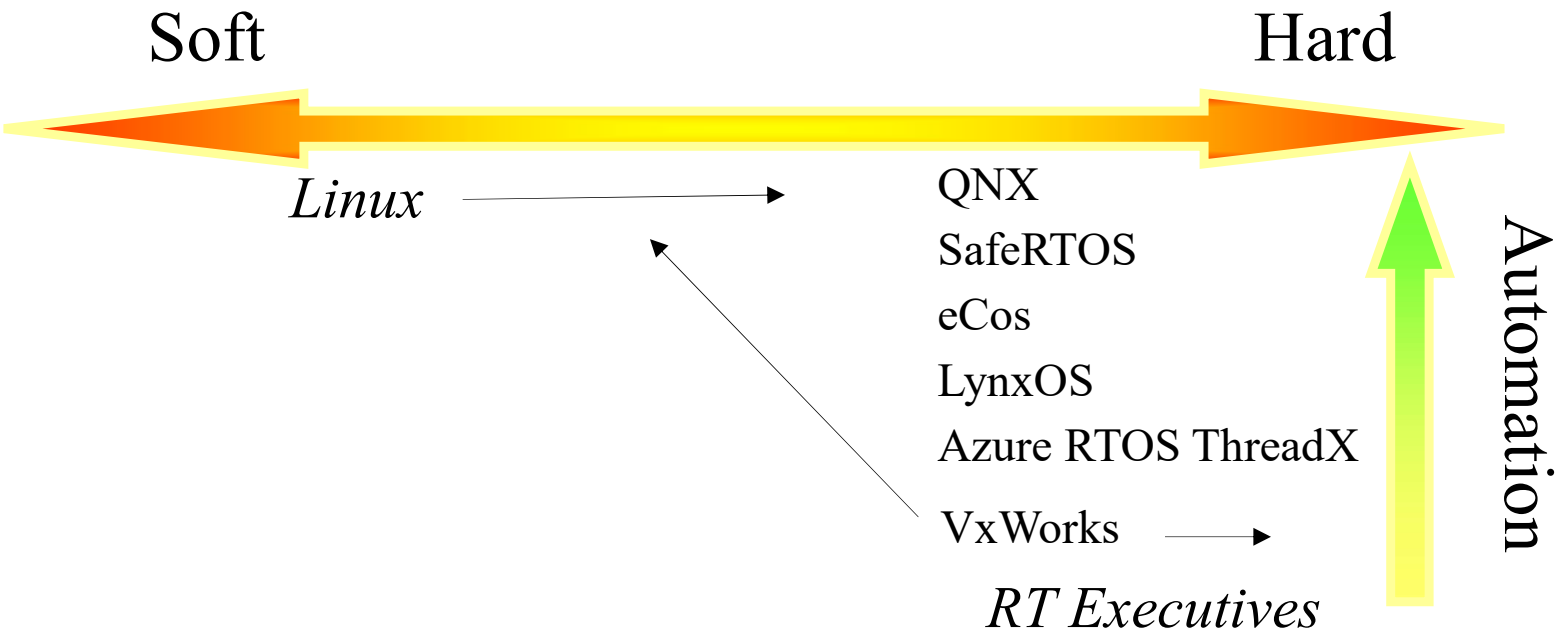
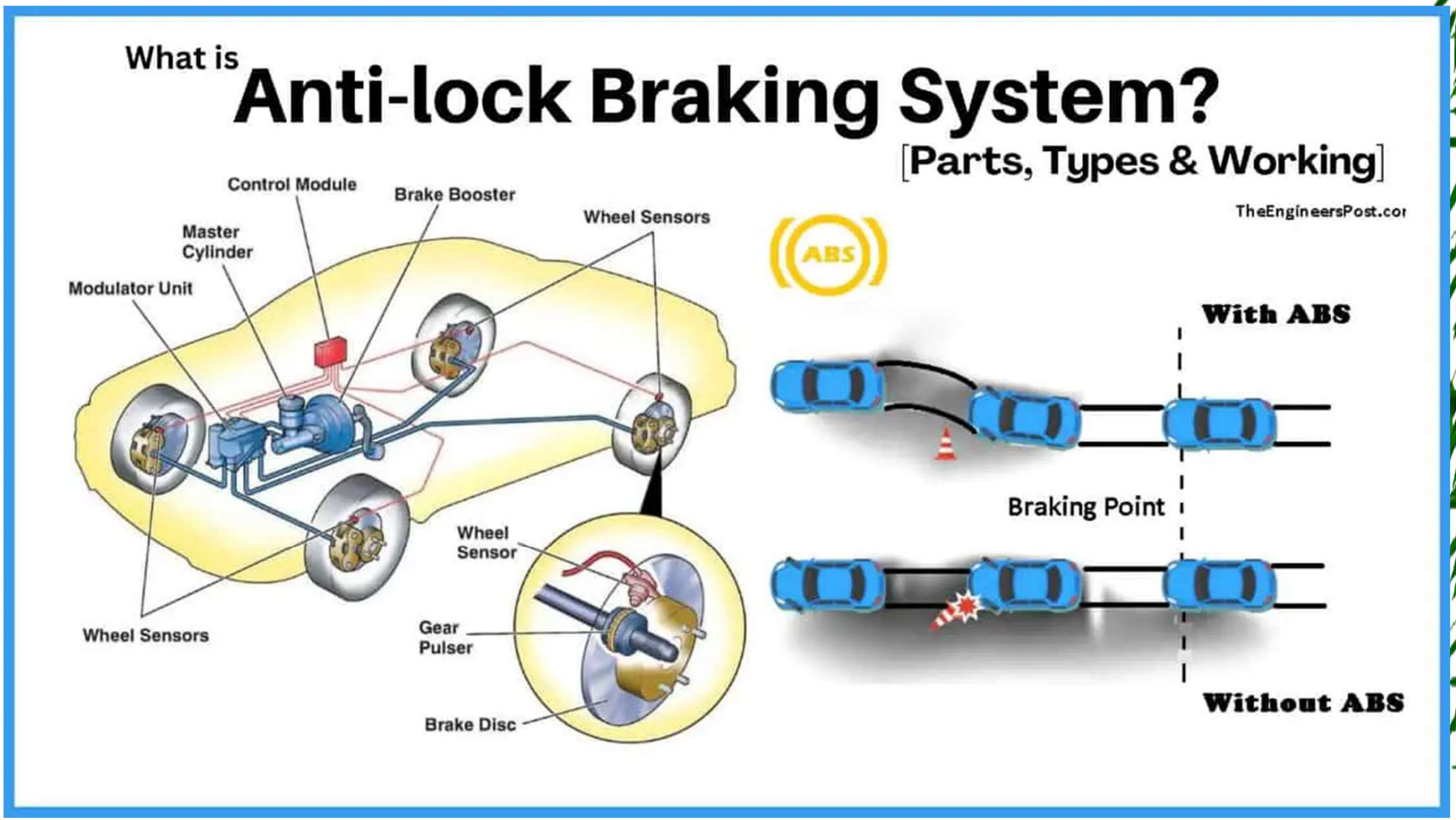
Why Real-Time?

Why Real-Time?

