SOFTWARE DE TEMPO-REAL

Professor Paulo Régis

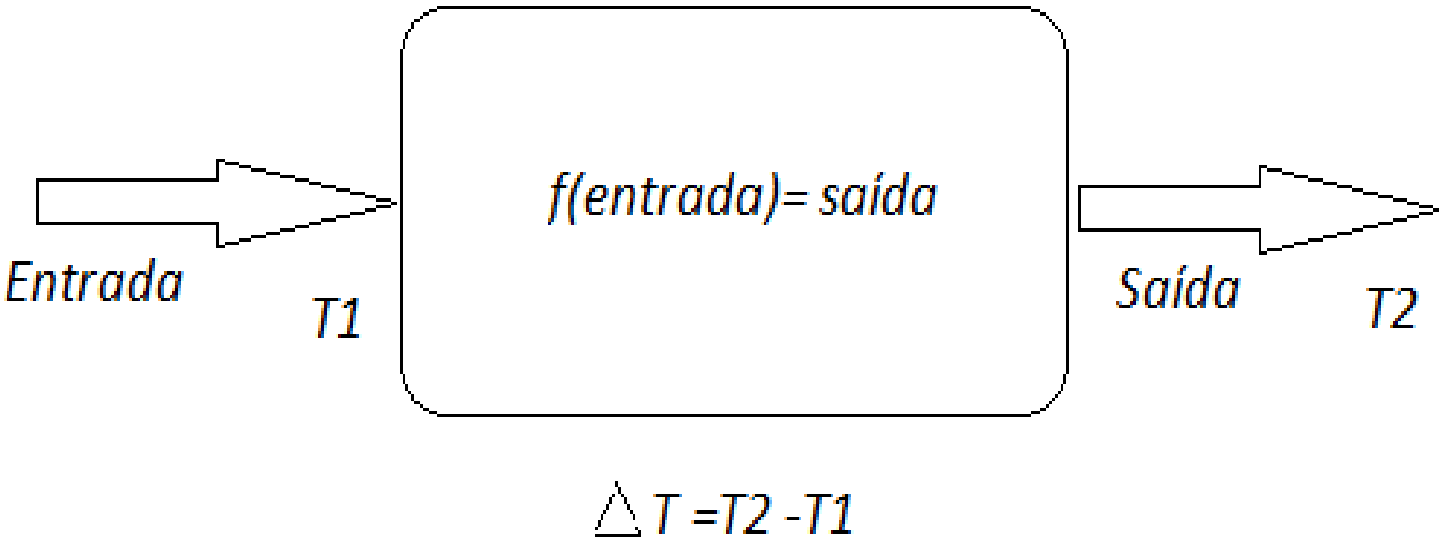
O que são sistemas de tempo-real?

- *Sistema em tempo real é qualquer sistema no qual o tempo cuja saída é produzida é significante.*
- *Qualquer sistema ou atividade de processamento de informação o qual deve responder a um estímulo de entrada gerada externamente dentro de um período finito e especificado.*
- *Conseqüentemente, o funcionamento correto de um sistema de tempo-real não está somente associado à resposta lógica correta, mas também ao tempo no qual a resposta foi produzida.*

Exemplos de aplicações ou sistemas de tempo-real

- *Sistemas de controle de veículos para automóveis, metrôs, aeronaves, ferrovias e navios;*
- *Controle de tráfego para auto-estradas, espaço aéreo, trilhos de ferrovias e corredores de navegação marítima;*
- *Controle de processo para usinas de energia, indústrias químicas e para produtos de consumo, como refrigerantes e cerveja;*

Bloco de um sistema de tempo-real



Tempo de resposta

- *Existem muitas interpretações do conceito exato de sistemas de tempo-real. Mas, todos têm em comum a noção de tempo de resposta. Ou seja, o tempo necessário para o sistema gerar uma saída de alguma entrada associada.*

Sistemas de tempo-real hard e soft

- Os sistemas de tempo-real ainda podem ser classificados como do *tipo Hard eSoft:*
- *O sistema de tempo-real hard é aquele que é imperativo (essencial) que respostas aconteçam dentro de um prazo-limite (deadline) específico.*
- *O sistema de tempo-real soft é aquele em que tempos de resposta são importantes, mas o sistema ainda irá funcionar corretamente se o prazo-limite for ocasionalmente perdido. Ou seja, nesses sistemas não existem deadlines explícitos.*

Exemplos de sistemas de tempo-real hard e soft

- *Como exemplo, podemos citar um sistema de controle de tráfego aéreo como um sistema de tempo-real hard. Pois, nesse sistema, se uma resposta da localização de uma aeronave não for produzida em um prazo específico uma catástrofe poderá acontecer.*
- *Como exemplo de um sistema de tempo-real soft, podemos citar o sistema operacional Windows. Se um atraso na abertura de uma aplicação acontecer, isso não será visto como falha do sistema. Nesse caso, observa-se que não há um prazo-limite específico.*

Características de sistemas de tempo-real

- *Nem todos os sistemas de tempo-real irão exibir as características abaixo. Mas, qualquer linguagem de proposta geral utilizada para desenvolver um sistema de tempo-real deve apresentar facilidades que suportem essas características.*

Características de sistemas de tempo-real

- Tamanho e Complexidade
- *A maioria dos problemas associados com o desenvolvimento de aplicações está relacionada ao tamanho e complexidade.*
- *Aplicações pequenas são fáceis de serem mantidas, recodificadas e entendidas por qualquer pessoa. Mas, uma linguagem que se proponha ao desenvolvimento de um sistema de tempo-real deve oferecer facilidades para minimizar o tamanho e complexidade de uma aplicação. Como por exemplo, oferecer facilidades para modularizar o código, oferecer número reduzido de instruções, apresentar funções eficientes, entre outras. A complexidade pode ser diminuída utilizando-se uma linguagem com poucas funções eficientes.*

Características de sistemas de tempo-real

- Operações com Números Reais
- A maioria dos sistemas de tempo-real monitoram ou controlam eventos do ambiente externo. Em geral, esse monitoramento pode ser realizado por sensores, cujas saídas podem ser dadas por um sinal de corrente ou tensão. Como exemplo, por correntes variando entre 4 a 2OmA ou tensões variando entre O a 10V.
- Assim, uma linguagem que se propõe ao desenvolvimento de sistemas de tempo-real, deve ser capaz de oferecer facilidades para operação com números reais.

Características de sistemas de tempo-real

- Confiabilidade e Segurança
- Em alguns exemplos de sistemas de tempo-real, uma falha pode ocasionar perdas de vidas humanas, danos ao meio ambiente ou grandes danos financeiros. Assim, esses sistemas podem ser considerados como sistemas críticos.
- *Confiabilidade é a probabilidade de operação livre de falhas durante um tempo espec em um dado ambiente, para um propósito específico.*
- *Segurança reflete a capacidade do sistema operar de forma normal e anormalmente, sem oferecer ameaças as pessoas ou ao ambiente.*
- *Disponibilidade.*

Características de sistemas de tempo-real

- Controle Concorrente de Componentes
- Sistemas de tempo-real lidam com eventos do mundo real, muitos deles ocorrendo simultaneamente.
- *Ou seja, dois ou mais eventos podem ocorrer simultaneamente. Assim, o hardware e o software devem apresentar técnicas ou implementações que permitam a concorrência desses processos.*
- *Assim sendo, um hardware que apresentasse múltiplas entradas e saídas, com um SO que permitisse escalonamento de processos e uma linguagem que apresentasse funções para criação de threads, permitiria uma concorrência em nível de hardware e software.*

Características de sistemas de tempo-real

- Linguagens e SO´s para tempo-real:
- *- RTOS: RTLinux QNX, CMX (da CMX company), AMX (da KADAK), Femto OS, Neutrino, entre outros;*
- *- Linguagens para STR: OCCAM, MODULA, ADA, MESA, entre outras.*

Características de sistemas de tempo-real

- Facilidades de Tempo-Real
- Tempo de resposta é essencial em uma aplicação de tempo-real. Para isso, o hardware utilizado deve apresentar bom desempenho, além da linguagem e do suporte de run-time apresentar facilidades como:
- - Especificar tempos nos quais as ações são executadas;
- - Especificar tempos nos quais as ações devem ser finalizadas;
- - Responder a situações em que todos os requisitos de tempo não podem ser atendidos;
- - Responder a situações em que os requisitos de tempo são mudados dinamicamente.

Características de sistemas de tempo-real

- Interação com Interfaces de Hardware
- Sistemas de tempo-real interagem intimamente com o mundo externo. Para isso, eles devem apresentar interfaces para monitoramento e controle de eventos do meio externo. Como exemplo, conversores AD podem ser adicionados ao sistema para monitorar variáveis externas. O sistema também pode oferecer o conceito de interrupção para monitorar eventos esporádicos, característicos do mundo externo.
- A linguagem utilizada também pode apresentar facilidades para programação em baixo-nível. Programação em baixo-nível permite melhor acesso e operação ao hardware do sistema.

