O QUE SÃO SISTEMAS DE TEMPO REAL?

Sistemas de tempo real são sistemas de computação que monitoram, respondem ou controlam um ambiente externo. Esse ambiente está conectado ao sistema de computação através de sensores, atuadores e outras interfaces de entrada-saída. Podem constituir-se de objetos físicos ou biológicos de qualquer forma e estrutura. Freqüentemente, seres humanos fazem parte do mundo externo conectado, mas um amplo domínio de outros objetos naturais e artificiais, assim como animais, são também possíveis.

O sistema de computação deve satisfazer a várias restrições, temporais e outras, impostas a ele pelo comportamento de tempo real do mundo externo com o qual faz interface. Daí vem o nome tempo real. Um outro nome para muitos desses sistemas é sistemas reativos, por que seu propósito primordial é responder ou reagir a sinais provenientes de seu ambiente. Um sistema de computação de tempo real também pode ser um componente de um sistema maior no qual está embutido; com razão, tal computador-componente é chamado de sistema embarcado.

Aplicações e exemplos de sistemas de tempo real são onipresentes e estão proliferando, aparecendo como parte de nossas infra-estruturas comerciais, governa mentais, militares, médicas, educacionais e culturais. Nesses, estão incluídos:

• 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;

1. Definição de Sistemas de Tempo-Real

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.*

Segundo o Dicionário Oxford de Computação.

sistema em tempo real é qualquer sistema no qual o tempo cuja saída é produzida é significante.

Young (1982) define sistema de tempo-real como

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.

- 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.

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.

- Exemplos de Sistemas de Tempo-Real

- Sistemas de controle de eletrodomésticos e eletro-eletrônicos.
- Sistema de controle de tráfego aéreo.
- Sistema de controle de veículos;
- Controle de processos para usinas de energia elétrica, nuclear, indústrias químicas, entre outras.

- 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.

- 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, nem sempre é possível elaborar códigos pequenos. Mas, uma

linguagem que se proponha ao desenvolvimento de um sistema de tempo-real deve oferecer facilidades para minimizar o tamanho e variedade de uma aplicação. Como por exemplo, oferecer facilidades para modularizar o código, oferecer número reduzido de instruções, apresentar funções eficientes, entre outras.

Outra característica importante é a complexidade de um código. Exemplo para minimizar a complexidade de uma aplicação pode ser utilizando linguagens que apresentem um conjunto de funções pequenas e bastante eficientes.

- Operações com Números Reais

Como visto anteriormente, 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 variando entre 4 a 2OmA ou O a 10V, respectivamente. Como exemplo, podemos citar um sensor de efeito hall, cuja saída pode variar entre O e 5V, para alguns modelos. Essa variação na saída está relacionada com a intensidade de campo magnético que atravessa a face do sensor. Assim sendo, sistemas de tempo-real devem ser capazes de operar e manipular com números reais, uma característica intrínseca de eventos do mundo real.

Outro exemplo é uma planta de um processo industrial. Como mostrado na Figura 1, um sinal de referência r(t) é aplicado ao sistema. Esse sinal de referência é comparado com o sinal de saída da planta y(t) para gerar um sinal de erro e(t). Esse sinal é, em geral, um número real. Posteriormente, esse sinal é aplicado ao controlador, que o processará, possivelmente através de funções matemáticas para gerar o sinal atuante sobre a planta.

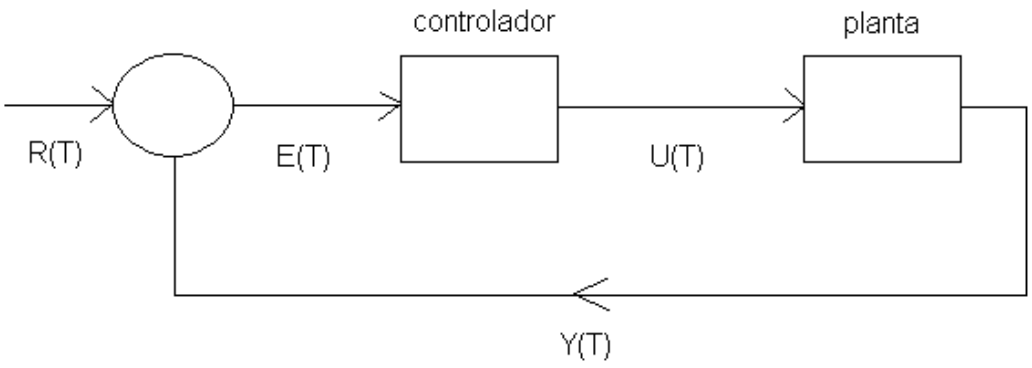


Figura 1. Um sistema de controle simples.

- 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 prejuízos financeiros. Assim, esses

sistemas podem ser considerados como sistemas críticos. Aplicações desenvolvidas para esses sistemas, devem oferecer confiabilidade e segurança.

Confiabilidade é a probabilidade de operação livre de falhas durante um tempo espec em um dado ambiente, para um propósito específico.

Assim sendo, podemos fazer uma pergunta: um sistema A apresenta uma falha em um ano de operação, enquanto um sistema B apresenta três falhas no mesmo período. Qual sistema é mais confiável? A resposta correta é sistema A.

Outra análise que pode ser feita é sobre o conceito de sistema de tempo-real. Como um sistema poderá oferecer uma saída em resposta a um estímulo de uma entrada se ele falhou. Assim, confiabilidade torna-se uma característica essencial de sistema de tempo-real.

Muitas vezes, sistemas de tempo-real controlam eventos do mundo real com estreita interação com pessoas e meio ambiente. Como exemplo, um sistema de controle de uma planta industrial, que com uma falha pode causar um desastre ambiental ou risco a vidas humanas. Outro exemplo, seria o sistema de controle de um avião, cuja falha poderia ocasionar a queda da aeronave. Para esses sistemas, devem-se implementar técnicas ou mecanismos para oferecer segurança em sua operação. Logo, *segurança reflete a capacidade do sistema operar de forma normal e anormalmente, sem oferecer ameaças as pessoas ou ao ambiente.*

- Controle Concorrente de Componentes do Sistema Separados

Sistemas de tempo-real lidam com eventos do mundo real, muitos deles ocorrendo simultaneamente. Como exemplo, podemos ter um sistema de controle que monitore a chegada de uma peça, role a esteira, pare para inserção de um componente, e volte a rolar a esteira novamente. Nesse caso, a chegada de uma peça pode ocorrer no mesmo momento da inserção de um componente. *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 una linguagem que apresentasse funções para criação de threads, permitiria uma concorrência em nível de hardware e software. Como exemplo, podemos citar alguns SOs e linguagens de 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.

- 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.

Essas facilidades são conhecidas como facilidades de tempo-real. Elas permitem que o programa se sincronize com o tempo. Como exemplo, o sistema pode apresentar facilidades de acesso ao clock, a data e calendário, funções de atraso, entre outras facilidades.

- 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.

- 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.

A eficiência pode ser também ser obtida na programação. Utilizar funções para codificar escopos repetitivos, usar escopos enxutos e eficientes, utilizar escopos em assembly quando necessário são exemplos de uma codificação eficiente.

2. 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.*

Processo é o objeto ativo de um sistema e é a unidade lógica de trabalho escalonada pelo sistema operacional. Ele tem um estado, normalmente representado por um descritor de dados ou bloco de controle de processos (BCP). O descritor armazena informações, como os valores das variáveis do processo, seu contador de instruções e os recursos que foram alocados para ele, por exemplo, memória principal ou dispositivos de ES.

