UNIVERSIDADE DE COIMBRA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

Sistemas de Tempo Real: Apontamentos Teóricos (Parte I)

Rui Araújo

Email: rui@isr.uc.pt
http://home.isr.uc.pt/~rui
Tel: 239796276
Tel. Int.: 421352

Coimbra
Setembro, 2022

Informação: Estes apontamentos destinam-se exclusivamente a utilização interna com o objectivo de fornecer apoio à leccionação e estudo de disciplinas na área dos Sistemas de Tempo Real (STR) em cursos no Departamento de Engenharia Electrotécnica e de Computadores (DEEC), da Faculdade de Ciências e Tecnologia da Universidade de Coimbra (FCTUC). Os apontamentos não se destinam a utilização comercial. Estes apontamentos inspiram-se em, e seguem e coligem, principalmente materiais existentes nos livros "Alan Burns and Andy Wellings, Real-Time Systems and Programming Languages, Second Edition, Addison-Wesley, Harlow, England, 1996", mas também em materiais existentes nos livros "C. M. Krishna and Kang G. Shin, Real-Time Systems, McGraw-Hill, New York, USA, 1997", "Phillip A. Laplante, Real-Time Systems Design and Analysis: An Engineer's Handbook, Second Edition, IEEE Press, New York, USA, 1997.", "William Stallings, Operating Systems: Internals and Design Principles, Third Edition, Prentice-Hall, Upper Saddle River, New Jersey, USA, 1998.". Estes apontamentos seguem também, na parte de trabalhos laboratoriais, conteúdos desenvolvidos pelo Prof. Paulo Menezes na leccionação de disciplinas na área STR no DEEC-FCTUC.

Sistemas de Tempo Real

Programa resumido da disciplina:

1. Temporização e predictabilidade.
2. Aplicações em tempo-real.
3. Problemas relacionados com arquitecturas de computadores.
4. Escalonamento.
5. Sincronização.
6. Estruturação de software de tempo-real.
7. Análise de desempenho.
8. Tolerância a falhas.
9. Sistemas integrados (embedded).
10. Programação directa, programação por módulos, projecto de um sistema integrado.

Bibliografia:

1. Alan Burns and Andy Wellings, *Real-Time Systems and Programming Languages*, Second Edition, Addison-Wesley, Harlow, England, 1996.

2. Computer Systems Research Group 650, *4.4BSD Programmer's Reference Manual*, A USENIX Assoc. Book Comp., Sebastopol, California, USA, 1994.

3. Computer Systems Research Group 650, *4.4BSD Programmer's Supplementary Ducuments*, A USENIX Assoc. Book Comp., Sebastopol, California, USA, 1994.

4. C. M. Krishna and Kang G. Shin, *Real-Time Systems*, McGraw-Hill, New York, USA, 1997.

5. Phillip A. Laplante, *Real-Time Systems Design and Analysis: An Engineer's Handbook*, Second Edition, IEEE Press, New York, USA, 1997.

6. José Alves Marques e Paulo Guedes, *Fundamentos de Sistemas Operativos*, Editorial Presença, Lisboa, 1994.

7. William Stallings, *Operating Systems: Internals and Design Principles*, Third Edition, Prentice-Hall, Upper Saddle River, New Jersey, USA, 1998.

8. Milan Milenkovic, *Operating Systems*, McGraw-Hill, New York, USA, 1992.

9. Steve Oualline, *Practical C Programming*, O'Reilly & Associates, Sebastopol, California, USA, 1992.

10. W. Richard Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley, Reading, Massachusetts, USA, 1992.

Classes de Aplicações

• Podemos dividir as aplicações em 2 partes: "transformativas" e "reactivas".

Aplicações Transformativas

• Aplicações batch

  – Antes de 1970.

  – Não têm requisitos temporais.

• Aplicações transaccionais

  – Por exemplo, uma compilação.

• Nesta classe, as aplicações recebem toda e qualquer entrada quando se inicia a execução, e fornecem os resultados quando terminam.

Aplicações Reactivas

• Aplicações interactivas

  – Sistemas operativos, browsers WWW, interfaces gráficos, ...

  – Os requisitos temporais são modestos

• Aplicações de controlo em tempo-real

  – Requisitos temporais exigentes

  – Metas

• Nesta classe, as aplicações recebem os dados ou entradas, e produzem as saídas (ou respostas) no decorrer do programa.

O que é um Sistema de Tempo Real?

• **Segundo o Oxford Dictionary of Computing:** "Qualquer sistema para o qual é importante o instante em que a saída é produzida. Isto porque normalmente a entrada corresponde a algum movimento no mundo físico, e a saída tem que se relacionar com esse mesmo movimento. O atraso entre os instantes de entrada e de saída deverá ser suficientemente pequeno para obter um desempenho razoavelmente atempado".

• **Segundo Young [1982]:** "Um sistema de tempo real é qualquer sistema de processamento de informação que tenha que responder a estímulos de entrada gerados externamente dentro de um período de tempo finito e conhecido."

• **Segundo Randell *et al.* [1995]:** "Um sistema de tempo real é um sistema ao qual é requerido que reaja a estímulos do ambiente (incluíndo a própria passagem do tempo - [o tempo também é uma entidade física...]) dentro de intervalos de tempo ditados pelo ambiente."

• **Outra definição:** Um sistema de tempo real pode ser definido como aquele em que o correcto funcionamento depende não só do resultado lógico do processamento realizado, mas também dos instantes temporais em que os resultados são produzidos.

• Uma falha em responder atempadamente é tão prejudicial como a geração de uma resposta errada.

• **Sistemas de tempo real duros ("hard"):** sistemas onde é absolutamente imperativo que as respostas ocorram dentro do prazo requerido. Por exemplo: sistemas de controlo de voo.

• **Sistemas de tempo real suaves ("soft"):** são aqueles em que os tempos de resposta são importantes mas que continuam a funcionar correctamente se ocasionalmente não forem respeitadas metas temporais. Por exemplo: um sistema de aquisição de dados numa aplicação de controlo de processos.

• **Sistemas de tempo real firmes:** são sistemas de tempo real suaves para os quais não existe nenhum benefício se os serviços forem concluídos com atraso.

• Muitos sistemas são simultaneamente sistemas de tempo real duros e sistemas de tempo real suaves.

• Alguns serviços podem mesmo ter metas duras e suaves. Por exemplo a resposta a algum evento de aviso pode ter uma meta de 50ms (para uma reacção com eficiência óptima) e uma meta de 200ms (para garantir que que não ocorre nenhum estrago no equipamento ou no pessoal).

Sistemas "Hard Real-Time"

• A maior parte dos sistemas "hard" são sistemas de controlo.

• A maior parte dos sistemas de tempo-real não são "hard".

• Muitos sistemas de tempo real "hard" são críticos em relação à segurança.

• Erro comum: "Tempo-Real = cálculos feitos a muito alta velocidade". → isto não é verdade!

• A execução deve ser feita a uma velocidade que permita cumprir os requisitos temporais.

Sistemas Embebidos

• Quando um computador é apenas um sistema de processamento de informação que se integra como componente de um sistema de engenharia mais vasto diz-se que se trata de um **computador embebido**.

• O computador começou a ser utilizado na industria para o controlo de processos no início da década de 60, sendo actualmente uma norma a utilização de microprocessadores para esse fim.

Eventos

• Evento: Qualquer ocorrência que cause que o apontador de programa varie de forma não-sequencial é considerada uma alteração no fluxo de controlo e, portanto, um evento.

• Os eventos podem ser:

  • Internos: se resultarem de uma transição de estado do sistema.

  • Externos: gerados por fontes externas - normalmente originárias do ambiente onde o sistema está integrado.

• Os sistemas de tempo real podem responder a eventos...

  • Eventos periódicos

  • Eventos aperiódicos: o intervalo médio entre chegadas pode ser reduzido

  • Eventos esporádicos: o intervalo médio entre chegadas é elevado

• Os eventos podem-se ainda classificar em duas categorias:

  • Eventos síncronos: ocorrem em instantes de tempo previsíveis no fluxo de controlo de processamento.

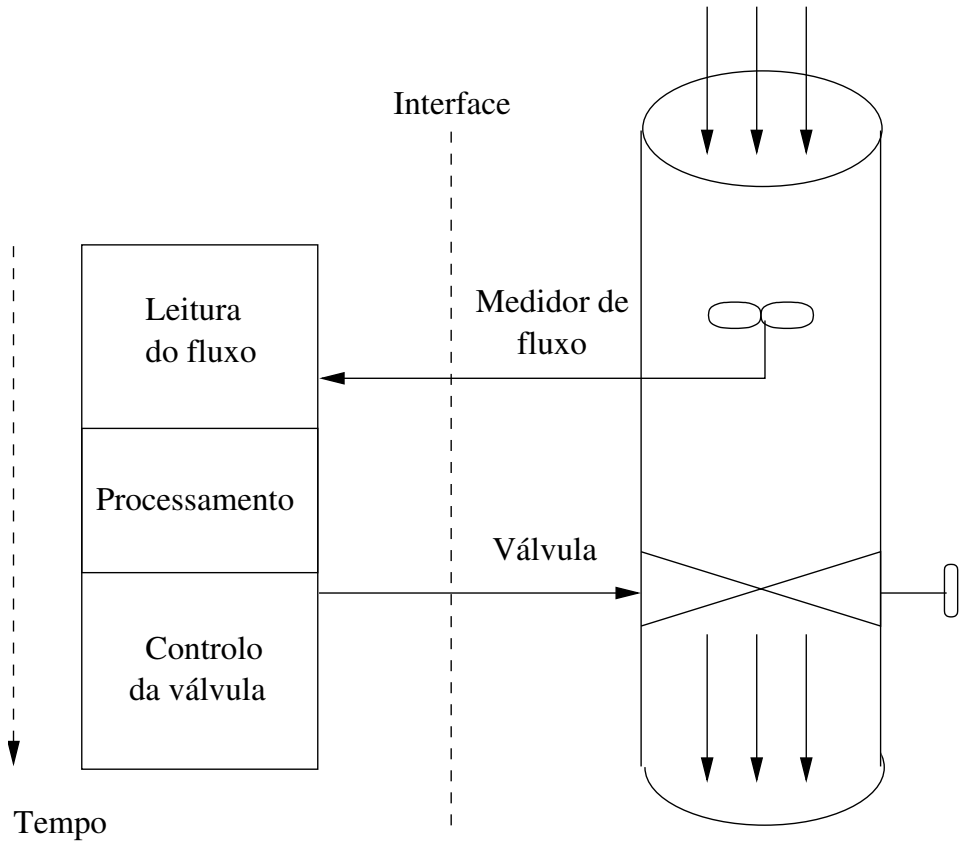
  • Eventos assíncronos: ocorrem em instantes imprevisíveis relativamente ao fluxo de controlo do processamento e são usualmente causados por fontes externas.

Aplicações em Tempo Real

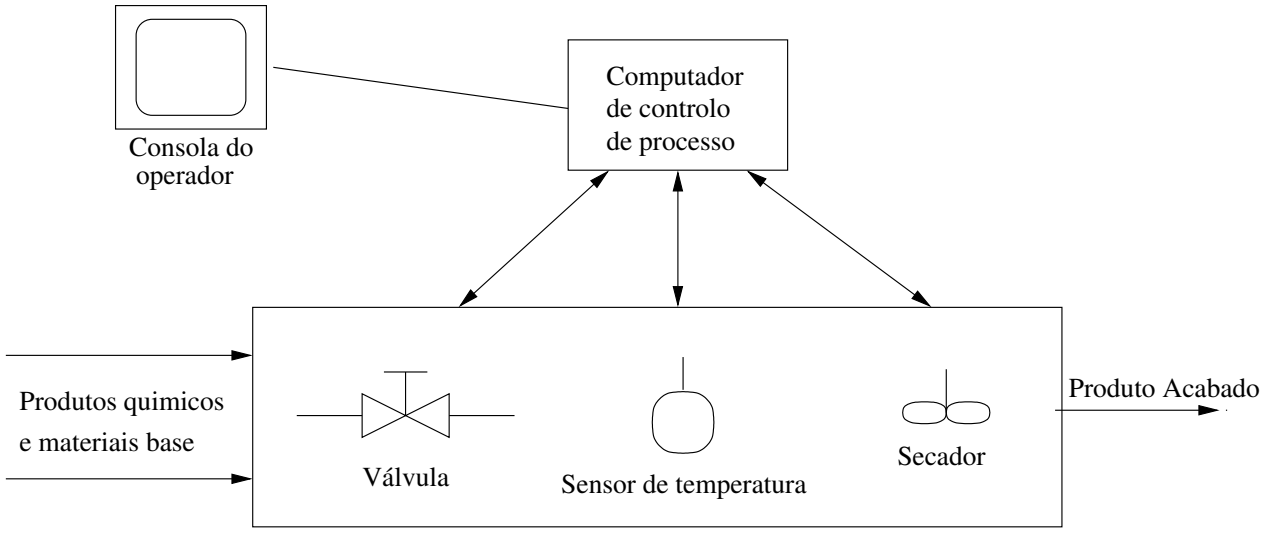
- Sistemas de controlo de processos
- Produção/manufactura
- Comunicação, comando e controlo
- Controlo de experiências de laboratório
- Robótica
- Controlo de aviões, helicópteros, etc
- Controlo de tráfego aéreo
- Telecomunicações
- Sistemas de controlo e comando militares
- Naves espaciais
- Estações espaciais
- Exploração submarina
- Veículos terrestres autónomos
- ...

Controlo de Processos

• Um sistema de controlo de fluxo (apenas um componente d1 sistema maior):



• Um sistema de controlo de processo:

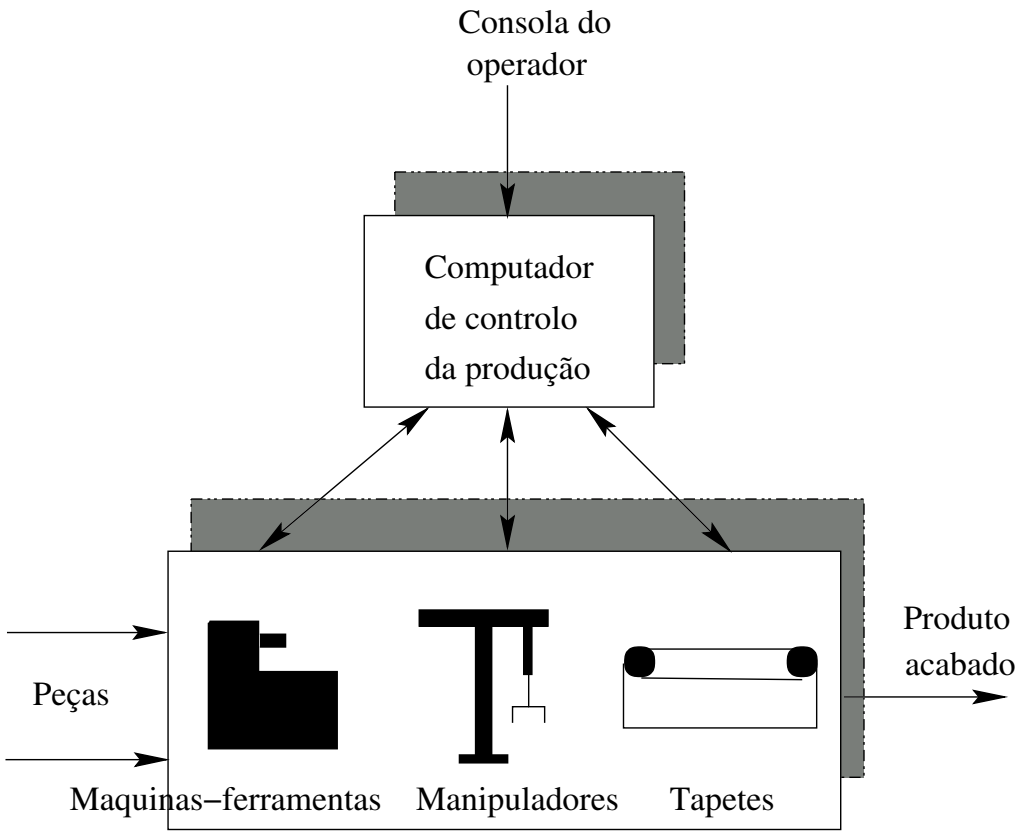


Manufactura

• O uso de computadores em manufactura tornou-se essencial nos últimos anos, pois permitem manter os custos de produção baixos e aumentar a produção.

• Os computadores permitem integrar todo o processo de manufactura desde o projecto do produto até ao seu fabrico.

• Exemplo de um computador de controlo de produção:



• As máquinas ferramentas, os manipuladores, e os tapetes rolantes são todos controlados e sincronizados pelo computador de controlo de produção.

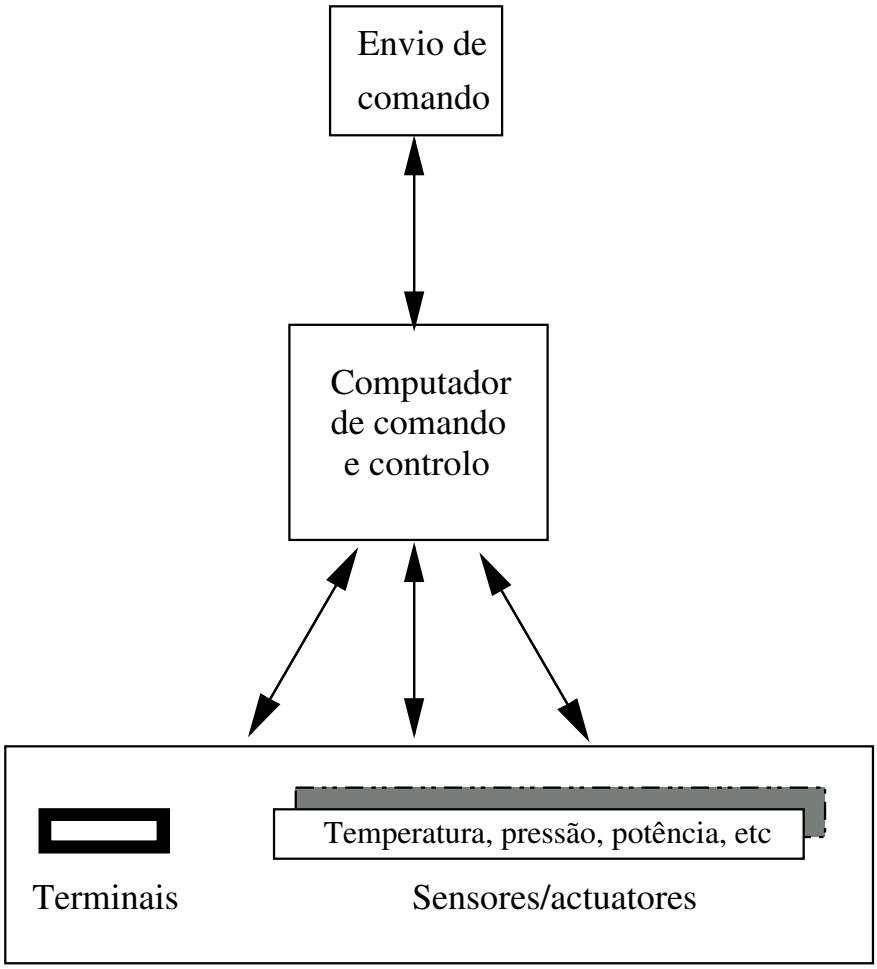
Comunicação, Comando e controlo

• Alguns exemplos: aplicações unidades de cuidados médicos automatizadas, controlo de tráfego aéreo, acesso remoto a contas bancárias, controlo de redes telefónicas.

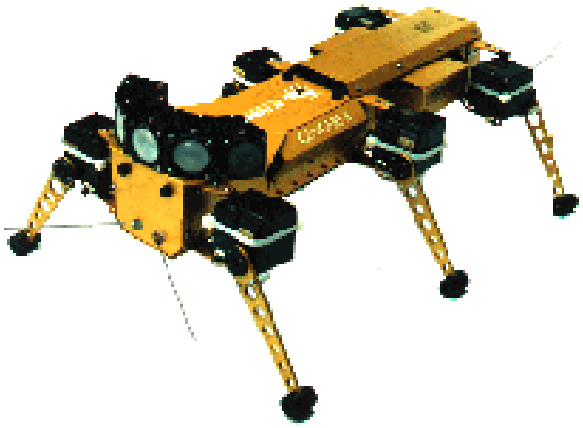
• Todos estes sistemas requerem conjuntos complexos de políticas, dispositivos de recolha de informação e procedimentos administrativos que suportem e permitam implementar decisões.

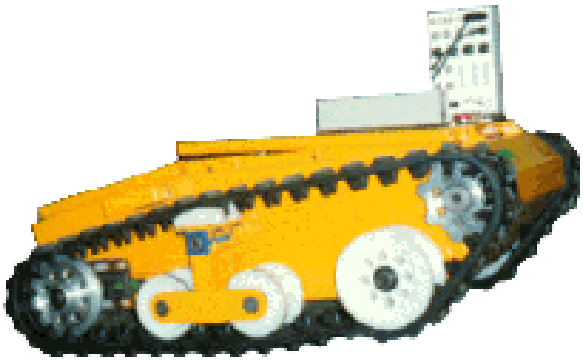
• *Sistemas distribuídos:* é comum ter os instrumentos e os dispositivos de recolha de informação distribuídos por uma área mais ou menos grande, o que requer a utilização de meios de comunicação entre os módulos do sistema.

• Exemplo de um sistema de comando e controlo:



Outros exemplos (sistemas robóticos ou robotizados):





Exemplos do dia a dia:

Hardware e Microcontroladores

• Placas com computadores embebidos muitas vezes usadas em sistemas de tempo-real:

• Alguns microcontroladores usados em computadores embebidos:

  • Intel: i386ex, i8051, i960;

  • Motorola: m68hc11, m68hc12;

  • Philips...

  • National Semiconductors...