Características de sistemas de tempo-real

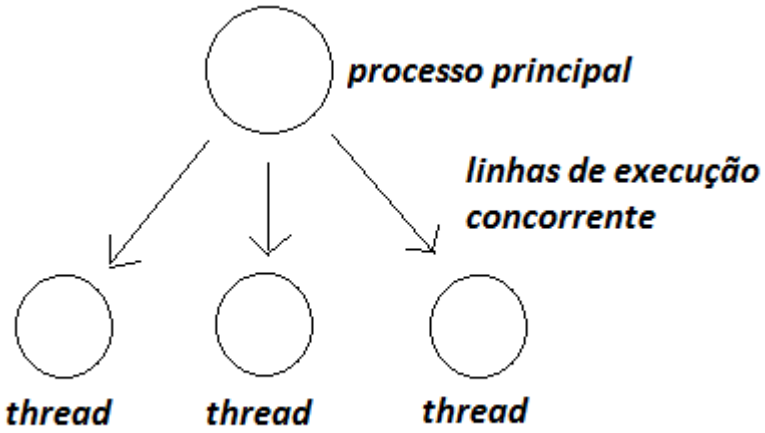
- Implementação Eficiente
- Desde que sistemas de tempo-real são críticos no tempo, uma implementação eficiente da aplicação torna-se necessária. Em muitas situações, a utilização de uma linguagem de alto nível deve ser evitada, pois pode ocasionar um tempo de resposta de milisegundos, em vez de microsegundos, como requerido pela aplicação.
- Como exemplo, uma aplicação simples em C pode gerar um tempo de resposta bem menor do que a mesma aplicação desenvolvida em JAVA.

Arquitetura e Programação Concorrente

- **A Noção de Processo**
- *Um programa concorrente pode ser conceituado como um conjunto de processos seqüenciais autônomos executando em paralelo*. Toda linguagem de programação concorrente deve incorporar, implícita ou explicitamente, a noção de processo. *Cada processo tem sua própria linha de execução de controle, chamada de thread.*

Arquitetura e Programação Concorrente

- *Processo é o objeto ativo de um sistema e é a unidade lógica de trabalho escalonada pelo sistema operacional*.



Arquitetura e Programação Concorrente

- **Processos e Modelos de Sistemas Baseados em Estados**
- ***Um processo pode ter uma ou mais linhas de execução, chamadas de threads. Uma linguagem que permite apenas uma linha de execução em um processo é chamada de monothread. Caso a linguagem permita a criação de duas ou mais threads, ela é chamada de multithread.***

Arquitetura e Programação Concorrente

- **O processo pode apresentar alguns estados:**
- *- Criado: neste estado, o processo pai está criando a thread que é levada a fila de prontos;*
- *- Executando: neste estado a linha de execução está usando a CPU;*
- *- Pronto: neste estado, a linha de execução avisa a CPU que pode entrar no estado de*
- *execução e entra na fila de prontos;*
- *- Bloqueado: neste estado, por algum motivo, a CPU bloqueia a linha de execução,*
- *geralmente enquanto aguarda algum dispositivo de I\O;*
- *- Término: neste estado são desativados os contextos de hardware e a pilha é deslocada*.

Arquitetura e Programação Concorrente

- **Processos Periódicos e Esporádicos**
- *Sistemas de tempo-real geralmente contêm dois tipos de processos: periódicos e esporádicos. Processos periódicos são ativados de forma regular entre intervalos fixos de tempo.*

Arquitetura e Programação Concorrente

- *Em contraste, processos esporádicos são dirigidos por eventos; eles são ativados por um sinal externo ou uma mudança de alguma relação. Um processo esporádico poderia ser usado para reagir a um evento indicando uma falha de alguma peça de equipamento ou uma mudança no modo de operação de um sistema.*

Arquitetura e Programação Concorrente

- *Na sua forma mais simples, um processo periódico P é caracterizado por uma tripla (c, p, d), onde c representa o tempo de computação, usualmente uma estimativa do pior caso, para P código; p é o período ou ciclo de tempo e d é o prazo-limite (deadline), com c <= d <= p. O significado é que o processo é ativado a cada unidade p de tempo e sua computação c deve ser completada antes que seu prazo-limite d expire. O período e o prazo-limite são obtidos ou derivados a partir dos requisitos do sistema; p e d são geralmente idênticos*.

Arquitetura e Programação Concorrente

- *Um processo esporádico é também representado por uma tripla (c, p, d), c <= d <= p. Aqui, c e d têm um significado similar ao do caso periódico. Na ocorrência de um evento, a computação deve ser concluída dentro do prazo-limite especificado; isto é, o tempo de conclusão* ***t*** *deve satisfazer*
- ***t <= te + d,***
- *onde* ***te*** *é o tempo de ocorrência de evento.* ***p*** *representa o tempo mínimo entre eventos; isto é, eventos sucessivos estão separados por, no mínimo,* ***p****.*

Arquitetura e Programação Concorrente

- **Execução Concorrente**
- *Embora construções para programação concorrente variem de uma linguagem para outra, existem 3 facilidades fundamentais que devem ser produzidas:*
- *- A expressão de execução concorrente através da noção de processo;*
- *- Sincronização de processos;*
- *- Comunicação inter-processo.*
- *Em relação à interação de processos, eles podem ser caracterizados da seguinte maneira:*
- *- Independentes;*
- *- Cooperantes;*
- *- Competidores.*

Arquitetura e Programação Concorrente

- **Representacão de Processos**
- **Fork e Join:**
- . A estrutura Fork especifica que a rotina projetada deveria iniciar executando concorrentemente com o invocador do Fork. A estrutura Join permite o invocador para sincronizar com a finalização da rotina invocada. Como exemplo:
- Função F retum...;
- Procedure P;
- -
- C:= fork F;
- -
- -
- J:= join C;
- -
- End P;
- Entre a execução do fork e do join, a procedure P e a função F irão executar em paralelo. No ponto do join a procedure irá esperar até a função finalizar (se ela já não tiver finalizado). A notação fork e join pode ser achada na linguagem Mesa. A versão de fork e join pode também ser encontrada no UNIX.

Arquitetura e Programação Concorrente

- **- Cobegin:**
- O cobegin (ou parbegin ou par) é um caminho estruturado de denotação de execução concorrente de uma coleção de estruturas:
- Cobegin
- S1;
- S2;
- S3;
- Sn;
- Coend
- A estrutura cobegin pode ser encontrada em OCCAM2.

Arquitetura e Programação Concorrente

- **Declaração de Processo Explícito:**
- O exemplo abaixo foi desenvolvido em linguagem Modula-1.

```
MODULE main;
   TYPE dimension = (xplane, yplane, zplane);
   PROCESS control (dim: dimension);
     VAR position: integer;
     setting : integer;
     BEGIN
        position:= 0 ;
        LOOP
            newsetting(dim, setting);
            position:= position + setting;
            move_arm (dim, position);
        END
    END control;
BEGIN
      control (xplane);
      control (yplane);
      control (zplane);
END main.
```

Confiabilidade e Tolerância a Faltas

- Se um sistema de controle tem que responder a um evento em um determinado tempo, a falha desse sistema impossibilitará a entrega dessa resposta no tempo apropriado. Como exemplo, um radar em uma sala de controle aéreo tem que varrer uma região a cada 3 segundos. Uma falha desse sistema impossibilitará a visualização de aeronaves nesse intervalo de tempo, podendo ocasionar um desastre aéreo.

Confiabilidade e Tolerância a Faltas

- **Fontes de faltas:**
- Existem, em geral, 4 fontes de faltas que podem resultar em uma falha do sistema de tempo-real:
- - Especificação inadequada. A maioria das faltas de software é originada por especificação inadequada.
- - Faltas introduzidas de erros de projeto em componentes de software.
- - Faltas introduzidas pela falha de um ou mais componentes processadores no sistema de tempo-real.
- - Faltas introduzidas por interferência permanente ou transiente no subsistema de comunicação.

Confiabilidade e Tolerância a Faltas

- **Diferenciando falta, erro e falha:**
- Uma falta poderia ser exemplificada como uma imperfeição física ou mecânica de algum componente do sistema ou a um erro no algoritmo de software do sistema.
- Um sistema de tempo-real pode ser caracterizado por um certo número de estados internos e externos. Em um determinado instante, um estado do sistema não esperado pode ser caracterizado por um estado errôneo, ou seja, por um erro.
- Uma falha é caracterizada por um desvio de comportamento do sistema no qual ele foi especificado para isso.

Confiabilidade e Tolerância a Faltas

- Exemplificando falta, erro e falha.

