IM420 Sistemas Embarcados de Tempo Real

Notas de Aula – Semana 07

Prof. Denis Loubach
dloubach@fem.unicamp.br

Programa de Pós-Graduação em Eng. Mecânica / Área de Mecatrônica
Faculdade de Engenharia Mecânica - FEM
Universidade Estadual de Campinas - UNICAMP

1^o Semestre de 2018

Tópicos

Fetch Robotics Introduces Warehouse Automation Robots

Silicon Valley startup Fetch Robotics recently unveiled a pair of robots designed to automate logistics in places like warehouses and distribution centers. The two robots, called Fetch and Freight, can work together to autonomously pick and deliver goods—tasks that are currently performed by human workers in most warehouses

Fonte: `http://spectrum.ieee.org/automaton/robotics/industrial-robots/fetch-robotics-introduces-fetch-and-freight-your-warehouse-is-now-automated/`

Comunicação inter-tarefas

Tarefas se comunicam e se sincronizam entre si com o auxílio de alguns elementos básicos (*intertask primitives*)

Estes objetos do *kernel* facilitam essa comunicação/sincronização

Exemplos:

- Semáforos (*semaphores*)
- *Mutex - mutual exclusion semaphore*
- Filas de mensagem (*message queues*)

Semáforos

Num sistema multi-tarefas, as tarefas concorrem entre si

Elas devem ser capazes de sincronizar sua execução para coordenar o acesso (mutuamente exclusivo) aos recursos compartilhados no sistema

Para tal, o *kernel* do RTOS propicia um objeto denominado semáforo juntamente com seus serviços de gerenciamento

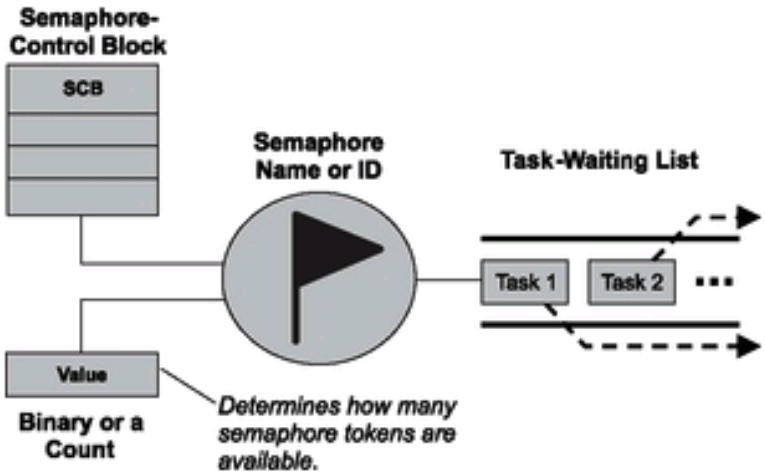
Definição

Semáforo

Um semáforo (as vezes chamada de *token* de semáforo) consiste de um objeto do *kernel* que uma ou mais tarefas podem adquirir e devolver para propósitos de sincronização ou exclusão mútua

Um semáforo é como uma chave que permite à tarefa continuar sua execução ou acessar algum recurso compartilhado

Ilustração de um semáforo



Tipos básicos de semáforos

Alguns tipos de semáforos oferecidos pelo *kernel*

- Semáforo binário
- Semáforo contador
- *Mutex*

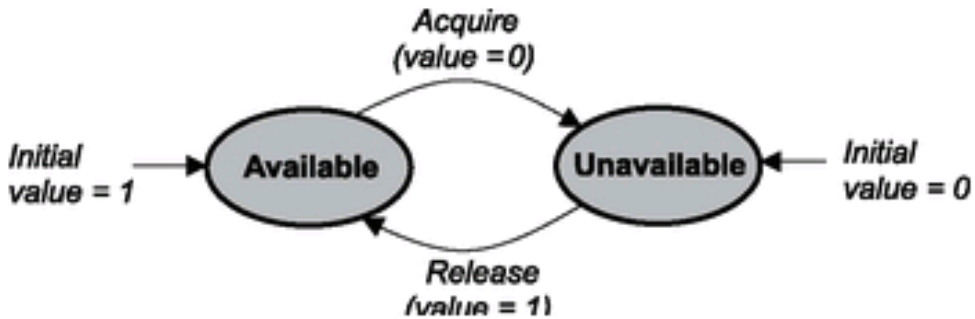
Semáforo binário (1/2)

Semáforo binário

Um semáforo binário pode ter o valores "1" ou "0". Quando o semáforo tem o valor "0", ele é considerado indisponível (ou vazio). Quando o semáforo tem o valor "1", ele é considerado disponível (ou cheio)

Semáforo binário (2/2)