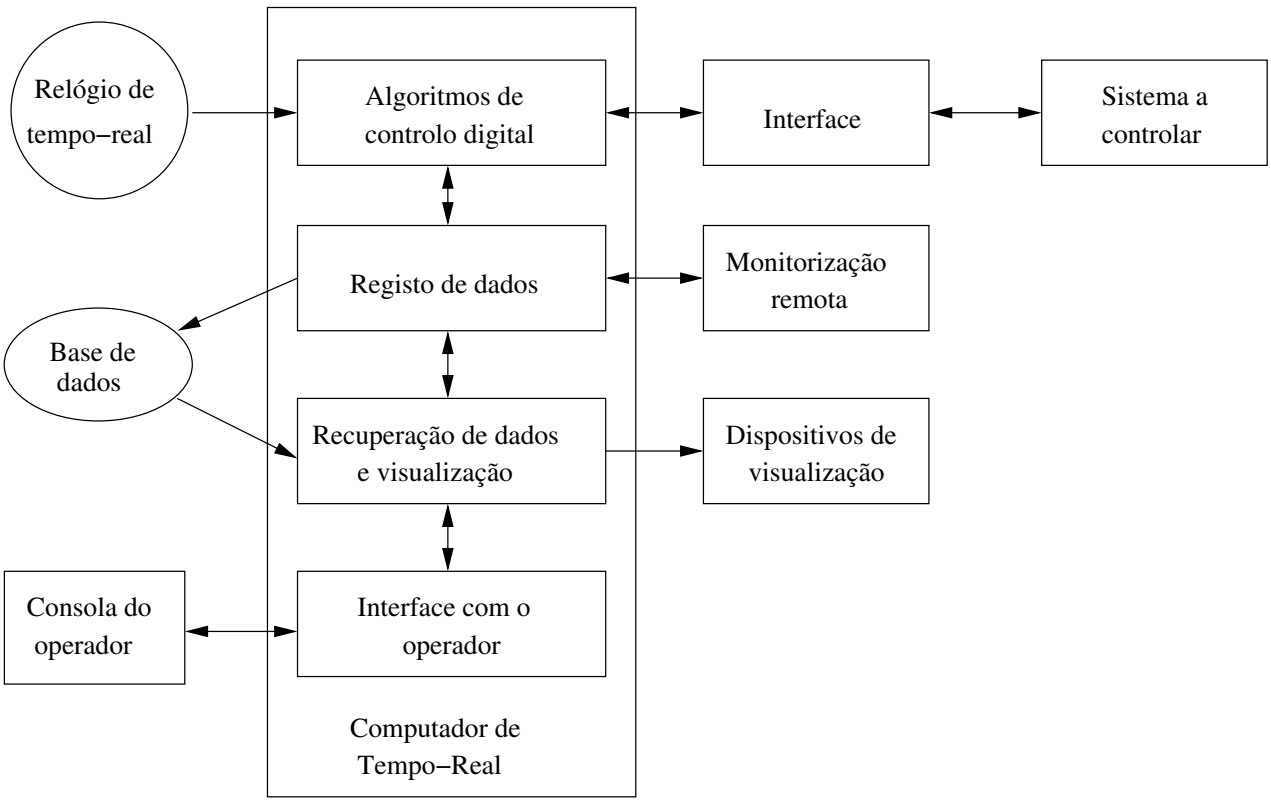
Sistema Embebido Genérico

• Sempre que é utilizado um computador para controlar dispositivos físicos a ele ligados, o computador necessita de efectuar **amostragens** sobre dispositivos de medida a intervalos regulares. Para isso é necessário um **relógio de tempo real**. Existe também em muitos casos uma **consola** através da qual um operador pode intervir manualmente. O operador é mantido informado sobre o estado do sistema através de vários **dispositivos de visualização**.

• Registos das alterações do estado do sistema são mantidos também numa base de dados, a qual pode ser consultada pelos operadores tanto no caso de análise de falhas como por questões administrativas.

• Normalmente o software que controla as operações de um sistema de tempo real pode ser escrito em módulos que reflectem a natureza física do ambiente.

• Um sistema embebido típico:



Características dos Sistemas de Tempo-Real

• Requisitos temporais → tempos de resposta garantidos: é necessário predizer com segurança os piores tempos de resposta do sistema; "A eficiência é importante mas a predictabilidade é essencial".

• Deve ser determinístico e previsível.

• Os piores tempos de resposta são mais importantes que os tempos médios

→ O sistema deve continuar a funcionar mesmo quando ocorra o pior caso em termos de ocorrência de eventos no ambiente, temporização, e carga de processamento no sistema.

• A cadência de execução e conclusão de tarefas deve depender apenas da evolução do tempo e não de factores como a carga do processador.

• Pode ser grande e complexo – pode variar desde algumas centenas de linhas de assembler ou C até 20 milhões de linhas de Ada estimadas na estação espacial Freedom, ou mesmo mais.

• Pode ser distribuído.

• Requisitos de fiabilidade e segurança muito elevados – tipicamente, os sistemas embebidos controlam o ambiente em que operam; uma falha de controlo pode resultar em perda de vida(s), prejuízo para o ambiente ou perdas económicas.

• Funciona durante longos períodos de tempo.

• Controlo concorrente de componentes separadas do sistema – no mundo real, vários dispositivos operam em paralelo; Por isso, torna-se melhor modelar este paralelismo através de entidades concorrentes no programa.

• Pode ter requisitos continuamente alterados – custo de reprojectar ou reescrever o software para responder a alterações contínuas de requisitos pode ser proibitivo.

→ Interessa que os sistemas sejam extensíveis para poderem sofrer manutenção e melhoramento durante os seus tempos de vida.

• Interacção forte com interfaces de hardware (por exemplo, através de portos de E/S, interrupções,...) – capacidade de programar dispositivos de uma forma segura e abstracta.

• Manipulação de números reais – pode ser necessário o processamento de números reais para se conseguir realizar determinadas tarefas antes do fim de cada meta temporal imposta. Só para dar um exemplo, podemos falar da implementação de algoritmos para controlo digital.

• Implementação eficiente é importante.

• É difícil de testar.

Concorrência

• (Quase) Todos os sistemas de tempo-real são concorrentes.

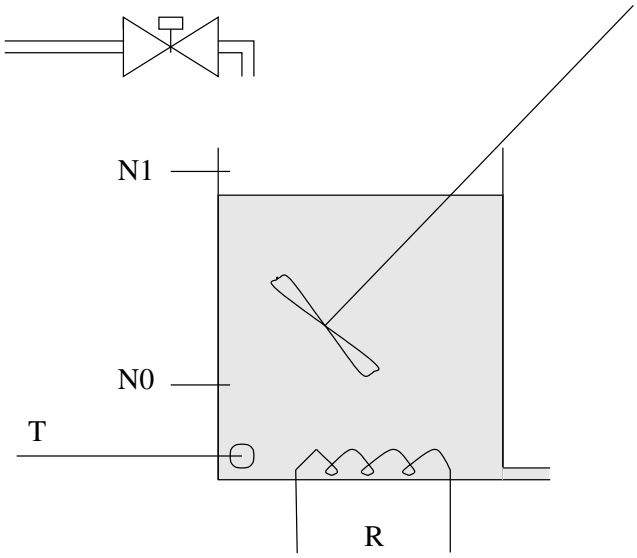
• Existem várias tarefas em execução no sistema.

• Cada tarefa ocupa-se de uma parte do funcionamento global do sistema.

• Por exemplo: podem ocorrer no sistema vários tipos de eventos; e cada tipo de evento tem uma tarefa associada.

• Isto torna o projecto e desenvolvimento do sistema mais cómodo pois muitas vezes projectam-se as tarefas separadamente apesar de fazerem parte do mesmo sistema.

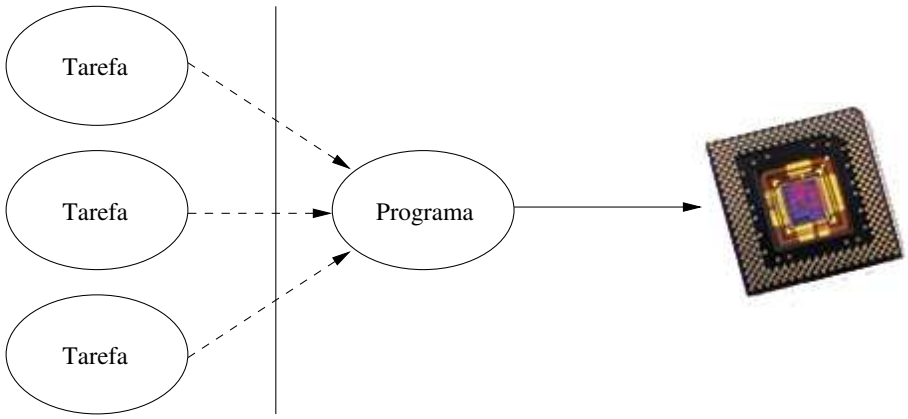
• Um exemplo:



• Neste exemplo temos 2 tarefas: controlar o nível e controlar a temperatura do fluído.

Paradigmas de Programação

• Programação Sequencial



• Uma solução possível:

```
REPETE
  ENQUANTO nivel acima de N0
    Mede Temperatura;
    Calcula o erro na temperatura;
    Calcula o comando para o aquecedor;
    Envia comando ao aquecedor;
    Espera x segundos;
  FIM ENQUANTO
  Abre valvula de entrada;
  ENQUANTO nivel abaixo de N1
    Mede Temperatura;
    Calcula o erro na temperatura;
    Calcula o comando para o aquecedor;
    Envia comando ao aquecedor;
    Espera x segundos;
  FIM ENQUANTO
  Fecha valvula de entrada;
PARA SEMPRE
```

Programação Concorrente

• (Quase) Todos os sistemas de tempo real são inerentemente concorrentes – os dispositivos do mundo real operam em paralelo.

• A programação concorrente é o nome dado às notações e técnicas de programação usadas para expressar o potencial paralelismo na solução de um problema, e para resolver os problemas de sincronização e comunicação resultantes da implementação paralela.

• A implementação real do paralelismo é um tópico de sistemas de computadores (hardware e software) que é razoavelmente independente da programação concorrente.

• A programação concorrente é importante porque fornece um enquadramento abstracto segundo o qual se pode estudar o paralelismo sem ficar atolado em detalhes de implementação.

• Um programa concorrente é convencionalmente visto como uma colecção de processos autónomos sequenciais que, em termos lógicos executam em paralelo.

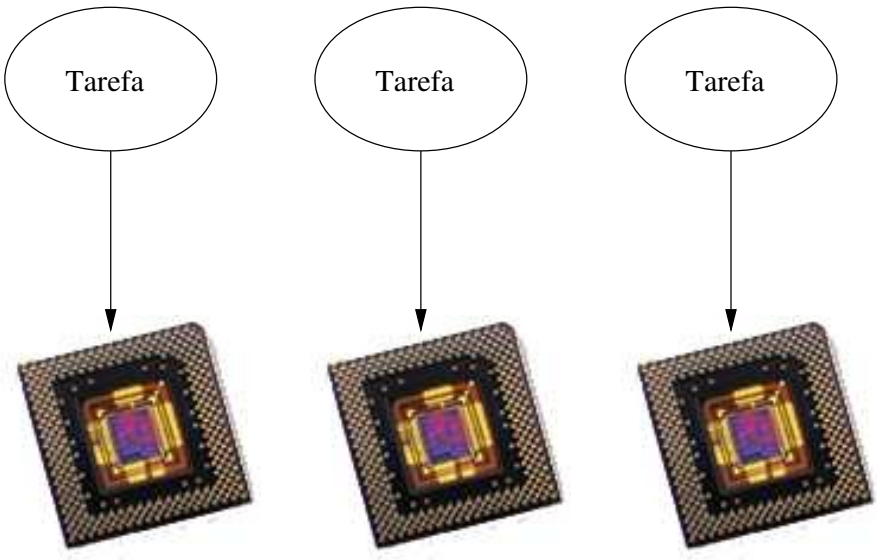
• Linguagens de programação concorrente, incorporam (elas próprias) explicitamente ou implicitamente, a noção de processo, aqui entendido como uma entidade que tem um único fio de controlo/execução.

• Linguagens de programação sequencial necessitam de suporte do sistema de operação para implementar programação concorrente.

• A real implementação (i.e. execução) de uma colecção de processos toma uma de duas formas. →

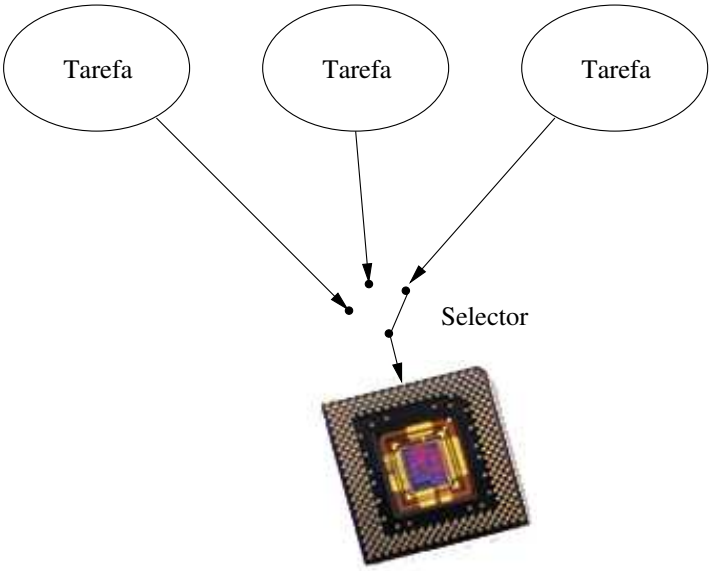
• **Programação Paralela:** quando existem várias entidades de processamento (e.g. processadores): dois exemplos →

- Multiprocessamento: os processos multiplexam as suas execuções num sistema multi-processador onde existe acesso a memória partilhada. Exemplo: uma hipótese é atribuir-se uma tarefa a cada processador:



- Processamento Distribuído: os processos multiplexam a sua execução em diversos processadores que não possuem memória partilhada.

• **Multi-programação:** um processador reparte o seu tempo executando sequencialmente várias tarefas. Os processos multiplexam a sua execução num único processador sendo criada a aparência de que os processos estão a correr em paralelo. Pode-se considerar que existe uma execução em paralelo em termos lógicos.



Sistemas Operativos de Tempo-Real

Comutam entre os vários processos em execução.

Fornecem primitivas de temporização aos processos.

Fornecem primitivas de suporte à comunicação e sincronização entre processos.

• Alguns exemplos de SOs de Tempo-Real

- QNX
- VxWorks
- VRTX
- RTKernel
- RTEMS
- Chimera
- Spring (Sun)
- LynxOS
- OS9
- iRMX
- ...

Linguagens de Programação em Tempo-Real

• Linguagens "Assembly".

• Linguagens sequenciais (e.g. C).

⇒ Nestas linguagens existe apenas um fio de controlo/execução em cada programa. Começam a execução nalgum estado e depois prosseguem executando uma instrução de cada vez até o programa terminar.

⇒ Estas requerem suporte por parte de um sistema operativo para:
- Implementar concorrência entre processos.
- Fornecer mecanismos de suporte à programação de tempo-real.

• Ada
• Modula-2
• ...

⇒ Estas são linguagens de "alto nível" viradas para a criação de aplicações multiprogramadas. Não necessitam de suporte do sistema operativo.

Algumas das Linguagens mais Usadas em Tempo-Real

• C/C++ com suporte de tempo-real (por parte do S.O.).

• Occam2 - Originalmente para transputers INMOS (SGS Thompson)

• Ada - Linguagem vencedora do concurso do DoD (EUA) durante a "software crisis". Os criadores foram CII Honeywell-Bull (França).

Formas de Multi-Programação (ou de Programação Concorrente)

• **Multiprogramação cooperativa:** o processo solicita ao sistema operativo para "dar a vez" de execução a outro processo.

- Requer programação bastante disciplinada pois está sujeita a erros. Por exemplo: se uma tarefa bloquear à espera de uma condição que nunca se verifique poderá bloquear todo o sistema.

• **Multiprogramação preemptiva:** o processo em execução sofre involuntariamente uma paragem da sua execução pelo sistema

- Neste último caso, o sistema operativo baseia-se em critérios predefinidos para interromper ou não o processo em execução e seleccionar o próximo a executar.
- A comutação de processos pode ser originada pela ocorrência de eventos como por exemplo interrupções de hardware geradas por um dispositivo periférico ou interrupções de software (e.g. uma chamada ao sistema por parte de um processo). As interrupções podem ocorrer de forma periódica, aperiódica, esporádica, ou uma combinação destas.
- Um relógio é uma possível fonte de interrupções periódicas. Sempre que ocorre uma tal interrupção, é criada uma oportunidade para o SO poder comutar de tarefas/processos.
- Para tal é necessário salvar o "contexto" do processo, para que mais tarde possa ser restaurado permitindo a continuação do processo a partir do ponto onde foi interrompido. O "contexto" inclui o conteúdo dos registos do processador (incluindo o ponteiro de instruções (IP)), e quaisquer outros dados relativos ao processo que possam ser alterados/utilizados por outro processo.

Processos e Fios de Execução

• Todos os sistemas de operação fornecem processos.

• Os processos são executados na sua própria máquina virtual, contendo um espaço de endereçamento separado para evitar a interferência entre processos.

• Recentemente os S.O. têm incluido mecanismos para criar entidades com características semelhantes a processos dentro de uma mesma máquina virtual. São os chamados fios de execução (ou "threads" ou "light-weight processes").

• Cada processo pode ter mais do que um fio de execução, existindo concorrência entre os diversos fios.

• Os fios de execução podem aceder sem restrições à máquina virtual do seu processo.

• O programador e a linguagem de programação devem propiciar protecção contra interferência indesejável entre processos.

• As linguagens Ada e occam2 fornecem mecanismos de concorrência dentro de um programa/processo.

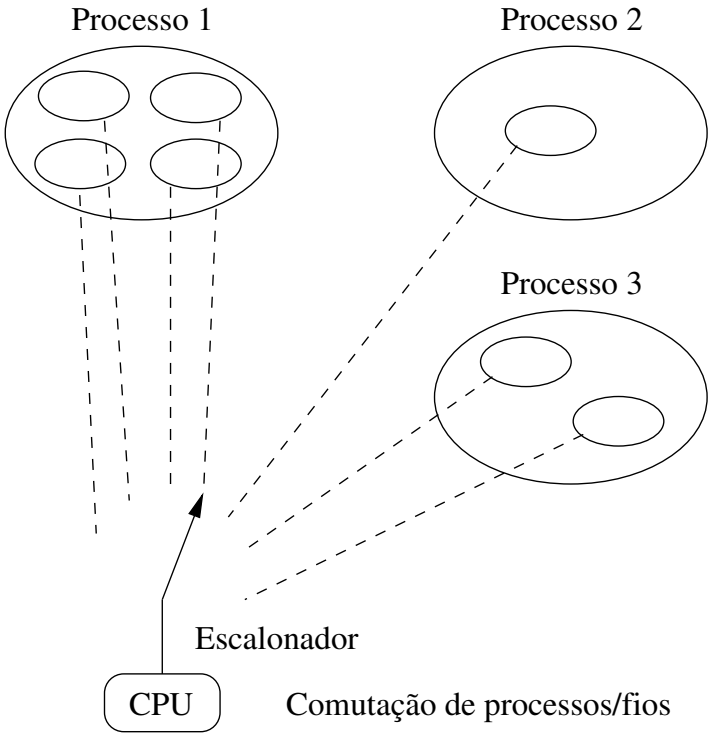
• As linguagens C e C++, só por si, não fornecem mecanismos de concorrência dentro de um processo. No entanto com o auxílio do S.O. e/ou de bibliotecas de funções estes mecanismos podem passar a estar presentes.

• **Processos:** Além de possuir um espaço de processamento independente, cada processo tem associado um grande contexto composto por um conjunto de variáveis de estado que incluem os registos do processador, tabelas de gestão de memória, tabelas de recursos utilizados (e.g. ficheiros abertos), etc.

• **Fios de execução:** Os fios de execução de um mesmo processo, por usarem um mesmo espaço de endereçamento comum, possuem a vantagem de possibilitarem uma troca de contexto mais rápida (e.g. salvando e repondo apenas os registos do processador) e a partilha de variáveis (e.g. para comunicação entre fios) e código.

• Um processo tem um ou mais fios de execução. O escalonador vai seleccionando entre fios de execução. No entanto a troca de contexto não requer sempre uma completa comutação de processos.

• Comutação entre processos/fios: exemplo →



• **Mecanismos de Programação Concorrente:** embora variem de uma linguagem (e/ou sistema de operação) para outra existem três mecanismos fundamentais que devem estar presentes:

- Execução concorrente através da noção de processo e/ou fio de execução.
- Sincronização entre processos.
- Comunicação entre processos.

• **Interacção entre processos:** podemos distinguir entre três tipos de comportamento.

- Independente: os processos não se comunicam nem se sincronizam entre si.
- Cooperante: os processos comunicam e sincronizam as suas actividades regularmente de forma a desempenhar alguma operação comum.

- Competição: os processos competem por obter/utilizar uma fatia justa dos recursos do sistema (e.g. dispositivos periféricos, memória, processador). A alocação destes recursos requer a sincronização e comunicação entre processos no sistema. No entanto os processos têm uma operação que é essencialmente independente.

• Trocas de Contexto Revisitadas

• Consiste em guardar toda a informação pertinente sobre o estado de um processo quando este é interrompido, e restaurar a correspondente informação relativa ao processo que vai ser re-activado/restaurado, para assim permitir a troca de execução. O processo restaurado continua a executar a partir do ponto/estado em que tinha sido a última vez interrompido.

• Informação a guardar/restaurar:

1. O conteúdo dos registos do processador, incluindo o valor do ponteiro de instrução.
2. Os registos do co-processador aritmético.
3. Registos relativos ao mecanismo de gestão de memória (e.g. relativos a segmentação e/ou paginação).
4. Variáveis especiais usadas pelo sistema operativo relativas a cada processo (e.g. ficheiros abertos).

