Introduction to Embedded & Real-Time Systems

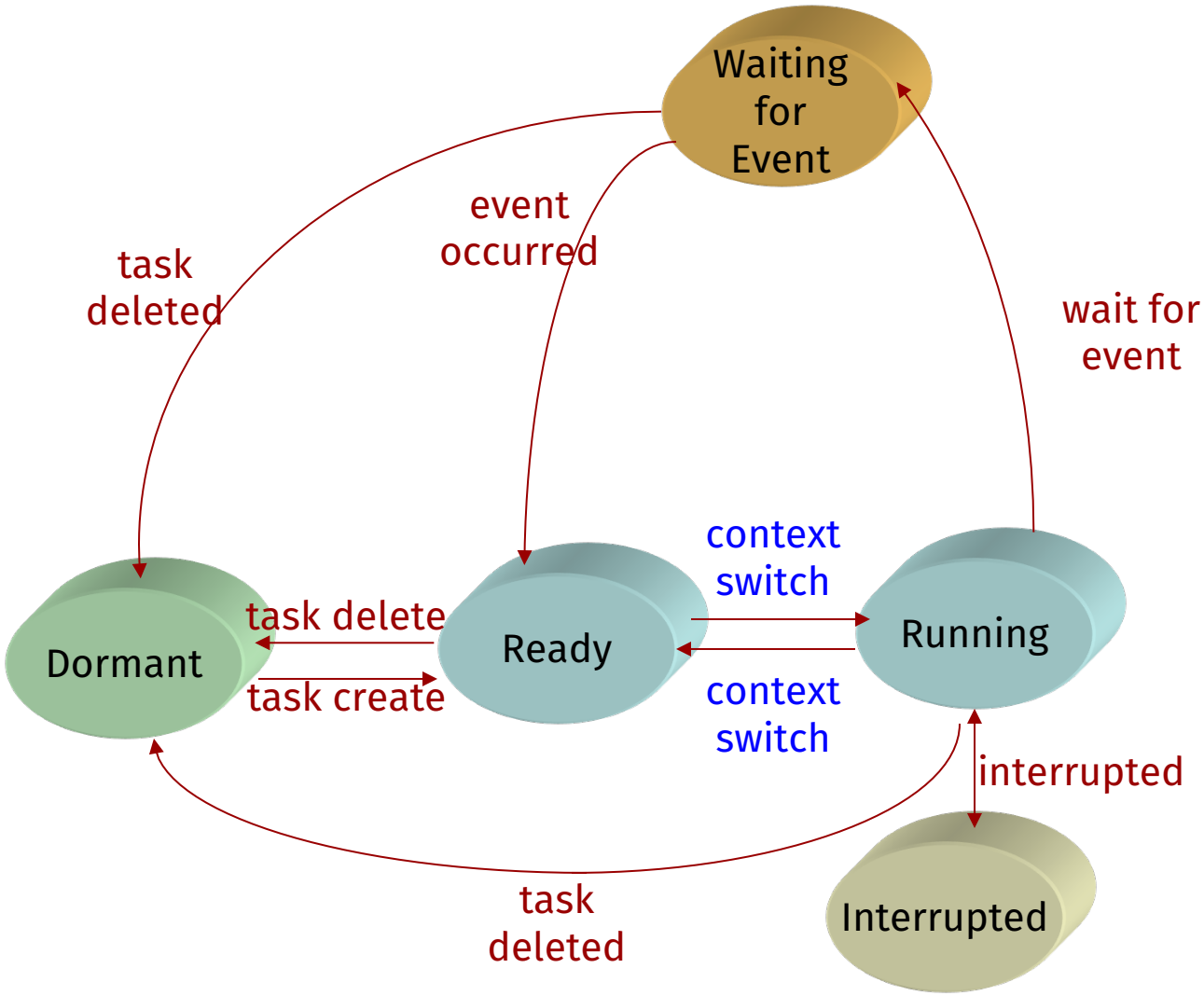
Scheduling and Concurrency

Sathish Gopalakrishnan
Electrical and Computer Engineering
The University of British Columbia

Outline of This Lecture

- Concurrency Primitives
- Real-time systems
  - Characteristics
  - Example real-time systems
- Terminology
  - Hard and soft real-time systems
- Rate-Monotonic Analysis (RMA)
  - Next week

Process State

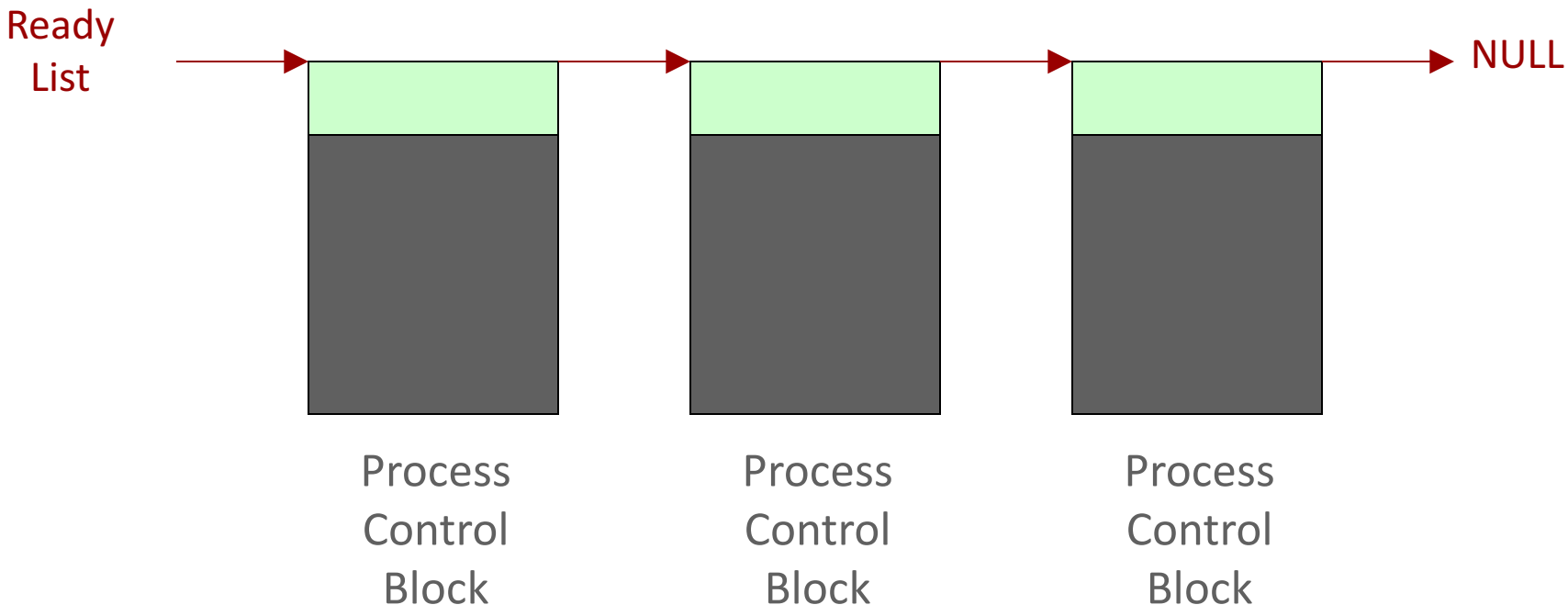


Process Scheduling

- What is the scheduler?
  - Part of the operating system that decides which process/task to run next
  - Uses a scheduling algorithm that enforces some kind of policy that is designed to meet some criteria

- Criteria may vary
  - CPU utilization keep the CPU as busy as possible
  - Throughput maximize the number of processes completed per time unit
  - Turnaround time minimize a process' latency (run time), i.e., time between task submission and termination
  - Response time minimize the wait time for interactive processes
  - Real-time must meet specific deadlines to prevent "bad things" from happening

Ready List

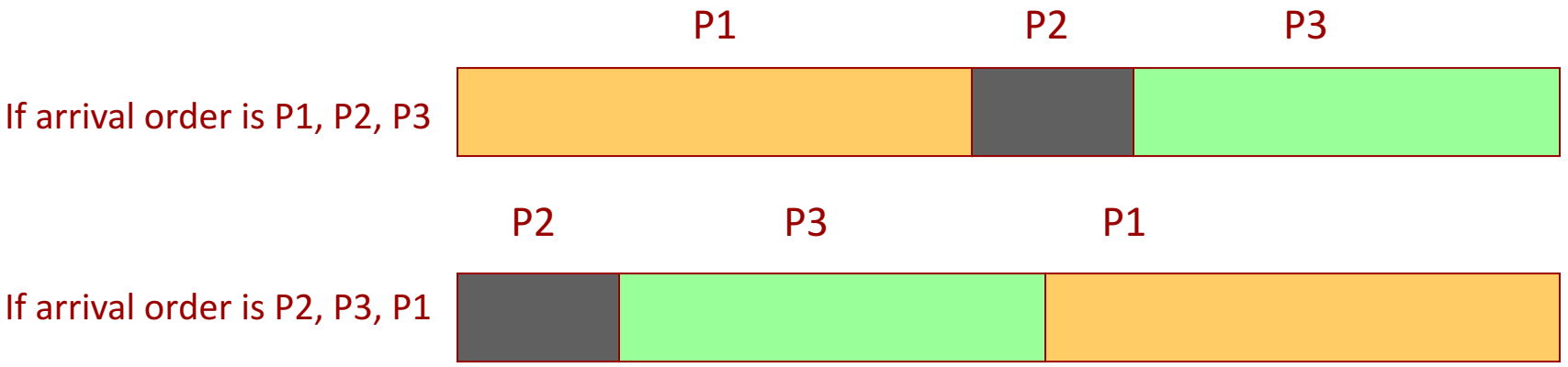

Ready
List
NULL
Process
Control
Block
Process
Control
Block
Process
Control
Block