Máquina de estados do semáforo binário



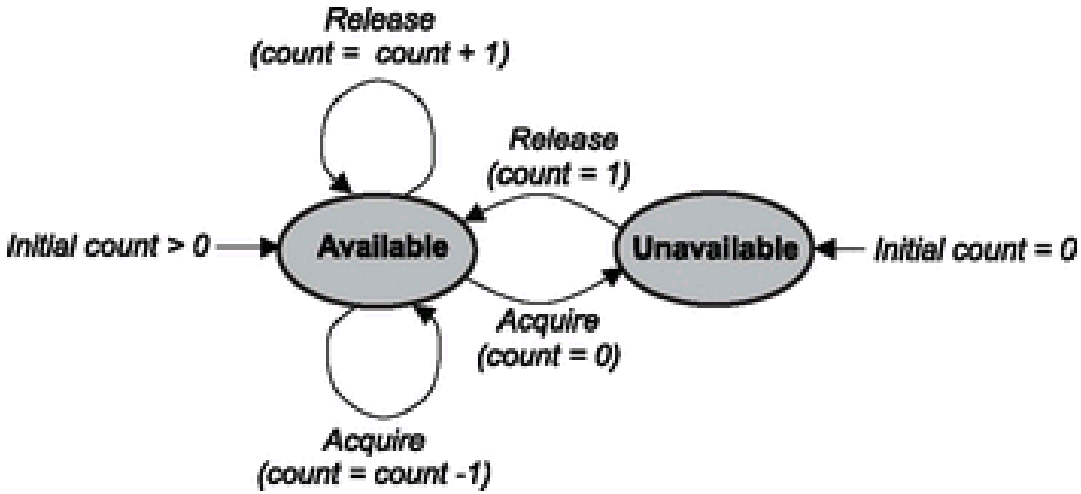
Semáforo contador (1/2)

Semáforo contador

Um semáforo contador usa um contador para permitir que ele seja adquirido e devolvido múltiplas vezes, definido pelo número do contador

Semáforo contador (2/2)

Máquina de estados do semáforo contador



Mutex (1/3)

Mutex

Um mutex é um tipo especial de semáforo binário que suporta posse, acesso recursivo, exclusão com segurança de uma tarefa, e um ou mais protocolos para evitar problemas de herança de exclusão mútua

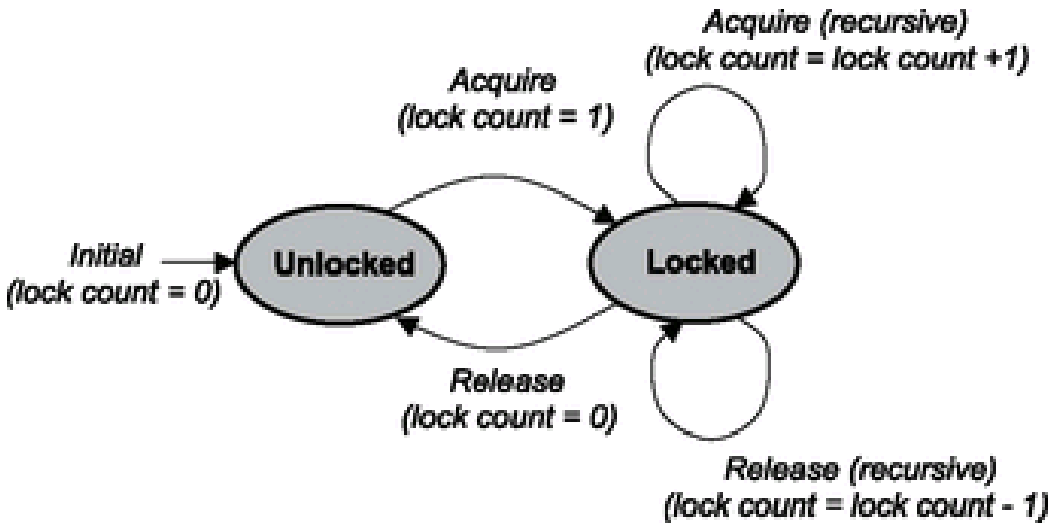
- Diferente do semáforo binário com os estados disponível/não disponível
- O *mutex* possui os estados destravado/travado

Mutex (2/3)

- A posse de um *mutex* é obtida por uma tarefa quando ela "trava" o *mutex* adquirindo-o numa primeira vez
- Quando o *mutex* encontra-se travado (na posse de uma tarefa), não é possível adquirí-lo ou liberá-lo em qualquer outra tarefa
- Contrastando com o conceito do semáforo binário, onde o semáforo pode ser liberado por qualquer tarefa, incluindo as que não o adquiriram originalmente
- Algumas implementações permitem que a tarefa "trave" o *mutex* por múltiplas vezes (recursividade)
- Enquanto a tarefa possui o *mutex*, ela não poderá ser excluída (garantia de exclusão segura da tarefa)

Mutex (3/3)

Máquina de estados do *mutex*



Filas de mensagem (1/3)

O mecanismo de semáforo permite uma comunicação entre tarefas de modo a sincronizar suas execuções