• Esta informação é de tal forma valiosa que é comum desactivar as interrupções durante o processo de salvaguarda/restauro do contexto, por forma a que não possa ocorrer qualquer tipo de interferência.

• A troca de contexto pode afectar o tempo de resposta do sistema. Por isso é um dos principais objectivos, no projecto de um sistema de tempo-real, que essa operação seja o mais rápida possível.

• Como obter uma troca de contexto rápida? – Isso é conseguido usando o mínimo de instruções para o fazer, escrevendo o código directamente em linguagem assembly e guardando apenas a quantidade de informação sobre cada processo que é estritamente necessária para mais tarde se poder restaurar o seu funcionamento.

Sistemas Baseados em Interrupções

• Nestes sistemas as várias tarefas estão associadas a rotinas de serviço às interrupções, sendo o escalonamento feito pelas interrupções de hardware e software. Por exemplo, um relógio que gere periodicamente interrupções, pode ser uma fonte de escalonamento periódico de uma tarefa.

- As interrupções podem ser periódicas, aperiódicas, esporádicas ou uma combinação de vários tipos.

Sistemas Background/Foreground

• Tarefas foreground são aquelas que são activadas por eventos.

• São uma melhoria aplicada aos sistemas baseados apenas em interrupções, nos quais o ciclo vazio ("tarefa idle") do programa principal é substituído por trabalho útil (uma "tarefa de background").

• Esse código é a parte background do sistema e executa apenas quando não há processos foreground (associados às interrupções) activos. Ou seja os processos background têm a prioridade mais baixa de todo o sistema.

• Tarefas background → usar para tarefas para as quais o tempo não seja um factor crítico.

• A duração média do ciclo de execução da tarefa "background" é

$$t = \frac{t_b}{1 - p}$$

onde t é o tempo de execução do ciclo da tarefa "background", p a percentagem de ocupação do processador pelas tarefas "foreground", t_b tempo de cálculo necessário para um ciclo da tarefa "background".

• O nível "foreground" é também chamado *nível de interrupção*.

• O nível "background" é também chamado *nível de tarefa*.

• **Arranque do Sistema:** Este tipo de sistemas começa por executar a tarefa background que efectua um conjunto de operações com vista a iniciar o funcionamento das tarefas de tempo-real:

1. Inibe as interrupções.
2. Instala os vectores de interrupção e pilhas das tarefas.
3. Testa a integridade do sistema.
4. Inicia o sistema.
5. Re-habilita as interrupções.

• Após o início do sistema, a tarefa background começa a desenvolver outro processamento que seja relevante para o funcionamento do sistema.

• Muitos sistemas de grande produção baseados em microcontroladores (e.g. fornos micro-ondas, frigoríficos, máquinas de lavar, telefones, brinquedos, etc) usam processamento "background/foreground".

Sistemas Multi-tarefa

• Um sistema multi-tarefa é um sistema concorrente. Baseia-se no escalonamento e comutação do CPU entre diversas tarefas. Um único CPU comuta a sua atenção entre diversas tarefas sequenciais.

• Um sistema multi-tarefa pode ser encarado como uma extensão de um sistema "background/foreground".

- Nos sistemas baseados em interrupções e nos sistemas "background/foreground", as oportunidades de comutação de tarefas são quando ocorre uma interrupção.
- Nos sistemas multi-tarefa, os momentos/oportunidades para realizar comutação de tarefas deixam de ser apenas a ocorrência de interrupções/eventos (as quais têm tarefas de resposta associados). A comutação de tarefas passa também a ser efectuada com uma certa cadência temporal (normalmente de forma periódica) por parte do SO.

Modelo de Blocos de Controlo de Tarefas (TCB)

• É uma estrutura de dados que serve de ferramenta para implementar os mecanismos do SO, como por exemplo a salvaguarda e o restauro do contexto de uma tarefa mas não só.

• Esta técnica associa a cada tarefa, o bloco de controlo de tarefa (TCB), onde são guardados todos os elementos relativos ao estado/contexto da tarefa, incluindo por exemplo os registos, o PC, uma identificação, e a prioridade (caso seja utilizada).

• Exemplo de um TCB:

IP
Estado
Identificador
EAX
EBX
...
Ponteiro para o próximo TCB

Escalonador

• O escalonador é uma entidade do sistema operativo responsável pela determinação da ordem pela qual as tarefas são executadas.

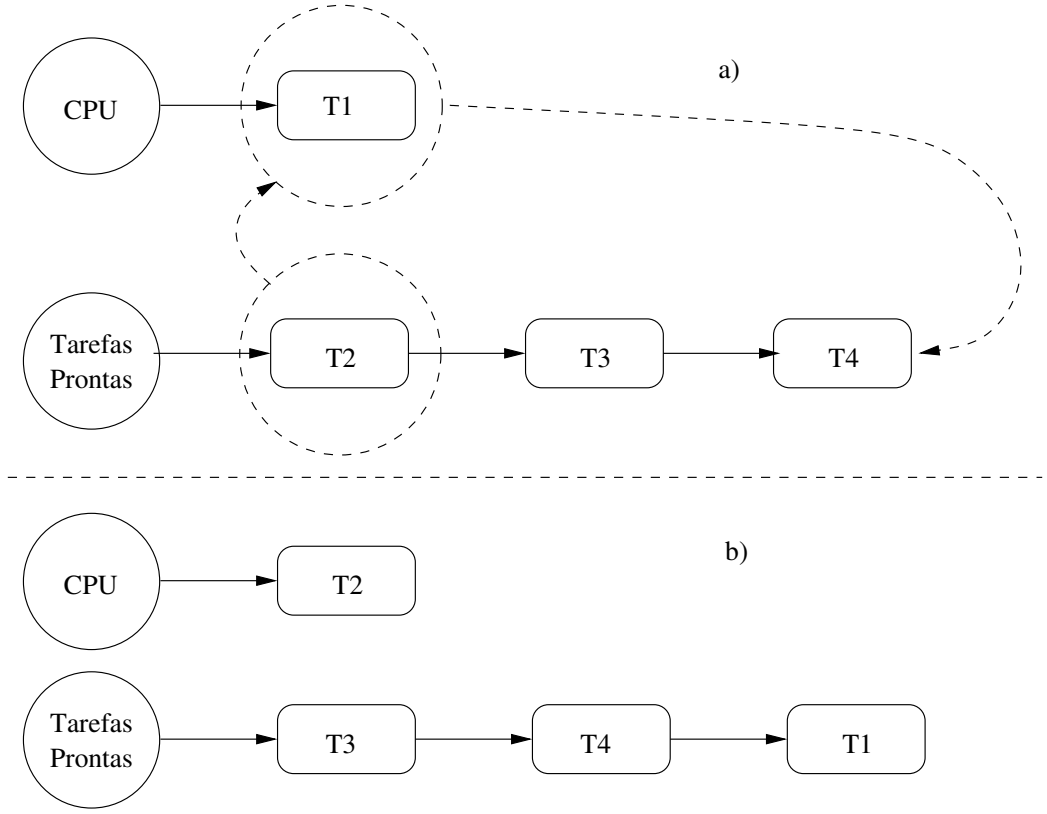
• As regras que determinam a ordem de execução das tarefas constituem a política de escalonamento.

• O escalonador é activado por uma interrupção de hardware (relógio ou E/S) ou indirectamente pelas tarefas (terminus, pedido de suspensão, etc).

• A troca de processos é feita de acordo com a política escolhida para o escalonador por em prática. → Várias hipóteses:

- "round-robin"
- "event-driven"
- prioridades
- "earliest deadlines"
- outras...

Escalonamento Round-Robin



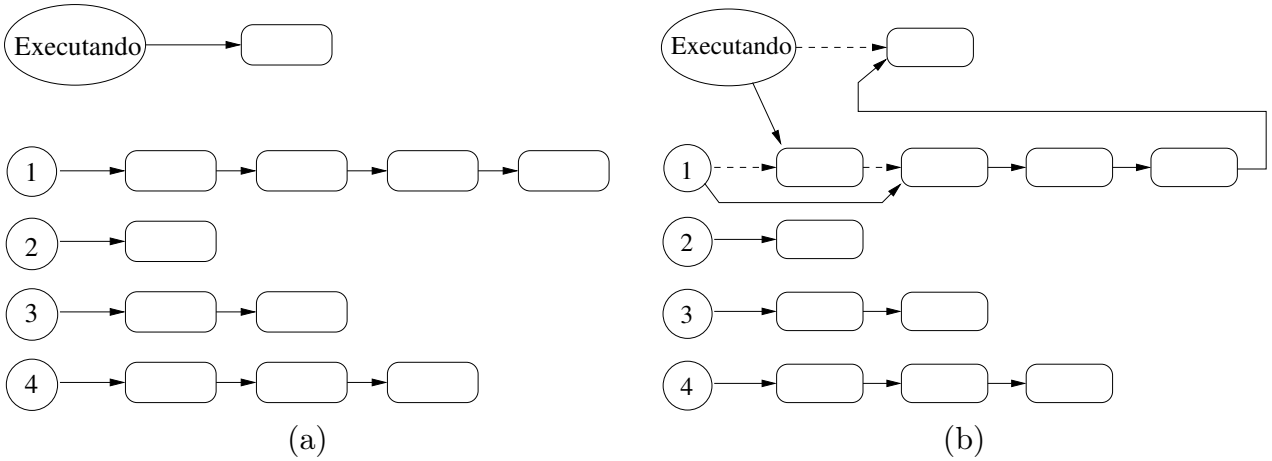
• A sequência de execução das tarefas desenvolve-se de forma cíclica e sequencial entre as várias tarefas.

• Quando estes sistemas utilizam partilha de tempo, cada tarefa recebe um *quantum* de tempo chamado *time slice* ou fatia de tempo.

• Se após esgotar a sua fatia de tempo a tarefa não tiver terminado, então o seu estado/contexto é guardado, e a tarefa é colocada no fim da lista de execução.

• Depois é restaurado o contexto da tarefa seguinte para que esta continue a execução como se não tivesse sido interrompida.

Escalonamento Baseado em Prioridades



• Neste tipo de sistemas, a cada tarefa é atribuído um nível de prioridade.

• O escalonamento é feito de forma a que a tarefa em execução em cada instante é a que tiver a prioridade mais elevada.

• As prioridades são atribuídas a cada tarefa de acordo com a urgência das operações que estão sob a sua responsabilidade.

• Quando um sistema é colocado em funcionamento, as tarefas são distribuídas por um conjunto de listas ligadas de acordo com a sua prioridade.

• Todas as tarefas que se encontram nestas listas estão à espera de uma oportunidade de utilizar o processador.

• As tarefas que se encontram à espera de algum evento são transferidas para listas de espera de eventos, não sendo portanto escalonáveis enquanto lá se encontrarem.

• A tarefa a executar é uma das que tiver prioridade mais elevada. Se houver mais do que uma tarefa no nível de prioridade correspondente, a utilização do processador é partilhada entre as tarefas com essa prioridade.

• Enquanto houver tarefas de prioridade elevada nas listas "prontas" as restantes tarefas não têm acesso ao processador podendo ficar nesse estado indefinidamente.

• Por esta razão a atribuição de prioridades é uma das fases mais importantes no projecto de um sistema de tempo-real → uma má escolha pode tornar o sistema não-funcional.

• Depois de escolhida a nova tarefa a executar é necessário retirar a tarefa que estava a utilizar o processador e substituí-la pela escolhida.

• A tarefa que sai pode ir para uma lista de espera de eventos se fez um pedido que a tal corresponda, ou regressar para o fim da lista "pronta" correspondente à sua prioridade.

• Existem sistemas onde as prioridades das tarefas podem variar de forma dinâmica. Um exemplo de aplicação deste tipo de esquemas é quando se verifica que os recursos da máquina são todos consumidos (incluindo o tempo de processamento) pelas tarefas de prioridade mais elevada. Nestes casos, as tarefas de prioridade mais baixa têm falta de recursos, podendo mesmo nunca chegar a obtê-los. Este fenómeno é designado por *starvation*.

Escalonamento "event-driven"

• A comutação de processos é desencadeada após o aparecimento de certos eventos, sejam eles interrupções externas ou a expiração de períodos de espera.

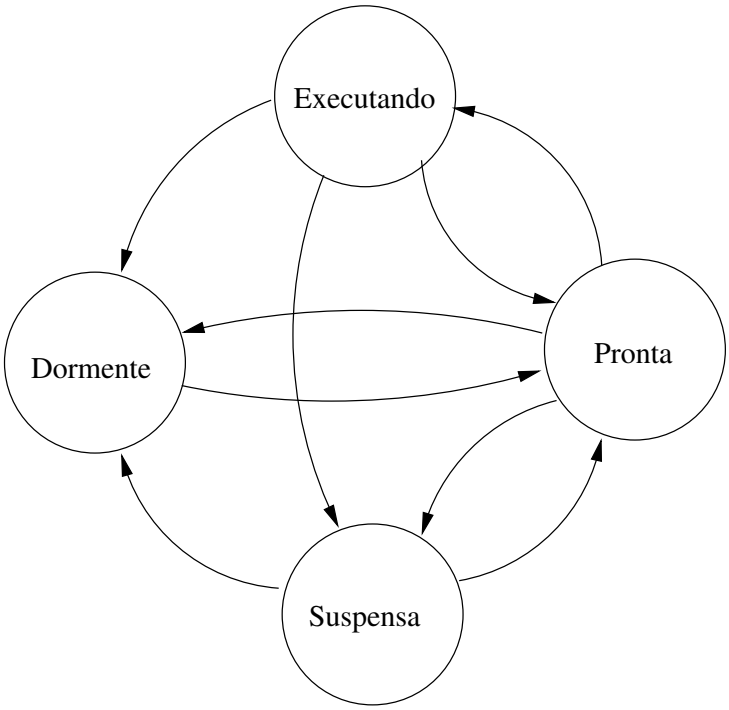
• Cada tarefa tem associado um nível de prioridade, que lhe foi atribuído durante a fase de concepção da aplicação e que é utilizado pelo escalonador para seleccionar a próxima tarefa a executar.

Estados das Tarefas

• Dependendo da sua situação actual cada tarefa pode estar num de vários estados. Os processos passam por diversos estados durante a sua execução.

• Por exemplo, se uma tarefa se encontra à espera de um evento, não necessita de competir pela utilização do processador uma vez que não vai fazer nada até ocorrer o dito evento. Por isso o sistema operativo retira o seu PCB/TCB da lista de processos "prontos" a executar e coloca-o na lista de processos "suspensos".

• Diagrama de transições de estado típico de uma tarefa:



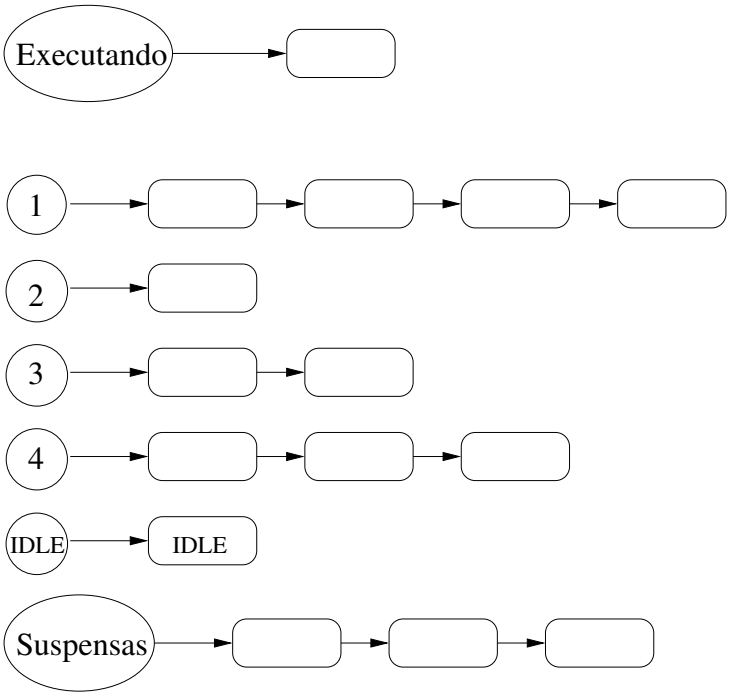
• Alguns dos estados de tarefas que mais comummente são utilizados em sistemas operativos de tempo real para organizar e caracterizar as tarefas são os seguintes:

- **A executar:** está neste estado o processo que está efectivamente a utilizar o processador nesse instante. Em sistemas com mais do que um processador podemos ter mais do que um processo neste estado.

- **Pronto:** Os processos neste estado aguardam a sua vez de utilizar o processador, não tendo mais nada que os impeça de executar que não seja a existência de outros processos em execução ou à sua frente no estado pronto. A troca de processos pode ser feita baseada em prioridades ou tipo round-robin.
- **Suspenso:** Neste estado estão os processos que aguardam qualquer acontecimento/evento, seja uma interrupção externa ou o fim de um período de temporização.
- **Dormente:** Utilizado em alguns sistemas para a criação/terminus dos processos. É neste estado que são fornecidos ou retirados ao processo todos os recursos necessários à sua execução.

Listas de Tarefas

• Os TCBs são normalmente organizados em listas de tarefas de acordo com o estado dessas mesmas tarefas.



Escalonamento de Meta Mais Cedo

• Uma hipótese de escalonamento que é por vezes utilizada é a de executar a tarefa para a qual a meta temporal (*deadline*) irá ocorrer mais cedo no futuro.

• A ideia básica deste método é que depois de cumprir a meta dessa tarefa, se espera ainda ficar com tempo para cumprir com as metas que vêm a seguir.

Escalonamento de Razão Monótona

• Num sistema de tempo real, tarefas diferentes têm periodicidades diferentes.

• Sistemas de razão monótona (*rate monotonic systems*): é o conjunto particular de sistemas preemptivos baseados em interrupções em que é atribuída maior prioridade às tarefas com frequência de execução/activação maior.

• Muito referenciados em estudos sobre sistemas de tempo real.

• **Resultado teórico:** Num sistema de razão monótona nenhuma meta temporal é falhada se a utilização do processador for inferior a 70%.

• No entanto estes sistemas apresentam alguns problemas como p. ex. o da inversão de prioridades. A inversão de prioridades pode acontecer devido a:

- Existirem 2 tarefas onde a mais urgente tem uma frequência de utilização menor. O resultado poderá ser um tempo de resposta demasiado longo.
  Possível solução: colocar o processo crítico num ciclo rápido e usar um contador para controlar a frequência de execução.
- Existindo exclusão mútua no acesso a recursos, uma tarefa de prioridade mais baixa pode bloquear uma outra de prioridade mais elevada.
  Uma solução que apesar de ser bastante referida na literatura é de difícil implementação é o chamado **priority inheritance protocol**. Este protocolo define que se uma tarefa de baixa prioridade bloqueia outra tarefa de elevada prioridade, então a primeira deverá ver a sua prioridade aumentada para o mesmo valor da prioridade da tarefa bloqueada. Isto apenas enquanto durar o bloqueio (i.e. enquanto o recurso não for libertado).
- Em sistemas orientados a objectos uma subclasse pode herdar uma prioridade que não tem a ver com a sua função.
  Nestes casos, uma solução é uma análise cuidada dos atributos de cada classe.

Sistemas Operativos de Tempo-Real

• Existem sistemas operativos de tempo-real que além do mecanismo foreground/background ou de serem multi-tarefa, contêm suporte para interfaces de rede, device drivers, e ferramentas de debugging.

• Muitas vezes estes sistemas utilizam escalonamento round-robin e/ou preemptivo com prioridades.

• Numa boa parte das aplicações os SOs de T-R comerciais são a melhor opção.

• Características:

- Prontos a utilizar
- Bastante testados → serviços robustos.
- São na sua maioria fáceis de utilizar.
- Existem versões para diversos tipos de máquinas o que torna a sua aplicação mais genérica.

• Problemas por vezes apresentados por SOs T-R comerciais → possíveis razões para construir o nosso próprio núcleo de multiprogramação.

- Por vezes → lentidão na resposta a eventos devido a TCBs demasiado genéricos para poderem ser utilizados numa vasta gama de arquitecturas de computadores.
- Tempos de resposta a interrupções ou de comutação de processos: muitas vezes são apresentados os melhores valores ou os valores "médios", quando em aplicações "hard real-time" o que interessa é o pior dos casos.
- Excesso de capacidades para agradar ao maior número de consumidores possível → sobrecarga na execução e/ou aumento do tamanho do código com problemas de armazenamento em ROM se tal for necessário.
- Nalguns casos é necessário pagar direitos por cada unidade produzida (e.g. de um sistema embebido). Pode aumentar os custos de produção dos sistemas.

Comunicação e Sincronização

• Processos concorrentes não são completamente independentes. Na maior parte dos casos coabitam na mesma máquina. Há vantagem de possibilitar a comunicação e/ou sincronização entre eles (e.g. em processos cooperantes).

• Existe um conjunto de recursos que são (ou podem ser) comuns a vários processos (ou a todos):

- Exemplos: periféricos (teclados, ecrãns, impressoras, etc), rotinas e mesmo variáveis.
- Exemplo indesejável numa impressora: um processo imprime um relatório e outro imprime um gráfico ao mesmo tempo.

• Rotinas partilhadas requerem a reentrância do código destas.

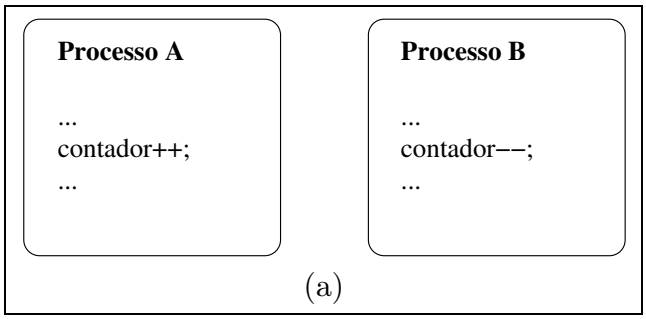
- Variáveis dessas rotinas devem existir na pilha do processo e não como variáveis globais.
- Variáveis globais em rotinas partilhadas → não pode existir mais que um processo a usar a rotina em cada instante.
- O problema aqui está na existência do recurso partilhado "variável global". Mesmo que existissem rotinas separadas a aceder à mesma variável global, poderiam surgir problemas.