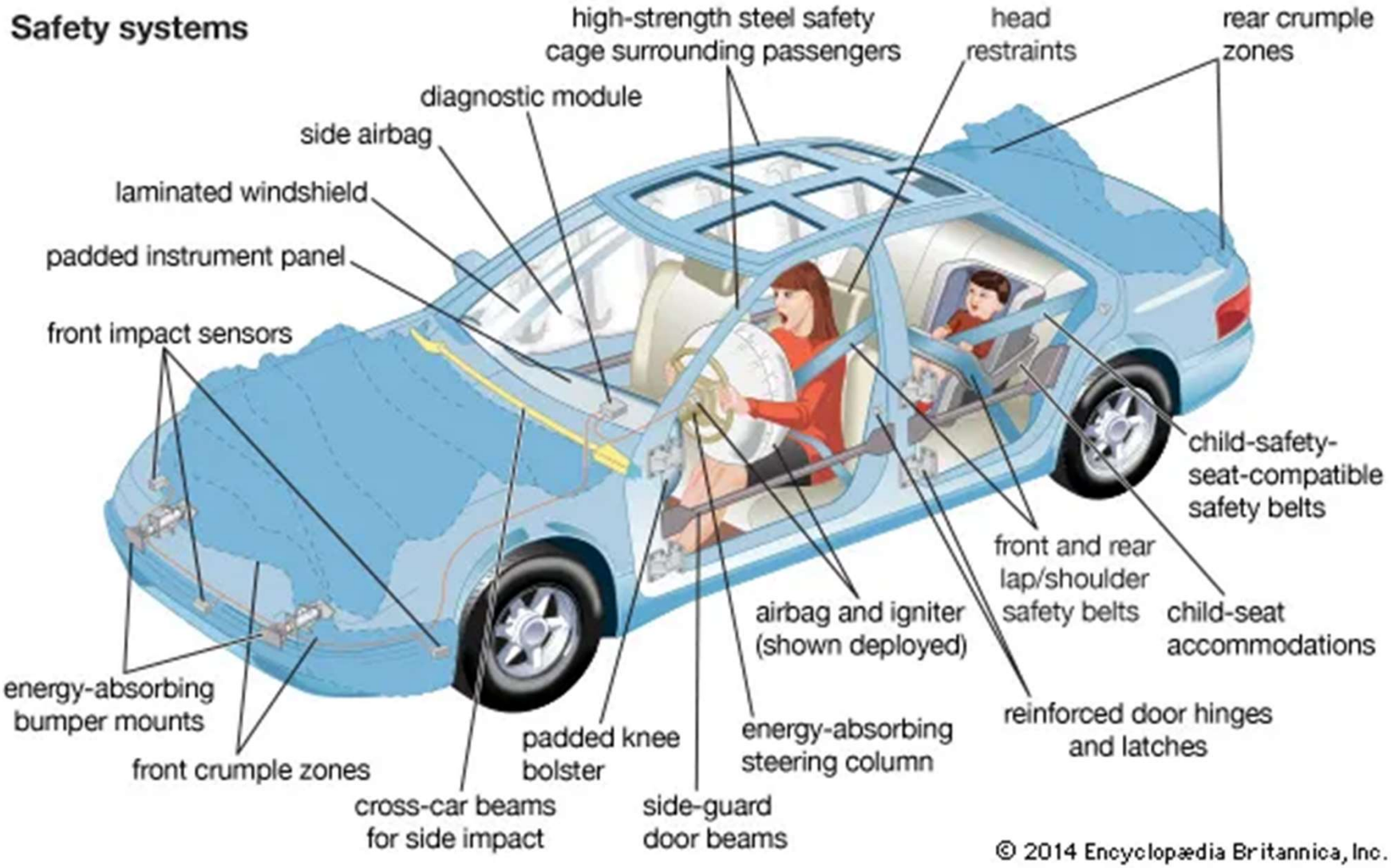


- Many applications require "real-time" response
  - Control, e.g., FTSE index, medical lasers, heart pacemaker, car/airplane engine/flight control (hard constraints, missing deadline is not an option)
  - Entertainment, e.g., video/chess game without real time constraint (soft)
- Real-Time requirement imposes
  - predictable timing (re-entrant code including library functions, avoid variable latency operations, e.g., caching, paging)
  - responsive constraints (fast interrupt handlers)

Hard Real-Time Examples

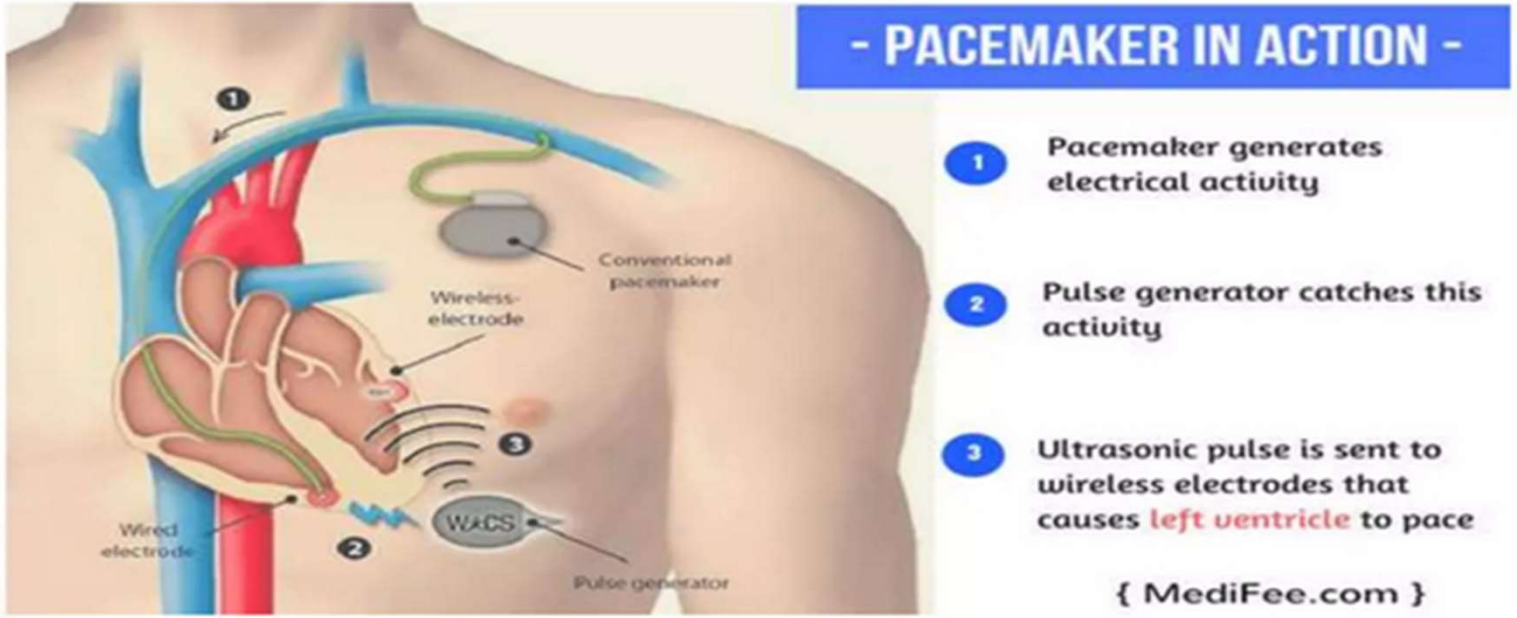


Hard Real-Time Examples



Hard Real-Time Examples

- **(1) A pacemaker.** This is a device that is inserted in someone's chest to provide electrical impulses at regular intervals to help their heart beat. If the device missed a beat, it may be fatal.



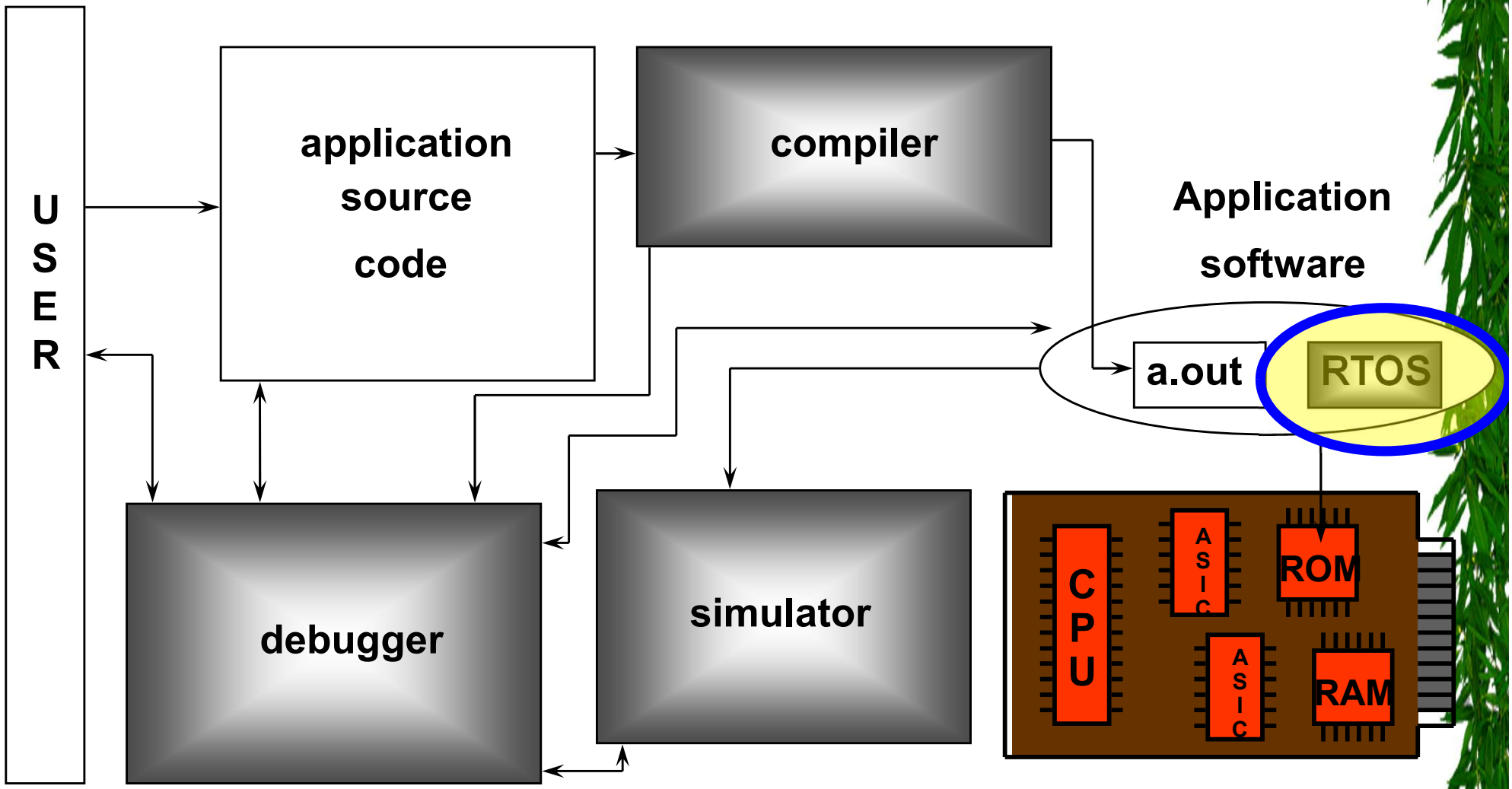
Hard Real-Time Examples

- (4) missile weapons defense system.