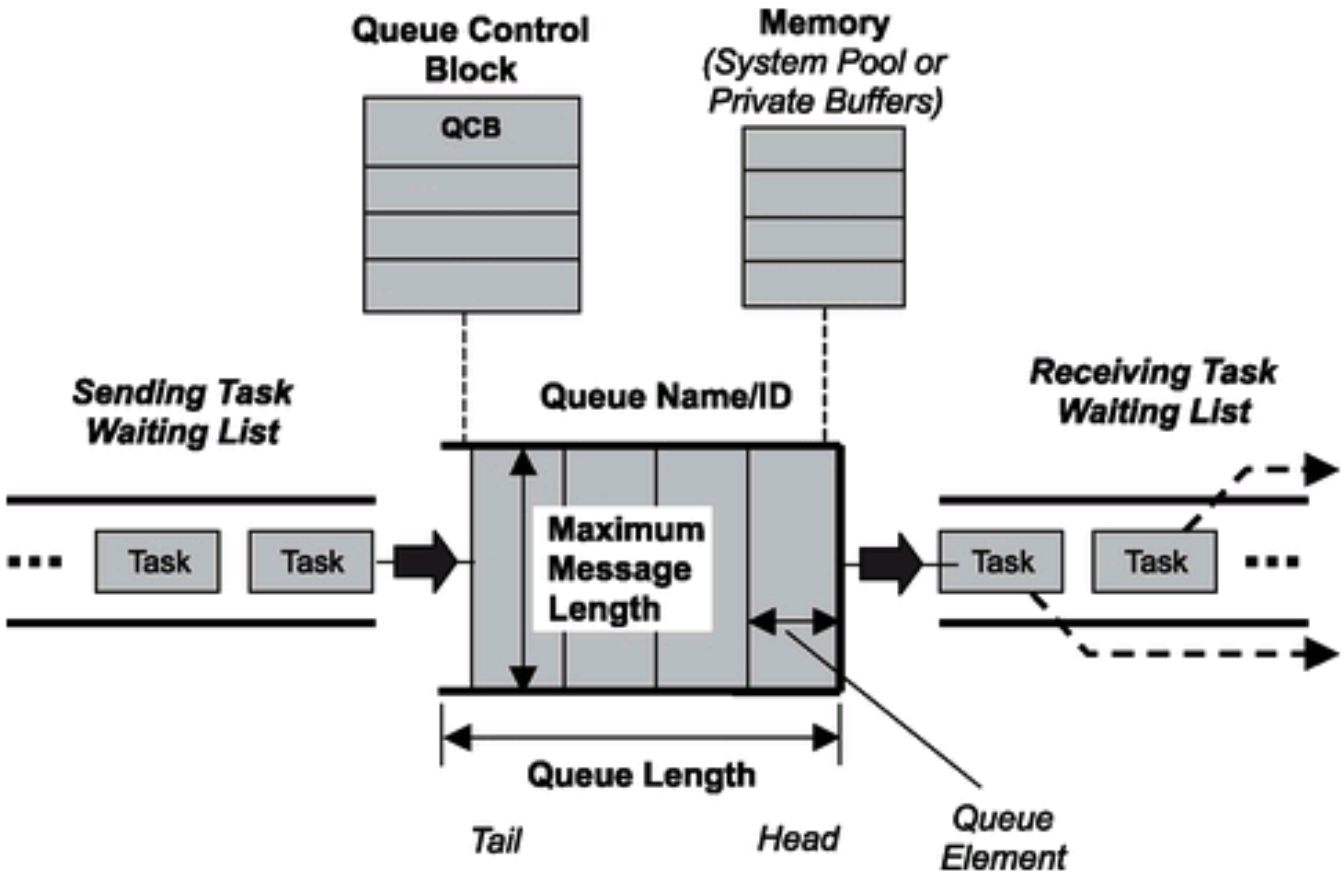
As filas de mensagem, por outro lado, permitem que as tarefas troquem dados (mensagens) entre si

Filas de mensagem

Uma fila de mensagem é um objeto do *kernel* tipo *buffer* através do qual tarefas e ISRs mandam e recebem mensagens para se comunicar e sicronizar utilizando dados

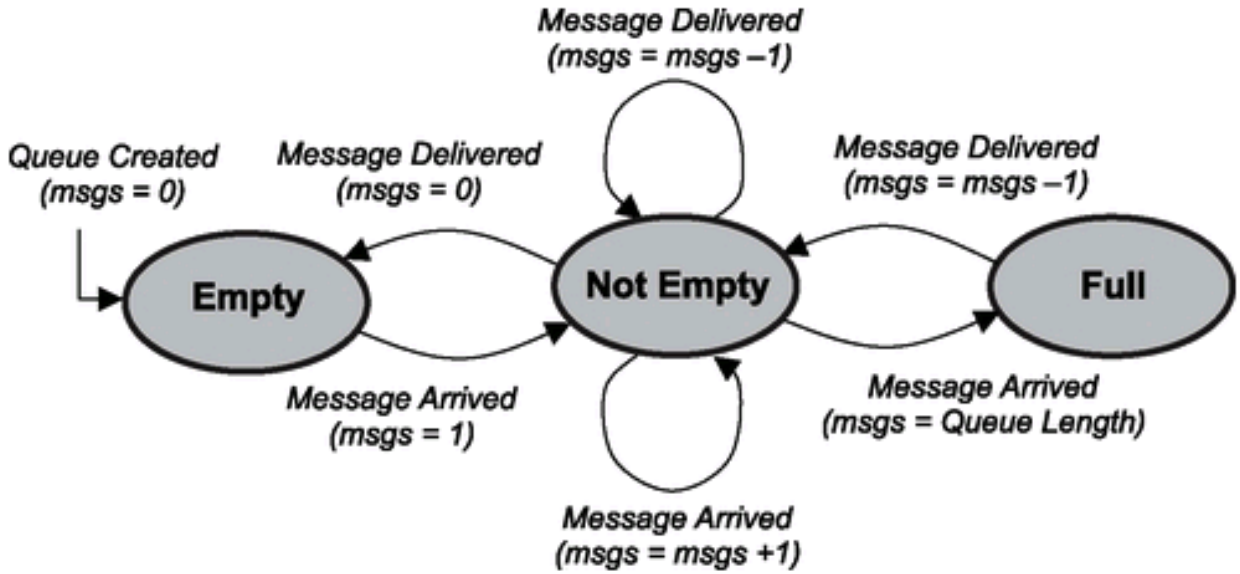
Filas de mensagem (2/3)

Ilustração de uma fila de mensagem



Filas de mensagem (3/3)

