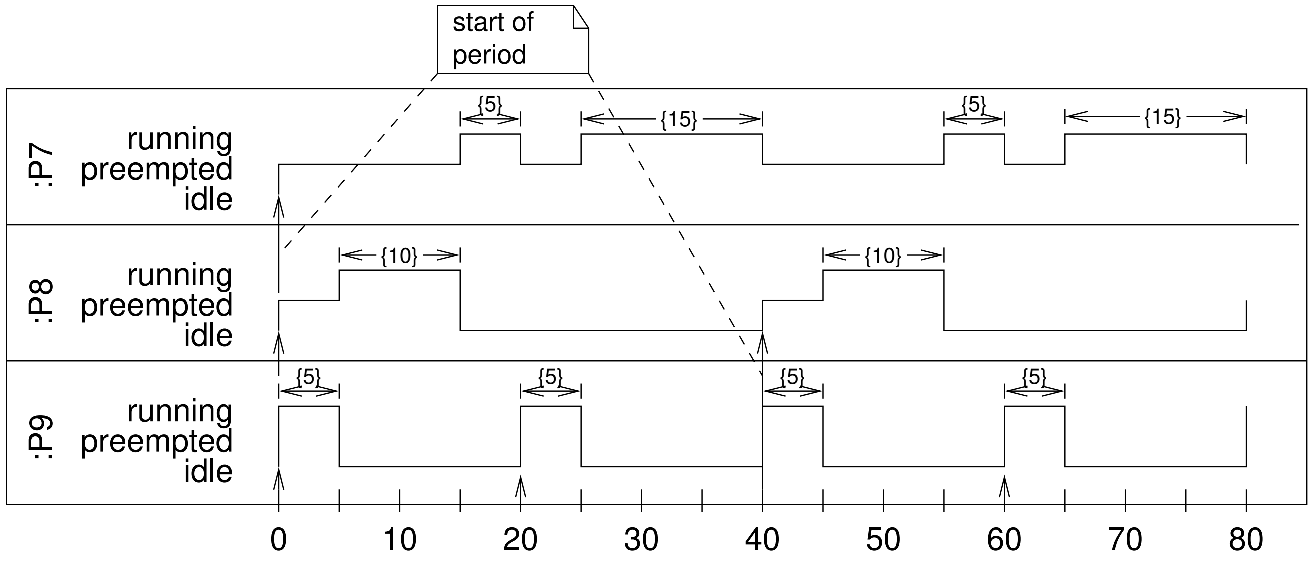
SYSC 3303 Real-Time Concurrent Systems

Real Time Scheduling



Correctness of Concurrent Programs

- In a non-real-time concurrent system, as long as the algorithms in each process are implemented correctly, and as long as the required mutual exclusion and condition synchronization are correctly programmed, the outputs of the program will be the same regardless of the order in which the processes are executed
  - in these slides, the terms process and thread are synonymous
- As long as the program's outputs are correct, the timing behaviour is not a major issue
- Real-time systems are different

What is a Real-Time System?

- "A real-time system ***must respond to unpredictable stimuli from its environment in a timely fashion*** while operating reliably and continuously in the presence of failures in its own (perhaps distributed) components and connections, and uncertainty about the state of its environment."
- A (hard) real-time system must guarantee that it meets all of its deadlines.
    - How do we do this?

Hard vs. Soft Real-Time

- *Hard* real-time system - responses must occur within specified deadlines
  - the system fails if its deadlines are missed, and this failure can have catastrophic results
- *Soft* real-time system — response times are important, but system functions correctly if a deadline is occasionally missed (e.g., responses occasionally delivered late or not at all)
- In comparison, a system is called *interactive* if it does not have specified deadlines but attempts to provide "adequate" response times

Process Scheduling in Real-Time Systems

- In other words, real-time systems have *temporal requirements* in addition to their other requirements
- Processes in a real-time system must be scheduled so that the system meets its temporal requirements
- A scheduling scheme provides
  – an algorithm for ordering the use of system resources (especially the CPU(s))
  – a means of predicting the worst-case behaviour of the system when the scheduling algorithm is applied

What We’ve seen So Far

- Earlier in the course, we discussed prioritized, preemptive scheduling, and mechanisms for controlling priority inversion (priority inheritance protocol, priority ceiling protocol), but our descriptions were qualitative in nature
- Now we'll go back to first principles, and provide a quantitative description of various scheduling schemes
- To simplify the analysis of the worst-case system behaviour, we'll initially use a very simple process model when presenting some standard scheduling schemes

A Simple Process Model

- The program consists of a fixed set of processes
- All processes are periodic, with known periods
- All processes are completely independent of each other
  - as such, at some point in time all processes will become ready-to-run at the same instant
- All system overheads (e.g., context switching times) are assumed to be 0
- All processes have a deadline equal to their period
- All processes have a fixed worst-case execution time

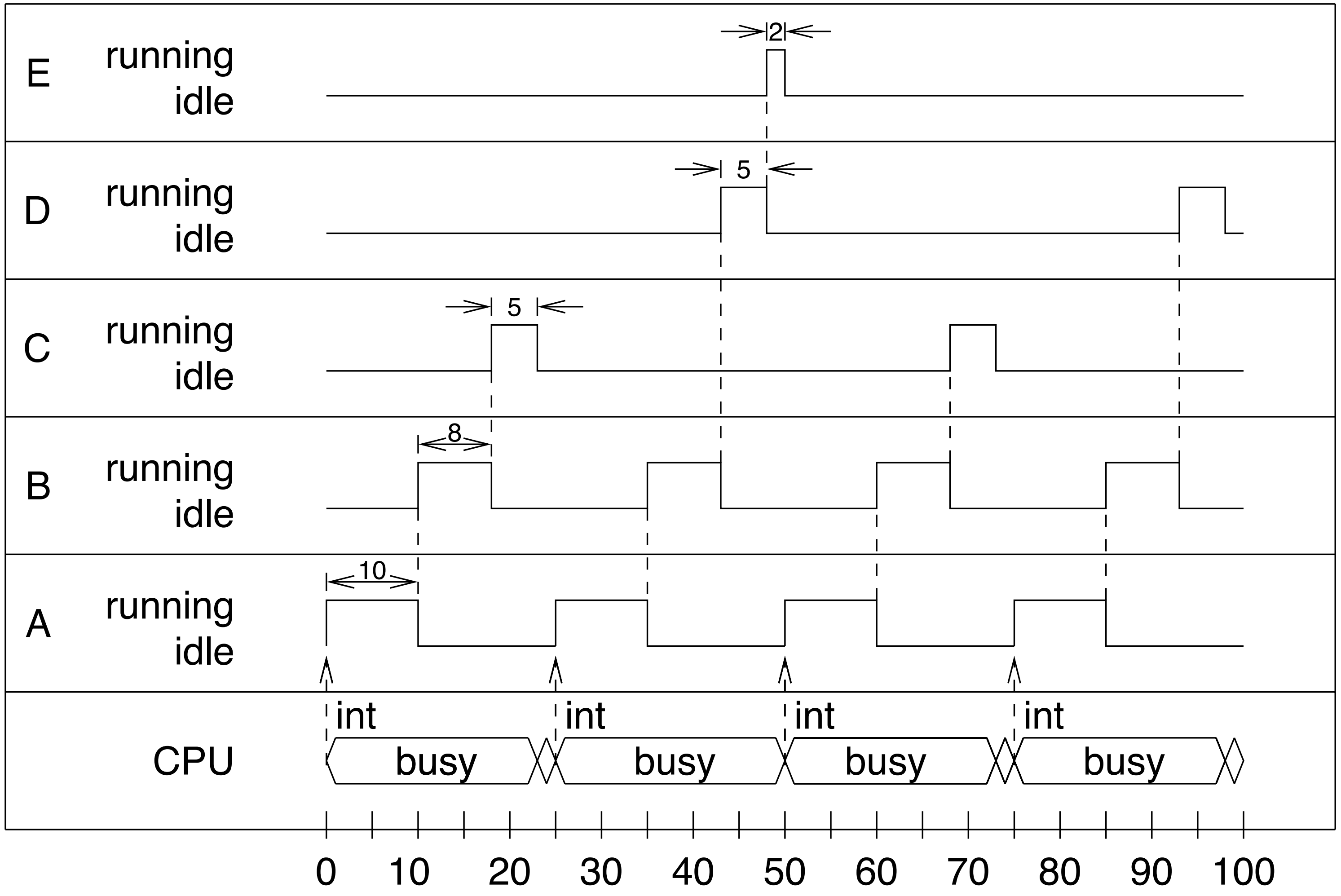
Cyclic Executive

- We plan a complete schedule such that the repeated execution of this schedule causes all processes to run at their correct rate
- Example:

Process	Period, T	Computation time, C
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2

- Here is one possible schedule:

Cyclic Executive



Cyclic Executive

```
for (;;) {
   wait_for_interrupt();
   A();
   B();                        minor cycle
   C();
   wait_for_interrupt();
   A();
   B();                        minor cycle
   D();
   E();
```

Cyclic Executive (II)

```
    wait_for_interrupt();
    A();
    B();                     } minor cycle
    C();
    wait_for_interrupt();
    A();
    B();                     } minor cycle
    D();
}
```

Cyclic Executive

- The complete loop body is known as a *major cycle*
- A major cycle is made up of *minor cycles*
- The occurrence of a timer interrupt triggers the execution of the next minor cycle
- Here, there are 4 minor cycles of 25 ms duration (the time between interrupts), making up a 100 ms major cycle
- We should verify that, with this schedule, the processes are executed at the correct rate

Cyclic Executive

- Process A, *T* = 25 ms
  - time between 2nd and 1st executions = 25 - 0 = 25
  - time between 3rd and 2nd executions = 50 - 25 = 25
  - time between 4th and 3rd executions = 75 - 50 = 25
  - time between 5th and 4th executions = 100 (start of 2nd major cycle, not shown) - 75 = 25
- Process B, *T* = 25 ms
  - time between 2nd and 1st executions = 35 -10 = 25
  - time between 3rd and 2nd executions = 60 - 35 = 25
  - time between 4th and 3rd executions = 85 - 60 = 25
  - time between 5th and 4th executions = 110 (in 2nd major cycle) - 85 = 25

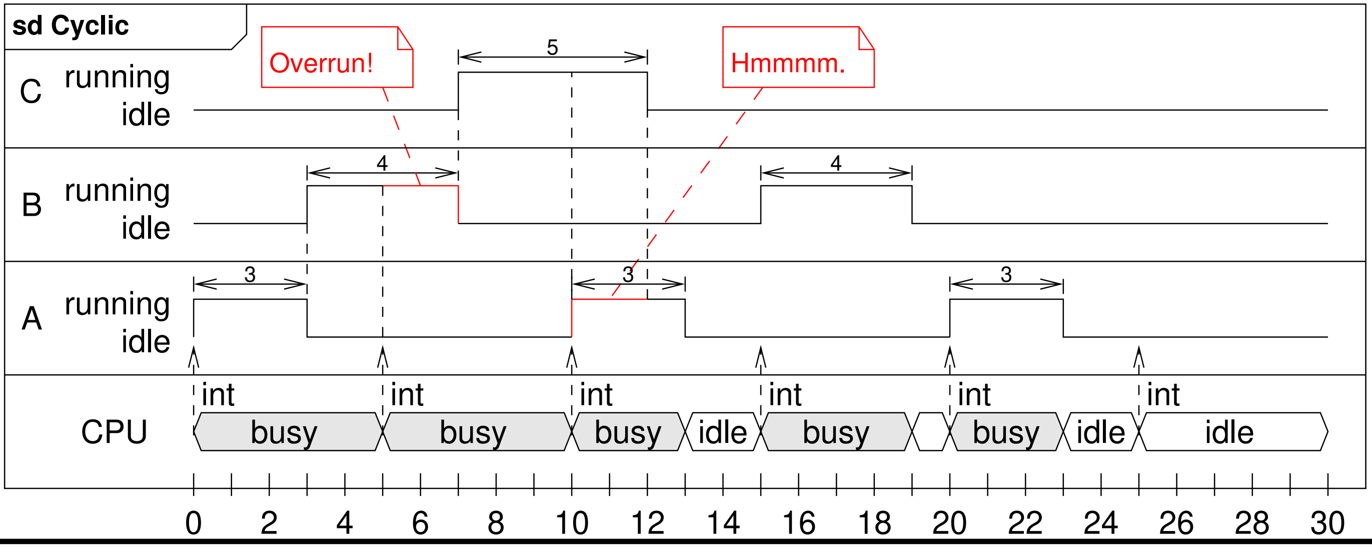
Cyclic Executive

- Process C, *T* = 50 ms
  - time between 2nd and 1st executions = 68 - 18 = 50
  - time between 3rd and 2nd executions = 118 (in 2nd major cycle, not shown) - 68 = 50
- Process D, *T* = 50 ms
  - time between 2nd and 1st executions = 93 - 43 = 50
  - time between 3rd and 2nd executions = 143 (in 2nd major cycle, not shown) - 93 = 50
- Process E, *T* = 100 ms
  - time between 2nd and 1st executions = 147 (in 2nd major cycle, not shown) - 47 = 100

Cyclic Executive

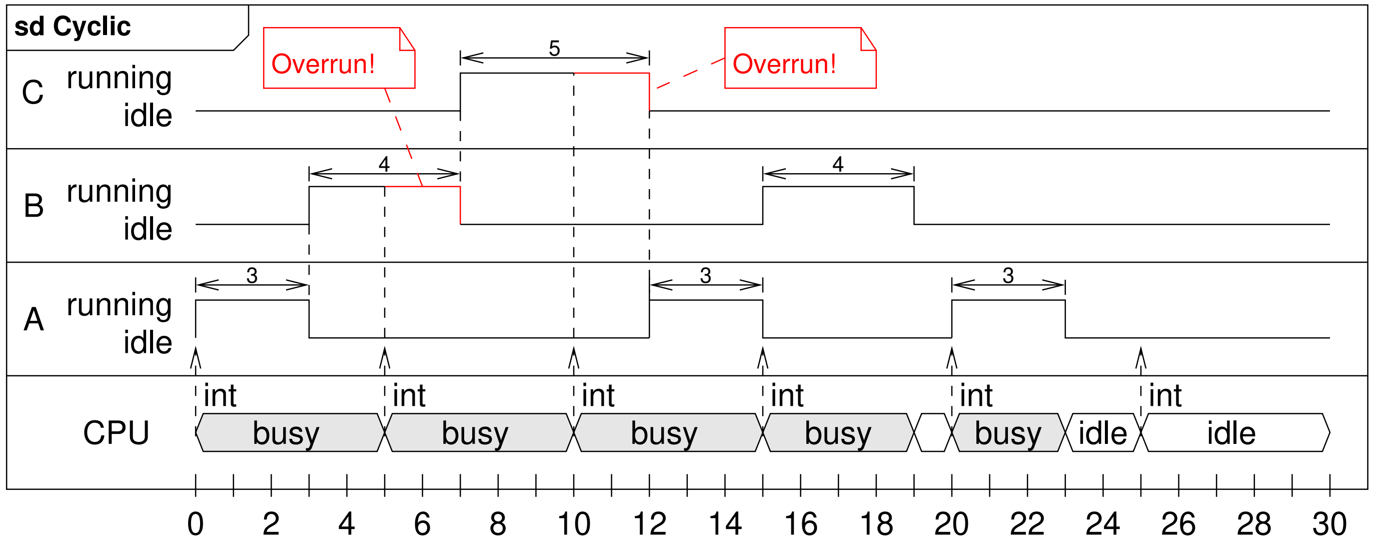
- What happens if the process set appears to be "unschedulable", such as the one shown below.

Process	Period, T	Computation time, C
A	10	3
B	15	4
C	20	5



Allow Interrupt Overruns

- Run A(), B() and C() in the interrupt handler.
- Allow overruns
    - The interrupt at t=5 is not serviced until the Interrupt Service Routine running at t=0 returns.
    - All deadlines met.
    - No interrupts missed.

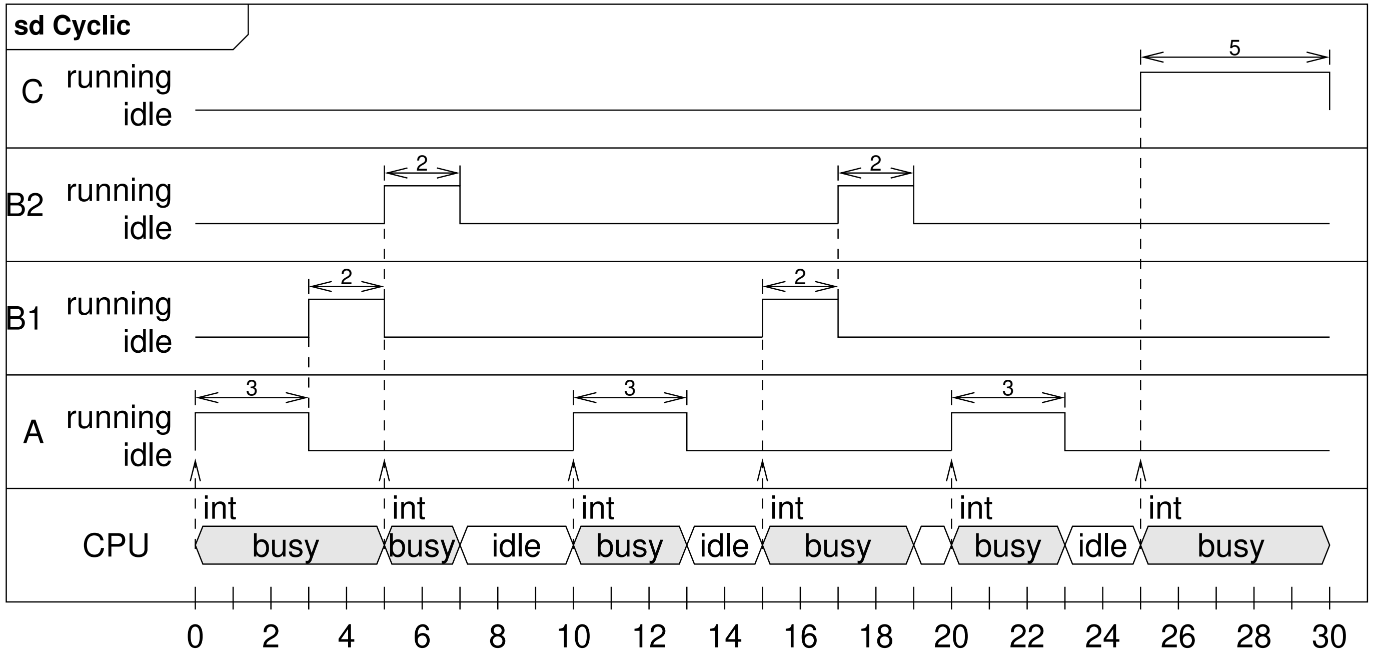


Cyclic Executive

```
ISR(){
  switch(count){
    case 0: A(); B(); break;
    case 5: C(); break;
    case 10: A(); break;
    case 15: B(); break;
    case 20: A(); break;
    case 25: break;
  }
  count = (count + 5) % 30;
}
```

Split Process

- Split one (or more) processes to make things fit.



```
void B(){
   for ( i = 0; i < 1000; ++i ) {
      /* do work */
   }
}
```

Splitting Processes

Note that I create two functions and split the work (represented by the stuff in colour).

```
void B1(){
  for ( i = 0; i < 500; ++i ) {
    /* do work */
  }
}


void B2(){
  for ( i = 0; i < 500; ++i ) {
    /* do work */
  }
}
```

- This may be somewhat challenging to do in real life.
- `B1()` and `B2()` may run in different minor cycles (e.g. at t=0 and t=15).

Cyclic Executive - Important Features

- Notice that each process can be implemented by a function/procedure, and that there is none of the context-switching overhead normally associated with concurrent processes — the scheduler executes each each “process” by performing a function call, and each minor cycle is just a sequence of function calls
- Functions share a common address space, and can pass data between themselves (no need to program mutex because concurrent access not possible)
- All “process” periods must be a multiple of the minor cycle time

Cyclic Executive - Drawbacks

- It is difficult to include aperiodic (sporadic) processes
- Extra work is required to incorporate processes with long periods
  - the major cycle time is normally the maximum period that can be accommodated
  - to accommodate processes with periods longer than the major cycle, we can add a procedure to the major cycle that will call a secondary procedure every *N* major cycles

Cyclic Executive - Drawbacks

- Each cyclic executive must be built "from scratch"
- A "process" with a long computation time (compared to the other processes) may need to be split into a fixed number of fixed sized procedures, which are called from different minor cycles
  - this may cut across the structure of the code from a software engineering perspective, so may be error-prone or difficult to maintain

Cyclic Executive

- If it is possible to construct a cyclic executive for a set of periodic processes, then no further schedulability test is required (the scheme is “proof by construction”)
- For systems with a large number of processes, building the executive is problematic
- An analogy can be made with the classic bin-packing problem: items of varying sizes (in one dimension) have to be placed in the minimum number of bins such that no bin is over-full
- The bin-packing problem is NP-hard, so is computationally infeasible for sizeable problems

Process-Based Scheduling

- A more flexible approach than a cyclic executive is to provide an environment (e.g., a real-time kernel or real-time o/s) that supports process creation and execution
- Processes can be in one of a number of states; e.g.,
  - runnable
  - suspended waiting for a timing event
  - suspended waiting for a non-timing event (appropriate for sporadic processes)

Process-Based Scheduling

- Each process has a priority attribute
  - in a real-time system, a process' priority is determined by its temporal requirements, not its importance
- Runnable processes are executed in the order determined by their priority
- A high-priority process may become runnable during the execution of a low-priority process
- In a preemptive, priority-based scheduling scheme, an immediate context switch to the higher-priority process occurs

Process-Based Scheduling

- In a **non-preemptive** scheduling scheme, the lower priority process is allowed to run until it relinquishes the processor, then the higher priority process executes
- Between these extremes, we can have **deferred preemption**
  - after the higher-priority process becomes runnable, the lower-priority process is allowed to continue executing for a bounded time
  - if it has not relinquished the processor by the end of that time, it is preempted

Rate Monotonic Priority Assignment

- Rate monotonic priority assignment is a priority assignment scheme for the simple process model described earlier
- Each process is assigned a unique priority based on its period
  - the shorter the period *T*, the higher the priority *P*
  - if $T_i < T_j$, $P_i > P_j$

Rate Monotonic Priority Assignment

- Example:

Process	Period, T	Priority, P
F	25	5
G	60	3
H	42	4
I	105	1
J	75	2

- Note that priority 1 is the lowest, priority 5 is the highest

Utilization-Based Schedulability Tests

- In 1973, Liu and Layland presented a simple test for schedulability when rate monotonic priority assignment is used
- This test considers only the utilization of the process set
  - utilization, *U*, of a process equals its computation time divided by its period: *C* / *T*
- In a system with *N* processes, all processes will meet their deadlines if the following condition holds:

$$\sum_{i=1}^{N} \left(\frac{C_i}{T_i} \right) \leq N\left(2^{\frac{1}{N}} - 1\right)$$

Utilization-Based Schedulability Tests

N	Utilization Bound (%)
1	100.0
2	82.8
3	78.0
4	75.7
5	74.3
10	71.8

- The bound asymptotically approaches 69.3%
- Any process set with a total utilization of less than 69.3% will always be schedulable by a preemptive priority-based scheduling scheme, with priorities assigned by the rate monotonic algorithm

Utilization-Based Schedulability Tests

- Example 1

Process	T	C	P	U
P1	50	12		
P2	40	10		
P3	30	10		

- Assign priorities (RMA) and calculate utilization:

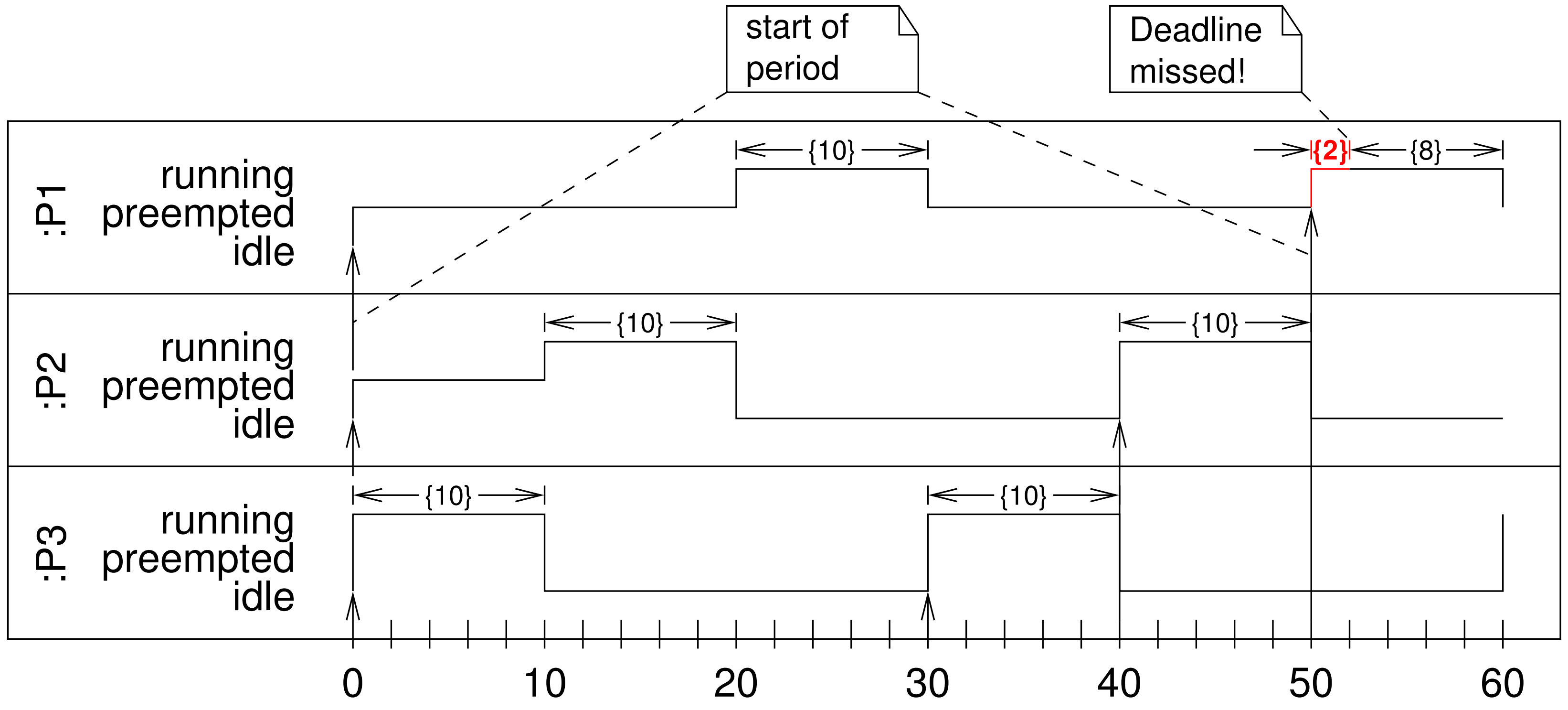
Process	T	C	P	U
P1	50	12	1	0.24
P2	40	10	2	0.25
P3	30	10	3	0.33

- Combined utilization is 0.82 (82%), which is above the bound for three processes (78%)

Utilization-Based Schedulability Tests

- Therefore, this process set fails the utilization test
- Does this mean that the processes are not schedulable by a preemptive, priority-based scheme so that they meet their deadlines?
- No — we need to do more analysis
- We can use a timeline to depict the behaviour of this process set (assume all processes runnable at t = 0)
- Notice, on the next slide, that P1 has executed for only 10 units before its deadline at t = 50
  - it needed 12 units of computation time, so it missed its first deadline

Utilization-Based Schedulability Tests



Verifying Schedulability with Timelines

- Timelines can be used on their own to test for schedulability
- How long a timeline must be drawn to verify that the process set will always meet its deadlines?
- Liu and Layland showed that for processes that are all runnable at $t = 0$, it can be shown that a timeline equal to the length of the longest period is sufficient
  - if all processes meet their first deadlines they will meet all future ones

Utilization-Based Schedulability Tests

- Example 2

Process	T	C	P	U
P4	80	32		
P5	40	5		
P6	16	4		

- Assign priorities (RMA) and calculate utilization:

Process	T	C	P	U
P4	80	32	1	0.400
P5	40	5	2	0.125
P6	16	4	3	0.250

- Combined utilization is 0.775 (77.5%), which is below the bound for three processes (78%)

Utilization-Based Schedulability Tests

- Therefore, this process set passes the utilization test, and the processes are guaranteed to meet all their deadlines
- Exercise for the student: draw the timeline for this process set

Utilization-Based Schedulability Tests

- Example 3

Process	T	C	P	U
P7	80	40		
P8	40	10		
P9	20	5		

- Assign priorities (RMA) and calculate utilization:

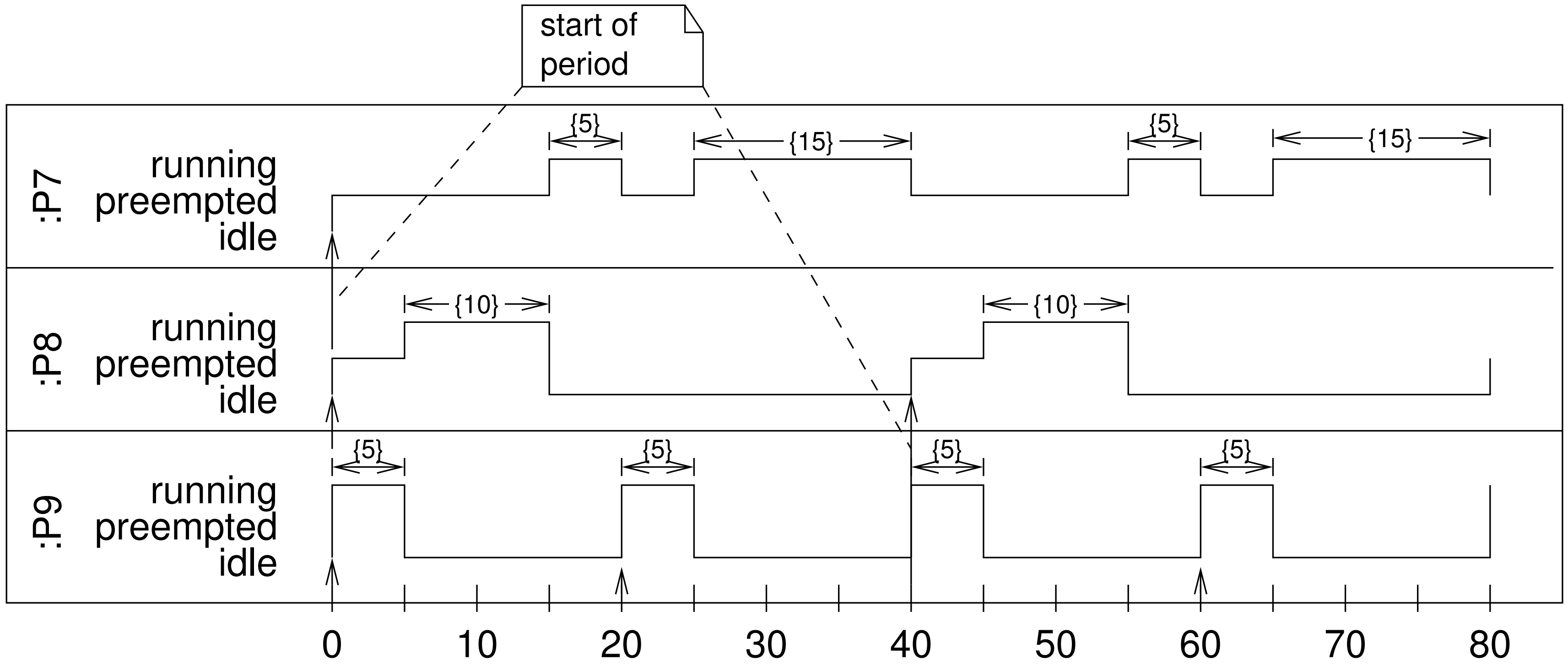
Process	T	C	P	U
P7	80	40	1	0.50
P8	40	10	2	0.25
P9	20	5	3	0.25

- Combined utilization is 1.00 (100.0%), which is above the bound for three processes (78%)

Utilization-Based Schedulability Tests

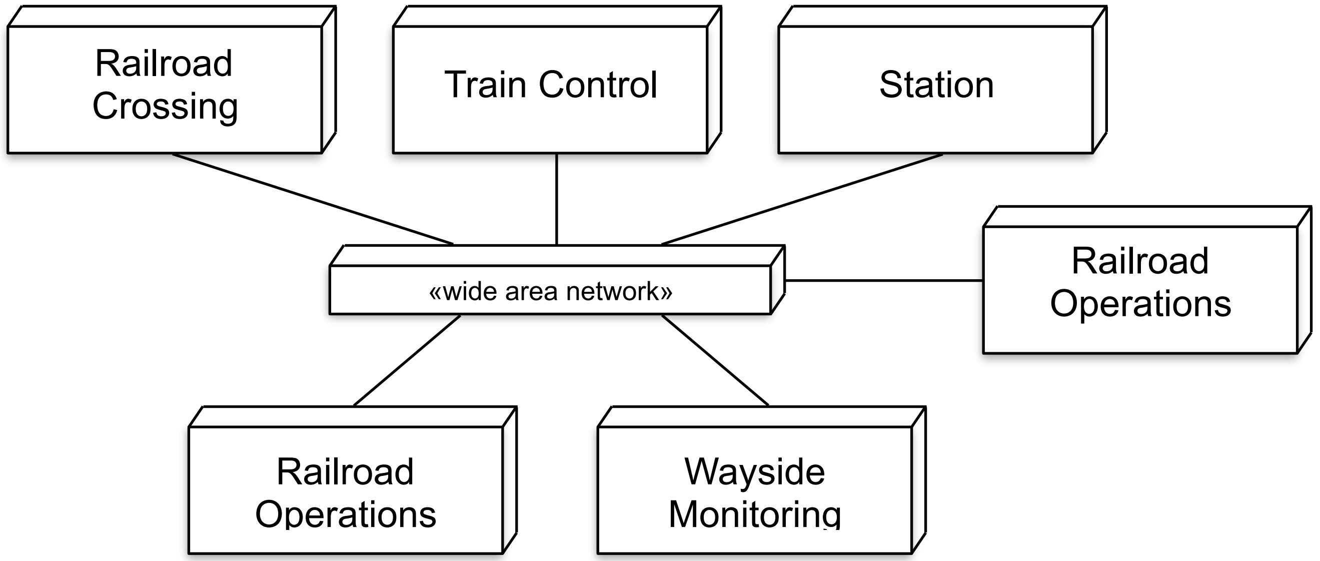
- Therefore, this process set fails the utilization test
- But, as the timeline on the following slide shows, all deadlines are met up to *t* = 80 (the longest period) so the processes are guaranteed to meet all their deadlines
- The utilization test is **sufficient** but not **necessary**
- If a process set passes the test, it will meet all its deadlines
- If a process set fails the test, it may or may not fail at run-time

Utilization-Based Schedulability Tests



SYSC 3303 Real-Time Concurrent Systems

Component-Based Software Architectures



- revised January 10th, 2019

You are here

1. System structural modelling.
2. Requirements modelling.
3. Analysis modelling:
    I. Static modelling.
    II. Dynamic state machine modelling.
    III. Object structuring.
    IV. Dynamic interaction modelling.
4. Design modelling.
5. Incremental software construction (your project!)
6. Incremental software testing (your project!)
7. System testing.