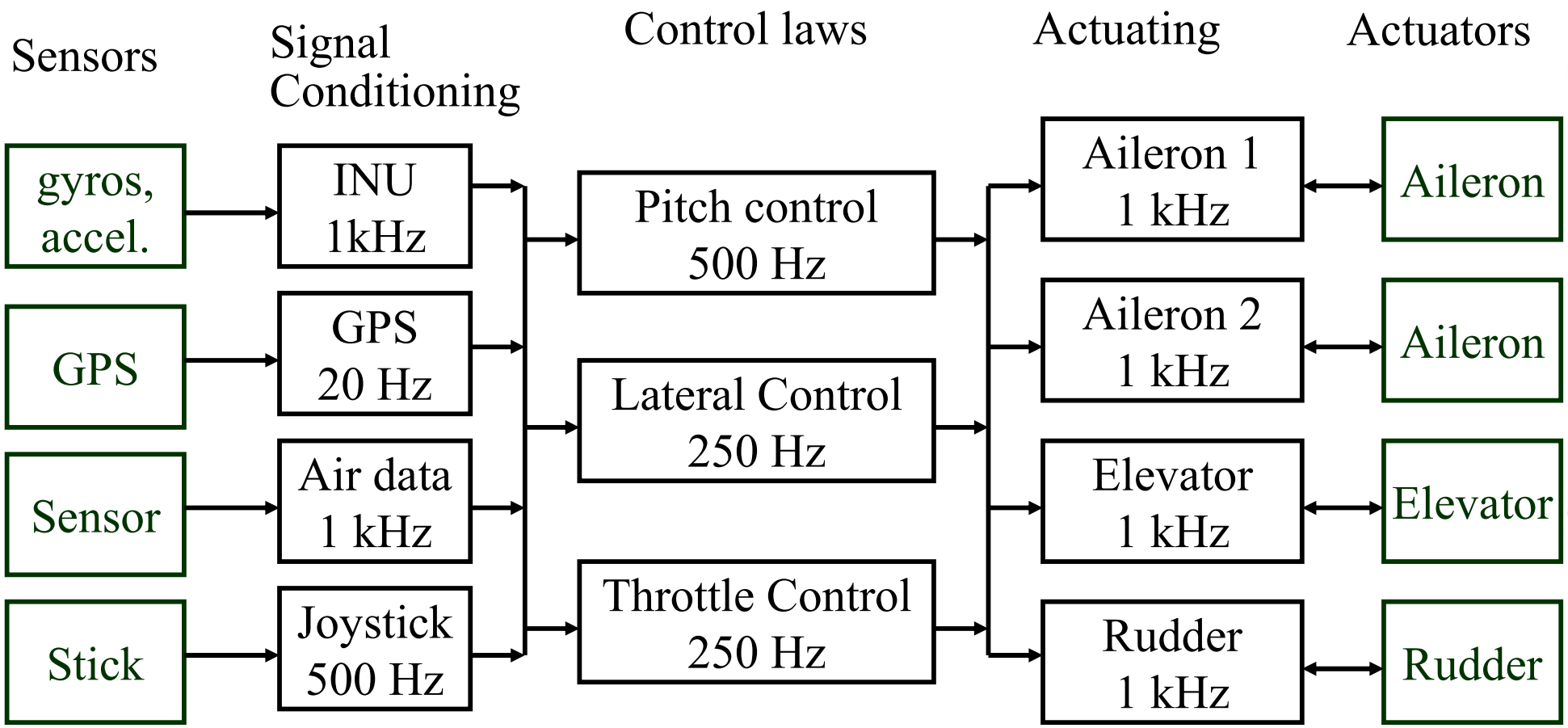
Real-Time Applications

- Complex RT systems (computation & communication)
- Original Linux designed for throughput, not response
  - kernel not preemptable, new efforts (SCHED_DEADLINE policy)
  - kernel disables interrupts
- Soft Real-Time or Hard Real-Time ⇒ determinism
  - Scheduling response time
  - Scheduling jitter
  - Interrupt response time

Example: Embedded Software

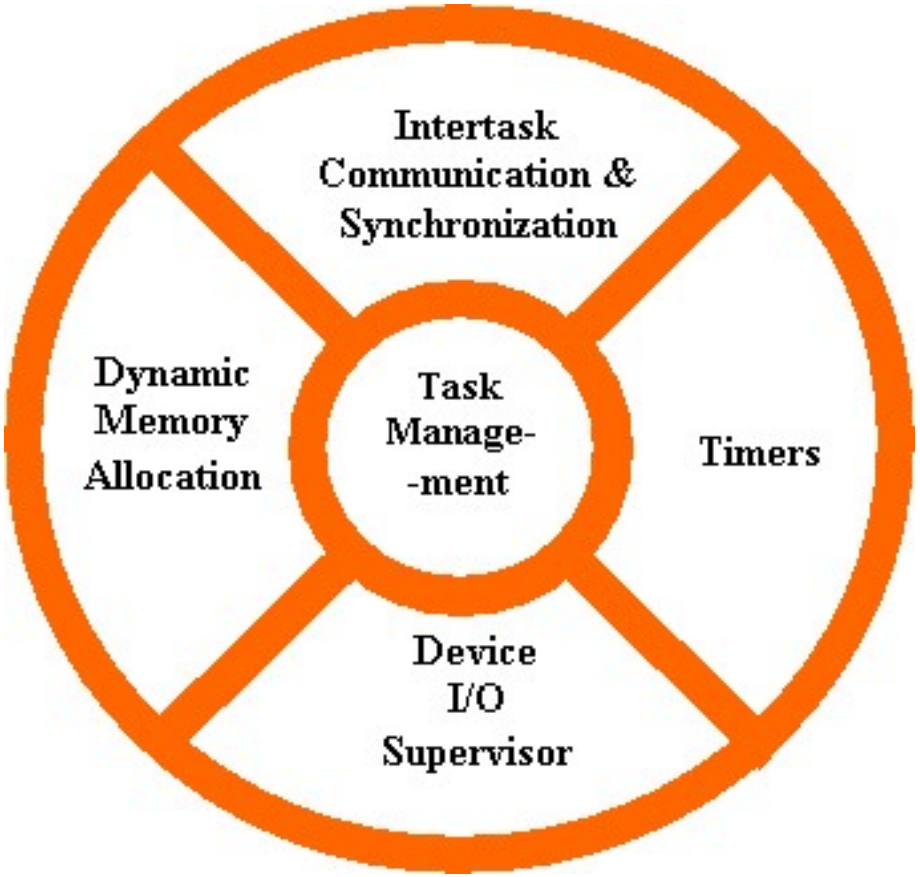


Example: Fly-by-wire Avionics



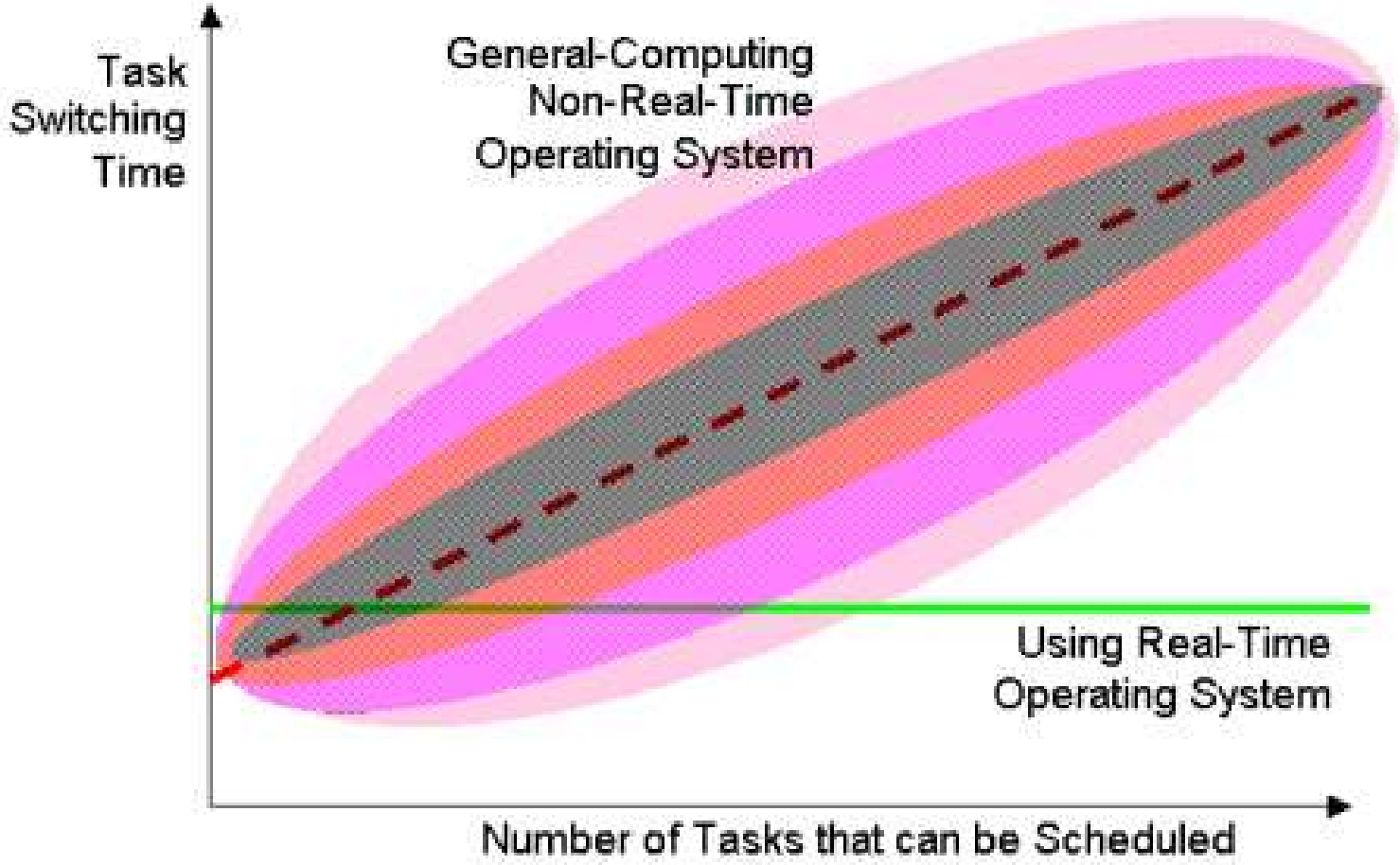
- Hard real-time system with multirate behavior