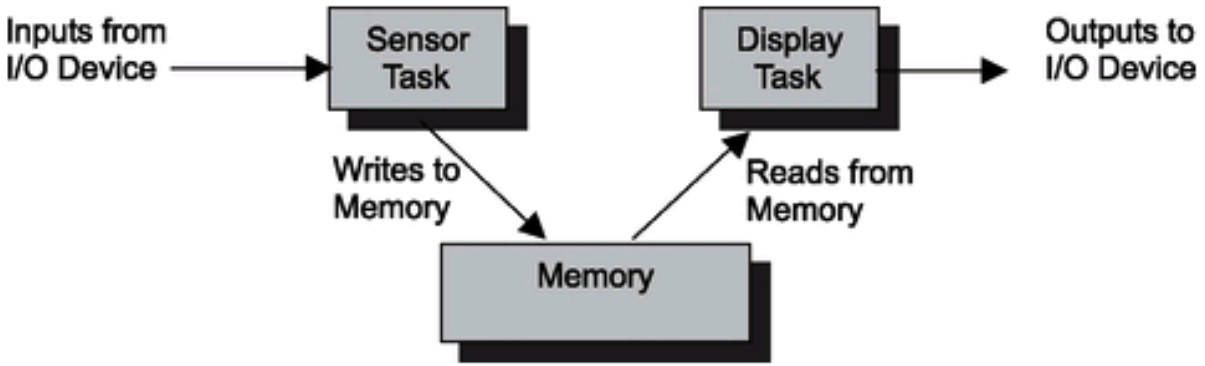
Máquina de estados de uma fila de mensagem



Seção crítica (1/2)

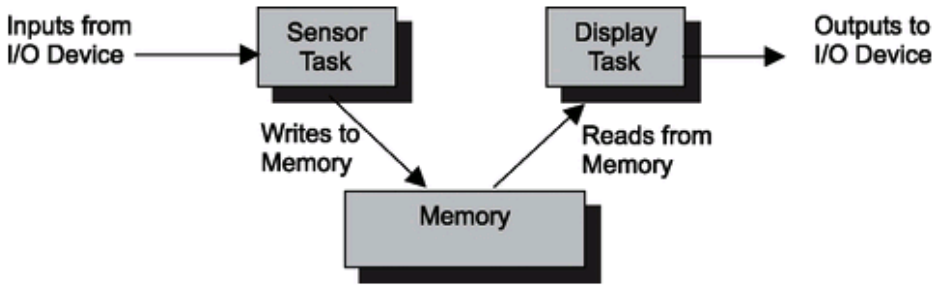
Seção crítica

Parte do código onde um recurso compartilhado é acessado



Um algoritmo de exclusão mútua garante que quando uma tarefa estiver executando sua seção crítica, tal execução será atômica, ou seja, não haverá interrupções por seções críticas concorrentes ou qualquer outra tarefa concorrente

Seção crítica (2/2)



A seção crítica da tarefa `Sensor` é a parte em que ela escreve dados na memória compartilhada

A seção crítica da tarefa `Display` é a parte em que ela lê dados da memória compartilhada

Estas duas seções críticas são chamadas de seções críticas concorrentes uma vez que elas requerem o mesmo recurso compartilhado

Paradigma tradicional (1/3)

Programação "sequencial"

- Sistema de execução ativo e sequencial
- Quando a tarefa precisa de algum evento (ex. mensagem, semáforo) para continuar sua execução, ela espera (bloqueia) até que o evento chegue
- Neste caso, a tarefa está no controle do sistema
- A tarefa ganha o controle para só depois checar os recursos que irá precisar
- Uma vez bloqueada, a tarefa não continua sua execução

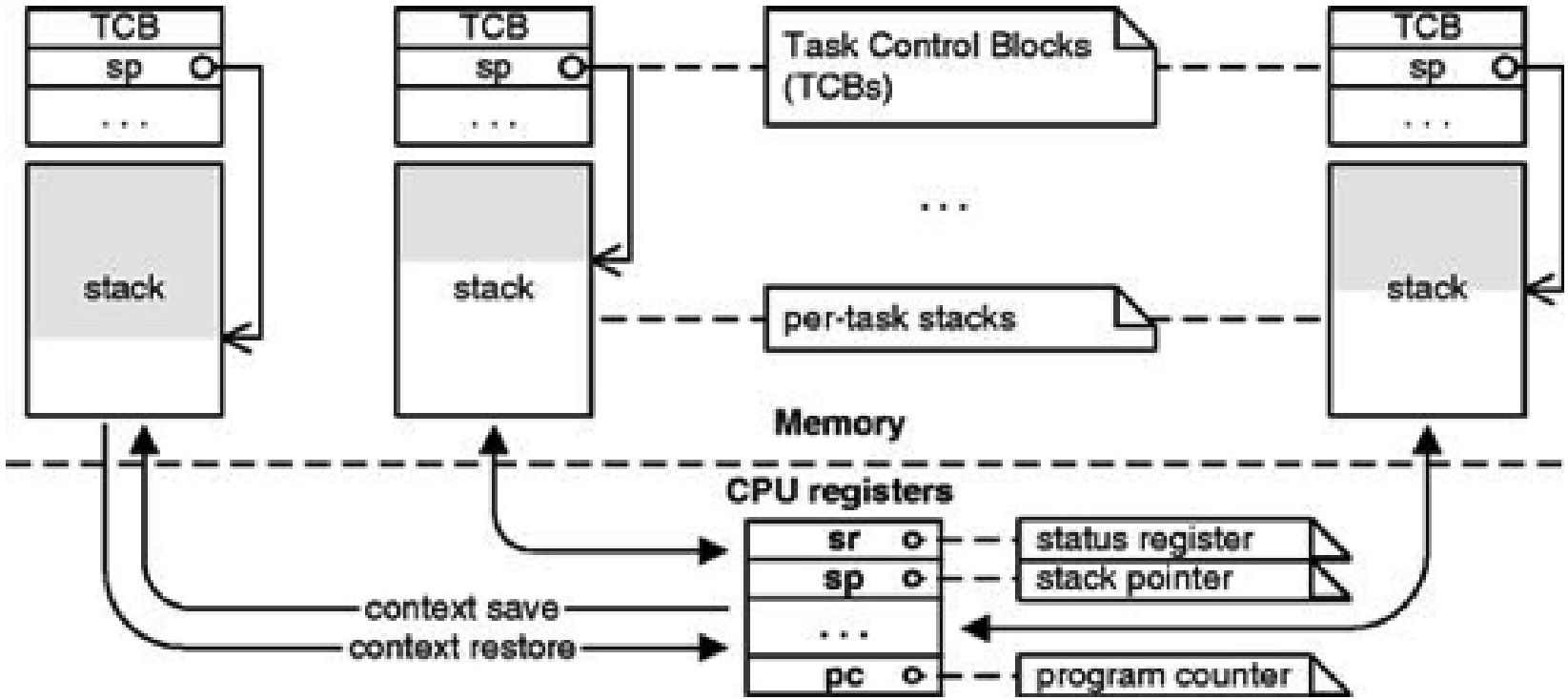
Paradigma tradicional (2/3)