- 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. 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.

- 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. Em geral, eles são usados para monitoramento sistemático, polling ou amostragem de informação a partir de sensores por um intervalo longo de tempo. Por exemplo, pode-se empregar um processo periódico para ler a temperatura de um líquido a cada 50 milissegundos, ou para varrer um espaço aéreo a cada 3 segundos.

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.

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 éo 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. O tempo de computação é obtido através de simulação, medição ou análise.

Um processo esporádico P poderia ser representado pelo código

P:: loop aguarde_por evento; P_código; end loop.

P é bloqueado até que o evento ocorra, tornando-se, então, habilitado. Uma vez que a computação de P_código seja concluída, ele é bloqueado novamente, esperando pela próxima instância do evento.

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***. Note que os requisitos para um processo esporádico poderiam ser atingidos por um processo periódico, desde que esse procure ou teste pelo evento com suficiente freqüência.

As definições padrões acima para processos periódicos e esporádicos não incluem os efeitos de interações entre processos. Se dois processos compartilham recursos, por exemplo, usando seções críticas protegidas por chaveamentos comuns, eles podem ser bloqueados, um pelo outro, várias vezes durante sua execução. Tempos de bloqueio, os períodos em que os processos aguardam por recursos ou mensagens requeridos, precisam ser incluídos em um modelo mais realista.

- 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.

Processos independentes não se comunicam ou sincronizam entre si. Processos cooperantes, por comparação, regularmente comunicam e sincronizam suas atividades para desempenhar alguma operação comum. Como exemplo, um sistema embarcado que necessita monitorar sensores de temperatura, pressão e nível simultaneamente, precisa de processos que interajam entre si para atingir essa operação.

Um sistema de computador tem um número finito de recursos os quais devem ser compartilhados entre processos; como exemplo, dispositivos periféricos, memória e processador. Assim, processos devem competir entre si para conseguirem esses recursos. Esses processos são chamados de competidores. A ação de alocação de recurso inevitavelmente requer comunicação e sincronização entre os processos do sistema.

- Representacão de Processos

Em termos de representação, existem três formas básicas para expressão de execução concorrente: fork e join; cobegin e declaração de processo explícito.

- Fork e Join:

Esta simples aproximação não produz uma entidade visível para um processo, mas meramente suporta duas rotinas. 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.

- 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
```

Este código causa as estruturas S1, S2, S3 e as outras para serem executadas concorrentemente. A estrutura cobegin termina quando todas as estruturas concorrentes terminam. Cada estrutura Si deve ter qualquer construção permitida dentro da linguagem, incluindo designações simples ou chamadas de procedimento. A estrutura cobegin pode ser encontrada em OCCAM2.

- Declaração de Processo Explícito:

Neste tipo de notação de programação concorrente, os processos são declarados de maneira mais clara. O exemplo abaixo foi desenvolvido em linguagem Modula-1. No exemplo, o processo control é declarado com um parâmetro para ser passado na criação.