Real-Time Operating Systems (RTOS) -Services



- Since many embedded systems are real-time (upper bound on execution time), the OS is a **real-time operating system (RTOS)** performing
  - management of timers & shared resources (IO, memory, CPU and task)
  - inter-task communication and synchronization, interrupts, QoS
- Protection mechanisms not always necessary

Task Switching Time: RTOS vs. OS



RTOS Classification

- Depending on how timing constraints are supported, current real-time operating systems are distinguished 3 main categories (R. Gupta):
  - ✓ **Fast fixed-priority kernels**: real-time clock, high performance (average response time) based on priorities (VxWorks, QNX)
  - ✓ **Real-time extensions of time-sharing OS**: improved real-time clock and timing constraints into task scheduling, (RTLinux, RTAI, etc)
  - ✓ **Hard real-time OS**: incorporate bounded execution time, communication and scheduling costs, clock interrupts

RTOS: Task States

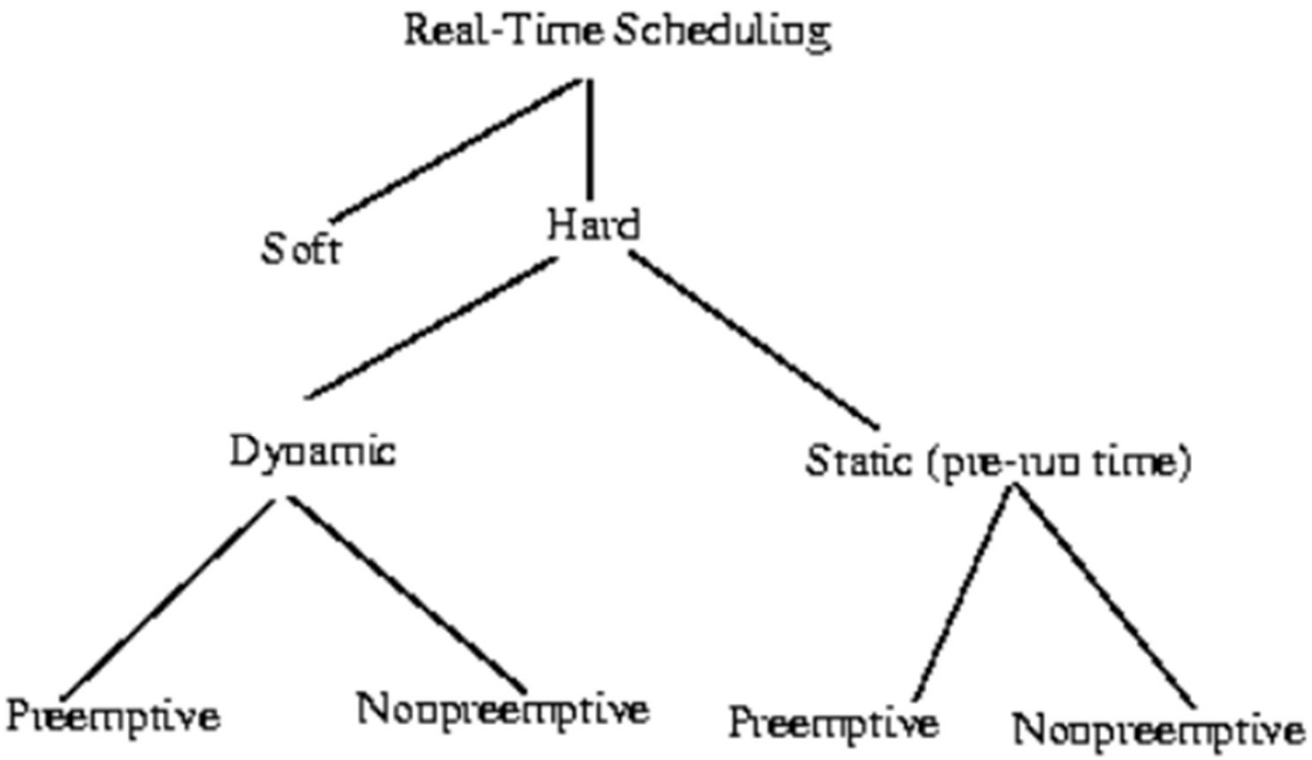
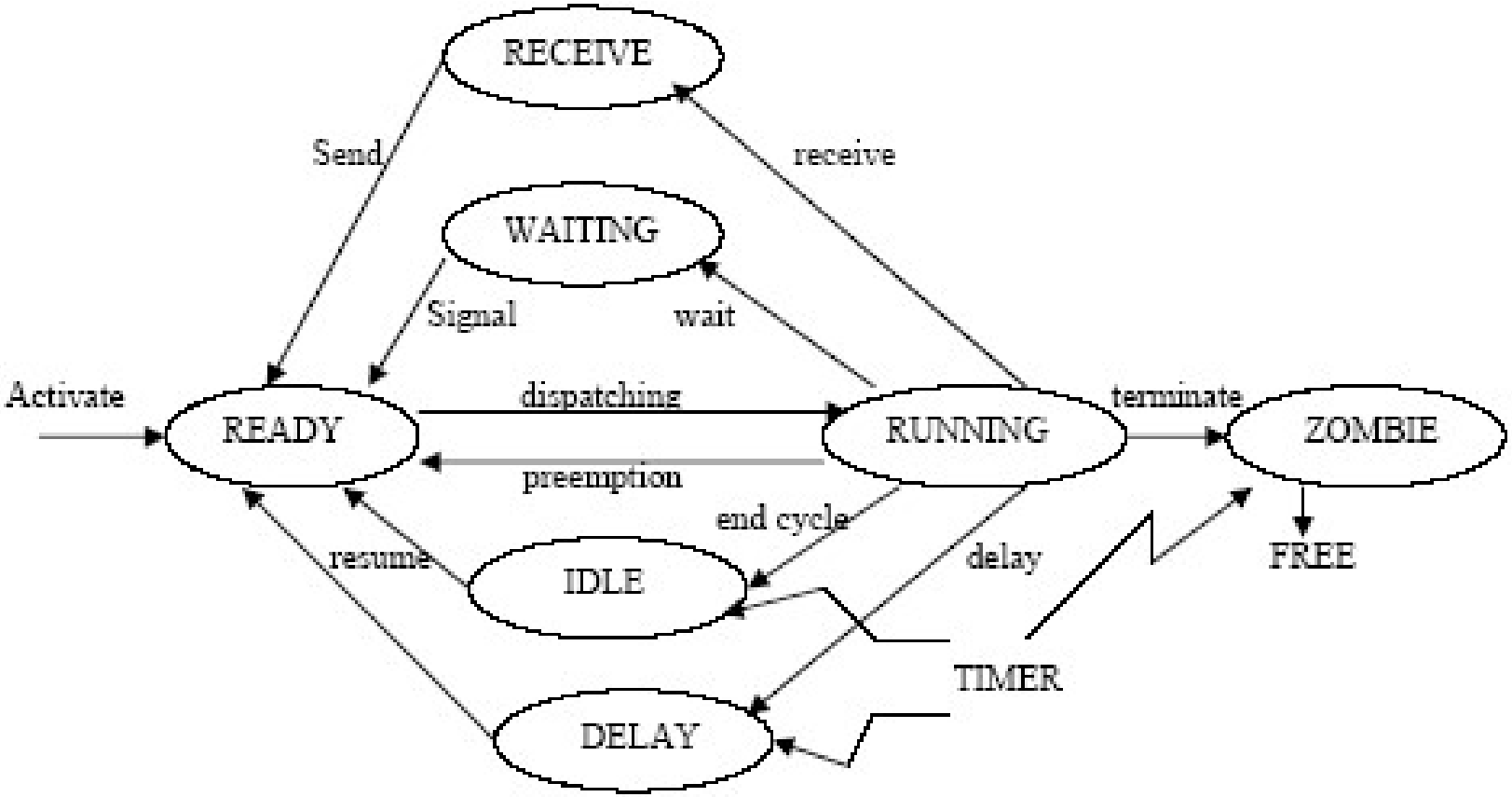


Fig 2: Taxonomy of Real-Time Scheduling

RTOS: Task States



RTOS: Task States

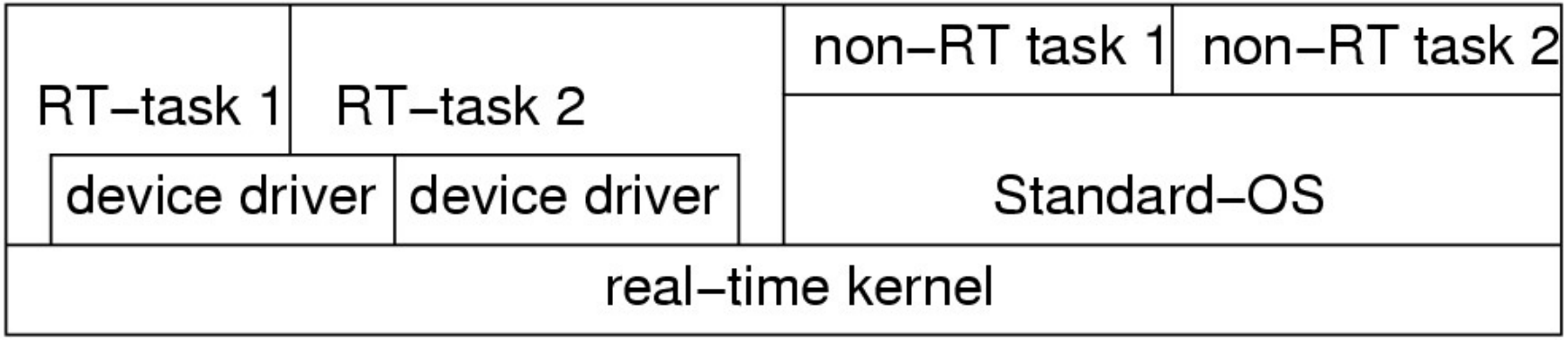
- RUNNING. A task enters this state as it starts executing on the processor.
- READY. This is the state of those tasks that are ready to execute but cannot be executed because the processor is assigned to another task.
- WAITING. A task can enter this state when it executes a synchronization primitive to wait for an event or on a busy resource. When the task is resumed by a signal free resource or an event it is inserted in the *ready* queue.
- IDLE. A periodic task enters this state when it completes its execution and has to wait for the beginning of the next period. At the right time, each job in the IDLE state will be awakened by the operating system and inserted in the ready queue.
- DELAY. A *delay* primitive which suspends a task for a given interval of time, puts the task in a sleeping state (DELAY), until the timer awakens it after the elapsed interval.
- RECEIVE. A task enters this state when it executes a *receive* primitive on an empty channel of a classical message passing mechanism. The job exits this state when a *send* primitive is executed by another task on the same channel.
- ZOMBIE. In real-time systems that support dynamic creation and termination of hard periodic tasks, a new state needs to be introduced for preserving the bandwidth assigned to the guaranteed tasks.. This problem arises because when a periodic task is aborted, its utilization factor cannot be immediately subtracted from the total processor load since the task could already have delayed the execution of other tasks. In order to keep the guarantee test consistent, its utilization factor can be subtracted only at the end of its period. In the interval of time between the abort operation and the end of its period, the task is said to be in ZOMBIE state since it does not exist in the system, but it continues to occupy processor bandwidth.