FCFS Scheduling

- **Firstcome, firstserved** (FCFS)
  - The first task that arrives at the request queue is executed first, the second task is executed second and so on
  - Just like standing in line for a rollercoaster ride

- FCFS can make the wait time for a process very long

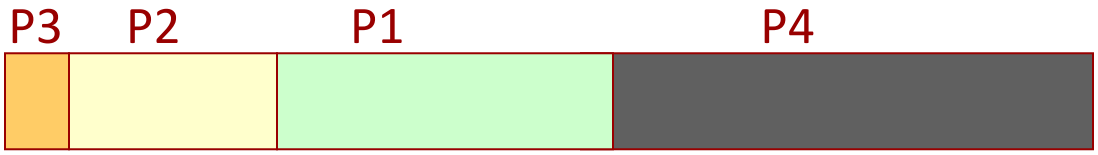
Process	Total Run Time
P1	12 seconds
P2	3 seconds
P3	8 seconds



ShortestJobFirst Scheduling

- Schedule processes according to their run-times

Process	Total Run Time
P1	5 seconds
P2	3 seconds
P3	1 second
P4	8 seconds

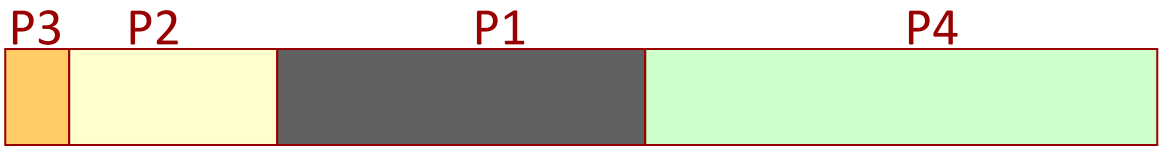


- May be runtime or CPU bursttime of a process
  - **CPU burst time** is the time a process spends executing in-between I/O activities
  - Generally difficult to know the run-time of a process

Priority Scheduling

- ShortestJobFirst is a special case of priority scheduling
- **Priority scheduling** assigns a priority to each process. Those with higher priorities are run first.
  - Priorities are generally represented by numbers, e.g., 0..7, 0..4095
  - No general rule about whether zero represents high or low priority
  - We'll assume that higher numbers represent higher priorities

Process	BurstTime	Priority
P1	5 seconds	6
P2	3 seconds	7
P3	1 second	8
P4	8 seconds	5



Interactions Between Processes

- Multitasking ⇨ multiple processes/tasks providing the illusion of "running in parallel"
  - Perhaps really running in parallel if there are multiple processors
- A process/task can be stopped at any point so that some other process/task can run on the CPU
- At the same time, these processes/tasks running on the same system might interact
  - Need to make sure that processes do not get in each other's way
  - Need to ensure proper sequencing when dependencies exist
  - *Rest of lecture*: how do we deal with **shared state** between processes/tasks running on the same processor?

Critical Section

- Piece of code that must appear as an atomic action
- **Atomic Action** action that "appears" to take place in a single indivisible operation

```
process one          process two
Balance += 20;       Balance -= 20;
```

- if "+=, -=" can execute atomically, then there is no race condition
- **Race condition** outcome depends on the particular order in which the operations takes place

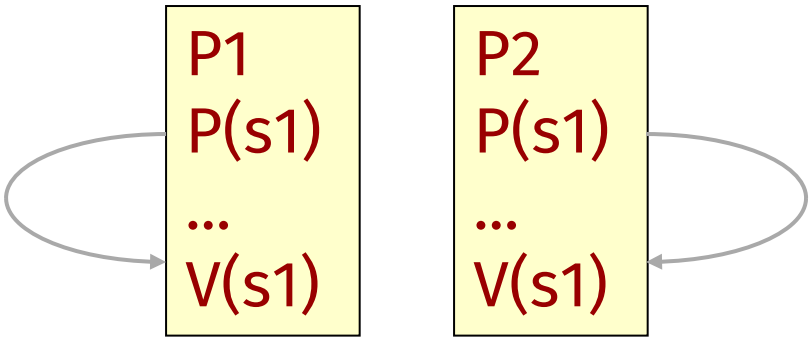
Semaphores

- **Semaphore** an integer variable (> 0) that normally can take on only nonzero values

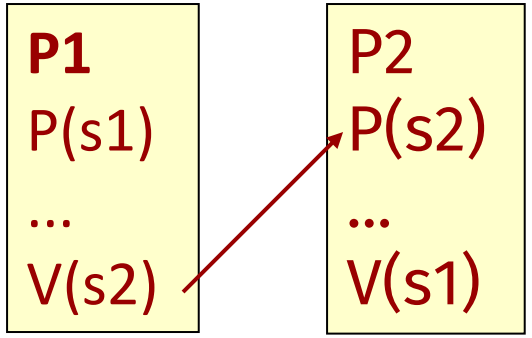
- Only three operations can be performed on a semaphore **all operations are atomic**
  - init(s, #)
    - Sets semaphore, s, to an initial value #
  - wait(s) or P(s)
    - If s > 0, then s = s 1;
    - Else suspend the process that called wait
  - signal(s) or V(s)
    - s = s + 1;
    - If some process P has been suspended by a previous wait(s), wake up process P
      - The process waiting the longest gets woken up

Types of Synchronization with Semaphores

- There are 2 basic types of synchronization
  - Mutex – where semaphores serve to protect a critical section (shared data/code)
  - Barrier synchronization – where semaphores are used - sort of - as an inter-task communication facility to "wake up" other waiting processes
- Remember P=wait and V=signal
- Here, s1 and s2 are semaphores



Initial value of s1 = 1
Mutex



Barrier Synchronization
Initial value of s1 = 1, s2 = 0

Atomic SWAP Instruction on the ARM

- SWP combines a load and a store in a single, atomic operation

```
ADR     r0, semaphore
SWPB    r1, r1,[r0]
```

  - SWP loads the word (or byte) from memory location addressed in Rn into Rd and stores the same data type from Rm into the same memory location
  - SWP<cond> {B} Rd, Rm, [Rn]

ARM Binary Mutex

```
locked    EQU 1
unlocked  EQU 0

lock_mutex_swp PROC
   LDR r2, =locked
   SWP r1, r2, [r0]          ; Swap R2 with location [R0], [R0] value placed in R1
   CMP r1, r2                ; Check if memory value was 'locked'
   BEQ lock_mutex_swp        ; If so, retry immediately
   BX  lr                    ; If not, lock successful, return
   ENDP

unlock_mutex_swp PROC
   LDR r1, =unlocked
   STR r1, [r0]              ; Write value 'unlocked' to location [R0]
   BX  lr
   ENDP
```

ARM Binary Mutex v6 and on

```
locked    EQU 1
unlocked  EQU 0

lock_mutex PROC
    LDR     r1, =locked
1   LDREX   r2, [r0]
    CMP     r2, r1              ; Test if mutex is locked or unlocked
    BEQ     %f2                 ; If locked - wait for it to be released, from 2
    STREXNE r2, r1, [r0]        ; Not locked, attempt to lock it
    CMPNE   r2, #1              ; Check if Store-Exclusive failed
    BEQ     %b1                 ; Failed - retry from 1
    ; Lock acquired
    DMB                         ; Data memory barrier
    BX      lr

2   ; Take appropriate action while waiting for mutex to become unlocked
   WAIT_FOR_UPDATE
   B       %b1          ; Retry from 1
   ENDP
```

ARM Binary Mutex v6 and on

```
locked    EQU 1
unlocked  EQU 0
; unlock_mutex
; Declare for use from C as extern void unlock_mutex(void * mutex);
    EXPORT unlock_mutex
unlock_mutex PROC
    LDR     r1, =unlocked
    DMB                 ; Required before releasing protected resource
    STR     r1, [r0]      ; Unlock mutex
    SIGNAL_UPDATE
    BX      lr
    ENDP
```

ARM Power Saving

```
MACRO
   WAIT_FOR_UPDATE
   WFE                ; Indicate opportunity to enter low-power state
   MEND

   MACRO
   SIGNAL_UPDATE
   DSB                ; Ensure update has completed before signalling
   SEV                ; Signal update
   MEND
```

The Producer/Consumer Problem

- One process produces data, the other consumes it
  - Example: I/O from keyboard to terminal

```
producer(){
  while(1){
    produce();
    appendToBuffer();
    signal(itemReady);
  }
}
```

```
consumer(){
    while(1){
        wait(itemReady);
        takeFromBuffer();
        consume();
    }
}
```

Initially, `itemReady = 0`

Another Producer/Consumer

- What if both `appendToBuffer()` and `takeFromBuffer()` cannot overlap in execution?
  - For example, if buffer is a shared link list between producer and consumer?
  - Or, multiple producers and consumers

```
producer() {
  while(1){
    produce();
    wait(mutex);
    appendToBuffer();
    signal(mutex);
    signal(itemReady);
  }
}
```

```
consumer() {
    while(1){
        wait(itemReady);
        wait(mutex);
        takeFromBuffer();
        signal(mutex);
        consume();
        }
}
```