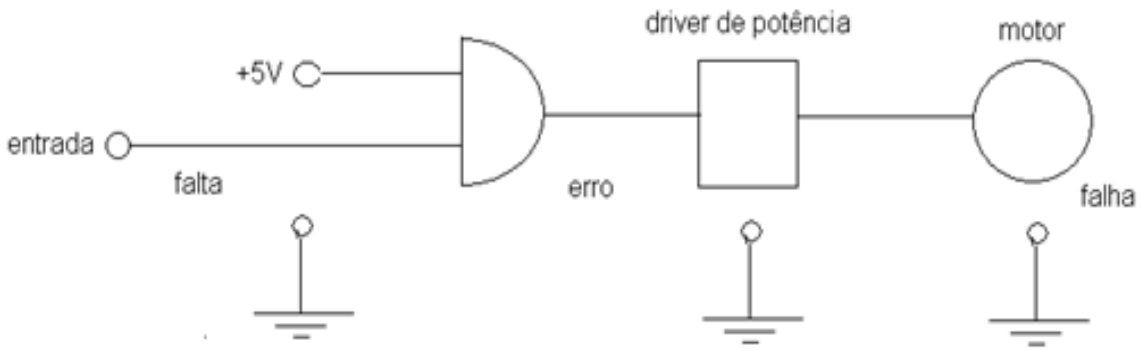


Figura 2. Diagrama para exemplificar falta, erro e falha.

Confiabilidade e Tolerância a Faltas

- **Tipos de faltas:**
- **Faltas transientes**: uma falta transiente inicia em um tempo particular, permanece no sistema por algum tempo e então desaparece. Exemplos de tais faltas estão em componentes de hardware os quais tem uma reação adversa a alguma interferência externa, tais como campos elétricos ou eletromagnéticos. Muitas faltas em sistemas de comunicação são transientes.

Confiabilidade e Tolerância a Faltas

- ***Faltas permanentes:*** faltas permanentes iniciam em um determinado tempo e permanecem no sistema até serem reparadas. Como exemplo, um fio partido ou um erro de projeto de software.
- ***Faltas intermitentes: s***ão faltas transientes que ocorrem de tempo em tempo. Um exemplo é um componente de hardware que é sensível ao calor. Ele trabalha por um período, aquece, pára de trabalhar, volta a esfriar e então trabalha novamente.

Confiabilidade e Tolerância a Faltas

- Prevenção de Faltas:
- Essa técnica permite eliminar qualquer possibilidade de faltas no sistema antes de se tornar operacional.
- Existem dois estágios de prevenção de faltas: ***fault avoidance*** (evitar faltas) e ***fault removal*** (remoção de faltas).

Confiabilidade e Tolerância a Faltas

- ***Fault avoidance*** é a técnica que possibilita limitar a introdução de potenciais componentes defeituosos durante a construção do sistema. Para o hardware, temos:
- - O uso de componentes mais confiáveis dentro do custo e desempenho exigidos;
- - O uso de técnicas refinadas para interconexão de componentes e para a montagem de subsistemas;
- - Encapsulamento do hardware para diminuir efeitos de interferências externas

Confiabilidade e Tolerância a Faltas

- Com relação ao software, faltas podem ser evitadas da seguinte maneira:
- - Especificação de requisitos rigorosos, senão formais;
- - Uso de metodologias de projeto comprovadas;
- - Uso de linguagens que facilitem a abstração de dados e modularidade;
- - Uso de ambientes de suporte de projeto que ajudem a desenvolver os componentes de software e também gerenciar a complexidade;

Confiabilidade e Tolerância a Faltas

- Técnicas de Remoção de Faltas:
- Mesmo com o uso de técnicas para se evitar faltas, faltas irão inevitavelmente estar presentes no sistema depois de sua construção. Em particular, devido a erros de projetos em componentes de hardware e software. Assim, um segundo estágio de prevenção de faltas é empregado. É a técnica de *remoção de faltas*.

Confiabilidade e Tolerância a Faltas

- Na técnica de remoção de faltas, testes são realizados e faltas são retiradas quando encontradas. Alguns problemas podem existir:
- Um teste pode somente ser usado para mostrar a presença de falta, não sua ausência;
- Em algumas situações, é impossível testar em condições realistas;
- Erros que são introduzidos em tempo de projeto, mas que só se manifestam quando o sistema passa a operar.
- Obs: Mesmo com o uso de técnicas de prevenção de faltas, faltas podem existir e levar o sistema a falhar.

Confiabilidade e Tolerância a Faltas

- ***Tolerância a faltas:***
- Por causa das inevitáveis limitações de técnicas de prevenção de faltas, projetistas de sistemas de tempo real devem considerar o uso de técnicas de tolerância a faltas.
- Existem alguns níveis diferentes de tolerância a faltas que podem ser produzidas para um sistema:
- ***falha operacional (fail operational):*** O sistema continua a operar na presença de faltas, por um período limitado de tempo, sem perda significante de sua funcionalidade ou desempenho;
- ***falha suave(failsoft):*** O sistema continua a operar na presença de erros, aceitando uma degradação de sua funcionalidade ou desempenho durante a recuperação ou reparo.
- ***falha segura (failsafe):*** O sistema mantém sua integridade enquanto aceita uma parada temporária em sua operação.

Confiabilidade e Tolerância a Faltas

- **Redundância**:
- Todas as técnicas para se conseguir tolerância a faltas dependem de elementos extras que são introduzidos no sistema para detectar e se recuperar de faltas. Esses componentes são redundantes, mas não são requeridos no sistema para que este tenha modo de operação normal.

Confiabilidade e Tolerância a Faltas

- A redundância pode ser classificada como: *redundância estática e redundância dinâmica.*
- **Redundância Estática:**
- Na redundância estática, os componentes redundantes são usados dentro do sistema para esconder os efeitos das faltas. Um exemplo de redundância estática é a *TMR.* TMR consiste de três subcomponentes idênticos e um circuito de votação majoritária. Esse circuito de votação compara a saída de todos os componentes, e se uma saída difere das outras duas, essa saída é mascarada.

Confiabilidade e Tolerância a Faltas

- Redundância Dinâmica:
- *Redundância dinâmica* é uma redundância produzida dentro do componente o qual indica explícita ou implicitamente que a saída está errada. Essa redundância fornece facilidades de detecção de erro em vez de facilidades de mascaramento de erro. Recuperação deve ser produzida para qualquer componente.

Confiabilidade e Tolerância a Faltas

- **Programação N-Versão (redundância estática de software):**
- **O sucesso de TMR e NMR de hardware tem motivado uma abordagem similar para tolerância a faltas de software. Embora software não se degrade com o uso, essa abordagem permite detectar faltas de projeto. *Programação N-versão é definida como a produção independente de N programas equivalentes funcionalmente de uma mesma especificação inicial*.**

Confiabilidade e Tolerância a Faltas

- *Programação N-Versão:*
- *Uma vez projetado e escrito, os programas executam concorrentemente com as mesmas entradas e seus resultados são comparados por um processo driver.* Em princípio, os resultados deveriam ser idênticos, mas na prática pode existir alguma diferença. O resultado consensual (da maioria) é considerado como o correto.

Confiabilidade e Tolerância a Faltas

- Programação N-Versão:
- Para a produção de versões diferentes, as N versões podem ser desenvolvidas em linguagens diferentes, ou com mesma linguagem, mas com compiladores diferentes, ou até com mesma linguagem e compilador, mas com escopos diferentes.

Confiabilidade e Tolerância a Faltas

- Um programa N versão é controlado por um processo driver o qual é responsável por:
- invocar cada uma das versões;
- esperar cada versão finalizar;
- comparar e agir sobre o resultado.

Confiabilidade e Tolerância a Faltas

- Componentes do processo driver e das versões:
- - Vetores de comparação;
- - Indicadores de status de comparação;
- - Pontos de comparação.

Confiabilidade e Tolerância a Faltas

- Vetores de comparação são estruturas de dados os quais representam as saídas, ou votos, das versões. Esta comparação deve ser realizada pelo driver.
- Indicadores de status de comparação são comunicados do driver para as versões; eles indicam que ações cada versão deve desempenhar como resultado da comparação do driver. Tais ações vão depender da comparação, é podem ser:
- continuação da execução da versão;
- término de uma ou mais versões;
- continuação e posterior mudança de um ou mais votos para valor majoritário.
- Pontos de comparação são os pontos nas versões em que eles devem comunicar seus votos para o processo driver.