Vantagens e desvantagens

- A principal vantagem deste paradigma é que ele é bem próximo do jeito com que linguagens procedurais convencionais trabalham (C, C++)
- A principal desvantagem é que o sistema pode ficar sem responder outros eventos enquanto espera por algum
- Uma grande vantagem da programação multi-tarefas é o melhor aproveitamento e utilização da CPU
- Considerando uma tarefa bloqueada, outras poderão assumir e executar na CPU

Paradigma tradicional (3/3)

Sistema multi-tarefas



Paradigma do controle invertido (1/2)

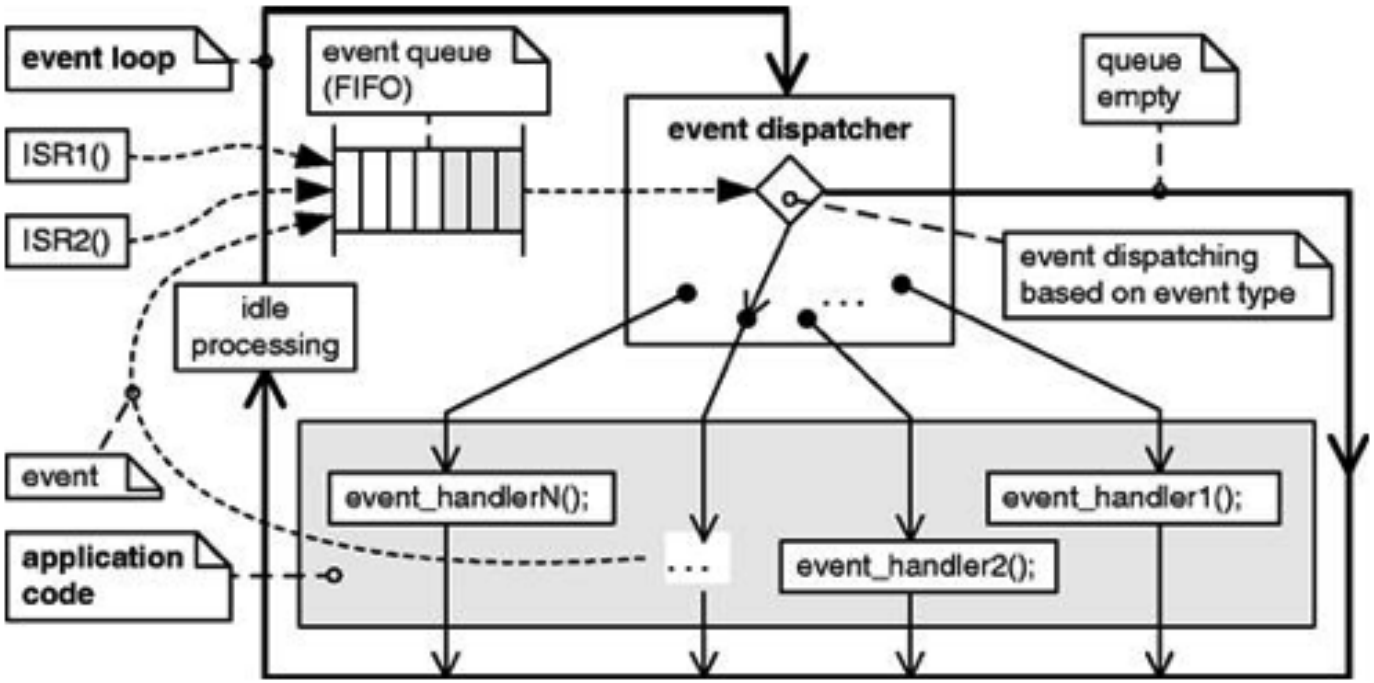
Programação dirigida por eventos e filas

- Sistemas reativos
- Inversão de controle (princípio de "*Hollywood*")
- Busca alta coesão e baixo acoplamento
- Divisão:
  - Infraestrutura supervisora de gerenciamento de eventos (controle)
  - Chamada/manipulação dos eventos

Figura: Princípio de Hollywood

Paradigma do controle invertido (2/2)

Ilustração da estrutura do controle invertido



Exemplo

Descrição do problema:
Sistema de controle de atitude de satélite geostacionário

Requisitos:

- Existe uma posição espacial (x, y, z) fixa conhecida em que o satélite deve permanecer
- Existem 3 sensores de posição (x, y, z) que devem ser verificados a cada 500 milisegundos
- Existem 3 atuadores no satélite (para cada eixo: x, y e z)
- Se a posição corrente não for a desejada, o sistema deve comandar a correção e aguardar até 1 segundo pela resposta do atuador

Por simplificação, assume-se que existe uma função $f(x, y, z) = pos$ que sintetiza a posição dos sensores e outra função $f'(x, y, z)$ que atua em cada eixo do satélite de maneira independente

Descrição do problema (cont...)

