Lecture 9 | Garbage Collection (2) |
| 5 | Tutorial 4 | Garbage Collection |
| | Lecture 10 | Implications of Multicore Systems |
| | Lecture 11 | Message Passing (1) |

# Timetable (2)

| Week | Lecture | Subject |
|---|---|---|
| 6 | Lecture 12 | Message Passing (2) |
| | Tutorial 5 | Message Passing |
| | | No lectures – programming assignment |
| 7 | | |
| | | |
| | | |
| 8 | Lecture 13 | Transactions |
| | Tutorial 6 | Transactions |
| | Lecture 14 | General Purpose GPU Programming (1) |
| 9 | Lecture 15 | General Purpose GPU Programming (2) |
| | Tutorial 7 | General Purpose GPU Programming |
| | Lecture 16 | Wrap-up |
| 10 | | No lectures |
| | | |
| | | |

# Assessment

- Level M course; 10 credits
- Coursework (20%)

| Exercise | Weight | Topic | Set | Due |
|---|---|---|---|---|
| 1 | 4% | Scheduling periodic tasks | Tutorial 1 | Tutorial 2 |
| 2 | 4% | Scheduling aperiodic/sporadic tasks | Tutorial 2 | Tutorial 3 |
| 3 | 12% | Programming message passing systems | Tutorial 5 | Tutorial 6 |

- Examination (80%)
  - Two hours duration; sample and past papers are available on Moodle
  - All material in the lectures, tutorials, and cited papers is examinable
  - Aim is to test your understanding of the material, not to test your memory of all the details; explain why – don't just recite what

# Pre- and co-requisites

- Required pre-requisites:
  - Computer Systems 2
  - Operating Systems 3
  - Advanced Programming 3
  - Functional Programming 4
- Recommended co-requisites:
  - Computer Architecture 4

# Required Reading

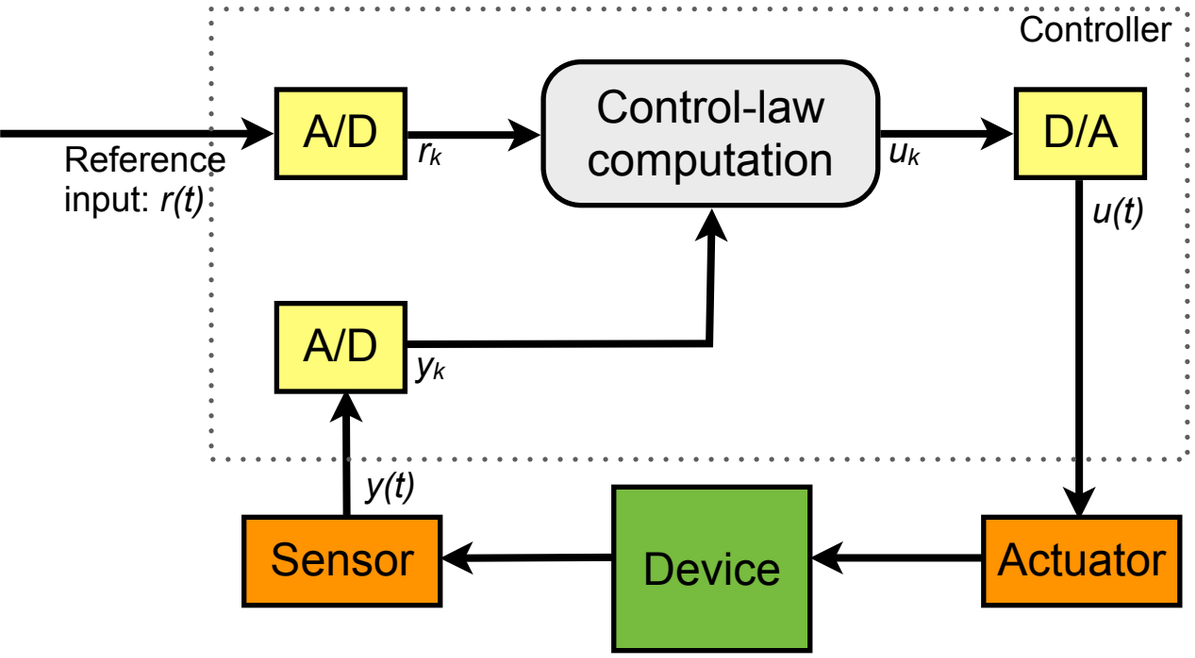
- No single set text book
- Research papers will be cited
  - DOIs will be provided; resolve via http://dx.doi.org/ – some papers behind paywalls, but accessible for free from on campus
  - You are expected to read and understand papers; it will be beneficial to follow-up on some of the references and do further background reading
    - Critical reading of a research paper is difficult and requires practice; read in a structured manner, not end-to-end, thinking about the material as you go
    - Advice on paper reading: http://www.eecs.harvard.edu/~michaelm/postscripts/ReadPaper.pdf
    - S. Keshav, “How to Read a Paper”, ACM Computer Communication Review, 37(3), July 2007 DOI: 10.1145/1273445.1273458
- Tutorials allow for discussion of papers and lectured material