Confiabilidade e Tolerância a Faltas

- Programa N-versão:

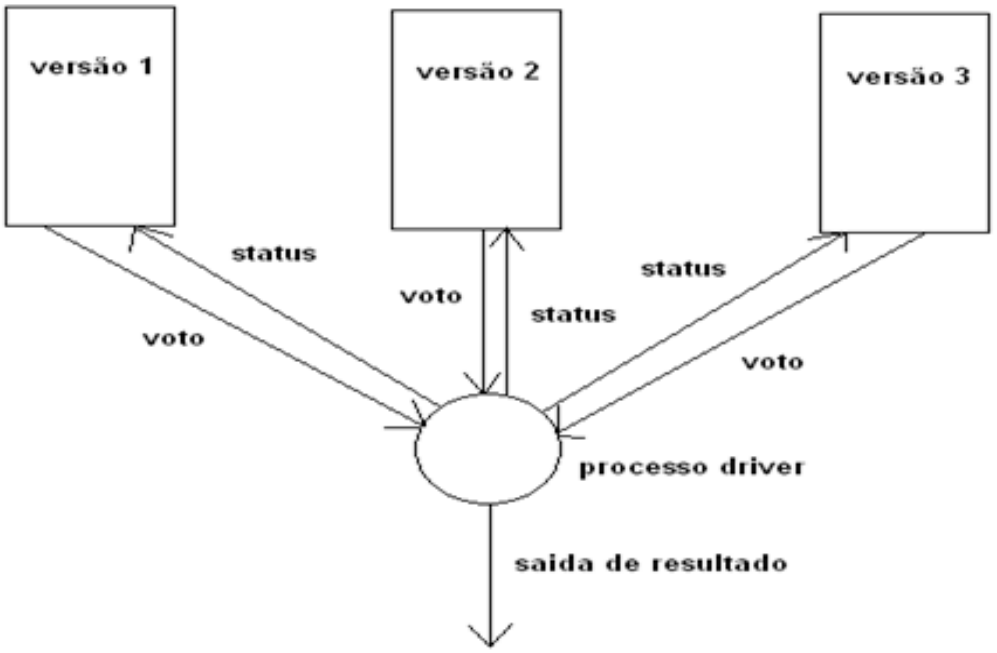


Figura 3. Diagrama de blocos para exemplificar programação N-versão.

Confiabilidade e Tolerância a Faltas

- **Redundância Dinâmica de Software:**
- Com redundância dinâmica, os componentes redundantes só irão operar se um erro for detectado.
- Esta técnica de tolerância a faltas possui quatro fases:
- *Detecção de erro;*
- *Avaliação e confinamento de danos;*
- *Recuperação de erro;*
- *Tratamento de falta e serviço continuado.*

Confiabilidade e Tolerância a Faltas

- Duas classes de técnicas de detecção de erros podem ser identificadas:
- ***- detecção de ambiente:*** Existem erros que são detectados no ambiente no qual o programa executa. Como exemplo, podemos citar: execução ilegal de instruções; overflow aritmético; violação de proteção; valor fora de range; ponteiro nulo não referenciado, entre outros.
- ***- detecção de aplicação:*** Existem erros que são detectados pela própria aplicação.

Confiabilidade e Tolerância a Faltas

- Detecção de aplicação:
- Muitos testes podem ser realizados na aplicação para detecção de erros:
- Teste de replicação;
- Teste de temporização (tempo);
- Teste reverso;
- Teste de codificação;
- Teste de razoabilidade;
- Teste estrutural;

Confiabilidade e Tolerância a Faltas

- **Avaliação e Confinamento de Dano:**
- Como pode existir algum atraso entre a ocorrência de uma falta e o aparecimento do erro, é necessário avaliar qualquer dano que tenha ocorrido. Uma informação errônea poderia se espalhar através do sistema e dentro do seu ambiente. Então, avaliação de dano está fortemente relacionada a precauções de confinamento de dano.

Confiabilidade e Tolerância a Faltas

- **Avaliação e confinamento de danos:**
- Existem duas técnicas que podem ser usadas para estruturação de sistemas, as quais irão ajudar no confinamento de danos: *decomposição modular e ações atômicas.* Decomposição modular é uma técnica que possibilita quebrar o sistema em componentes menores, em que cada componente pode representar um ou mais módulos.
- *A atividade de um componente é dita ser atômica se não existe interação entre a atividade e o sistema durante a ação*. Nesse caso, para o resto do sistema, uma ação atômica aparenta ser indivisível e executada instantaneamente. Nenhuma informação pode ser passada da ação atômica para o sistema ou vice-versa.

Confiabilidade e Tolerância a Faltas

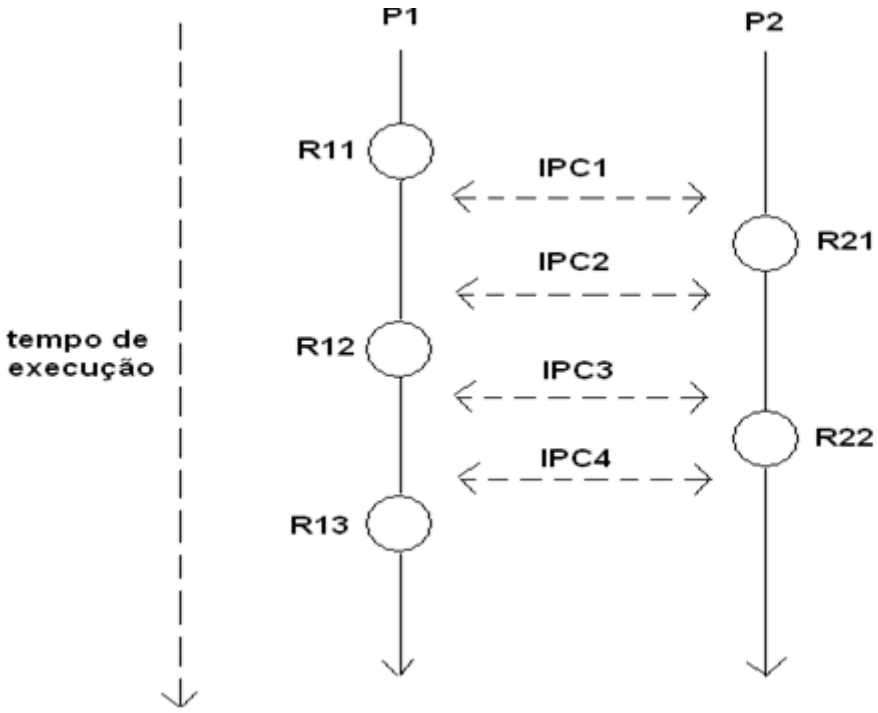
- **Recuperação de Erro**
- *É a fase que deve converter um estado errôneo do sistema em um estado de operação normal, embora haja uma degradação do serviço*. Duas abordagens para recuperação de erro são utilizadas: *recuperação para frente e recuperação para trás*.

Confiabilidade e Tolerância a Faltas

- *Recuperação de erro para frente permite continuar de um estado errôneo pela correção seletiva para o estado do sistema*.
- *Recuperação de erro para trás permite restabelecer o sistema a um estado seguro antes do erro ocorrido*. Uma seção alternativa do programa é então executada. A seção alternativa a ser executada deve ter algoritmo diferente da seção que causou o erro. Como na programação N-versão, é desejável que a seção alternativa não resulte na mesma falta recorrente. *O ponto no qual o processo deve ser restabelecido é chamado de ponto de recuperação e a ação de substituição é chamada de check-pointing*.

Confiabilidade e Tolerância a Faltas

- Problemas de IPC:
-



Confiabilidade e Tolerância a Faltas

- **Tratamento de Falta e Serviço Continuado**
- Tratamento de faltas pode ser dividido em dois estágios: localização da falta e reparo do sistema. Técnicas de detecção de erro podem ajudar a encontrar a falta do componente. Para um componente de hardware, essa localização deve ser precisa, com o reparo sendo feito pela substituição do componente. Uma falta de software pode ser removida em uma nova versão do código.

Confiabilidade e Tolerância a Faltas

- **Blocos de Recuperação:**
- **Técnica de recuperação de erro para trás.**

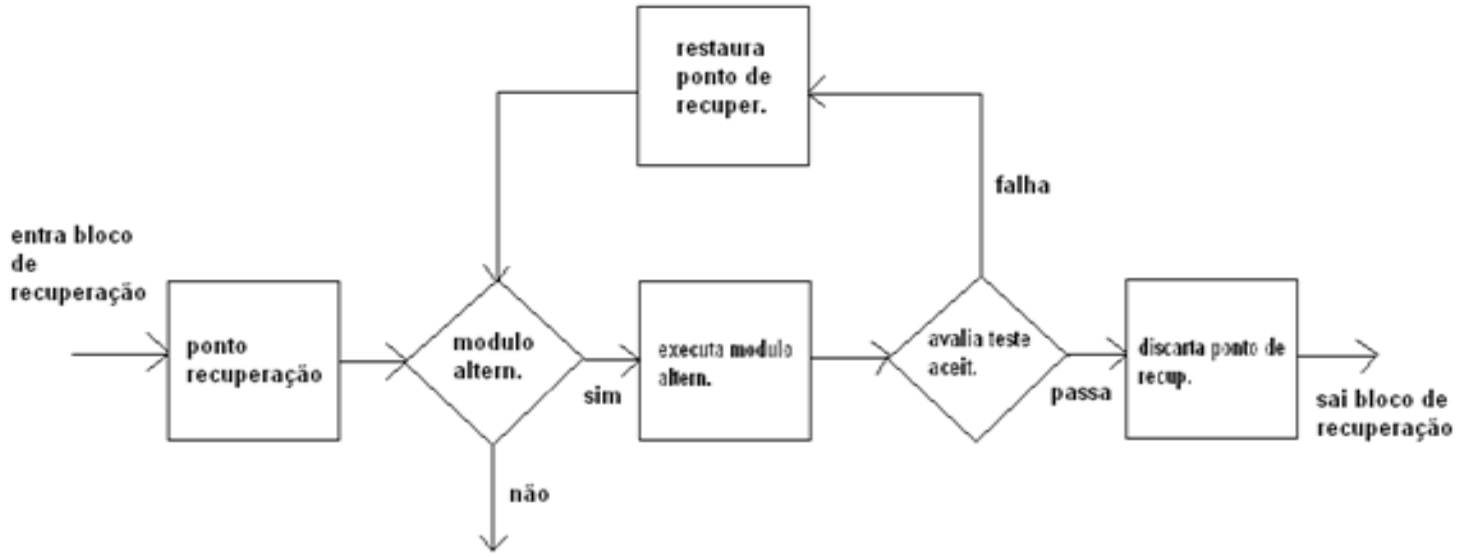


Figura 5. Diagrama de blocos representado blocos de recuperação.