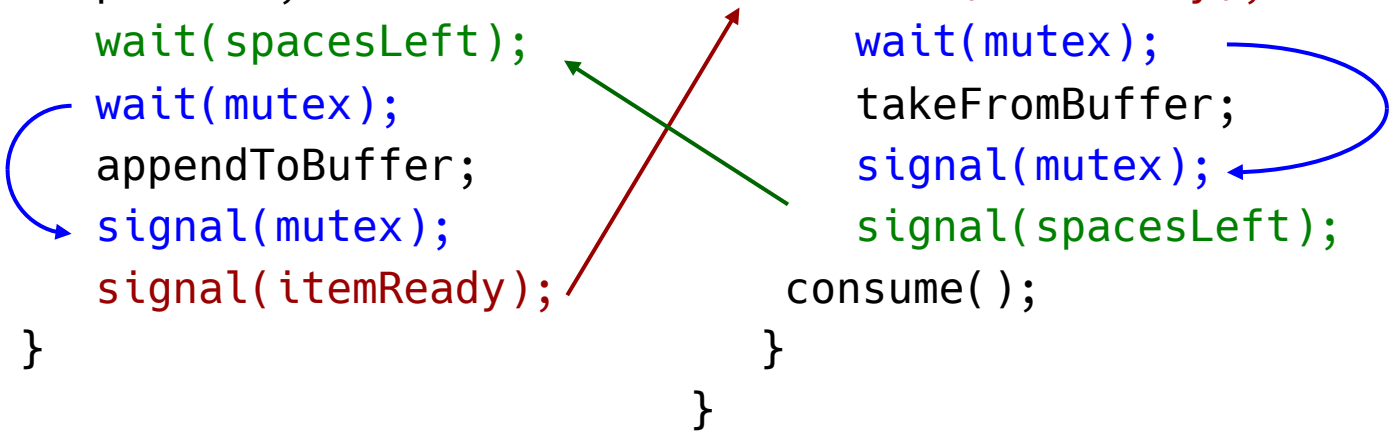
- Initially, `mutex = 1, itemReady = 0`

Bounded Buffer Problem

- Assume a single buffer of fixed size
  - Consumer blocks (sleeps) when buffer is empty
  - Producer blocks (sleeps) when the buffer is full

```
producer( ) {
  while(1) {
     produce;
     wait(spacesLeft);
     wait(mutex);
     appendToBuffer;
     signal(mutex);
     signal(itemReady);
  }
}
```

```
consumer( ) {
    while(1){
        wait(itemReady);
        wait(mutex);
        takeFromBuffer;
        signal(mutex);
        signal(spacesLeft);
     consume( );
    }
 }
```



- Initially, s = 1, n = 0; e = sizeOfBuffer;

Deadlocks

- When a program is written carelessly, it may cause a... **deadlock**!

```
P1::
P(s1);
P(s2);
...
V(s2);
V(s1);
```

```
P2::
P(s2);
P(s1);
...
V(s1);
V(s2);
```

Deadlocks

- Several processes “executing” at the same time, and one is waiting (blocked) for (by) another, which in turn is waiting for another, which in turn... which in turn is waiting for the first.
  - No process can finish, since they are all waiting for something

- The difference between deadlock and starvation of a process is that
  - In deadlock, a process must be able to acquire a resource at first, and then go into deadlock.
  - In starvation, the request may be deferred infinitely
    - The resource may be in use for an infinite amount of time

Resources

- What could be a resource?
  - Hardware devices e.g. printer, eeprom needed by a process to do useful work
  - Software resource e.g. mutex
- Processes utilize resources in the following sequence
  - Request (must wait if request is not granted)
  - Use
  - Release
- How do we get into a deadlock?
  - Consider a system with one printer and one eeprom
  - Process 1 requests printer and the request is granted
  - Process 2 requests eeprom and the request is granted
  - Process 1 also needs eeprom to finish its job
    - Must wait for Process 2 to release eeprom
  - Process 2 also needs printer to finish its job
    - Must wait for Process 1 to release eeprom
  - Deadlock!

Necessary Conditions for a Deadlock

- Formally: (***all*** of these conditions ***must*** hold for deadlock to occur)
  - Mutual exclusion: some resource is nonsharable, (eg, a printer)
  - Hold and wait: there must exist a process holding for a resource (e.g, a printer) and waiting for another resource (eg, a scanner)
  - No preemption: the system will not preempt the resources in contention
  - Circular wait: there must exist a circular chain of processes.
    - One is holding a resource and waiting for the next process

Deadlock Handling

- **Prevention**: structure the system is such a way as to avoid deadlocks (i.e., in a way to avoid one of the conditions above).
  - This is done in the design phase: design a system and ensure that there is no deadlock in the system
- **Avoidance**: does not make deadlock impossible (as in prevention)
  - Instead, it rejects requests that cause deadlocks by examining the requests before granting the resources. If there will be a deadlock, reject the request
- **Detection and Recovery**: After the deadlock has been detected, break one of the 4 conditions above.
  - The mechanisms of deadlock detection and deadlock recovery are very much tied to each other

Deadlock Prevention

- Ensure one of the four necessary condition is never satisfied
  - Allow all resources to be shared (so prevent mutual exclusion)
    - Some resources are inherently non-sharable
  - Don't allow hold and wait, i.e., force processes to either acquire all the needed resources or to release the acquired (currently held) resources if one or more requests are not granted
    - Starvation of processes that need many resources
  - Allow preemption of resources
  - Enforce total ordering in resource acquisition process and require each process to request resources in an increasing order
    - Example: Printer order=3, EEPROM order = 7
      - If process 1 needs printer and EEPROM in any order, it must request printer first and then EEPROM

Can cause deadlock

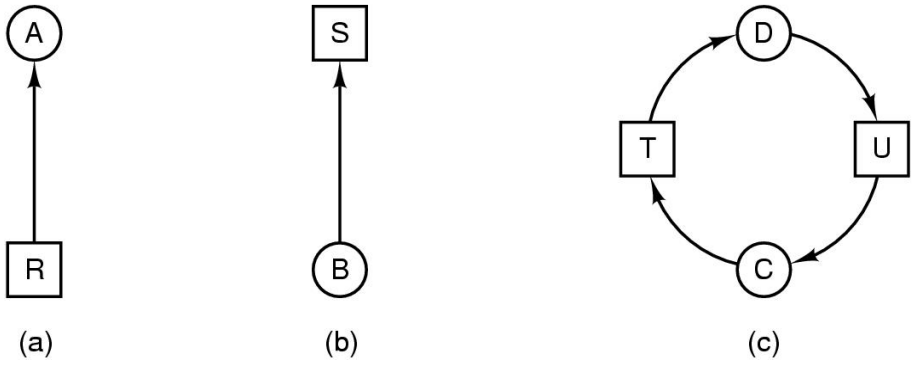
```
P1::        P2::
P(s1);      P(s2);
P(s2);      P(s1);
...         ...
V(s2);      V(s1);
V(s1);      V(s2);
```

Prevents deadlock

```
P1::        P2::
P(s1);      P(s1);
P(s2);      P(s2);
...         ...
V(s2);      V(s1);
V(s1);      V(s2);
```

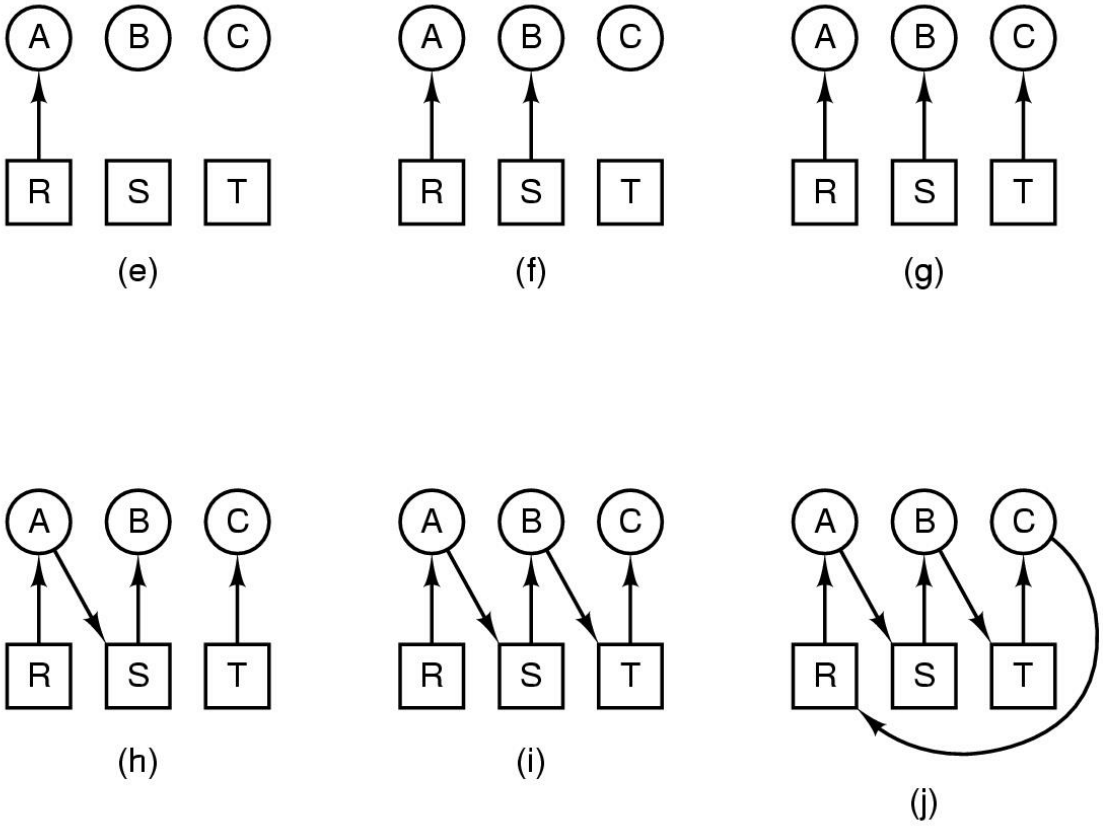
Deadlock Detection

- Graph models for deadlock detection
- If each resource has a single instance in a system, a *directed* graph model can directly tell you if there is a deadlock in the system
  - Called resource allocation graph
  - Resource R assigned to process A
  - Process B is requesting/waiting for resource S
  - Process C and D are in deadlock over resources T and U