Component Based Software Architectures

- Components are initially designed based on the subsystem structuring criteria.
- Components encapsulate information and provide concurrency.
  - The goal is a flexible design which is easy to modify and deploy on a variety of architectures.
    - i.e., components can all run on one node on a small system but multiple nodes for a large system.
- Components can contain other components.
- Since components can be geographically distributed, all communication between components is through messages.

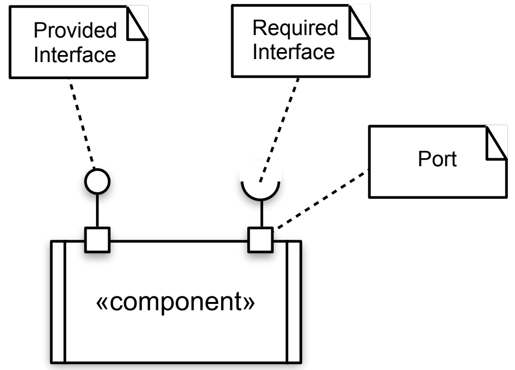
Designing Distributed Components

Three steps:

1. Design the software architecture.
   - Structure the software such that it can be grouped into components that can run on separate nodes.
2. Design the constituent components as single or composite components.
   - Create components and composite components as needed.
3. Deploy the architecture.
   - Deploy components to nodes and connect them to each other.

Some Nomenclature

- The interface specifies the externally visible operations.
  - **Provided**: what the component supplies to other components.
  - **Required**: what the component needs.
  - A component may have any number (>0) of interfaces.
- Components interact with each other using **ports**.
  - Can provide, require or both provide and require (*complex*)



- A **connector** joins the provided interface of one component to the required interface of another.
- A **complex component** aggregates components — no new functionality is incorporated.

Component Structuring Criteria

- The system must be designed with an understanding of the environment in which is going to operate.
  - Large and distributed?
  - Small and centralized?
- If there is a possibility that the system will be distributed, the it should be designed using components that can be distributed.
- Components will be assigned to nodes later when the system is configured for the target environment.
- Some guidelines follow…

Component Structuring Criteria (II)

Proximity to the Source of the Data

- Particularly important is data access rates are high.
- Software and sensors are within a common component are called a **smart device**.

Localized Autonomy

- Distributed components often perform site-specific functions and are duplicated at distinct physical locations.
  - Allows autonomous operation for each node.
  - A failure at a node will not affect the other nodes in the system.

Component Structuring Criteria (III)

Performance

- Executing a time-critical component exclusively on a single node, without involving components on other nodes, yields more predictable performance.
  - Move less time-critical components to other nodes.

Specialized Hardware

- Run software for specialized hardware on the same node (see proximity to data earlier).

Example:

- Telephone switch — line card, shelf controller, main control.

Design of Service Components

- Service components usually are used for storing and loading data, either by accessing a file system directly, or through a database.
  - Service components typically encapsulate entity objects.
  - Service components usually only respond to requests from other components.
- Service components may be either completely sequential or allow multiple simultaneous readers and writers.
  - Sequential components can only complete one operation at a time and this may be a bottleneck when accessing slow devices such as disks.

Concurrent Service Components

- One approach — multi-threading.
  - Each incoming request is assigned a new thread.
  - Attention must be paid to any critical sections.
  - It may be necessary to limit the number of concurrent threads otherwise system resources may be exhausted.
  - This approach may still lead to too much blocking because read-only requests will still have to wait for the mutex.
    - Therefore, multiple readers can run in parallel, but writers must execute sequentially and after all reads have completed.
    - Beyond the scope for this course.

Distribution of Data

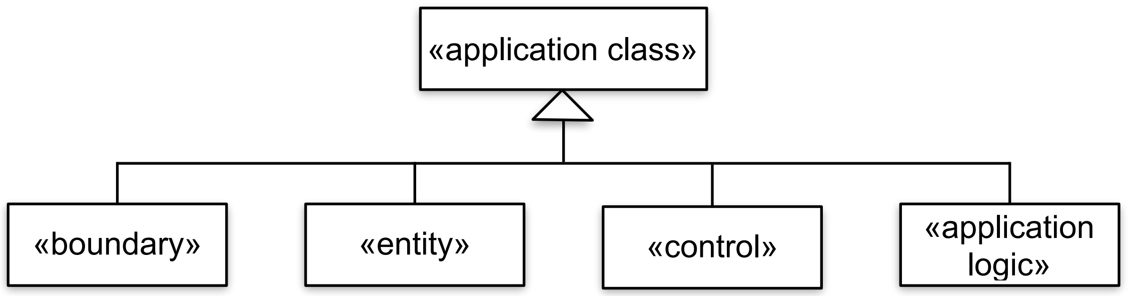
- Service subsystems are usually single-service, so the data is centralized.
- For a distributed system, this is a limitation.
- Two approaches:
  1) Distributed service — data collected at a particular location is also stored there. Clients make requests for data as needed (partitioned)
  2) Replicated data — data is copied over to multiple nodes, clients access the closest copy. Frequent updates may be expensive.

Software Deployment

1. Define the instances of the components — how many are needed?
2. Map the components to physical nodes.
   - More than one component can run on a node.
3. Interconnect the component instances.
   - Time to add software connectors.

SYSC 3303 Real-Time Concurrent Systems

Software Architectures



You are here

1. System structural modelling.
2. Requirements modelling.
3. Analysis modelling:
    1. Static modelling.
    2. Dynamic state machine modelling.
    3. Object structuring.
    4. Dynamic interaction modelling.
4. Design modelling.
5. Incremental software construction (your project!)
6. Incremental software testing (your project!)
7. System testing.

Sequential Software Architecture

- The program has a single thread of control.
- All objects are passive.
- Used by *cyclic executives* (to come).
- Plenty of limitations for embedded real-time systems though.

You are here 😀.

Concurrent Software Architectures

- Rather than a single thread of control, there are multiple threads that can run concurrently.
    - threads can run while others are blocked for external events.
    - the system can run on a multi-core processor, or can be distributed to multiple computers.
    - Multiple input streams can be handled in parallel.
    - The overall design is simplified as each task can do one thing and one thing only.
        - The "Single Responsibility Principle".
            - (NB: You **need to know** the Liskov Substitution Principle too!)
- Which leads us to…

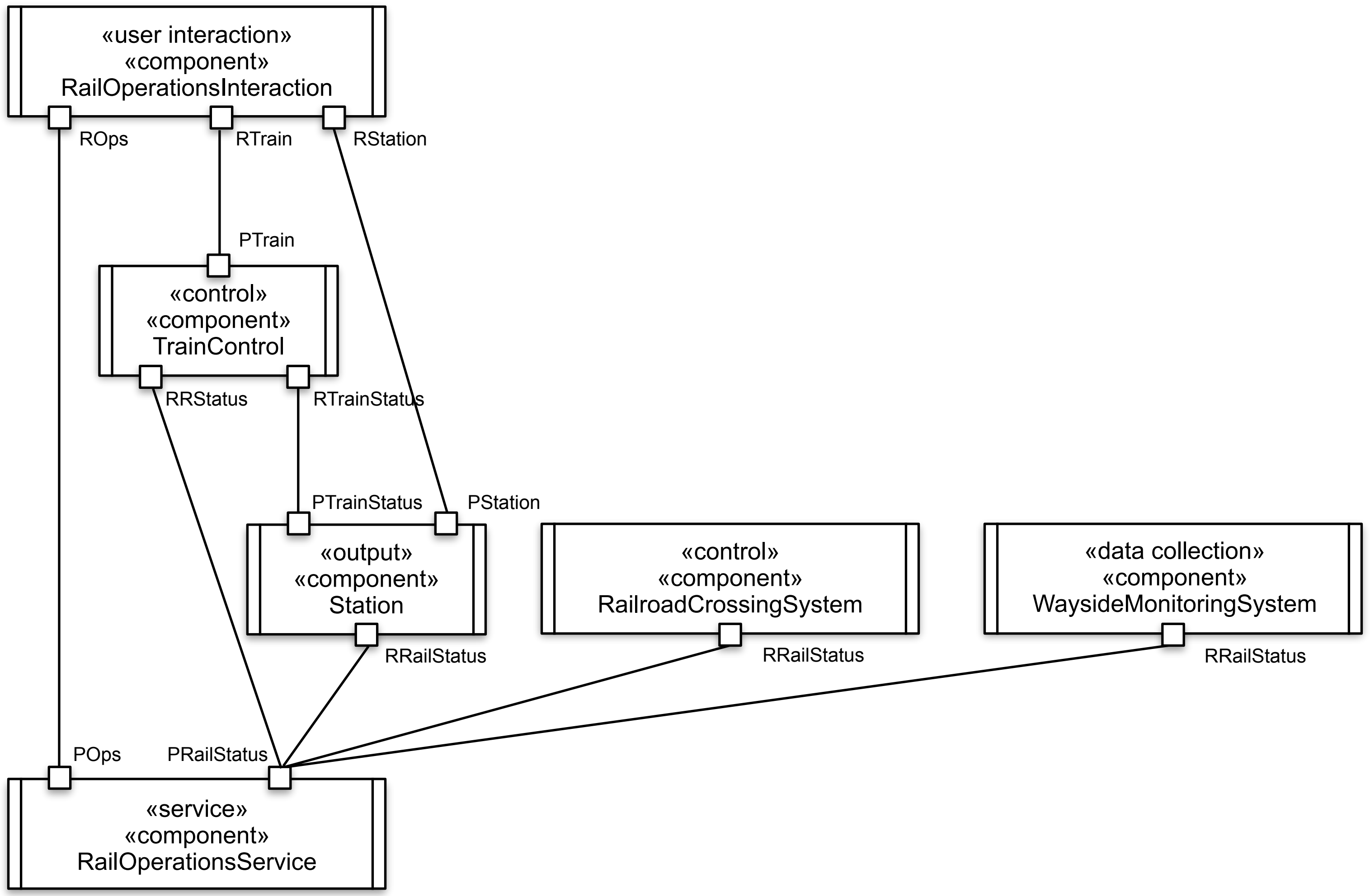
Component-Based Software Architecture

- Each component is self-contained and encapsulate information.
- Components can contain other components (*composite component*).
  - Composite components are often called *subsystems*.
- Components are treated as black boxes.
  - Interact with each other through well-defined interfaces.
- Different types of “glue” can be used to connect components depending on whether they exist on the same node or not.

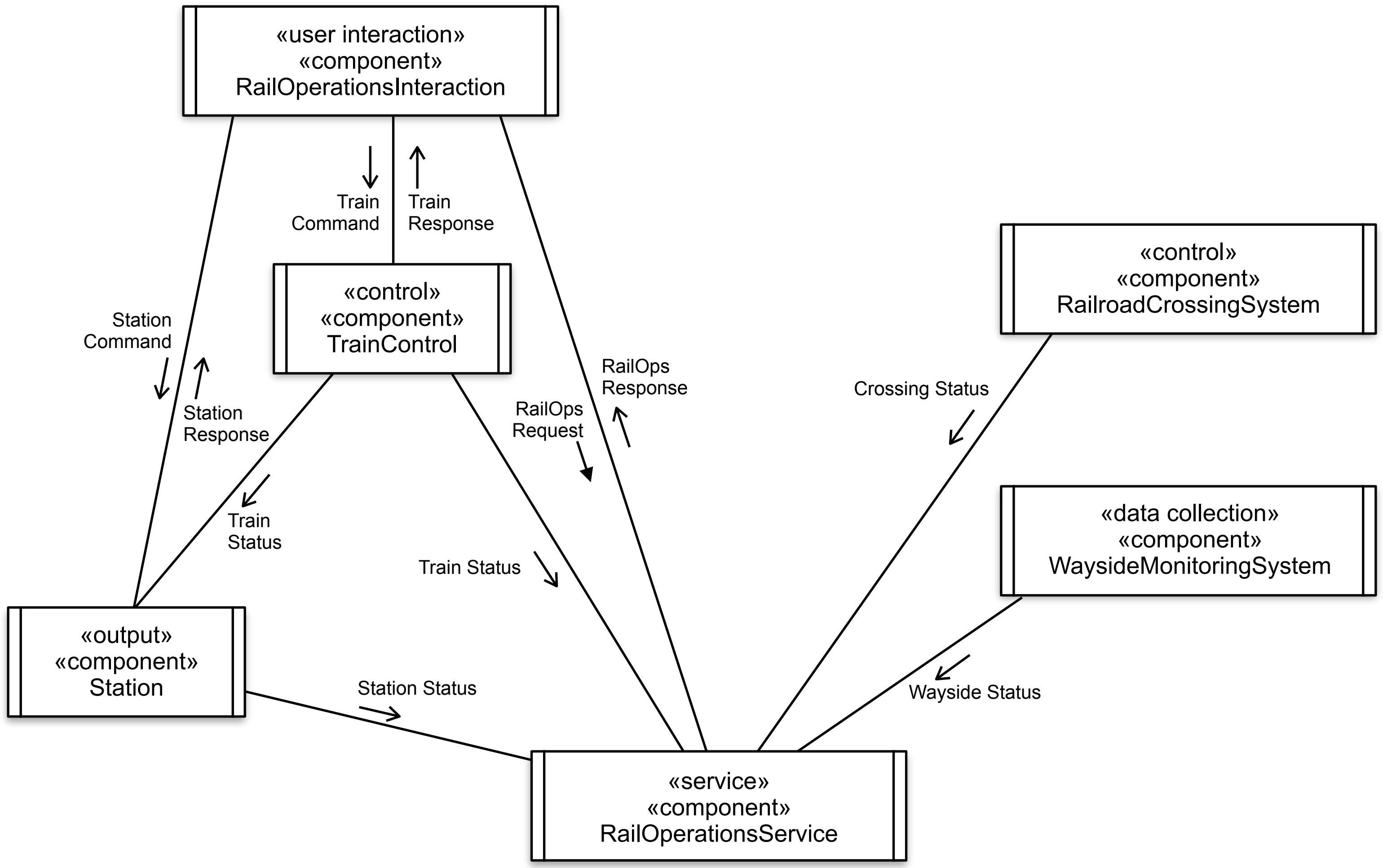
Views of a Software Architecture

- Structural View (next slide):
  - A static view (like a class diagram,).
  - Shows component types, ports and connectors.
  - Role and architecture stereotypes shown.
- Dynamic View (following slide):
  - Shows all possible interactions between objects.
- Deployment View (third slide):
  - Shows the physical configuration of the system.

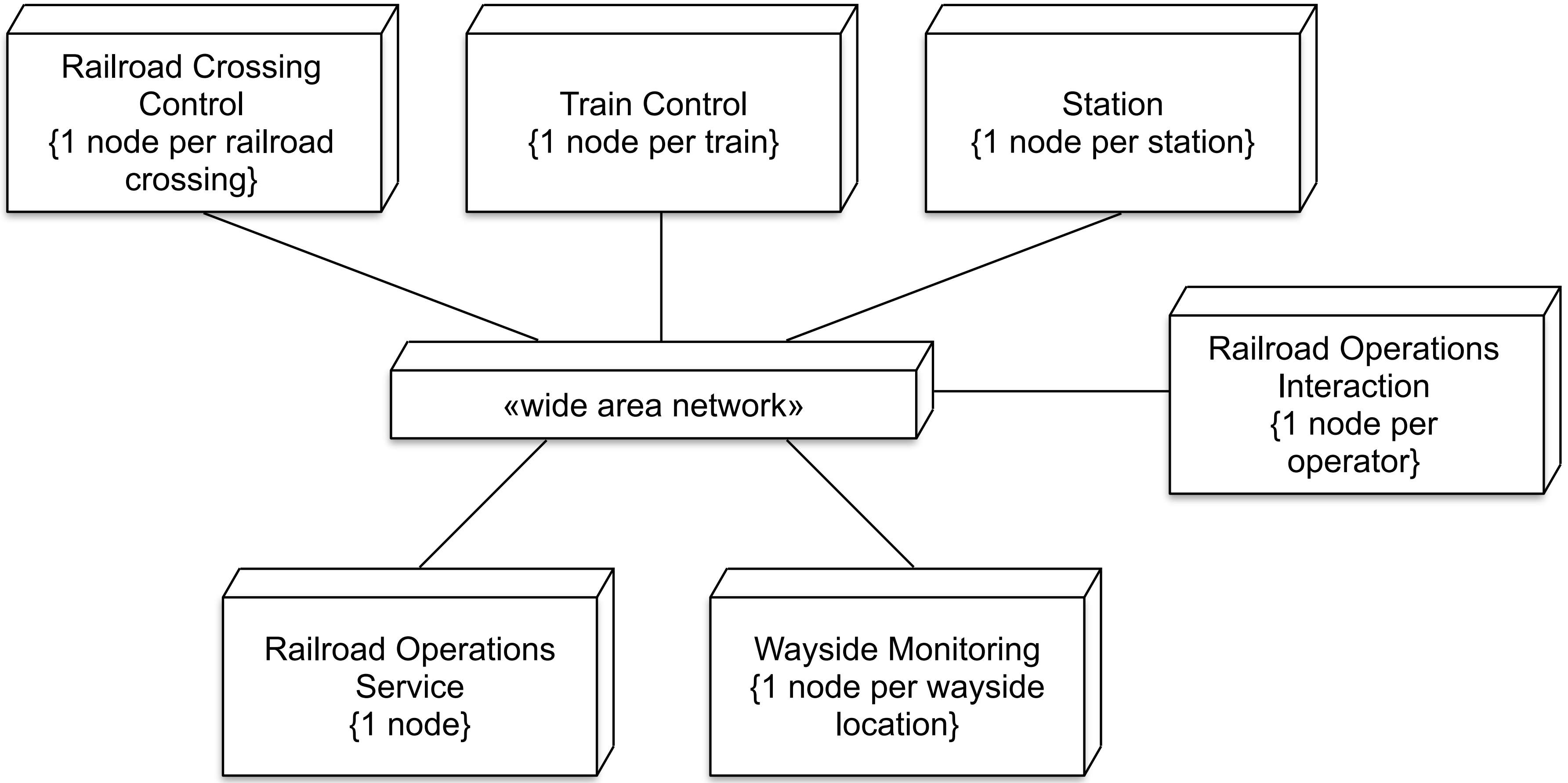
Composite Structure Diagram



Subsystem Communication Diagram

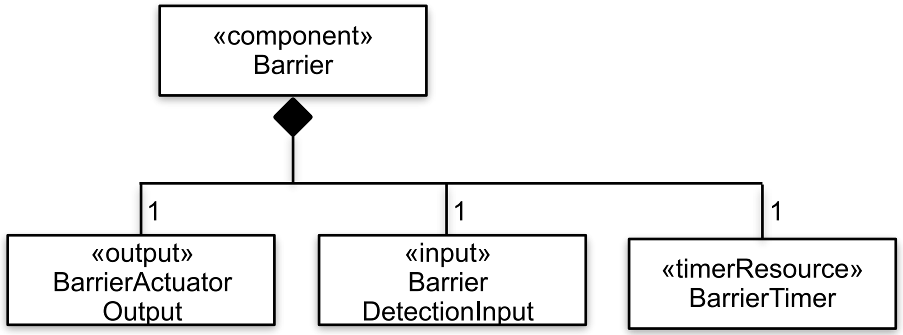


Deployment Diagram

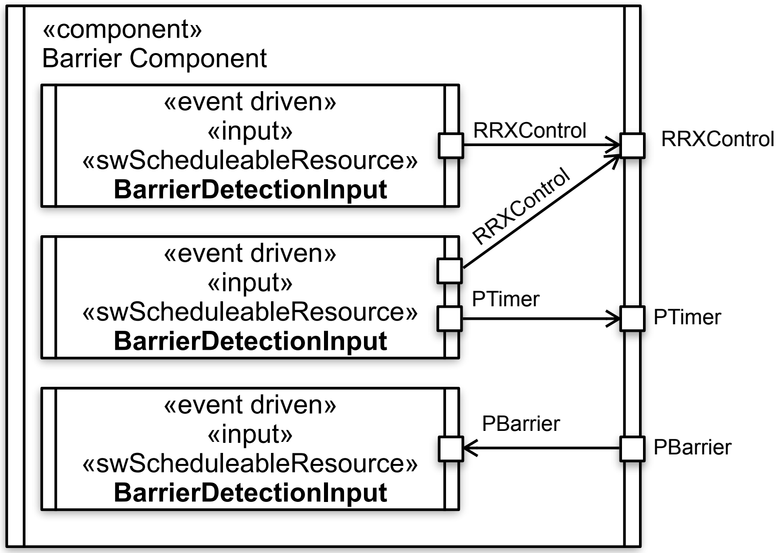

Railroad Crossing Control {1 node per railroad crossing}
Train Control {1 node per train}
Station {1 node per station}
«wide area network»
Railroad Operations Interaction {1 node per operator}
Railroad Operations Service {1 node}
Wayside Monitoring {1 node per wayside location}

Separation of Concerns

- The goal is to make subsystems self-contained so that different concerns are addressed by different subsystems.
  - Subsystems are treated as a whole, so they are a composite object formed by composition, not aggregation.



a) Composite Class



b) Composite Component

Separation of Concerns (II)

- Two objects that *can potentially* be on separate nodes should be in separate subsystems.
- Clients and services should be in separate subsystems.
- User interactions should be in separate subsystems (think Model-View-Controller).
- An external component should only interface to one subsystem.
- A control object and the entity objects that it interacts with should all be part of the same subsystem.

Subsystem Structuring

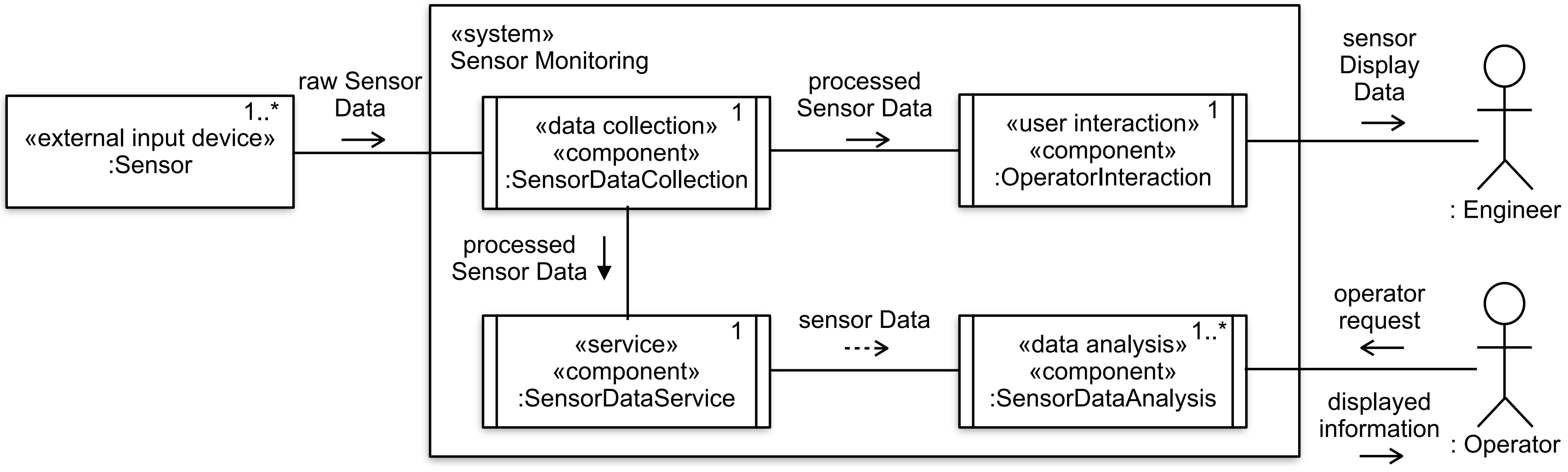
- Control Subsystem
  - Receives inputs from and sends output to the external environment.
  - Usually state-dependent.
  - May interact with other subsystems as needed.
- Coordinator Subsystem
  - Found in systems with multiple control subsystems to coordinate their actions
  - Can you think of an example? 😀

Subsystem Structuring (II)

- User-Interaction Subsystem
  - Provides the user interface and performs the role of the client in a client-server system.
  - There may be more than one.
  - It may also be a composite object consisting of several simpler user-interaction objects.
  - It may contain entity objects for local storage and caching.
- Input/Output Subsystem
  - Performs input and output on behalf of other subsystems.
  - It may contain entity and control objects.

Subsystem Structuring (III)

- Data Collection Subsystem
  - Collects data from the external environment.
  - It may store data, or simply perform data reduction and pass the result on to another subsystem.
- Data Analysis Subsystem
  - Provides reports and analysis.
  - Often not real-time.

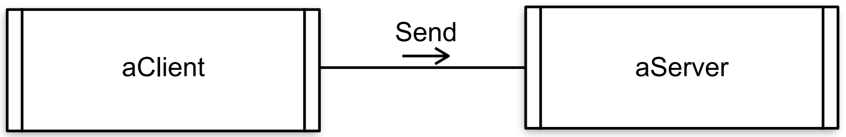


Subsystem Structuring (IV)

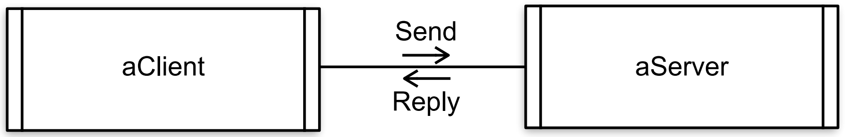
- Client Subsystem
  - Essentially any UI.
  - May communicate with one subsystem or many subsystems.
- Service Subsystem
  - Provides some sort of service to other subsystems.
  - Does not initiate actions on its own.
  - May be composed of entity and/or coordinator objects.

Message Communication Between Subsystems

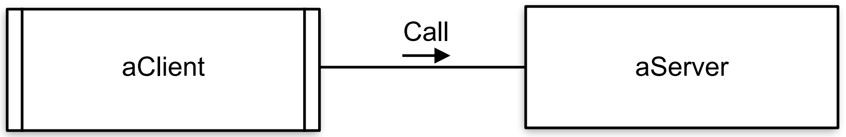
- The interface specification between subsystems.
  - Asynchronous or Synchronous?
  - Unidirectional or bidirectional?



a) Asynchronous, one way.