Confiabilidade e Tolerância a Faltas

- *Blocos de recuperação são blocos no sentido de uma linguagem de programação normal, exceto que a entrada de um bloco é um ponto de recuperação automático e a saída é um teste de aceitação*. *Teste de aceitação é usado para testar se o sistema está em um estado aceitável depois da execução do bloco, ou módulo primário, como é geralmente chamado*.

Confiabilidade e Tolerância a Faltas

- O escopo de código abaixo apresenta um exemplo de bloco de recuperação:
- Ensure (teste de aceitação)
- By
- (módulo primário)
- Else by
- (módulo alternativo)
- Else by
- (módulo alternativo)
- Else by
- (módulo alternativo)
- -
- -
- -
- Else error

Confiabilidade e Tolerância a Faltas

- **Exceção:**
- Vimos anteriormente que um erro é uma manifestação de uma falta, e que uma falta é um desvio de especificação de um componente. Os erros podem ser antecipados ou não antecipados. *Uma exceção pode ser definida como a ocorrência de um erro*. A apresentação de uma condição de exceção para o invocador da operação é chamada de *raising the exception (aumentar a exceção) e a resposta do invocador é chamada de handling the exception (manipulação de exceção).*

Confiabilidade e Tolerância a Faltas

- Abaixo, é apresentado um escopo de código que exemplifica a exceção e sua manipulação.
- if chamada_função(parâmetro) = ERRO então
- Código de manipulação de erro
- senão
- Código de retorno normal
- fim;
-

Software de Tempo Real

- **Sincronização e Comunicação baseada em Memória Compartilhada**
- A maior dificuldade associada à programação concorrente deve-se à interação de processos concorrentes.
- A troca de dados ou informações entre processos é chamada de comunicação entre processos. Essa troca de dados pode ser por variável compartilhada ou passagem de mensagem.
- A sincronização de processo pode ser representada por um processo que deve aguardar pela ação de outro processo ou do próprio sistema.
- Variável compartilhada é uma variável que vários processos podem acessar.

Software de Tempo Real

- **Exclusão Mútua**
- Embora variáveis compartilhadas sejam uma boa facilidade para passagem de informações entre processos, elas apresentam problemas sérios quando há alterações múltiplas de seus valores.
- Como a alteração de seu valor não é uma ação única e indivisível, a alteração múltipla pode apresentar leituras erradas de seus valores. Como exemplo, suponha dois processos que queiram executar a seguinte atribuição: X = X+1.
- A seqüência de estruturas que deve aparecer para ser executada indivisivelmente é chamada de seção crítica. A sincronização requerida para proteger uma seção crítica é chamada de exclusão mútua.

Software de Tempo Real

- **Condição de Sincronização**
- Muitas vezes, um processo deve esperar até que outro processo execute certa ação. Nesse caso, não há necessidade de exclusão mútua, pois os dois processos não concorrem a uma variável única. *Esse é um caso típico de condição de sincronização, em que um processo deve esperar pela ação de um outro processo*.
- A implementação de qualquer forma de sincronização implica que processos devem, em tempo, serem seguros até que seja apropriado para prosseguirem. *Exclusão mútua e condição de sincronização devem ser programadas usando* ***busy wait loops e flags.***

Software de Tempo Real

- **Busy Waiting**
- Um caminho para implementar sincronização é ter processos alterando e testando variáveis compartilhadas que estão atuando como flags.
- Para sinalizar uma condição, um processo seta (coloca em um) o valor de uma flag. Para esperar por esta condição qualquer outro processo testa esta flag, e prossegue somente quando o valor apropriado é lido. Observe o exemplo abaixo:

```
process P1;   (* processo esperando)
   -
   while flag = down do
        null
   end;
   -
end P1;
process P2;   ( * processo sinalizando)
   -
   flag: = up
   -
   -
end P2;
```

Software de Tempo Real

- **Flags e Busy Waiting**
- Vamos exemplificar exclusão mútua através de dois processos que têm mutuamente seções críticas. Vejamos exemplo abaixo:

```
process P1;
  loop
     flag1:= down
     while flag 2= down do
        null
     end;
     <seção crítica>
     flag1:=up;
     <seção não-crítica>
     -
   end
end P1;
process P2;    loop
     flag2:= down
     while flag 1= down do
        null
     end;
     <seção crítica>
     flag2:=up;
     <seção não-crítica>
     -
   end
end P2;
```

- Qual fenômeno acontece nesse algoritmo?

Software de Tempo Real

- **Flags e Busy Waiting**
- Vejamos outro exemplo de exclusão mútua:

```
process P1;
  loop
     while flag 2= down do
        null
     end;
     flag1:= down
     <seção crítica>
     flag1:=up;
     <seção não-crítica>
     -
   end
end P1;
process P2;
  loop
     while flag 1= down do
        null
     end;
     flag2:= down
     <seção crítica>
     flag2:=up;
     <seção não-crítica>
     -
   end
end P2;
```

- O que acontece nesse algoritmo?

Software de Tempo Real

- **Flags e Busy Waiting**
- Vejamos outro exemplo de exclusão mútua:

```
process P1;
   loop
      while turn= 2 do
         null
      end;
      <seção crítica>
       turn:=2;
       <seção não-crítica>
       -
   end
 end P1;
process P2;
 loop
      while turn = 1 do
         null
      end;
      <seção crítica>
      turn:=1;
      <seção não-crítica>
      -
  end
end P2;
```

- Resolve o problema de exclusão mútua e livelock?

Software de Tempo Real

- **Flags e Busy Waiting**
- Vejamos outro exemplo de exclusão mútua proposto por Peterson, em 1985:

```
process P1;
   loop
      flag2:= up;
      turn:= 2;
      while flag2 = up turn = 2 do
         null
      end;
      <seção crítica>
      flag1:= down;
      <seção não-crítica>
       -
   end
end P1;
process P2;
   loop
      flag1:= up;
      turn:= 1;
      while flag1 = up turn = 1 do
         null
      end;
      <seção crítica>
      Flag2:=  down;
      <seção não-crítica>
       -
   end
end P2;
```

Software de Tempo Real

- Semáforo
- Utilizado para implementar exclusão mútua e condição de sincronização.
- Eles foram originalmente desenvolvidos por Dijkstra, em 1986, e tem as seguintes vantagens:
- Eles simplificam os protocolos para sincronização;
- Eles removem a necessidade de busy wait loops.

Software de Tempo Real

- Semáforo
- *Um semáforo é uma variável inteira não negativa que, excluindo a inicialização, pode ser operada por dois procedimentos*. Esses procedimentos são chamados de “P” e “V” por Dijkstra, mas são referenciadas por “*wait*” e “*signal*” por outros autores. As semânticas de *wait e signal* são as seguintes:
- *WAIT (S)* : *Se o valor do semáforo, S, é maior do que zero, então decrementa seu valor de um. Caso contrário, atrasa o processo até S ficar maior do que zero (e então decrementa seu valor).*
- *SIGNAL (S) : incrementa o valor do semáforo, S, de um.*

Software de Tempo Real

- Semáforo
- Uma propriedade importante de *wait e signal* é que suas ações são atômicas (indivisíveis). Dois processos executando operações *wait* no mesmo semáforo não podem interferir entre eles. Outra vantagem é que um processo não pode falhar durante a execução de uma operação semáforo.