# Principles of Real-time Systems

# Introduction to Real-time Systems

- Real-time systems deliver services while meeting timing constraints
  - Not necessarily fast, but must meet some deadline
  - Many real-time systems embedded as part of a larger device or system: washing machine, photocopier, phone, car, aircraft, industrial plant, etc.
- Frequently require validation for correctness
  - Many embedded real-time systems are safety critical – if they don't work in a timely and correct basis, serious consequences result
  - Bugs in embedded real-time systems can be difficult or expensive to repair – e.g., can't easily update software in a car!

# Typical System Model




- Control a device using actuator, based on sampled sensor data
  - Control loop compares measured value and reference
  - Depends on correct control law computation, reference input, 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

- Simple control loop conceptually easy to implement
- Complexity comes from multiple control loops running at different rates, of if the system contains aperiodic components

# 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 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
- Desire proofs of correctness, ways of raising the level of abstraction programming such systems

# Reference Model for Real-time Systems

- A reference model and consistent terminology let us reason about real-time systems
  - Cannot prove correctness without well-defined system model
- Reference model needs to characterise:
  - Applications running on a system (*jobs* and *tasks*) and the *processors* supporting their execution
  - The *resources* those applications use
  - Scheduling algorithms to determine when applications execute and use resources, and the *timing constraints* they must meet

# Jobs, Tasks, Processors, and Resources

- A *job* is a unit of work scheduled and executed by the system
- A *task* $T = \{J_1, J_2, ..., J_n\}$ is a set of related jobs that together perform some operation

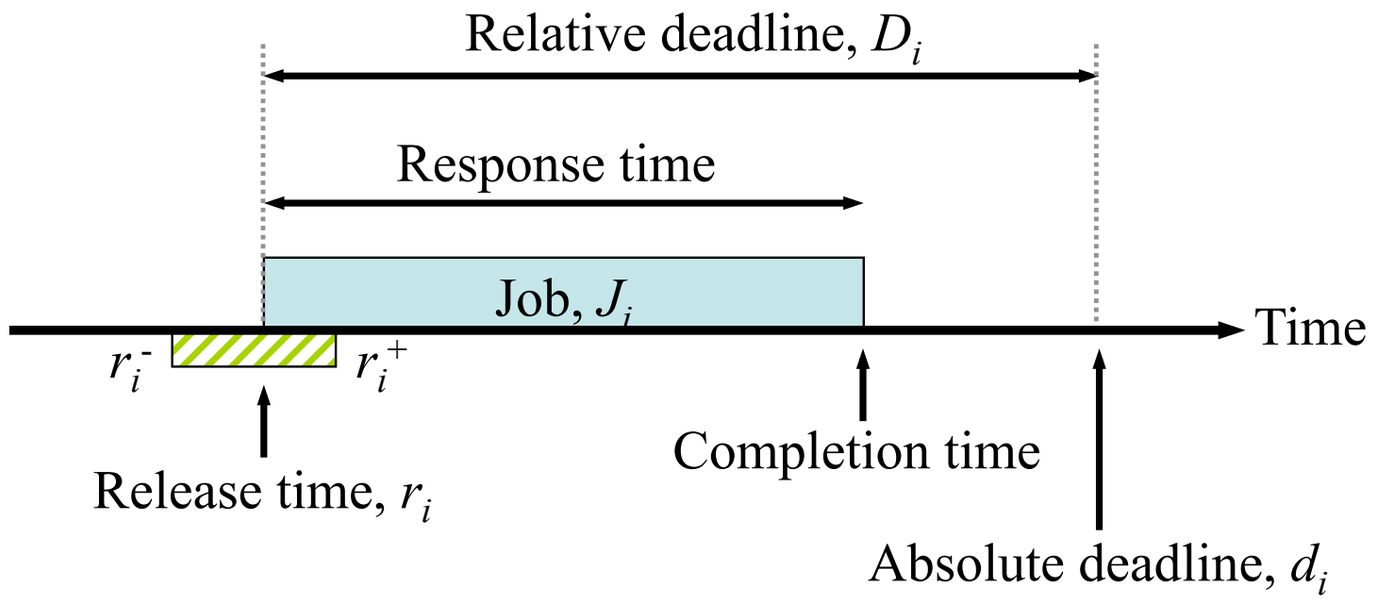
- Jobs execute on a *processor* and may depend on some *resources*
- A *scheduling algorithm* describes how jobs execute