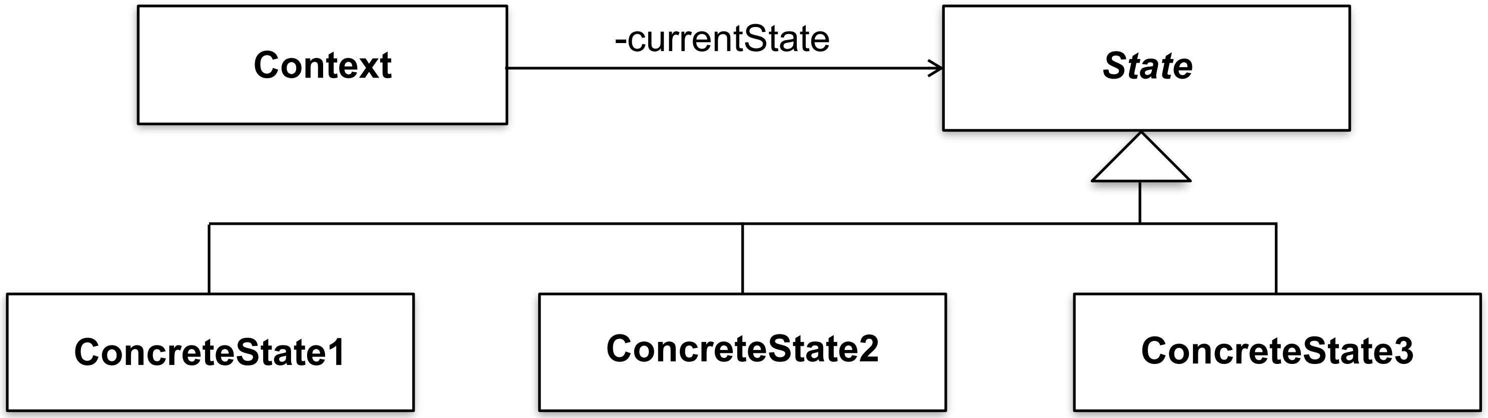
b) Asynchronous, two ways (Remote Procedure Call).



c) Synchronous, two ways (Procedure Call).

Software Design Patterns

- A *design pattern*:
    a) describes a recurring problem to be solved,
    b) a solution to the problem, and
    c) the context in which the solution works.
- It's a larger-grained form of reuse than a class.
- It involves more than one class along with their interactions.
- Example: the "state pattern" for implementing state machines shown earlier in this course.



Kinds of Patterns

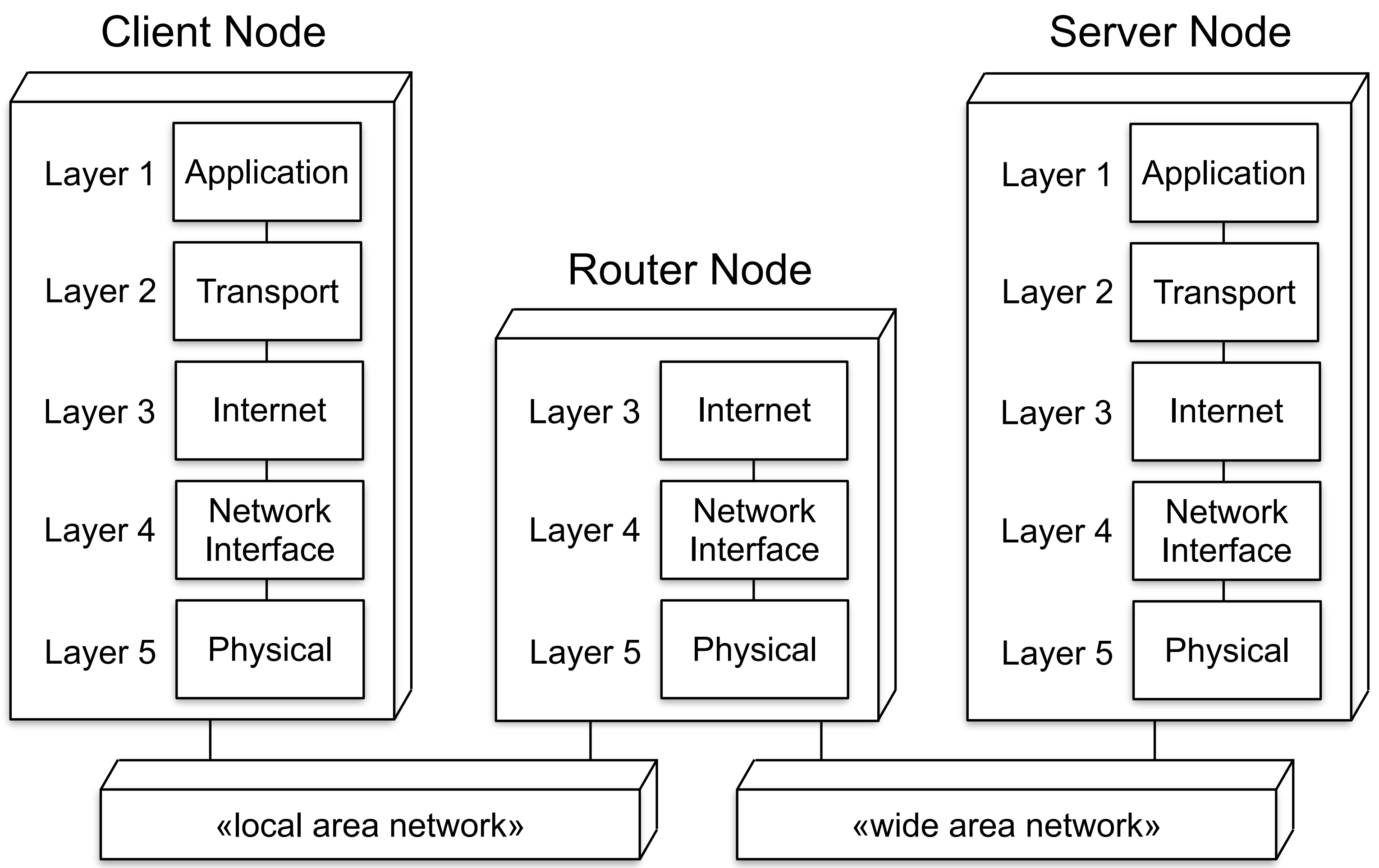
- Design Patterns – a small group of collaborating objects (See the “Gang of Four” Gamma et. al. 1995).
- Architectural Patterns – larger-grained than design patterns, addressing the structure of major subsystems (Buschmann et. al. 2007).
- Analysis Patterns – similarities found during the analysis of different problem domains.
- Domain-Specific Patterns – design patterns tailored to a specific application domain.
- Idioms – low-level patterns tailored to a specific programming language (Eg “Slide 62”).
- Design Anti-Patterns – bad code seen again and again.

Control Patterns for Real-Time SW Architectures

1. Layered
2. Control Patterns for Real-Time
    1. Centralized control
    2. Distributed Collaborative Control
    3. Distributed Independent Control
    4. Hierarchical Control
    5. Master/Slave Control
3. Client-Server

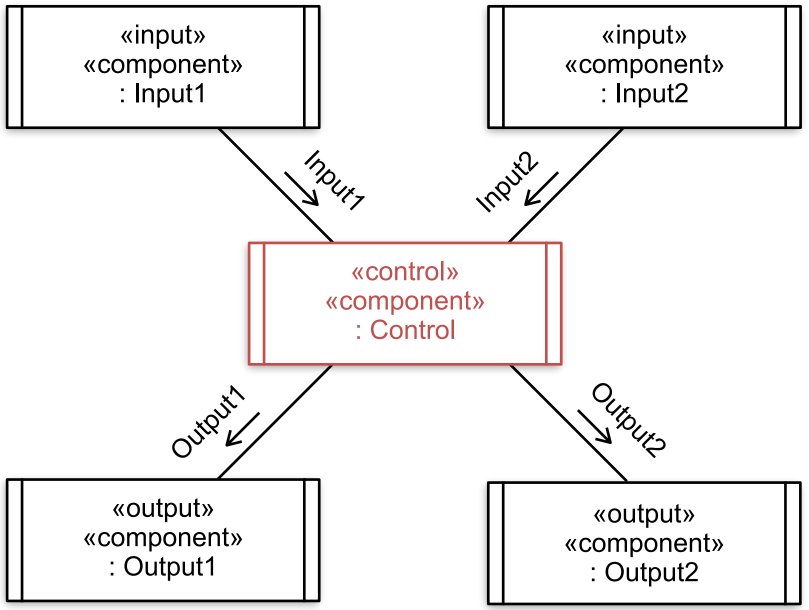
Layered

- Layer uses services provided by the layer below it.
  - Strict (immediate) versus loose (any lower layer).



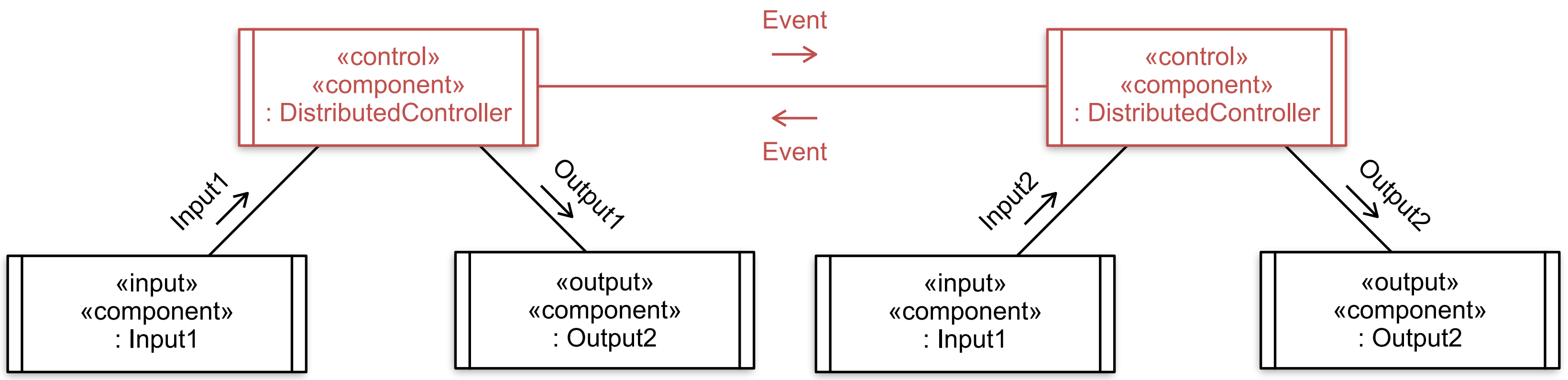
Centralized Control

- Only one control component.
- All inputs go to, and all outputs originate from the centralized control component.
  - Eg: nuclear power station.



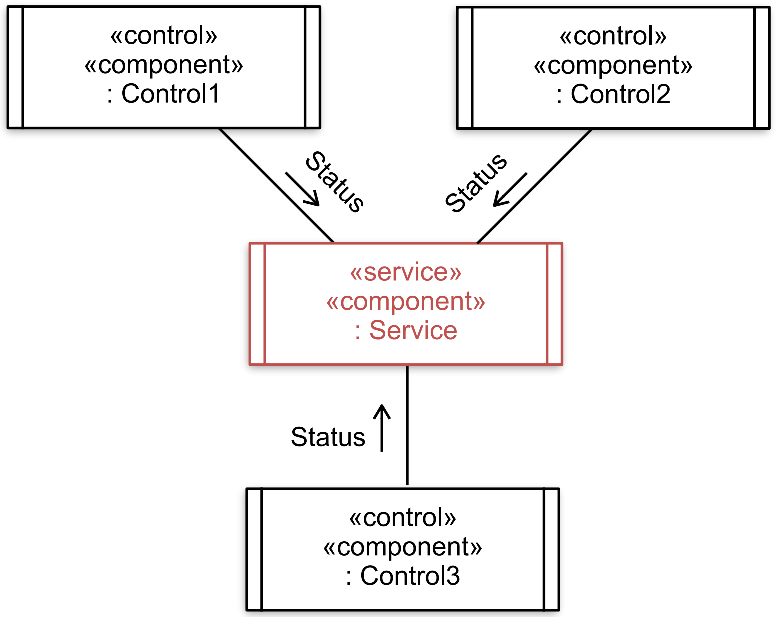
Distributed Collaborative Control

- Several Control Components exist.
  - Each controller is a peer to the others.
  - Each controls a subset of the overall system.
  - Control Components exchange events for overall coordination.
  - Eg: Air traffic control system.


«control» «component» : DistributedController
Event
Event
«control» «component» : DistributedController
Input1
Output1
Input2
Output2
«input» «component» : Input1
«output» «component» : Output2
«input» «component» : Input1
«output» «component» : Output2

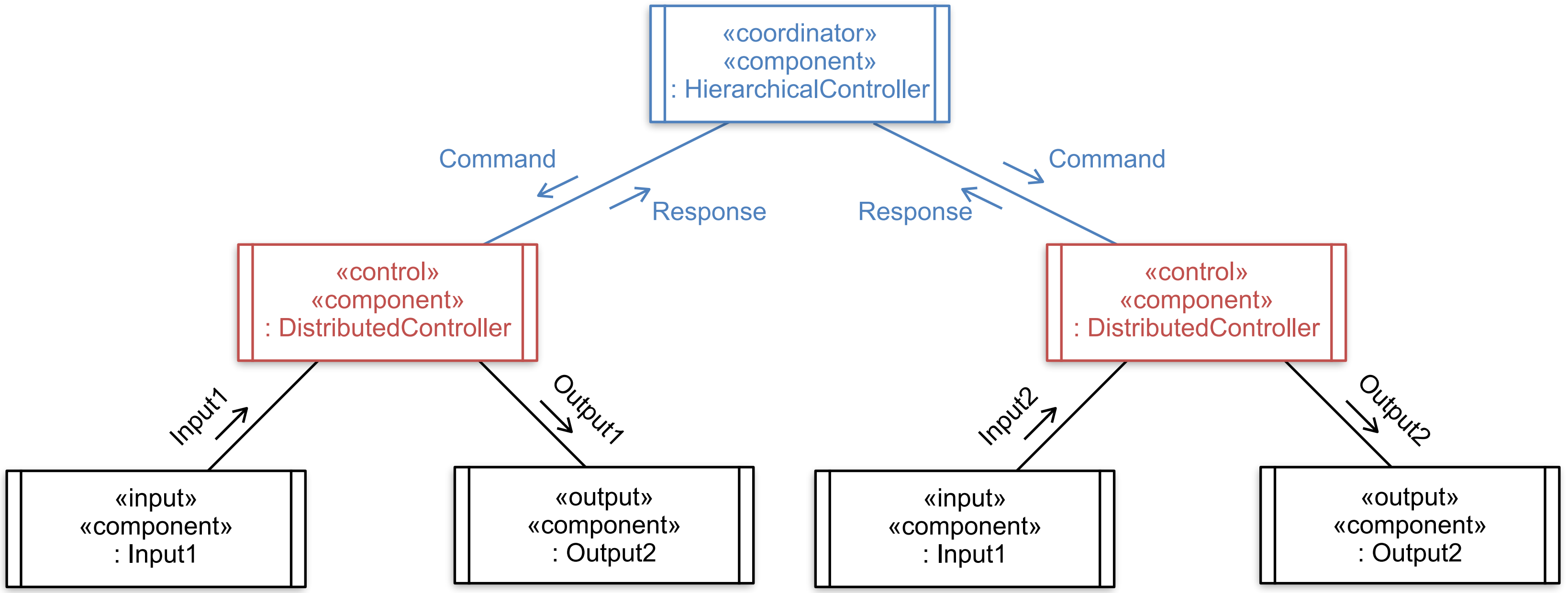
Distributed Independent Control

- Control is distributed to a set of control components.
  - However, no communication between the control components.
  - The "service" does not provide coordination to the control components.
  - ex: Railway crossing and Rail operations.



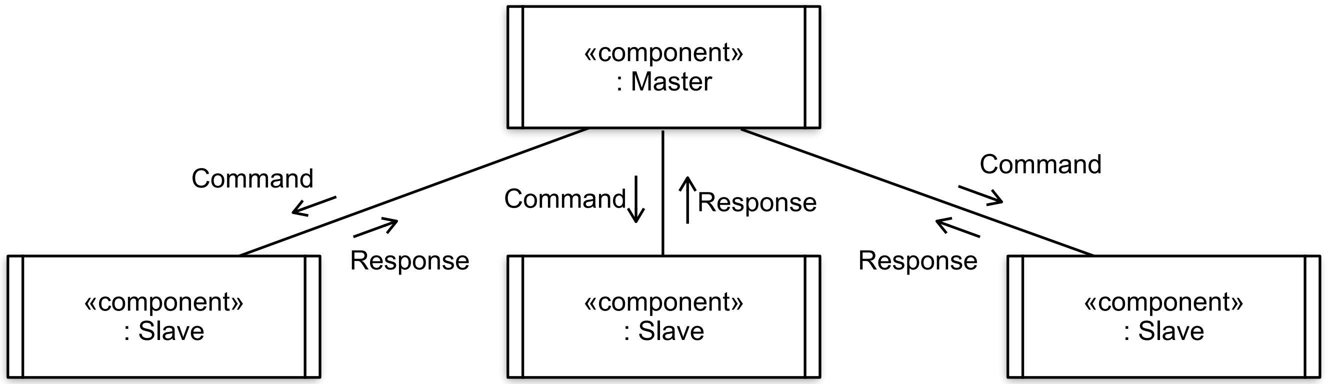
Hierarchical Control

- Top-level component coordinates actions of control components.
  - May nest, eg: main control, shelf control, line card…
  - Eg: telephone switch, your project 😱



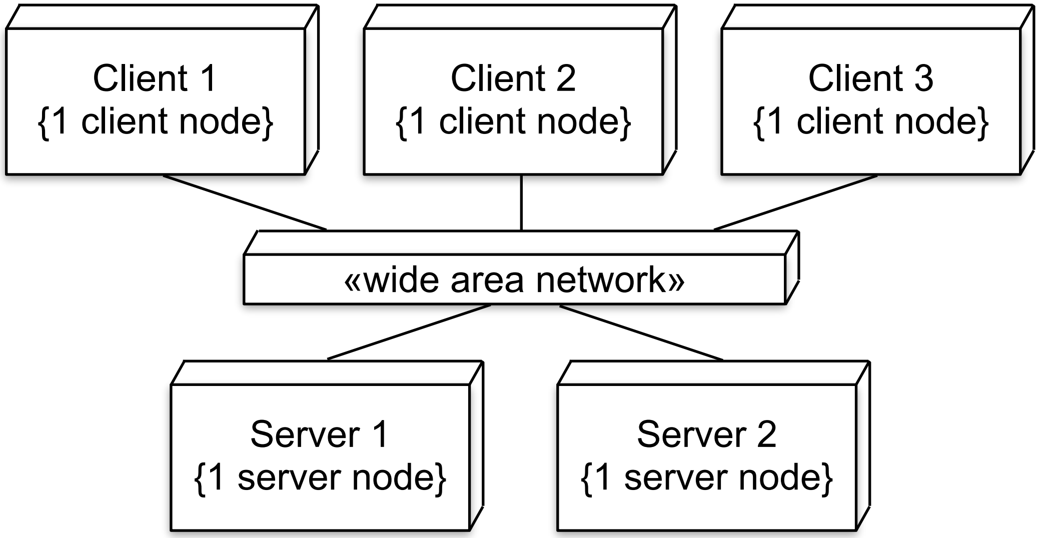
Master/Slave Control

- There is one control component (the master) that divides work among the slaves.
  - The master integrates the results of the slaves.
  - The slaves do not interact with each other.
  - The slaves do not have control objects.
  - Eg: A GPU.



Client-Server

- Client *nodes* (which may be embedded systems) communicate with one or more servers.
  - More general than embedded systems
  - May be hierarchical (three-tier client-server).
  - Eg: World Wide Web (Chrome connects to Apache).

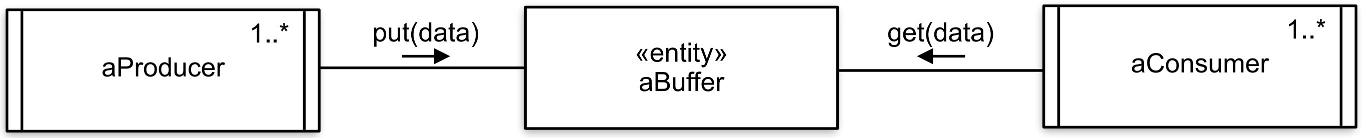


Basic Software Communication Patterns

- Addresses the way concurrent and/or distributed components communicate with each other.

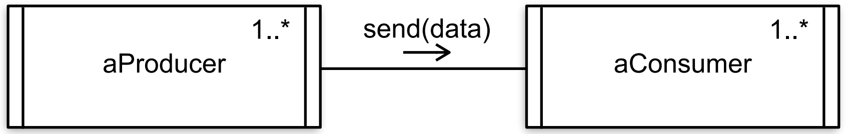
Synchronous Object Access Pattern

- Two or more active objects share data through a passive object.
  - Mutual exclusion is required for the shared data.
  - Eg: Producer-Consumer.



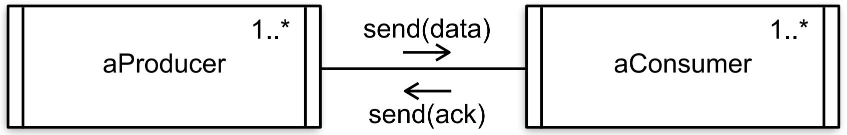
Asynchronous Message Communication Pattern

- The sender sends without waiting.
  - The data is sent in the message.
  - The producer continues on right after the send.
  - Messages may queue (wait) at the consumer.
    - If the producer sends faster than the consumer receives, messages will be dropped.
  - The consumer waits if no messages are available.



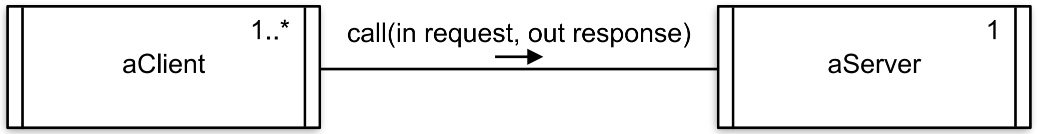
Bidirectional Asynchronous Message

- Used when a the sender needs a reply, but not immediately.
  - Multiple sends from the producer can occur before a send from the consumer back is required.
  - Peer-to-peer communication.
  - Eg: TCP (multiple packets can be transmitted before and ACK is required).



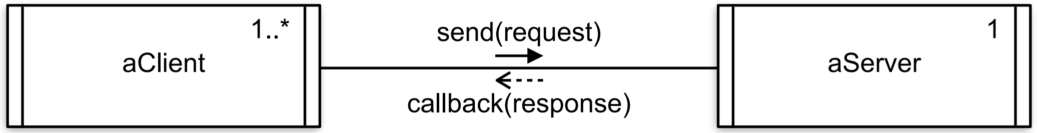
Synchronous Message with Reply

- The client sends a message to the server then blocks.
- The server receives the message, processes it, then replies.
    - Remote Procedure Call (RPC) or Remote Method Invocation (RMI)
    - The client and server can reside on the same node, they just have to be different threads.
    - The send and reply messages can be asynchronous.



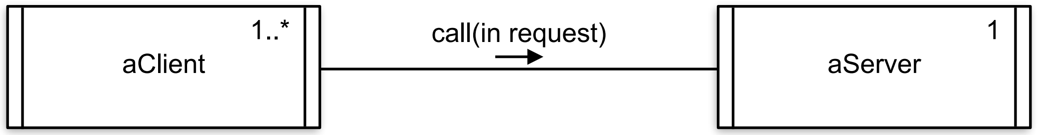
Asynchronous Message with Callback

- The client does not need to wait for a reply from a server, but does expect a response at some point in the future.
  - Asynchronous Remote Procedure Call
  - Like the client-server paradigm, but the client can run a bit more asynchronously.
  - Similar to the bidirectional asynchronous pattern, except that only one request can be outstanding at a time.



Synchronous Message without Reply

- The producer blocks until the consumer receives the message.
  - As soon as the message is received, the producer is released.
  - No reply — one-way data flow.
  - Throttles the producer if the consumer is too slow.
  - Eg: transputer.



Comparison

- Asynchronous Communication:

  (+) More parallelism — sender continues after message sent.

  (-) More copying (same node) — from sender to message then from message to receiver.

  (-) Lost buffers — if sender is faster than receiver.

- Synchronous Communication:

  (+) Less copying (same node) — sender and receiver can share address space.

  (-) Less parallelism — sender blocked until receiver replies (this may not be a bad thing).

Broker Patterns

- Used with client-server architectures where the client queries the broker to locate the server.
  - Provides location transparency — client only has to know the location of the broker, the server can move around.
  - Provides platform transparency — server can run on different hardware/software platforms.
  - Broker may simply be queried once to provide server information, or the broker may act as an intermediary and forward requests to a server.
  - A little more overhead — client must ask the broker for server information before accessing the server.

Group Message Patterns

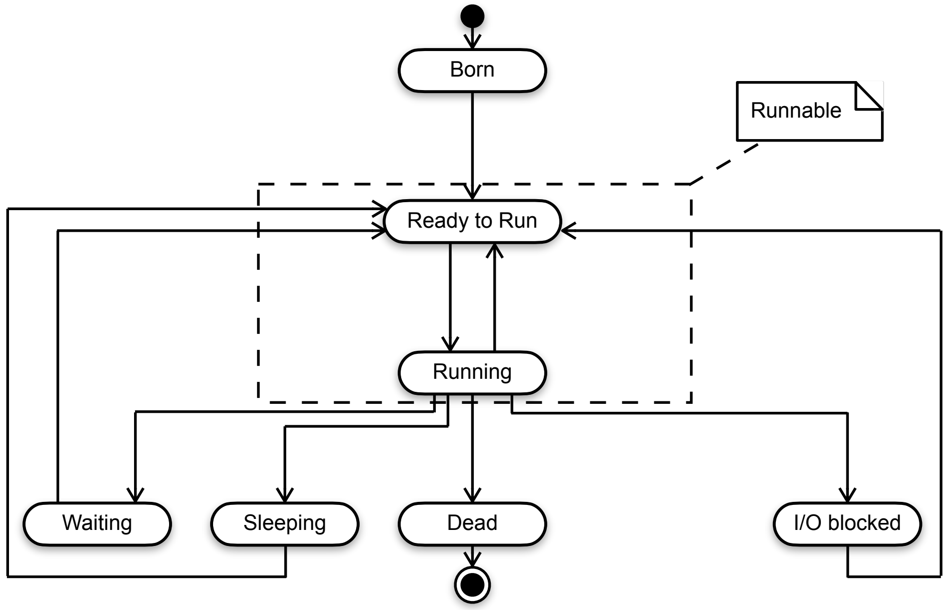
- Broadcast:
  - Messages are sent to all potential recipients regardless of whether the message is wanted or not.
  - Recipients simply ignore messages that they are not interested in.
  - More message traffic.
- Subscription/Notification:
  - Recipients register with a group for messages.
  - Publisher sends messages to the group without knowing who is in the group.
    - This is the “observer” pattern.

Summary

- Architectural patterns are used to address the structure of a software architecture.
- The job in software design is to employ an appropriate pattern for the task at hand.
- The pattern chosen can often be found from dynamic modelling of the system.
- More than one pattern may be used.