Software de Tempo Real

- Semáforo
- Condição de sincronização utilizando semáforo:
- (* condição de sincronização*)
- var consyn: semáforo; (*inicializado com 0*)
- process P1; (*processo esperando*)
- -
- wait (consyn);
- -
- end P1;
- process P2; (*processo sinalizando*)
- -
- signal (consyn);
- -
- end P2;

Software de Tempo Real

- Semáforo
- Exclusão mútua utilizando semáforo:

```
(* exclusão mútua*)
var mutex: semáforo; (*inicializado com 1*)
process P1;
    loop
        wait (mutex);
        <seção crítica>
        signal (mutex);
        <seção não crítica>
    end
end P1;
process P2;
    loop
        wait (mutex);
        <seção crítica>
        signal (mutex);
        <seção não crítica>
    end
end P2;
```

Software de Tempo Real

- **Semáforo**:
- Na definição de *wait* está claro que se o semáforo é 0, então processo deve ser atrasado. Um método de atraso (busy wait) foi comentado anteriormente, mas apresentava problemas. Todas as primitivas de sincronização negociam com atrasos pela remoção do processo do conjunto de processos executáveis. Um novo estado de ***suspenso*** (algumas vezes chamado de bloqueado ou não executável) é necessário.
- Quando um processo executa um wait em um semáforo em 0, o RTSS(Run Time Support System) é invocado, e o processo é removido do processador, e colocado na fila de processos suspensos (que é a fila de processos suspensos em um semáforo particular). No exemplo abaixo, é apresentado como o processo é suspenso.

Software de Tempo Real

- **Semáforo**:
- WAIT (S) :
- if S > 0 then
- S:= S - 1
- else
- número suspenso:= número suspenso +1
- suspende processo chamador
- SIGNAL(S) :
- if número suspenso > 0 then
- número suspenso:= número suspenso -1
- coloca o processo suspenso como executável novamente
- else
- S:= S + 1

Software de Tempo Real

- **Deadlock:**
- O uso de semáforo pode causar o conceito de *deadlock*. Ou seja, um ou mais processos suspensos por tempo infinito ou indeterminado.
- O exemplo abaixo apresenta o conceito de *deadlock:*

P1	P2
wait (S1);	wait (S2);
wait (S2);	wait (S1);
-	-
-	-
signal (S2);	signal (S1);
signal(S1);	signal (S2);

Software de Tempo Real

- **Starvation:**
- Outro problema menos sério que ocorre com a utilização de semáforos é o conceito de ***starvation***. Esse problema é caracterizado por um processo que deseja ganhar acesso a um recurso, via seção crítica, e nunca é permitido acessar tal recurso, pois há outro processo que sempre ganha acesso antes dele.

Software de Tempo Real

- **Semáforos Binários:**
- Em todos os exemplos adotados até agora, somente os valores 0 e 1 foram utilizados para o semáforo. Uma forma simples de semáforo, chamado de *semáforo binário*, pode ser implementado para adotar esses valores.
- Um semáforo binário pode ser utilizado para prover sincronização a apenas um recurso. Como exemplo, 3 processos que concorrem a apenas um bloco de memória podem utilizar um semáforo binário.

Software de Tempo Real

- **Semáforo de Quantidade:**
- Qualquer outra variação da definição normal de um semáforo é chamado de semáforo de quantidade. Com essa estrutura o conjunto a ser decrementado por um wait ou incrementado por um signal não é fixado em 1, mas é dado como um parâmetro para as procedures:
- WAIT (S, i) :
- if S >= i then
- S:= S – i
- else
- delay
- S:= S – i
- SIGNAL (S,i) :
- S:= S + i

Software de Tempo Real

- **Semáforo de Quantidade:**
- O exemplo abaixo apresenta o uso de um semáforo de quantidade.
- procedure allocate (size : integer);
- begin
- wait(QS, size)
- end;
- procedure deallocate (size : integer);
- begin
- signal(QS, size)
- end;
- No exemplo acima, o semáforo deve ser inicializado com o número total de recursos no sistema.

Software de Tempo Real

- **Monitores:**

```
monitor buffer;
    export append, take;
    const size = 32;
     var BUF  : array[0…size-1] of integer;
          top, base : 0…size-1;
          spaceavailable, itemavailable  : condition;
          NumberInBuffer : integer;
     procedure append (I : integer);
     begin
        if NumberInBuffer = size then
           wait(spaceavailable);
        BUF[top] := I;
        NumberInBuffer := NumberInBuffer + 1;
        top := (top+1) mod size;
        signal(itemavailable);
     end append;
    procedure take (I : integer);
     begin
        if NumberInBuffer = 0 then
           wait(itemavailable);
         I:= BUF[base];
         base := (base+1) mod size;
        NumberInBuffer := NumberInBuffer - 1;
        signal(spaceavailable);
     end take;
begin (*inicialização*)
     NumberInBuffer := 0;
     top := 0;
     base := 0;
end;
```

Software de Tempo Real

- **Monitores**:
- Abaixo, é apresentado exemplo de uso de monitor na linguagem MODULA-1.

```
INTERFACE MODULE resource_control;
    DEFINE allocate, deallocate;
    VAR busy : BOOLEAN;
      Free : SIGNAL;
    PROCEDURE allocate;
    BEGIN
        IF busy THEN WAIT(free) END;
        busy := TRUE
    END;
    PROCEDURE deallocate;
    BEGIN
        busy := FALSE
        SEND(free)
    END;
BEGIN (*inicialização  do módulo*)
    busy := FALSE
END
```

Software de Tempo Real

- **Sincronização e Comunicação baseada em Mensagem**
- Uma alternativa para comunicação e sincronização de processos concorrentes é utilizando passagem de mensagem. A semântica para passagem de mensagem é baseada em três temas:

- o modelo de sincronização;
- o método de nomeação de processos;
- e estrutura da mensagem.

Software de Tempo Real

- **Sincronização de Processos**

  Variações no modelo de sincronização de processos são baseadas na semântica do processo (sender) origem ou remetente, que podem ser classificadas da seguinte maneira:

- ***Assíncrono:*** **o processo origem ou remetente prossegue imediatamente, desconsiderando se a mensagem foi recebida ou não;**
- ***Síncrono:*** **o processo origem ou remetente somente prossegue quando a mensagem for recebida;**
- ***Invocação remota:*** **o processo origem ou remetente somente prossegue quando um retorno (replay) for enviado pelo processo receptor ou destinatário. O paradigma requisição de resposta de comunicação é modelado pelo envio de invocação remota e é encontrada em ADA e em vários sistemas operacionais.**

Software de Tempo Real

- **Modelo síncrono conseguido de modelo assíncrono:**
- P1
- asyn_send(message)
- wait(acknowledgement)
- P2
- wait (message)
- asyn_send(acknowledgement)

Software de Tempo Real

- **Modelo de invocação remota conseguido com modelo síncrono:**

P1	P2
syn_send(message)	wait (message)
wait(replay)	constrói replay
	syn_send(replay)

Software de Tempo Real

- **Nomeação de Processos (Process Naming)**
- Nomeação de processos envolve dois sub-temas: ***direção versus indireção, e simetria***. No esquema de ***nomeação direta***, o emissor da mensagem explicitamente nomeia o receptor:
- send (message) to (process-name)
- Com esquema de ***nomeação indireta***, o emissor ou origem nomeia alguma entidade intermediária (conhecida por canal, mailbox, link ou pipe).
- send (message) to (mailbox)

Software de Tempo Real

- Um esquema de nomeação é simétrico se ambos, emissor e receptor, nomeiam cada um (direta ou indiretamente).

send (message) to (process-name)

wait (message) from (process-name)

send (message) to (mailbox)

wait (message) from (mailbox)

Software de Tempo Real

- O esquema é assimétrico se o receptor não nomeia nenhuma fonte específica, mas aceita mensagem de qualquer processo ou entidade intermediária. Veja exemplo abaixo:
- wait (message)

- Nomeação assimétrica encaixa-se no paradigma cliente-servidor, no qual o processo servidor produz serviço em resposta a mensagens de qualquer número de processos clientes. No entanto, uma implementação deve suportar uma fila de processos esperando pelo servidor.

Software de Tempo Real

- **Estrutura de Mensagem**
- Idealmente, uma linguagem deveria permitir qualquer objeto de dado de qualquer tipo definido para ser transmitido em uma mensagem. Obter esse ideal é muito difícil, particularmente se objetos de dados têm representações diferentes no emissor (origem) e no receptor (destino). E ainda mais se as representações incluem ponteiros. Por causa dessa dificuldade, algumas linguagens, como OCCAM, têm restrição no conteúdo de sua mensagem para dados não-estruturados, exige objetos de tamanho fixo e de tipos definidos pelo sistema.

Software de Tempo Real

- **Espera seletiva (Seletive Waiting)**
- É uma forma de passagem de mensagem no qual o processo destino deve esperar até que um processo específico ou canal libera uma mensagem.

Software de Tempo Real

- **Exemplo de espera seletiva em OCCAM:**
- *While true*
- *Alt*
- *ch1 ? I*
- *chout ! I*
- *ch2 ? I*
- *chout ! I*
- *ch3 ? I*
- *chout ! I*

Software de Tempo Real

- ***Exemplo de espera seletiva em C Paralelo:***
- *CHAN *CAN[3];*
- *int x, y;*
- *x = alt_wait(3, CAN[0], CAN[1], CAN[2]);*
- *chan_in_word(&y,CAN[x]);*

Software de Tempo Real

- **Ação Atômica**
- Em uma aplicação em que processos concorrem a recursos comuns, o conceito de ação atômica deve estar bem claro.
- Então, podemos conceituar uma ação atômica como:
- Uma ação é atômica se o processo desempenhando a ação não se preocupa com a existência de qualquer outro processo ativo, e nenhum outro processo ativo se preocupa com a atividade do processo durante o tempo que o processo está desempenhando a ação.