RTOS Parameters

Table 1. List of example RTOS parameters that affect the timing of RTES

• Scheduler: Scheduling policy, associated timing overhead, support for task preemption, scheduler triggering events (e.g., blocking of a task, timer ISR, etc.) • Priority inversion resolution protocols: Priority inheritance protocol, priority ceiling protocol, disabling preemption during access to shared data [4]
• Timer granularity
• Preemptability & preemption points of kernel services • Kernel thread priority
• Set of system calls and their duration • Task synchronization primitives (e.g., lock, semaphore, mutex, etc.) • Set of IPC facilities (message passing, barrier, etc) and their characteristics (i.e., blocking / non-blocking)
• Set of ISRs (Timer ISR, I/O ISR) and their duration • Effect of splitting I/O interrupt into *top/bottom half*

How to make Linux a real-time OS?

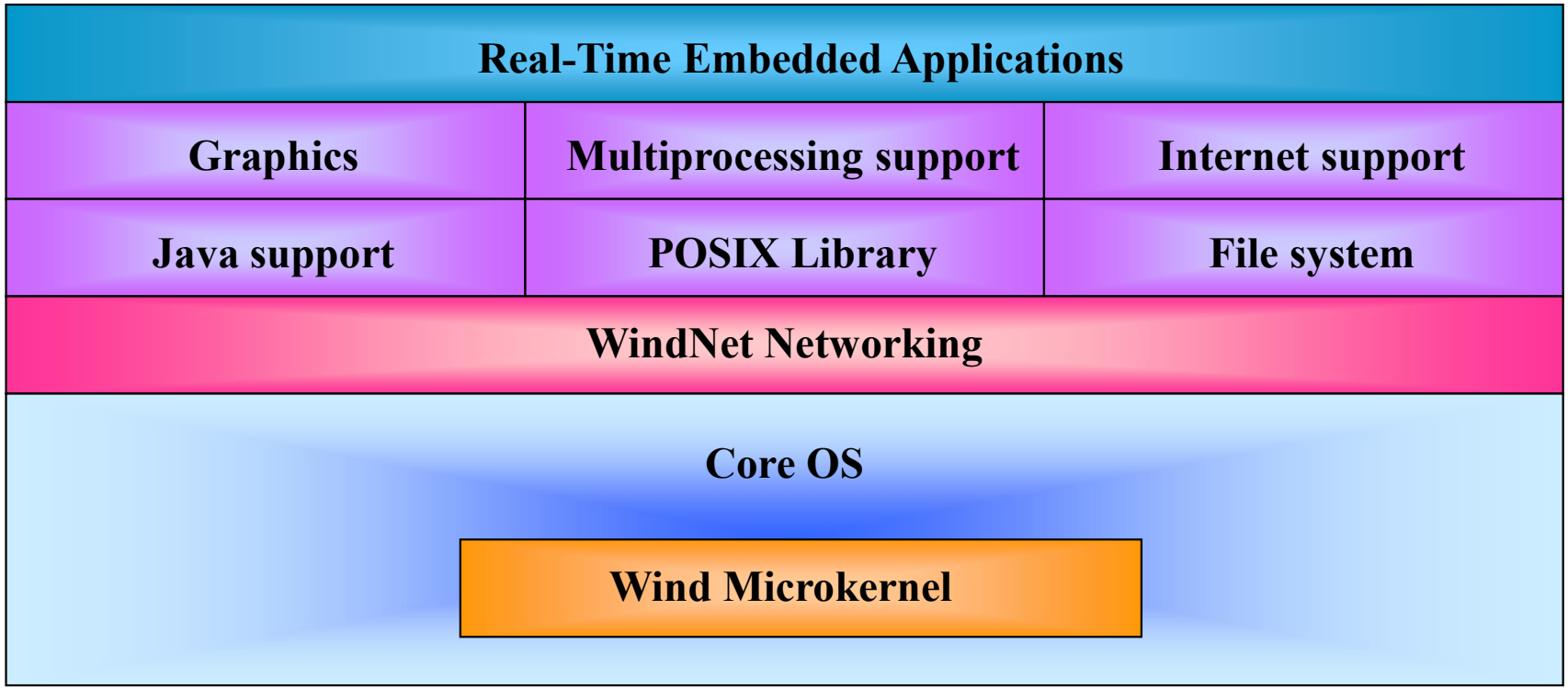


- Real-time extension to Linux
  - O(1) RT scheduler (v2.4.19)
  - Complex preemption kernel capable of meeting requirements for a soft real-time system (v2.5.4)
  - POSIX.1b compliance
- Priority inheritance inside kernel (TimeSys)
- Low-latency patch (Molnar & Morton)
  - Insert reschedule points inside kernel; need cooperation
- Micro-kernel: Linux sits on another RTOS; real-time code external

Outline on RTOS

- Wind River Systems Inc. VxWorks - http://www.wrs.com
  - System
  - Kernel
  - Supported processors
  - Closely coupled multiprocessor support
  - Loosely coupled multiprocessor support
  - Custom hardware support
- Task management
  - multitasking, unlimited number of tasks
  - preemptive and round-robin (static) scheduling
  - fast, deterministic context switch
  - inter-task communication (message queue, shared memory, control sockets, POSIX pipes, semaphore)
  - Fast, efficient interrupt and exception handling
  - 256 priority levels

VxWorks



VxWorks 5.4 Scalable Run-Time System

Task States

- Ready State: waiting in ready queue
- Running State: CPU executing the task
- Blocked: waiting in the semaphore queue until shared resource is free

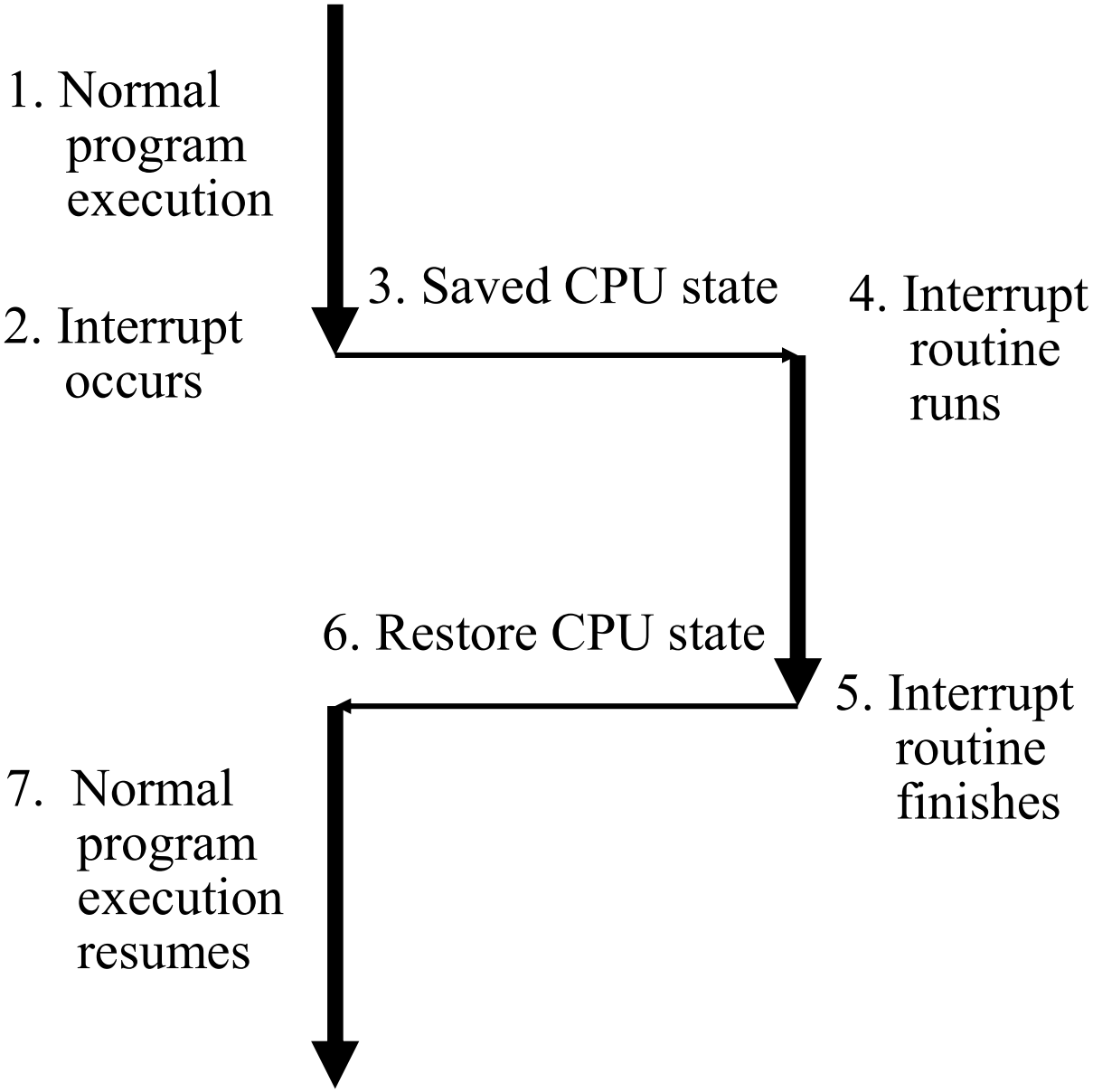
Do I Need an RTOS?

