IM420 Sistemas Embarcados de Tempo Real

Notas de Aula – Semana 09

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

Looking into the future

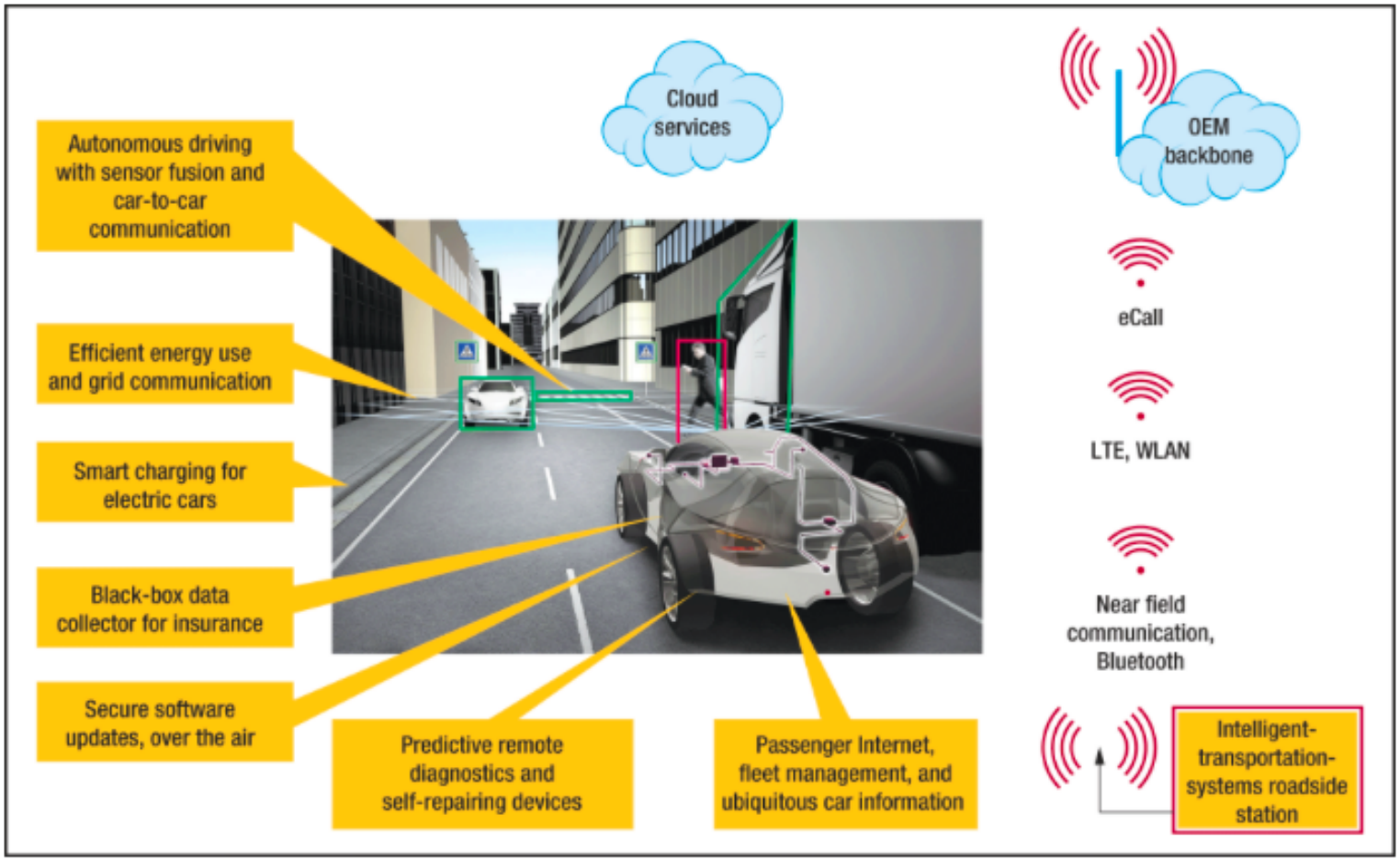


Figure 1.Software innovation fuels automotive advances, from smart energy efficiency to autonomous driving.