Software de Tempo Real

- **Ação Atômica**
- Outro conceito também pode ser apresentado para ação atômica. Uma ação é atômica se o processo desempenhando a ação não troca informações com outros processos enquanto a ação está sendo desempenhada. Ou, uma ação é atômica se o processo desempenhando a ação não detecta nenhuma alteração de estado, exceto aquele desempenhado pela própria ação, e se não revela sua alteração de estado até o término da ação.
- Um conceito simplificado para ação atômica pode ser apresentado como segue: uma ação é dita atômica se ela é vista pelo sistema como instantânea e indivisível.

Software de Tempo Real

- **Ações Atômicas de Duas Fases**
- Idealmente, todos os processos envolvidos em uma ação atômica deveriam obter os recursos requeridos, durante a ação. Esses recursos poderiam ser desalocados após a ação atômica terminar. Se essas regras forem seguidas, então não há necessidade para uma ação atômica interagir com qualquer ente externo, que é uma definição estrita para ação atômica. Infelizmente, esse ideal pode levar a uma utilização pobre de recursos.

Software de Tempo Real

- **Ações Atômicas de Duas Fases**
- Como primeiro passo, é necessário permitir uma ação atômica iniciar sem seu número total de recursos. Em algum ponto, um processo dentro da ação irá requerer uma alocação de recurso: a ação atômica deve então comunicar com o gerenciador de recurso. Para uma definição estrita de ação atômica, esse gerenciador de recursos deveria ter que participar da ação atômica, com o efeito de serializar todas as ações envolvendo o gerenciador de recursos. Claramente, isso é indesejável, e então é permitido a uma ação atômica comunicar externamente com gerenciadores de recurso.

Software de Tempo Real

- **Ações Atômicas de Duas Fases**
- Se recursos podem ser obtidos mais tarde e desalocados mais cedo poderia ser possível para uma mudança de estado externo para ser afetado por um recurso liberado e observado pela aquisição de um novo recurso. Isto poderia quebrar a definição de ação atômica. Logo, uma política segura para uso de recursos deve possuir duas fases distintas. Na primeira fase, chamada de *growing phase*, recursos são somente requeridos. Na segunda fase, chamada de *shrinking phase,* os recursos podem ser liberados (mas, nenhuma nova alocação pode ser produzida). Com tal estrutura, a integridade da ação atômica é garantida.

Software de Tempo Real

- Transações Atômicas
- Dentro da teoria de sistemas operacionais e banco de dados o termo transação atômica é geralmente usado. Uma transição atômica tem todas as propriedades de uma ação atômica, com uma característica adicional, sua execução acontece com sucesso ou falha.
- Se uma ação atômica falha então os componentes do sistema os quais estão sendo manipulados pela ação devem ser deixados em um estado inconsistente. Com uma transação atômica isto não pode acontecer, pois os componentes são retornados para o estado original. Transações atômicas são comumente chamadas de ações recuperáveis e, infelizmente, os termos ações atômicas e transações atômicas são usados com o mesmo significado.

Software de Tempo Real

- As duas propriedades principais de transações atômicas são:

- falha de atomicidade: significa que a transação deve ou completar com sucesso ou não ter nenhum efeito.
- sincronização de atomicidade: significa que a transação é indivisível no sentido que sua execução parcial não pode ser observada por qualquer transação executando concorrentemente.
- Obs: Transações atômicas são muito utilizadas para sistema de banco de dados.

Software de Tempo Real

- **Requerimentos para ações atômicas**
- Se uma linguagem de programação de tempo-real é capaz de suportar ações atômicas, então deve ser possível expressar os requerimentos necessários para sua implementação. Esses requerimentos são independentes da noção de processo, e da forma de comunicação inter-processo produzida pela linguagem. São eles:

- *Limites bem definidos*: cada ação atômica deveria ter um início e um fim, e um limite bem definido de cada lado. O limite de início é a localização em cada processo envolvido na ação atômica em que a ação é considerada a iniciar. O limite de fim é a localização em cada processo envolvido na ação atômica em que a ação é considerada a terminar. O limite de lado separa esses processos envolvidos na ação atômica do resto do sistema.

Software de Tempo Real

- **Requerimentos para ações atômicas**

- ***Indivisibilidade***: uma ação atômica não deve permitir uma troca de qualquer informação entre os processos ativos dentro da ação e aqueles fora da ação. Se duas ações atômicas compartilham dados, então o valor destes dados depois das ações é determinado por um seqüenciamento estrito das duas ações em alguma ordem.
- ***Aninhamento***: ações atômicas devem ser aninhadas desde que não sobreponham outras ações atômicas. Conseqüentemente, somente aninhamentos estritos são permitidos (duas estruturas são aninhadas estritamente se uma é completamente contida dentro da outra).

Software de Tempo Real

- **Requerimentos para ações atômicas**

- *Concorrência*: deveria ser possível executar diferentes ações atômicas concorrentemente. Uma maneira de forçar invisibilidade é rodar ações atômicas seqüencialmente. O efeito geral de executar uma coleção de ações atômicas concorrentemente deveria ser o mesmo que obtido da serialização de suas execuções.

- *Procedimentos de recuperação:* como a intenção de ações atômicas é a de formar a base para confinamento de danos, elas devem permitir que procedimentos de recuperação sejam programados.

Software de Tempo Real

- **Ações atômicas em linguagens concorrentes**

- Uso de semáforos para se conseguir ação atômica

Veja como podemos implementar uma ação atômica utilizando o conceito de exclusão mútua por semáforo:

```
wait(mutual_exclusion_semaphore)
   atomic_action
signal(mutual_exclusion_semaphore)
```

Software de Tempo Real

- **- Uso de monitor para se conseguir ação atômica**
- Atomic_action_begin1, atomic_action_begin2: semaphore :=1;
- Atomic_action_end1, atomic_action_end2: semaphore := 0;
- Procedure code_for_first_process is
- Begin
- - Start atomic action
- Wait(atomic_action_begin1)
- - get resource in non-sharable mode
- - update resource
- - signal second process that it is ok
- - for it to acess resource
- - any final processing
- Wait (atomic_action_end2);
- - return resource
- Signal(atomic_action_end1);
- Signal(atomic_action_begin1);
- End code_for_first_process;
- Procedure code_for_second_process is
- Begin
- - Start atomic action
- Wait(atomic_action_begin2)
- - initial processing
- - wait for first process to signal
- - that it is ok to acess resource
- - acess resource
- Signal(atomic_action_end2);
- Wait (atomic_action_end1);
- Signal(atomic_action_begin2);
- End code_for_second_process;

Software de Tempo Real

- **Controle de recursos**

  - A coordenação entre processos deve ser requerida se eles compartilham acesso a recursos escassos tais como dispositivos externos, arquivos, campos de dados compartilhados, buffers e algoritmos codificados. Esses processos são conhecidos como

  processos competidores.

  - Para que processos concorram a recursos escassos um controle ou uma gerência que coordene o acesso dos processos a esses recursos é de suma importância. Como recursos, podemos citar: dispositivos de hardware (timer, portas, interface serial), arquivos, blocos de memórias, campos de dados, entre outros.

Software de Tempo Real

- Controle de recurso e ações atômicas
- Um controle de recursos pode ser realizado por uma troca simples e confiável de mensagens, não necessitando de uma ação atômica para isso.

Software de Tempo Real

- **Gerência de recursos**

  - Uma das maneiras para se ter um controle de acesso ao recurso é empacotar ou encapsular o recurso. Com isso, consegue-se sincronizar o acesso dos processos ao recurso. O exemplo abaixo é utilizado pela linguagem ADA:

- *Package RESOURCE_MANAGER is*
- *type RESOURCE is private;*
- *function ALLOCATE return RESOURCE;*
- *function FREE (THIS_RESOURCE:RESOURCE)*
- *Private*
- *type RESOURCE is ...*
- *End RESOURCE_MANAGER;*

Software de Tempo Real

- **Potência expressiva e facilidade de uso**

  - Bloom (1979) usa o termo potência expressiva para representar a abilidade de uma linguagem de expressar restrições requeridas na sincronização. Facilidade de uso de uma primitiva de sincronização engloba:

  - a facilidade de expressão de cada restrição de sincronização;

  - a facilidade com o qual ele permite a restrição para ser combinada em ordem para conseguir esquemas de sincronização mais complexos.