SYSC 3303 Real-Time Concurrent Systems

Life Cycle of Java Threads
Scheduling Java Threads



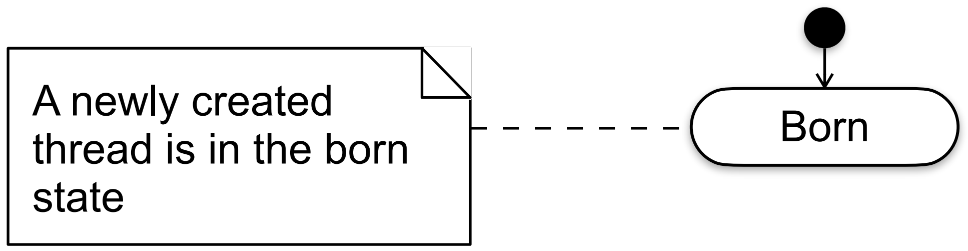
Introduction

- The lifecycle of a thread, its states, and transitions between the states.
- Preemption.
- Thread priorities.
- Priority inversion, what it is, why it’s bad and how to deal with it.
- The Red-Green show

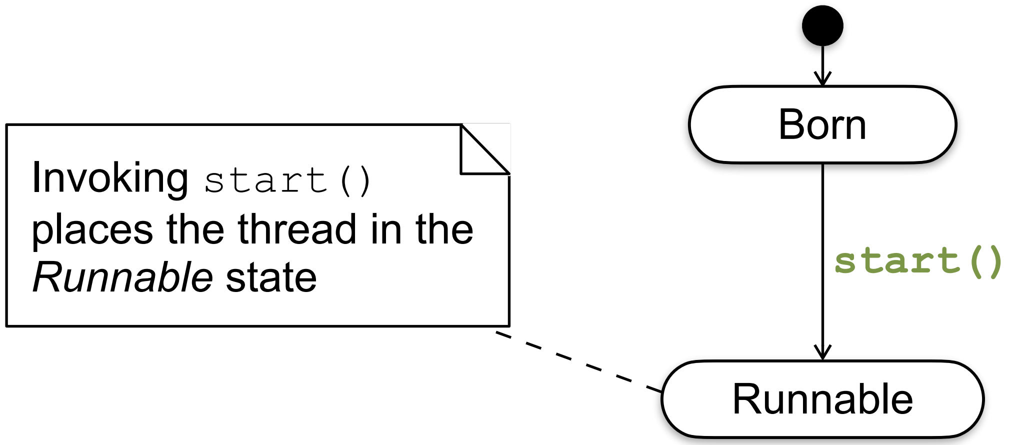
Every Thread Has a Life Cycle

- A thread's life cycle can be modelled by a *finite state machine* (FSM) that represents the various states that the thread can be in and the thread operations that cause the transitions between the states
- We'll now look at each of these states and state transitions

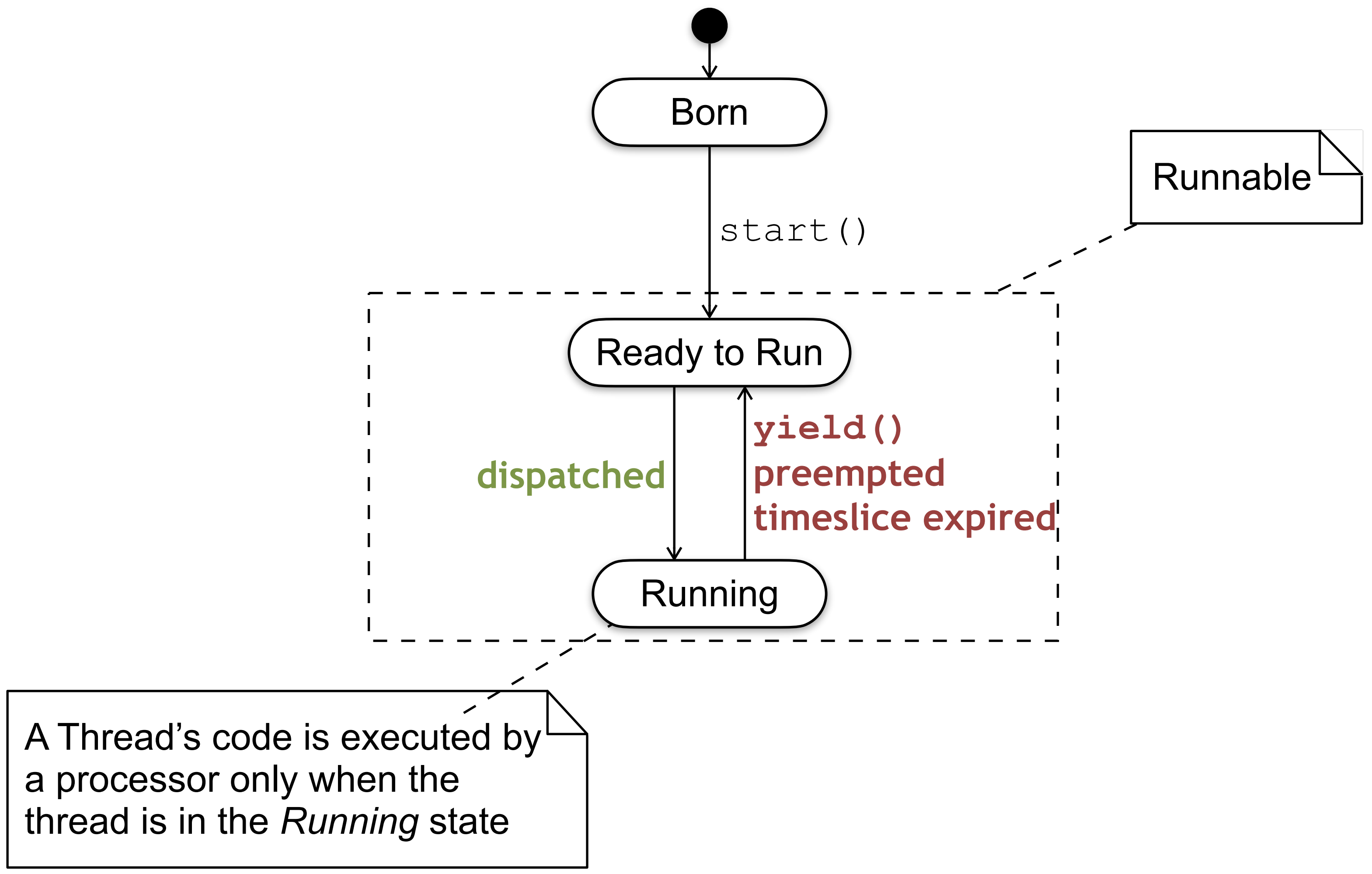
Initial Thread State



Starting the Thread



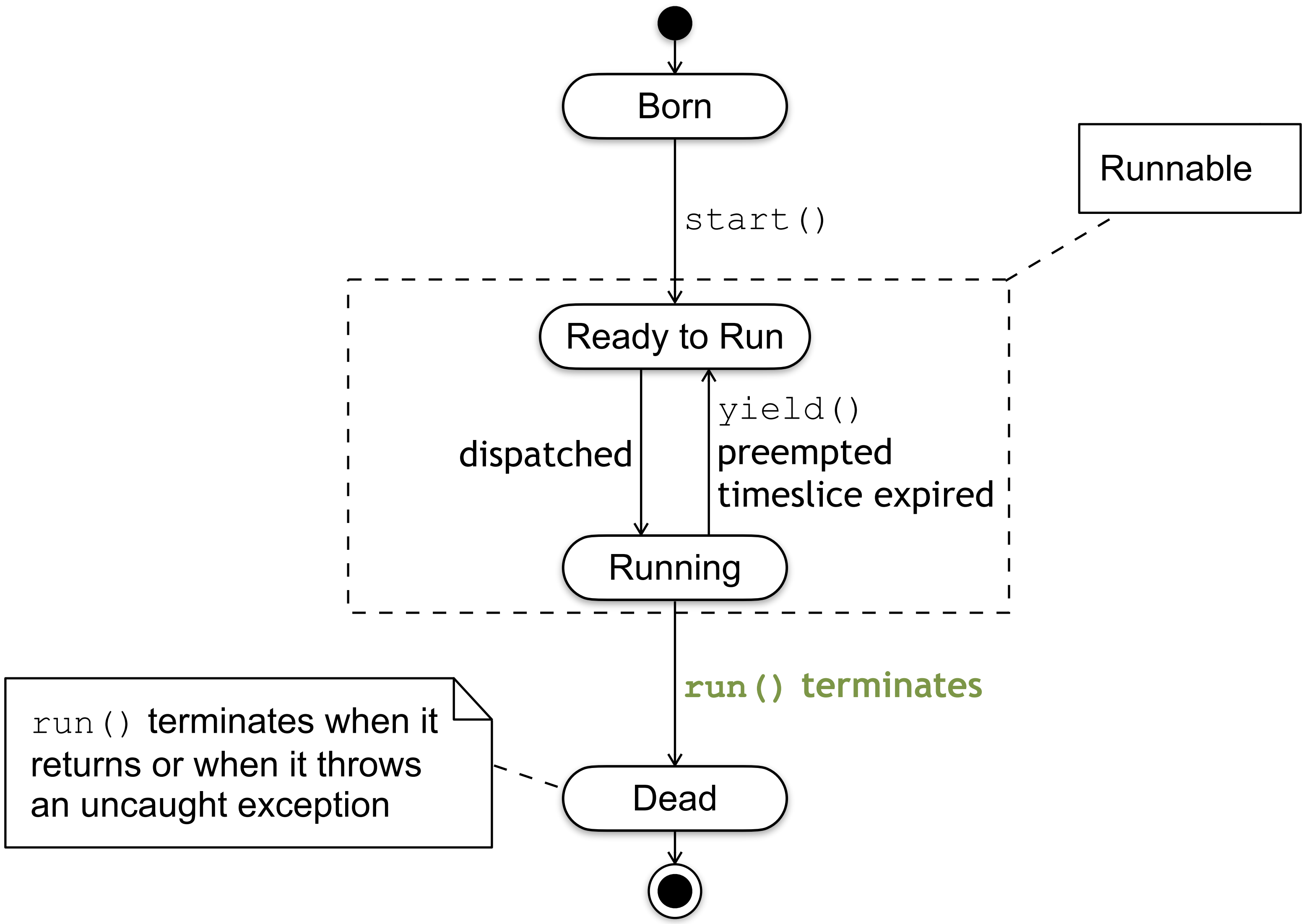
Ready to Run versus Running


Born
start()
Runnable
Ready to Run
dispatched
yield()
preempted
timeslice expired
Running
A Thread's code is executed by a processor only when the thread is in the *Running* state

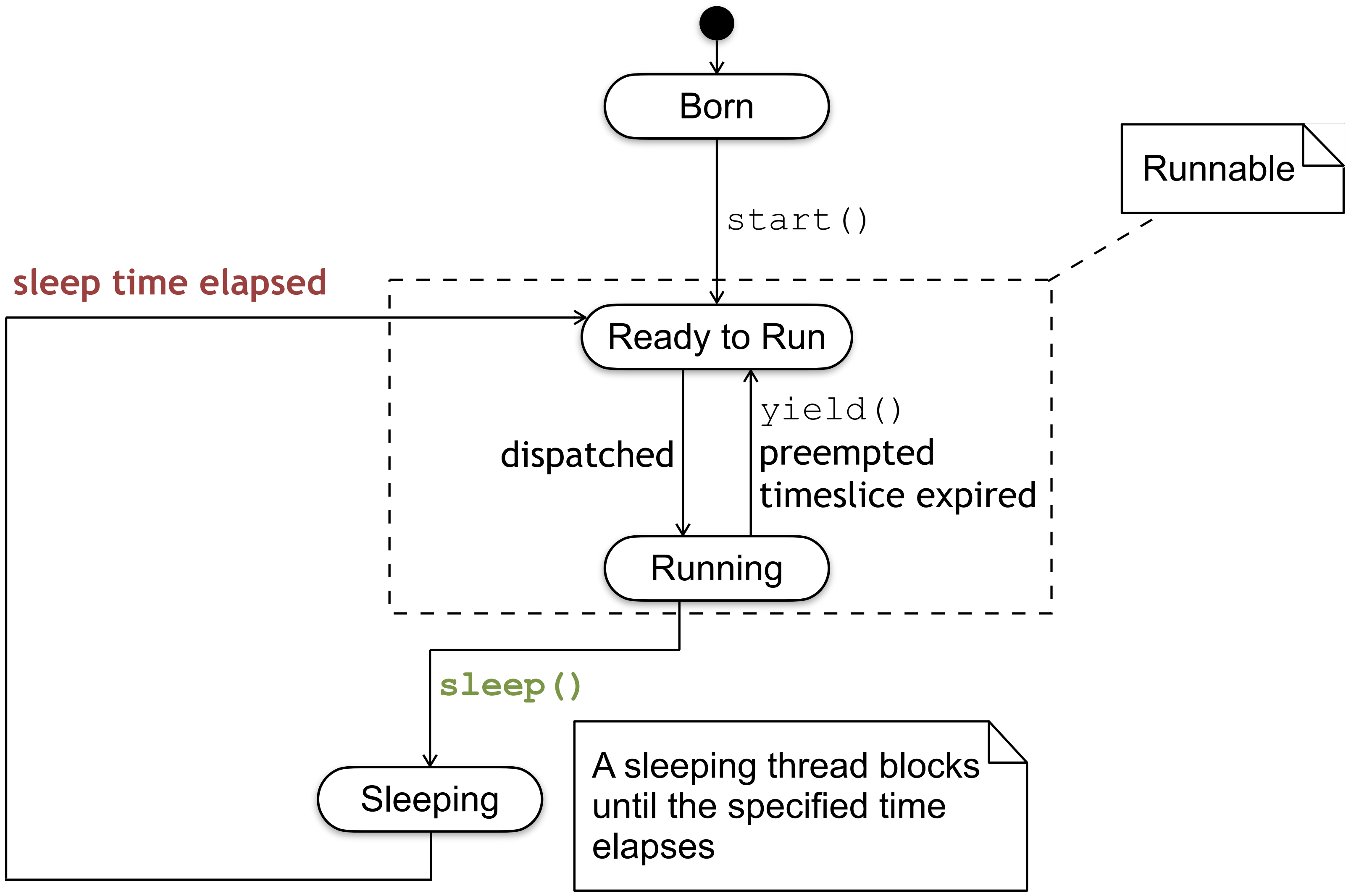
Ready To Run vs. Running

- If we have sufficient processors (parallel thread execution) all threads would be running (a.k.a. *active*, *current*) whenever they are in the Runnable state
- If we do not have enough processors, concurrency is realized by sharing the processors between the Runnable threads, so we distinguish between threads being *Ready to run* and *Running*
  - on a single processor computer, at most one thread at a time is in the Running state
  - on a multi-processor computer, at most one thread per processor is in the Running state

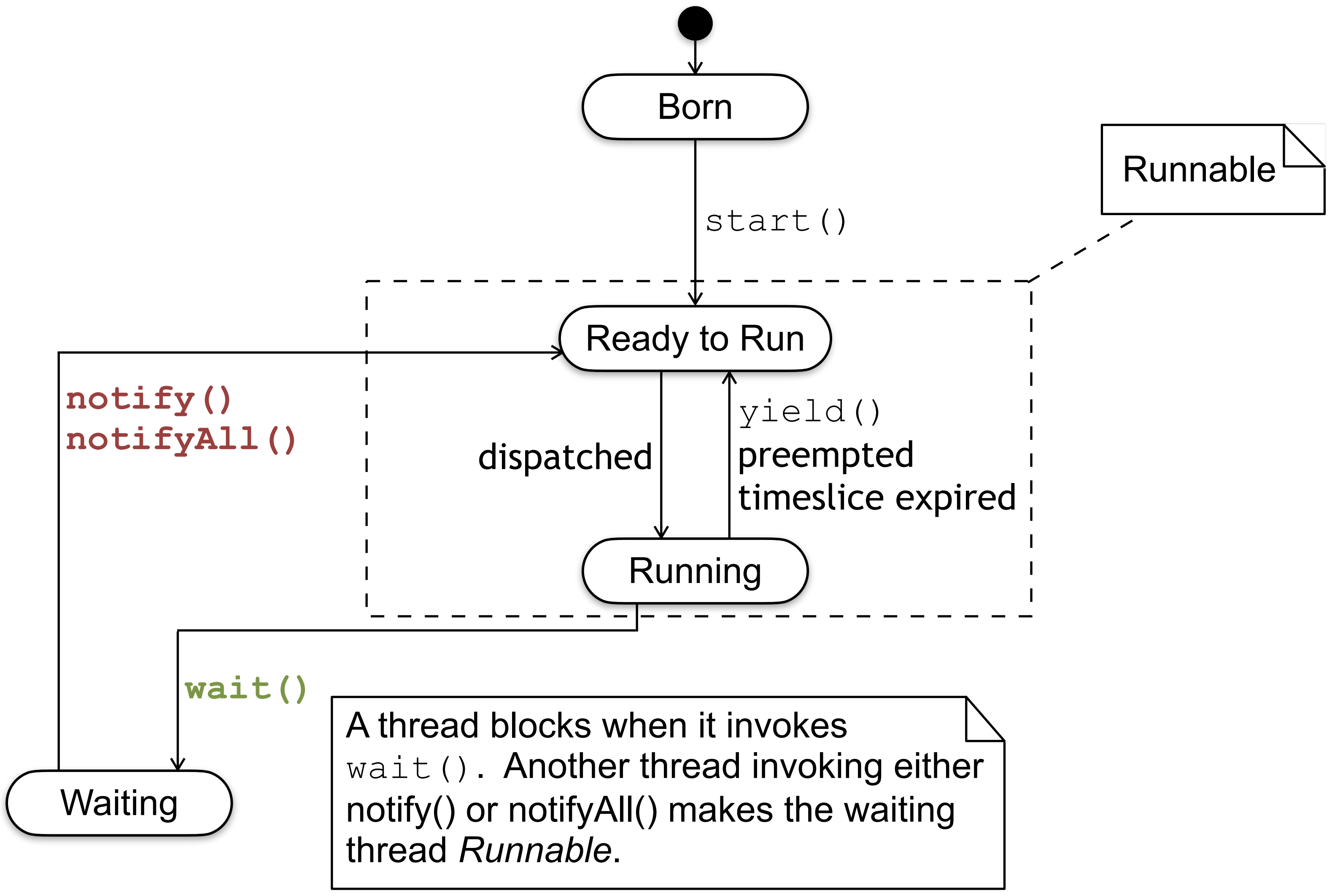
Thread Death



Thread Sleeping

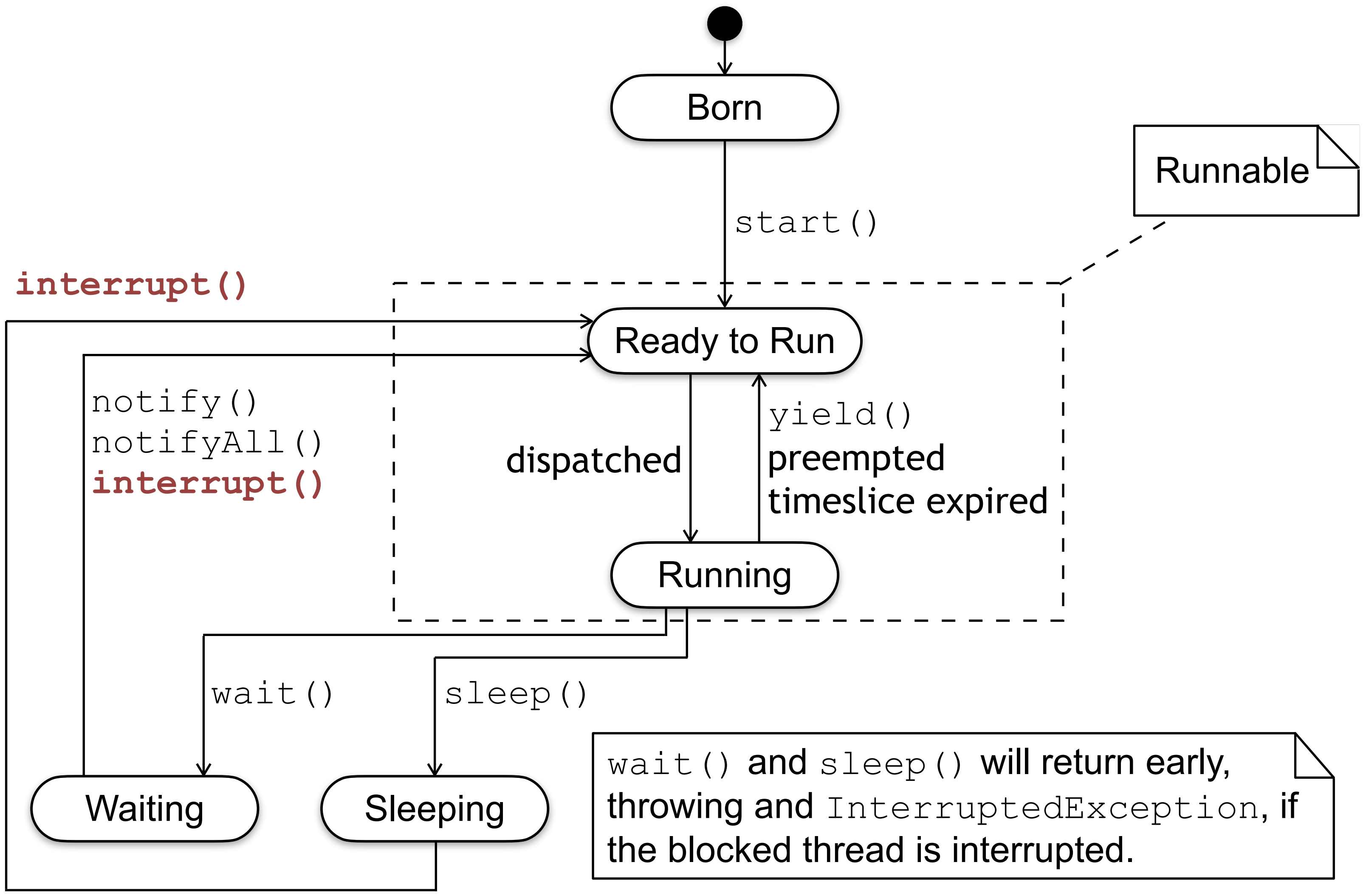


Thread Waiting



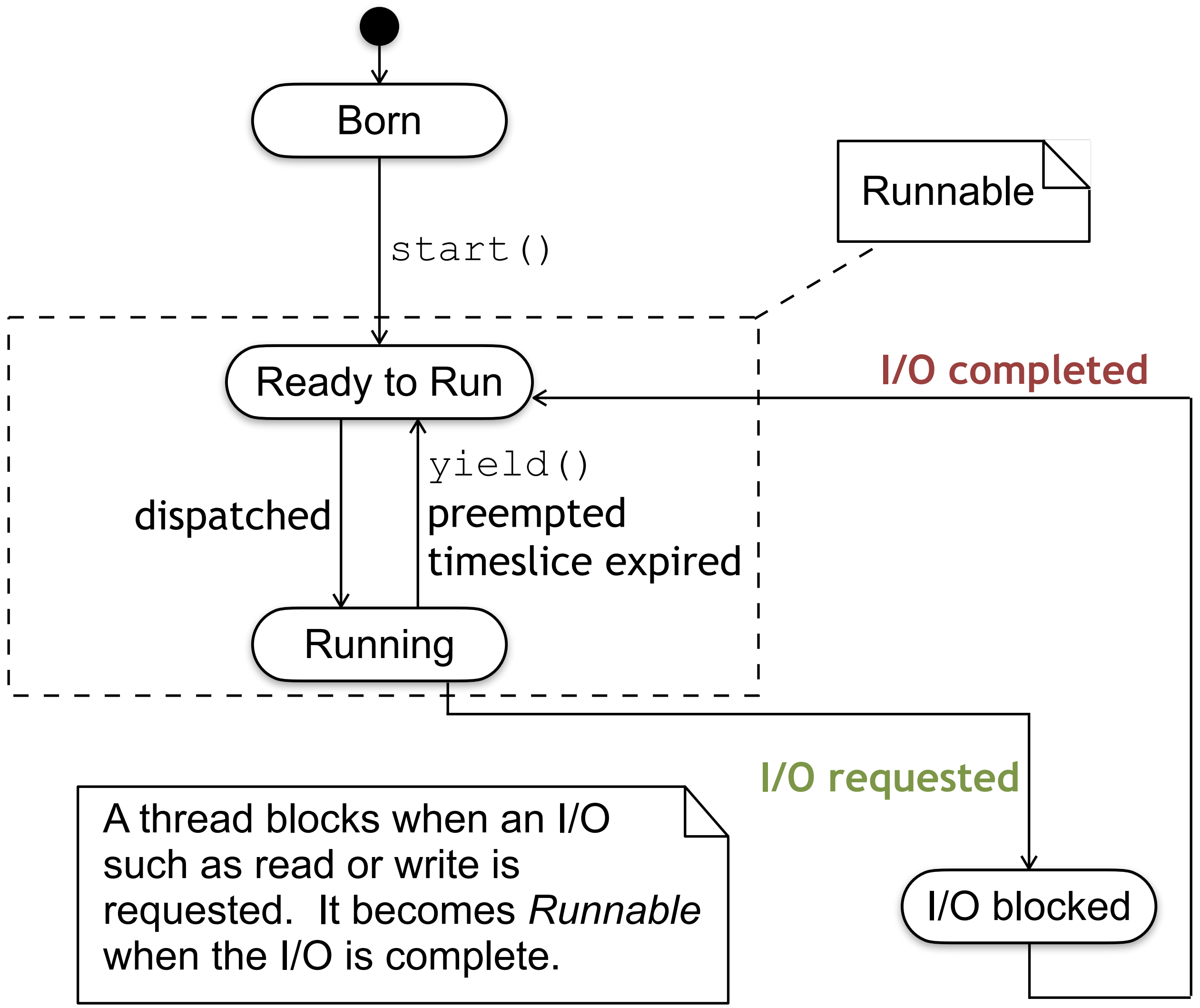
A thread blocks when it invokes `wait()`. Another thread invoking either notify() or notifyAll() makes the waiting thread *Runnable*.

Thread Interruption



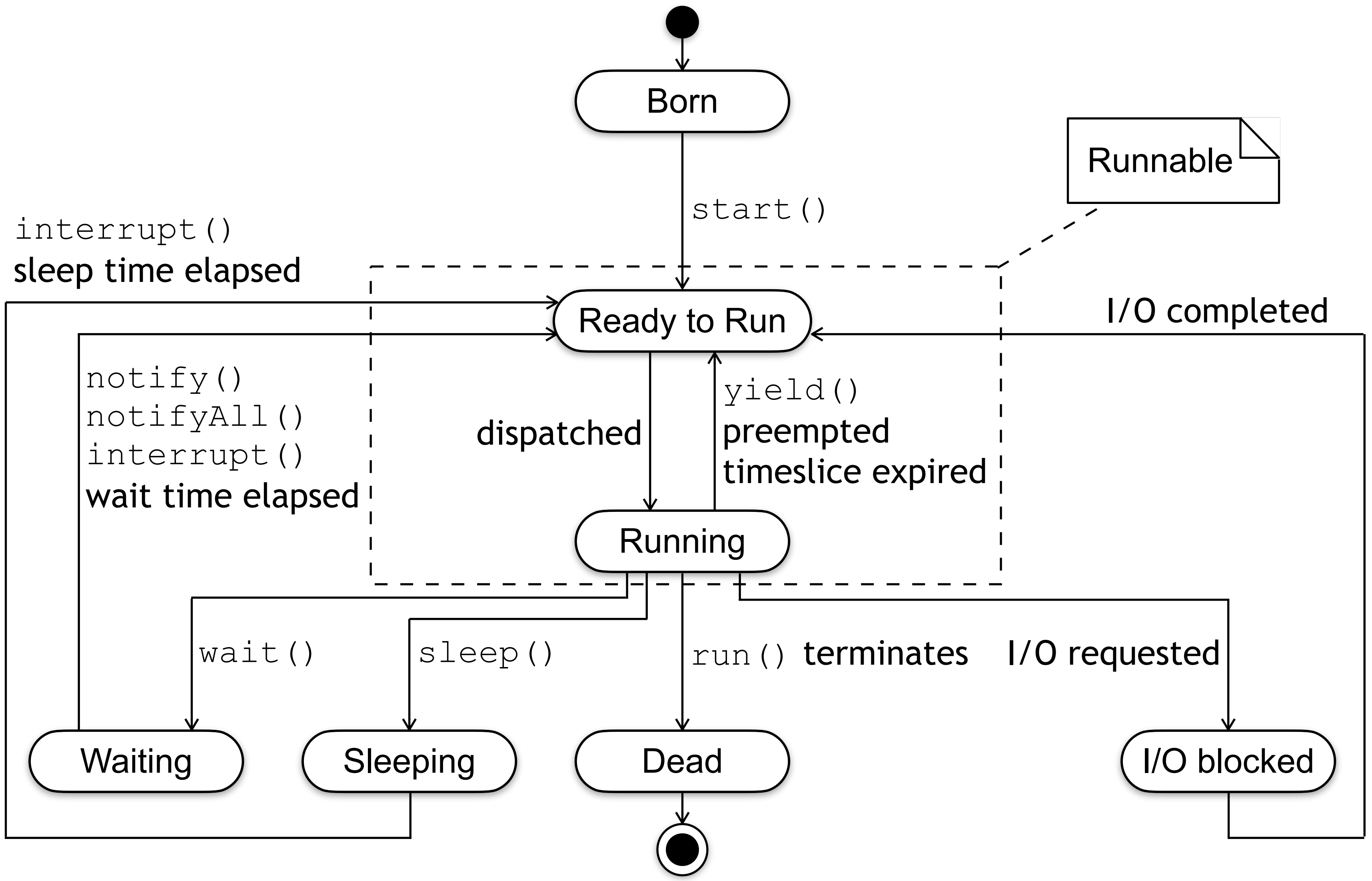
`wait()` and `sleep()` will return early, throwing and `InterruptedException`, if the blocked thread is interrupted.

Thread I/O Blocking



A thread blocks when an I/O such as read or write is requested. It becomes *Runnable* when the I/O is complete.

Thread Life Cycle



Key Ideas About Thread Execution

- At any time, only one thread per processor is executing
- A thread can invoke a method only when it is executing
- Methods are always executed in the invoking thread's context
- If a method invocation causes a thread to relinquish the processor, that method does not return until the thread is once again scheduled for execution and is running

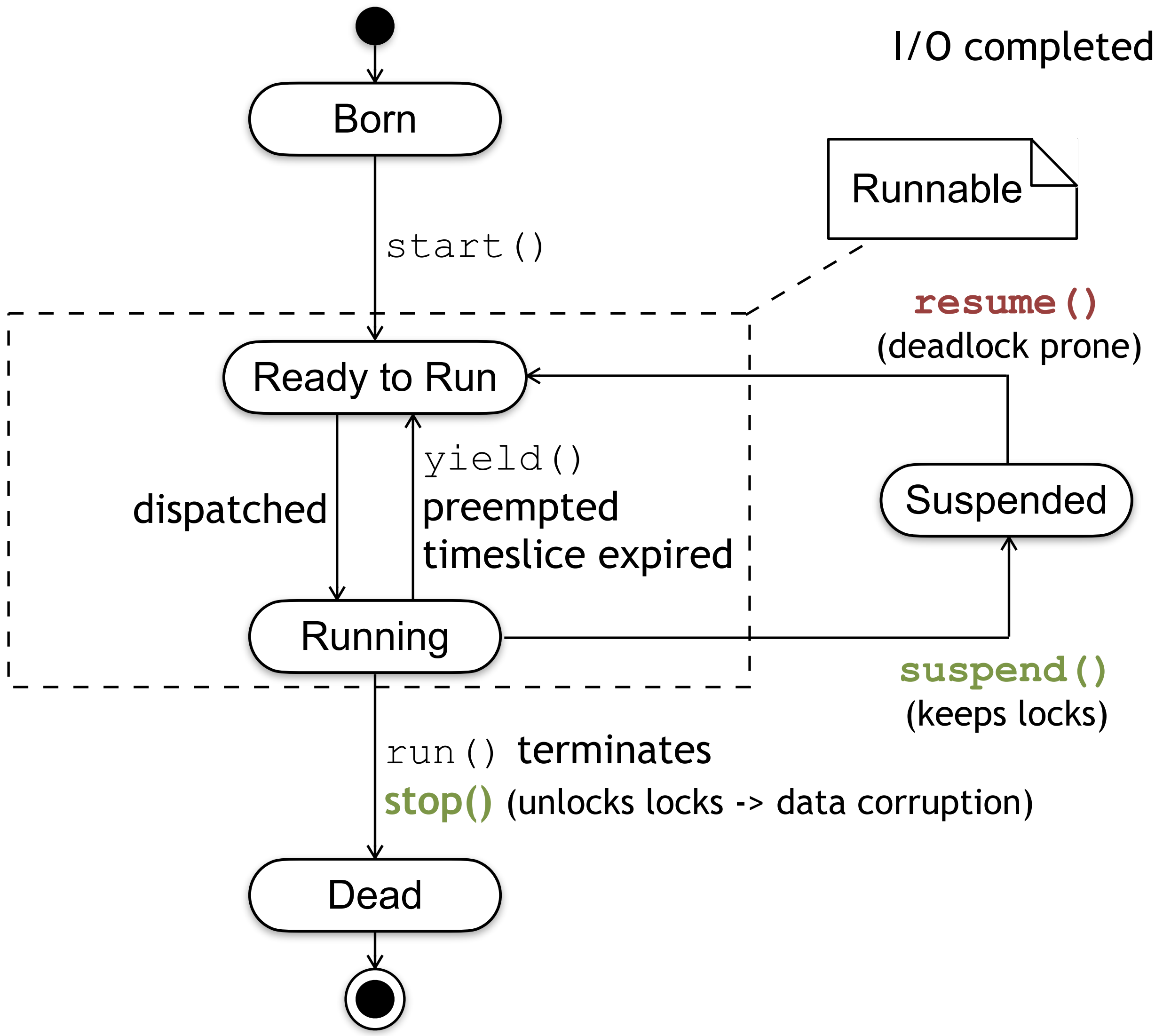
Deprecated `Thread` Methods

- We will look at the effect of the following methods on a thread's life cycle: `stop()`, `suspend()`, and `resume()`, even though they have been *deprecated* and should never be invoked
  - `stop()` - forces the thread to stop, regardless of what it is doing
    - this method is dangerous
    - all locks held by the thread are unlocked, and the objects protected by those locks may be in an inconsistent state, but can now be accessed by other threads

Deprecated `Thread` Methods

- `suspend()` - suspends the thread
- `resume()` - resumes the suspended thread
- these two methods are deprecated because `suspend()` is deadlock prone
  - the suspended thread holds onto any locks that it owns when it is suspended
  - consider what would happen if the thread that invokes `resume()` to resume the suspended thread first tried to acquire the lock held by the suspended thread

Yield versus Suspend/Resume versus Stop



Thread Scheduling

- We're now ready to look at how the Java runtime system schedules thread execution; i.e., shares the processors between multiple threads that are in the Runnable state
- Before we look at thread scheduling in detail, we need to introduce the concept of *thread priorities*

Thread Priorities

- Every thread has a *priority*
- `Thread` defines:
  - `Thread.MAX_PRIORITY` (currently 10)
  - `Thread.NORM_PRIORITY` (currently 5)
  - `Thread.MIN_PRIORITY` (currently 1)
- A thread's priority must be an integer in the range `Thread.MIN_PRIORITY` to `Thread.MAX_PRIORITY`, inclusive

Thread Priorities

- When a thread creates a new `Thread` object, the new thread's priority is set equal to the priority of the thread that created it
- Thread priorities may be changed at run-time
  - `setPriority()` sets a thread's priority to a specified value
  - `getPriority()` returns the thread's priority
- The currently executing threads are (normally) the threads in the Runnable state with the highest priorities. Later, we'll look at what happens when two or more threads having the same priority make this impossible.