```
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.
```

3. Confiabilidade e Tolerância a Faltas

Confiabilidade e requisitos de segurança são muito mais importantes para sistemas de tempo-real do que para outros sistemas de computadores. 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.

Cada vez mais, a sociedade está entregando o controle de suas vidas a sistemas de computadores. Logo, torna-se essencial que estes sistemas não falhem.

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.

- Falta, Erro e Falha

Para o conceito de confiabilidade, é interessante diferenciar os termos 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. Como exemplo, imagine um circuito lógico que em determinado instante deve apresentar em sua saída um nível lógico alto. Se a saída apresentada nesse instante for nível lógico baixo, então, no sistema, é caracterizado um erro.

Uma falha é caracterizada por um desvio de comportamento do sistema no qual ele foi especificado para isso. Como exemplo, se um sistema que deveria parar uma esteira na detecção de um produto defeituoso não o faz, isso é visto como uma falha do sistema. Logo, podemos dizer que uma falta de um componente pode causar um erro (estado errôneo) no sistema; e esse erro ou estado errôneo pode causar uma falha do sistema.

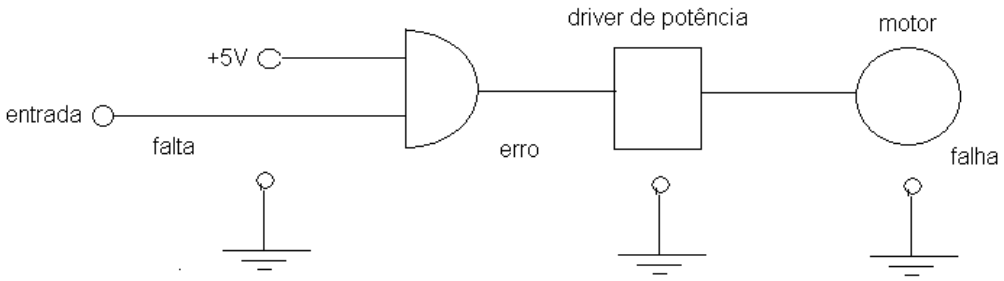


Figura 2. Diagrama para exemplificar falta, erro e falha.

Na Figura 2, podemos exemplificar os conceitos: falta, erro e falha. Como exemplo, uma solda fria no pino de entrada da porta AND caracteriza uma falta. Devido a essa falta, em um determinado instante, a saída da porta AND não apresentará nível lógico alto, como esperado. Isso é caracterizado como um erro. Devido a esse erro, o motor não será acionado no tempo esperado. Isso é caracterizado com falha.

Três tipos de faltas podem ser distinguidas:

- ***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.

- ***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.

Para criar um sistema confiável todos esses tipos de faltas devem ser prevenidos, evitando assim o surgimento de comportamento errôneo.

- Prevenção de Faltas e Tolerância a Faltas

Duas técnicas podem ajudar projetistas a tornarem seus sistemas confiáveis. A primeira é conhecida como *prevenção de faltas.* Essa técnica permite eliminar qualquer possibilidade de faltas no sistema antes de se tornar operacional. O segundo é *tolerância a faltas.* É a técnica que possibilita o sistema continuar funcionando mesmo na presença de faltas.

- Prevenção de faltas:

Existem dois estágios de prevenção de faltas: ***fault avoidance*** (evitar faltas) e ***fault removal*** (remoção de 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.

Com relação ao software, faltas podem ser evitadas da seguinte maneira:
- Especificação de requerimentos 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 manipular componentes de software e também gerenciar a complexidade;

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. O segundo estágio de prevenção de faltas é a técnica de *remoção de faltas.* Técnicas de remoção de faltas podem nunca remover todas as potencias faltas. Nessa técnica, 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.

Mesmo com o uso de técnicas de prevenção de faltas, faltas podem existir e levar o sistema a falhar. Logo, técnicas de tolerância a faltas podem ser utilizadas para permitir que o sistema funcione mesmo com a presença de 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.

O nível de tolerância a faltas a ser utilizada depende da aplicação. Em teoria, a maioria dos sistemas críticos em segurança requer o nível de falha operacional. Mas, quando não se pode implementar falha operacional, níveis de falha suave podem ser implementadas para suprir a necessidade de falha operacional.

Em algumas situações, é apenas necessário desligar o sistema em um estado seguro. Esse sistema de falha segura permite limitar um conjunto de danos causado por uma falha. Como exemplo, um erro no acionamento hidráulico dos flaps da asa de um avião pode fazer com que o sistema de controle coloque os flaps em alinhamento horizontal (estado seguro) e desligue.

Abordagens para o projeto de sistemas tolerantes a faltas produzem três suposições:

1- Os algoritmos do sistema devem ser corretamente projetados;
2- Todos os modos possíveis de falha dos componentes devem ser conhecidos;
3- Todas as interações entre o sistema e o ambiente devem ser previstos.

- 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. Isso é normalmente conhecido como *redundância protetiva*. A meta de tolerância a faltas é minimizar a redundância enquanto maximiza a confiabilidade produzida, sujeito a restrição de custo e tamanho em um sistema. Cuidado deve ser tomado na estrutura de sistemas tolerantes a faltas, pois a inserção de componentes redundantes aumenta a complexidade do todo o sistema. Essa complexidade pode levar o sistema a ficar menos confiável.

A redundância pode ser classificada como: *redundância estática e redundância dinâmica*. Na redundância estática, os componentes redundantes são usados dentro do sistema para esconder os efeitos das faltas. Um exemplo de redundância estática é a *redundância modular tripla (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. A suposição aqui é que a falta não é devido a um aspecto comum dos subcomponentes, mas devido a um falta transiente ou deterioração de algum componente. Claramente, para mascarar faltas de mais de um componente é requerida mais redundância. O termo geral *redundância modular N (NMR)* é utilizado para essa aproximação.

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. Exemplos de redundância dinâmica são checksums em comunicações e bits de paridades em memória.

- Programação N-Versão:

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*. A produção independente de N programas significa que N escopos de código produzem N versões requeridas do software sem interação. *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.

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. Para uma proteção a faltas físicas, as N versões podem estar em computadores diferentes.

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.

Observa-se que deve haver uma comunicação e sincronização entre o driver e as versões para que o driver realize\a comparação dos votos. Essa interação é especificada nos requerimentos para as versões. Consiste em três componentes:

- Vetores de comparação;
- Indicadores de status de comparação;
- Pontos de comparação.

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. Na Figura 3, é apresentado um diagrama de blocos de uma aplicação de programação N versão.

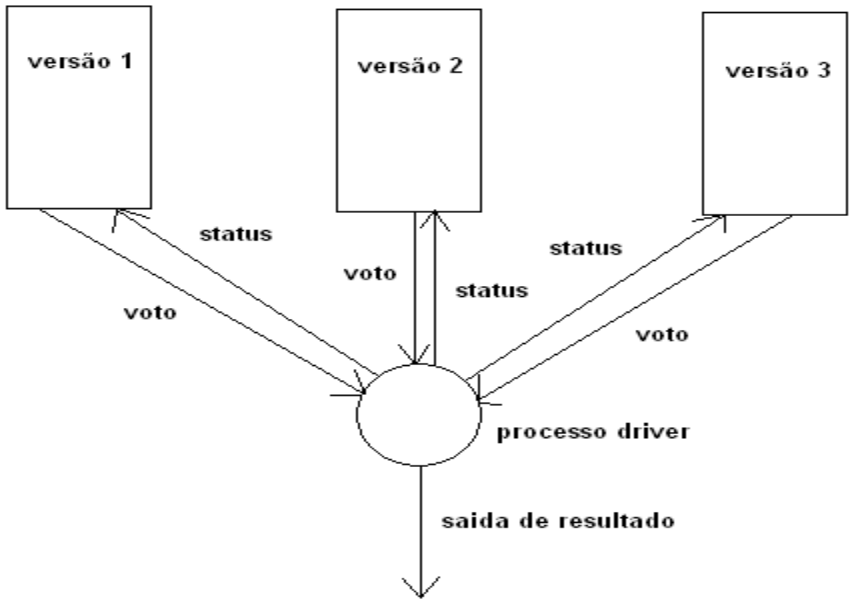


Figura 3. Diagrama de blocos para exemplificar programação N-versão.

- Redundância Dinâmica de Software

Programação N-versão é um equivalente de software da redundância estática, em que faltas dentro de um componente são escondidas do meio externo. É dita estática porque cada versão de software tem um relação fixa com as outras versões, bem como com o driver. Por esse razão, o sistema irá operar se faltas ocorrerem ou não. 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:*** muitas faltas irão eventualmente se manifestar em forma de um erro; nenhum esquema de tolerância a faltas pode ser utilizado até um erro seja detectado.

- ***avaliação e confinamento de danos:*** quando um erro for detectado, deve ser avaliado qual a extensão do dano no sistema; deve, se possível, confinar esse dano para que não se espalhe pelo sistema. Uma demora na ocorrência da falta e na manifestação do erro associado a ela pode espalhar o dano por todo o sistema.