- Not always. **Cyclic executive plus interrupt routines (**good for DSP)

```
loop
  do part of task 1
  do part of task 2
  do part of task 3
end loop
```

- Execute user-specified instruction upon interrupt to avoid loop iterations, e.g. copy peripheral data into a buffer, if "byte arrived" on serial channel
- Advantages
  - Simple, cheap implementation
  - Low overhead, sometimes direct hardware support
  - Predictable interrupt handler (no context switch, no environment save)
- Disadvantages
  - Can't handle sporadic events, since everything operates in lockstep
  - Code must be scheduled manually

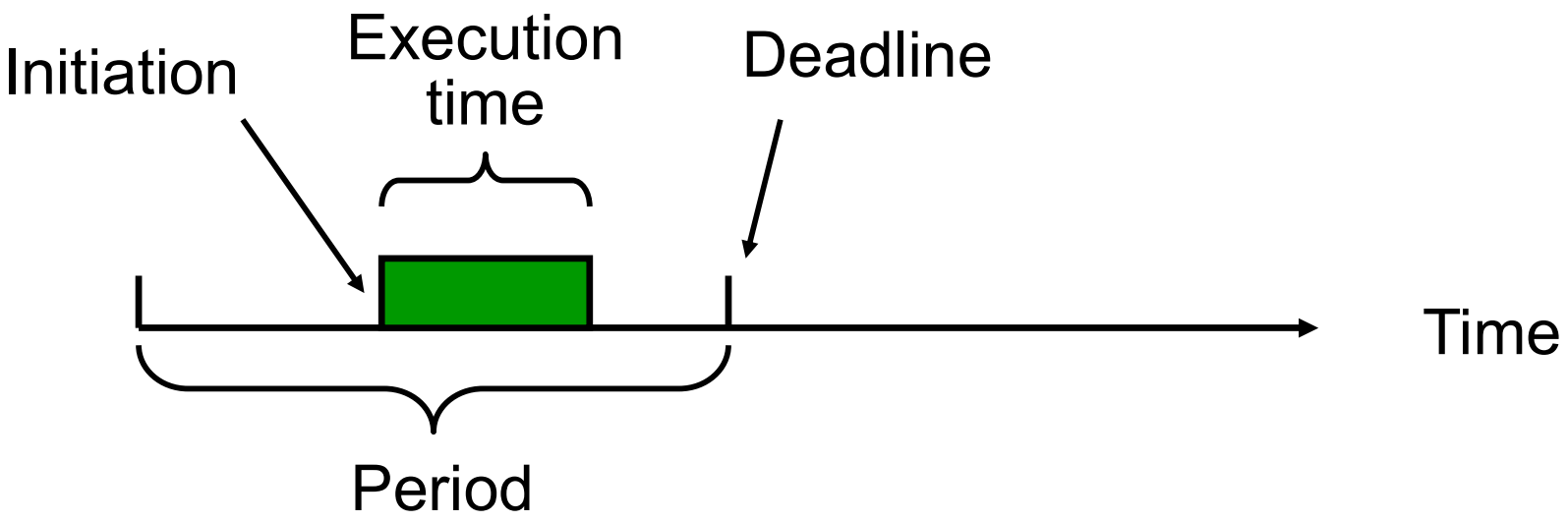
Handling an Interrupt



Real-Time Issues

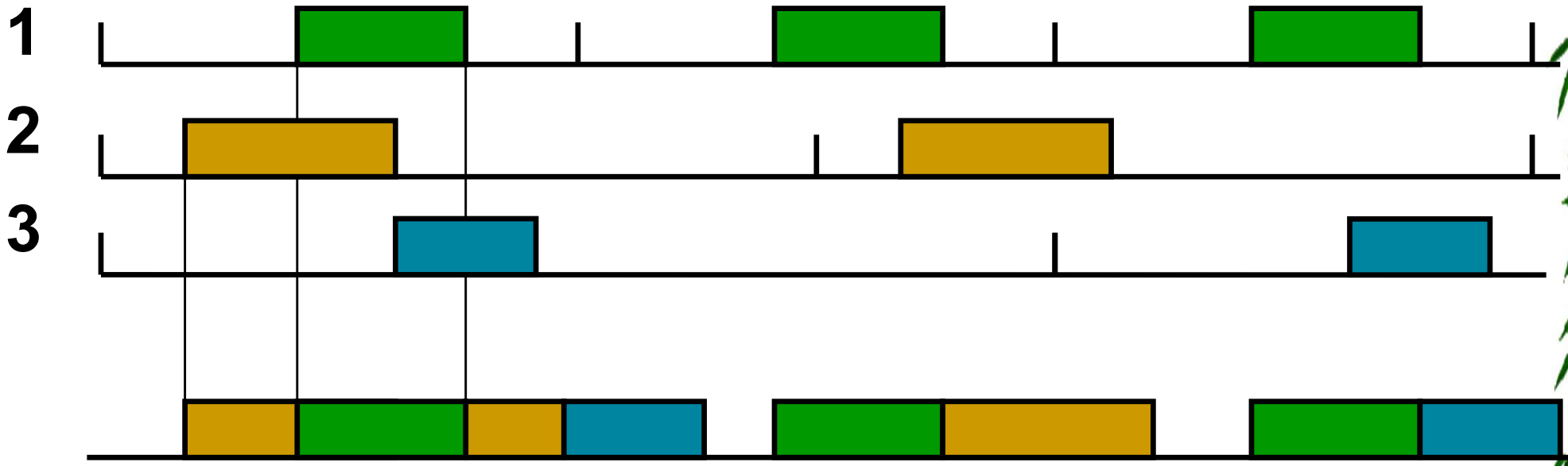
- The main goal of an RTOS scheduler is meeting deadlines
- Fairness concept (from time-sharing OS schedulers) does not help you meet deadlines
- Priority-based scheduling:
  - Typical RTOS based on fixed-priority preemptive scheduler
  - Assign each process a priority
  - At any time, scheduler runs highest priority process ready to run
  - Process runs to completion unless preempted

Typical RTOS Task Model



- Each task a triplet: `(execution time, period, deadline)`
- Can be initiated any time during the period
- Usually, `deadline = period` and task initiated any time before deadline

Priority-based Preemptive Scheduling



- Always runs the highest-priority runnable process. Multiple processes at the same priority level? A few solutions:
  - Simply prohibit: Each process has unique priority
  - Time-slice processes at the same priority (extra context-switch overhead, no starvation)
  - Processes at the same priority never preempt the other (more efficient, still meets deadlines if possible)
- Deadlines, not fairness, the goal of RTOSes

Rate-Monotonic Scheduling (RMS)

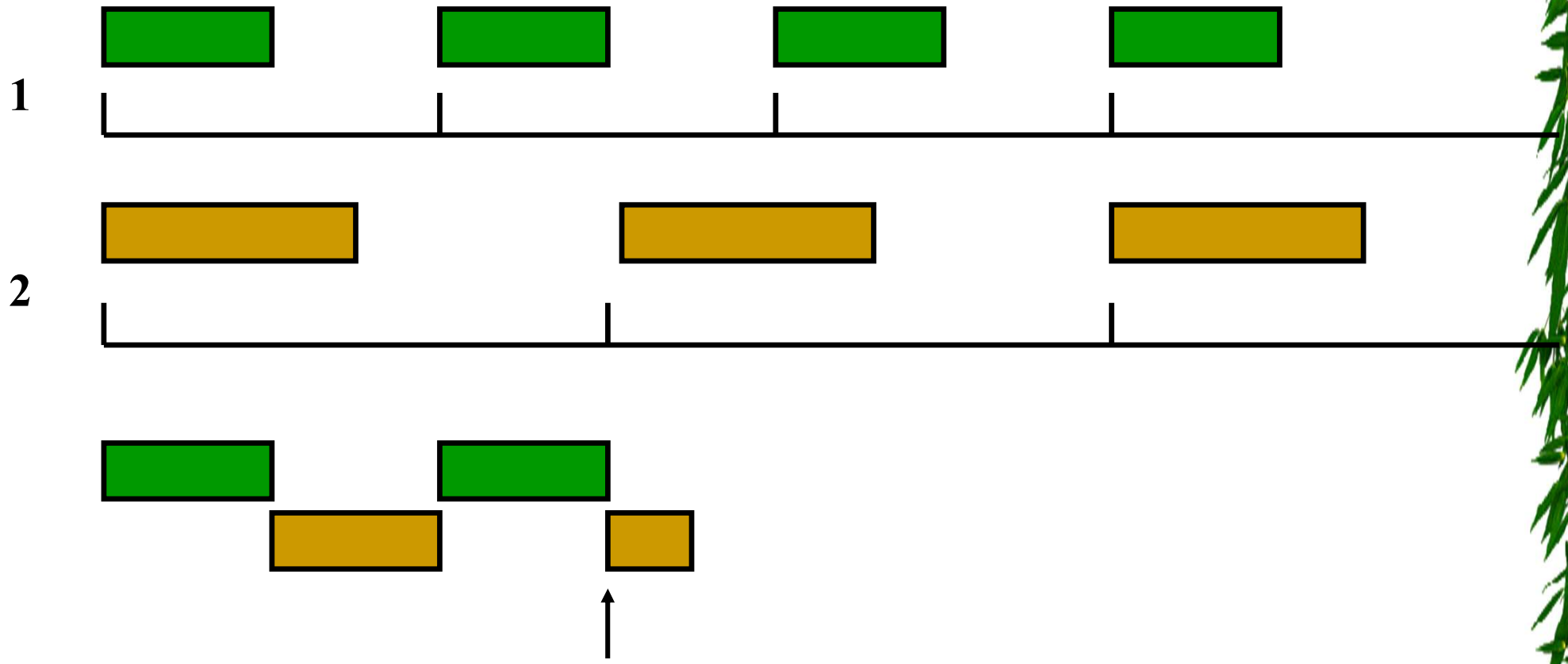
Period	Priority	
10	1	(highest)
12	2	
15	3	
20	4	(lowest)