Source: Modern Operating Systems by A. Tanenbaum

Deadlock detection with one resource of each type



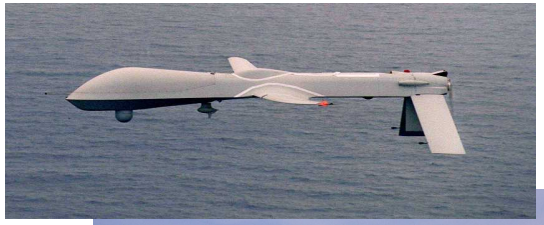
Source: Modern Operating Systems by A. Tanenbaum

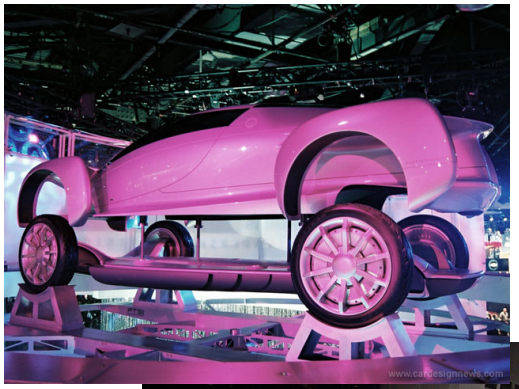
Real-Time Basics

- Real-time systems
  - Monitor, respond to, or control an external environment
  - Consist of sensors, actuators and other input-output interfaces
  - Reactive systems: respond to, or react to, signals from the environment
  - Subject to both temporal and logical constraints

- Examples of real-time systems
  - Transportation: automobiles, railways, subways, aircraft, ships, elevators
  - Traffic control: airspace, highways, shipping
  - Medical: pacemakers, radiation, patient monitoring
  - Military: command and control, missile defense, radar tracking
  - Manufacturing: automated plants
  - Any other examples?

Examples of Real-Time Systems





What Makes a Real-Time Program Different?

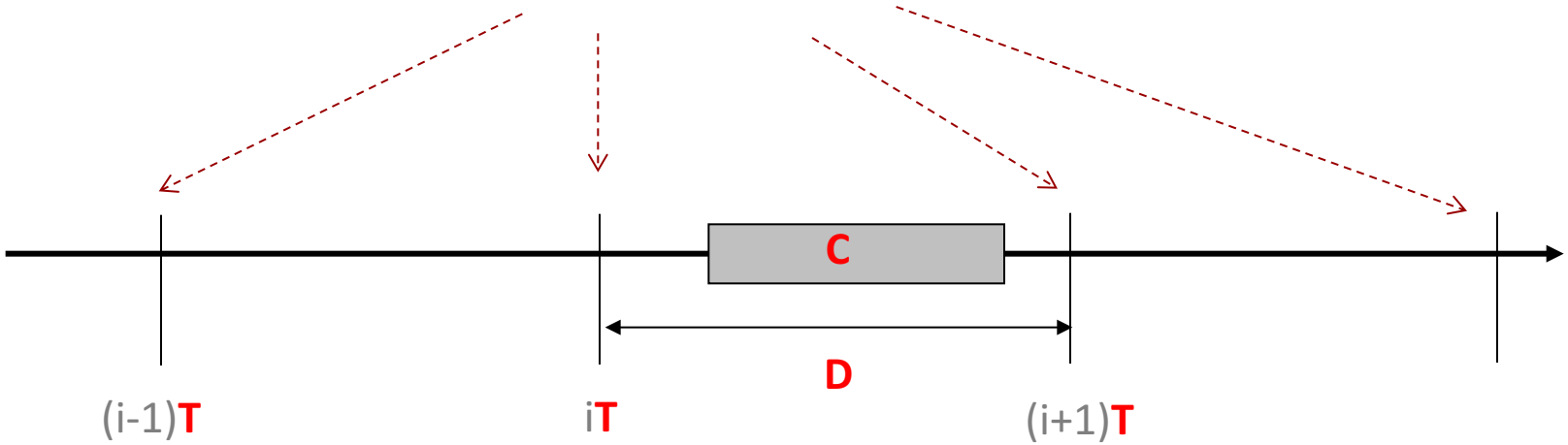
- What does real-time mean?
  - A real-time service is one that is required to be delivered within time intervals dictated by the environment
  - Temporal constraints are a part of the results' correctness criteria
- What makes a real-time program different?
  - Timing constraints
    - Must satisfy timing constraints involving relative and absolute times
      - Example: a deadline is a limit on the amount of time for completing an operation or a computation
  - Concurrency
    - Must deal with the natural concurrency of the physical world
      - Sensors can fire simultaneously
      - Environmental signals can arrive at the same time
    - Needs information about real-world time and multiple threads of execution

"Real-Time" Does Not Mean "Fast"

- "Real-time" does not mean "fast"
  - Real-time = meeting timing constraints
  - Fast = doing something quickly

- Real-time system can be slow
- Fast system can be non-real-time
- Of course, most real-time systems are designed to be fast and to operate at high speeds

Representation of a Real-Time Task



- Real-time periodic task represented as (C, T, D)
  C = worst-case computation/execution time of process/task P
  T = period or cycle time (how often the process/task P is activated)
  D = deadline for completing execution of process/task P

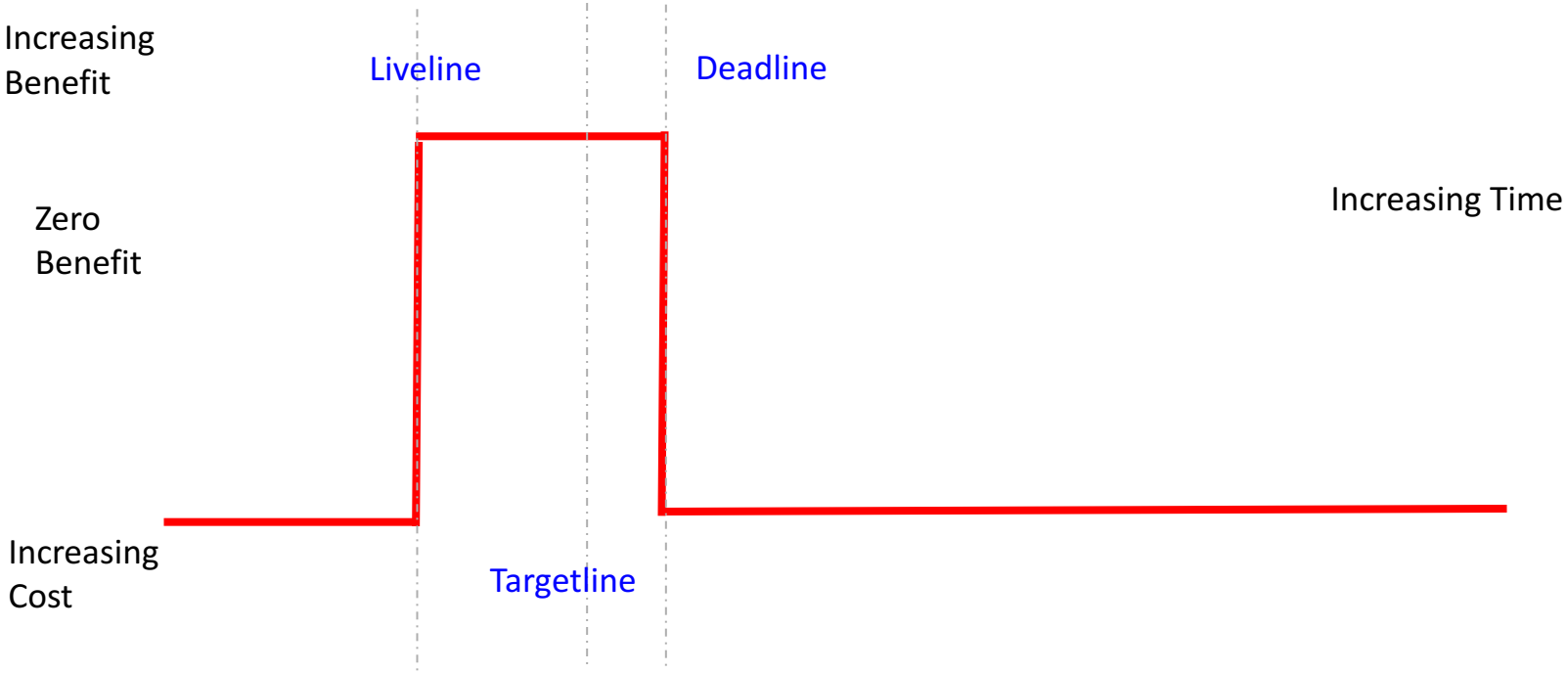
- Constraints
  $C <= D <= T$

Terminology

- Deadline
  - End of a time interval within which a real-time service is required to be delivered
- Liveline
  - Start of a time interval within which a real-time service is required to be delivered
- Targetline
  - Time at which the system designer aims to deliver the real-time service
- Priority
  - Measure of the cost of missing the timing constraints, or deadlines
  - Higher the cost of the task missing its deadline, the higher the task's priority