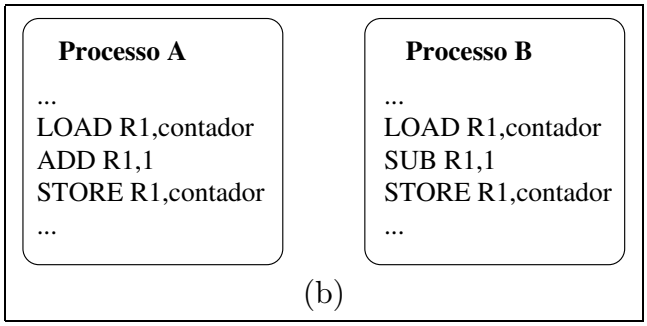
• Variáveis partilhadas em memória partilhada são uma forma natural de comunicação entre processos. No entanto a sua utilização pode levantar alguns problemas.

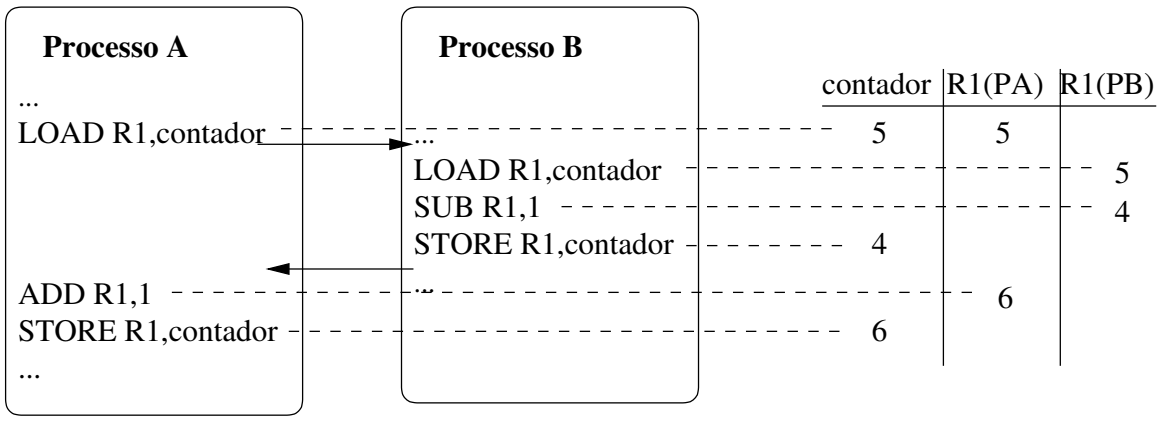
• Que tipo de problemas podem surgir neste tipo de operações tão simples?

• O problema surge porque as instruções de alto nível não são indivisíveis (atómicas).

- Uma instrução de alto nível corresponde a várias instruções máquina.
- Se ocorrer uma interrupção que leve à troca de contexto entre os dois pedaços de código vamos ter problemas/inconsistências.







• É necessário garantir o acesso a recursos (no exemplo acima - a variável) de forma exclusiva.

• **Regiões críticas:** código que efectua acesso a este tipo de recursos partilhados.

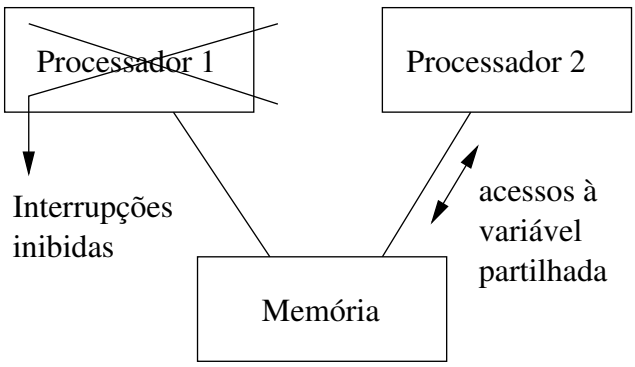
• **Mecanismos de exclusão mútua:** mecanismos que permitem garantir a utilização exclusiva no acesso a recursos.

• **Desactivar interrupções:** seria uma hipótese para implementar exclusão mútua. Mas: é um mecanismo exagerado/excessivo:

→ Não é necessário garantir que mais nenhum processo executa.

→ Apenas pretendemos garantir que mais nenhum processo entra na zona crítica.

→ Só funciona em sistemas uni-processador.



• Poderia-se pensar que uma variável de controlo de acesso ("flag") permitiria a implementação de exclusão mútua. Mas esse mecanismo não irá resolver o problema porque a variável de controlo de acesso não seria mais que uma variável partilhada.

```
Processo A              Processo B
...                     ...
while(ocupado);         while(ocupado);
ocupado=1;              ocupado=1;
zona_critica();         zona_critica();
ocupado=0;              ocupado=0;
...                     ...
```

Semáforos

• Mecanismo proposto por Dijkstra para implementar exclusão mútua.

• Recebeu muita atenção e está presente em muitos sistemas operativos comerciais e experimentais.

• A implementação de semáforos é feita à custa de duas primitivas: SIGNAL e WAIT. (Os nomes originais usados por Dijkstra são P e V respectivamente.)

• Primitiva WAIT(s): Se o valor do semáforo "s" for 0 (zero) então espera. Se/quando o valor for maior que 0 (zero) decrementa-o e prossegue a execução. A operação de teste e decrementação terá que ser indivisível.

• Primitiva SIGNAL(s): Incrementa o valor do semáforo "s" de forma indivisível.

• Implementação de semáforos:

– Inibição/desinibição de interrupções durante o curto espaço de tempo de acesso ao semáforo;

– Já vimos que este processo tem desvantagens em sistemas multi-processadores.

```
...
desactivar interrupções;
aceder à região crítica;
reactivar interrupções;
...
```

– Utilização de instruções de "test-and-set" → muitos processadores possuem este tipo de instruções que têm a vantagem de serem implementadas para poderem ser utilizadas para resolver o problema em sistemas multi-processadores.

• Semáforos para implementar exclusão mútua:

• WAIT(sem) antes da entrada na região crítica.

• SIGNAL(sem) após a saída da região crítica.

Processo A

```
...
Wait(sem);
critica();
Signal(sem);
...
```

Processo B

```
...
Wait(sem);
critica();
Signal(sem);
...
```

• Os processos têm que ser cooperantes na utilização dos semáforos para implementação de exclusão mútua: se um processo não usar semáforos pode entrar na região crítica mesmo que outro, que tendo respeitado os semáforos, já lá esteja dentro.

• Semáforos podem também ser úteis para sincronização entre dois processos.
• Sincronização assimétrica:

Processo A

```
CICLO
  Aquisição de dados;
  Signal(pronto);
  ...
FIM CICLO
```

Processo B

```
CICLO
  Wait(pronto);
  Calcula FFT;
  ...
FIM CICLO
```

• Sincronização simétrica:

Processo A

```
CICLO
  Signal(pronto1);
  Wait(pronto2);
  ...
FIM CICLO
```

Processo B

```
CICLO
  Wait(pronto1);
  Signal(pronto2);
  ...
FIM CICLO
```

• Semáforos são um mecanismo simples para implementar exclusão mútua e/ou sincronização entre processos.

• A utilização de semáforos requer forte disciplina de programação.

→ Uma primitiva mal colocada ou esquecida pode ser difícil de detectar e pode levar a resultados desastrosos.

→ Pode-se não conseguir assegurar exclusão mútua e pode ocorrer impasse em situações raras mas críticas → leva a dificuldades de teste e descoberta de erros.

Impasse ("Deadlock")

• A má utilização de mecanismos de exclusão mútua pode levar à ocorrência de (situações de) impasse.

• Exemplo:

```
Processo A          Processo B
...                 ...
Wait(S1);           Wait(S2);
Wait(S2);           Wait(S1);
...                 ...
Signal(S2);         Signal(S1);
Signal(S1);         Signal(S2);
...                 ...
```

• Condições que se verificadas em simultâneo definem a existência d1 impasse:

1. Exclusão mútua: cada recurso só pode ser utilizado por um número limitado de processos em cada instante.
2. Obtenção e espera: processos que detêm alguns recursos encontram-se à espera de conseguir acesso a outro(s) recurso(s).
3. Não haver preempção: os processos não podem ser obrigados a libertar recursos já adquiridos.
4. Espera circular: existe uma cadeia circular de processos em que cada processo detém pelo menos um recurso requerido pelo processo seguinte.

Prevenção de Impasses

• Solução para prevenir a ocorrência de situações de impasse passa por fazer com que uma das condições acima descritas não se verifique.

• Vejamos as consequências de negarmos cada uma das condições:

• Exclusão mútua: normalmente impossível pois a maior parte dos recursos só pode ser utilizada por um processo de cada vez.

• Obtenção e espera: requeria que todos os processos pré-reservassem todos os recursos de uma só vez, antes de realmente os utilizar, e se não fosse possível fazer a atribuição de um recurso, todos os outros seriam libertados.

→Principal desvantagem: os processos requisitam recursos que em alguns casos só muito tempo depois (dependendo dos casos: horas, dias, semanas, ...) iriam efectivamente ser utilizados, ficando os outros processos impossibilitados de os utilizar, mesmo aqueles que necessitam apenas de uma utilização rápida e imediata.

• Não haver preempção: contrariar esta regra poderia ser desastroso pois poderia implicar retirar a um processo um recurso num estado inconsistente (e.g. uma impressora com uma página meio impressa).

• Espera circular: uma forma de contrariar esta condição seria obrigar os processos a pedir os recursos por uma ordem pré-definida: recursos tipo A, depois B, etc.
→ Desvantagem: Possibilidade de requisitar recursos antes da sua utilização efectiva.

$\Longrightarrow$ Prevenção contra a ocorrência de impasses deve ser feita durante a fase de projecto do sistema.

• Outras formas de lidar com impasses:

  – Evitar a ocorrência: analisando os pedidos dos vários processos.

  – Detecção e recuperação: Verificar se existem impasses e tentar resolvê-los.

• Os dois métodos anteriores para lidar com impasses seriam implementados durante o funcionamento do sistema (e.g. pelo sistema de operação).

Problemas Clássicos de Comunicação/Sincronização Entre Processos

- Problema dos produtores/consumidores.
- Problema dos leitores/escritores.
- Problema dos produtores/consumidores
  - Existe um ou mais produtores gerando dados e colocando-os num buffer.
  - Existe um único consumidor que retira items do buffer um de cada vez.
  - Apenas um agente (produtor ou consumidor) pode aceder ao buffer de cada vez.
- Exemplo de solução → produtores consumidores com buffer limitado:

```
program boundedbuffer;
const   sizeofbuffer=...;
var     s: semaphore (:=1); (* mutual exclusion *)
        n: semaphore (:=0); (* new item *)
        e: semaphore (:=sizeofbuffer); (* buffer not full *)
procedure producer;
begin
  repeat
    produce;
    wait(e);
    wait(s);
    append;
    signal(s);
    signal(n);
  forever
end;
procedure consumer;
begin
  repeat
    wait(n);
    wait(s);
    take;
    signal(s);
    signal(e);
    consume;
  forever
end;
begin (* main program *)
  parbegin
    producer; consumer;
  parend
end;
```

• Problema dos leitores/escritores

  – Existe uma área de dados partilhada por um número de processos.

  – Existe um número de processos que apenas lê da área de dados.

  – Existe um número de processos que apenas escreve na área de dados.

  – Qualquer número de leitores pode simultaneamente ler dados do ficheiro.

  – Apenas um único escritor de cada vez pode escrever nos dados.

  – Se um escritor está a escrever os dados, nenhum leitor os pode ler.

Monitores

• São módulos de software constituídos por:

  1. Um ou mais procedimentos.
  2. Uma sequência de inicialização.
  3. Dados locais.
  4. Variáveis de condição.

• Características mais importantes de um monitor:

  1. Dados locais são acessíveis apenas pelos procedimentos do monitor e não por qualquer procedimento.
  2. Um processo entra num monitor através da invocação de um dos seus procedimentos.
  3. Apenas um processo pode estar a executar dentro de um monitor a cada instante. Qualquer outro processo que tenha invocado o monitor é suspenso à espera que o monitor fique disponível.

• Permite implementar exclusão mútua no acesso aos dados que se definam como locais ao monitor.

• Para ser útil → monitor deve incluir ferramentas de sincronização.

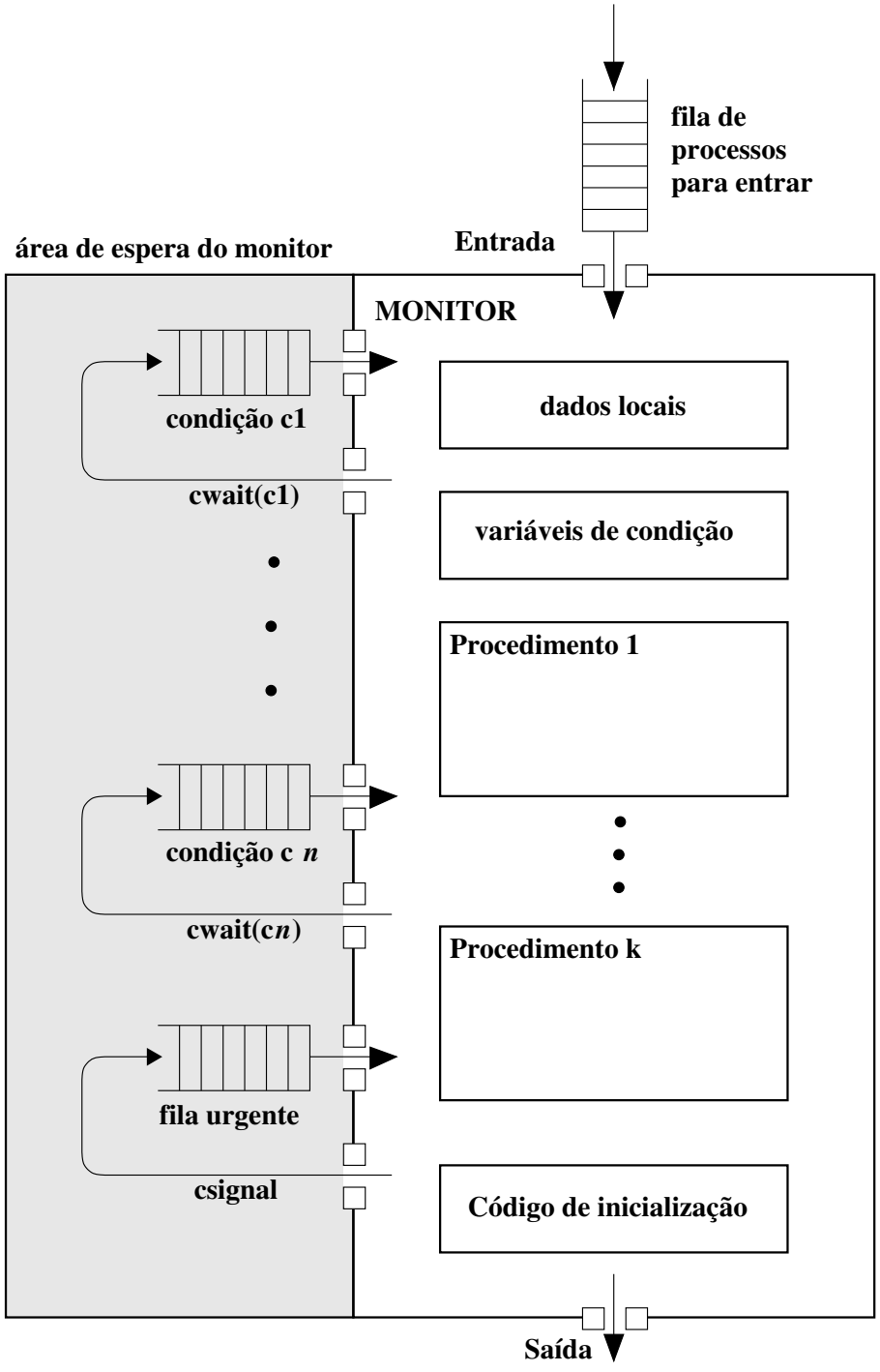
• Monitor suporta sincronização através do uso de variáveis de condição que são contidas dentro do monitor e são acessíveis apenas dentro deste.

- Duas primitivas que operam sobre variáveis de condição:

  cwait(c): suspender o processo que chamou "pendurando-o" na condição "c". O monitor fica disponível para utilização por outro processo.

  csignal(c): continuar a execução de algum processo suspendido após um "cwait" na mesma condição "c". Se existirem diversos processos à espera, escolher um deles (e suspender o que enviou o csignal); se não existir nenhum, então não faz nada (o sinal é perdido e não tem efeito).

- Estrutura de um monitor:



• Nota: as operações com cwait/csignal de um monitor são diferentes das wait/signal de um semáforo. Se uma tarefa a executar dentro de um monitor envia um csignal e nenhuma tarefa está à espera desse sinal, então o sinal é perdido.

• Fila urgente (possibilidade): para os processos que enviaram um sinal mas ainda não terminaram o procedimento do monitor, não terem que ir para a fila de entrada $\Longrightarrow$ passa a ter prioridade sobre os processos na fila de entrada. (Nota: csignal $\Rightarrow$ processo que sinalizou é suspenso e a execução passa para o sinalizado).

Mensagens

• Uma das formas mais naturais de comunicar.
• Conjunto mínimo de operações:

`envia(destino,mensagem);`	ou	`send(destination,message);`
`recebe(origem,mensagem);`	ou	`receive(source,message);`

• Características de projecto:

1. Sincronização.
2. Endereçamento.
3. Formato.
4. Disciplina da fila.

1. Sincronização:

• Enviar/envio: bloqueante ou não bloqueante.

• Receber/recepção: bloqueante, não-bloqueante, teste de recepção.

• Envio bloqueante: o processo que envia bloqueia até a mensagem ser recebida pelo receptor.

• Envio não-bloqueante: o processo que envia continua (regressa da primitiva "envia") imediatamente não se efectuando a espera pela recepção da mensagem.

• Recepção bloqueante: o processo que recebe bloqueia até que "chegue" uma mensagem com características desejadas (e.g. uma certa origem) para ser fornecida ao processo receptor.

• Recepção não-bloqueante: a primitiva "recebe" regressa imediatamente devolvendo a mensagem, se ela existir, ou um código informando da presente inexistência para recepção de qualquer mensagem com as características desejadas.

• Teste de recepção: o processo que recebe pode usar uma primitiva para testar se já está disponível para recepção alguma mensagem com certas características desejadas.

• Troca assíncrona de mensagens: (envio não-bloqueante e recepção bloqueante ou não)

  - O processo emissor, após efectuar o envio da mensagem continua normalmente.
  - Este método requer a utilização de buffers para armazenar mensagens em envio ou ainda não lidas pelo receptor.

• Troca síncrona de mensagens: (envio e recepção bloqueantes)

  - Este método é designado frequentemente de *rendezvouz*.
  - O emissor bloqueia até a mensagem ser (efectivamente) entregue/recebida (quando o receptor executa a primitiva de recepção).
  - Neste caso o envio e a recepção são concretizados em simultâneo pelo que não é requerida a utilização de um buffer.

• Invocação remota:

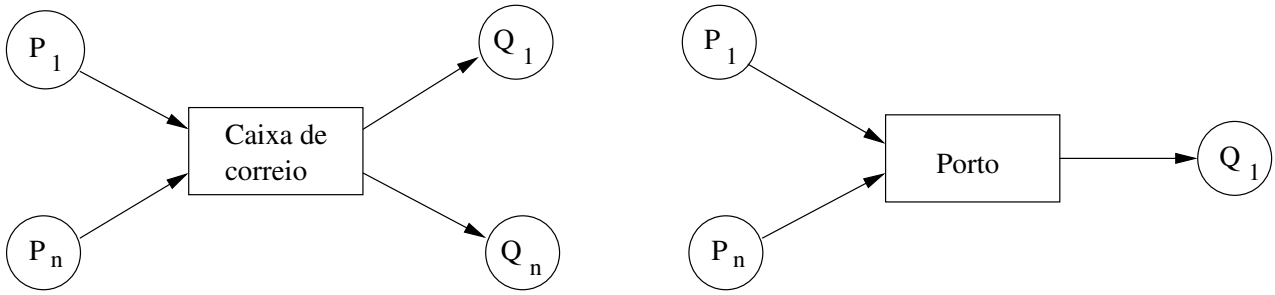
  - Baseia-se no conceito de "chamada de procedimento remoto".
  - Utilizada em sistemas distribuídos.
  - O emissor continua a execução após o destinatário enviar uma resposta (que pode ser simplesmente uma confirmação ou conter ainda outros dados).

2. Endereçamento das mensagens: directo ou indirecto.

• Endereçamento directo: o processo que envia fornece um identificador específico do processo de recepção.

- Processo que recebe:
  - Directo explícito: processo que recebe designa qual o processo do qual quer receber.
  - Directo implícito: processo que recebe deixa em aberto a origem da mensagem, mas o parâmetro "origem" contém no regresso o identificador do processo que realmente enviou a mensagem devolvida pela primitiva de recepção.

• Endereçamento indirecto: a mensagem é enviada para (e lida de) uma "caixa postal" (ou caixa de correio) - "mailbox":



• Vantagens:

- Não obriga à identificação precisa do processo destinatário mas sim de uma caixa de correio que pode receber um nome (ou endereço) pré-definido.
- Possibilidade de permitir a comunicação de várias formas:
  - Um para um.
  - Muitos para muitos.
  - Muitos para um (clientes/servidor...; neste caso: caixas de correio frequentemente designadas por portos).
  - Um para muitos (broadcast ou multicast).