Sharing the Processor Between Threads

- When the JVM schedules a thread for execution, the thread executes until:
  - it voluntarily relinquishes the processor; e.g.,
    - `Thread.yield()` causes the currently executing thread to relinquish the processor
    - `Thread.sleep()` causes the currently executing thread to temporarily cease execution for a specified amount of time
    - `Object.wait()` cases the currently executing thread to wait for a notification
      - cont'd on next slide

Sharing the Processor Between Threads

- it terminates
  - by running off the end of its `run()` method
  - by throwing an exception that propagates beyond its `run()` method
- it is preempted by a higher-priority thread becoming ready to run
- if timeslicing is supported, its timeslice expires

- This is called *preemptive, priority-based scheduling*
  - it relies on the preemption of lower-priority threads to ensure that (high-priority) threads that need processor time urgently get it

Scheduling Equal Priority Threads

- For simplicity, let’s assume a uniprocessor:
- Suppose we have two ready to run Java threads, *t1* and *t2*, with equal priority, and no higher priority thread is ready to run
- Suppose *t1* is scheduled for execution
- What does the JVM do in this case?
- With some versions of the JVM, *t1* will run until it relinquishes the processor, terminates, or is preempted by a higher priority thread
  - *t2* will not be scheduled for execution until *t1* (and any higher priority threads) are not eligible to run

Timeslicing

- Other versions of the JVM support *timeslicing*
  - thread scheduler schedules equal-priority, ready to run threads with the highest priority in a round-robin fashion, allowing each thread to run for a fixed amount of time
- The JVM will *timeslice* between *t1* and *t2*, but will not schedule any lower priority ready to run threads until *t1* and *t2* are no longer eligible to run
- Either approach is permitted by the Java specification

Thread Priorities: Example

- Here is a version of the factorial/fibonacci thread example that uses priorities
  - `factorialThread` has priority `Thread.NORM_PRIORITY - 1`
  - `secondFactorialThread` has priority `Thread.NORM_PRIORITY - 3`
    - notice that these two threads execute the same code but have different priorities
  - `fibonacciThread` has priority `Thread.NORM_PRIORITY - 2`

Thread Priorities: Example

- None of the threads relinquish the processor (the calls to `sleep()` have been removed)
- Assuming we have a uniprocessor:
  - `factorialThread` has higher priority than the other two threads, so it runs to completion before the other two are executed
  - `fibonacciThread` has higher priority than `secondFactorialThread`, so it runs to completion before `secondFactorialThread` is executed
- Here is the code...

Thread Priorities: Example

```
class ThreadPriorityExample
{
    public static void main(String[] args) {
        Thread factorialThread =
            new Thread(new Factorial(),
                       "Factorial calculator");
        factorialThread.setPriority(
            Thread.NORM_PRIORITY - 1);
        System.out.print("Created: " +
                         factorialThread + '\n');
```

Thread Priorities: Example

```
Thread secondFactorialThread =
    new Thread( new Factorial(),
          "Second factorial calculator");
secondFactorialThread.setPriority(
    Thread.NORM_PRIORITY - 3);
System.out.print("Created: " +
           secondFactorialThread + '\n');

Thread fibonacciThread =
    new Thread(new Fibonacci(),
               "Fibonacci calculator");
fibonacciThread.setPriority(
    Thread.NORM_PRIORITY - 2);
System.out.print("Created: " +
                 fibonacciThread + '\n');
```

Thread Priorities: Example

```
        System.out.print("Starting threads\n");
        factorialThread.start();
        secondFactorialThread.start();
        fibonacciThread.start();
    }
}
```

Thread Priorities: Example

```
/**
 * This thread calculates 0! through 10!, where
 *  0! = 1
 *  n! = n * (n-1)!, n > 0
 */
class Factorial implements Runnable
{
    public void run() {
        // 0! = 1
        long factorial = 1;
        System.out.print(Thread.currentThread() +
                         ": 0! = " + factorial +
                         '\n');
        for (int n = 1; n <= 10; n++) {
```

Thread Priorities: Example

```
                // n! = n * (n-1)!
                factorial = n * factorial;
                System.out.print(
                            Thread.currentThread() +
                            ": " + n + "! = " +
                            factorial + '\n');
            }
            System.out.print(Thread.currentThread() +
                             " finished\n");
        }
    }
```

Thread Priorities: Example

```
/**
 * This thread calculates fib(1) through fib(10),
 * where
 * fib(1) = 1
 * fib(2) = 1
 * fib(n) = fib(n-1) + fib(n-2), n >2
 */
class Fibonacci implements Runnable
{
    public void run() {
        // fib(1) = 1
        long firstFib = 1;
        System.out.print(Thread.currentThread() +
                         ": fib(1) = " +
                         firstFib + '\n');
```

Thread Priorities: Example

```
        // fib(2) = 1
        long secondFib = 1;
        System.out.print(Thread.currentThread() +
                                 ": fib(2) = " +
                                 secondFib + '\n');
        for (int n = 3; n <= 10; n++) {
             // fib(n) = fib(n-1) + fib(n-2)
             long fibN = firstFib + secondFib;
             System.out.print(Thread.currentThread()
                    + ": fib(" + n + ") = "
                    + fibN + '\n');
             secondFib = firstFib;
             firstFib = fibN;
        }
        System.out.print(Thread.currentThread() +
                                 " finished\n");
    }
}
```

Relinquishing the Processor & Thread Preemption

- The next example has one higher priority factorial thread and one lower priority fibonacci thread
- Before calculating n! for the current value of n, the factorial thread relinquishes the processor by sleeping for 2 ms
- Again, assuming a uniprocessor, while the factorial thread sleeps, the lower priority fibonacci thread executes
- When the factorial thread wakes up, it preempts the fibonacci thread
- Here is the code (Fibonacci is unchanged from slides 32-33)

Relinquishing the Processor & Thread Preemption

```
class ThreadPreemptionExample
{
    public static void main(String[] args) {
        Thread factorialThread =
            new Thread(new Factorial(),
                       "Factorial calculator");
        factorialThread.setPriority(
            Thread.NORM_PRIORITY - 1);
        System.out.print("Created: " +
                         factorialThread + '\n');
```

Relinquishing the Processor & Thread Preemption

```
        Thread fibonacciThread =
            new Thread(new Fibonacci(),
                       "Fibonacci calculator");
        fibonacciThread.setPriority(
            Thread.NORM_PRIORITY - 2);
        System.out.print("Created: " +
                         fibonacciThread + '\n');

        System.out.print("Starting threads\n");
        factorialThread.start();
        fibonacciThread.start();
    }
}
```

Relinquishing the Processor & Thread Preemption

```
/**
 * This thread calculates 0! through 10!, where
 *  0! = 1
 *  n! = n * (n-1)!, n > 0
 */
class Factorial implements Runnable
{
    public void run() {
        // 0! = 1
        long factorial = 1;
        System.out.print(Thread.currentThread() +
                         ": 0! = " + factorial +
                         '\n');
        for (int n = 1; n <= 10; n++) {
```

Relinquishing the Processor & Thread Preemption

```
          // Sleep for 2 ms before calculating n!
           try {
               Thread.sleep(2);
           } catch (InterruptedException e) {}

           // n! = n * (n-1)!
           factorial = n * factorial;
           System.out.print(
                        Thread.currentThread() +
                        ": " + n + "! = " +
                        factorial + '\n');
        }
        System.out.print(Thread.currentThread() +
                         " finished\n");
    }
}
```

Unbounded Priority Inversion

- Unbounded priority inversion is a problem that can occur in concurrent systems that provide some form of lock to protect critical sections, and use priority-based preemptive scheduling
  - it is not unique to Java

Bounded Priority Inversion: Overview

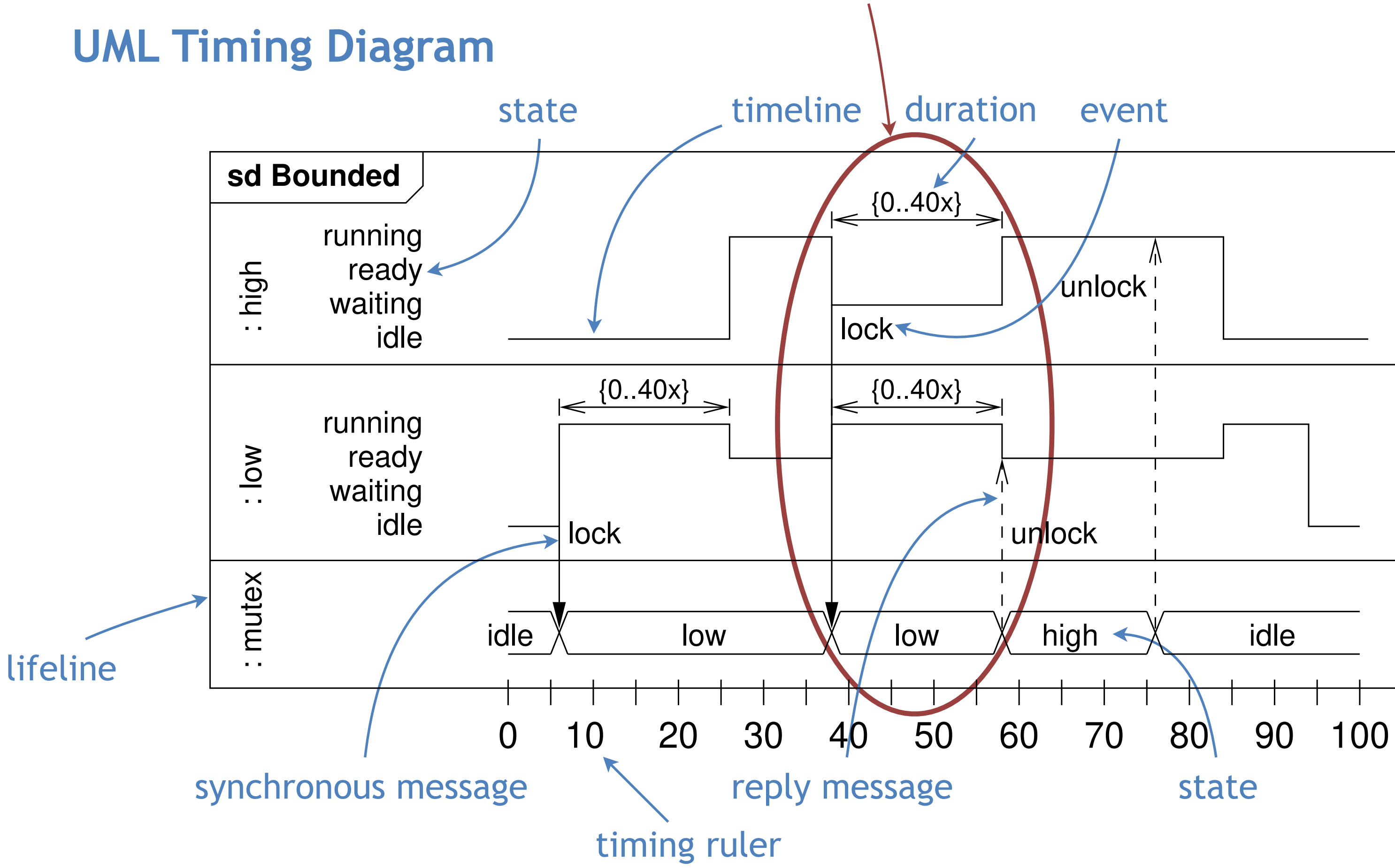
- Suppose we have a low priority thread that has obtained a lock and entered a critical section (e.g., in Java, is executing a synchronized method)
- The lower priority thread is preempted by a higher priority thread
- The high priority thread wants to enter the critical section, so it attempts to acquire the lock and blocks, and the low priority thread is again dispatched
- The high priority thread remains blocked until the low priority thread leaves the critical section and releases the lock

Bounded Priority Inversion: Overview

- In effect, the priority of the high priority thread has been temporarily reduced to that of the low priority thread
- This is called *bounded priority inversion*
  - bounded, because we can determine the amount of time that the low priority thread will block the high priority thread from entering the critical region

Bounded Priority Inversion

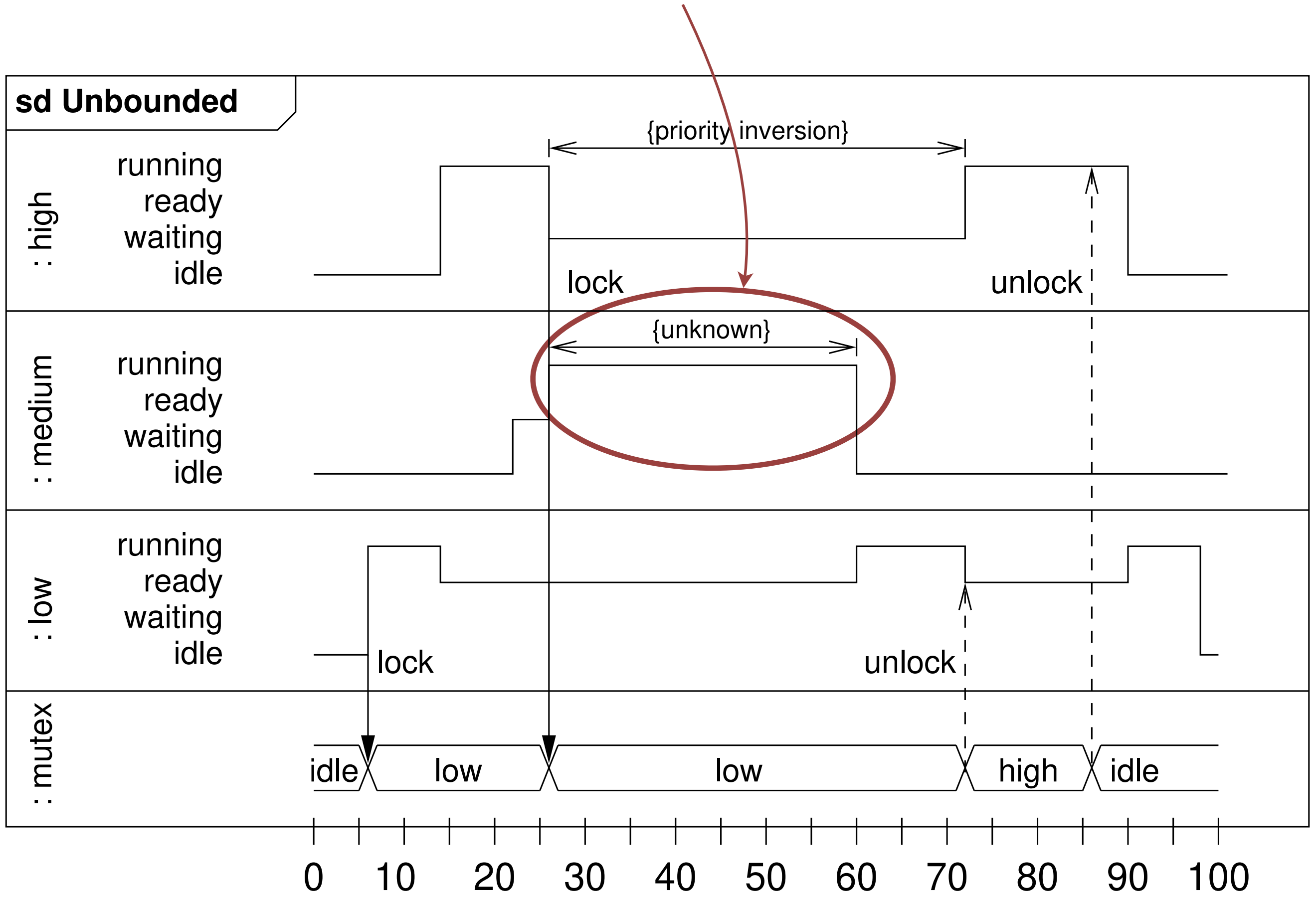
UML Timing Diagram



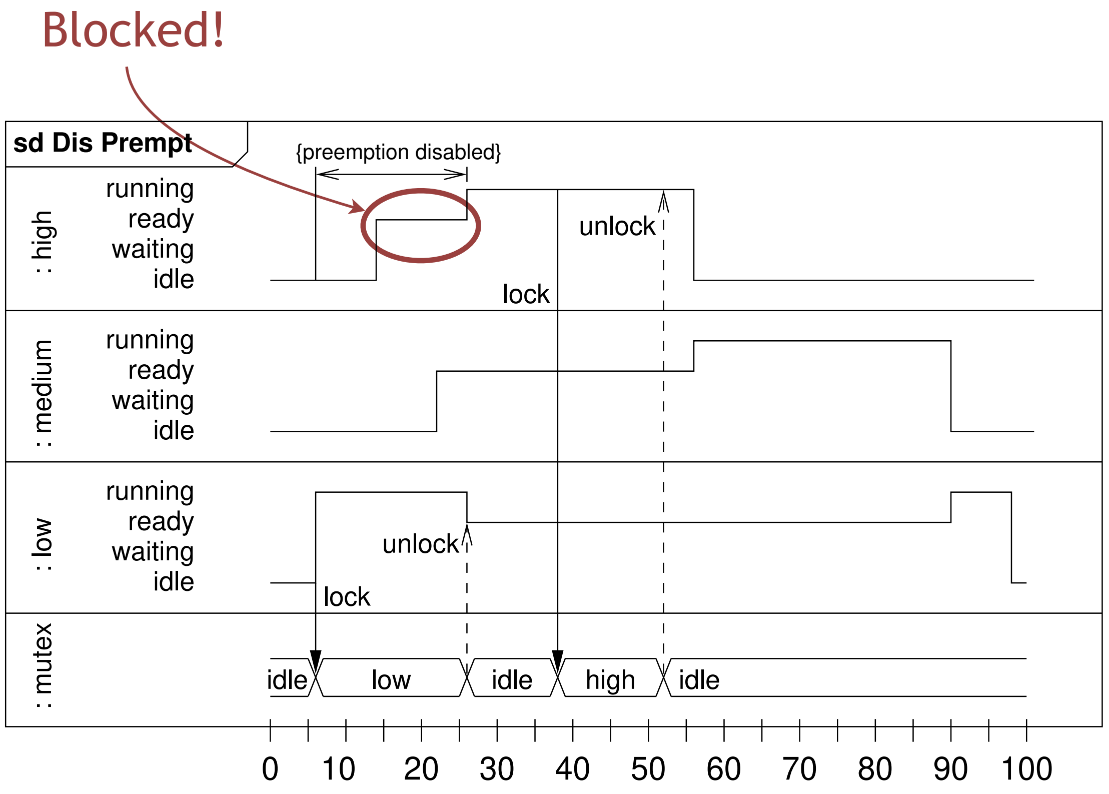
Unbounded Priority Inversion: Overview

- Now suppose that while the low priority thread is in the critical section, it is preempted by one or more medium priority threads
- The lower priority thread does not execute for an indeterminate amount of time
- So, the higher priority thread that is waiting to enter the critical section also waits for an indeterminate amount of time, while the medium priority threads execute
- This is *unbounded* priority inversion

Unbounded Priority Inversion



Disable Preemption



Solving the Priority Inversion Problem x

- The thread scheduler could disable preemption of the lower priority thread while it is in the critical section
  - this is not the best approach, as it disables the execution of other threads (with high priority) that will not attempt to enter the critical section
- The thread scheduler could increase the priority of the lower priority thread while it is in the critical section
  - two ways to do this are commonly used:
    - priority inheritance protocol
    - priority ceiling (priority protect) protocol

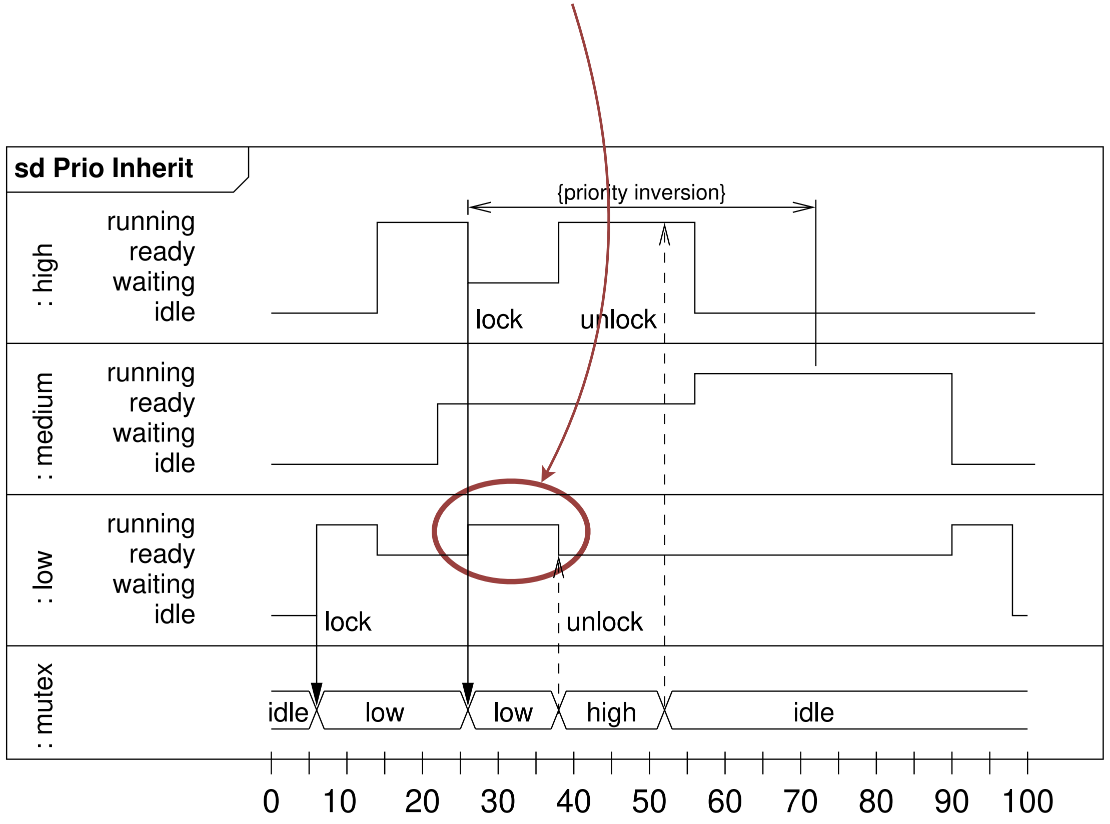
Priority Inheritance Protocol

- The lower priority thread starts executing the critical section at its assigned (low) priority
- When a higher priority thread attempts to enter the critical section (e.g., invokes a synchronized method and blocks attempting to acquire object's lock), the low priority thread's priority is raised to be the same as that of the higher priority thread
- So, the lower priority thread cannot be preempted by a medium priority thread, and the time that the higher priority thread is blocked is bounded

Priority Inheritance Protocol

- The low priority thread's priority is restored to its original value when it leaves the critical section
- *Summary*: the priority of a thread holding a lock is the higher of
  - its assigned priority
  - the priority of the highest priority thread that is blocked, waiting to acquire the lock
- Many implementations of the JVM use priority inheritance to prevent unbounded priority inversion of threads waiting to execute synchronized methods

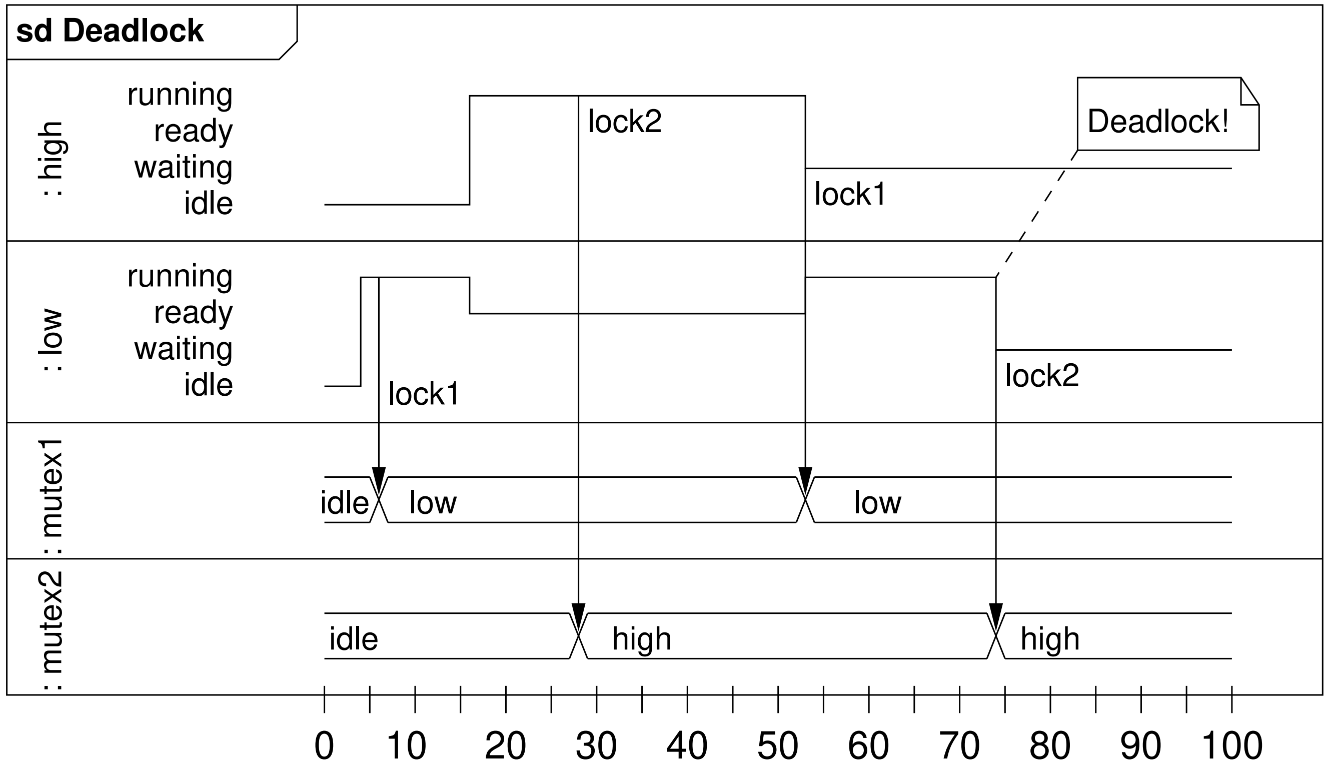
Priority Inheritance



Deadlocks

- Deadlock can arise in a system with multiple threads and multiple shared resources when one thread requests one resource and then another, while another thread requests the same to resources but in opposite order (see next slide).
- Solutions:
  - Deadlock detection
    - watchdog timers — reset the system when the timer goes off.
    - locate cycles in the request graph at runtime.
  - Deadlock prevention — always make requests to shared resources in the same order.
    - Requires discipline. 🤣
  - Deadlock avoidance — somehow stop deadlocks from occurring regardless of the ordering of requests.
    - But how??? 🤔

Deadlock from Shared Resources



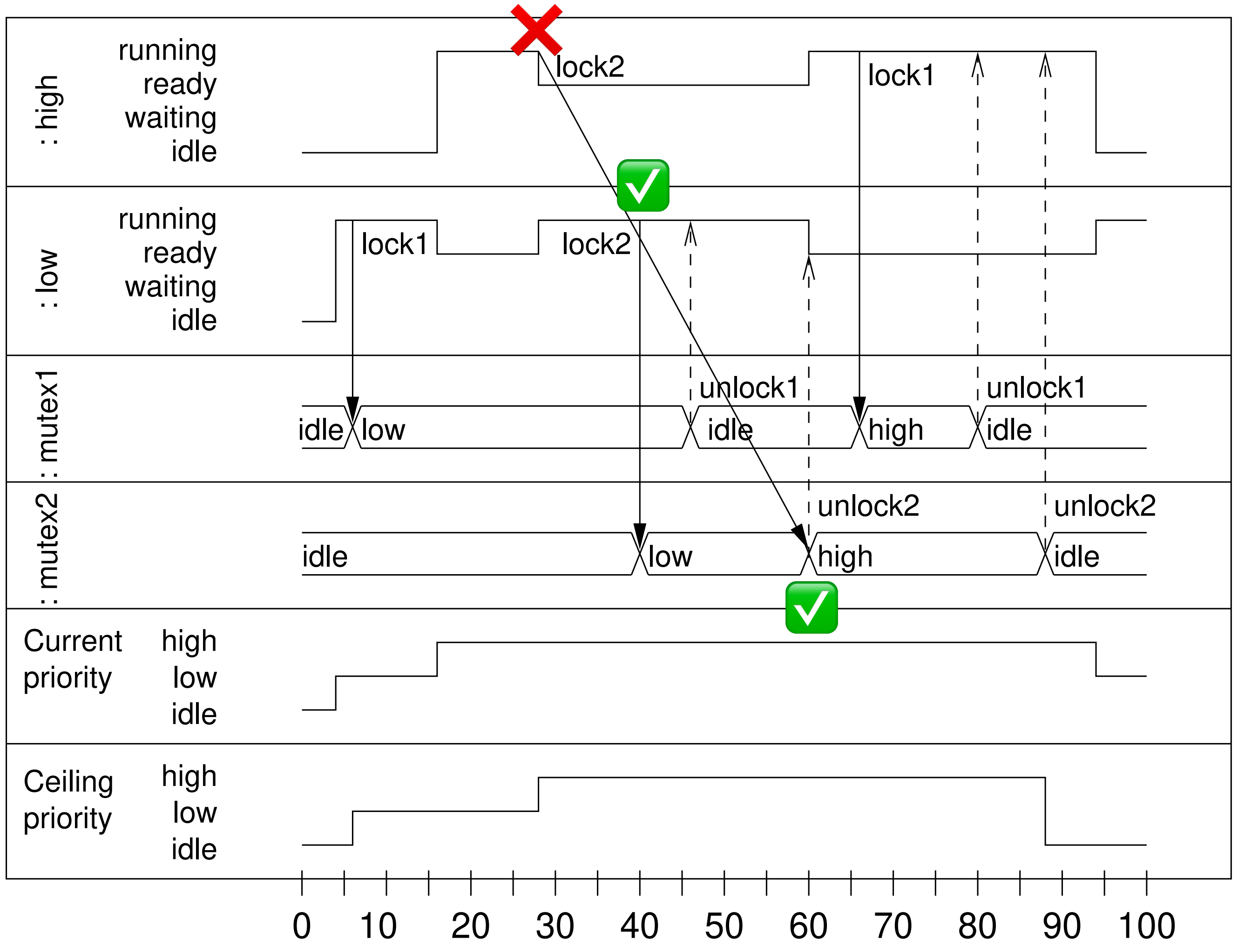
Priority Ceiling Protocol

- Locks used to protect critical sections are allocated a priority when the lock is created, called the *ceiling priority*, that is at least as high as the priority of the highest priority thread that can acquire the lock.
- Only threads with a priority <u>higher</u> than the global ceiling priority can preempt the lower priority thread.
- When a thread tries to acquire a lock for a shared resource, the global ceiling priority is raised to the thread's priority.
    - The high priority thread is blocked without acquiring the lock even though the lock may be free.
    - The priority of the thread holding the shared resource is raised (like priority inheritance).