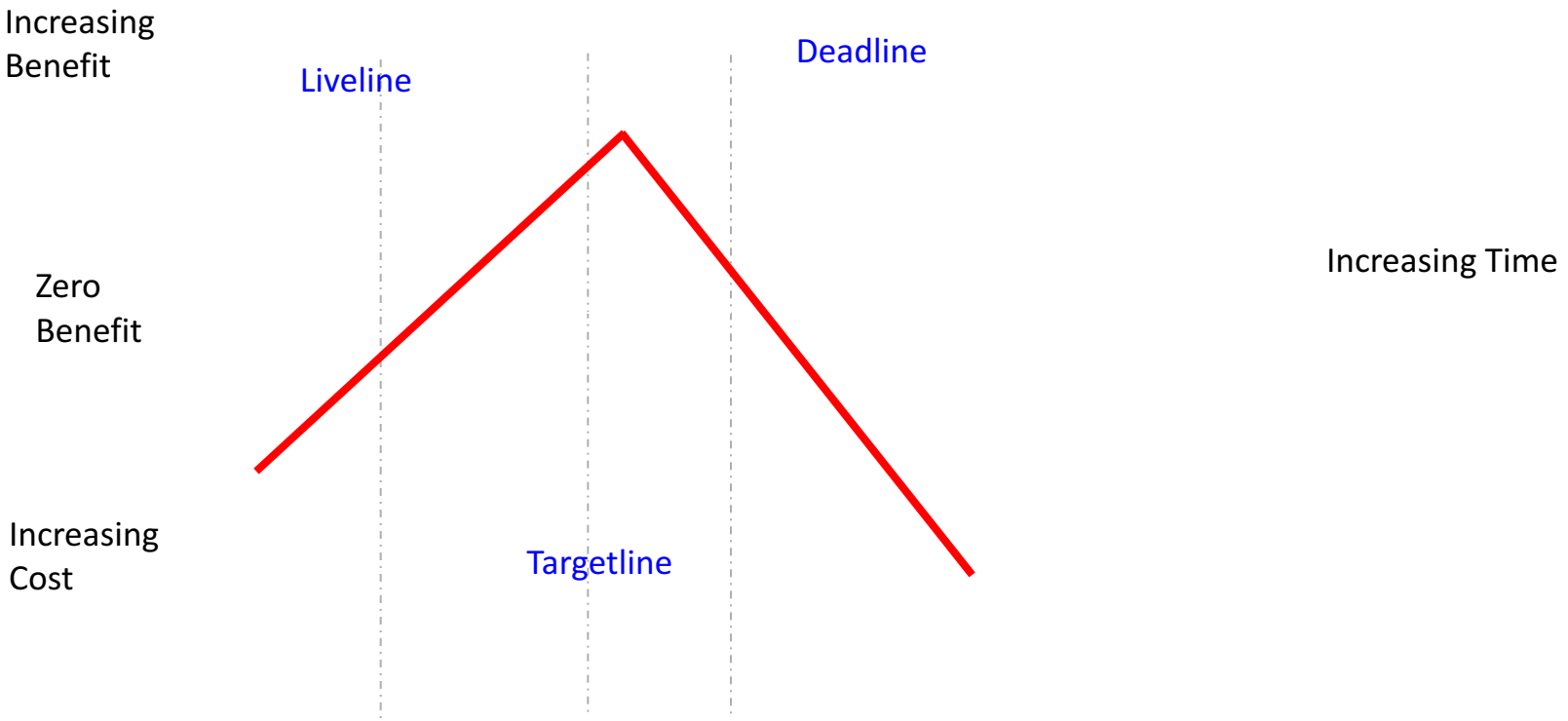
Hard Real-Time Systems

- Hard real-time systems
  - Must meet timing constraints without exception
  - Has zero/negative utility if delivered outside a certain time interval
  - If a timing constraint is violated, system fails, often with catastrophic consequences
  - Example: automobile braking system

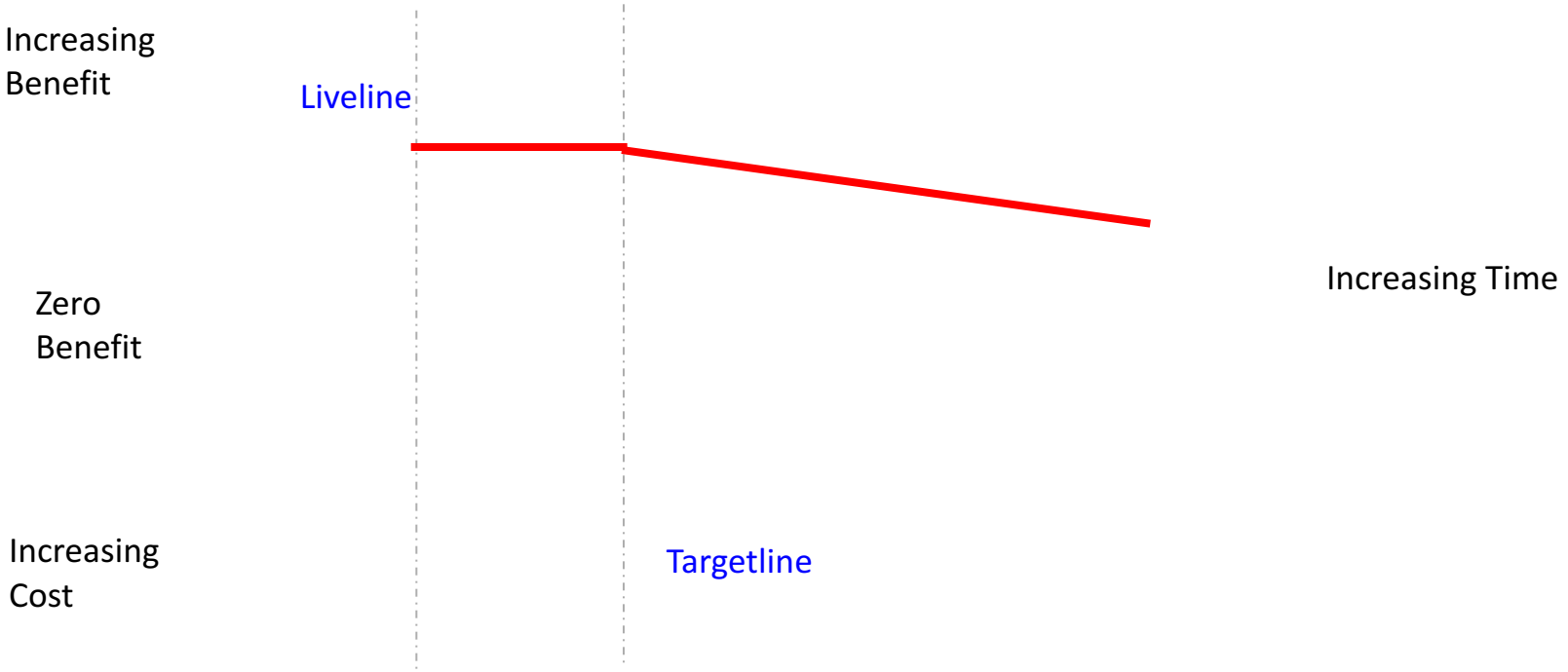


Soft Real-Time Systems

- Soft real-time systems
  - Can be considered successful despite missing some timing constraints
  - Has positive, but sub-optimal, utility if delivered outside a certain time interval
  - Might have zero/negative utility if delivered outside a wider time interval
  - Example: telephone system that occasionally fails to make a connection



Non Real-Time Systems



Kinds of Real-Time Tasks

- Periodic processes/tasks
  - Time-driven or synchronous
  - Activated on a regular basis between fixed time intervals
  - For periodic polling, monitoring or sampling of sensors
- Sporadic processes/tasks
  - Event-driven or asynchronous
  - Activated by an external entity or an environmental change
  - For events such as faults or mode changes
- Aperiodic processes/tasks
  - Event-driven or asynchronous
  - Multiple simultaneously or closely arriving events
  - For bursty events or bursty actions

Bounded-Demand Design

- Do not take the possibility of missing deadlines into account when designing the system because
  - A system that fails to meet its deadlines is not a real-time system

- Arrangements to tolerate missed deadlines are not interesting cases
- Design the system to assure that deadlines are always met
- System never goes out of its operational envelope
- How do you go about doing this?
- When is this a useful practice?
- When is this not feasible?