• Esta diversidade de tipos de comunicação é possível pois uma mailbox pode ser partilhada por vários processos; e o processo que envia apenas necessita de saber o "nome" da mailbox e não o nome dos destinatários, o que nalguns casos seria praticamente impossível.

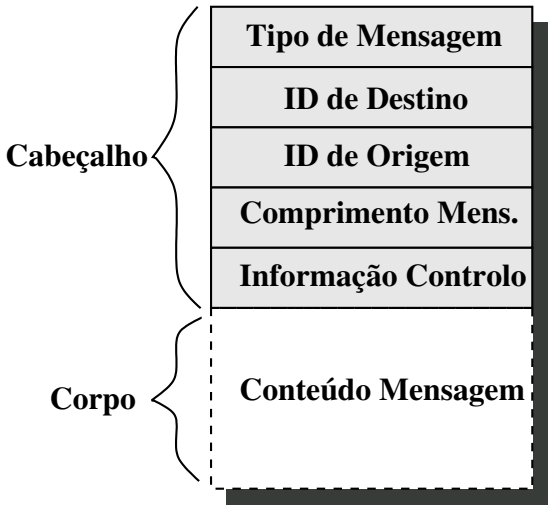
• Associação processos/mailboxes pode ser

- Estática: mailboxes (e.g. portos) associados permanentemente a um processo particular. Também em relações um-para-um a associação é normalmente estática.
- Dinâmica: Quando existem muitos processos a enviar, a associação "sender/mailbox" pode ser dinâmica. Para tal são muitas vezes usadas primitivas tais como "connect" e "disconnect".

3. Tipo/conteúdo/formato das mensagens

• As mensagens são na maior parte dos casos estruturas de dados definidas pelo utilizador ou pelo programador do sistema.

• Conteúdo e comprimento das mensagens: pode ser fixo ou variável.



• O receptor necessita de conhecer o formato do conteúdo dos dados para os poder interpretar → ...

• Exemplo: o conjunto de 4 bytes "0x53, 0x54, 0x52, 0x00" pode ser interpretado de muitas formas:

→ Um número inteiro de 32 bits na forma "little endian" (1398034944).

→ Um número inteiro de 32 bits na forma "big endian" (5995539).

→ Uma cadeia de caracteres terminada pelo caracter nulo (STR).

→ outras...

• Várias formas de interpretar um conjunto de dados → emissor e receptor devem usar a mesma representação ou, se isso for impossível, fazer a conversão adequada.

• Na comunicação entre máquinas de arquitecturas diferentes pode ser necessário fazer a conversão do formato dos dados para poderem ser manipulados pelos processos.

• Por exemplo, diferentes ordens dos bytes para representar inteiros → hipótese: adoptar ordem padrão e as máquinas que usam ordens diferentes devem efectuar a conversão dos dados após as "recepções" e antes de "envios" de mensagens.

• Mensagens podem ter o seu formato definido pelo sistema ou pelo utilizador.

• Num espaço de endereçamento comum (memória partilhada) as mensagens podem ser passadas por:

- Referência: as primitivas de envio e recepção usam um ponteiro para a zona de memória onde se encontra a mensagem; Vantagens: →
  - Mais rápida pois só o ponteiro é efectivamente enviado; depois o ponteiro é usado pelo receptor para localizar a mensagem.
  - Economiza memória pois existe apenas uma cópia de cada mensagem e não uma cópia por processo.
  - Desvantagem: o processo que envia não pode reutilizar a memória da mensagem pois corre o risco de o receptor ainda não a ter lido.
- Valor: copiando os dados.

• Passagem por referência → deve ser feita de forma síncrona → para evitar que o emissor re-escreva uma nova mensagem por cima da anterior antes de o receptor a ler.

• Buffers: são parcelas de memória utilizadas para o transporte de mensagens entre o emissor e o receptor.

• Os buffers permitem que os processos funcionem de forma assíncrona, pois o emissor copia a mensagem para um buffer e este é depois colocado na mailbox do receptor.

• Problemas e limitações no uso de buffers:

→ O tamanho dos buffers é limitado e muitas vezes fixo.

→ Se um processo envia uma mensagem para uma mailbox cheia pode ficar bloqueado o que pode levar à inversão de prioridades.

→ A utilização de buffers requer mais memória que poderá ser escassa em sistemas embebidos.

• Problema da inversão de prioridades: um processo prioritário fica bloqueado porque um processo de prioridade mais baixa tem a mailbox cheia → possível resolução com uma primitiva de envio por tentativa, i.e. a primitiva só envia se existir espaço na mailbox, senão devolve um código de erro.

→ é possível implementação de uma primitiva de recepção não bloqueante com o mesmo funcionamento perante um buffer vazio.

• Problema da memória: tamanho de buffers limitado → possível solução pela definição de um protocolo de fragmentação/defragmentação de mensagens.

→ Mas se existe muita limitação de memória, o melhor é não usar buffers na comunicação entre processos.

• Podem ser usadas mensagens para sincronização e comunicação entre tarefas.

• Exemplo da utilização de mensagens como forma de sincronização e comunicação de dados entre processos:

Processo A
CICLO
Aquisição de dados;
envia(ProcessoB,dados);
...
FIMCICLO

Processo B
CICLO
recebe(ProcessoA,dados);
Calcula FFT;
...
FIMCICLO

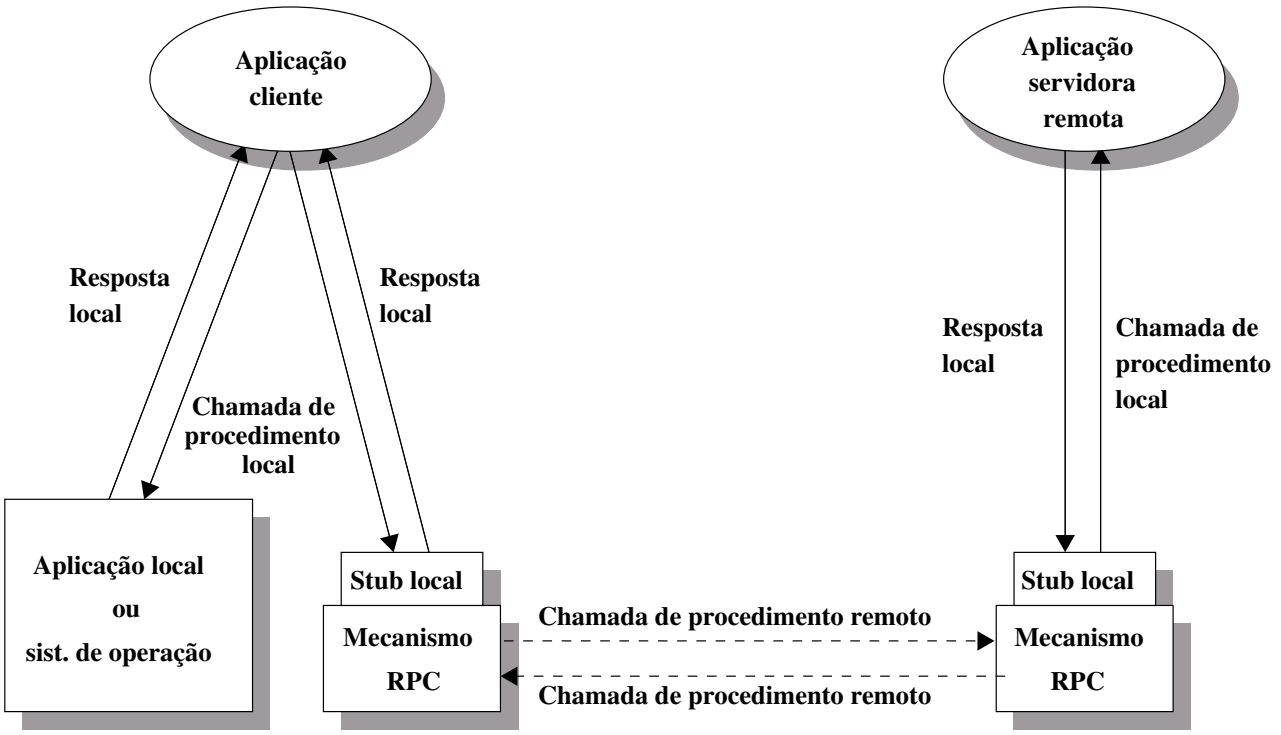
• As mensagens são uma forma natural de sincronização/comunicação entre sistemas distribuídos. Nestes sistemas existem (normalmente muitos) pares de entidades de processamento que não possuem memória partilhada entre si.

Chamada de Procedimento Remoto

• "Remote Procedure Call (RPC)".

• Permitir que programas em diferentes máquinas interactuem usando a semântica de chamada/regresso de procedimentos, tal como se os dois programas estivessem a correr na mesma máquina.

• A chamada ao procedimento é usada para aceder a serviços remotos:



• O mecanismo RPC pode ser visto como um refinamento de passagem bloqueante de mensagens.

• O programa que chama o RPC faz uma chamada a procedimento normal com parâmetros na sua máquina. Por exemplo:

`CALL P(X,Y)`, onde, `P` = nome do procedimento,
`X` = parâmetros de entrada,
`Y` = parâmetros devolvidos.

- Vantagens:

  1. Chamada de procedimentos → abstracção largamente aceite, usada, e compreendida.

  2. RPC permite que interfaces remotos sejam especificados como um conjunto de operações com certos nomes e envolvendo certos tipos de dados. Assim, o interface pode ser claramente documentado e pode-se verificar estaticamente a ocorrência de erros de tipo de dados em programas distribuídos.

  3. É especificado um interface padrão e precisamente definido ⇒

  ⇒ o código de comunicação de uma aplicação pode ser gerado automaticamente.

  ⇒ podem-se escrever módulos cliente e servidor que podem ser transferidos entre computadores e sistemas de operação com pouca ou nenhuma modificação ou recodificação.

- **Passagem de parâmetros em RPC:** por valor ou por referência.

  - Passagem por valor: são passados os próprios valores dos parâmetros. Os parâmetros são simplesmente copiados para uma mensagem e enviados ao sistema remoto.

  - Passagem por referência: são passados ponteiros para uma localização que contém os valores. Em sistemas distribuídos é mais difícil implementar a chamada por referência. É necessária a existência de ponteiros válidos em todo o sistema distribuído. A dificuldade de implementar esta capacidade pode não valer a pena o esforço.

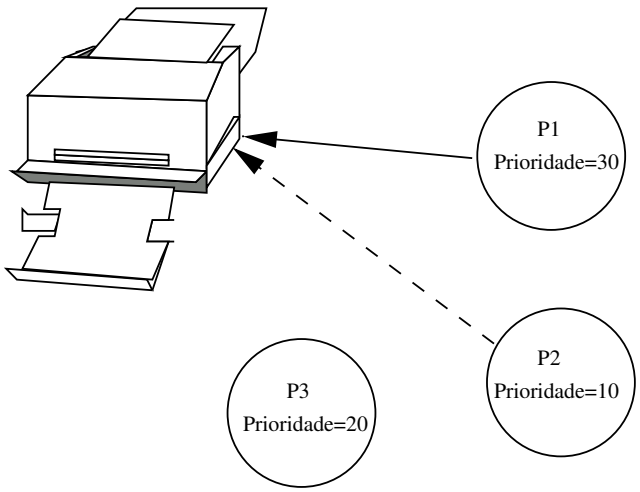
- **Representação de parâmetros em RPC:** quando estão envolvidos diferentes tipos de máquinas, linguagens de programação, ou sistemas de operação pode ser necessário implementar conversão de dados num mecanismo RPC. Isso pode ser feito por exemplo em torno de uma representação padrão.

- **Ligação cliente/servidor em RPC:** persistente ou não persistente.
  - Ligação não-persistente (LNP): a ligação é desencadeada quando se realiza um RPC e logo que sejam devolvidos dados a ligação é desmantelada. Ligação requer informação de estado em ambos os lados ⇒ são usados menos recursos com uma LNP; mas RPC frequentes ⇒ muito "overhead" no estabelecimento de ligações.
  - Ligação persistente: uma ligação é estabelecida através de um RPC e depois a ligação mantém-se ao longo da utilização de RPCs repetidos no futuro.

- **RPCs Síncronos versus Assíncronos:** Síncronos → como uma chamada a uma sub-rotina: o processo que chama espera até que lhe sejam devolvidos valores de resposta; Assíncronos → processo que chama não precisa de esperar pela devolução dos resultados, podendo continuar para só mais tarde eventualmente solicitar os resultados. Possibilita aproveitar melhor o paralelismo num sistema distribuído.

Inversão de Prioridades

- Exemplo de inversão de prioridades devido a contenção no acesso a um recurso:



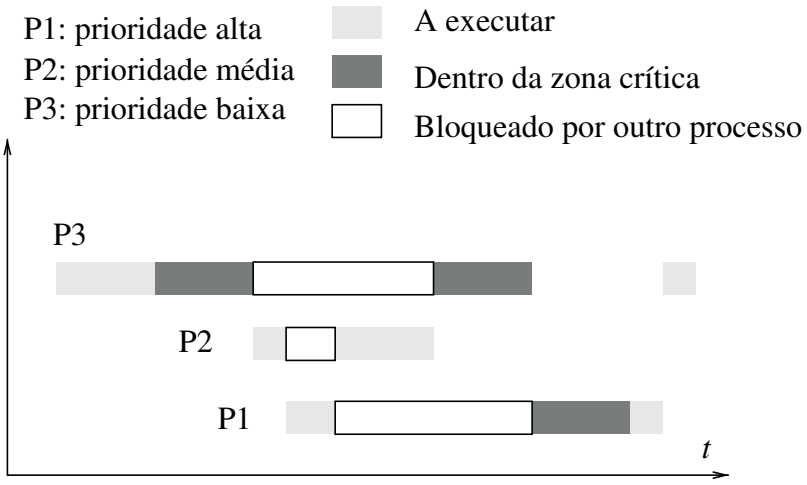
• Outro exemplo: um processo P_3 está a usar um recurso com exclusão mútua (e.g. dentro de uma região crítica delimitada por um semáforo).

• Qualquer outro processo P_1, qualquer que seja a sua prioridade ($P_1 > P_3$, $P_1 = P_3$, ou $P_1 < P_3$) que pretenda usar o recurso terá que esperar até que P_3 o liberte.

• No entanto, se aparecer qualquer outro processo com prioridade P_2, tal que $P_1 < P_2 < P_3$ (P_1 é o mais prioritário), e que fique pronto a executar vai (P_2) "retirar o processador" ao processo P_1 devido ao mecanismo de inversão de prioridades.

• O processo P_2 vai desta forma atrasar a execução do processo P_1 apesar de ter prioridade inferior ($P_1 < P_2$) e de não ter qualquer envolvimento com o recurso esperado por P_1. O processo P_2, ao atrasar P_3, atrasa indirectamente P_1 pois P_1 está bloqueado num recurso que P_3 possui.

• O exemplo graficamente:



• Prioridade mais alta → numericamente mais baixa.

• A inversão de prioridades pode acontecer para qualquer número de processos podendo bloquear indefinidamente o processo de prioridade mais alta.

• Também pode ocorrer inversão de prioridades quando se faz a passagem síncrona de mensagens ("rendezvous").

• **Problema da Inversão de Prioridades: Soluções**

• Uma solução seria: não permitir a interrupção de tarefas dentro de secções críticas. → Só aceitável se o tempo de execução dentro d1 secção crítica for muito pequeno, uma vez que pode levar ao bloqueamento de tarefas de elevada prioridade.

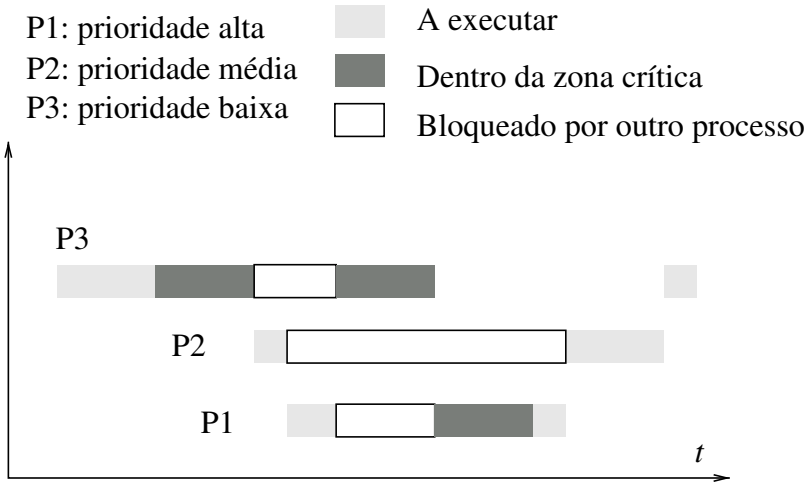
• Outra solução →

Protocolos de Herança de Prioridades (PHP)

• Quando uma tarefa T bloqueia outras tarefas de prioridade superior, então T vê o seu nível de prioridade subir para o nível de prioridade mais elevado das tarefas bloqueadas (por T) até sair da secção crítica, altura em que a sua prioridade é restabelecida no valor original.

$\Longrightarrow$ O processo que entrou primeiro na zona crítica executa tão rapidamente quanto possível por forma a não atrasar mais do que é indispensável o processo de prioridade mais elevada

• Aplicando o PHP ao exemplo anterior temos:



• O processo P_1 fica bloqueado por pretender entrar na secção crítica onde já está o processo P_3.

• A prioridade do processo P_3 é elevada para o mesmo valor de P_1.

• Assim o processo P_3 não é bloqueado por P_2, terminando a execução da zona crítica mais rapidamente.

• Quando P_3 sai da zona crítica, a sua prioridade regressa ao valor original, pelo que o processo P_1 o interrompe e executa até ao fim.

• Após P_1 terminar é executado o processo P_2 até ao fim.

• Quando P_2 termina é então realizado o resto da execução de P_3.

• A conclusão dos três processos obedece assim à ordem estabelecida pelas prioridades.

Definição do Protocolo Básico de Herança de Prioridades (PBHP)

1. A tarefa T com prioridade mais alta do conjunto de tarefas "prontas a executar" é colocada em execução. Antes de T entrar numa secção crítica tem de primeiro fechar um semáforo que protege a secção crítica. Se o semáforo já estiver fechado, o acesso é negado e a tarefa é bloqueada. Caso contrário, a tarefa prossegue com a execução da secção crítica. Quando a tarefa abandona a secção crítica, o semáforo correspondente é aberto e se houver alguma tarefa de prioridade mais elevada bloqueada por T, essa tarefa é acordada.

2. A tarefa T utiliza a prioridade que lhe foi atribuída a menos que se encontre numa secção crítica e, por essa razão, esteja a bloquear tarefas de prioridade mais elevada. No caso de haver tarefas de prioridade mais elevada bloqueadas por esta tarefa, T herda a prioridade mais elevada das tarefas por ela bloqueadas. Quando T abandona a secção crítica regressa à prioridade que tinha inicialmente.

3. A herança de prioridades goza da propriedade transitiva. I.e. considerando três tarefas T_1, T_2 e T_3 de prioridades decrescentes; se T_3 bloqueia T_2 e T_2 bloqueia T_1, então T_3 herda a prioridade de T_1 através de T_2. Por outro lado, as operações de herança e restauro de prioridades devem ser indivisíveis.

4. Uma tarefa T pode retirar a execução a T_l se T não estiver bloqueada e a sua prioridade for superior a T_l (atribuída ou herdada).

• A utilização destes protocolos leva a duas formas de blocagem:

- Blocagem directa: ocorre quando um processo de alta prioridade é bloqueado quando tenta trancar um semáforo já trancado (e.g. por uma tarefa de baixa prioridade).

- "Push-through": ocorre quando um processo de prioridade média é bloqueado por um processo de prioridade baixa que herdou a prioridade de um processo de prioridade mais elevada.

Características do Protocolo Básico de Herança de Prioridades

• Possibilidade de prever o tempo durante o qual uma tarefa de prioridade elevada pode ficar bloqueada por outra de prioridade inferior.

1. Uma tarefa T_a pode estar bloqueada por uma tarefa de prioridade inferior T_b, no máximo durante a execução de uma secção crítica, independentemente do número de semáforos que estas tarefas partilham.

A prova disto é obtida verificando simplesmente que uma tarefa de prioridade baixa só pode bloquear uma de prioridade superior se a primeira já se encontrar a executar uma zona crítica partilhada por ambas. Assim que a primeira tarefa sai da zona crítica, pode ser-lhe retirado o processador pela tarefa de prioridade mais elevada, e a partir daqui T_b não pode voltar a bloquear T_a.

2. Da mesma forma, se para uma tarefa T_a existirem n tarefas de prioridade inferior, T_a pode ser bloqueada no máximo durante a duração de uma secção crítica em cada um dos conjuntos com quem partilha semáforos.

3. Um semáforo S pode bloquear uma tarefa T através de "push-through" apenas se S é acedido por uma tarefa de prioridade inferior a T e por outra tarefa de prioridade igual ou superior a T.

Se uma tarefa de prioridade baixa T_b acede a um semáforo S, e se S não é acedido por uma tarefa de prioridade igual ou superior a T_a, então a zona crítica de T_b não pode herdar uma prioridade igual ou superior à de T_a e por isso é interrompida por T_a.

4. Uma tarefa T pode ser bloqueada no máximo numa secção crítica para cada semáforo.

Como já vimos, uma tarefa T_b pode bloquear outra de maior prioridade T_a, apenas se a primeira se encontrar a executar numa secção crítica comum. Assim que T_b sair da secção crítica, regressa à prioridade que tinha à entrada e por isso é imediatamente interrompida por T_a. Por isso, apenas uma secção crítica das que são comuns pode bloquear T_a.