Sistema de controle de atitude de satélite geostacionário

Requisito de projeto:

- Implementar utilizando conceitos de multitarefa
- Utilizar as ferramentas de modelagem apresentadas

Heurísticas para projetar o sistema

1. Identificar todos os objetos ativos com comportamento reativo
2. Atribuir responsabilidades e recursos para os objetos
3. Identificar os eventos do sistema
4. Utilizar diagramas de caso de uso, de classes, de sequência, ...
5. Implementar uma máquina de estados para cada objeto

Cada objeto ativo se tornará uma **tarefa** no sistema

A utilização de modelos (abstração) ajuda a estruturar o pensamento levando a uma solução melhor elaborada

Projeto do sistema (1/5)

1. Objetos ativos: sensor, atuador
2. Sensor: computar posição corrente do satélite, ativar correção se necessário
3. Atuador: comandar atuadores até que a posição seja corrigida
4. Eventos do sistema: CORRIGIR_POS, POS_CORRIGIDA, TICK

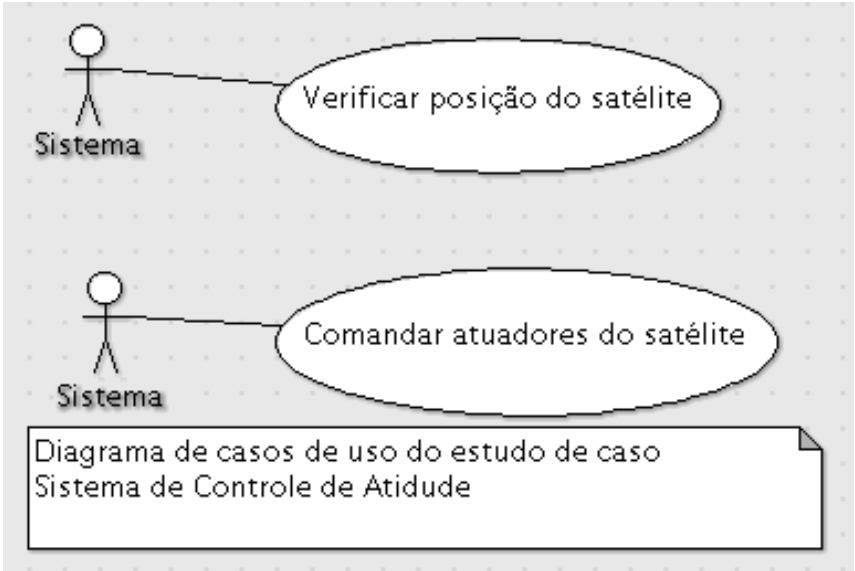


Figura: Diagrama de Caso de Uso

Projeto do sistema (2/5)

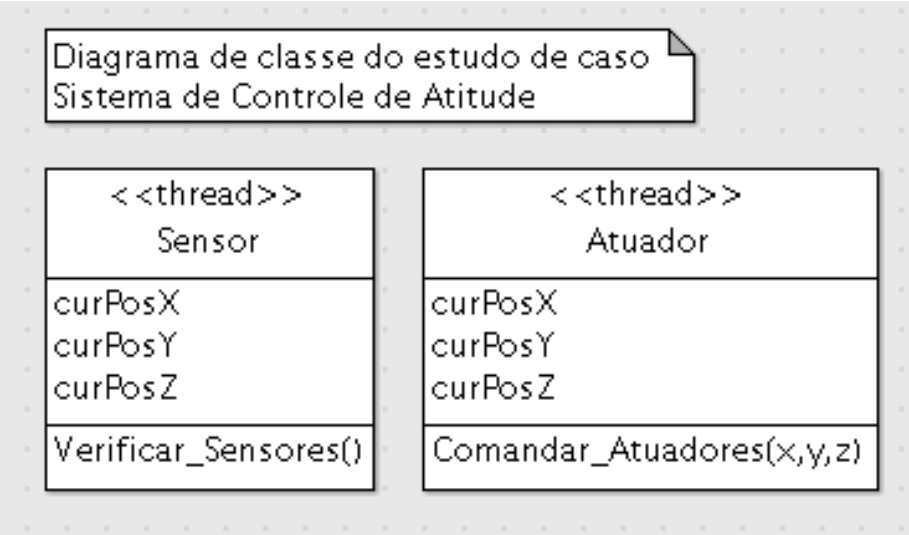


Figura: Diagrama de Classe

Projeto do sistema (3/5)