Figura: FONTE: `http://doi.ieeecomputersociety.org/10.1109/MS.2015.142`

Outros objetos do *kernel*

Além dos objetos chaves do *kernel* (tais como tarefas, semáforos e filas de mensagens) o *kernel* ainda provê outros objetos importantes

Observar que devida a grande variedade de implementações existentes, a disponibilidade de tais objetos nas diferentes versões pode apresentar variações

Alguns desses outros objetos são:

- *Pipes*
- Registradores de eventos
- Sinais
- Variáveis condicionais

Pipes

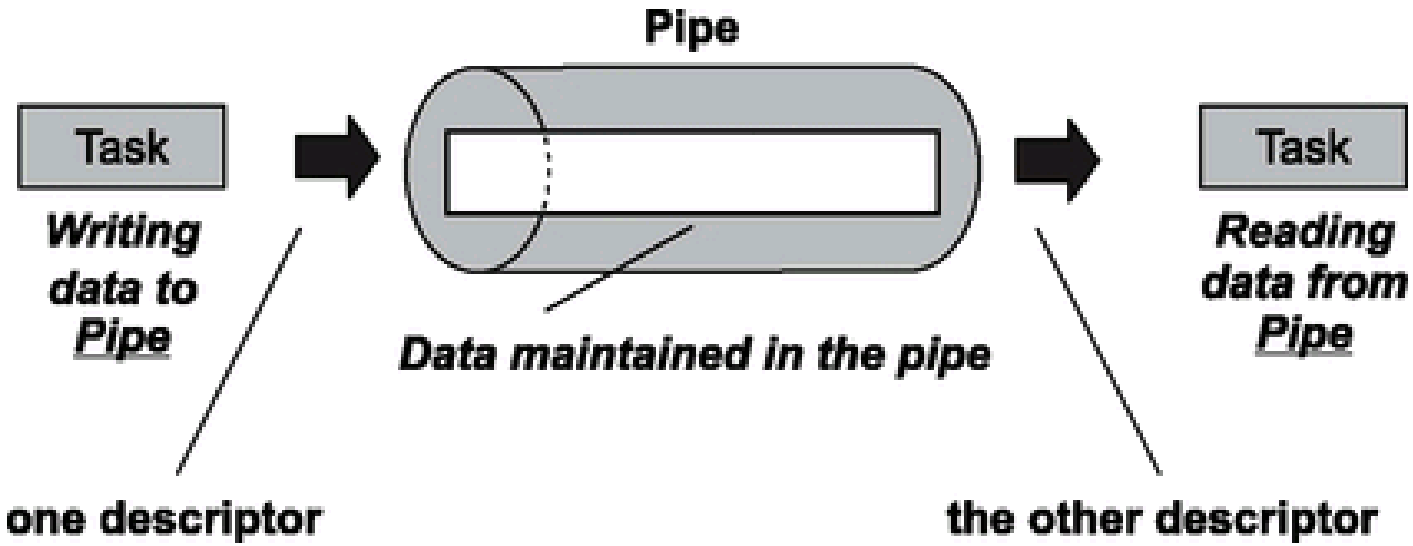
Pipes são objetos do *kernel* que provêem uma troca de dados não estruturados entre tarefas e também facilitam a sincronização entre tarefas

Numa implementação tradicional, um *pipe* suporta a troca de dados de maneira unidirecional

Dois descritores (um para leitura e outro para escrita) são retornados quando um *pipe* é criado

Neste caso, o dado é escrito via um descritor e lido via outro descritor, um em cada ponta do *pipe*

Ilustração de um objeto *pipe*



Pipes (cont...)

A leitura do dado não estruturado do *pipe* é dado for FIFO (*first-in, first-out*)

O mecanismo do *pipe* é simples

- A tarefa que lê fica bloqueada quando faz uma leitura de um *pipe* vazio
- A tarefa que escreve fica bloqueada quando tenta escrever num *pipe* cheio

Uso típico é para implementação de um processo "produtor-consumidor"

É possível ter várias tarefas que escrevem num mesmo *pipe*, bem como várias tarefas que leem num *pipe*

Pipes (cont...)