• Característica adicional consequência das anteriores:

$\Longrightarrow$ Com o protocolo básico de herança de prioridades, se existirem m semáforos que podem bloquear T, então T pode ser bloqueado no máximo m vezes (por tarefas de prioridade inferior).

$\rightarrow$ É possível determinar o pior caso de bloqueio que uma tarefa pode experimentar, desde que não existam impasses ("deadlocks").

Problemas com o PBHP

• Apesar de permitir resolver o problema da inversão de prioridades, o PBHP apresenta as seguintes deficiências:

1. Não previne por si só a ocorrência de impasses ("deadlocks").

2. A duração do bloqueio de uma tarefa, se bem que limitada, pode ser consideravelmente longa por a tarefa encontrar bloqueio em várias secções críticas.

• Exemplo de impasse: dois processos P_1 e P_2 que usam dois recursos S_1 e S_2; com prioridade(P_1) > prioridade(P_2);

- P_2: wait(S_1) $\Rightarrow$ P_2 continua (S_1 "aberto").
- P_1: activado $\Rightarrow$ P_2 suspende (prioridade(P_1) > prioridade(P_2)).
- P_1: wait(S_2) $\Rightarrow$ P_1 continua (S_2 "aberto").
- P_1: wait(S_1) $\Rightarrow$ P_1 bloqueia (S_1 já trancado por P_2),
  $\Rightarrow$ (prioridade(P_2) $\leftarrow$ prioridade(P_1)).
- P_2: wait(S_2) $\Rightarrow$ P_2 bloqueia (S_2 já trancado por P_1),
  $\Rightarrow$ situação de (inter-) impasse.

$\Longrightarrow$ Emergiu uma situação de Impasse; Interblocagem; "deadlock".

Protocolo de Tecto de Prioridade

• "Priority Ceiling Protocol (PCP)".

• É uma solução para o problema da inversão de prioridades.

• O objectivo deste protocolo é prevenir a criação de impasses ("deadlocks") e blocagens múltiplas.

• Ideia base: fazer com que quando uma tarefa T bloqueia outra tarefa que se encontra dentro de uma secção crítica, e executa a sua própria secção crítica, esta nova secção crítica vai ser executada com uma prioridade superior a todas as prioridades (herdáveis) das secções críticas interrompidas (bloqueadas).

• Se esta condição não puder ser satisfeita, então a tarefa T vê negado o seu acesso à secção crítica e é bloqueada.

• Isto é conseguido começando por atribuir uma prioridade de "tecto" a cada um dos semáforos, a qual é igual à maior das prioridades que pode utilizar esse semáforo.

• É permitido a uma tarefa iniciar uma nova secção crítica se a sua prioridade for superior a todas as prioridades de "tecto" de todos os semáforos presentemente fechados por outras tarefas.

• Prioridade de tecto de um semáforo: é a prioridade da tarefa com maior prioridade que pode fechar esse semáforo.

⇒ A prioridade de tecto de um semáforo S, constitui portanto a prioridade mais elevada que pode ser herdada por uma tarefa durante a execução da secção crítica correspondente.

⇒ Se uma tarefa T fecha o semáforo S, a secção crítica correspondente de T pode herdar no máximo uma prioridade igual à prioridade de tecto de S.

Definição do Protocolo de Tecto de Prioridade

• A tarefa T, que tem a prioridade mais elevada de entre as tarefas prontas a executar, recebe acesso ao processador.
Seja S^* o semáforo com prioridade de tecto mais elevada de todos os semáforos actualmente fechados por outras tarefas que não T.

• Antes de T entrar numa zona crítica, tem de primeiro conseguir acesso fechando um semáforo S que a protege.

→ Se a prioridade de T não é superior à prioridade de tecto de S^*, então T é bloqueada e o fecho do semáforo S é negado. Neste caso, diz-se que T está bloqueada no semáforo S^*.

→ Caso contrário, T fecha o semáforo S e entra na secção crítica.

→ Quando uma tarefa T sai duma sua secção crítica, o semáforo associado é aberto e se existirem processos bloqueados nesse semáforo, será acordado o que tiver a prioridade mais elevada.

• Uma tarefa T utiliza a prioridade que lhe foi atribuída, a menos que esteja numa secção crítica e que, por essa razão, tenha bloqueado tarefas de prioridade superior. Neste caso, T herda a prioridade mais elevada dessas tarefas.

→ Quando T abandona uma secção crítica, retoma a sua prioridade que tinha à entrada da secção crítica.

→ A herança de prioridades é transitiva e as operações de herança e retoma de prioridades devem ser indivisíveis.

• Uma tarefa T, quando não pretende entrar numa secção crítica, pode bloquear outra tarefa T_b se a sua prioridade for superior à prioridade (atribuída ou herdada) de T_b.

• O PCP pode levar a três formas de blocagem →

- Blocagem directa: ...
- "Push-through": ...
- "Ceiling blocking": este tipo de blocagem é necessário para evitar a ocorrência de deadlocks, pertencendo à classe de protocolos péssimistas os quais criam por vezes blocagens desnecessárias → mas garantem a inexistência de impasses.

• Apesar desta nova forma de blocagem,

→ o pior caso de blocagem em relação ao PBHP é bastante melhorado uma vez que com o PCP uma tarefa pode ser bloqueada no máximo durante a maior das secções críticas de prioridade inferior,

→ ao contrário do PBHP onde a duração máxima do bloqueio de um processo é $\min(m, n)$ secções críticas, onde m é o número de processos de prioridade inferior que podem bloquear a tarefa, e n o número de semáforos utilizados por esta.

Sobre a Implementação do PCP

• Terá que existir no sistema uma lista de semáforos trancados, ordenada pelas prioridades de tecto. Cada semáforo trancado terá que armazenar informação sobre a tarefa que o trancou e o seu valor de tecto.

• As chamadas ao sistema (indivisíveis) para trancar ou libertar semáforos mantêm actualizadas as listas de processos e semáforos trancados.

• A função "wait" poderá ainda detectar qualquer tentativa de auto-bloqueio de uma tarefa e neste caso abortar essa tarefa uma vez que isso só poderá ser devido a um erro de programação.

Características do Protocolo de Tecto de Prioridades

• Nota: uma tarefa só pode ser bloqueada por outra de prioridade inferior se a segunda se encontrar a executar uma secção crítica quando a primeira é iniciada.

• Definição: diz-se que há blocagem transitiva se uma tarefa T é bloqueada por T_i que por sua vez é bloqueada por T_j.

Características:

1. Uma tarefa só pode ser bloqueada no primeiro semáforo que requer, pois a partir desse momento, todos os pedidos de wait sobre outros semáforos serão concedidos.
2. Uma tarefa T pode ser bloqueada por uma de prioridade inferior T_b apenas se a prioridade de T não for superior à maior prioridade de tecto de todos os semáforos trancados por T_b quando T é iniciada.
3. Se T_i é interrompida por T_j, a qual entra numa secção crítica, então T_i não pode herdar a prioridade de T_j enquanto T_j não terminar.
4. O protocolo PCP previne a blocagem transitiva.
5. Se T_b for uma tarefa de prioridade inferior à da tarefa T_i, então T_i poderá ser bloqueada por T_b por um tempo máximo igual à duração de uma secção crítica comum.

• Uma das vantagens que advém da utilização de protocolos PCP é a prevenção de deadlocks, porque se uma tarefa tenta aceder a uma zona crítica só o poderá fazer se puder completar a execução sem ficar bloqueada.

• Se uma tarefa T se suspende n vezes durante a sua execução, então esta poderá ser bloqueada no máximo por $(n + 1)$ elementos não necessariamente distintos do conjunto de tarefas que têm secções críticas comuns e cujas prioridades são inferiores à de T.

Protocolo de Tecto Imediato de Prioridades (PTIP, Herança Imediata)

• Tal como no PCP, a cada processo é atribuída uma prioridade estática por defeito (por exemplo, por um método de metas mais cedo).

• O PTIP é um protocolo baseado no anterior, mas cuja diferença consiste em elevar a prioridade de um processo assim que ele tranca um recurso. Um processo tem uma prioridade dinâmica que é o máximo entre a sua própria prioridade estática e os valores de tecto de quaisquer recursos que tenha trancados na sua posse.

• Isto é possível pois cada semáforo/recurso tem estaticamente associado o tecto, que é a máxima prioridade dos processos que o utilizam.

• Como consequência, um processo só será bloqueado durante o início da sua execução. Assim que o processo inicia a execução, todos os recursos por ele utilizados estarão disponíveis, caso contrário teria ficado bloqueado pois alguns processos teriam uma prioridade igual ou superior à sua.

Estimação dos Tempos de Execução dos Programas

• Dado que os sistemas de tempo-real devem cumprir metas temporais, é importante poder estimar de uma forma mais ou menos correcta os tempos de execução dos programas.

• No entanto esta tarefa apresenta-se bastante difícil pois depende de vários factores de difícil modelação:

• Optimização do Código Fonte: o código pode ser escrito de forma mais ou menos optimizada para tirar partido das particularidades da arquitectura a que se destina. Em função disso, o código pode executar mais ou menos rapidamente.

• Compilador: A transformação entre código fonte e código máquina não é unívoca. Por isso, diferentes compiladores geram diferentes códigos que poderão ter diferentes graus de optimização.

• Arquitectura da máquina: há aspectos das arquitecturas das máquinas/sistemas que têm efeitos no tempo de execução dos programas que são muitas vezes difíceis de estimar. Exemplos:

→ O tempo que um bocado de código demora a efectuar a comunicação através de um dispositivo de rede não é determinístico: depende do tipo de rede, da carga da mesma, da carga da máquina remota, etc.

→ Número de registos.

→ Tamanho e velocidade da memória.

→ Utilização de memórias "cache".

→ Número de ciclos por instrução que poderá depender por exemplo da utilização ou não de "pipelining".

• Sistema Operativo: O escalonamento dos processos e a gestão de memória são da responsabilidade do sistema operativo e ambos têm uma forte influência no tempo de execução.

⇒ Estes factores que acabámos de listar têm ainda a particularidade de interagirem entre si de forma mais ou menos complexa, e para poder fazer estimativas dos tempos de execução é necessário modelar estas interacções de forma muito precisa.

• Exemplo da influência do compilador no tempo de execução:

→ Consideremos o seguinte código:

```
L1: a=b*c;
L2: b=d-e;
L3: d=a-f;
```

→ O tempo total de execução é $\sum_{i=1}^{3} T_{\text{exec}}(L_i)$. Mas o tempo, L_i, para executar cada uma das linhas depende do trabalho do compilador. Por exemplo:

```
L1.1: obtém endereço de c
L1.2: load c
L1.3: obtém endereço de b
L1.4: load b
L1.5: multiplica
L1.6: obtém endereço de a
L1.7: armazena resultado em a
```

→ E se as/algumas variáveis já estiverem em registo?...

• Exemplo de incerteza intrínseca ao próprio programa:

```
while(P){
   I1;
   I2;
   I3;
 }
```

→ P é uma condição e I1, I2, I3 são instruções.

→ Depende da evolução do valor lógico da condição P.

→ Por esta razão, algumas linguagens de tempo-real não permitem a utilização de ciclos while.

Testes de Escalonamento e Tempos de Resposta

• Dado que muitas aplicações em tempo-real são aplicações embebidas, e ainda dada a natureza multi-tarefa das mesmas, estas são extremamente difíceis de testar.

• Logo: importante desenvolver formas de verificação do seu funcionamento ainda durante a fase de projecto.

• Um dos aspectos que é absolutamente indispensável verificar é se todas as tarefas poderão executar de acordo com as restrições temporais impostas para o bom funcionamento do sistema.

• Surgem então os testes de escalonamento de tarefas e o
cálculo de tempos de resposta de tarefas.

Modelo de Processos Periódicos

• Assume-se que a aplicação é constituída por um número fixo de processos.

• Todos os processos são periódicos com períodos conhecidos.

• São ignorados todos os "overheads" do sistema, tempos de comutação, etc (assume-se que têm custo temporal nulo).

• Todos os processos têm uma meta temporal que é igual ao seu período (i.e. cada processo deve completar antes de ser de novo activado).

• Todos os processos têm um tempo fixo de execução no pior caso.

• Os processos são completamente independentes uns dos outros.

⇒ Instante crítico: ocorre num instante de tempo em que todos os processos sejam activados em conjunto.

Executivo Cíclico

• Com um conjunto de processos puramente periódicos, é possível construir um escalonamento completo que, quando repetidamente executado, causa que todos os processos corram com o período correcto.

• O executivo cíclico é essencialmente uma tabela de chamadas a procedimentos, onde cada procedimento representa um processo.

• A tabela completa é conhecida como um ciclo maior que tipicamente é constituído por um conjunto de ciclos menores.

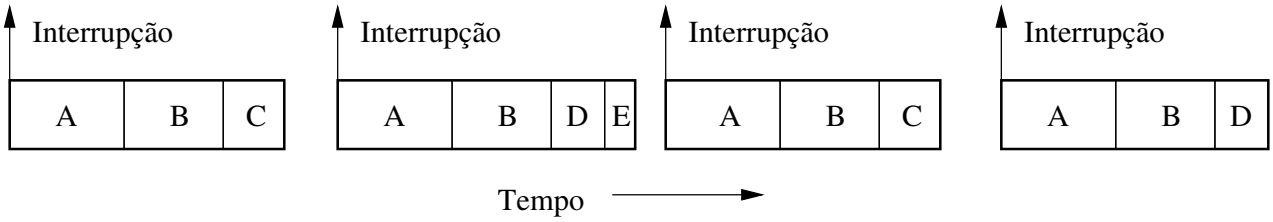
• Cada ciclo menor é iniciado quando ocorre uma interrupção periódica que controla o executivo em causa.

• Em cada ciclo menor é executado sequencialmente um (sub-) conjunto de processos. Esta sequência de processos repete-se no correspondente ciclo menor do ciclo maior seguinte.

• Exemplo: com 4 ciclos menores de 25 ms cada que compõem um ciclo maior de 100 ms.

Processo	*Período, T*	*Tempo de Cálculo*, C
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2

• Uma possível transformação/implementação num executivo cíclico:



• O código para um tal sistema poderia ter a seguinte forma simples:

ciclo

```
  Espera_por_Interrupção;
  Procedimento_do_Processo_A;
  Procedimento_do_Processo_B;
  Procedimento_do_Processo_C;
  Espera_por_Interrupção;
  Procedimento_do_Processo_A;
  Procedimento_do_Processo_B;
  Procedimento_do_Processo_D;
  Procedimento_do_Processo_E;
  Espera_por_Interrupção;
  Procedimento_do_Processo_A;
  Procedimento_do_Processo_B;
  Procedimento_do_Processo_C;
  Espera_por_Interrupção;
  Procedimento_do_Processo_A;
  Procedimento_do_Processo_B;
  Procedimento_do_Processo_D;
```

fimciclo;

• Algumas características deste método do executivo cíclico:

  - Nenhum processo real é executado; cada ciclo menor é apenas uma sequência de chamadas a procedimentos.
  - Os procedimentos partilham um espaço de endereçamento comum e podem portanto passar dados entre eles. Estes dados não precisam de ser protegidos (por um semáforo, por exemplo) se cada processo não possuir nenhum recurso quando termina um ciclo (menor) de execução (o que implica que não existam acessos concorrentes).
  - Se for possível construir o executivo cíclico, então nenhum teste de escalonabilidade é necessário (a prova de que o sistema cumpre as metas é efectuada 'por construção'). Contudo, para sistemas com elevada utilização a construção do executivo pode ser difícil - trata-se de um problema complexo.

• Dificuldades e desvantagens do executivo cíclico:

- Todos os períodos dos processos devem ser múltiplos do tempo de ciclo-menor.
- Dificuldade de incorporar processos esporádicos.
- Dificuldade de incorporar processos com períodos longos; o tempo do ciclo-maior é o máximo período que pode ser usado sem recurso a escalonamento secundário (i.e. um processo que chame um procedimento secundário em cada N ciclos menores).
- A dificuldade de efectivamente construir o executivo.
- Um processo com tempo de computação variável pode precisar de ser dividido num número de procedimentos de duração fixa. Mas isto dificulta a estruturação do software, e é portanto muito sujeito a erros.

Escalonamento Baseado em Processos

• O funcionamento do executivo cíclico é baseado apenas na execução de uma sequência de chamadas a procedimentos. Não existe a noção de processo durante a execução.

• Alternativa: suportar directamente a execução de processos (como num sistema operativo genérico) e determinar qual o processo a executar em cada instante a partir de um atributo de prioridade.

• Com este método assume-se que um processo está num de um conjunto de estados possíveis. Por exemplo, assumindo que não existe comunicação entre processos podemos pensar nos seguintes estados:

- pronto;
- suspenso à espera de um evento de temporização - apropriado para processos periódicos;
- suspenso à espera de um evento que não de temporização - apropriado para processos esporádicos;

• Com escalonamento baseado em prioridades, um processo de alta prioridade pode ser activado durante a execução de um processo de baixa prioridade. Nesta situação existem duas hipóteses:

- Num sistema preemptivo realizar-se-á uma comutação imediata para o processo de alta prioridade.
- Num sistema não-preemptivo é permitido que o processo de baixa prioridade complete a sua presente execução antes do outro executar.

• Em geral os sistemas preemptivos são preferidos porque permitem que os processos de alta prioridade sejam mais reactivos.

• Entre os extremos da preempção e da não-preempção, existem estratégias alternativas que permitem que um processo de baixa prioridade continue a executar durante um período limitado de tempo (não necessariamente até à conclusão).

Atribuição de Prioridades por Razão Monótona

• Com o modelo de processos periódicos existe um método simples mas óptimo de atribuição de prioridades: atribuição de prioridades por razão monótona.

• É atribuída a cada processo uma prioridade (única) em função do seu período: quanto menor o período, maior a prioridade, i.e. $T_i < T_j \Rightarrow P_i > P_j$, onde:

- $T_i \rightarrow$ tempo mínimo entre activações do processo i (período do processo).
- $P_i \rightarrow$ prioridade do processo i.

• Esta atribuição de prioridades é óptima no seguinte sentido: se um conjunto de processos pode ser escalonado usando escalonamento preemptivo baseado em prioridades fixas, então esse conjunto de processos também pode ser escalonado com escalonamento de razão monótona.

• Notemos que esta optimalidade aparece num contexto em que os valores das metas temporais são iguais aos períodos de activação: $D_i = T_i$.

Teste da Possibilidade de Escalonamento Baseado no Factor de Utilização

• Aplica-se a um modelo de processos periódicos com escalonamento por prioridades com razão monótona.

• Este teste consiste em verificar se a seguinte condição se verifica:

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq N\left(2^{1/N} - 1\right) \qquad \text{(Liu e Layland, 1973)}$$

• $C_i \rightarrow$ pior tempo de cálculo do processo i.

• $T_i \rightarrow$ tempo mínimo entre activações do processo i (período do processo).

• Notemos que a soma calcula a percentagem total de utilização do processador por parte do conjunto de todos os processos.

• Para valores grandes de N o limite superior para a soma dos factores de utilização (C_i/T_i) tende para 69.3%.

• Tabela de limites de utilização para alguns valores de N:

N	*Limite de utilização (%)*
1	100.0
2	82.8
3	78.0
4	75.7
5	74.3
10	71.8

• Este teste de escalonamento poderá fornecer apenas uma simples resposta em termos de "sim" ou "não". Não dá qualquer indicação sobre o verdadeiro tempo de resposta dos processos.

• Este teste é uma condição suficiente mas não necessária.

- Se um conjunto de processos passa no teste então serão cumpridas todas as metas; mas se falhar o teste, nada se pode concluir (pode falhar ou não).

• Alternativa para testar a escalonabilidade →

• Diagrama de Gantt: desenhar diversas linhas-temporais cada uma das quais representa um processo. Cada linha temporal identifica numa escala temporal os intervalos de tempo em que o processo está em execução, em preempção, ou inactivo, e os instantes de tempo em que são cumpridas ou falhadas metas temporais - para desenhar estas linhas é portanto necessário simular o funcionamento dos processos.

• Teste de escalonabilidade pelo Diagrama de Gantt: até que instante temporal teremos que completar o diagrama até poder concluir que nenhuma meta será falhada no futuro?

• Para processos com o mesmo instante de activação (i.e. partilham um instante crítico) basta desenhar/analisar até decorrer o maior dos períodos dos processos (Liu e Layland, 1973).

• Portanto, se todos os processos cumprem a sua primeira meta temporal, então também cumprirão todas as metas futuras.

Análise/Cálculo dos Tempos de Resposta

• O tempo de resposta de um processo é o tempo que vai desde o instante em que foi activado até ao instante em que conclui as operações relativas ao evento que o activou.

• Notação: $R_i \rightarrow$ tempo de resposta do processo i; no pior caso; $D_i \rightarrow$ meta temporal do processo i.

• Um processo isolado ou um processo de máxima prioridade têm um tempo de resposta R igual ao seu tempo de computação medido no pior caso, C.

• A meta temporal de um processo deverá ser sempre igual ao mínimo período de tempo entre os eventos que o activam, ou seja $D = T$.

• Consideremos um conjunto de processos sujeitos a escalonamento preemptivo com prioridades atribuídas por razão monótona.

• Neste caso o tempo de resposta do processo i depedende não só do seu pior tempo de computação, C_i, mas também da interferência, I_i, dos processos de prioridade superior:

$$R_i = C_i + I_i \tag{1}$$

onde I_i é a máxima interferência que um processo i pode experimentar em qualquer intervalo $[t, t + R_i[$

• A interferência máxima ocorre numa situação em que todos os processos de prioridade mais elevada sejam activados no mesmo instante que o processo i (um instante crítico portanto). Sem perda de generalidade, podemos considerar que todos os processos são activados no instante 0.

• Consideremos um processo j cuja prioridade é superior à de i. Este processo será activado pelo menos uma vez no intervalo $[0, R_i[$ (porquê?), podendo no entanto ser activado um número de vezes superior dado por:

$$\text{Activações}_j = \left\lceil \frac{R_i}{T_j} \right\rceil$$

onde a função de tecto ($\lceil \cdot \rceil$) devolve o menor inteiro que é maior ou igual ao seu argumento (e.g. $\lceil \pi \rceil = 4$).