- ***recuperação de erro:*** este é um dos mais importantes aspectos de tolerância a faltas. Técnicas de recuperação de erro têm com objetivo transformar um sistema corrompido em um estado no qual o sistema pode continuar sua operação normal.

- ***tratamento de falta e serviço continuado:*** um erro é um sintoma de uma falta; embora o dano tenha sido reparado, a falta ainda existe, e pode voltar a gerar novos erros. Logo, técnicas de tratamento ou remoção de faltas devem ser utilizadas para que o serviço seja continuado.

A seguir, será apresentada uma descrição de todas as fases utilizadas na redundância dinâmica de software.

- Detecção de Erro

A eficiência de um sistema tolerante a faltas depende da eficiência na técnica de detecção de erros. 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. Muitos testes podem ser realizados na aplicação para detecção de erros:

Teste de replicação: como na programação N–versão, versões replicadas e um driver podem ser utilizados para detecção de erro; *teste de temporização (tempo):* técnicas de watchdog timer, bem como temporização com timeout em comunicação, podem ser utilizadas para detectar erros; *teste reverso:* é utilizado em sistema que tem uma relação de um para um entre entrada e saída. Tal teste, pega a saída, faz um cálculo inverso para obter a entrada, e compara com a entrada corrente; *teste de codificação:* este teste é usado para verificar dados corrompidos. Ele é baseado em informação redundante contido no próprio dado. Um exemplo é o teste chamado *checksum; teste de razoabilidade:* é baseado no conhecimento do projeto interno e construção do sistema. É testada a razoabilidade de estado de valor ou dado de um objeto. Se for muito diferente do esperado, um erro é gerado; *teste estrutural:* é usado para testar a integridade de objetos de dados tais como

lista e filas. Consiste na contagem do número de elementos no objeto, ponteiros redundantes ou informações de status extras.

- 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. Um erro que foi detectado irá dar à rotina de manipulação de erro alguma idéia do dano. 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; uma preocupação que desenvolvedores de sistemas tolerantes a faltas devem ter. Confinamento de dano está relacionado com a estrutura do sistema, e é utilizado para minimizar os danos causados por elementos faltosos.

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. Nesse caso, a interface entre esses módulos é bem definida, e os detalhes internos de cada módulo são escondidos do mundo externo. Assim, há uma dificuldade de um erro em um módulo ser passado para outro módulo.

Outra técnica que pode ser usada para confinamento de danos é a utilização de *ações atômicas*. *A atividade de um componente é dita ser atômica se não existe interação entre a atividade e o sistema na duração da 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.

- Recuperação de Erro

Uma vez que uma situação de erro tem sido detectada e o dano avaliado, então procedimentos de recuperação de erro devem ser iniciados. Esta é a mais importante fase de qualquer técnica de tolerância a faltas. *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*.

Recuperação de erro para frente permite continuar de um estado errôneo pela correção seletiva para o estado do sistema. Para sistemas de tempo real, essa técnica deve envolver produção segura de qualquer aspecto do ambiente controlado, o qual deve ser danificado pela falta. Embora recuperação de erro para frente seja eficiente, ela é específica do sistema e depende da predição precisa da localização e causa de erros. Exemplos de recuperação de erro para frente são ponteiros redundantes em estruturas de dados e o uso de códigos de auto-correção, tais como Hamming Codes.

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*. Para estabelecer o

ponto de recuperação é necessário salvar a informação do estado do sistema apropriado em tempo de execução.

Restabelecimento de um estado tem como vantagem a eliminação de um estado errôneo. Recuperação de erro para trás pode, no entanto, ser usada para recuperar de faltas não antecipadas, incluindo erros de projeto. Como sempre, sua desvantagem é não poder desfazer qualquer efeito que a falta tenha causado no ambiente do sistema de tempo real. Como exemplo, é impossível desfazer um lançamento de um míssil.

Uma observação importante é a utilização de recuperação de erro para trás em programação concorrente. Em programação concorrente, processos realizam IPC (inter process communication – comunicação inter-processo). Logo, estabelecer pontos de recuperação em processos que se comunicam entre si é muito complicado. Veja exemplo na figura 4.

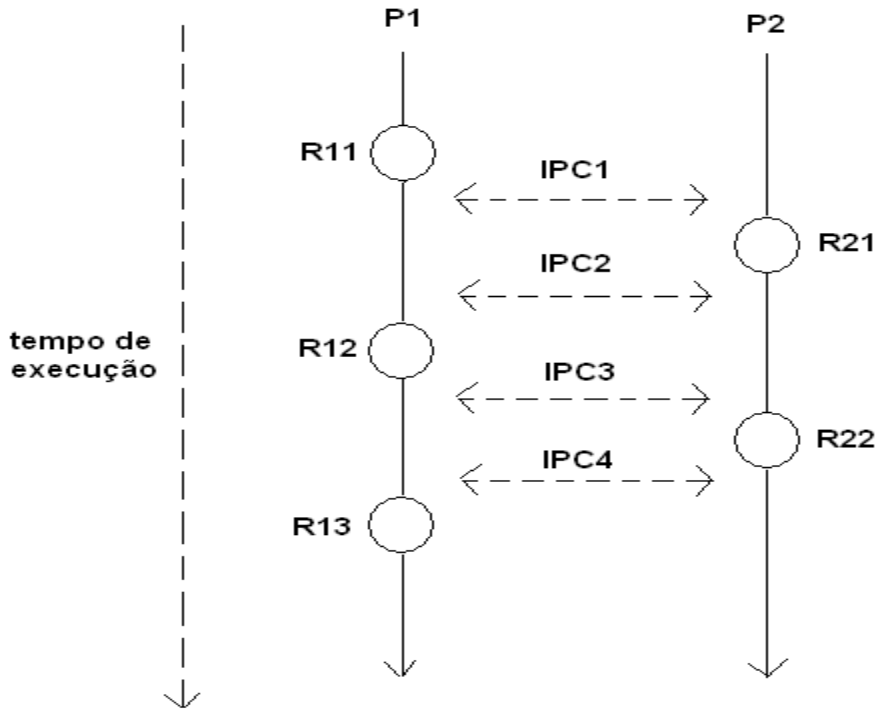


Figura 4. Exemplo de recuperação para trás em programação concorrente.

Na figura 4, um exemplo de comunicação entre dois processos é apresentado. Se um erro ocorrer no processo P1, ele tem que voltar para o ponto de recuperação R11. Observa-se que logo depois ele tem que fazer uma comunicação com o processo P2. Mas, o processo P2 está em execução em um ponto muito adiante. Se a comunicação é sincronizada, o processo P1 ficará travado nesse ponto. Esse é um grande problema nesse tipo de aplicação.

- Tratamento de Falta e Serviço Continuado

Um erro é uma manifestação de uma falta, e embora a fase de recuperação de erro tenha retornado o sistema a um estado livre de erro, a falta deve ocorrer. No entanto, a fase final de tolerância a faltas é erradicar a falta do sistema e então permite a continuação do serviço normal.

O tratamento automático de faltas é difícil de ser implementado, e tende a ser específico de cada sistema. Conseqüentemente, alguns sistemas não produzem provisão para tratamento de faltas, assumindo que todas as faltas são transientes. Outros assumem que técnicas de recuperação de erro são suficientes para evitar todas as faltas recorrentes.

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. Em muitas aplicações ininterruptas é necessário modificar o programa enquanto ele está executando, o qual representa um problema técnico significante.

- Blocos de Recuperação

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. A falha no teste de aceitação resulta no programa ser restaurado a um ponto de recuperação no início do bloco, e então um módulo alternativo é executado. Se o módulo alternativo falha no teste de aceitação, então o programa é restaurado ao ponto de recuperação, e, assim, um outro módulo é executado. Teste de aceitação produz mecanismo de detecção de erro, o qual então habilita o sistema para explorar redundância. O projeto do teste de aceitação é crucial para eficácia no esquema de bloco de recuperação. Todas as técnicas de detecção de erro podem ser usadas para compor teste de aceitação. Cuidado deve ser tomado no projeto de teste de aceitação, pois um teste de aceitação errado pode deixar erros residuais não detectados no sistema. Na Figura 5, é apresentado um diagrama de blocos representado blocos de recuperação.

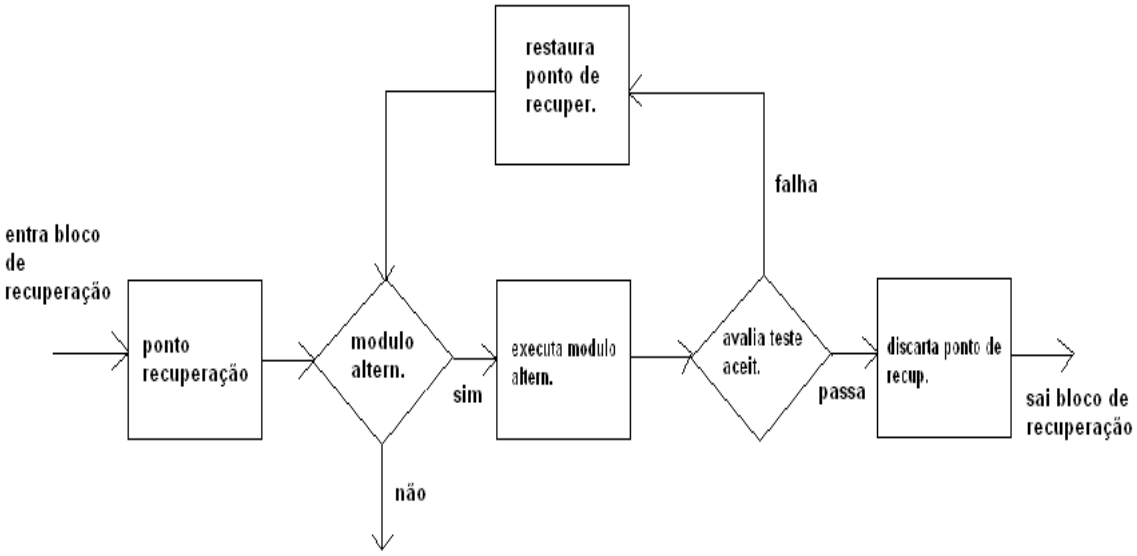


Figura 5. Diagrama de blocos representado blocos de recuperação.

Em termos das quatro fases de tolerância a faltas de software: detecção de erro é conseguida pelo teste de aceitação; avaliação de danos não é necessária em recuperação de erros para trás, pois é assumido que o estado errôneo é eliminado; e tratamento de falta é conseguido pelo uso de um componente reserva. 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
```