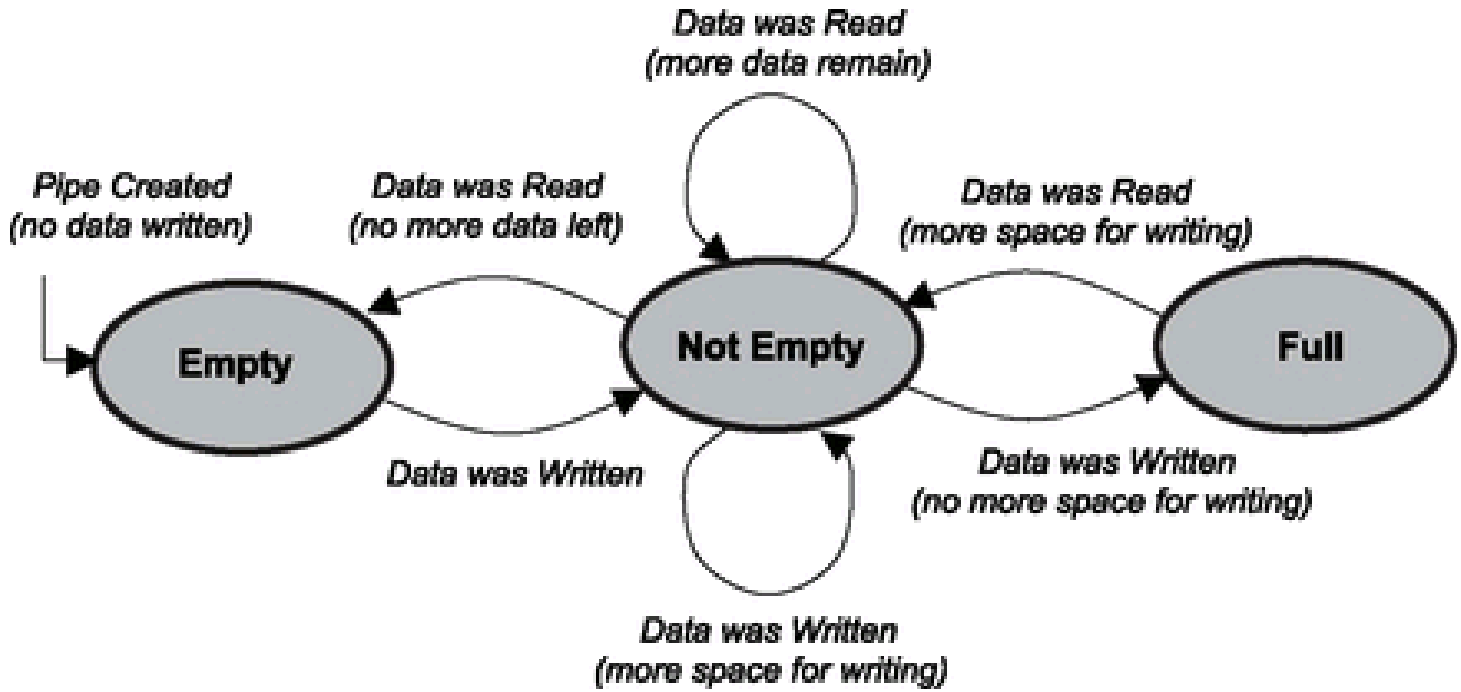
O *pipe* é conceitualmente semelhante a *message queue*, entretanto possui algumas diferenças importantes

Diferente de uma *message queue*, um *pipe* não armazena várias mensagens (o conceito aqui é de um *stream* de dados)

Dados no *pipe* não podem ser priorizados

Fluxo de dados é estritamente baseado em FIFO

Máquina de estados de um *pipe*



Registradores de eventos

Registradores de eventos ou *event registers*

Algumas implementações possuem um registrador especial como parte do *task control block*, chamado *event register*

Consiste de um grupo de *flags* de eventos binários utilizados para rastrear a ocorrência de algum evento específico

Atentar para o fato do tamanho do *event register* ser dependente de implementação (8-, 16-, 32-bit)

Registradores de eventos (cont...)