• Como cada activação de j causa uma interferência máxima C_j, então o valor máximo de interferência produzida pelo processo j durante a execução do processo i é dado por:

$$I_{\max} = \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

• Generalizando agora para qualquer número de processos, facilmente se conclui que todos os processos de prioridade superior vão interferir com i, logo:

$$I_i = \sum_{j \in \text{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2}$$

onde $\text{hp}(i)$ é o conjunto de processos cuja prioridade é superior à de i.

• Substituindo (2) em (1) obtemos:

$$R_i = C_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{3}$$

• Esta equação é de difícil resolução por conter uma operação de vírgula fixa ($\lceil \cdot \rceil$), pois vão existir inúmeras soluções para a mesma. O menor dos valores de solução para R_i representa o tempo de resposta do processo no pior caso.

• Audsley apresentou uma forma de resolver a equação através da seguinte relação recorrente:

$$w_i^{n+1} = C_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \tag{4}$$

onde o conjunto de valores $\{w_i^0, w_i^1, w_i^2, \ldots, w_i^n, \ldots\}$ é monotonamente não-decrescente.

• Quando $w_i^{n+1} = w_i^n$, então é essa a solução da equação (4). Se $w_i^0 < R_i$ então w_i^n é a menor solução, sendo portanto a que procuramos.

• Por outro lado, se a equação (4) não tiver solução, então os valores de w vão crescer indefinidamente. É o que se passa quando existe um processo de baixa prioridade inserido num conjunto de processos onde o factor de utilização total é superior a 100%. Este caso é facilmente detectado aplicando a restrição $w_i \leq T_i$. Se isto não se verificar (ou seja se ocorrer crescimento indefinido) então o processo não poderá cumprir a sua meta temporal pois apresenta um tempo de resposta superior.

• Algoritmo: cálculo dos tempos de resposta dos processos de um conjunto.

para cada processo i de 1 a N
  $n = 0$
  $w_i^n = C_i$
  ciclo
    calcula w_i^{n+1} através da equação (4)
    se $w_i^{n+1} == w_i^n$ **então**
      $R_i = w_i^n$
      termina (valor encontrado)
    fimse;
    se $w_i^{n+1} > T_i$ **então**
      avisa que i não cumpre a meta
      termina (valor não encontrado)
    fimse;
    $n = n + 1$
  fimciclo;
fimpara;

• Este algoritmo mostra uma forma para obter os tempos de resposta para os vários processos ou a indicação de que um dos processos não poderá cumprir a sua meta temporal.
• Pode ocorrer o caso em que o método permite verificar que determinado conjunto de processos pode ser escalonado apesar do teste baseado no factor de utilização falhar.
• Os valores para os piores tempos de execução, C_i, deverão ser medidos ou estimados.
• Forma possível de efectuar a medição: placa de E/S digital + controlo de bit no início e fim da execução + medição com osciloscópio digital ou analisador lógico.
• No entanto esta medição poderá não ser conclusiva pois nem sempre poderemos ter a certeza que medimos o pior dos tempos de execução.
• Alternativa para estimar os piores tempos de execução → através de estimação → baseada no modelo do processador utilizado e analisando os diversos blocos do programa.

Processos Esporádicos e Aperiódicos

• No teste apresentado acima sobre a possibilidade dos processos cumprirem as suas metas temporais baseado no cálculo dos seus tempos de resposta, entra-se em linha de conta com o período entre activações dos processos.

• Mas os processos esporádicos são por definição aperiódicos e por isso não podemos definir um período T_i para os mesmos.

• Como podemos entrar em linha de conta com processos que tanto podem ser activados apenas algumas vezes por dia como em determinadas alturas ser activados milhares de vezes por minuto?

• A solução para expandir o modelo anterior de forma a incluir requisitos de processos esporádicos (e aperiódicos) é interpretar o período T_i como o tempo mínimo entre chegadas da tarefa i.

• É garantido que um processo esporádico com $T_i = 20\,\text{ms}$ não chega mais do que uma vez em cada intervalo de 20 ms.

→ Na realidade pode chegar com uma frequência muito inferior a uma vez em cada 20 ms mas o teste do tempo de resposta assegurará que a taxa máxima se possa aguentar (se passar no teste).

• Outro requisito para a inclusão de processos esporádicos diz respeito à definição da meta temporal.

• No modelo analisado anteriormente era assumido que $D = T$. Mas para processos esporádicos isto não é razoável.

• Para processos esporádicos é natural assumir que temos uma meta temporal, D, inferior ao período de chegada, T, i.e.: $D < T$.

• Uma inspecção do algoritmo de análise de tempos de resposta descrito na página 79 revela que:

1) este funciona perfeitamente para valores de D_i inferiores a T_i desde que se passe a utilizar o critério de paragem $w_i^{n+1} > D_i$;

2) este funciona perfeitamente para qualquer ordem de ordenamento de prioridades - hp(i) fornece sempre o conjunto de processos de prioridade superior à de i.

• Com a ligeira alteração do critério de paragem obtemos o seguinte algoritmo:

para cada processo i de 1 a N
  $n = 0$
  $w_i^n = C_i$
  ciclo
    calcula w_i^{n+1} através da equação (4)
    se $w_i^{n+1} == w_i^n$ **então**
      $R_i = w_i^n$
      termina (valor encontrado)
    fimse;
    se $w_i^{n+1} > D_i$ **então**
      avisa que i não cumpre a meta
      termina (valor não encontrado)
    fimse;
    $n = n + 1$
  fimciclo;
fimpara;

• Embora algumas ordens de prioridades sejam melhores que outras, o teste fornecerá os tempos de resposta no pior caso para o ordenamento de prioridades utilizado.

• Mais à frente será definida uma ordem óptima de prioridades para $D_i < T_i$.

• Também existe um algoritmo mais genérico, e uma ordem óptima de prioridades, para o caso geral em que de $D_i < T_i$, $D_i = T_i$, $D_i > T_i$.

Processos "Hard" e "Soft"

• Em muitos casos os valores obtidos para os piores tempos de computação ou para o número máximo de eventos por unidade de tempo que activam os processos esporádicos estão muito acima dos valores médios.

• Isto leva a que os tempos de resposta calculados sejam muito elevados e que o sistema tenha que ser de tal forma sobre-dimensionado que o factor de utilização do processador seja muito baixo.

• Como em qualquer área de engenharia, o sobre-dimensionamento conduz a custos elevados. Muitas vezes somos confrontados com a questão de como reduzir o sobre-dimensionamento mantendo a fiabilidade do sistema.

• Podemos tomar duas regras como base para estabelecer os requisitos mínimos:

1. Todos os processos deverão poder cumprir as suas metas temporais quando utilizamos valores médios para os tempos de execução e para os intervalos entre eventos esporádicos.

2. Todos os processos "hard" deverão poder cumprir as respectivas metas temporais mesmo no pior dos casos.

• Como consequência da primeira regra teremos situações em que nem todos os processos poderão atingir as suas metas. Esta condição é conhecida como **sobrecarga transitória**, e esses processos deverão ser claramente identificados na fase de projecto como processos não-críticos ("soft").

• Quanto aos processos críticos a segunda regra garante que estes nunca falharão as suas metas temporais.

• A regra 2 poderá dar origem a factores médios de utilização do processador bastante reduzidos, mas se a etiquetagem "hard" tiver sido correctamente feita, pouco haverá a fazer para além de tentar diminuir os tempos de execução nos piores casos e/ou as frequências de chegada nos piores casos.

Atribuição de Prioridades por Ordenamento Monótono de Metas Temporais (APOMMT)

• O método de atribuição de prioridades por razão monótona é óptimo para o caso em que os processos apresentam metas temporais iguais aos períodos entre activações ($D_i = T_i$).

• É possível formular um método semelhante para a atribuição de prioridades para o caso em que $D_i < T_i$. Neste caso, a atribuição de prioridades é feita com base nas metas temporais D_i.

• Neste método, quanto mais curto for o tempo entre o instante de activação de um processo e o instante limite para este produzir o resultado correspondente (meta) maior será a prioridade atribuída ao mesmo: $D_i < D_j \Rightarrow P_i > P_j$.

• Notemos que este método, se aplicado ao caso em que $D_i = T_i$, resulta na atribuição de prioridades por razão monótona.

• Esta atribuição de prioridades é óptima no seguinte sentido: se um conjunto de processos pode ser escalonado usando um determinado método de prioridades fixas, então esse conjunto de processos também pode ser escalonado com atribuição de prioridades por ordenamento monótono de metas temporais.

• Exemplo: a seguinte tabela dá a atribuição de prioridades apropriada para um conjunto de processos simples. Também inclui os tempos de resposta no pior caso (calculado com o algoritmo da página 81). Notemos que o ordenamento de prioridades por razão monótona não permitiria o escalonamento destes processos.

Tarefa	*Período, T*	*Meta, D*	*Tempo de Cálculo, C*	*Prioridade, P*	*Tempo de Resposta, R*
1	20	5	3	4	3
2	15	7	3	3	6
3	10	10	4	2	10
4	20	20	3	1	20

• Outro exemplo: dado um conjunto de processos com as seguintes características:

Processo	*Período, T*	*Meta, D*	*Tempo de Cálculo, C*	*Prioridade, P*	*Tempo de Resposta, R*
1	40	3	2	alta	2
2	20	7	3	média	5
3	10	10	5	baixa	10

• Se utilizassemos razão monótona teriamos um ordenamento inverso para as prioridades dos processos: (P3,P2,P1).

• Vamos calcular os tempos de resposta dos processos.

  • Para o processo 1: como este é o que tem a prioridade mais elevada temos simplesmente:

$$R_1 = C_1 = 2$$

  • Para o processo 2 vem:

$$\begin{aligned} w_2^0 &= 3 \\ w_2^1 &= 3 + \left\lceil \frac{3}{40} \right\rceil \times 2 = 5 \\ w_2^2 &= 3 + \left\lceil \frac{5}{40} \right\rceil \times 2 = 5 \\ R_2 &= 5 \end{aligned}$$

  • Finalmente para o processo 3 vem:

$$\begin{aligned} w_3^0 &= 5 \\ w_3^1 &= 5 + \left\lceil \frac{5}{40} \right\rceil \times 2 + \left\lceil \frac{5}{20} \right\rceil \times 3 = 10 \\ w_3^2 &= 5 + \left\lceil \frac{10}{40} \right\rceil \times 2 + \left\lceil \frac{10}{20} \right\rceil \times 3 = 10 \\ R_3 &= 10 \end{aligned}$$

• Comparando os tempos de resposta obtidos com as respectivas metas temporais verificamos que os três processos conseguem atingir as suas metas temporais com esta distribuição de prioridades.

• Exercício: verifique que se utilizarmos a atribuição de prioridades de razão monótona baseada no período dos processos, apenas 1 consegue cumprir a sua meta temporal.

Tempos de Resposta em Situações de Bloqueio

• Tal como já vimos, os algoritmos (protocolos) de herança de prioridades garantem que um processo é bloqueado por outro de prioridade inferior durante um tempo limitado.

• Vê-se facilmente que um processo pode ser bloqueado por outro de prioridade inferior no máximo uma vez. Por isso existe um número máximo de vezes que um processo pode ser bloqueado por outros de prioridade inferior.

• Consideremos o caso do protocolo básico de herança de prioridades (PBHP).

• Se um processo tiver m secções críticas (onde poderá ser bloqueado) então esse processo poderá bloquear no máximo m vezes. Isto porque quando um processo inicia a execução, cada uma das m secções críticas pode já estar a executar por um processo de prioridade inferior.

• Mas se existirem apenas n (com $n < m$) processos de prioridade inferior então o número máximo de bloqueiros que o processo pode encontrar será n. Temos portanto que:

$$\text{Bloqueios} = \min(m, n)$$

• Seja B_i o tempo máximo de bloqueio que um processo i pode sofrer quando se utilizam protocolos de herança de prioridades.

• Seja K o número de secções críticas (recursos) existentes. A equação seguinte fornece um limite superior para B_i:

$$B_i = \sum_{k=1}^{K} \text{usada}(k, i) \times CS(k) \tag{5}$$

onde usada(k, i) é uma função que devolve o valor 0 ou 1 e é definida da seguinte forma: usada$(k, i) = 1$ se o recurso k é utilizado por pelo menos um processo de prioridade inferior à de i, e por pelo menos um processo de prioridade igual ou superior à de i. Nos restantes casos a função devolve o valor 0. $CS(k)$ corresponde ao custo (duração) de executar a secção crítica k.

• O tempo de bloqueio de um processo B_i contabiliza não só o tempo em que este se encontra impossibilitado de entrar numa secção crítica por outro processo de prioridade inferior a estar a utilizar, mas também o bloqueio que advém do facto de um processo de prioridade inferior ter herdado uma prioridade igual ou superior à sua por partilhar uma secção crítica com processos de prioridade igual ou superior à de i.

• Cálculo dos Tempos de Resposta com o PBHP:

• O tempo de resposta de um processo é dado por:

$$R = C + B + I$$

ou seja, é a soma do tempo de cálculo, com o tempo de bloqueio em recursos, e com o tempo que o processo esteve bloqueado por interferência dos processos de prioridade mais elevada.

• Reescrevendo a equação (3) temos:

$$R_i = C_i + B_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{6}$$

• Tal como dantes, esta equação pode ser resolvida utilizando a mesma equação de recorrência utilizada em (4), mas agora adaptada para incluir o tempo de bloqueio:

$$w_i^{n+1} = C_i + B_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \tag{7}$$

• Uma vez que é possível determinar os tempos de resposta na presença de situações de bloqueio, podemos utilizar esse cálculo como teste para verificar a possibilidade de as tarefas poderem ser escalonadas e cumprir as suas metas.

• Notemos que esta formulação pode ser demasiado pessimista (i.e. suficiente mas não necessária). O facto de o processo sofrer ou não o seu máximo bloqueio possível depende da "fase relativa" entre os diversos processos. Contudo, em geral, a equação (6) representa um teste de escalonamento eficiente para sistemas de tempo real cooperantes.

Tempos de Resposta Usando PCP

• O bloqueio de um primeiro recurso é permitido. O efeito do protocolo é assegurar que um segundo recurso pode ser bloqueado apenas se não existir um processo de prioridade mais elevada que usa ambos os recursos.

• Uma das consequências da utilização do protocolo de "tecto de prioridades" (PCP) é que um processo de prioridade elevada pode ser bloqueado no máximo uma vez (por activação) por processos de prioridade inferior.

• Consequentemente, o tempo máximo que um processo pode ser bloqueado, B_i, é igual ao tempo de execução da maior secção crítica de prioridade inferior que é acedida por tarefas de prioridade não inferior; i.e. a equação (5) torna-se na seguinte:

$$B_i = \max_{k=1}^{K} \text{usada}(k, i) \times CS(k)$$

• Por outro lado, surge como desvantagem o facto de que, com este protocolo, mais processos irão encontrar bloqueio no acesso aos recursos.

• Este tempo de bloqueio em recursos no pior caso mantém-se no protocolo de tecto imediato de prioridades.

Tempos de Resposta com Escalonamento Cooperativo

• Consideremos o seguinte modelo de escalonamento cooperativo, chamado preempção diferida:

- Cada tarefa é dividida em blocos, não sujeitos a preempção e cujos tempos de execução estão limitados a $B_{\max}$.
- No fim de cada um destes blocos a aplicação oferece ao sistema ("kernel") um pedido/oportunidade de de-escalonamento. Se um processo de (mais) elevada prioridade estiver pronto a executar, então o sistema provoca uma troca de contexto, senão o processo presentemente em execução continuará para o próximo bloco "não-preemptível".

• A execução do conjunto de processos que compõem a aplicação é portanto totalmente cooperativa.

• A exclusão mútua fica assegurada desde que todas as secções críticas estejam completamente contidas dentro de chamadas de de-escalonamento.

• Este método requer portanto a colocação cuidadosa de todas as chamadas de de-escalonamento.

• Para fornecer protecção contra software incorrecto (ou corrompido), podem usar-se sinais assíncronos ou abortar o processo para o remover se algum bloco não preemptível durar mais do que $B_{\max}$.

• O uso de preempção diferida tem duas vantagens importantes. **(1)** Aumenta a escalonabilidade do sistema, e **(2)** pode levar a menores valores de C.

• Vejamos a primeira vantagem calculando os tempos de resposta: Usando preempção diferida não pode haver interferência durante o último bloco de execução d1 tarefa.

• Seja F_i o tempo de execução do bloco final ($F_i < B_{\max}$), de forma a que quando o processo já consumiu um tempo $C_i - F_i$, acabou de entrar no bloco final. Podemos alterar a equação (7) (pág. 86) para calcular o tempo de resposta da parte inicial "$C_i - F_i$" do processo:

$$w_i^{n+1} = C_i - F_i + B_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

• Quando esta equação iterativa convergir (i.e. $w_i^{n+1} = w_i^n$), basta somar o tempo de resposta do último bloco para obter o tempo total:

$$R_i = w_i^n + F_i$$

• Na prática, o último bloco do processo é executado com uma prioridade mais elevada (a máxima) do que a do resto do processo.

• A outra vantagem da preempção diferida (menores valores de C) deve-se à possibilidade de predizer com maior precisão os tempos de execução dos blocos não-preemptíveis do processo. Sem blocos não preemptíveis, os estimadores dos piores tempos de execução são forçados a ignorar as vantagens de caches, "prefetch queues", e "pipelines", obtendo assim estimativas pessimistas pois a preempção invalida caches e "pipelines".

• A certeza de não preempção pode ser usada para predizer o *speed-up* que ocorrerá na prática. Contudo se o custo de oferecer uma troca de contexto for elevado, isto poderá diminuir estas vantagens.

Ruído de Activação

• No modelo simples, é assumido que todos os processos são activados com perfeita periodicidade: i.e. se um processo L tem periodicidade T_i, então é activado exactamente com essa frequência.

• Processos esporádicos são incorporados no modelo considerando/assumindo que o seu mínimo intervalo inter-activações é T.

• No entanto, os processos esporádicos ou periódicos podem ter outro tipo de incerteza de activação: trata-se, não de incerteza medida em relação ao instante temporal da última activação, mas sim em relação aos instantes definidos pela periodicidade do processo T.

• Define-se assim uma variação máxima para os instantes de activação do processo tal como esses instantes são determinados pelo período T. Esta variação é designada por ruído ("jitter"), J.

• Exemplo: consideremos um processo esporádico S que é activado no fim de cada activação de um processo periódico, L, a executar noutro processador. Numa das activações L poderá ter completado apenas o mais tarde possível, i.e. só após R_L; mas na activação seguinte L pode ter completado no mínimo tempo possível C_L ($C_L < R_L$, em particular podemos ter $C_L = 0$). Isto faz com que o processo S tenha um ruído de activação de $R_L - C_L$. As duas execuções do processo esporádico estão separadas não por T_L mas sim por $T_L - (R_L - C_L)$.

• O período do processo L é T_L e o processo esporádico terá o mesmo período, mas é incorrecto assumir que a máxima interferência que S exerce em processos de prioridade inferior pode ser representada na equação (3) como um processo periódico de período $T_S = T_L$. A máxima variação na activação do processo esporádico, S, é designada "jitter", $J_S = R_L - C_L$.

• Obviamente que o ruído vai ter influência na interferência exercida no tempo de resposta dos processos de prioridade inferior. Notemos que apenas tem interesse quando L é remoto. Se não for esse o caso, então as variações na activação de S poderiam ser tidas em conta com as equações standard assumindo a existência de um instante crítico entre o activador e o activado.

• O máximo impacto em termos de interferência ocorrerá quando os instantes reais de activação do processo forem: 0, $T - J$, $2T - J$, $3T - J$ etc (ocorrerem o mais cedo possível).

• Seja S um processo esporádico com características T e J. Um processo i sofrerá de S uma interferência se $R_i \in [0, T - J[$, duas se $R_i \in [T - J, 2T - J[$, três se $R_i \in [2T - J, 3T - J[$, etc. Ou de outra forma: uma interferência se $R_i + J \in [0, T[$, duas se $R_i + J \in [T, 2T[$, três se $R_i + J \in [2T, 3T[$, etc.

• Isto pode ser representado de forma semelhante à equação (6) como se segue:

$$R_i = C_i + B_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j$$

• Em geral, processos periódicos não sofrem ruído de activação. Uma implementação pode contudo restringir a granularidade do temporizador do sistema que activa os processos periódicos. Numa tal situação, um processo periódico pode também sofrer ruído de activação.

• Por exemplo, $T = 10$ e uma granularidade no sistema de 8 implicará um valor de 6 para o ruído - no instante 16 o processo periódico será activado para a sua invocação do instante '10'.

• Neste caso, se quisermos medir o tempo de resposta, $R_i^{\text{periódico}}$, em relação ao instante real de activação, então deve-se adicionar o valor do ruído ao valor previamente calculado.

$$R_i^{\text{periódico}} = R_i + J_i.$$

Se este novo valor for superior a T_i então terá que se usar a seguinte análise.

Tempos de Resposta para Metas Arbitrárias

• Para situações em que D_i (e portanto potencialmente também R_i) podem ser maiores que T_i, tem que se adaptar novamente o método de análise.

• Quando $D_i \leq T_i$ é necessário considerar apenas uma única activação do processo para o qual se está a calcular o tempo de resposta. O instante crítico quando todos os processos são activados ao mesmo tempo, representa a interferência máxima e portanto o tempo de resposta a seguir a um instante crítico constitui o pior caso.

• Contudo, quando a meta é superior ao período têm de se considerar várias activações do próprio processo. Na análise seguinte assume-se que a activação de um processo será atrasada até que quaisquer activações anteriores do mesmo processo tenham completado.