- Common way to assign priorities, Liu & Layland, 1973 (JACM)
- Simple to understand and implement, e.g. processes with shorter period get higher priority
- Rate-monotonic scheduling is optimal: **if there is fixed-priority schedule that meets all deadlines, then RMS will produce a feasible schedule**
- Task sets do not always have a schedule (require >100% CPU utilization), e.g.

  P1 = (10, 20, 20), P2 = (5, 9, 9)

RMS Missing a Deadline

- p1 = (10,20,20), p2 = (15,30,30) have utilization 100%



P2 misses first deadline

Would have met the deadline if p2 = (10,30,30), utilization reduced 83%

When Is There an RMS Schedule?

- Consider sum of compute time divided by period for each process $U=\Sigma c_i/p_i$
- No schedule can possibly exist if $U>1$, i.e. no CPU can run 110% of time
- RMS schedule always exists if $U < n\ (2^{1/n} - 1)$

n	Bound for U	
1	100%	Trivial: one process
2	83%	Two process case
3	78%	
4	76%	
…		
∞	69%	Asymptotic bound

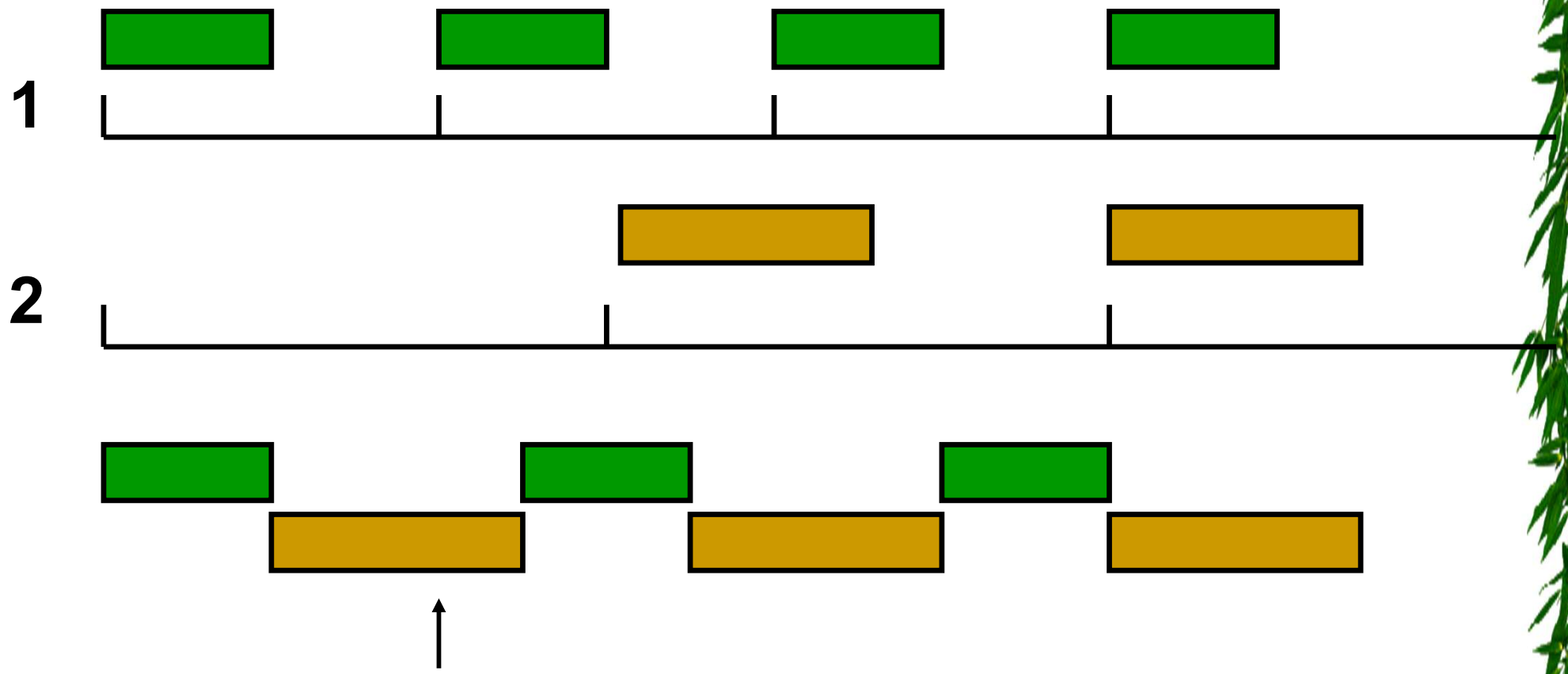
- Rate-monotonic analysis : **if the required processor utilization is under 69%, RMS will always give a valid schedule**
- Converse is not true, i.e. if the required processor utilization is over 69%, RMS might still give a valid schedule, but there is no guarantee

EDF Scheduling

- RMS assumes fixed priorities. Can you do better with dynamic priorities?
- Earliest deadline first: Processes with soonest deadline given highest priority (EDF harder to analyze)
- Earliest deadline first scheduling is optimal: **If a dynamic priority schedule exists, EDF will produce a feasible schedule**
- Earliest deadline first scheduling is efficient: RMS only guarantees feasibility at 69% utilization, EDF guarantees it at 100% or less

EDF Meeting a Deadline

- p1 = (10,20,20) p2 = (15,30,30) utilization is 100%



P2 takes priority because its deadline is sooner

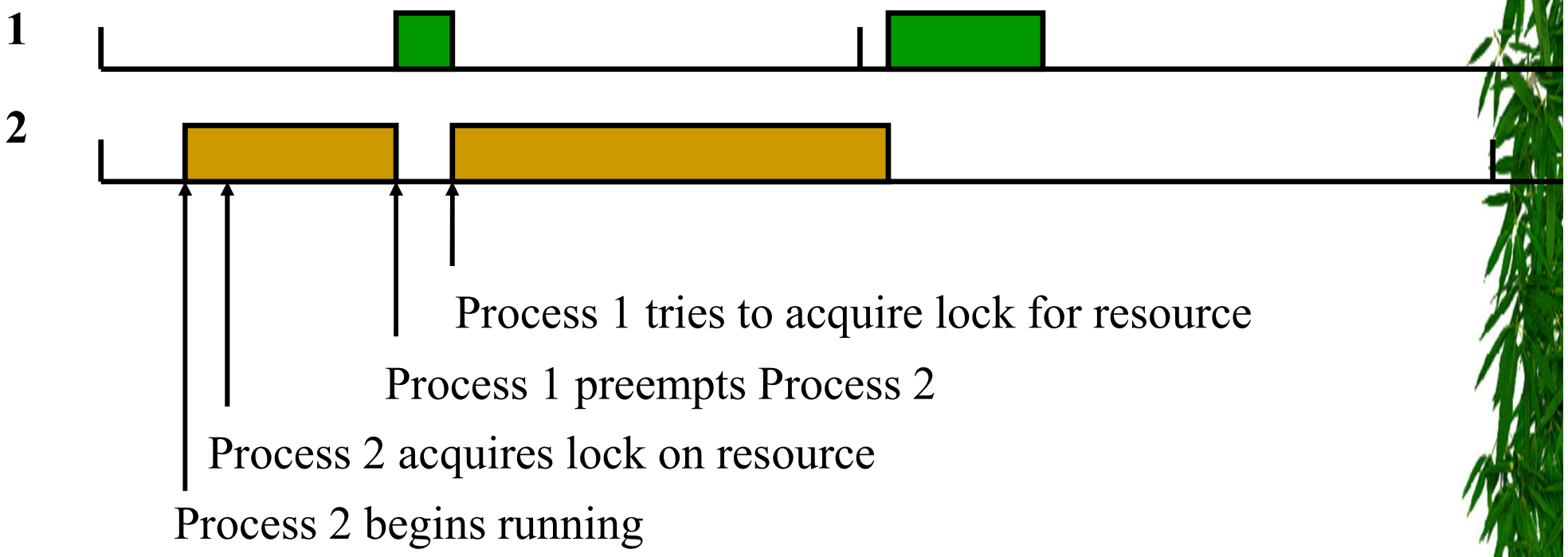
Priority Inversion

- Lower-priority process effectively blocks a higher-priority one
  - Lower-priority owns lock that prevents higher-priority process from running
  - Makes high-priority process runtime unpredictable