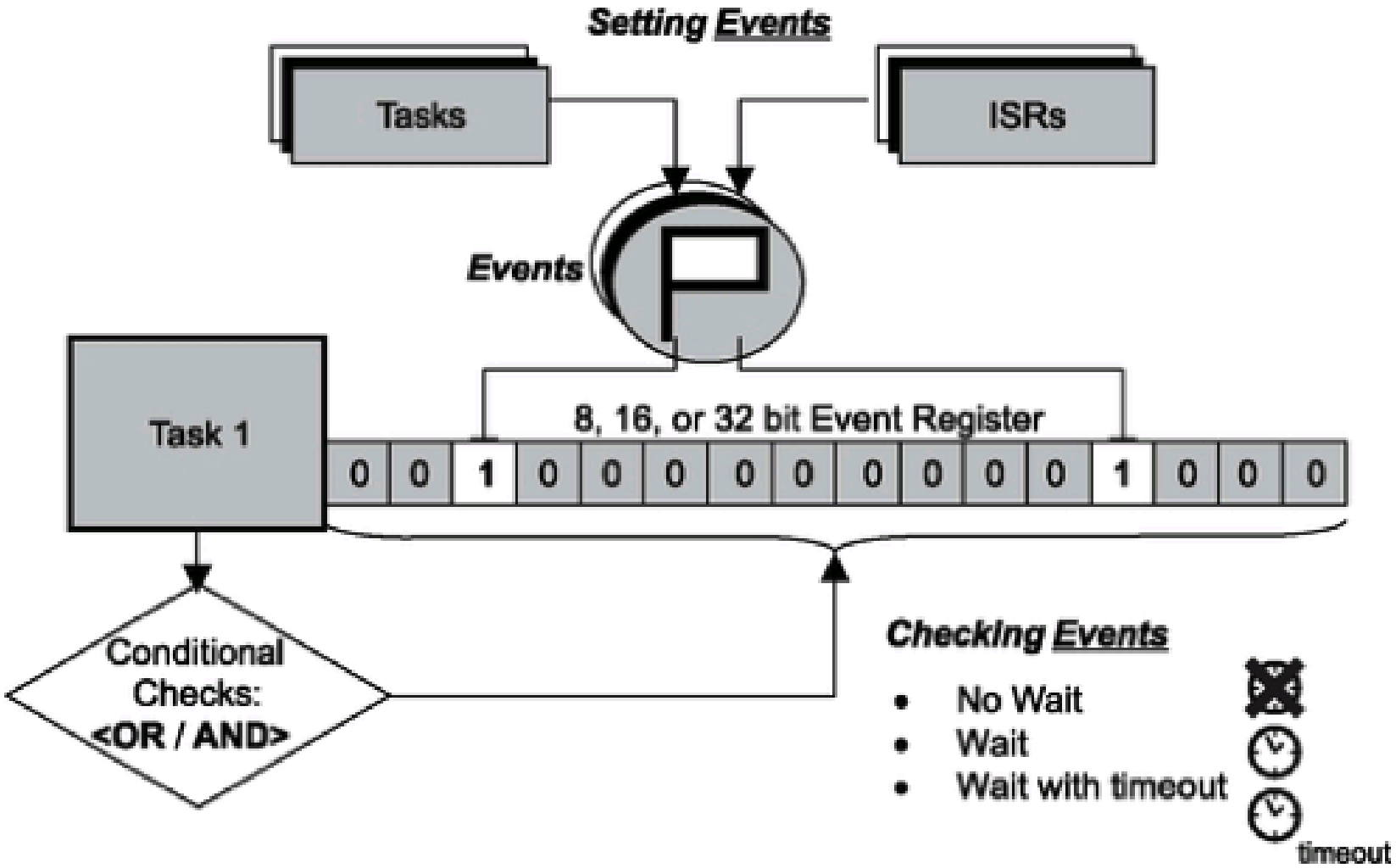
Cada bit no *event register* é tratado como o *flag* binário (*a.k.a., event flag*)

Assim o evento pode ser setado (lógico 1) ou resetado (lógico 0)

Através de um *event register*, a tarefa pode checar a presença de eventos particulares

Uma fonte externa, como uma ISR, pode setar bits no *event register* para informar a tarefa que um evento ocorreu

Ilustração de um objeto *event register*



Registradores de eventos (cont...)