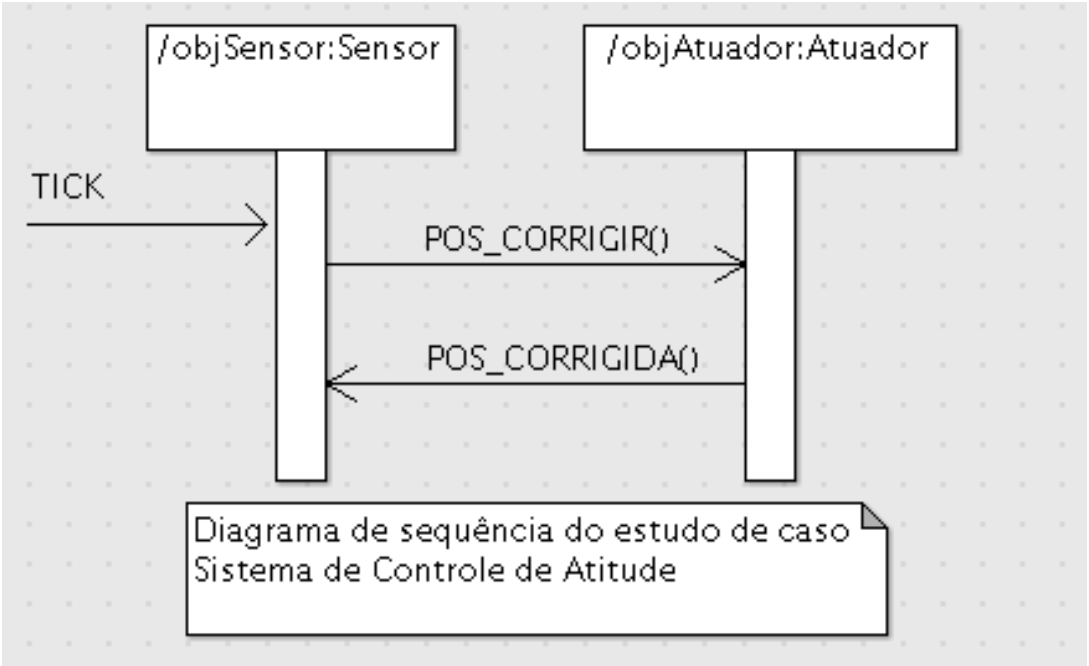


Figura: Diagrama de Sequência

Projeto do sistema (4/5)

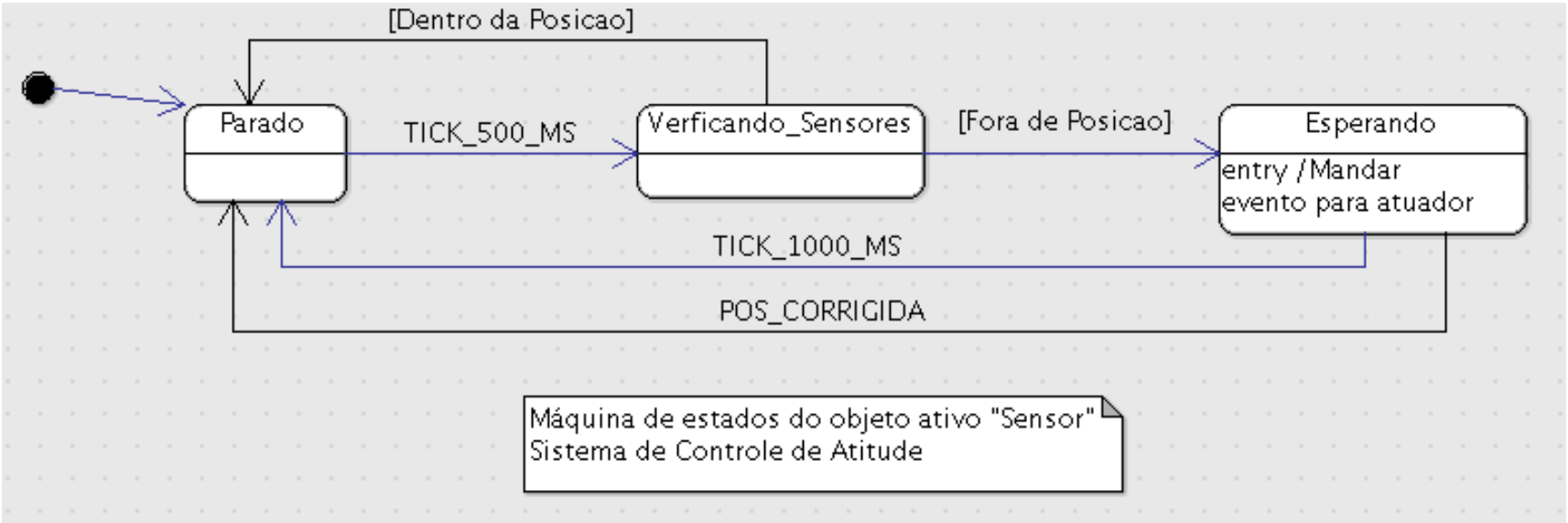


Figura: Máquina de Estados do Sensor

Projeto do sistema (5/5)