Priority Ceiling Protocol

- As long as the low priority thread doesn't relinquish the processor, no threads of any priority that need to acquire the lock will execute while the low priority thread is in the critical section.
  - **Deadlock free!** 😀
- Also, the low priority thread can't be preempted by medium priority threads (threads with priority < the lock's ceiling priority) that aren't interested in the critical section.
- High priority threads can finish faster when compared to priority ceiling. 😀
- The priority ceiling protocol is not currently supported by Java.

Priority Ceiling Protocol



Comparison of the Protocols

- Priority inheritance is managed transparently by the system.
- Priority ceiling requires the programmer to configure the priority ceilings of the locks.
- Priority inheritance does not associate a priority with a critical section (priority promotion of the low priority thread executing the critical section occurs only when a higher priority thread attempts to acquire the lock).
- Priority ceiling associates a priority with all locks, and therefore with the critical sections that they protect.

JVM Implementations - Green Threads

- The JVM runs on top of an operating system that is unaware of Java threads
  – the JVM is completely responsible for thread management; e.g., for every thread, the JVM maintains a stack, program counter, and other bookkeeping info
  – context switching between threads is handled by the JVM
- This approach is equivalent to using a *user-level* threading package with C - no calls to the underlying operating system are required to manage threads

JVM Implementations - Green Threads

- Early implementations of the JVM on many Unix platforms used the green-thread model
- Green-thread implementations support priority-based, preemptive thread scheduling, but most implementations do not support time slicing
- The *reference implementation* of the green-thread JVM uses priority inheritance

JVM Implementations - Native Threads

- Java threads are implemented using the threads supported by the operating system that hosts the JVM
  - thread scheduling is handled by the operating system
  - differences between operating systems lead to subtle differences in Java thread scheduling

JVM Implementations - Windows Native Threads

- For most JVM implementations for 32-bit Windows operating systems, there is a one-to-one mapping between Java threads and Win-32 threads
  - these threads are scheduled by the operating system
- Priority-based, preemptive scheduling is supported
  - 10 Java thread priorities are mapped onto 6 Win-32 priorities
- Priority inheritance is provided
- Equal priority threads timeslice
- Thread priorities are temporarily increased (quietly) to reduce starvation

Other JVM Implementations

- Solaris (Sun) implementation uses Solaris native threads
  - native threads use both user-level threads and system-level threads (LWPs)
  - scheduling is complex and beyond the scope of this course
- Linux/Mac OSX use native threads.

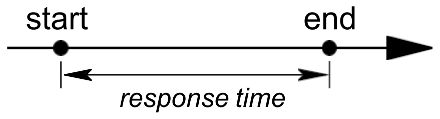
Other JVM Implementations

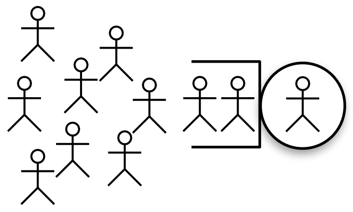
- Embedded systems - Java threads normally mapped to the native system threads/tasks/processes supported by the real-time operating system (RTOS) hosting the JVM
  - Java thread scheduling = RTOS thread scheduling
- Bottom line: underlying thread model is dependent on the underlying JVM implementation and support provided by the host OS.
  - Operating system support is required to achieve concurrency with multiple processors or in the presence of blocking system calls.

Summary

SYSC 3303 Real-Time Concurrent Systems

Measurement

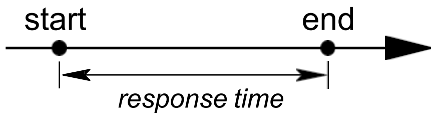




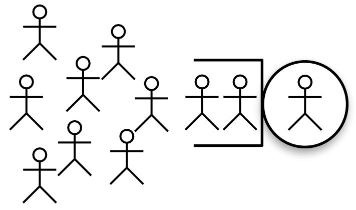
What is Performance?

Measures

- Response time:
  - from start event (such as sending a request)
  - to end event (such as getting an output)
- Throughput capacity
  - max allowed frequency of responses
  - may overlap in time — (many web server clients)
- Utilization of a resource
  - percent time busy



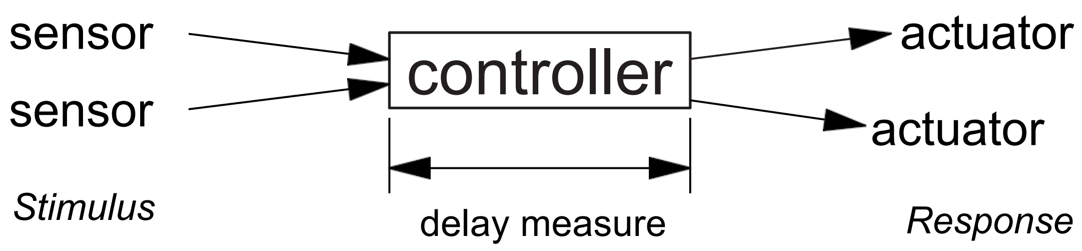
(a) Response Time



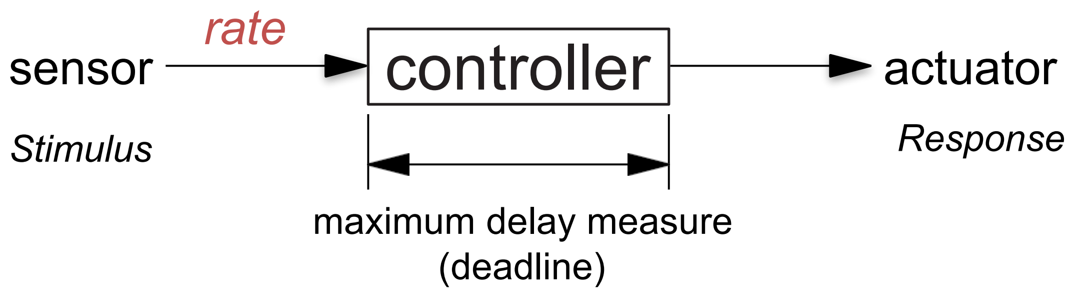
(b) Throughput

Embedded systems

- Real-time controllers and sensors

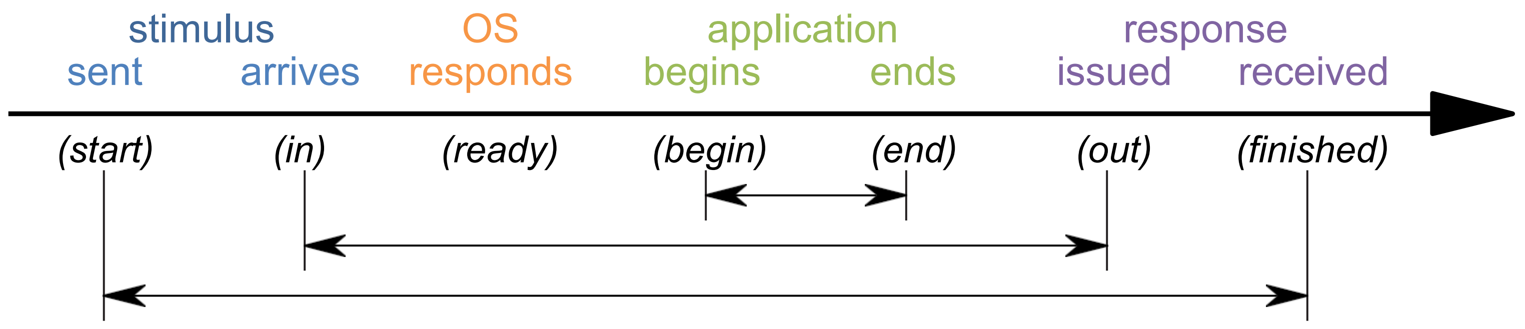


- Often an array of single-input, single-output controllers



Measurement by Events

- Measurement is based on significant events, and subsystem states
- Events: stimulus, OS, application, response



- Event time = “wall clock time”
  - measure with a system timer (or a stop watch for start-to-end)
- Intervals include execution, transfer latency and waiting (queueing).
- Event log: `[time, entity, event code, ...additional data]`
- Counters for events (get throughput by division of count by duration)

Measurement of Entity State

1. Typical state is *"device busy"*
   - record a log of events that "go busy" and "go idle".
   - find total time in each state.
   - compute utilization $U = \%\mathrm{busy}$ from times.
2. Sample the state.
   - have a daemon on a timer
   - periodically sample the state
   - count how often busy and how often idle.

What is Going on in the Computer?

- Seen at different levels
  - application: operations requested by the user.
  - middleware: calls requested by the app.
  - OS services: calls from above (e.g. file read)
  - OS scheduler: run processes and threads
  - processor: execute instructions
- Additional devices such as NIC and disk driver have their own services and lower layers

Processor activity and measurements

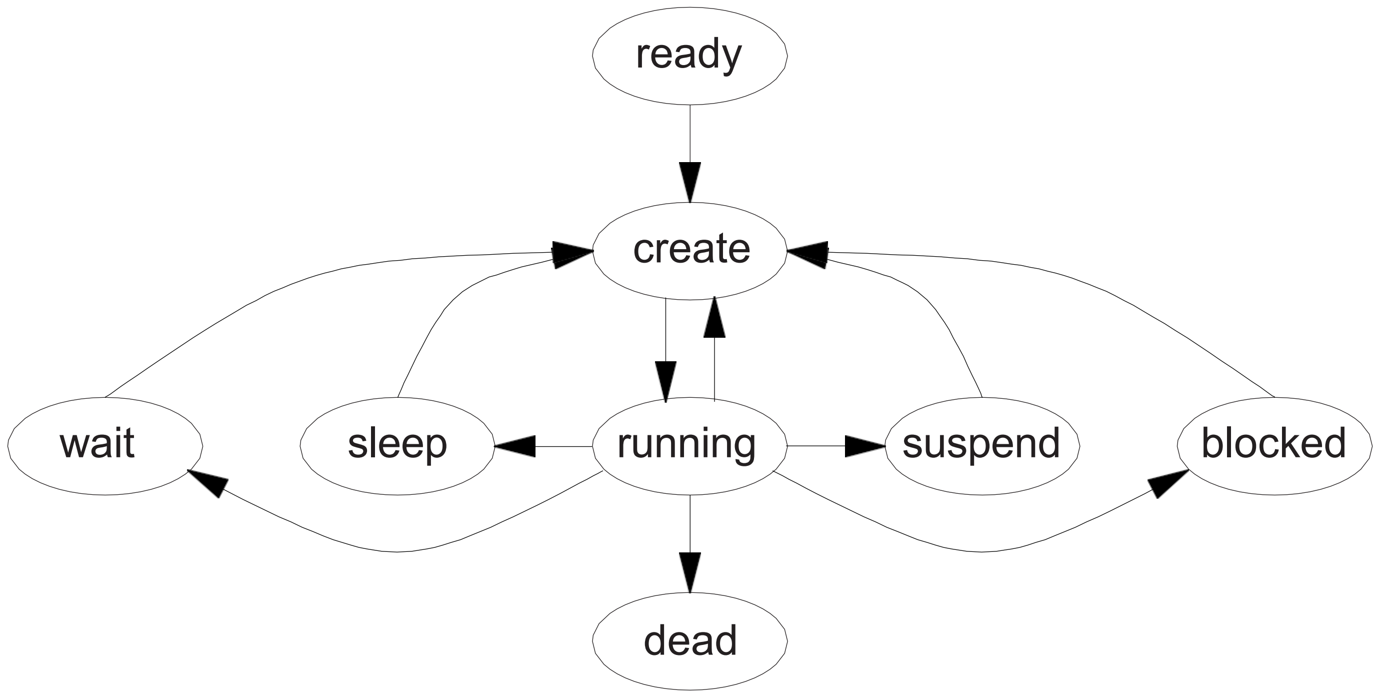
- Fetch, execute.
- Log the % busy, time spent blocked on memory or the bus, cache misses (hardware counters, OS support to read them)
- No application context, so we can’t tell about the important delays at the application level.
    - if a particular application function has important cache misses, we can see the misses but not which function is to blame.

OS Scheduler Level

- sees threads which it decides to run, one at a time for each processor it controls
  - *user threads* run application and middleware
    - a process is a collection of threads and an address space
    - OS thread runs a user-defined thread as in Java
    - middleware is system software that runs in user space, such as JVM, J2EE, RMI or CORBA
  - *system threads* run the OS services and the scheduler itself (may be hundreds of these)
- sees the thread but not its logical significance. Thus a particular thread may run slowly, but OS does not identify it by symbolic name.
- sees system-level aggregate behaviour of the population of threads

OS scheduler-level view

- OS sees tokens representing threads, that move through states and may belong to queues awaiting services (ready, waiting). For example (Java thread states)



OS scheduler-level measurements

- The OS scheduler implements the queue of threads for the processor, so higher levels see the CPU as the service.
- OS accounting keeps track of time and resource use by processes and threads, as part of scheduling them
  - UNIX: process CPU time as *system time* and *user time*, from initiation

OS service level

- Disk i/o’s, memory usage, network operations (packets sent and received).
    - granularity: usage by system, process, thread, class, method
    - disk and memory by thread
    - network for system as a whole
- Exotic: OS tracing tools record event times by thread

Middleware and application measurements

- The application can read the clock with probes embedded at significant events (e.g. start and end of a method or an operation)
  - may store an event trace.
  - intervals by subtraction.
  - instrumentation cost.
  - clock granularity problem.
- For Java measurement tools, see:
  javarevisited.blogspot.ca/2014/07/top-5-java-performance-tuning-books.html
  - this site emphasizes that measurements must be done in some well controlled and repeatable situation.

Application measurement by a sampling profiler

- Tool to answer “where in the code is the CPU spending time?”
- For a single *process* in C or C++
  - insensitive to threads unless their source code is distinct
- Periodically samples the instruction counter, uses a symbol table to decode to a method or procedure.
  - % of counts gives % of time spent in a method
- Also: samples the stack to find the calling methods in order, to give the context of the call
  - give % for all the nodes in a tree of calls, breaking the % for the higher methods into parts for the lower ones
- Java has its own profiler in the JVM.

Middleware view

- Middleware instrumentation does not require application developer effort to install
- Some middleware (CORBA, RMI) sees application calls for services from other applications, and knows the application identities
    - but not the user-level operation that is underway.
- JVM instrumentation

Measurement Techniques

Three major steps:

1. Specify measurements (i.e., what do you want to measure.)
2. Instrument and gather data.
   - attach monitors.
   - run benchmark.
3. Analyze and transform data.

Tools

- Objective: measure behaviour of system without perturbing it.
- Two *modes* for collecting data:
    1. event trace.
    2. sampling.
- Three types of monitors:
    1. Hardware
    2. Software
    3. Hybrid.

Hardware Monitors

- External device such as oscilloscope or logic analyzer connected to system under test.
  - (+) External to system, so doesn’t perturb results.
  - (+) Portable.
  - (−) Hard to use.
  - (−) Difficult to map event to cause.

Software Monitors

(+) Can record anything of interest.

(−) Can interfere with system significantly. Two types:

1. Accounting system (e.g. sar on Unix):
   - Used for billing CPU time and I/O usage.
   - May not collect right data.
   - Problem with task with many “users” such as Apache (or a database).
2. Program analyzers:
   - .e.g. quantify, valgrind, lttng, oprofile, …

Hybrid Monitors

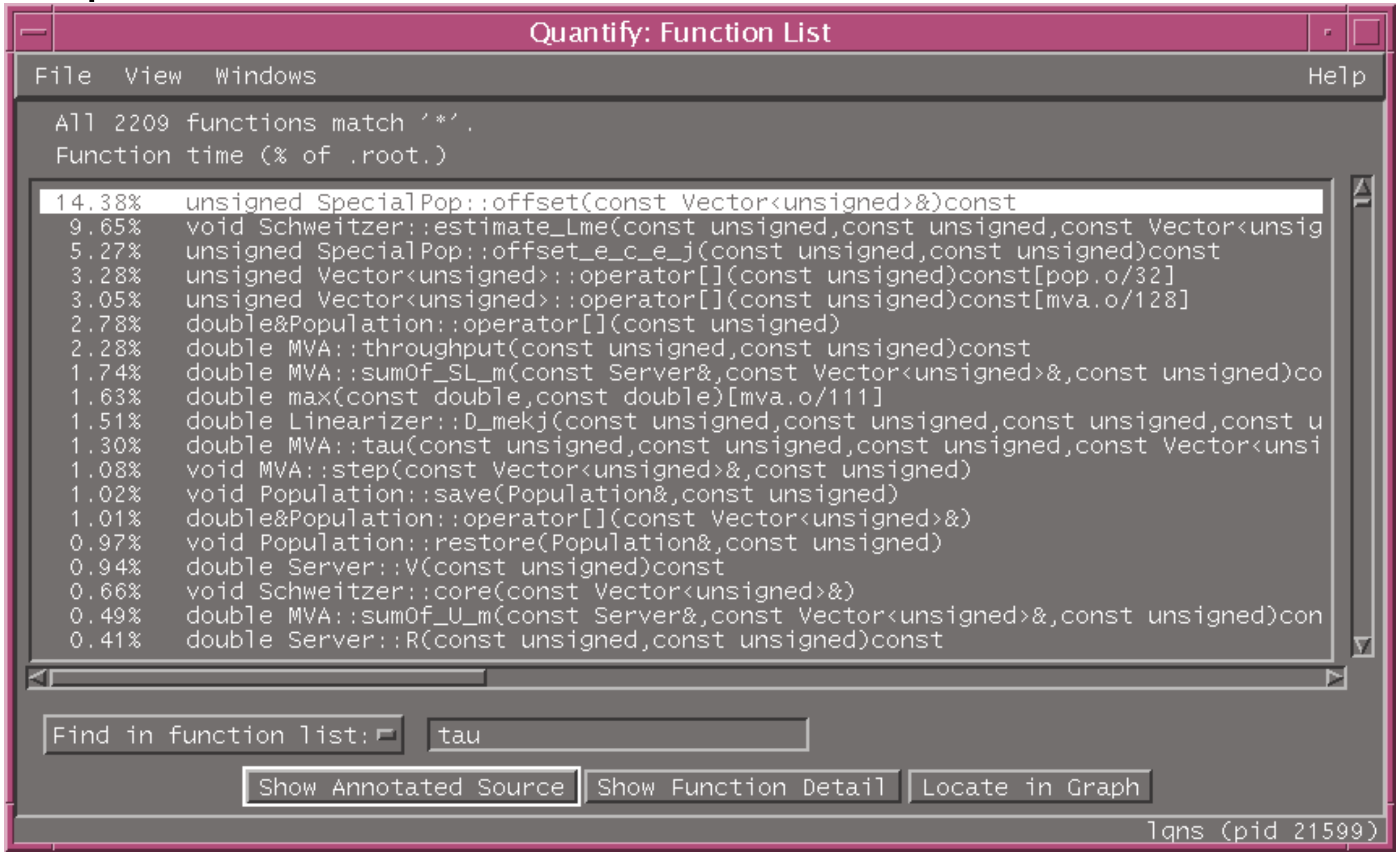
- Event signals embedded in the software are processed by an external device.
- Often custom-built for a specific system.

Monitoring Modes

1. Event Trace Mode.
   - Collects info on specific state changes.
   - Probes are inserted into the operating system.
   - Overhead of 5% is o.k. Sticking probes in interrupt handlers is probably not a good idea.
2. Sampling Mode.
   - Polls state of system periodically.
   - accuracy is proportional to overhead, e.g., for high accuracy, lots of samples are required, so poll frequently.

Rational (IBM) Quantify ($$$)

List of functions ordered by time taken (not too useful for response time.

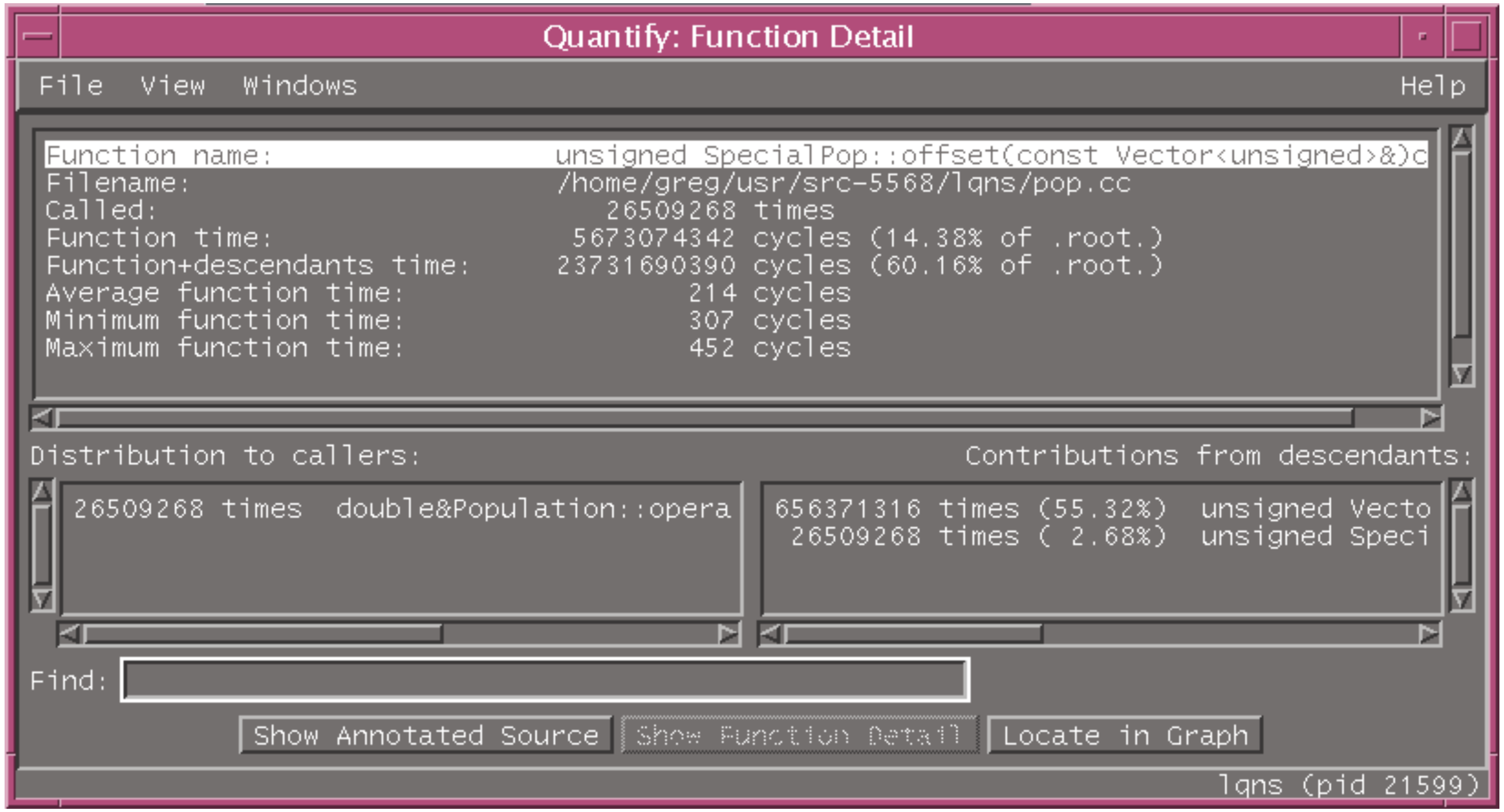


```
All 2209 functions match '*'.
Function time (% of .root.)

14.38%   unsigned SpecialPop::offset(const Vector<unsigned>&)const
 9.65%   void Schweitzer::estimate_Lme(const unsigned,const unsigned,const Vector<unsig
 5.27%   unsigned SpecialPop::offset_e_c_e_j(const unsigned,const unsigned)const
 3.28%   unsigned Vector<unsigned>::operator[](const unsigned)const[pop.o/32]
 3.05%   unsigned Vector<unsigned>::operator[](const unsigned)const[mva.o/128]
 2.78%   double&Population::operator[](const unsigned)
 2.28%   double MVA::throughput(const unsigned,const unsigned)const
 1.74%   double MVA::sumOf_SL_m(const Server&,const Vector<unsigned>&,const unsigned)co
 1.63%   double max(const double,const double)[mva.o/111]
 1.51%   double Linearizer::D_mekj(const unsigned,const unsigned,const unsigned,const u
 1.30%   double MVA::tau(const unsigned,const unsigned,const unsigned,const Vector<unsi
 1.08%   void MVA::step(const Vector<unsigned>&,const unsigned)
 1.02%   void Population::save(Population&,const unsigned)
 1.01%   double&Population::operator[](const Vector<unsigned>&)
 0.97%   void Population::restore(Population&,const unsigned)
 0.94%   double Server::V(const unsigned)const
 0.66%   void Schweitzer::core(const Vector<unsigned>&)
 0.49%   double MVA::sumOf_U_m(const Server&,const Vector<unsigned>&,const unsigned)con
 0.41%   double Server::R(const unsigned,const unsigned)const
```

Rational (IBM) Quantify

If the task is structured such that each event calls a function then picking off the response time is easy.



```
Function name:                 unsigned SpecialPop::offset(const Vector<unsigned>&)c
Filename:                      /home/greg/usr/src-5568/lqns/pop.cc
Called:                          26509268 times
Function time:                  5673074342 cycles (14.38% of .root.)
Function+descendants time:     23731690390 cycles (60.16% of .root.)
Average function time:                 214 cycles
Minimum function time:                 307 cycles
Maximum function time:                 452 cycles
```

```
Distribution to callers:                        Contributions from descendants:
26509268 times  double&Population::opera        656371316 times (55.32%)  unsigned Vecto
                                                 26509268 times ( 2.68%)  unsigned Speci
```

GPROF

- Assignment 2. Converted to C++
  - the resolution of the clock is too course 😟, so all we get is the number of calls.

```
 %   cumulative   self              self     total
time   seconds   seconds    calls  Ts/call  Ts/call  name
 0.00      0.00     0.00      907     0.00     0.00  std::chrono::duration<long, std::ratio<1l, 1l> >::count()
 0.00      0.00     0.00      803     0.00     0.00  __gthread_active_p()
 0.00      0.00     0.00      750     0.00     0.00  __gnu_cxx::__enable_if<std::__is_char<char>::__value,
 0.00      0.00     0.00      700     0.00     0.00  std::chrono::duration<long, std::ratio<1l, 1000l> >::count()
 0.00      0.00     0.00      600     0.00     0.00  std::remove_reference<unsigned int&>::type&&
 0.00      0.00     0.00      403     0.00     0.00  __gthread_mutex_lock(pthread_mutex_t*)
 0.00      0.00     0.00      403     0.00     0.00  std::chrono::duration<long, std::ratio<1l, 1l> >
 0.00      0.00     0.00      402     0.00     0.00  std::mutex::lock()
```

GPROF

- If we could get timing, we could find the execution time from the call graph output.
- Java Profilers should be able to do the same thing.

```
                0.00    0.00      3/3            void std::__invoke_impl<void, Chef>(std::… Chef&&) [131]
[113]    0.0    0.00    0.00      3          Chef::operator()() [113]
                0.00    0.00    203/203          Chef::get_chef_type[abi:cxx11]() const [21]
                0.00    0.00    103/103          Table::get(std::__cxx11::basic_string<char,…> > const&) [40]
                0.00    0.00    100/402          std::mutex::lock() [15]
                0.00    0.00    100/400          std::mutex::unlock() [18]
                0.00    0.00    100/100          std::chrono::duration<…> >::duration<int,…>(int const&) [78]
                0.00    0.00    100/200          std::pair<std::__cxx11::basic_string<char,…) [34]
                0.00    0.00    100/100          void std::this_thread::sleep_for<long,…) [58]
```

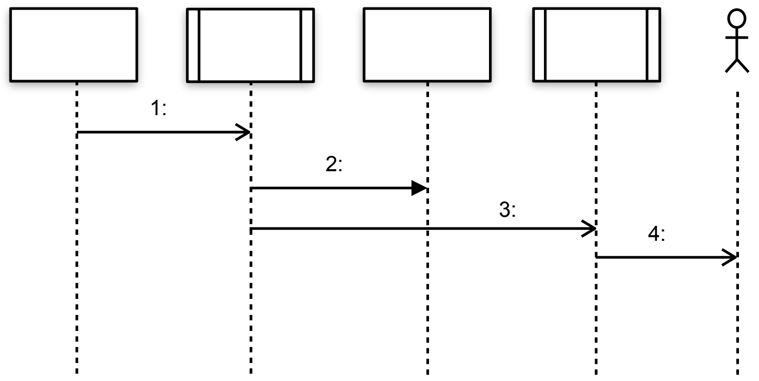
- There are lots of other profilers out there (valgrind, oprofile, quantify, …), but I haven’t had time to play with them.

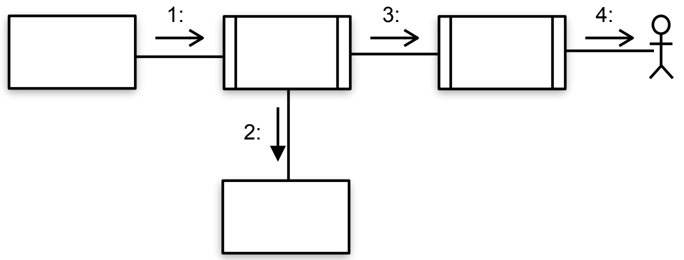
Summary

- Why do we measure?
  - Find and fix hotspots (not so much in this course)
  - Collect data for analysis (later in this course)
- Important measures:
  1. Response time (time to complete something).
  2. Utilization (percent busy).
- Ways to measure:
  1. Sampling (profiling).
  2. Capturing events (tracing).
  - Each has their strengths and weaknesses.

SYSC 3303 Real-Time Concurrent Systems

Dynamic Modelling





You are here…

1. System structural modelling.
2. Requirements modelling.
3. Analysis modelling:
    - I. Static modelling.
    - II. Dynamic state machine modelling.
    - III. Object structuring.
    - IV. Dynamic interaction modelling.
4. Design modelling.
5. Incremental software construction (your project!)
6. Incremental software testing (your project!)
7. System testing.