Tipicamente, quando um RTOS possui a implementação de um *event register*, ele associa ao objeto um *control block*

Então, a tarefa pode especificar o conjunto de eventos que ela deseja estar ciente

Assim, os eventos de interesse são mantidos num registrador de eventos recebidos

A tarefa pode indicar um *timeout* para especificar por quanto tempo ela irá esperar pela chegada de um certo evento

Event registers são basicamente utilizadas para sincronização unidirecional

Sinais

Um sinal é uma **interrupção de software** gerada quando um evento ocorre

Ele altera a execução do caminho normal de execução do recebedor para uma rotina de processamento assíncrono responsável por tratar o sinal

Assim como uma interrupção normal, esses eventos são assíncronos para a tarefa recebedora e não ocorrem em um ponto pré-determinado na execução desta tarefa

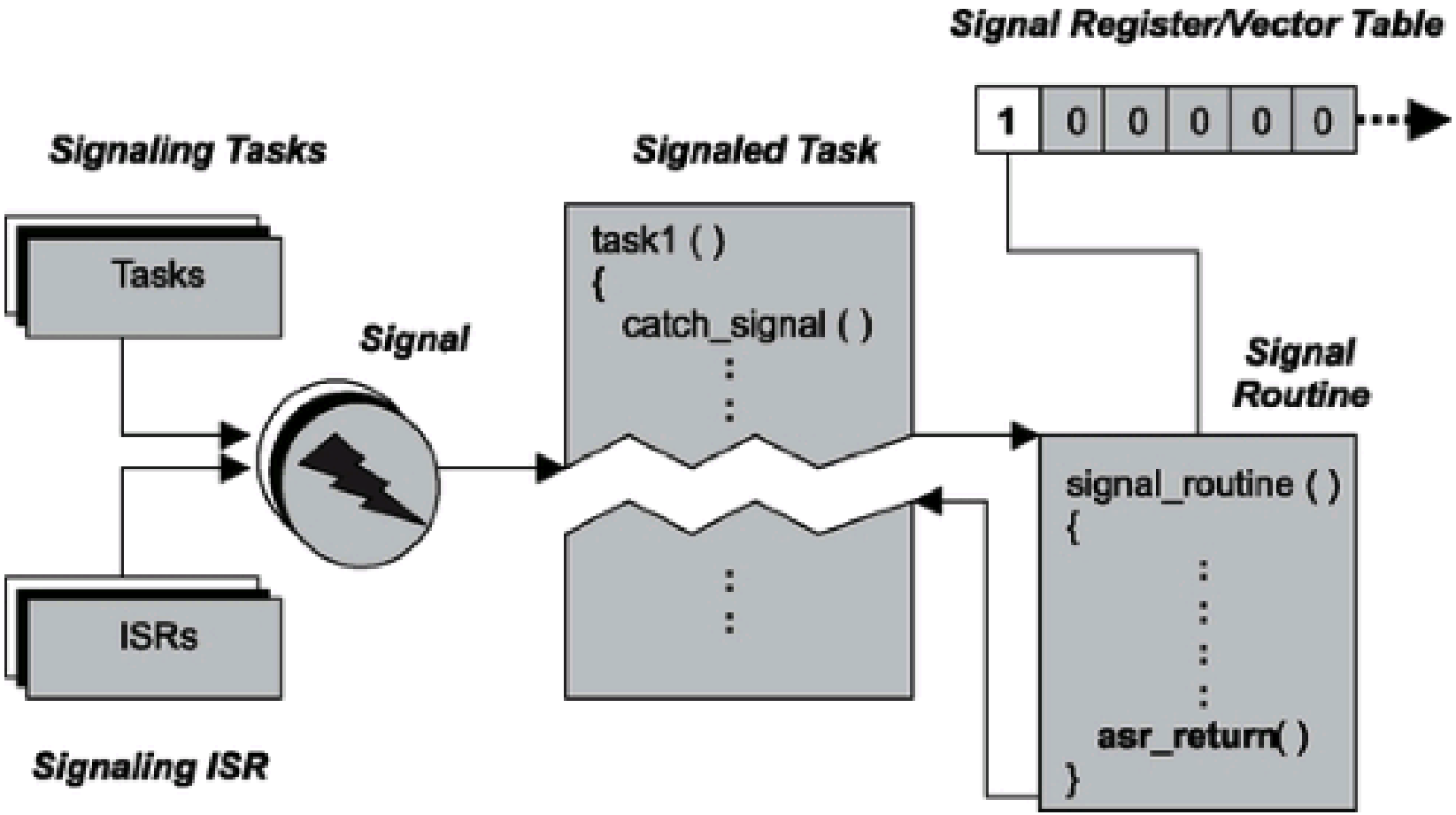
Sinais (cont...)