Unbounded-Demand Design

- Design the system to take into account the possibility of missed deadlines because
  - It is not always possible to guarantee that deadlines will be met because the system environment is not completely certain
- Arrangements to tolerate missed deadlines are interesting
  - Notion of graceful degradation – what happens when you miss a deadline?
  - Notion of behavior outside of the operational envelope
- When does this makes sense?
- How do you go about doing this?

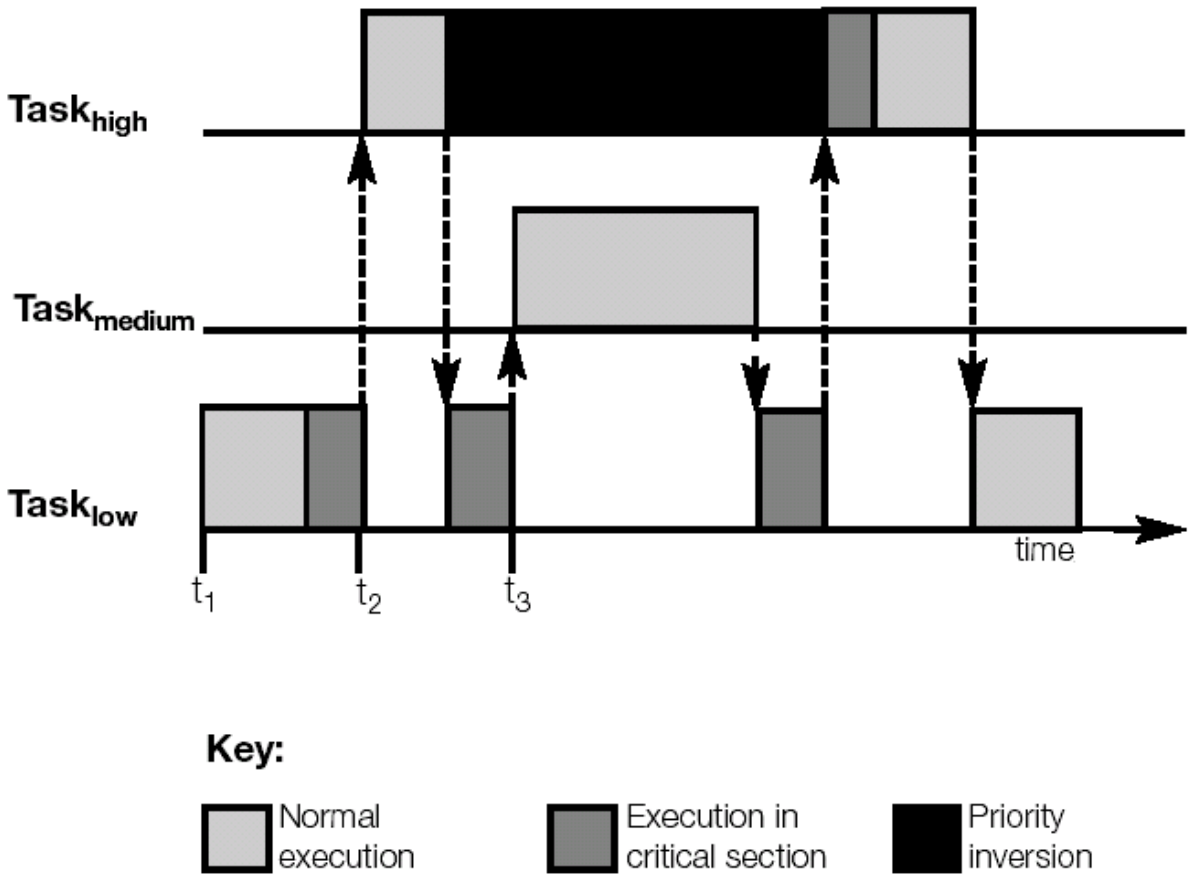
Bounded vs. Unbounded Demand

- To design for behavior outside the operational envelope
  - A larger operational envelope must be defined – how large?
- With bounded-demand, what do we do if the system does go outside the operational envelope, e.g., earthquake?
- Graceful degradation is probabilistic service with no tight guarantees
- Results of arguments
  - Bounded-demand model for hard real-time services
  - Unbounded-demand model for soft real-time services
  - Hard real-time tasks must be able to preempt the resources in a bounded time
  - Hard real-time tasks must be able to acquire I/O channels in a bounded time

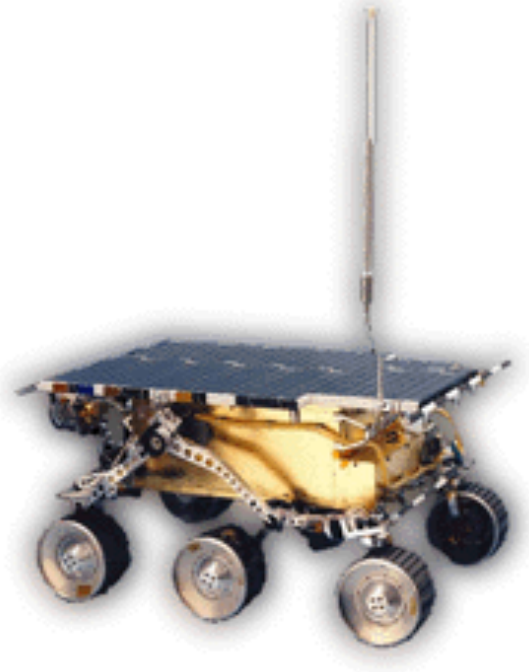
Priority Inversion

- Priority inversion
  - Delay of a higher-priority task's execution caused by interference from lower priority tasks
- Sources of priority inversion
  - Synchronization and mutual exclusion
  - Non-preemptable regions of code
  - FIFO (first-in-first-out) queues
- Ways to avoid priority inversion
  - Priority inheritance & priority ceiling protocols

Priority Inversion



How Real is Priority Inversion?



- Mars Pathfinder experienced total system resets
- Operating system used: WindRiver's VxWorks
  - Preemptive priority scheduling of threads
- Pathfinder's priority-based architecture
  - High-priority thread managed the information bus
  - Medium-priority thread ran a communications task
  - Low-priority data-gathering thread used bus to publish data
  - Bus access governed by mutex
- What happened?
  - High-pri task blocked, waiting for low-pri task to release mutex
  - Interrupt would occur, causing med-pri task to be scheduled
  - Watchdog timer would notice that high-pri task did not run for a while, and cause a total system restart

Why Are Deadlines Missed?

- For a given task, consider
  - Preemption: time waiting for higher priority tasks
  - Execution: time to do its own work
  - Blocking: time delayed by lower priority tasks

- The task is schedulable if the sum of its preemption, execution, and blocking is less than its deadline.

- **Focus**: identify the biggest hits among the three and reduce, as needed, for schedulability