Priority Inversion

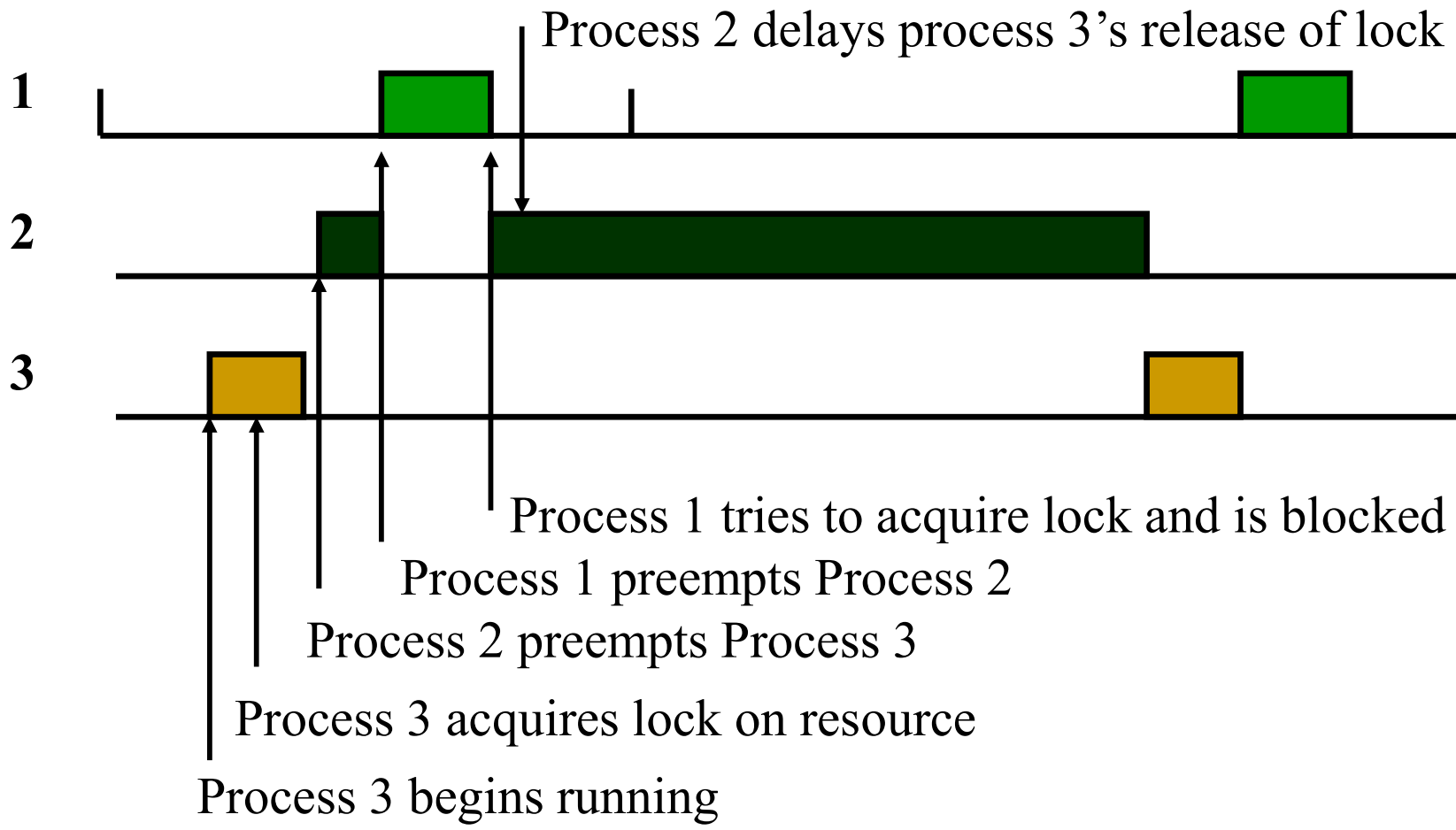
- Priority inversion is undesirable, since a high priority task gets blocked (waits for CPU) by lower priority tasks. Example:
  - Let T_1 , T_2, and T_3 three periodic tasks with decreasing priority order.
  - Let T_1 and T_3 share a resource
  - T3 obtains a lock on the semaphore *S* and enters its critical section
  - T1 is ready to run and preempts T3. Then, T1 tries to enter its critical section, trying to lock *S*. But, *S* is acquired by T3, hence T1 is blocked
  - T2 is ready to run. Since only T2 and T3 are ready to run, T2 preempts T3 while T3 is in its critical section.
- Ideally, the highest priority task (T1) should be blocked no longer than the time for T3 to complete its critical section. However, duration of blocking is unpredictable, since task T2 is executed in between.

Priority Inversion



- RMS and EDF assume no process interaction
- Often a gross oversimplification
- Consider the following scenario:

Nastier Example



- Higher priority process blocked indefinitely

Priority Inversion in Real World

- Mars Pathfinder mission (July 4, 1997)
- VxWorks (real-time OS), preemptive priority scheduling of threads (e.g., RMS)
- Priority inversion involving three threads:
  - Information bus task (T1), meteorological data gathering task (T3), communication task (T2). Priority order: T1>T2>T3
  - Shared resource: information bus (used mutex)
- Same situation as described in the previous example had occurred
- **Findings**
- Priority ceiling protocol was found to be disabled initially, then it was enabled online and the problem was corrected

Priority Inheritance

- Solution to priority inversion by temporarily increasing a process's priority when it acquires a lock
- High enough priority assigned to prevent preemption by another process
- Danger: Low-priority process acquires lock, gets high priority and hogs the processor
- Basic rule: low-priority processes should acquire high-priority locks only briefly
- No equivalent theoretical results (RMS/EDF) when locks and priority inheritance is used

Priority Inheritance

task	operation sequence on critical section
T_1	$Lock(CS_2)$ $Lock(CS_1)$ $Unlock(CS_1)$ $Unlock(CS_2)$
T_2	$Lock(CS_1)$ $Lock(CS_2)$ $Unlock(CS_2)$ $Unlock(CS_1)$

time	task	action
t_0	T_1	starts execution
t_1	T_1	locks CS_2
t_2	T_2	activated and preempts T_1 due to its higher priority
t_3	T_2	locks CS_1
t_4	T_2	attempts to lock CS_2, but is blocked because T_1 has a lock on it
t_5	T_1	inherits the priority of T_2 and starts executing
t_6	T_1	attempts to lock CS_1, but is blocked because T_2 has a lock on it
$\geq t_7$	-	both the tasks cannot proceed (deadlocked)

Figure 1: Example for priority inheritance protocol resulting in deadlock

Priority Ceiling Protocol

- For each semaphore, a *priority ceiling* is defined, whose value is the highest priority of all the tasks that may lock it.
- The priority ceiling protocol is the same as the priority inheritance protocol, except that a task T_i can also be blocked from entering a critical section if any other task is currently holding a semaphore whose priority ceiling is greater than or equal to the priority of task T_i.
- Prevents mutual deadlock among tasks
- A task can be blocked by lower priority tasks at most once

Priority Ceiling Protocol

- For the previous example, the priority ceiling for both CS_1 and CS_2 is the priority of T_2.
- From time t_0 to t_2, the operations are the same as before.
- At time t_3, T_2 attempts to lock CS_1, but is blocked since CS_2 (which has been locked by T_1) has a priority ceiling equal to the priority of T_2.
- Thus, T_1 inherits the priority of T_2 and proceeds to completion, thereby preventing deadlock situation.

Fault Tolerance - Byzantine Agreement

- Erroneous local clocks can have an impact on the computed local time.
- Advanced algorithms are fault-tolerant with respect to Byzantine errors. Excluding k erroneous clocks is possible with 3k+1 clocks (largest and smallest values will be excluded.
- Many publications in this area.

Real-time data bases

- Transaction is a sequence of read/write operations.
- Properties of transactions
  - **Atomic** transaction: either completed or has no effect at all
  - **Consistent** set of values retrieved from several data accesses
  - **Isolation:** no user should see intermediate transaction states
  - **Durability:** results of transactions should be persistent
- For hard discs, access times are hardly predictable. Possible solutions:
  - Relax ACID requirements
  - Main memory data bases
  - Access to remote objects through middleware (RT-CORBA, RT-MPI)

Priority Ceiling Emulation

- Once a task locks a semaphore, its priority is immediately raised to the level of the priority ceiling of the semaphore.
- Deadlock avoidance and block at-most-once result of priority ceiling protocol still holds.
- Restriction: A task cannot suspend its execution within the critical section.

Modeling Blocking Time and Earlier Deadline

- Blocking time (Bi) encountered by task Ti by lower priority tasks can be modeled by increasing Ti′s utilization by Bi/Pi.
- Earlier deadline (Di < Pi) can also be modeled as blocking time for Ei = Pi – Di.
- Net increase in task Ti′s utilization is (Bi + Ei) / Pi.

Modeling Blocking and Earlier Deadline (Cont.)

- Schedulability Check (T1 > T2 > ... > Tn)

$$\forall i, 1 \le i \le n, \frac{C_1}{p_1} + \frac{C_2}{p_2} + \cdots + \frac{(C_i + B_i + E_i)}{p_i} \le i(2^{1/i} - 1)$$

- Completion time Test
  - Earlier deadline case : no change (i.e., same as discussed in RM exact analysis)
  - Blocking time: add Bi to Ci. So, the modified
    Ci′ = Ci + Bi

  (Note: Blocking Time calculation will be learned thro homework)

RTOS Comparison

name	Scheduling	Determinism	Responsiveness
VxWorks (WindRiver)	✷POSIX ✷preemptive: FIFO, RR (per thread) ✷non-preemptive	✷reentrant code ✷avoid deadlock via multi-owner semaphore	✷interrupt/task locking ✷avoid priority inversion via inheritance
OSEK/VDX (Siemens)	✷preemptive(FIFO) ✷non-preemptive	✷avoid deadlock/priority inversion: prevent preemption if resource acquired	✷fast ISR (stack/no-stack) ✷scheduler locking