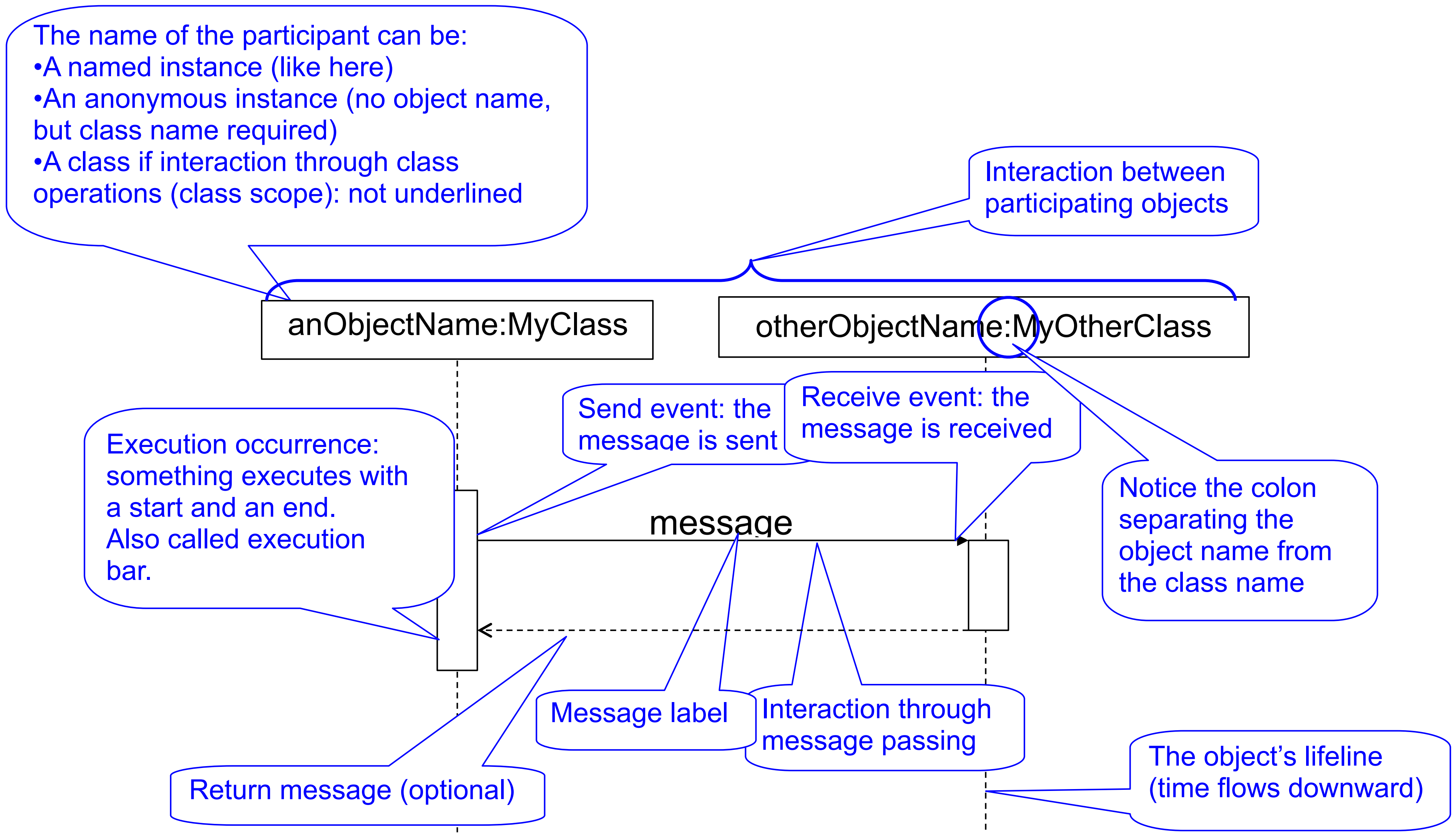
Dynamic Interaction Modelling

- It is necessary to determine how objects that participate in a use case interact with each other.
- Start with the big picture (subsystems).
- Assume that all non-entity (and algorithm) objects are active.
  - Messages between active objects are assumed to be asynchronous.
  - Messages to passive objects is assumed to be synchronous.
- Interaction types may change during design.
- Two types of diagrams: Communication and Sequence.

Dynamic Interaction Modelling (II)

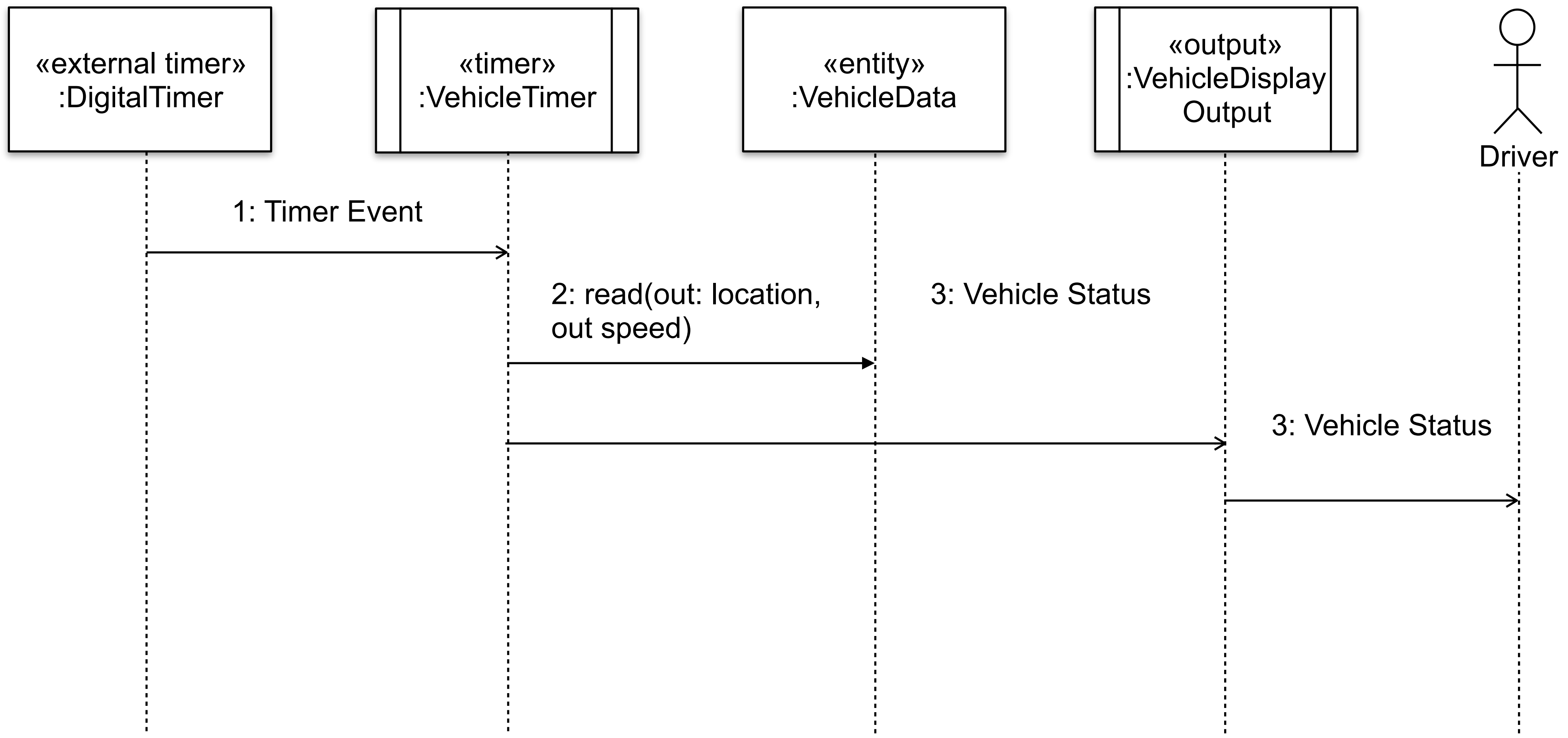
- Sequence Diagrams:
  - Show an exchange of messages between objects arranged in a time sequence.
  - Useful for timing analysis (later).
- Communication Diagrams:
  - Emphasize the relationships between objects along which the messages are exchanged.
  - Diagram is used for structuring, used mostly during design.
- Sequence and collaboration diagrams can be used interchangeably.

Sequence Diagram Notation

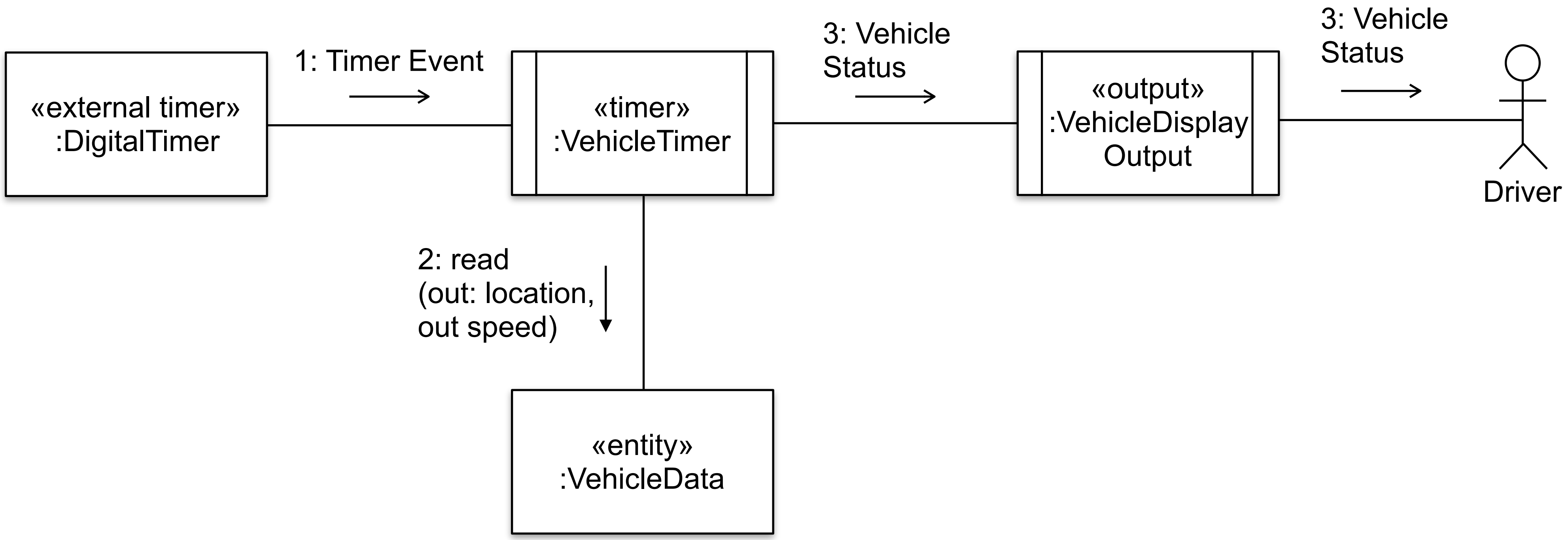


The name of the participant can be:
•A named instance (like here)
•An anonymous instance (no object name, but class name required)
•A class if interaction through class operations (class scope): not underlined
Interaction between participating objects
anObjectName:MyClass
otherObjectName:MyOtherClass
Execution occurrence: something executes with a start and an end. Also called execution bar.
Send event: the message is sent
Receive event: the message is received
Notice the colon separating the object name from the class name
message
Message label
Interaction through message passing
Return message (optional)
The object's lifeline (time flows downward)

Sequence Diagram



Communication Diagram



Stateless Dynamic Interaction Modeling

1. Analyze Use-Case model.
    - Start from the primary actor.
2. Determine objects needed to realize the use case
    a. Determine boundary objects
    b. Determine internal software objects
3. Determine message communication sequence.
4. Determine alternative sequences.

State-Dependent Dynamic Interaction Modelling

- There will be at least one state-dependent control object.
- Both messages and events are involved. A message is an event plus the data that accompanies the event.

1. Determine the boundary objects (which objects receive events from the external environment?)
2. Determine the state-dependent control object. More than one may be required.
3. Determine the other software objects needed.

State Dependent Modelling (II)

4. Determine the object interactions in the main sequence scenario. This step will be done in conjunction with Step 5.
5. Determine the execution of the state machine.
6. Consider alternative sequence scenarios.

- Messages arriving at the control object correspond to events for the machine.
- Actions arising from transitions correspond to messages departing from the control object.

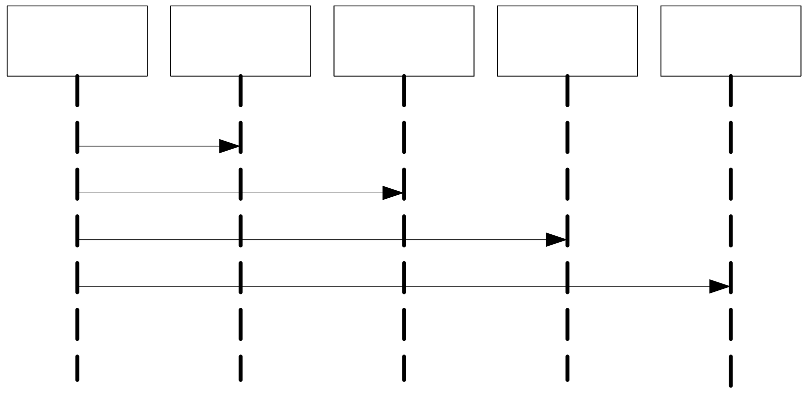
Heuristics for Sequence Diagrams

- First column: actor who initiates the use case
- Second column: boundary object
- Third column: control object that manages the rest of the use case
- Control objects are created by boundary objects initiating use cases
- Other boundary objects are created by control objects
- Entity objects are accessed by control and boundary objects
- Entity objects never access boundary or control objects

Fork and Stair Diagram

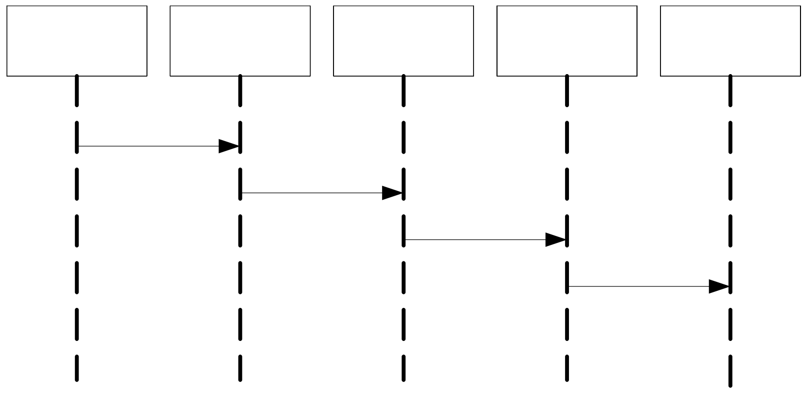
Fork Diagram

- Much of the dynamic behaviour is placed in a single object, usually the control object. It knows all the other objects and often uses them for direct questions and commands.



Stair Diagram

- The dynamic behaviour is distributed. Each object delegates some responsibility to other objects. Each object knows only a few of the other objects and knows which objects can help with a specific behaviour.

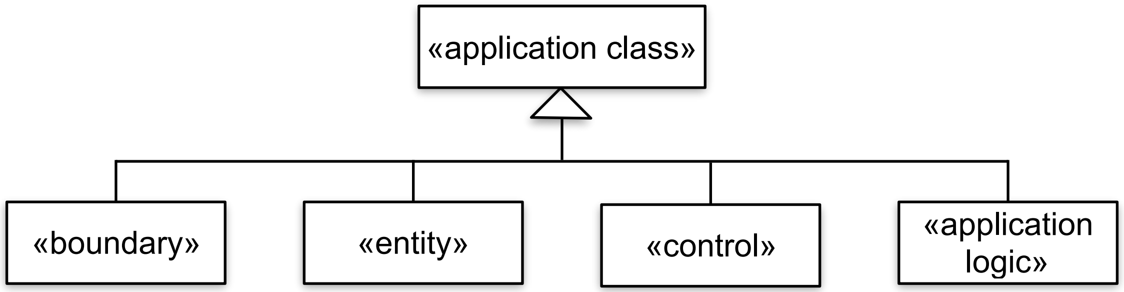


Summary

- Both Sequence and communication diagrams are used.
- Develop the analysis diagrams from the use cases of the system.
- Flow should generally be from left to right for a sequence diagram.
- Diagrams should be consistent with static analysis diagrams (class diagrams).
- For state-based control objects, message in the sequence/communication diagram will correspond to events and actions in the state machine.

SYSC 3303 Real-Time Concurrent Systems

Object and Class Structuring

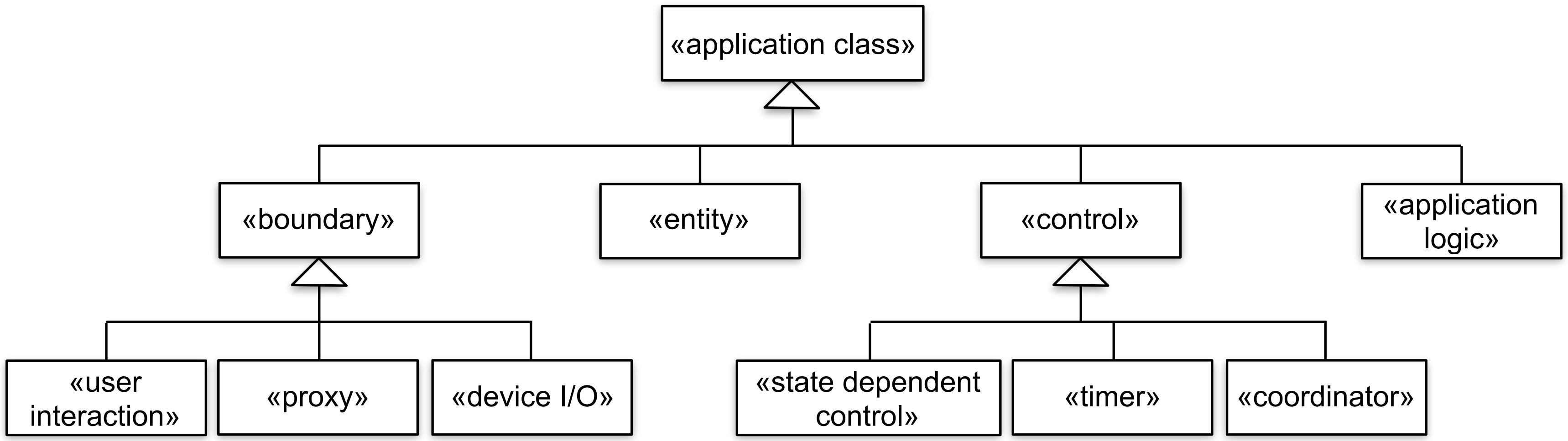


You are here…

1. System structural modelling.
2. Requirements modelling.
3. Analysis modelling:
   1. Static modelling.
   2. Dynamic state machine modelling.
   3. Object structuring.
   4. Dynamic interaction modelling.
4. Design modelling.
5. Incremental software construction (your project!)
6. Incremental software testing (your project!)
7. System testing.

Object and Class Structuring Categories

- Boundary, entity and control classes are often seen elsewhere (SYSC 3020, SYSC 3120).
- Application logic is separated out here.
- Control and Boundary classes are particularly interesting in Real Time and Embedded systems.



Rules for Classes

- Four rules apply to their communication:
  1. Actors can only talk to boundary objects.
  2. Boundary objects can only talk to controllers and actors.
  3. Entity objects can only talk to controllers.
  4. Controllers can talk to boundary objects and entity objects, and to other controllers, but not to actors
- Communication allowed:

	Entity	**Boundary**	**Control**
Entity	Yes		Yes
Boundary			Yes
Control	Yes	Yes	Yes

Object and Class Structuring Categories (II)

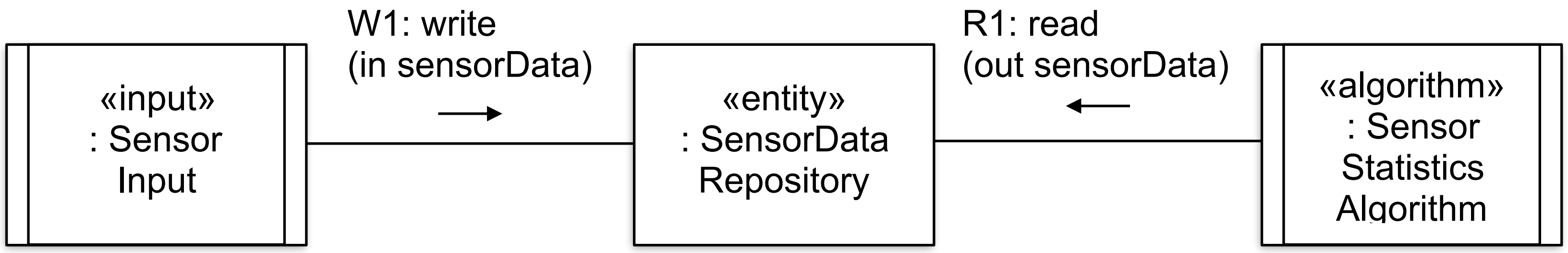
1. **Boundary Object**: defines interfaces to and communicates with the external environment.
   - **User interaction:** the interface to a human user.
   - **Proxy:** interfaces to and communicates with an external system.
   - **Device I/O:** receives input from and/or sends output to a hardware device.
2. **Entity Object**: Encapsulates information. This information may be serialized (stored persistently).

Object and Class Structuring Categories (II)

3. **Control:** provides overall coordination for a collection of objects.
    - **State dependent control:** controls other objects and is state-dependent.
    - **Timer:** controls other objects on a periodic basis.
    - **Coordinator:** controls other objects, but is not state dependent.
4. **Application logic:** encapsulates algorithm and service logic.

Object Behaviour and Patterns

- Our first stab at concurrency!
- As a general rule, all non-entity object are assumed to be concurrent.
  - Each object is **active**, that is it has its own thread of control and can operate in parallel with other objects.
- Entity objects are considered passive, that is they are called from active objects.
- Initially, communication between active objects is asynchronous, communication with passive objects is synchronous (method call).

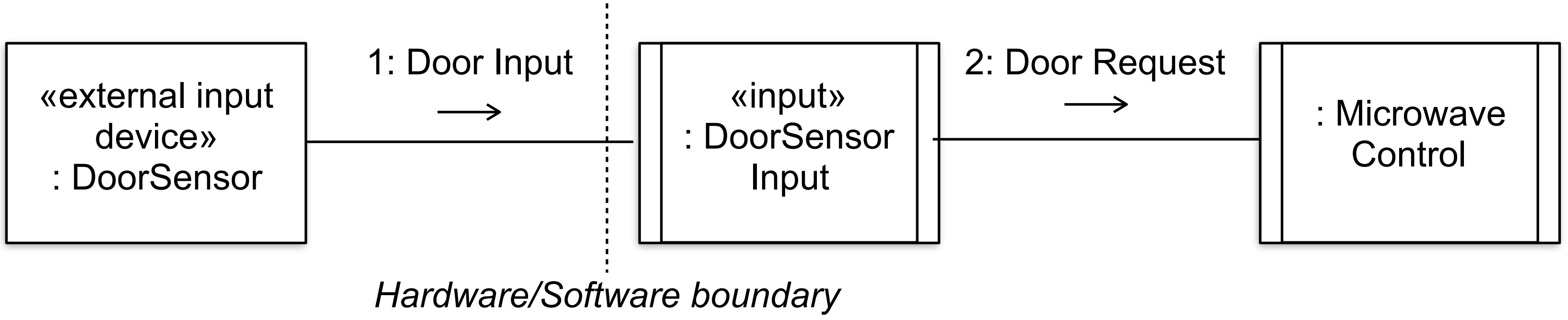


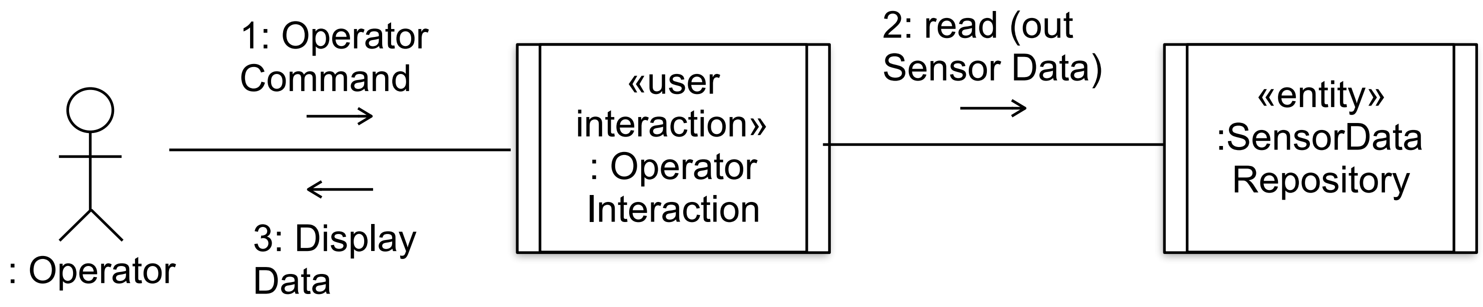
Boundary Classes and Objects

- Boundary objects interface external objects to software boundary objects within the system as follows:
  - An **external device object** represents an I/O device and provides input to and/or receives input from a **device I/O boundary object**. Note that some I/O objects provide input only (keyboard), or output only (display). Others perform both input and output (disk).
  - An **external system object** interfaces with a **proxy object** (i.e., another system)
  - An **external time**r signals to a **timer object**.
  - An **external user object** (i.e., a human) interacts with a **user interface object**.

Boundary Classes and Objects (II)

- The purpose of a boundary object is to abstract away the details of communicating with the "hardware" side of the Hardware/Software boundary.


«external input device» : DoorSensor
1: Door Input →
«input» : DoorSensor Input
2: Door Request →
: Microwave Control
Hardware/Software boundary


: Operator
1: Operator Command →
← 3: Display Data
«user interaction» : Operator Interaction
2: read (out Sensor Data) →
«entity» :SensorData Repository

Entity Classes and Objects

- Entity objects are used to store data. There are two types.
  - Data abstraction objects - transient data.
  - Database objects - persistent data.
- Persistent data may or may not be stored using a database. If so, the database object is simply a wrapper (columns in the DB map to attributes in the object, etc.)
- Entity objects are almost always passive, so mutex must be used in a multi-threaded environment.

Control Classes and Objects

- Control objects are used to control all of the other objects in the system.
- There are three kinds of control objects:
  1. State dependent:
     - Behaviour depends on the object's current state.
     - State changes are caused by inputs (events) from other objects.
  2. Coordinator:
     - Overall decision making object, not necessarily state-based (e.g., the scheduler in the project)
  3. Timer: periodically triggered by an external clock.

Application Logic Classes and Objects

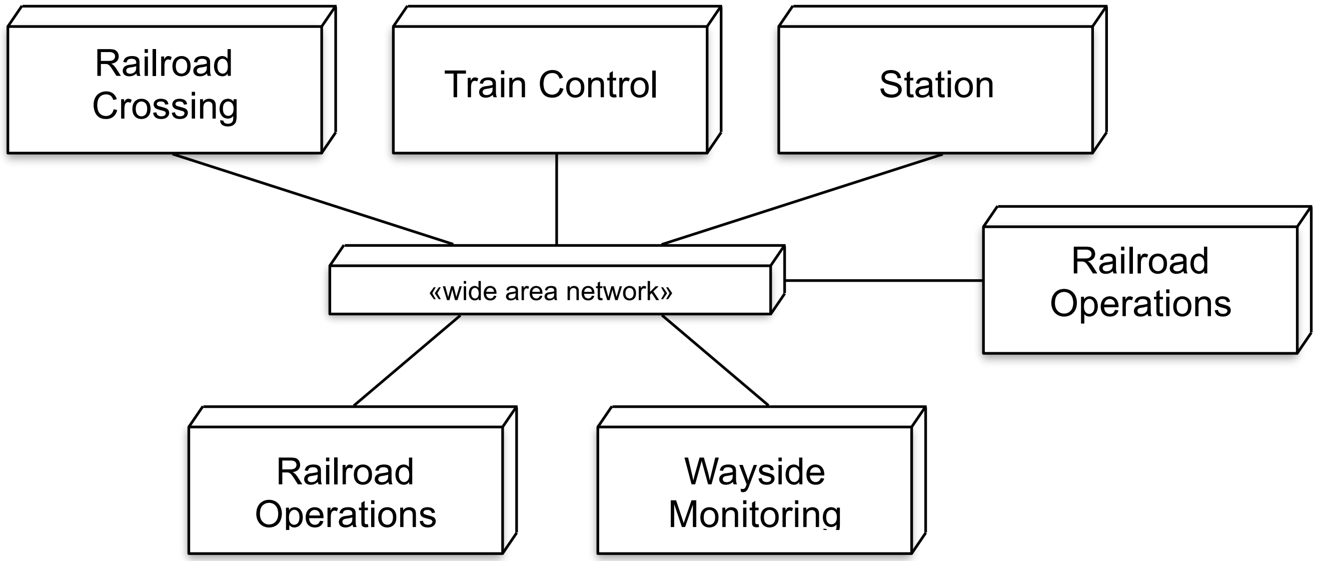
- Application logic classes are used to partition application logic from its data.
  - Why? Logic is expected to change.
- Two types:
  1. Algorithm. Simple algorithms are often encapsulated within entity objects. Complex algorithms are in a separate class and interact with multiple other objects similar to the way a coordinator works.
  2. Service. Provides a service to another object (such as reading and writing data, or forwarding messages across a network).

Summary

- Four types of classes:
    1. Boundary
        - Interface
    2. Entity
        - Stores data
    3. Control
        - Runs the system
    4. Algorithm
        - Miscellaneous stuff that doesn’t fit above.

SYSC 3303 Real-Time Concurrent Systems

Concurrent Real-Time Software Task Design



Concurrent Task Structuring Issues

- An important consideration for a system is what tasks are needed to accomplish its work and how those tasks communicate with each other.
- Multiple tasks are beneficial, especially in this era of cheap multi-core processors.
- Too many tasks can unnecessarily add complexity, communication overhead and context switching.
- Too few tasks can make the software overly complex (fragile, brittle).
- Recall: Goldilocks and the Three Bears.

Categorizing Concurrent Tasks

- Tasks are schedulable resources which means they need to execute on a CPU.
  - MARTE stereotype: «swSchedulableResource»
- Three types of task exist:
  1. Periodic: scheduled a regular intervals.
  2. Event driven: scheduled upon the arrival of an event (such as an I/O interrupt) — aperiodic.
  3. Demand driven: scheduled upon the receipt of a request from another task — aperiodic.

  ✴Periodic and aperiodic are important for scheduling purposes (to come).

Task Structuring Criteria

Four broad groups:

1. I/O task structuring: mapping I/O devices to tasks and determining how and when the task is activated.
2. Internal task structuring: mapping internal objects to tasks and determining how and when the task is activated.
3. Task priority: addressing the issue of relative importance among tasks — first things first!
4. Task clustering: determining whether or not a set of tasks can be aggregated into a single task.
    - Store state on the stack, or in an entity?