Igualmente a blocos ordinários, blocos de recuperação também podem ser aninhados.

- 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).* Manipulação de exceção pode ser usada para técnicas de recuperação de erros para trás e para frente.

Uma das formas mais primitivas de mecanismo de manipulação de exceção é o *valor de retorno não usual ou retorno de erro* de um procedimento ou função. A principal vantagem é sua simplicidade e o não requerimento de qualquer mecanismo da linguagem. 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;
```

4. Sincronização e Comunicação baseada em Memória Compartilhada

A maior dificuldade associada à programação concorrente deve-se à interação de processos concorrentes. Na programação concorrente, a sincronização e comunicação entre processos são características intrínsecas. Os dois conceitos estão intimamente associados, pois uma comunicação entre processo requer alguma forma de sincronização, assim como, a sincronização entre eles pode requerer alguma forma de comunicação.

Comunicação de dados é geralmente baseada em alguma forma de memória compartilhada ou passagem de mensagem. Variáveis compartilhadas são objetos que vários processos devem acessar. Comunicação pode ser obtida, para cada processo, referenciando essa variável quando necessário.

Passagem de mensagem envolve a troca explícita de dados entre dois processos. Essa ação é representada por uma mensagem que passa de um processo para outro por um algum agente de comunicação, tal como um canal de comunicação.

4.1. Exclusão Mútua e Condição de Sincronização

Embora variáveis compartilhadas sejam uma boa facilidade para passagem de informações entre processos, elas apresentam problemas sérios quando há atualizações múltiplas de seus valores. Como a atualização de seu valor não é uma ação única e indivisível, a atualização múltipla pode apresentar leituras erradas de seus valores. Como exemplo, suponha dois processos que queiram executar a seguinte atribuição: X = X+1. Esse procedimento poderá ocorrer em 3 passos:

- carrega o valor da variável em algum registrador;

- incrementa o valor desse registrador,
- armazena o valor do registrador de volta a memória (variável).

Observa-se que depois do valor do registrador ser incrementado pelo primeiro processo, um segundo processo pode carregar o valor de X novamente no registrador, causando um armazenamento de um valor errado na variável X, em memória.

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.*

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*.

Um exemplo típico de condição de sincronização é quando dois processos comunicam-se indiretamente, utilizando, como exemplo, um buffer. O uso de um buffer para comunicar dois processos em programação concorrente é conhecido como sistema consumidor-produtor.

Duas condições de sincronização são necessárias se um buffer finito é utilizado. Primeiro, o processo produtor não deve depositar dados no buffer se o buffer estiver cheio. Segundo, o processo consumidor não pode extrair objetos do buffer se ele estiver vazio. Logo, se depósitos e extrações simultâneas são possíveis, então exclusão mútua deve ser garantida para que dois produtores, por exemplo, não corrompam o próximo slot livre do buffer.

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*.

4.2. Busy Waiting

Um caminho para implementar sincronização é ter processos alterando e testando variáveis compartilhadas que estão atuando como flags. Essa aproximação trabalha bem para implementação de condição de sincronização, não é um método simples quando há exclusão mútua. 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;
```

Se a condição ainda não está setada (colocada em um), P1 não tem escolha, mas fica testando a flag através do loop. *Isso é busy waiting*.

Quando há exclusão mútua é necessário um algoritmo mais complexo. Vamos exemplificar através de dois processos que têm mutuamente seções críticas. Para proteger o acesso dessas seções críticas é assumido que cada processo executa um protocolo de entrada antes da seção crítica e executa um protocolo de saída depois da seção crítica. 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;
```

Ambos processos anunciam suas intenções para entrar em suas seções críticas, e então testam para ver se o outro processo está em sua seção crítica. Essa solução apresenta um sério problema, ambos processos podem permanecer em seus loops de busy wait. O resultado é que nenhum deles pode sair mais do loop. *Esse fenômeno é conhecido como*

livelock, e representa uma condição de erro sério. Para resolver esse problema, podemos inverter a estrutura, testando a flag antes de sinalizá-la. Veja exemplo abaixo:

```
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;
```

Nesse caso, evita-se que os processos fiquem presos em seus loops de busy wait, evitando assim o *livelock*. Mas, gera-se outro problema crítico, os processos podem simultaneamente entrar na região crítica, evitando assim a exclusão mútua. Assim, observa-se que a aproximação correta é usar somente uma flag que indica qual processo deverá ser o próximo a entrar na seção crítica. O exemplo abaixo, apresenta uma solução que resolve isso.

```
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;
```

Com essa estrutura, resolve-se o problema da exclusão mútua e do livelock se ambos os processos têm execução cíclica. Infelizmente, se o processo P1 falha em sua seção não-crítica, então turn irá eventualmente obter o valor 1, e irá permanecer com esse valor. Sempre quando a execução normalmente usa uma variável simples turn requer que processos tenham uma execução cíclica com mesma taxa. Não é possível para P1 entrar em sua seção crítica três vezes entre visitas por P2. Essa restrição é inaceitável para processos autônomos.

Finalmente, é apresentado um algoritmo que produz exclusão mútua e ausência de livelock. O algoritmo foi apresentado inicialmente por Peterson, em 1985. A aproximação de Peterson utiliza duas flags (flag1 e flag2) que são manipuladas pelo processo que quer ser prioritário, e uma variável turn que é sempre usada quando existe contenção para entrada de uma seção crítica.

```
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
      flag2:= up;
      turn:= 1;
      while flag1 = up turn = 1 do
         null
      end;
```

```
    <seção crítica>
    Flag2:= down;
    <seção não-crítica>
     -
   end
end P2;
```

Se somente um processo deseja entrar em sua seção crítica, então a outra flag irá ser down (baixa) e a entrada será imediata. Mas, se ambas as flags têm sido elevadas (up), então o valor turn torna-se significante. Para quatro situações de interleaving há a sincronização dos processos, em que um processo entra na seção crítica e o outro processo fica no busy loop.

Outros métodos e primitivas podem ser adicionados para facilitar a exclusão mútua e condição de sincronização em programação concorrente. *Entre elas, podemos citar semáforos.*

4.3. Semáforos

Semáforos são mecanismos simples para programação de exclusão mútua e condição de sincronização. Eles foram originalmente desenvolvidos por Dijkstra, em 1986, e tem os seguintes benefícios:

- Eles simplificam os protocolos para sincronização;
- Eles removem a necessidade para busy wait loops.

Um semáforo é uma variável inteira não negativa que, excluindo a inicialização, pode ser manipulada 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.*

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.

Condição de sincronização e exclusão mútua podem ser programadas facilmente com semáforos. O exemplo abaixo considera a condição de sincronização:

```
(* 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;
```

Quando P1 executa *wait* no semáforo, que foi inicializado em 0, ele irá esperar até que o processo P2 execute um *signal* sobre o semáforo. Isso colocará o semáforo *consyn* em 1, e então o *wait* não terá mais efeito. P1 irá continuar e *consyn* será decrementado para zero. Note que se P2 executa o *signal* primeiro o semáforo irá ser *setado* (colocado para 1) e então P1 não irá ser atrasado pela ação do *wait*.

Exclusão mútua também pode ser conseguida por semáforo. Veja exemplo abaixo:

```
 (* 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;
```

Se P1 e P2 estão em contenção, então eles irão executar suas estruturas de *wait* simultaneamente. Como sempre, como *wait* é atômico então um processo irá completar a execução de sua estrutura antes do outro começar. Um processo irá executar um *wait* (*mutex*) com *mutex* = 1, o qual permitirá o processo para proceder em sua seção crítica e coloca *mutex* para 0. O outro processo irá executar *wait (mutex)* com *mutex* = 0, e então deve ser atrasado. Uma vez que o primeiro processo tem saído de sua seção crítica, ele irá executar um *signal (mutex).* Isso irá causar o semáforo para ser 1 novamente, e permitirá o segundo processo para entrar em sua seção crítica (e coloca *mutex* para 0).

O valor inicial do semáforo irá restringir o número máximo de execuções concorrentes do código. Se o valor inicial é 0, nenhum processo entrará na seção crítica. Se o valor é igual a1, então um processo simples deve entrar (exclusão mútua). Para valores maiores do que 1, então o número de execuções concorrentes do código é permitida.

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.

```
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 proceso suspenso como executável novamente
   else
     S:= S + 1
```

Vimos no capítulo anterior que o uso de *busy wait* pode causar o conceito de ***livelock***. Ou seja, um processo executando em um loop infinito ou por tempo indefinido. 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);
```

No exemplo acima, o processo P1 executa um *wait* em S1 e P2 executa um *wait* em S2. Posteriormente, o processo P1 executa um *wait* em S2 e P2 executa um *wait* em S1. A partir daí, os dois processos ficarão suspensos por tempo indeterminado.

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.

4.4. Semáforos Binários e de Quantidade

A definição de um semáforo é um inteiro não negativo. 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, isto é, a sinalização de um semáforo o qual tem valor 1, não tem efeito – o semáforo retém o valor 1. A construção de um semáforo geral de dois semáforos binários e um inteiro pode ser então ser feito se a forma geral é requerida.

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
```

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.

4.5. Monitores

O principal problema com regiões condicionais é que elas podem ser dispersas através do programa. Monitores são desenvolvidos para minimizar esse problema pela produção de regiões de controle mais estruturadas. Eles também usam uma forma de condição de sincronização que é mais eficiente de se implementar.

A região crítica a ser protegida é escrita com uma *procedure* e é encapsulada juntamente em um módulo simples, chamado ***monitor.*** Como um módulo, todas as variáveis que devem ser acessadas em uma exclusão mútua são escondidas; assim, como um monitor, todas as chamadas a uma *procedure* no módulo são garantidas para serem mutuamente exclusivas.

Monitores aparecem como um refinamento de regiões críticas condicionais. No exemplo abaixo, é apresentada a estrutura de um monitor.

```
monitor buffer;
   export append, take;
   var (* declaração de variáveis*)

   procedure append (I: integer);
   -
   -
   end;

   procedure take (I : integer)
   -
   -
   end;

begin
      (*inicialização  das variáveis do monitor)
end
```

No escopo acima, o buffer apresenta-se como um módulo distinto. Essa aproximação é também conhecida como monitor. A única diferença entre um módulo e um monitor é que no monitor as chamadas concorrentes para *append e take* são serializadas, por definição. Nenhum semáforo de exclusão mútua é necessário. Embora para implementação de exclusão mútua ainda exista a necessidade de condição de sincronização dentro do monitor. Em teoria, semáforos poderiam ainda ser usados, mas normalmente uma

primitiva de sincronização mais simples é apresentada. No monitor de Hoares, (Hoare, 1974), esta primitiva é chamada de *variável de condição* e pode ser operada sobre dois procedimentos, igualmente aos semáforos, *wait e signal*.

Quando um processo realiza uma operação *wait*, ele é bloqueado (suspenso) e é colocado em uma fila associada com esta variável de condição. Nesse caso, um processo realizando um *wait* em uma variável de condição sempre bloqueia, diferente de uma operação *wait* em um semáforo. Um processo bloqueado então libera sua exclusividade mútua no monitor, permitindo um outro processo a entrar. Quando um processo executa uma operação *signal* então ele liberará um processo bloqueado. Se nenhum processo está bloqueado na variável específica então a operação *signal* não terá efeito algum. Veja exemplo abaixo.

```
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;
```

Se um processo executa um *take* quando não existe nenhum buffer, então ele irá ser suspenso no *itemavailable*. Um processo executando um *append* no item irá sinalizar para o processo suspenso quando o item estiver disponível.

As semânticas para *wait e signal* dadas acima não estão completas. Observa-se que dois ou mais processos podem estar ativos dentro do monitor. Isso só deveria ocorrer seguindo uma operação *signal* no qual um processo bloqueado foi liberado. O processo liberado e o processo que o liberou estão então ambos executando dentro do monitor. Para proibir essa ação indesejável a semântica de *signal* deve ser alterada. Três aproximações diferentes são usadas em algumas linguagens:

- Um *signal* é permitido somente como a última ação de um processo antes de ele deixar o monitor;

- Uma operação *signal* tem o efeito de uma execução de *return*, isto é, o processo é forçado a deixar o monitor;

- Uma operação *signal* o qual desbloqueia outro processo tem o efeito de se bloquear. Este processo irá somente executar novamente quando o monitor está livre.

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
```

O procedimento *wait(s,r)* atrasa o processo até ele receber um *signal s*. O processo que atrasou recebe uma prioridade (ou atrasado *rank r*) em que r deve ser uma expressão inteira positiva, que por *default* é um (prioridade mais alta).

O procedimento *send(s)* envia um *signal s* para este processo com a prioridade mais alta, o qual tem sido aguardado por *s*. Se vários processos bloqueados têm a mesma

prioridade então o processo que tem esperado por mais tempo recebe o *signal*. O processo executando o *send* é suspenso. Se nenhum processo está esperando, a chamada não tem efeito.

A função Boolean é aguardada em *s*. É alterada para\o valor TRUE se existe pelo menos um processo bloqueado em s; caso contrário, é alterado para FALSE.

5. 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.

5.1. Sincronização de Processos

Com todo o sistema baseado em mensagem existe uma sincronização implícita, no qual o processo receptor não pode obter a mensagem antes dela ter sido mandada. Ou seja, um processo receptor só pode ler a mensagem quando ela for enviada. Observa-se que isso pode ser comparado com o conceito de memória compartilhada, já que podemos usar um canal para passagem de mensagem entre os processos concorrentes.

Se o processo executa uma operação de recepção de mensagem incondicional quando nenhuma mensagem está disponível, então ele fica em estado suspenso até a chegada da mensagem.

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.

Para apresentar a diferença entre os modelos considere a seguinte analogia. A postagem de uma carta é um envio assíncrono. Uma vez que a carta tenha sido colocada na caixa de correio, o remetente segue seu caminho sem se preocupar com o destino da carta. Somente pelo retorno de qualquer outra carta é que o remetente saberá que a primeira carta realmente chegou. Do ponto de vista do receptor, uma carta pode somente informar ao seu leitor a data do evento de envio, mas nada sobre a situação corrente do remetente.

Um telefone é a melhor analogia para uma comunicação síncrona. O emissor ou sender agora espera até o contato acontecer e a identidade do receptor ser verificada antes

da mensagem ser enviada. Se o receptor pode retornar (mandar um replay) imediatamente, então a sincronização é do tipo invocação remota. Devido ao emissor e receptor aguardarem juntos para uma comunicação sincronizada, ela é geralmente chamada de ***rendezvous***. A forma de invocação remota é conhecida como ***rendezvous extendida***.