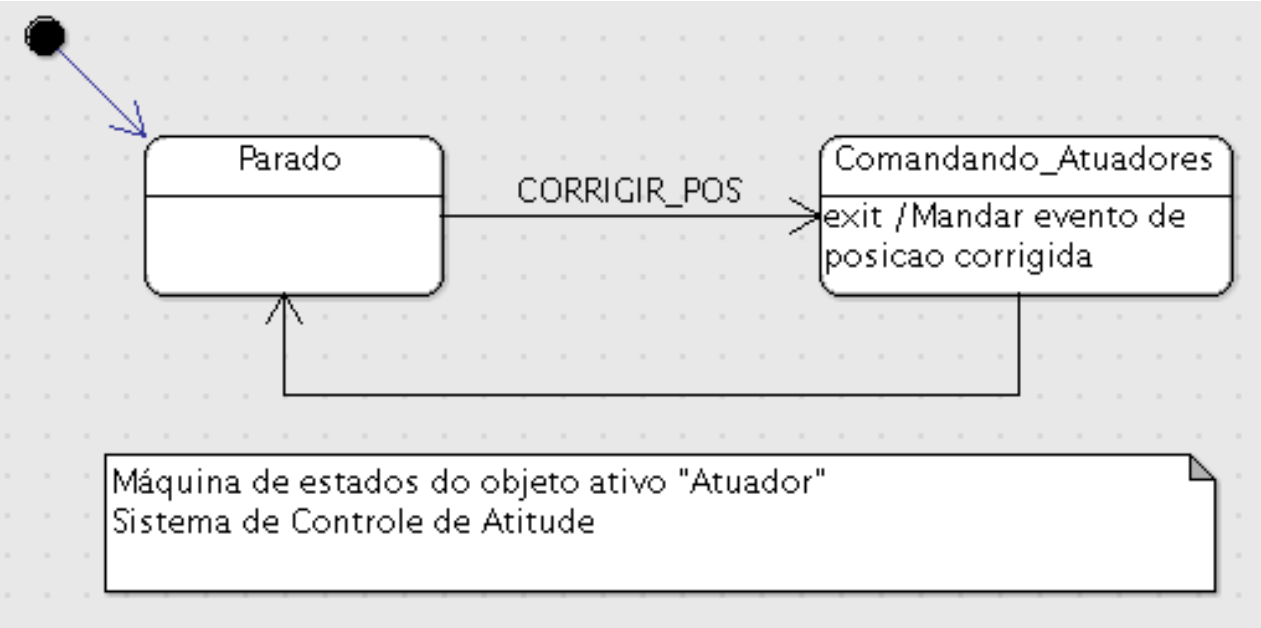


Figura: Máquina de Estados do Atuador

Discussão

Existem pelo menos duas abordagens para se implementar o sistema multitarefa proposto neste estudo de caso

1. Paradigma tradicional (controle sequencial básico)
   - Basicamente, uma função por tarefa
   - Utiliza recursos como semáforos e filas
   - Pode bloquear a tarefa
   - Tem restrições para tratar múltiplos eventos
   - *Loop* infinito
2. Paradigma de controle invertido
   - Basicamente, uma função por estado da tarefa
   - (Geralmente) não utiliza recursos como semáforos e filas
   - Visa não bloquear a tarefa
   - Bom desempenho ao tratar múltiplos eventos
   - Executa e devolve o controle

Implementação (1/6)

Paradigma tradicional (controle sequencial básico)

Algumas considerações:

- Escalonador baseado em prioridade fixa
- Utilização de filas de mensagem para comunicação entre as tarefas
- Tarefa bloqueia quando verificar que o evento esperado não está na fila
- Exemplo utilizando sistema operacional de tempo real MQX (*Message Queue eXecutive*)

A utilização de outros sistemas operacionais de tempo real é semelhante

Implementação (2/6)

Paradigma tradicional (controle sequencial básico)

Trecho de código para as máquinas de estado

```
1
2     switch(currenState)
3     {
4       case Parado:
5       // verificar fila de mensagem
6       waitEvent(CORRIGIR_POS);
7       currenState = Comandando_Atuadores;
8       break;
9
10      case Comandando_Atuadores:
11      // commandando atuadores
12      f_linha(x, y, z);
13      // posicao corrigida, mandar evento
14      // para a tarefa do Sensor
15      sendEvent(POS_CORRIGIDA);
16      currenState = Parado;
17      break;
18    }
```

Implementação (3/6)

Paradigma tradicional (controle sequencial básico)

Trecho de código para criação de tarefas em MQX

```
1
2     #include <mqx.h>
3     #include <bsp.h>
4
5     _task_create(0, SENSOR_TASK, 0);
6     _task_create(0, ATUADOR_TASK, 0);
7
8     TASK_TEMPLATE_STRUCT MQX_template_list[] =
9     {
10      {SENSOR_TASK, sensor_task, 1500, 10, "tSensor", 0, 0, 0},
11      {ATUADOR_TASK, atuador_task, 1500, 9, "tAtuador", 0, 0, 0},
12      {0,0,0,0,0,0,0,0}
13    };
```

Implementação (4/6)

Paradigma controle invertido

Algumas considerações

- Cada estado da máquina é codificado separadamente
- Utilização de eventos para disparar as atividades
- Tarefa não bloqueia
- Exemplo utilizando moldura de tempo real QP (*Quantum-Leap*)

Implementação (5/6)

Paradigma controle invertido

Trecho de código para as máquinas de estado em QP

```
1
2     QState sensor_Parado(Sensor *me, QEvt const *e)
3     {
4       switch (e->sig)
5       {
6         case TICK_500_MS:
7         // verificar sensores
8         me-> sensor_Verificando_Sensores;
9         return Q_HANDLED();
10        break;
11      }
12
13      return Q_SUPER(&QHsm_top);
14    }
```

Implementação (6/6)

Paradigma controle invertido

Trecho de código para criação de tarefas em QP

```
1     QActive_start(
2     AO_Sensor,
3     1,                                                  /* priority  */
4     l_sensorQueueSto, Q_DIM(l_sensorQueueSto), /* evt queue */
5     (void *)0, 0,                          /* no per-thread stack */
6     (QEvt *)0);                          /* no initialization event */
```

Informação ao leitor

Notas de aula baseadas nos seguintes textos:


Q. Li and C. Yao, *Real-Time Concepts for Embedded Systems*.
CMP books, 2003.


M. Samek, *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems*.
Newnes (Elsevier), 2nd ed., 2009.