I/O Task Structuring Criteria

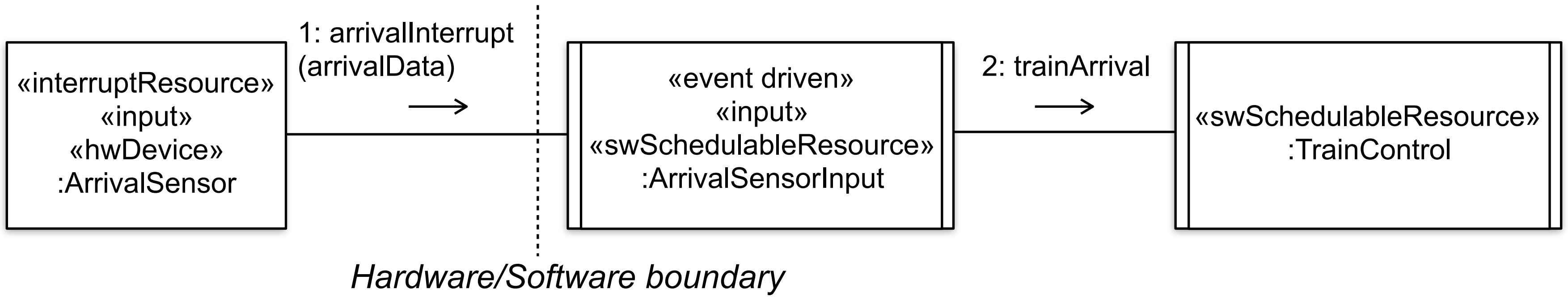
- I/O devices behave in a variety of different ways, so this will have an effect on the task used to access it.
- Three types of hardware devices:
  1. Event driven (or asynchronous I/O): The device generates an interrupt when input is ready to be read, or when output has been sent.
  2. Passive I/O devices: The device has to be polled to read input or send output.
     - Eg: digital to analogue and analogue to digital converters.
     - Can polling be on-demand or periodic, and if the latter, how frequently?

I/O Task Structuring Criteria (II)

3. Smart devices: The device has its own CPU (though maybe limited such as a Direct Memory Access (DMA) controller.
   - The device may be connected directly to the node, access memory directly and interrupt the main CPU as needed (disk drives, network adapters).
   - The device may exist as a separate node and send and receive messages to and from the node using TCP/IP, CAN Bus or some other protocol stack.

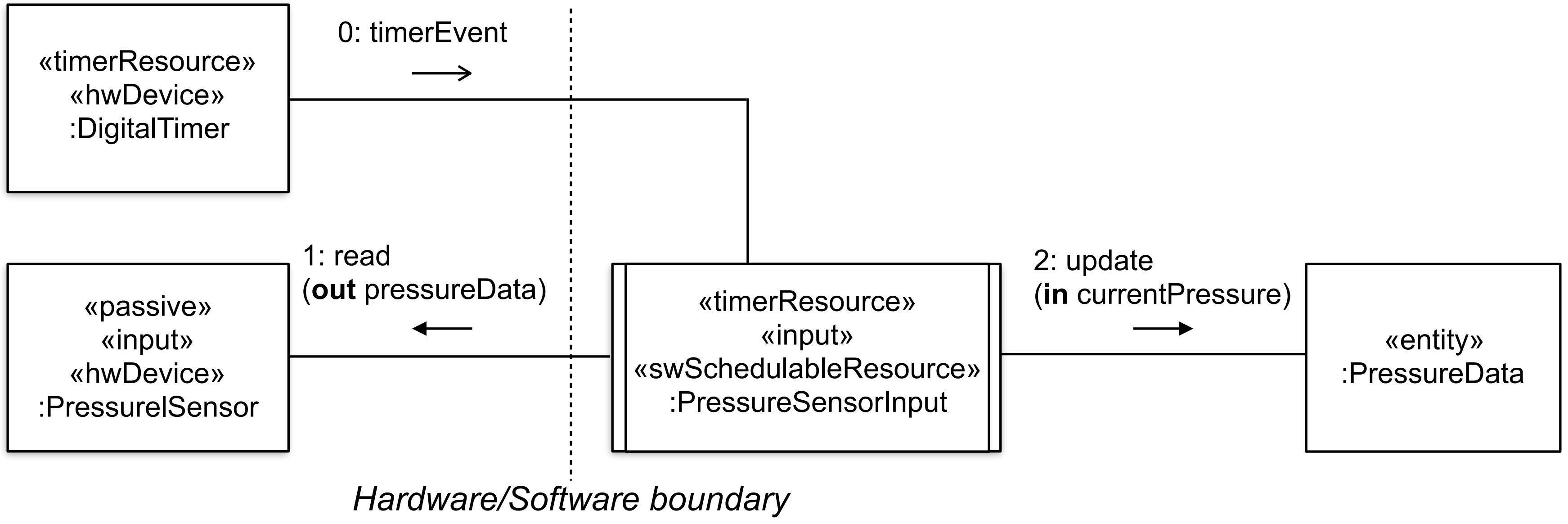
Event Driven I/O Tasks

- Ideally, interrupt handlers do as little as possible — they only service the hardware device and pass the data on to an I/O task who does the heavy lifting.
  - Awoken when an interrupt arrives.
  - The device driver task can then access entity objects or send regular inter-process messages to other tasks.
  - This task is often called a device driver task.



Periodic I/O Tasks

- Passive I/O devices do not generate interrupts, so they have to be polled.
  - One task may poll a large number of sensors — interrupt handling may be excessive otherwise.
  - Sensors may be cheaper if they don't generate interrupts.



Periodic I/O Tasks (II)

- The polling task is generated by a timer event
  - Safety tip: Do not reset the clock in the timer task, rather, set the timer to run in continuous mode.
  - Otherwise you will get timer drift.
- The higher the frequency of the timer, the greater the overhead on the system.
  - Poll devices whose value changes slowly (such as temperature) infrequently.

Demand Driven I/O Tasks

- Demand driven I/O tasks are used for passive I/O devices that do not need to be polled.
  - More typically used for output than for input, i.e., set the output of the digital to analogue converter to this new value.
- A special type of demand driven I/O task is a resource monitor.
  - If multiple tasks can share the same I/O device, the resource monitor is used to control access.
    - e.g., access to a printer.

Internal Task-Structuring Criteria

Types of internal tasks:

- Periodic tasks:
  - Runs at fixed time intervals to run internal activities.
  - Often used for timing purposes.
    - i.e., count seconds to schedule something else.
- Demand driven (or aperiodic) tasks:
  - Runs as the result of an arrival of an event or message from someplace else.
- State-dependent control tasks:
  - Often demand driven.
  - Contains an internal state machine.

Internal Task-Structuring Criteria

- Coordinator tasks:
  - Often demand driven.
  - A decision making task that is not state dependent.
- User interaction and service tasks.
  - Often event driven.
  - Interfaces to typical user I/O devices such as a keyboard, mouse and display.
  - More often than not, the host operating system provides these services.

Task Priority Criteria

- At this time, identify time critical versus non-time critical tasks.
  - Priorities for periodic tasks will be determined by the period of the task.
- Generally, the more time critical a task is, the higher its priority.
  - If a task has a high priority component, and a low priority component, split it into two tasks.
    - Eg: drop control rods, generate log event.
  - Control tasks more often than not have to run at a high priority.

Task Priority Criteria (II)

- Non-time-critical computationally intensive tasks should be run at low priority.
- Maintenance type system diagnostic tasks can be run at low priority.
- User-interface tasks should most likely run at a priority between the high priority critical tasks and low priority computationally intensive tasks.
- Low priorities also have the benefit of shifting work to low demand times.

Task Clustering Criteria

- The analysis model will more often than not produce a large number of objects, each of which can run concurrently with each other.
  - (+) Highly flexible!
  - (-) Too many tiny tasks resulting in a lot of context switching overhead.
- Group objects together to reduce concurrency using:
  1. Temporal clustering.
  2. Sequential clustering.
  3. Control clustering.

Temporal Clustering

- Tasks running with the same period can be grouped together.
  - There can be no sequential dependencies which may cause the task to block on itself 😱!
  - Ordering of tasks is somewhat arbitrary although tasks with the same period, but earlier deadlines, should be executed first.
  - E.g.: polling passive temperature and pressure sensors.
- Grouping tasks with the same period but with differing functions is not considered a good idea from a design standpoint, but may be necessary to lower overhead.

Issues with Temporal Clustering

- If one task is more time critical than another, then split into separate tasks with separate priorities.
  - One could conceivably run tasks with periods which are multiples of each other in one task, for example 25ms, 50ms and 100ms (run once, every two invocations and every four invocations respectively).
  - However, if the periods don’t share a common multiple even though they do share a common divisor, for example, 15ms, 20ms and 25ms. The common task would need a period of 5ms which defeats the purpose.
- If tasks can run on separate cores, then split them.

Temporal Clustering Example

```
class temperature {
void run() {
  while( true ) {
    wait( 0.025 );
    read(temperature_sensor);
  }
}
```

```
class pressure {
void run() {
  while( true ) {
    wait( 0.050 );
    read(pressure_sensor);
  }
}
```

```
class combined {
void run() {
  int iteration = 0;
  while( true ) {
    wait( 0.025 );
    read(temperature_sensor);

// read every other cycle.
    if (iteration==1) {
      read(pressure_sensor);
      iteration = 0;
    } else {
      iteration = 1;
  }
}
```

Sequential Clustering

- Tasks which perform some function, then send the result to another can be clustered sequentially.
- Tasks should not be sequentially clustered if:
  - a task can receive input from more than one task.
  - a task in the sequence can cause an earlier task to miss a deadline. This subsequent task should have a lower priority.
  - a task in the sequence was assigned a lower priority earlier in the analysis.
- If tasks are sequentially clustered, the overhead of shared data is eliminated.

Control Clustering

- A control object is combined with other objects which execute actions caused by actions triggered by the state machine.
  - State dependent action: state transition causes an action which completes almost immediately. The objects called from triggering the action can be clustered.
  - State dependent activities: an activity is performed continuously while in a given state. This activity should be performed by a separate task so that the state machine task can wait for new input.

Design Restructuring Using Task Inversion

- A process of reducing the number of tasks in a systematic way.
  - The limit is only one task for the entire system.
  - More tasks means more overhead from context switching.
- Typical case is with control objects.
  - Rather than instantiating a control object for each interaction, create exactly one instance and have it reference a passive entity object for storing the state.
  - The main line determines which entity instance is required.
    - Eg: call processing.

Developing the Task Architecture

Develop the task architecture in the following order:

1. I/O tasks: Should a task be event, periodic or demand driven? Can it be clustered temporally?
2. Control tasks: A control task most likely should be demand driven. Any objects that are triggered by the control task can potentially be clustered (control or sequential).
3. Periodic tasks: Cluster tasks if they are triggered by the same event. Cluster sequentially triggered tasks.
4. Everything else.

Task Communication and Synchronization

- The goal of clustering is to reduce the amount of inter-task communication and synchronization (which is overhead).
- This step now chooses the style of communication between tasks (see 12-Software Architectures).
  - Asynchronous (one way, buffers needed, most concurrency, possible message loss)
  - Synchronous (two way, shared memory, more blocking).
- This may or may be dictated by the operating system.
  - Java is synchronous.
  - UDP is asynchronous.

External and Timer Event Synchronization

Three types:

1. External event (hardware interrupt)
2. Timer event (timer interrupt)
3. Internal event (software interrupt?)

- The first two are typically handled by interrupt handlers which pass the events to a device driver task (out of scope for this course).
- The latter can be an asynchronous message or a software interrupt (`signal()`). Handling the latter is generally messy.

Summary

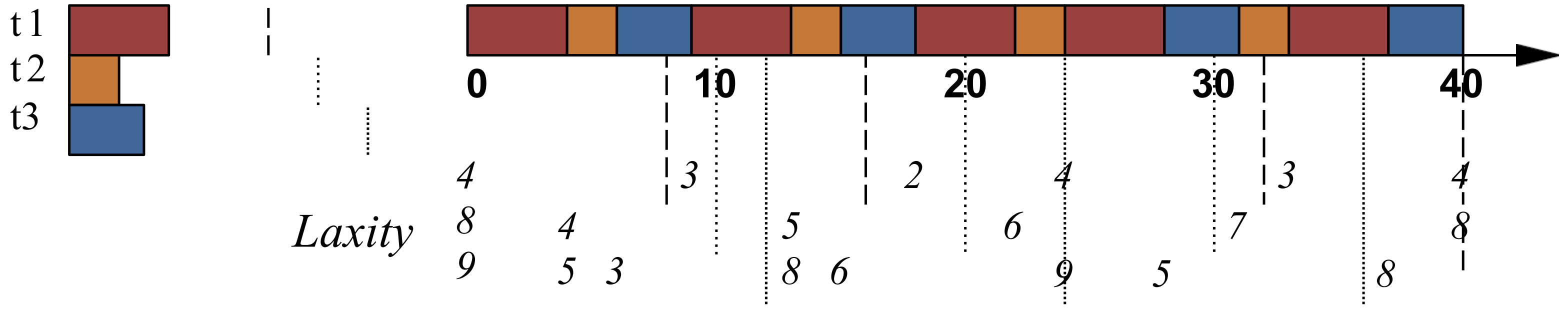
- This phase maps the analysis model to the design model.

1. Mapping I/O devices to tasks and determining how and when the task is activated.
2. Mapping internal objects to tasks and determining how and when the task is activated.
3. Assigning Task priorities.
4. Clustering tasks.

Time to code 😃 🎉 !

SYSC 3303 Real-Time Concurrent Systems

Advanced Scheduling



-
- revised March 16th, 20191

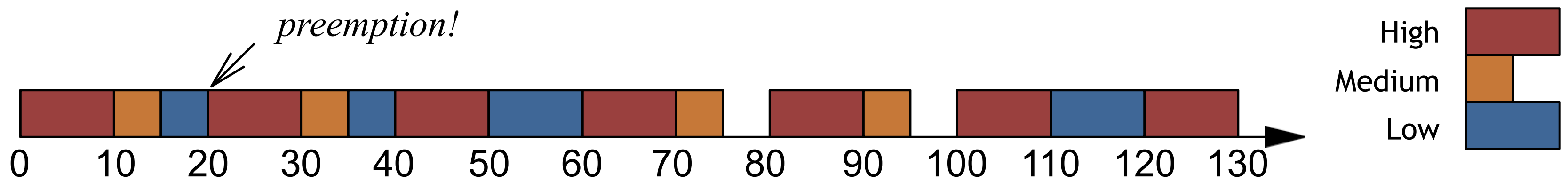
Introduction

- Review of Rate Monotonic Scheduling.
- Getting Parameters 😱.
- Deadline Monotonic Scheduling.
- Completion Time Theorem.
- Earliest Deadline First Scheduling.
- Least Laxity First Scheduling.
- Priority Inversion 🤯.
- Generalized Utilization Bound Theorem.

Rate Monotonic Scheduling

- Rate-monotonic scheduling gives fixed priorities:
  - for deadline = period
  - higher priority to shorter period.
  - stable in conditions of transient overload (though tasks with longer periods might miss deadlines).

Period (T_i)	20	30	50
WCET (C_i)	10	5	10
Priority	high	medium	low



Getting Parameters 😱

Worst Case Execution Time of a task:

- Must be worst case over data variations in arguments.
  - can’t have recursive or iterative algorithms that don’t have a bounded execution!
- This is the **weakest part** of schedulability analysis.
  - cache behaviour causes variability.
  - worst case is with no caching! (maybe 100X worse!!)
  - instruction-level parallelism in processors also causes variability.
  - 🤯

Getting Parameters (II)

Some ideas:

1. Count instructions, take into account instruction length, ignore cache.
2. Count instructions, but simulate the cache as you count (e.g., for a loop, instructions are cached after the first pass).
3. Measure during tests, take the max plus a generous margin.

Completion Time Theorem

A set of *n* independent periodic tasks scheduled by the rate monotonic algorithm will always meet its deadlines for all task phasings, if and only if:

$$\forall i, 1 \leq i \leq n, \min \sum_{j=1}^{i} C_j \frac{1}{pT_k} \left\lceil \frac{pT_k}{T_j} \right\rceil \leq 1, (k,p) \in R_i$$

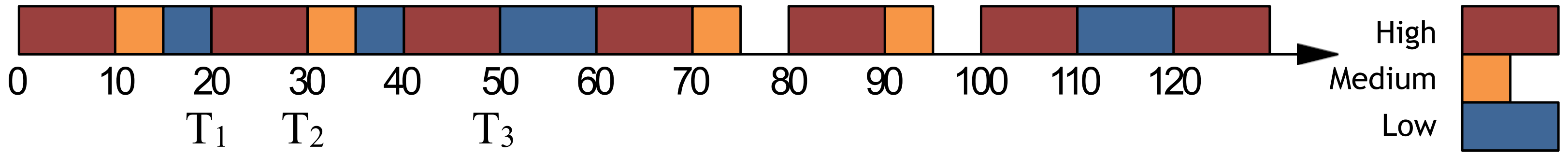
where C_j and T_j are the execution time and period of task T_j respectively and

$$R_i = \{(k,p) \mid 1 \leq k \leq i, p = 1, \ldots, \lfloor T_i / T_k \rfloor \}$$

- T_i – task to be checked.
- T_k – higher priority tasks.
- p – Scheduling point of task T_k.

Completion Time Theorem (II)

Period (T_i)	20	30	50
WCET (C_i)	10	5	10



- First scheduling point:

$$C_1 + C_2 + C_3 \le T_1, \;\; 10 + 5 + 10 > 20, \;\; p = 1, \;\; k = 1$$

  - t_3 is preempted!

- Second scheduling point:

$$2C_1 + C_2 + C_3 \le T_2, \;\; 20 + 5 + 10 > 30, \;\; p = 1, \;\; k = 2$$

- Third scheduling point:

$$3C_1 + 2C_2 + C_3 \le T_3, \;\; 30 + 10 + 10 \le 50, \;\; p = 1, \;\; k = 3$$

  - Note: utilization is 1.0

A Slot for Sporadic Tasks

- Sporadic tasks arrive more or less at random, anything that isn't periodic.
- A periodic slot can be scheduled for these.
  - Long enough for the longest execution time.
- They queue for the slot, possibly waiting several cycles.
  - This is like having a server of their own provided by the slot.
- Either no deadline guarantee (best effort).
  - Or, if there is a big enough minimum inter-arrival time it may be possible to give guarantees.

Deadline Monotonic Scheduling

- Tasks run periodically like RMS, but they now must complete at some time D_i after they start.
- Assign the highest priority to the task with the shortest ***deadline***.
- Response time of t_i: $R_i = C_i + I_i$.
    - I_i is the ***interference*** from higher priority tasks.

$$I_i = \sum_{\forall k \in H_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

- Response time for ti is found by iterating:

$$R_i^{n+1} = C_i + \sum_{\forall k \in H_i} \left\lceil \frac{R_i^n}{T_k} \right\rceil C_k$$

Example of Deadline Monotonic Scheduling

Task	T_i	C_i	D_i
1	250	5	10
2	10	2	10
3	330	25	50
4	1000	29	1000

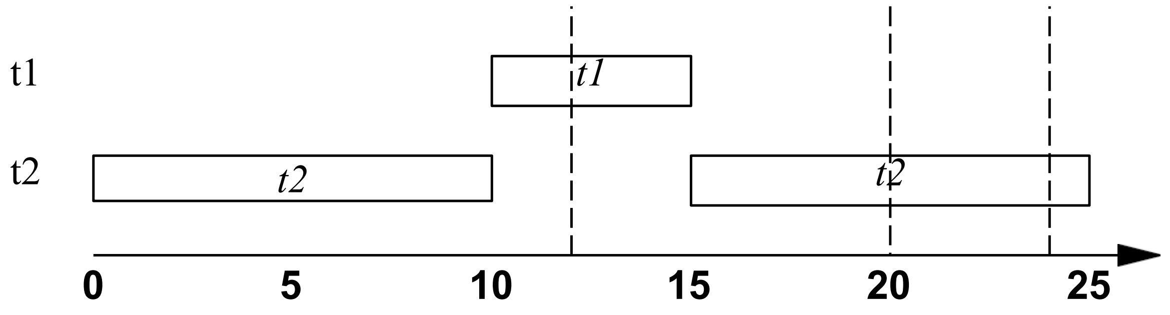
Worst case response time for t_3:

Step	R^n	C_3	I	R^{n+1}
1	0	25	0	25
2	25	25	$5 + 3 \times 2 = 11$	36
3	36	25	$5 + 4 \times 2 = 13$	38
4	26	25	$5 + 4 \times 2 = 13$	38

Earliest Deadline First

- Earliest absolute deadline gives priority
- Priority only changes when a task is released or finishes

Tasks	t_1	t_2
Computation time (C_i)	5 ms	10 ms
Deadline (D_i)	20 ms	12 ms



- Utilization = 5/20 + 10/12 = 1.08 > 100% (infeasible) ☹

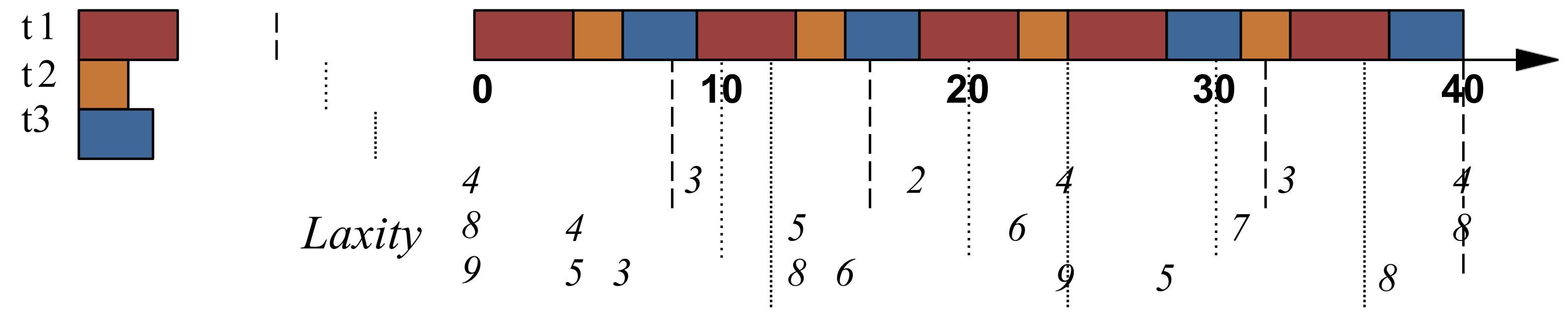
Optimality of Earliest Deadline First

- Earliest Deadline first can schedule any task sequence with $U \leq 1.0$.
- Advantages:
  - Simple analysis 😀.
- Disadvantages:
  - overhead for recomputing priorities constantly 😥.
  - Behaviour unpredictable in the presence of overloading 😯.

Least Laxity First

- Time to next deadline gives priority.
- Priority only changes when a task is released or finishes

Period (T_i)	8	10	12
WCET (C_i)	4	2	3



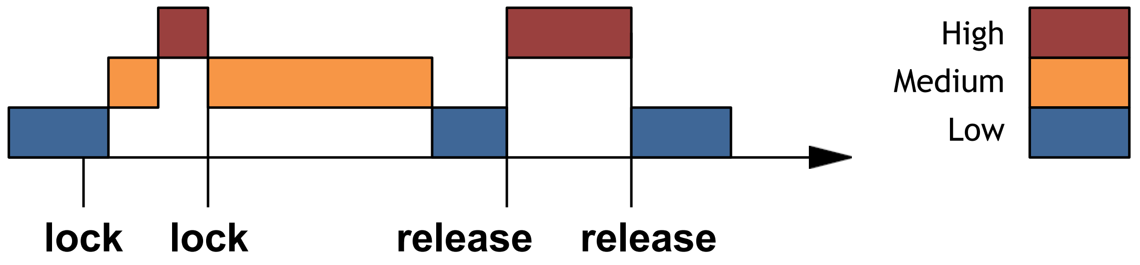
- Utilization = 4/8 + 2/10 + 3/12 = 0.95 ≤ 100% (feasible) 😀

Blocking on Shared Resources

- Such as a critical section to do:
  - a disk operation,
  - a network operation,
  - using shared memory (i.e., `synchronized`)
- Tasks are no longer independent. A resource held by another task (not running) can block a task and make it wait.
- So: adapt the analysis, adding worst case blocking time B_i to C_i.
- But first we have to deal with Priority Inversion.

Priority Inversion

- Low priority P1 has a resource but cannot run because of higher priority task P2.
- High priority P2 must wait for the resource.
- So a medium priority P3 can preempt P1 and run (making P1 and P2 wait even longer)



- Blocking can be unbounded (no limit on the number of times an intermediate priority task may run)

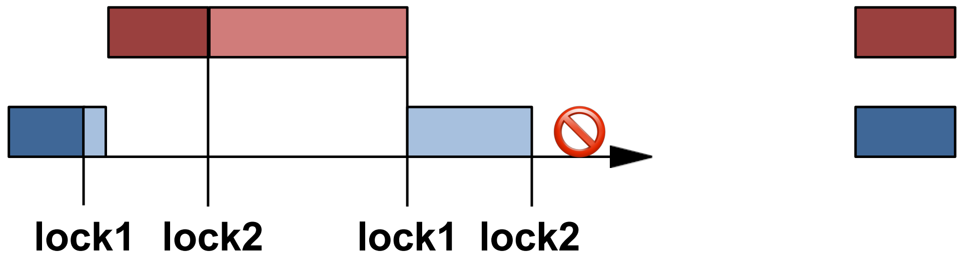
Solutions to priority inversion

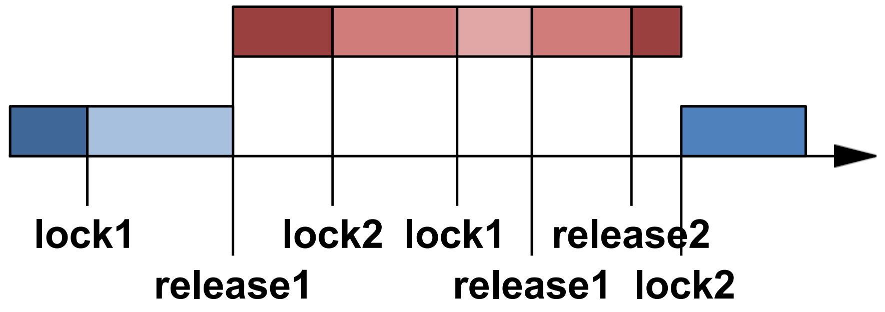
- One solution: ***Priority Inheritance***
  - the task using the resource (critical section, semaphore) gets its priority raised to that of any job waiting for the resource.
  - now it gets to finish right away and hand over to the highest priority waiting job.
  - now the higher priority jobs only wait for the one lower priority job at the resource.
  - however, blocking B_i for higher priority jobs can be large.

Priority Ceiling Protocol

- Each resource has a ceiling priority, that is the highest priority of any task that uses it.
- Every task that uses the resource inherits the ceiling protocol right away.
- Now while it holds the resource, no other task can begin to wait for it (assuming a single processor)
  - Hence, no deadlocks possible.
- Queueing for the resource is non-preemptive priority (head of line), so task *i* jumps over earlier requests by lower priority tasks

Priority Ceiling Protocol (II)





Generalized Utilization Bound Theorem

- For independent periodic tasks with blocking times B_i
- Any static task priorities (not necessarily RMS)
- Uses a worst-case artificial utilization U_i that may occur during one period T_i of task i.
- Four factors are needed:
  1. Preemption by higher priority tasks with periods less than t_i. Call this set H_n with members j. Tasks in this set can preempt t_i many times.
  2. Execution time for task t_i. Period T_i, consumes C_i.
  3. Preemption by higher priority tasks with longer periods. Call this set H_1 — they can only preempt t_i once.
  4. The maximum blocking time B_i by lower-priority tasks holding a resource

Generalized Utilization Bound Theorem

$$U_i = \left(\sum_{j \in H_n} \frac{C_j}{T_j} \right) + \frac{1}{T_j} \left(C_i + B_i + \sum_{k \in H_1} C_k \right)$$

- If U_i is less than the worst-case upper bound, then this task will meet its deadlines.
- This equation must be applied to every task — a given task meeting its deadline is no guarantee that higher priority tasks meet theirs.

Blocking time

- With the Priority Ceiling Protocol, at most one lower-priority task can block task *i*, for each resource requested
- Higher priority task time is included elsewhere
- Lower priority tasks can't get ahead of task *i*.
- For each resource used by task *i* we can include in B_i either:
  - the max C_j for any lower task that uses the resource
  - if we know it, the max C_j^* which is the time that task j holds the resource for (the service time for the resource when used by task j)

Example

- Four tasks, two periodic, two aperiodic.
  1. Periodic task t_1: $C_1 = 20, T_1 = 100, U_1 = 0.2.$
  2. Aperiodic task t_2: $C_2 = 15, T_2 = 150, U_2 = 0.1.$
  3. Int. Aperiodic task t_a: $C_a = 4, \quad T_a = 200, U_a = 0.02.$
  4. Periodic task t_3: $C_3 = 30, T_3 = 300, U_3 = 0.1.$
- t_1, t_2, t_3 all access shared memory *s*.
- t_a given highest priority (not rate-monotonic!)
- $U_a = 0.2 + 0.1 + 0.02 + 0.1 = 0.42 < 1.0$

Example (II)

- Task t_a will meet deadlines ($U_a = 0.02$) 😀.
- Task t_1 (factors):
  1. No tasks with periods less than T_1.
  2. $U_1 = 0.2$
  3. Preemption by higher priority tasks with longer periods: t_a.

     $T_1 = C_a/T_1 = 4/100 = 0.04$
  4. Blocking time from priority ceiling protocol: worst case is t_3.

     $T_1 = B_3/T_1 = 30/100 = 0.30$

- $U_1 = 0.04 + 0.2 + 0.3 = 0.54 < 0.69$ (feasible) 😀.

Example (II)

- Task t_2 (factors):
  1. t_1 has a periods less than T_2. $U_1 = 0.2$.
  2. $U_2 = 0.1$.
  3. Preemption by higher priority tasks with longer periods: t_a.

     $T_2 = C_a/T_2 = 4/150 = 0.03$
  4. Blocking time from priority ceiling protocol: worst case is t_3.

     $T_2 = B_3/T_2 = 30/150 = 0.2$

- $U_2 = 0.2 + 0.1 + 0.03 + 0.2 = 0.53 < 0.69$ (feasible) 😀.

Example (III)

- Task t_3 – lowest priority (factors):
  1. t_1, t_2, t_a all have periods less than T_3.

     U = 0.2 + 0.1 + 0.02 = 0.32.
  2. $U_3 = 0.1$
  3. No higher priority tasks have longer periods.
  4. No blocking time from priority ceiling protocol.

- $U_3 = 0.32 + 0.1 = 0.42 < 0.69$ (feasible) 😀.

Summary

- Hard Real Time: Must be able to guarantee deadlines.
  - Guaranteed through *schedulability analysis*.
  - Use worst case execution, not average.
- Rate-monotonic scheduling: shortest period is given highest priority.
- O.K. if utilization bound $U \leq n(2^{1/n} - 1)$ satisfied.
- Otherwise, check completion time from “critical instant”.
  - Lots of ways shown.
  - For a small number of tasks, just draw a picture.
- Sporadic tasks (arrive “randomly”): add a slot!

Summary (II)

- Deadline Monotonic Scheduling – highest priority to shortest deadline.
- Earliest Deadline First – task whose deadline is next is given highest priority.
- Least Laxity First – task which has least time between completion and deadline gets higest priority.
- Priority Inversion – **BAD!** 😡
    - Priority inheritance (easy).
    - Priority ceiling (hard to set up, prevents deadlock).
- Generalized Utilization Theorem – accounts for blocking and other factors.