- 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
  - A hardware device, for example
- Resources have different types or sizes, but have no speed attribute and are not consumed by use
- Jobs compete for resources, and can block if a resource is in use

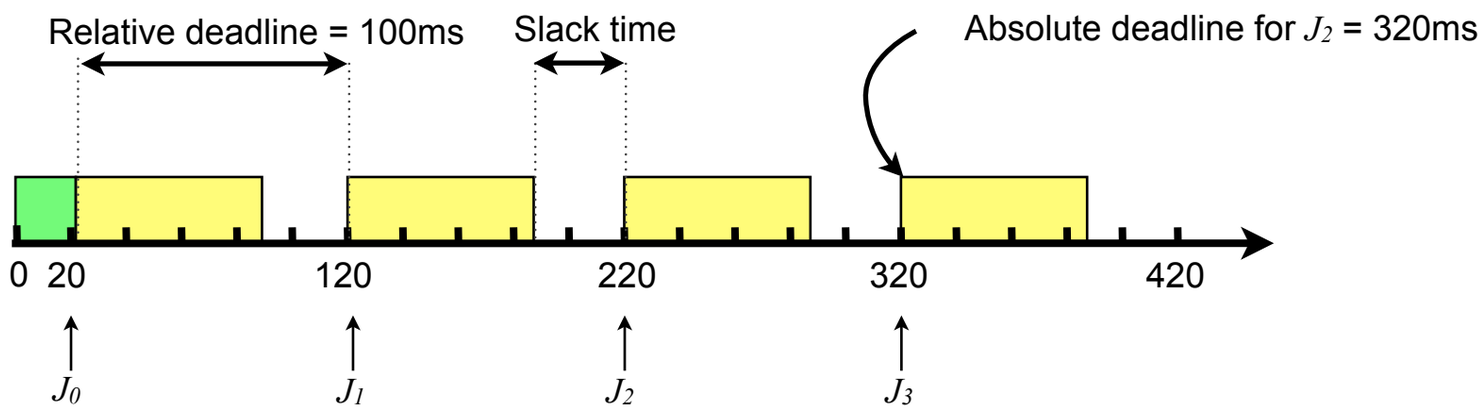
# Timing Constraints

- Job $J_i$ executes for time $e_i$ – time to finish $J_i$ given sole use of processor, and all required resources
  - Execution time depends on input data – use worst case for safety
- Jobs have timing constraints – relative or absolute 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



# Periodic Tasks

- If jobs occur on a regular cycle, the task is *periodic* and characterised by parameters $T_i = (\varphi_i, p_i, e_i, D_i)$
  - Phase, $\varphi_i$, of the task is the release time of the first job (if omitted, $\varphi_i = 0$)
  - Period, $p_i$, of the task is the time between release of consecutive jobs
  - Execution time, $e_i$, of the task is the maximum execution time of the jobs
  - Relative deadline, $D_i$, is the minimum relative deadline of the jobs (if omitted, $D_i = p_i$)
  - Utilisation of a task is $u_i = e_i / p_i$ and measures the fraction of time for which the task executes
  - The total utilisation of a system $U = \sum_i u_i$
- Common in real-world control systems

# Aperiodic and Sporadic Tasks

- If jobs have unpredictable release times, a task is termed *aperiodic*
- A *sporadic* task is an aperiodic task where the jobs have deadlines once released

- Greatly complicate reasoning about correctness
  - Helpful if bounds or probability distributions of release times and deadlines can be determined

# The Real-time Scheduling Problem

- Need to schedule jobs and manage resources
- In a *valid* schedule for a set of jobs:
  - Processors are assigned at most one job at once; jobs are assigned at most one processor at once
  - No job is scheduled before its release
  - Processor time assigned to each job equals its maximum execution time
  - All the precedence and resource usage constraints are satisfied
- A *feasible* schedule is valid, and jobs meet timing constraints – not all valid schedules are feasible
- An *optimal* scheduling algorithm will always find a feasible schedule if it exists

# Hard and 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

# Further Reading

- Next few lectures will focus on real-time scheduling

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