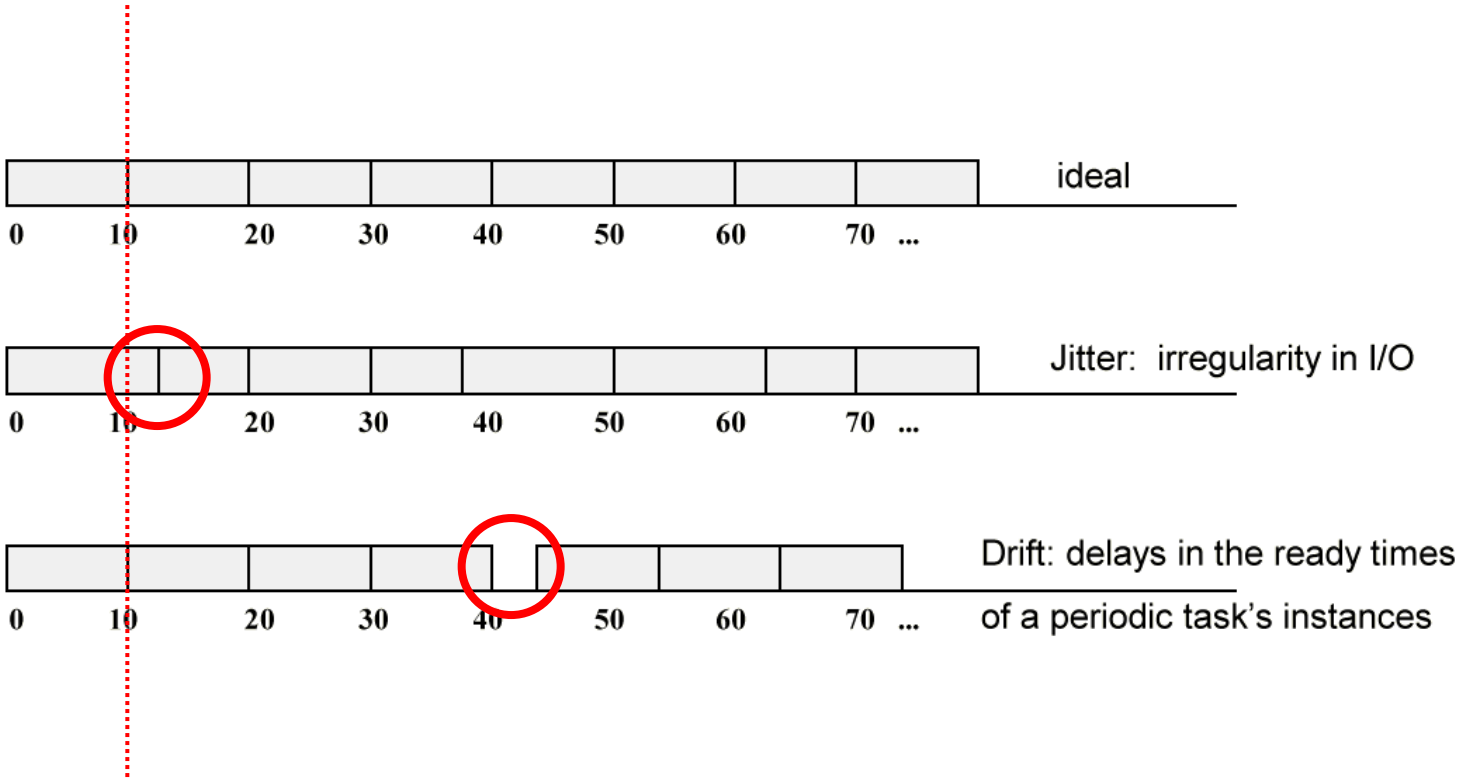
Metrics in Real-Time Systems – I

- End-to-end latency:
  - E.g. worst-case, average-case, variance, distribution
  - Can involve multiple hops (across nodes, links, switches and routers)
  - Behavior in the presence or absence of failures
- Jitter
  - Variability in metrics (e.g., variability in throughput)
- Throughput
  - How many requests can be processed in unit time?
  - How many messages can be transmitted in unit time?
- Robustness
  - How many faults can be tolerated before system failures?
  - What functionality gets compromised?

Metrics in Real-Time Systems – II

- Safety & Certification
  - Is the system "safe"?
  - Can the system get into an 'unsafe' state? Has it been 'certified'?
- Modes and reconfiguration
  - What happens when the system mission changes?
  - What happens under mode changes?
    - What are examples of mode changes?
  - What happens when individual elements fail?
  - Can the system reconfigure itself dynamically?
  - How does the system behave after re-configuration?
- Security
  - Can the system's integrity be compromised?
  - Can violations be detected?
  - Renewed interest in this area with appliances being connected to the Internet

Drift and Jitter



Drift can be eliminated completely but one can only hope to minimize jitter in general

Sources of Drift and Jitter

- What can cause drift?
- What can cause jitter?
- How would you prevent drift?
- How would you prevent jitter?

- If I gave you a piece of code and asked you to spot all of the places where jitter and drift could occur, how you would go about it?

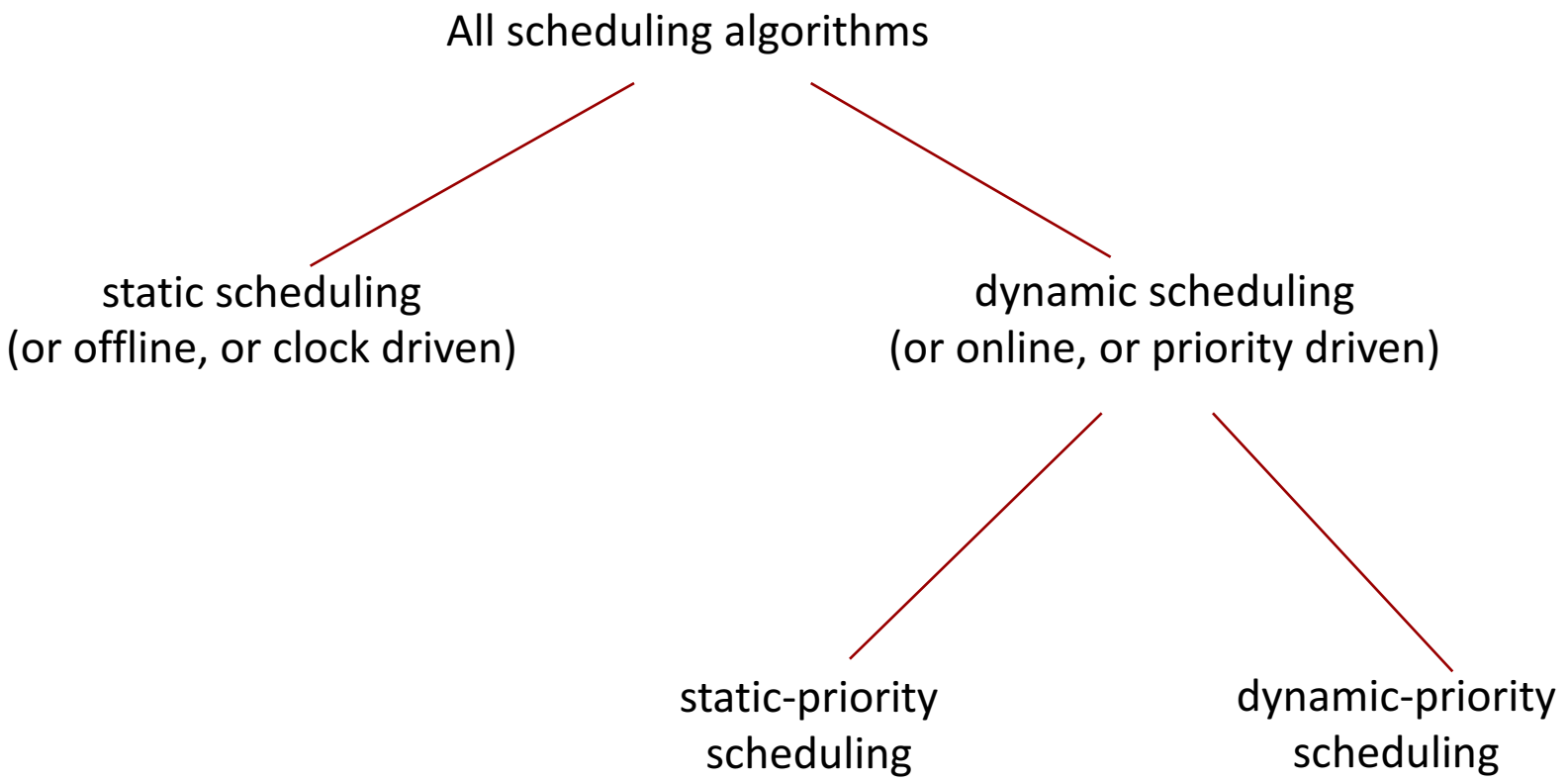
Scheduling

- Scheduler = resource allocator that affects the timing of real-time services
- Offline scheduler = does allocation at design time
- Online scheduler = does allocation at run-time
- Schedulers need to know dependencies and worst-case behavior

- Worst-case execution time (WCET) is often required to be known in real-time systems

- How do you determine worst-case behavior?

Classification of Scheduling Algorithms



Scheduling Algorithms (No Priority)

- Non-priority-based
  - All tasks are created equal
  - There is no way to indicate which tasks are more "important" (more critical) than others

- First-In-First-Out (FIFO) or First-Come-First-Served (FCFS)
  - Ready tasks are inserted into a list
  - Tasks are dispatched from the list in their order of entry in the list
  - No preemption, i.e., a task runs to completion before the next task runs
  - No consideration of task priorities

- Round-Robin Preemptive
  - Ready tasks are dispatched in turn
  - Each task given its fair share of fixed execution time
  - Preemption of running task at the end of the fixed interval
  - No consideration of task priorities

Scheduling Algorithms (Fixed-Priority)

- Fixed-priority based
  - All tasks are not created equal
  - Some tasks have more importance (higher priority) than others
  - Once a task's priority is assigned, it cannot change during run-time

- **Rate Monotonic Analysis (RMA)**
  - Shorter the period, the higher the priority of the task
  - Assigns fixed priorities in reverse order of period length
  - Tasks requiring frequent attention have higher priority and get scheduled earlier

- Least Compute Time (LCT)
  - Shorter the computation time, the higher the priority of the task
  - Assigns fixed priorities in reverse order of computation length
  - Tasks finishing quickly have higher priority and get scheduled earlier

Scheduling Algorithms (Dynamic-Priority)

- Dynamic-priority based
  - All tasks are not created equal
  - Some tasks have more importance (higher priority) than others
  - Task's priority (and thus, the schedule) may change during execution

- Shortest Completion Time (SCT)
  - Ready task with the smallest remaining compute time gets scheduled first

- Earliest Deadline First (EDF)
  - Ready task with the earliest future deadline gets scheduled first

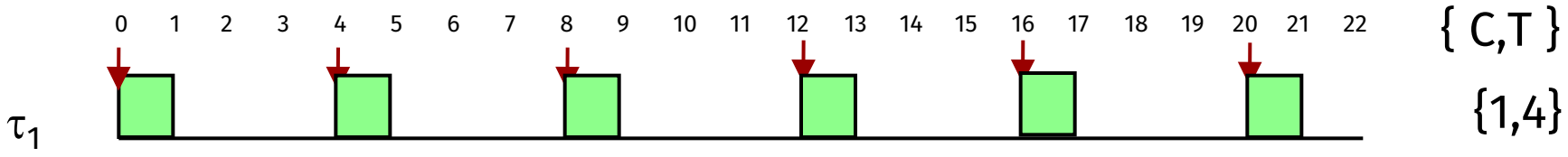
- Least Slack Time (LST)
  - Ready task with the smallest amount of free/slack time within the cycle gets scheduled first