Claramente, existe relação entre as três formas de envio. Dois eventos assíncronos podem essencialmente constituir uma relação síncrona se o reconhecimento da mensagem é sempre enviado. Veja exemplo abaixo:

P1	P2
asyn_send(message)	wait (message)
wait(acknowledgement)	asyn_send(acknowledgement)

No exemplo acima, observa-se a utilização de duas funções de envio assíncrono para se conseguir uma comunicação síncrona. Nesse exemplo, o processo emissor ou origem é o processo P1 e o processo receptor ou destino é o processo P2. A sincronização do processo P1 é conseguida pela utilização da função de recepção (wait) para aguardar pelo processo destino P2.

Ainda podemos utilizar duas comunicações síncronas para conseguir uma comunicação com invocação remota. Veja exemplo abaixo:

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

5.2. 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)

Note que mesmo com mailbox a passagem de mensagem pode ainda ser síncrona. Nomeação direta tem a vantagem da simplicidade, enquanto nomeação indireta ajuda na decomposição do software. Um mailbox pode ser visto como uma interface entre partes distintas de um programa.

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)

A função *wait* equivale à função para recepção de mensagem pelo processo destino.

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.

Se a nomeação é indireta, existem alguns temas que devem ser considerados:

- estrutura de muitos-para-um (ou seja, qualquer número de processos poderia escrever nele, mas somente um processo poderia ler dele). Esse caso, encaixa-se no paradigma cliente-servidor;
- estrutura de muitos-para-muitos (ou seja, muitos processos poderiam escrever nele, e muitos processos poderiam ler dele). Esse caso, encaixa-se no paradigma muitos clientes e muitos servidores;
- estrutura de um-para-um (ou seja, um cliente e um servidor).

5.3. 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. Mas, algumas linguagens modernas têm retirado essas restrições.

5.4. 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.

Exemplo de espera seletiva em OCCAM:

```
While true
  Alt
    ch1 ? I
       chout ! I
    ch2 ? I
       chout ! I
    ch3 ? I
       chout ! I
```

No exemplo acima, o escopo esperará uma mensagem de qualquer uma das entradas *chx*. Quando chegar uma mensagem, essa mensagem é atribuída a *I*, e, posteriormente, atribuída ao canal de saída, chamado de *chout.*

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]);
```

No exemplo acima, é declarado um vetor de três elementos do tipo canal. Logo depois, é chamada a função *alt_wait*, que deve esperar pela chegada de uma mensagem de qualquer um dos canais especificados em seu argumento. Essa função devolve um inteiro relacionado ao índice do canal no vetor. Posteriormente, é chamada a função *chan_in_word(&y,CAN[x])*, que pegará a mensagem recebida do canal e colocará em y.

6. Ações Atômicas

Antes de falarmos de ações atômicas, falaremos, mais detalhadamente, de três tipos de processos em programação concorrente: *processos independentes, processos cooperantes e processos competidores.*

Processos independentes não se comunicam ou sincronizam com outros processos. Conseqüentemente, se um erro ocorrer dentro de um processo, então procedimentos de recuperação podem ser iniciados pelo processo isoladamente do resto do sistema.

Processos cooperantes, por comparação, regularmente comunicam e sincronizam suas atividades para desempenharem alguma operação comum. Se qualquer condição de erro acontecer, é necessário que todos os processos envolvidos desempenhem recuperação de erro. Se processos cooperantes se comunicam ou sincronizam utilizando recurso compartilhado, então recuperação de erro deve envolver o próprio recurso.

Processos competidores se comunicam e sincronizam para obterem recursos. Eles são essencialmente independentes. Se um erro acontece em um processo não deveria ter efeito sobre os outros. Infelizmente, nem sempre isso acontece, principalmente se o erro ocorreu enquanto um processo estava na ação de alocar um recurso.

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.

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.

6.1. Ações atômicas de duas fases

Idealmente, todos os processos envolvidos em uma ação atômica deveriam obter os recursos requeridos, pela duração da ação. Esses recursos poderiam ser liberados após a ação atômica terminar. Se essas regras forem seguidas, então não há necessidade para uma ação atômica interagir com qualquer entidade externa, que é uma definição estrita para ação atômica. Infelizmente, esse ideal pode levar a uma utilização pobre de recursos.

Como primeiro passo, é necessário permitir uma ação atômica iniciar sem seu complemento 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.

Se recursos podem ser obtidos mais tarde e liberados 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. Como sempre, deveria ser notado que se recursos são liberados mais cedo, então deveria ser mais difícil produzir recuperação se a ação atômica falhar. Isso acontece porque recursos têm sido atualizados e qualquer outro processo deve observar o novo estado do recurso. Qualquer intenção de invocar recuperação no outro processo deve levar ao efeito dominó. Ações recuperáveis não liberam qualquer recurso até a ação ser completada com sucesso.

6.2. 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. Por falha, significa que se um erro ocorreu, a transação não pode ser recuperada, como um processador que falhou, como exemplo. 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.

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.

Embora transações atômicas são muito utilizadas para aplicações de banco de dados, sua utilização para programação de sistemas tolerantes a falhas não é aconselhado. Isso ocorre porque alguma forma de mecanismo de recuperação deve ser fornecido pelo sistema. Embora transações atômicas produzam alguma forma de recuperação de erro para trás, elas não permitem que procedimentos de recuperação sejam desenvolvidos.

6.3. 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.

- *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).

- *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.

6.4. Ações atômicas em linguagens concorrentes

Muitas linguagens não possuem suporte para implementação de ações atômicas. Assim sendo, algumas primitivas de sincronização e comunicação podem ser utilizadas para produzir ações atômicas. Seguem, abaixo, primitivas já estudadas para se conseguir ações atômicas.

- 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)
```

Nesse caso, a ação atômica está representada pelo uso da região crítica pelo processo chamador do *wait*. Observa-se que enquanto o processo está usando a região crítica, nenhum outro processo usa essa região e o processo executando a ação atômica não interage com nenhum outro processo nesse momento.

- Uso de monitor para se conseguir ação atômica

Pelo encapsulamento da ação atômica em um monitor é mais fácil de certificar que execuções parciais não são observadas. O programa 10.1 implementado como monitor é apresentado no programa 10.2. A estrutura IF no início de cada procedure certifica que somente um processo tem acesso. A variável condition então produz uma sincronização correta dentro da ação.

```
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;
```

Programa 10.1

Existem dois problemas com essa solução. Não é possível para dois processos estarem ativos dentro do monitor simultaneamente. Isto é geralmente mais restritivo do que necessário. Além do mais, a implementação de ações aninhadas, e alocação de recursos, irá requerer chamadas de monitores aninhadas. Manter um monitor travado enquanto executando uma chamada de monitor aninhada poderia desnecessariamente atrasar processos tentando executar dentro de uma ação externa.

```
Monitor atomic_action
    Export code_for_first_process, code_for_second_process;

    First_process_active: Boolean := false;
    second_process_active: Boolean := false;
    First_process_finished: Boolean := false;
    second_process_finished: Boolean := false;
    no_first_process, no_second_process : condition;
    atomic_action_ends1, atomic_action_ends2 : condition;

Procedure code_for_first_process
Begin
   If first_process_active then
       Wait(no_first_process);
   First_process_active := true;
   - - get resource in non-sharable mode
   - - update resource
   - - signal second process that it is ok
   - - for it to acess resource
   - - any final processing
   If not second_process_finished then
       Wait(atomic_action_ends2);
   First_process_finished := true;
   - - release resource
   Signal (atomic_action_ends1);
   First_process_active := false;
   Signal (no_first_process);
End;


Procedure code_for_second_process is
Begin
   If second_process_active then
       Wait(no_second_process);
   second_process_active := true;
   - - initial processing
   - - wait for first process to signal
   - - that it is ok to acess rsource
   - - access resource
   Signal (atomic_action_ends2);
```

```
    second_process_finished := true;
  If not first_process_finished then
     Wait(atomic_action_ends1);
  second_process_active := false;
  Signal (no_second_process);
End;
```