A diferença entre um sinal e uma interrupção normal/tradicional é que os sinais são chamados **interrupção de software** e são gerados via execução de algum software dentro do sistema

Contrastando com as interrupções normais que são geradas pela chegada de um sinal de interrupção em um pino externo da CPU (por exemplo)

O número e o tipo dos sinas são dependentes de implementação do RTOS

Ilustração de um objeto *signal*



Sinais (cont...)

A utilização de sinais pode ser "custosa" devido a complexidade do objeto sinal para ser utilizado como mecanismo de sincronização inter-tarefas

O sinal altera o estado de execução da tarefa recebedora

Como o sinal ocorre de maneira assíncrona, a tarefa recebedora se torna não-determinística (o que não é desejável em um sistema de tempo real)

Variáveis condicionais

Tarefas, na maioria das vezes, utilizam recursos compartilhados (tais como periféricos e canais de comunicação)

Quando a tarefa precisa utilizar um recurso compartilhado, ela pode querer esperar que o recurso também esteja num estado particular

O jeito que o recurso atinge o tal estado desejado pode ser pelo processamento ou ação de uma outra tarefa

Considerando este contexto, a tarefa precisa de um mecanismo para determinar a condição do recurso compartilhado

Uma possibilidade para isso é a utilização de uma **variável condicional**

Variáveis condicionais (cont...)

Uma variável condicional é um objeto do *kernel* associado a um recurso compartilhado permitindo que uma tarefa espere que outra(s) tarefa(s) crie(m) a condição desejada para o recurso compartilhado

Variáveis condicionais (cont...)

Uma variável condicional implementa um **predicado**

Um predicado é um conjunto de expressões lógicas relativas as condições do recursos compartilhado

O predicado é avaliado e resulta em `TRUE` ou `FALSE`

Uma tarefa avalia um predicado

- Se a avaliação retornar `TRUE`, a tarefa assume que a condição foi satisfeita e continua sua execução
- Se retornar `FALSE`, a tarefa deve esperar por outras tarefas para que estas criem a condição esperada

Variáveis condicionais (cont...)

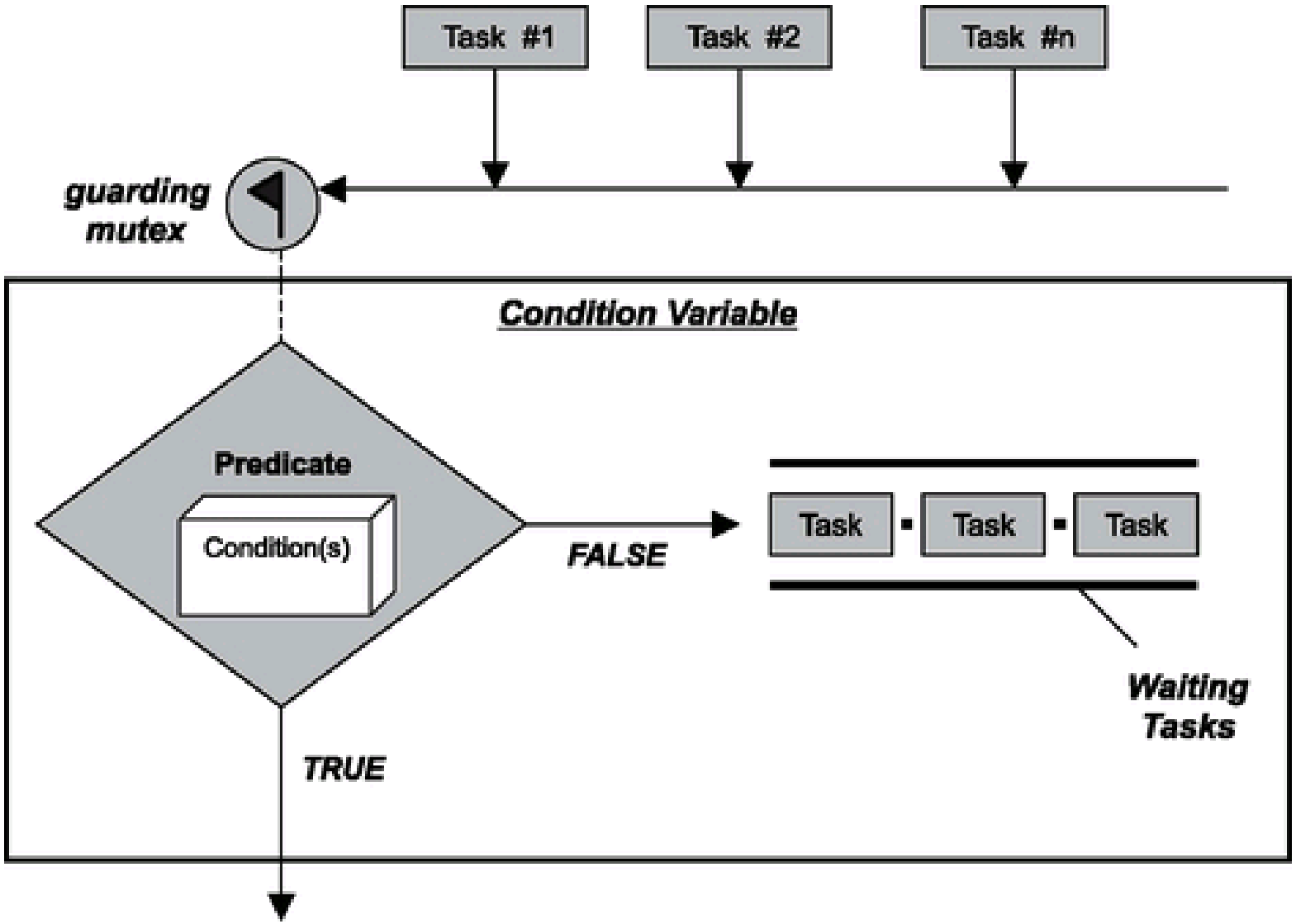
Quando uma tarefa avalia uma variável condicional, a tarefa deve possuir acesso exclusivo a variável condicional