Real-Time Standards

- Real-Time Operating System standards
  - IEEE 1003.1b POSIX Real-Time Extensions
  - OSEK (automotive real-time OS standard)
- Real-Time (and Concurrent) Programming Languages
  - Real-Time Specification for Java
  - Ada 83 and Ada 95
- Real-Time Middleware
  - Real-Time CORBA
- Networks/buses
  - CANbus (Controller Area Network bus)
  - TTA: Time-Triggered Architecture (www.tttech.com)
  - FlexRay (www.flexray.org)

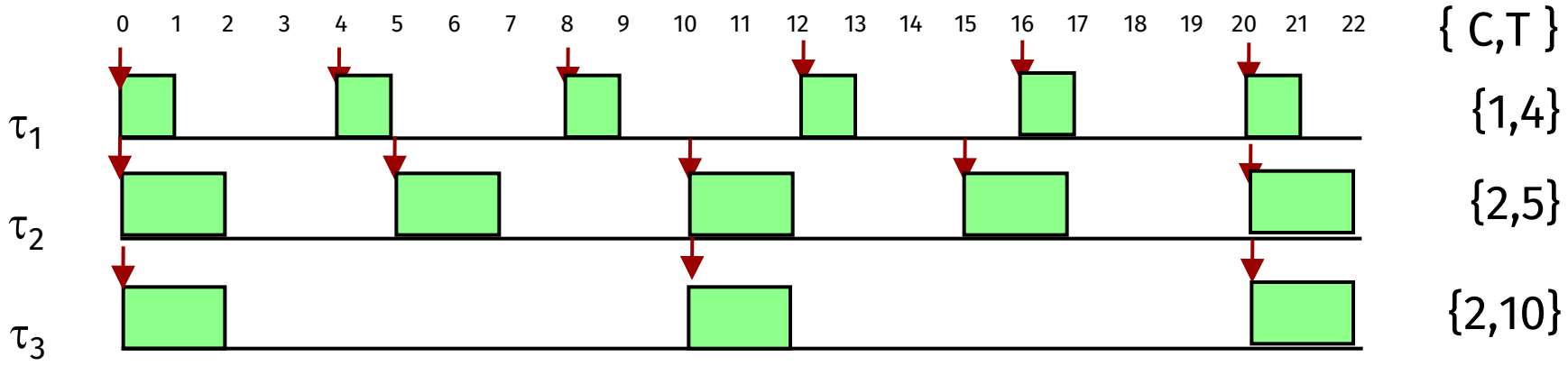
Rate Monotonic Scheduling

- A task with a shorter period has a higher deadline
  - Shorter Period -> Higher priority



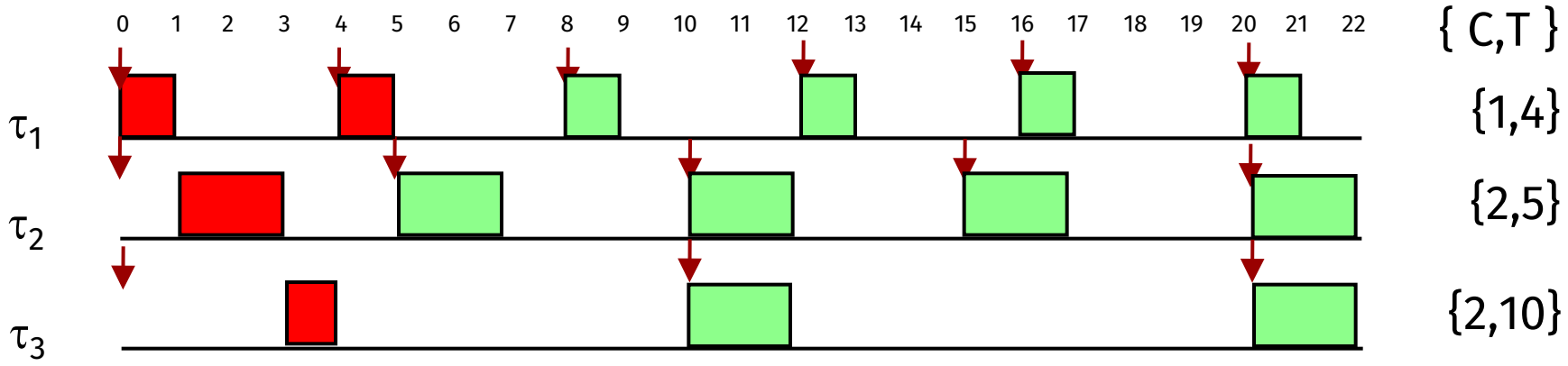
Rate Monotonic Scheduling

- A task with a shorter period has a higher deadline
  - Shorter Period -> Higher priority



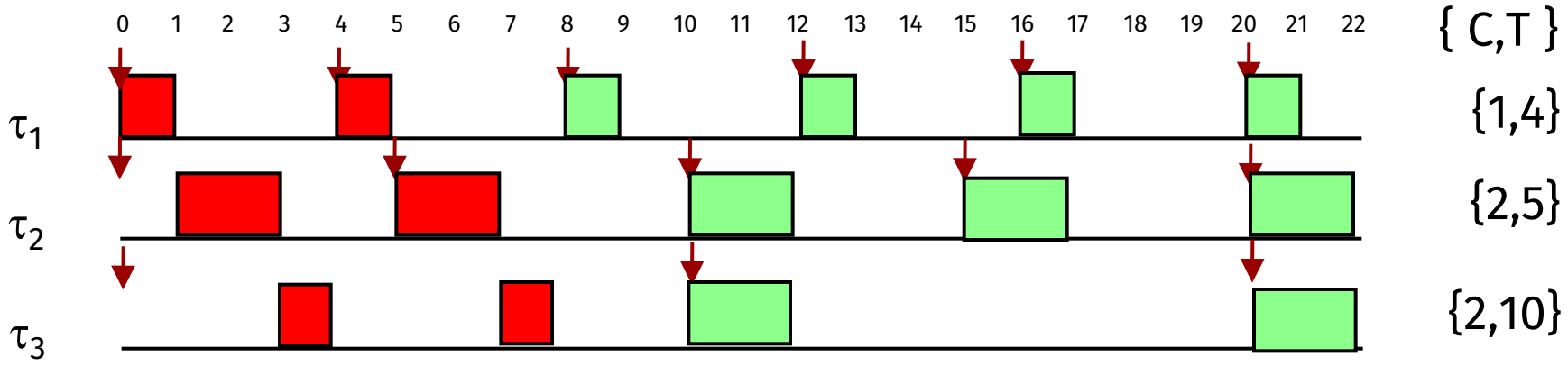
Rate Monotonic Scheduling

- Optimal Static Priority Scheduling
- A task with a shorter period has a higher deadline
  - Shorter Period -> Higher priority



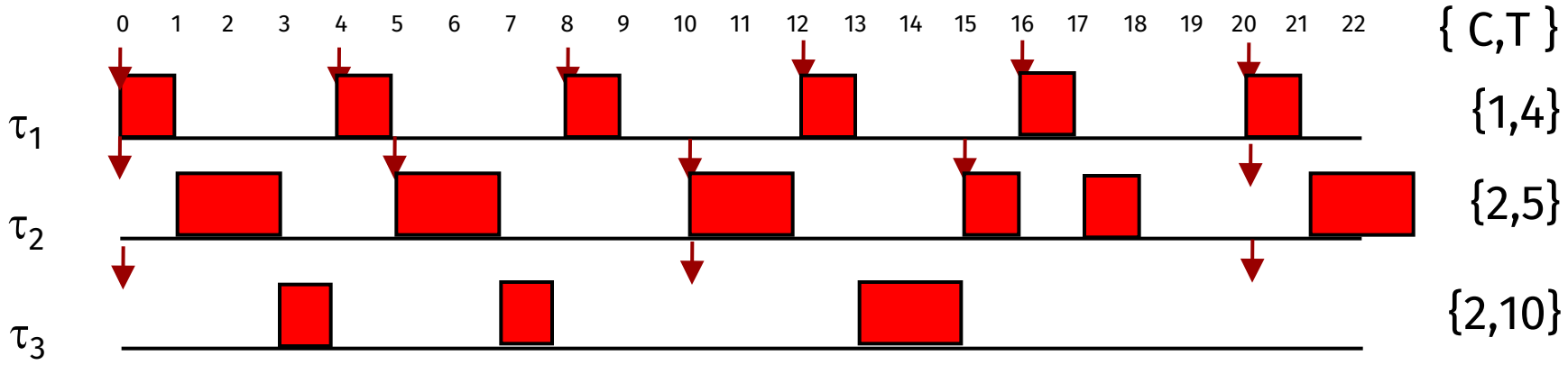
Rate Monotonic Scheduling

- Optimal Static Priority Scheduling
- A task with a shorter period has a higher deadline
  - Shorter Period -> Higher priority



Rate Monotonic Scheduling

- Optimal Static Priority Scheduling
- A task with a shorter period has a higher deadline
  - Shorter Period -> Higher priority



Summary

- Example real-time systems
  - Simple control systems, multi-rate control systems, hierarchical control systems, signal processing systems
- Terminology
- Priority inversion
- Scheduling algorithms (Overview)

- **Next Up:** Rate Monotonic Scheduling and Analysis