Programa 10.2

7. 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. Em muitos casos, comportamento de tempo-real pode estar associado com a alocação de recursos entre processos competidores. Embora os processos não comuniquem diretamente com o outro para passar informação sobre suas próprias atividades, eles devem se comunicar para coordenar acesso a recursos compartilhados. Em um caso de poucos recursos para muitos processos já limita o acesso concorrente desses recursos.

Este capítulo apresenta o problema de controle de recurso confiável. A alocação geral de recursos entre processos competidores também é considerado. Embora tais processos sejam dependentes dos outros, a ação de alocação de recursos tem implicações na confiabilidade. Em particular, a falha de um processo poderia resultar em um recurso alocado indisponível para outros processos. Além do mais, processos podem vir a ficar travados pela espera de recursos que outros processos requerem enquanto ao mesmo tempo requerem mais recursos.

Para processos que concorrem 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.

7.1. Controle de recurso e ações atômicas

Embora processos necessitem comunicar e sincronizar em ordem para desempenhar alocação de recurso, não se faz necessário ser na forma de ação atômica. Isso se deve porque é necessário apenas uma troca de informações para se conseguir um correto compartilhamento de recurso. Não é possível trocar informações arbitrariamente. Como resultado disso, o controlador de recurso, seja ela o processo ou o monitor, está apto a certificar a acessibilidade global de qualquer mudança em seu dado local. Se isso não fosse o caso, então um processo que alocou um recurso falho deveria informar de sua falha a todos os processos que recentemente comunicaram com o controlador de recurso. Assim, o código necessário para um processo particular comunicar com o controlador deveria ser uma ação atômica. Mas, nenhum outro processo no sistema pode interromper um processo

quando ele está alocando ou liberando um recurso. Além do mais, um alocador de recurso e um processo cliente devem usar recuperação de erro para trás e para frente para lidar com qualquer condição de erro antecipado e não-antecipado.

7.2. Gerência de recurso

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;
```

Em occam2, a forma natural de um gerenciador de recurso é um processo que tem sido instanciado de uma procedure (PROC) com parâmetros canal:

```
PROC resource.manager ([ ] CHAN of any request,
                       [ ] CHAN of resource allocate,
                       [ ] CHAN of resource free)
....
:
```

Uma maneira de controlar o acesso de processos a recursos é usando o conceito de processos servidores e clientes. Processos servidores controlam acesso para recursos do sistema. Processos clientes produzem uso de serviços disponíveis. Essa distinção leva ao paradigma cliente-servidor. Onde comunicação e sincronização são baseadas em monitores, geralmente é possível encapsular um recurso com somente um monitor; nenhum processo servidor é necessário.

Critérios para avaliação de primitivas de sincronização podem ser utilizados no contexto de gerenciamento de recurso. Esses critérios podem ser estentidos para cobrir qualquer interação cliente-servidor, dos quais gerenciamento de recurso é um caso particular. Na próxima seção, cobriremos esse conceito.

7.3. 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.

No contexto de interações cliente-servidor e controle de recurso, a informação necessária para expressar essas restrições pode ser categorizada como segue:

- tipo de serviço requerido;
- ordem no qual o pedido chega;
- o estado do servidor e qualquer objeto que ele gerencia;
- os parâmetros do pedido ou requisição.

7.2. 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)
```

8. 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.

8.1. 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.

Exemplo 1 de acesso ao clock em OCCAM:

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

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();
```

8.2. 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;

Será a melhor maneira de realizar atraso de um processo? Observa-se que o processo realiza esse atraso em execução. Ou seja, ele fica sempre em execução, dentro de um loop, até que o tempo passe.

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

delay 10.0; espera dez segundos

Observa-se a eliminação do *busy wait.*

Em linguagem C para PIC, temos a seguinte função:

delay_ms(1000); espera um segundo.

Essas linguagens contam com um suporte de run-time que colocam o processo em estado suspenso enquanto aguarda o tempo passar.

8.3. 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.

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;
```

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).

Exemplo de timeout em OCCAM:

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

8.4. Deadline (Prazo de Encerramento)

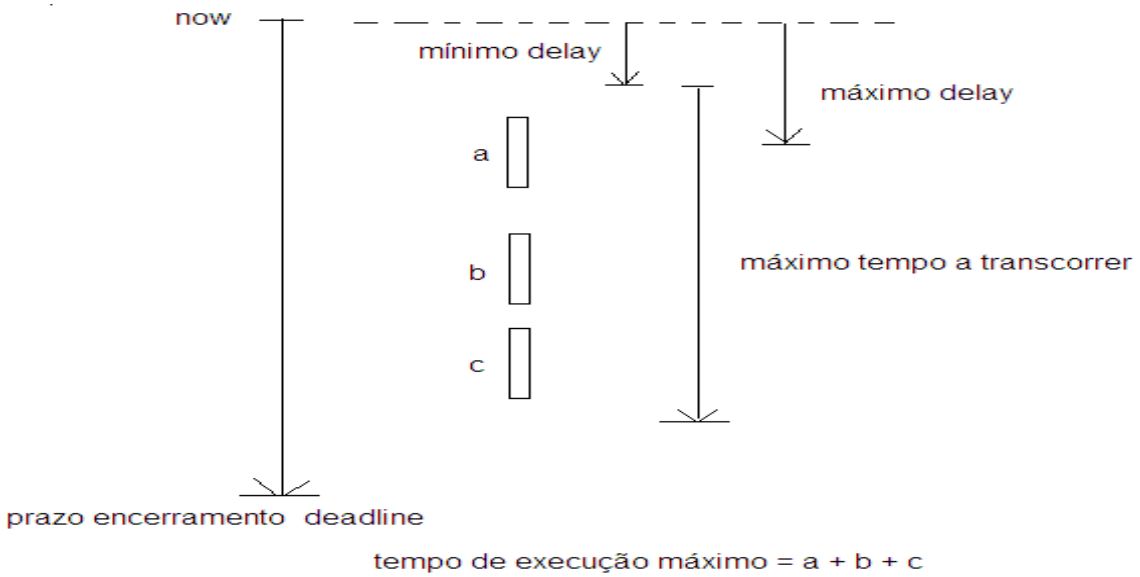
Antes, temos que falar primeiro sobre alguns conceitos importantes, como tipos de processos em relação ao tempo. Existem dois tipos de processos em relação ao tempo:
- Processos periódicos: são processos que amostram dados ou executam um loop de controle e têm um prazo de encerramento explícito que deve ser atendido;
- Processos aperiódicos ou esporádicos: em geral, são processos executados por eventos assíncronos fora do sistema embarcado; esses processos têm um tempo de resposta especifico associados a eles.

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.

- 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.

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



- 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.