Caso contrário, outra tarefa pode alterar a condição da variável condicional "no mesmo tempo" e gerar uma condição errônea

Por isso, um *mutex* é sempre utilizado em conjunto com uma variável condicional

O objetivo do *mutex* é garantir acesso exclusivo a variável condicional para as tarefas, de maneira que nenhuma outra tarefa fará uso da variável condicional antes que a primeira termine seu uso

Ilustração de um objeto variável condicional



Variáveis condicionais (cont...)

A tarefa deve sempre adquirir o *mutex* antes de avaliar o predicado

Após avaliar o predicado, a tarefa deve liberar o *mutex*

Caso o predicado seja avaliado como `FALSE`, a tarefa deve esperar pela criação da condição desejada

Variáveis condicionais (cont...)

Com a utilização da variável condicional, o *kernel* garante que a tarefa pode liberar o *mutex* e esperar no estado bloqueado (***unlock => block-wait***) pela condição desejada numa operação atômica

A operação atômica é considerada a essência da variável condicional

Uma operação atômica é aquela que não pode ser interrompida

Atentar para o fato de que a variável condicional não é um mecanismo para sincronização de acesso a um recurso compartilhado

Ao invés disso, utiliza-se variável condicional para que as tarefas esperem por um recurso compartilhado num determinado estado

Variáveis condicionais (cont...)

O *kernel* garante a **atomicidade** das operações com a utilização da variável condicional:

- ***unlock-and-wait***, quando o predicado avaliar como `FALSE` e a tarefa for bloqueada e ficar esperando pela condição
- ***resume-and-lock***, quando a tarefa voltar do estado bloqueado (condição satisfeita) e voltar a executar pegando o *mutex* antes de tudo

Ilustração de um objeto variável condicional

```
Task 1
Lock mutex
        Examine shared resource
        While (shared resource is Busy)
                WAIT (condition variable)
        Mark shared resource as Busy
Unlock mutex

Task 2
Lock mutex
        Mark shared resource as Free
        SIGNAL (condition variable)
Unlock mutex
```

A operação `WAIT` coloca a tarefa que a chamou na fila de tarefas em espera dessa variável condicional e faz o `unlock` do *mutex* associado numa operação atômica

Informação ao leitor

Notas de aula baseadas nos seguintes textos:

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