Software de Tempo Real

Deadlock

- Situação comumente encontrada quando alguns processos concorrem a um número finito de recursos. Como exemplo, um processo P1 está usando o recurso R1, enquanto espera o acesso ao recurso R2. Se um processo P2, que está usando o recurso R2, também quiser acesso ao recurso R1, tem-se um caso de *deadlock*.

Uma das maneiras de se evitar o deadlock é não implementar o hold and wait.

- Exemplo:
- *P1* *P2*
- *Main()* *Main()*
- *allocate (R1)* *allocate (R2)*
- *allocate (R2)* *allocate (R1)*

Software de Tempo Real

- **Recursos de Tempo-Real**

  Abaixo, seguem alguns recursos importantes de tempo-real:

  - Acesso ao clock;
  - Retardo de processos;
  - Implementação de Timeout;
  - Especificação de prazo de encerramento (deadline) e escalonamento.

Software de Tempo Real

- **Acesso ao Clock**

  - Para aplicações de tempo-real é importante que a linguagem a ser utilizada interaja de alguma maneira com o tempo. Ou seja, tendo acesso de alguma maneira a informação do tempo ou, pelo menos, a medida do tempo que já passou.

  Isso pode ser feito de duas maneiras:

  - Implementando funções de acesso ao clock na linguagem;

  - Implementando um driver de dispositivo de clock associado aos processadores ou microcontroladores.

Software de Tempo Real

- **Exemplo 1 de acesso ao clock em OCCAM:**

```
TIMER clock:
INT Time:
SEQ
    clock ? Time
```

Software de Tempo Real

- Exemplo 2 de acesso ao clock em OCCAM:

```
TIMER clock:
INT old, new, interval:
SEQ
    clock ? Old
    other functions
    clock ? new
    interval := new MINUS old
```

- Exemplo 3 de acesso ao clock na linguagem C para o PIC:

```
Get_timer0();
Get_timer1();
Restart_wdt();
```

Software de Tempo Real

- **Atrasando um Processo**

  - Além do acesso ao clock do sistema, processos em aplicações de tempo-real devem também ser capazes de se atrasarem por um período de tempo. Esse procedimento pode ser realizado através do seguinte exemplo:

```
NOW:= CLOCK;
Loop
   exit when (clock-NOW) > 10.0;
End Loop;
```

Software de Tempo Real

- Muitas linguagens apresentam funções de atraso que eliminam os *busy waits*. Ada apresenta a seguinte função:

delay 10.0; espera dez segundos

- Em linguagem C para PIC, temos a seguinte função:

delay_ms(1000); espera um segundo. Essa função é implementada com busy wait loop, e há processamento na espera.

Software de Tempo Real

Implementando Timeout

- Um timeout é uma restrição no tempo de um processo que está esperando por um evento. A utilização de timeout é de suma importância para detecção em falhas na passagem de mensagens entre processos, na eliminação de *deadlocks*, entre outros.

Software de Tempo Real

- **Implementando Timeout em ADA:**

```
task CONTROL is
   entry CALL(V:VOLTAGE)
end CONTROL;
task body CONTROL is
begin
  loop
     select
        accept CALL(V:VOLTAGE) do
            NEW_VOLT:=V;
         end CALL;
      or
         delay 10.0;
      end select;
   end loop;
end CONTROL;
```

Software de Tempo Real

- **Exemplo de espera com timeout em C para PIC:**

```
boolean erro_timeout;
char getc_timeout();
{
    long int tempo;
    erro_timeout = false;
    while(!kbhit() && (++tempo<500) delay_ms(10);
    if (kbhit()) return (getc());
    else{
        erro_timeout = true;
        return (0);
    }
 }
```

Obs: A função KBHIT verifica se há algum dado sendo recebido pela USART (serial padrão).

Software de Tempo Real

- **Exemplo de timeout em OCCAM:**

```
WHILE TRUE
  SEQ
    ALT
      call ? new_voltage
        -- outras ações
      clock ? time
        -- ação para timeout
```

Software de Tempo Real

- **Deadline (Prazo de Encerramento)**

  - Antes, temos que falar primeiro sobre alguns conceitos importantes, como tipos de processos em relação ao tempo. Existem dois tipos de processos em relação ao tempo:
    - Processos periódicos;
    - Processos aperiódicos ou esporádicos.

  Obs: ambos processos devem ser analisados para dar seus tempos de execução de pior caso.

  Para facilitar a especificação dos vários conceitos em aplicações de tempo-real, temos que explicar o que é TS (Temporal Scope – Escopo Temporal). Tais escopos identificam um conjunto de estruturas com uma restrição de tempo associada.

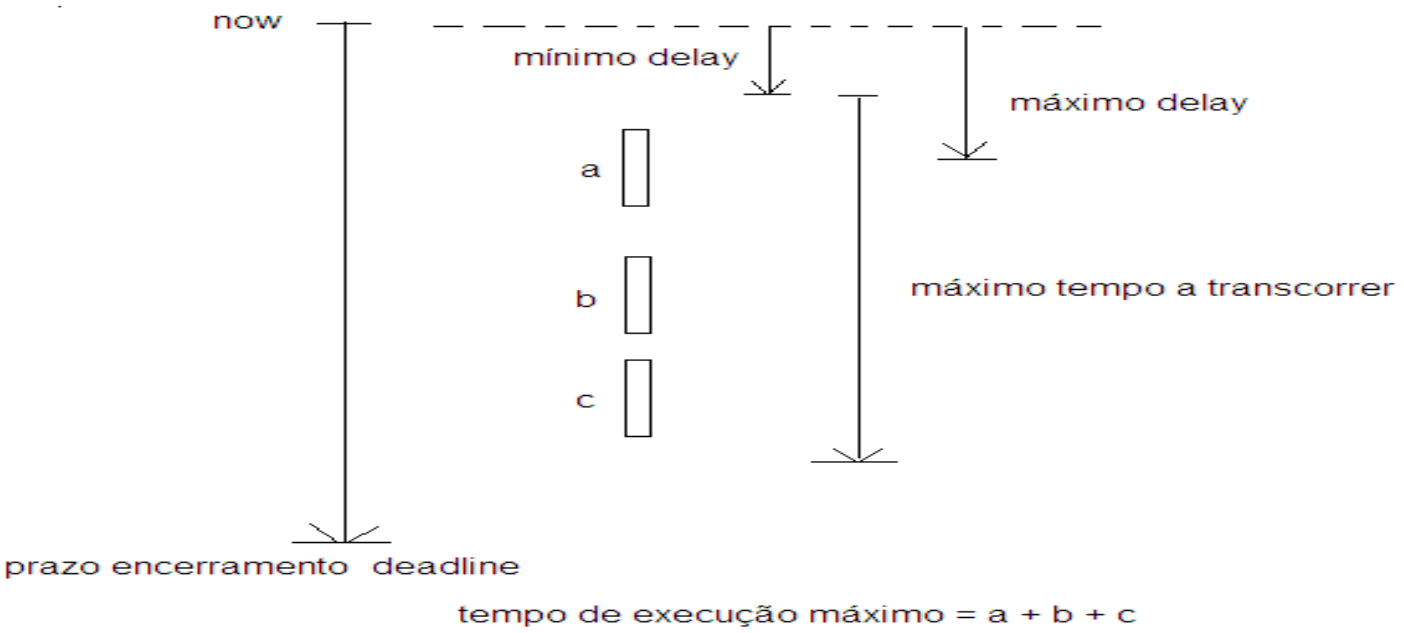
Software de Tempo Real

- Atributos possíveis de um TS:

- Deadline: tempo no qual a execução do TS deve ser finalizada;
- Mínimo delay: o mínimo tempo que deve transcorrer antes do início da execução do TS;
- Máximo delay: o máximo tempo que deve transcorrer antes do início da execução do TS;
- Máximo tempo de execução: do TS;
- Máximo tempo a transcorrer: do TS.

Software de Tempo Real

- Na figura abaixo, é apresentado um gráfico com todos os atributos de um escopo temporal.



Software de Tempo Real

- **Escalonamento para tempo-real**

  Escalonamento de processos é uma atividade no qual um programa do kernel do SO seleciona um processo para ser executado pela CPU. O programa responsável em fazer o escalonamento é o escalonador.

  Para um RTOS, o escalonamento deve apresentar quesitos de tempo-real. Deve ser pre-emptivo e possuir políticas de escalonamento que levem em conta os prazos de encerramento (deadlines) dos processos.

Software de Tempo Real

- **Escalonamentos de tempo-real**
- Rate Monotonic: Usado para escalonamento de processos periódicos. Designa prioridade fixa para cada processo com base no seu período. Ou seja, o processo com menor período terá maior prioridade.
- Earliest Deadline: Nesse algoritmo, o escalonador deve ter informações explícitas dos deadlines dos processos. Assim, o processo com deadline mais próximo, terá maior prioridade.
- Least Slack Time: Nesse algoritmo, o escalonador precisa saber não somente todos os deadlines dos processos, mas também o tempo de execução requerido para cada processo antes do seu deadline. Ou seja, o processo com menos tempo livre terá maior prioridade.