Real Time Systems for Automation
Real Time module

Dante Piotto

Spring semester 2023

Chapter 1

Introduction

1.1 A typical control system

1.2 Software vision

tasks share data using shared buffers

In a real time system correctness also depends on the time at which results are produced

Modularity: subsystems must be developed without knowing details about the other subsystems

Modularity can be achieved by:

- partitioning the system into a set of subsystems, each managed by one or more computational tasks
- defining precise interfaces between tasks, each specifying:
  - data exchanged (I/O)
  - function of the task
  - validity assumptions (e.g. admissible ranges)
  - performance requirements

Buffers used in RTOS are tipically non blocking: *Asynchronous communication mechanisms*:

- Cyclic Asynchronous Buffers
- A message is not consumed by a receiving process but is maintained in the CAB structure until a new message is overwritten

1.3 RTOS

A RTOS is responsible for:

- managing cuncurrent tasks
- scheduling
- handling critical sections
- managing interrupts

Sources of non determinism in RT systems:

- Architecture (Cache, pipelining, interrupts, DMA)
- OS (Scheduling, synchronization, communication)
- Language (lack of explicit support for time)
- Design methodologies: lack of analysis and verification techniques
  - low reliability

A RTOS requires bounded/predicable latency (controlled by architecture and choice of OS)

1.3.1 Achieving predictability

DMA

Problem: cycle stealing
Possible solution: time-slice method
each memory cycle split into two adjacent time slots, one reserved for the CPU, other oen reserved for the DMA device. Higher performance cost than cycle stealing but overall more predictable as the CPU is guaranteed half of the memory cycle

Cache

Problem: hit ratio. Preemption destroys locality. Cache related preemption delay is difficult to precisely estimate

Interrupts

Source: peripheral devices
Can introduce unbounded delays. ISR usually have static priorities. In generic OS I/O interrupts have real time constraints, whereas in RTOS a control process may be more urgent than interrupt handling
3 approaches:

1. Disable all interrupts except from the timer, and handle devices with polling
    - predictable
    - inefficient
2. Disable all interrupts except from the timer, and manage devices via periodic kernel routines
    - similar to above
3. Leave all interrupts enabled, driver only activates device management task that gets scheduled like other tasks.
    - no busy waiting
    - unbounded overhead due to drivers

System calls

Problem: can be difficult to evaluate worst-case execution time of each task
Solution: system calls in RTOS should be implemented with bounded latency. Moreover, it would be desirable that system calls be preemtpable.

Semaphores

Problem: unbounded priority inversion
Solutions: priority inheritance, priority ceiling (Resource Access Protocols).

Memory

Problem: demand paging / page faults
Solution: static partitioning. Predictable, but low flexibility in dynamic environments.

Programming Language

Problems:

- dynamic data structures
- recursion
- cycles

Solution: high level languages for programming hard real time applications

Chapter 2

Modeling Real Time Activities

Task

sequence of instructions that, in the absence of other activities, is continuously executed by the peocessor until completion.
Dispatching: Taking the first task in the ready queue and making it ready for execution Scheduling: Deciding in which order tasks should be dispatched for execution. Decides the order of tasks in the ready queue
Formally:

Schedule

Given a task set $\Gamma = \{J_1, \ldots, J_n\}$, a schedule is a function $\sigma : \mathbb{R}^+ \to \mathbb{N}$ that associates an integer k to each time slice $[t_i, t_{i+1})$ with meaning:

- $k = 0$: in $[t_i, t_{i+1})$ the processor is idle
- $k > 0$: in $[t_i, t_{i+1})$ the processor executes J_k

Preemptive schedule

Preemption is the act of suspending a running task and placing it in the ready queue Pros:

- timely response to issues (exception handling)
- different levels of criticality: processor executes most critical tasks
- higher CPU utilization

Cons:

- introduces runtime overhead
- program locality is destroyed

Real-Time task

A task characterized by a timing constraint on its response time, called deadline

- d_i is the absolute deadline
- D_i is the relative deadline
- The completion time is $f_i - s_i = R_i - (s_i - a_i)$

R_i is the response time

2.0.1 Feasibility

Feasible task

A Real-Time task τ_i is said to be feasible if it completes within its absolute deadline, that is, if $f_i \leq d_i$, or equivalently if $R_i \leq D_i$

Feasible Schedule

A schedule σ is said to be feasible if all the tasks can complete according to a specific set of constraints

Feasible task set

A set of tasks Γ is said to be feasible (or schedulable) if there exists at least one algorithm that can produce a feasible schedule for it.

Lateness

$L_i = f_i - d_i$
in a real time system lateness should be negative or at worst 0. Negative lateness is also called *slack*

slack time/laxity

The difference between the relative deadline and the computation time:

$$X_i = d_i - a_i - C_i$$

tardiness/exceeding time

$$E_i = max(0, L_i)$$

Job

Instance of a task (task is a set of instructions)

2.0.2 Activation mode

Time driven activation

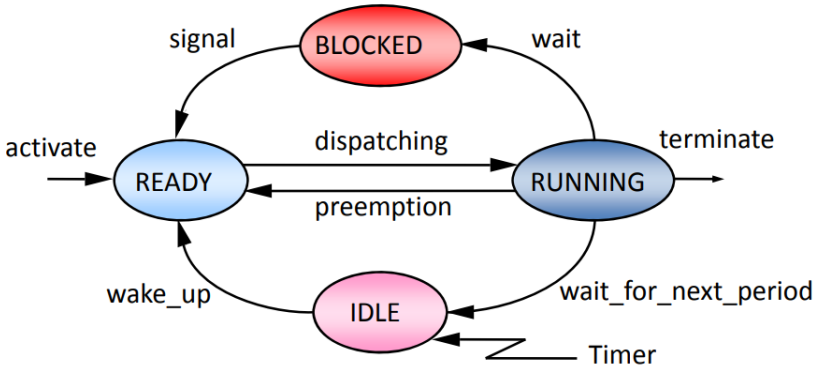
- Periodic task, τ_i
- the task is automatically activated by the OS at predefined time instants
- $a_{i,k+1} = a_{i,k} + T$

Event driven activation

- Aperiodic task: job, J_i
- The task is activated at an event arrival, or by explicitly invoking a system call
- $a_{i,k+1} > a_{i,k}$

2.0.3 Periodic task

After completing its computation, each instance enters an idle state



A metascheduler is in charge of periodically waking up idle tasks. This can be done in several ways:

- using semaphores: wait/signal
- using message passing: blocking receive/send (QNX/neutrino)
- by operating on the process' state: suspend/resume (vxWorks)

The term periodic task usually means that $D_i = T_i$. If $D_i < T_i$, we call it a sporadic task

Sporadic task

A sporadic task is an aperiodic task with a *minumum interarrival time* between consecutive jobs

$$a_{i,k+1} \geq a_{i,k} + T_i$$

utilization factor

$$U_i = \frac{C_i}{T_i}$$

in order for a task to be feasible, U_i must not exceed 1