• Se um processo continuar a executar para dentro do próximo período de tempo, então têm-se de analisar ambas as activações para ver qual dá origem ao maior tempo de resposta. Se a segunda activação não se completa antes da terceira, então esta nova activação também deve ser considerada, e assim sucessivamente.

• Para cada tarefa, consideram-se sucessivamente $q = 0, 1, 2, \ldots$ sobreposições de activação, calculando-se sucessivamente o tempo de resposta total das $(q+1)$ activações envolvidas.

• Ignorando o ruído de activação, a equação (7) pode ser extendida da seguinte forma:

$$w_i^{n+1}(q) = (q+1)C_i + B_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j \tag{8}$$

Notemos que se houver sobreposição de activações, o termo B_i (tempo blocagem por processos de prioridade inferior) só tem influência uma vez (porquê?).

• Define-se portanto uma janela de resposta $w(q)$ para a activação $(q+1)$ $(q = 0, 1, 2, \ldots)$. Por exemplo, com $q = 2$ ocorrerão 3 activações do processo dentro da janela de resposta.

• Para cada valor de q chega-se iterativamente a um valor estável de $w(q)$ - tal como na equação (7). O tempo de resposta é então dado por (retirando q períodos ao total):

$$R_i(q) = w_i^n(q) - qT_i \tag{9}$$

Por exemplo, com $q = 2$ o processo começou $2T_i$ dentro da janela e portanto o tempo de resposta é o tamanho da janela menos $2T_i$.

• O número de sobreposições de activação que se tem que considerar é o menor valor de q para o qual a seguinte relação é verdadeira:

$$R_i(q) \leq T_i \tag{10}$$

Neste ponto, o processo termina antes da próxima activação e portanto não existe sobreposição nas activações subsequentes.

• O pior tempo de resposta é então dado pelo pior tempo de resposta dentro dos obtidos para todos os q:

$$R_i = \max_{q=0,1,2,\ldots} R_i(q)$$

• Notemos que para $D_i \leq T_i$, a relação da equação (10) é verdadeira para $q = 0$ (se o escalonamento do processo puder ser garantido). Neste caso, as equações (8) e (9) podem ser simplificadas para a equação original (6).

• Se para algum q, $R_i(q) > D_i$ então o processo não é escalonável.

• Se combinarmos esta formulação de meta arbitrária com o efeito de ruído de activação são necessárias duas alterações na análise anterior:

1. O factor/tempo de interferência tem que ser aumentado se algum processo de prioridade mais elevada sofrer de ruído de activação:

$$w_i^{n+1}(q) = (q+1)C_i + B_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{w_i^n(q) + J_j}{T_j} \right\rceil C_j$$

2. Se o próprio processo puder sofrer ruído de activação, então pode ocorrer sobreposição de duas janelas consecutivas se o tempo de resposta mais o ruído for superior ao período. Para levar em conta este facto, temos de alterar a equação (9):

$$R_i(q) = w_i^n(q) - qT_i + J_i$$

Atribuição de Prioridades

• Problema: como ordenar as prioridades dos processos no caso de metas arbitrarias? A formulação dada na secção anterior tem a propriedade de não existir nenhum algoritmo simples (tal como razão monótona, ou "metas monótonas") para o ordenamento de prioridades óptimo para os processos.

• No entanto a seguinte propriedade permite-nos atribuir prioridades em situações de metas arbitrárias.

Propriedade: se se atribuir a um processo τ a menor prioridade e assim ele for escalonável então, se existir uma ordem de prioridades admissível (escalonável) para o conjunto completo de processos, existirá também uma ordem admissível com a menor prioridade atribuída a τ.

• Se um processo for escalonável na prioridade mais baixa pode-se-lhe atribuir essa prioridade e tudo o que resta é atribuir as restantes $N - 1$ prioridades.

• A propriedade pode ser depois aplicada recursivamente ao conjunto dos processos restantes. Através da sucessiva aplicação pode-se obter uma ordem de prioridades completa (se uma existir).

• Algoritmo de atribuição de prioridades (aqui `Set(1)` → menor prioridade; `Set(N)` → maior prioridade); a função "`Process_Test`" testa se o processo K é escalonável nessa posição da tabela de prioridades:

```
procedure Assign_Pri (Set :  in out Process_Set; N : Natural;
                      Ok :  out Boolean) is
begin
  for K in 1..N loop
    for Next in K..N loop
      Swap(Set, K, Next);
      Process_Test(Set, K, Ok);
      exit when Ok;
    end loop;
    exit when not Ok; - failed to find a schedulable process
  end loop;
end Assign_Pri;
```

Influência do Sistema Operativo nos Tempos de Resposta

• Se existir um sistema de operação (núcleo, "kernel") os atrasos por ele provocados também têm de ser levados em conta para a análise de tempos de resposta e escalonabilidade.

• Podemos enumerar as seguintes características típicas de muitos núcleos de tempo-real:

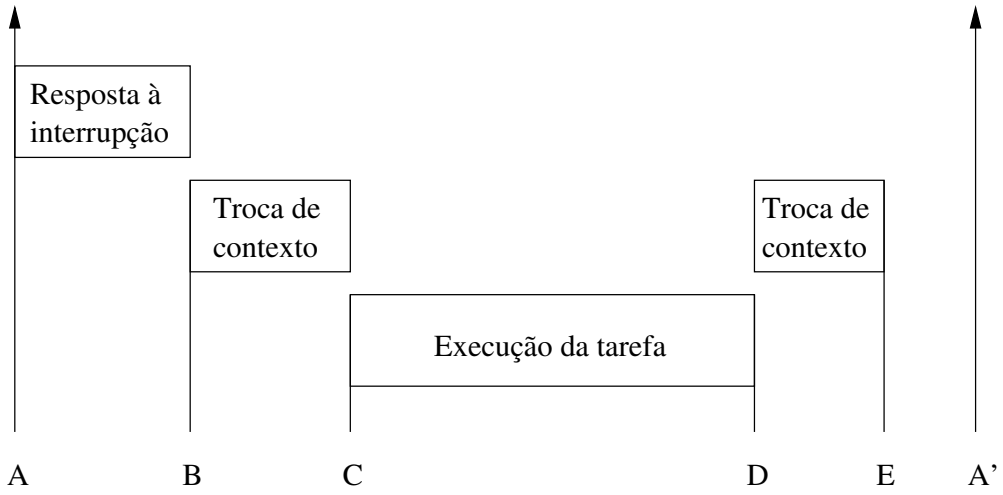
- O custo de uma troca de contexto não é desprezável e pode ter mais do que um valor possível. O custo de uma troca de contexto para um processo de prioridade mais elevada (e.g. na sequência de uma interrupção de relógio) pode ser mais elevado do que a troca de contexto para um processo de prioridade mais baixa (no fim da execução de um processo de prioridade alta). Com um número elevado de processos periódicos pode-se incorrer num custo adicional para manipular a fila de (processos) temporizados.

- Todas as trocas de contexto são não-preemptivas.
- O custo de responder a uma interrupção (que não o relógio) e activar uma aplicação periódica não é desprezável. Além disso, DMA e dispositivos controlados por canal envolvem o uso de memória partilhada o que pode ter um impacto não trivial no desempenho no pior caso; o melhor será evitar tais dispositivos em sistemas de tempo-real duros.
- Em cada interrupção (periódica) de relógio pode ser desencadeada uma transferência de um conjunto de processos da fila de temporizados para a fila de prontos. O custo desta operação depende do número de processos transferidos.

Modelação dos Tempos de Troca de Contexto

• Muitos modelos de análise dos tempos de resposta ignoram os tempos de troca de contexto. Contudo, isso é demasiado simplista se o custo total das trocas de contexto não for desprezável em comparação com o tempo de cálculo da própria aplicação.

• A figura seguinte ilustra um conjunto de eventos importantes na execução de um típico processo periódico:



A: A interrupção de relógio que corresponde ao instante em que o processo deveria iniciar a execução (assumindo a inexistência de ruído de activação e de atrasos preemptivos; se as interrupções estiverem inibidas devido a uma operação de troca de contexto, então a rotina de resposta à interrupção do relógio teria a sua resposta atrasada; isto seria levado em conta nas equações de escalonamento através do factor de bloqueio B);

B: O instante mais tarde em que a rotina do relógio poderá terminar. É neste instante que começará a troca de contexto para o novo processo (assumindo que este é o de prioridade mais elevada pronto a correr);

C: O verdadeiro início da execução do processo;

D: O instante de conclusão do processo (o processo pode sofrer várias vezes preempção entre os instantes C e D);

E: A conclusão da troca de contexto à saída do processo;

A': A próxima activação do processo.

• Um requisito temporal típico é que o processo termine antes da sua próxima activação ($D < A'$), ou antes de uma meta prévia à próxima activação. Em ambos os casos, D é o instante importante e não E.

• Outro requisito típico pode ser o de estabelecer um limite superior para o tempo entre o início e o fim da execução (i.e. $D - C$). Isto pode ocorrer quando a primeira acção é uma entrada, a última é uma saída, e existe uma meta temporal que as relaciona.

• Estes factores afectam o significado que tem a "meta do processo" (e portanto o seu tempo de resposta) mas não afectam a interferência que este processo tem em processos de prioridade mais baixa, onde o custo de ambas as trocas de contexto contam.

• Como já vimos (equação (6)), a equação básica de cálculo dos tempos de resposta tem a seguinte forma:

$$R_i = C_i + B_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{11}$$

• Levando agora em conta as trocas de contexto temos:

$$R_i = CS_1 + C_i + B_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS_1 + CS_2 + C_j) \tag{12}$$

onde CS_1 é o custo da troca de contexto inicial (para o processo) e CS_2 é o custo da troca de contexto após a execução do processo.

• Esta medida do tempo de resposta é em relação ao ponto B. Para medir em relação ao ponto C deverá remover-se (apenas) o primeiro dos termos CS_1 (porquê só o primeiro?). Para medir em relação ao ponto A é necessário medir/modelar o comportamento da rotina do relógio.

• O custo de colocar o processo na lista de temporizados (se for periódico) é incorporado em C_i. Na prática este valor pode depender do tamanho da fila - poderá ser necessário incorporar um valor máximo em C_i.

Modelação dos Tempos de Troca de Contexto

• Para processos esporádicos activados por outros processos (periódicos ou esporádicos), a equação (12) mantém-se um modelo válido. Contudo, o tempo de cálculo do processo, C_i, tem que incluir os custos do bloqueio que é efectuado no agente que controla a sua activação (a activação do processo i).

• Quando os processos esporádicos são activados por interrupções, pode ocorrer inversão de prioridades. Mesmo se o processo esporádico tiver uma prioridade baixa (e.g. devido a ter uma meta longa), a própria interrupção vai ser executada com uma prioridade de hardware elevada.

• Seja Γ_s o conjunto de processos esporádicos activados por interrupções. Assume-se que cada fonte de interrupção tem a mesma característica de chegada que o processo esporádico que activa. A interferência adicional que as rotinas de interrupção têm em cada processo é dada por:

$$\sum_{k \in \Gamma_s} \left\lceil \frac{R_i}{T_k} \right\rceil IH_k$$

onde IH_k é o custo de processar a interrupção k (e regressar ao processo que está a correr, após ter activado o processo esporádico). $T_k \rightarrow$ minimo tempo entre activações do processo k.

• Podemos agora reescrever a equação (12) da seguinte forma:

$$R_i = CS_1 + C_i + B_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS_1 + CS_2 + C_j) + \sum_{k \in \Gamma_s} \left\lceil \frac{R_i}{T_k} \right\rceil IH_k \quad (13)$$

Modelação do Relógio de Tempo-Real

• Para implementar processos periódicos, o sistema deverá ter acesso a um relógio de tempo-real que gere interrupções em instantes apropriados:

- Um sistema ideal utilizará um temporizador de intervalos que gerará interrupções apenas quando uma tarefa necessita de ser activada.
- Um método mais comum é utilizar um relógio que gera interrupções com uma frequência fixa (e.g. $T = 10\,\mathrm{ms}$) e a rotina de resposta deve decidir se devem ser activados zero, um, ou mais processos.

• O método ideal pode ser modelado por um processo idêntico ao usado para os processos esporádicos (equação (13)).

• Com o método do relógio regular é necessário desenvolver um modelo mais detalhado. A interferência da rotina de relógio pode depender do número de processos a transferir da fila de temporizados para a fila de processos prontos (número de processo activados).

Estado da fila de temporizados	*Tempo para processar o relógio* μs
Nenhum processo na fila	16
Processos na fila mas nenhum retirado	24
Um processo retirado	88
Dois processos removidos	128
Vinte cinco processos removidos	1048

• Como todo o processamento da rotina de interrupção é efectuado com prioridade de hardware elevada (a mais elevada) irá ocorrer inversão de prioridades.

• Um modelo para representar o custo de movimentar N processos periódicos da fila de temporizados para a fila de processos prontos é o seguinte:

$$C_{clk} = CT_c + CT_s + (N-1)CT_m$$

onde CT_c é o custo constante (assumindo que existe sempre um processo na fila de temporizados), CT_s é o custo de fazer uma única transferência, e CT_m é o custo de cada subsequente transferência.

• O custo de mover um único processo é frequentemente elevado quando comparado com o custo de mover processos adicionais. Exemplo:

$$CT_c = 24\,\mu\text{s} \qquad CT_s = 64\,\mu\text{s} \qquad CT_m = 40\,\mu\text{s}$$

• Para reduzir o pessimismo de assumir que este custo é consumido em cada execução da rotina do relógio esta carga pode ser espalhada ao longo de um conjunto de interrupções de relógio. Isto é válido se o menor período de todos os processos, T_{min}, for maior que o período do relógio, T_{clk}. Seja M definido por:

$$M = \left\lceil \frac{T_{min}}{T_{clk}} \right\rceil$$

• Se $M > 1$ então o custo da rotina do relógio pode ser repartida/atribuída ao longo de M execuções. Nesta situação se $M \leq N$ a rotina do relógio é modelada como um processo com período T_{min} e tempo de cálculo:

$$C'_{clk} = M(CT_c + CT_s) + (N - M)CT_m$$

• Tomando em conta o relógio, a equação (13) pode ser escrita da seguinte forma:

$$R_i = CS_1 + C_i + B_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS_1 + CS_2 + C_j) + \sum_{k \in \Gamma_s} \left\lceil \frac{R_i}{T_k} \right\rceil IH_k + \left\lceil \frac{R_i}{T_{min}} \right\rceil C'_{clk}$$

• Para obter aperfeiçoamentos adicionais no modelo é necessária uma representação mais exacta da execução da rotina de relógio. Por exemplo usando apenas CT_c e CT_s no modelo do relógio podemos obter a seguinte equação:

$$\begin{aligned} R_i &= CS_1 + C_i + B_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS_1 + CS_2 + C_j) + \sum_{k \in \Gamma_s} \left\lceil \frac{R_i}{T_k} \right\rceil IH_k + \\ &\quad + \left\lceil \frac{R_i}{T_{clk}} \right\rceil CT_c + \sum_{g \in \Gamma_p} \left\lceil \frac{R_i}{T_g} \right\rceil CT_s \end{aligned}$$

onde $T_g \rightarrow$ tempo mínimo entre activações do processo g;
$\Gamma_p \rightarrow$ conjunto de processos periódicos.

Fiabilidade e Tolerância a Falhas

• Os sistemas embebidos, por estarem a implementar de forma crescente funções previamente realizadas por operadores humanos ou por métodos de controlo analógicos, têm requisitos de fiabilidade muito maiores que outros sistemas computacionais.

• Esta necessidade de fiabilidade pode ter a ver com:

  • colocação de vidas em risco (e.g. aviões, automóveis, "pacemakers", centrais nucleares, industria química, etc);

  • possibilidade de consequências económicas desastrosas (e.g. "bank accounting", produção industrial, etc);

  • consequências ambientais desastrosas (e.g. centrais nucleares, industria química).

• A fiabilidade de um sistema computacional depende de:

  • componentes físicos;

  • software que controla as operações.

Falhas

• Os componentes de "hardware" estão sujeitos a falhas devidas a defeitos de fabrico ou ao envelhecimento.

• O software está sujeito apenas a falhas devidas a defeitos de projecto e de desenvolvimento.

• Hecht (90) estabeleceu os seguintes valores típicos para caracterizar os defeitos de software: em 1 milhão de linhas de código há cerca 20000 "bugs"

  • 90% descobertos na fase de testes;

  • 200 descobertos no primeiro ano de utilização;

  • 180 ficam por detectar e podem ser despoletados em qualquer altura ou nunca chegarem a ser descobertos.

Origem das Faltas que Podem Levar um Sistema a Falhar:

• Especificação inadequada - tem sido sugerido que a grande maioria das faltas de software se ficam a dever a especificação inadequada;

• Faltas introduzidas por erros de projecto de componentes/módulos do sistema;

• Erros no processador ou noutros componentes eléctricos do sistema embebido;

• Erros devidos a transitórios ou interferências permanentes no subsistema de suporte de comunicação no sistema.

• Um sistema *confiável* pode ser definido informalmente de diversas formas:

  • Um sistema do qual o utilizador possa depender;

  • Um sistema que "passe o teste do tempo";

  • Um sistema cujo tempo "em baixo" é inferior a um determinado limiar;

  • Um sistema do qual não há erros catastróficos conhecidos;

  • Um sistema cujos resultados são previsíveis (i.e. sistema determinístico);

  • Um sistema robusto - com a capacidade de recuperar graciosamente de erros.

• Outras caracterizações informais utilizadas em sistemas de tempo-real:

  • Determinismo dos eventos;

  • Determinismo temporal;

  • Carga do processador "razoável";

  • Carga da memória "razoável".

Fiabilidade, Falha, e Falta

Fiabilidade (Randell): "medida do sucesso com o qual o sistema se mantém de acordo com as especificações feitas para o seu comportamento".

Falha (Randell): "quando o comportamento do sistema se desvia daquilo que foi especificado diz-se que ocorreu uma falha".

• Fiabilidade e falhas são conceitos associados ao comportamento externo do sistema. Falhas resultam de problemas internos inesperados que se manifestam no comportamento externo do sistema. Estes problemas são chamados de **erros** e as suas causas **faltas**.

Falta: causa mecânica, eléctrica ou algorítmica que leva ao surgimento de erros no sistema e consequentemente a falhas.

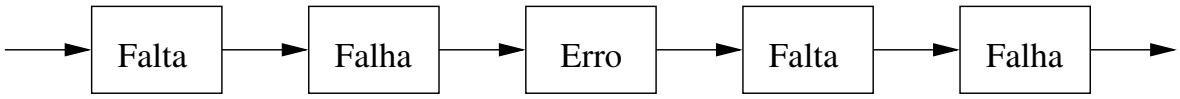
• Quando visto em termos de transições de estados, um sistema pode ser caracterizado por um conjunto de estados *externos* e *internos*.

• Um estado externo que não faz parte da especificação de comportamento do sistema é visto como uma falha do sistema.

• O sistema é composto por um conjunto de componentes cada um com os seus próprios estados que contribuem para o comportamento externo do sistema. A combinação dos estados destes componentes é designada estado interno do sistema.

• Um estado interno que não faça parte das especificações é designado por erro e diz-se que o componente que produziu a transição de estado ilegal está em falta.

• Cada componente de um sistema pode ser considerado um sistema. Uma falha num sistema pode levar a uma falta noutro o que pode resultar num erro e potencialmente numa falha desse sistema. Podem assim ocorrer falhas em cadeia:



• Podemos distinguir **três tipos de faltas**:

- **Faltas transitórias:** começam num instante particular, permanecem no sistema por algum tempo e depois desaparecem. E.g. um componente de hardware sujeito a um campo eléctrico de interferência. Após desaparecer a perturbação, o mesmo acontece com a falta, mas não necessariamente o erro que foi induzido. Muitas faltas em sistemas de comunicação são transitórias.
- **Faltas permanentes:** começam num instante particular e permanecem no sistema até que sejam reparadas: e.g. um fio partido ou um erro de projecto de software.
- **Faltas intermitentes:** ocorrem de tempos a tempos. Por exemplo, um componente de hardware sensível ao calor pode funcionar durante algum tempo, parar de trabalhar, arrefecer e começar novamente a trabalhar.

Tipos de Falhas

• Um sistema pode falhar de muitas formas diferentes. Se um sistema X que é usado para implementar um sistema Y falha de forma diferente do que foi assumido como "falhas espectáveis" para o projecto de Y, então como consequência Y pode também falhar.

• Os tipos de falhas podem ser classificados de acordo com o impacto que estas têm nos serviços que um sistema fornece:

- Falha por valor: o valor associado com o serviço está errado.
- Falha temporal: o serviço é fornecido no instante errado. Várias hipóteses:
  - Demasiado cedo;
  - Demasiado tarde (também designado por erro de desempenho);
  - Infinitamente tarde (falha de omissão);

• Combinações de falhas temporais e falhas por valor são frequentemente designadas por falhas arbitrárias.

• Falha de incumbência: quando um serviço não esperado é fornecido. É muitas vezes difícil de distinguir uma falha simultaneamente no domínio do valor e do tempo de uma falha de incumbência seguida de uma falha de omissão.

• Dada a classificação acima sobre os tipos de falhas, podemos considerar as seguintes hipóteses sobre como pode um sistema falhar:

- **Falha não controlada** - o sistema pode produzir erros arbitrários tanto no domínio temporal como no dos valores;
- **Falha por atraso** - serviços correctos no domínio dos valores mas podem surgir demasiado tarde;
- **Falha silenciosa** - o sistema produz serviços correctos tanto no domínio dos valores como do tempo até que falha;
- **Falha parada** - o sistema entra em falha silenciosa mas permite que os outros sistemas detectem que o sistema entrou no estado de falha silenciosa;
- **Falha nunca** - o sistema produz sempre serviços correctos nos domínios dos valores e do tempo.

Fiabilidade do Software

• Permite uma caracterização menos ambígua do que tentar definir o que é "software confiável".

• Sendo S um sistema de software e T o instante em que o software falha, então a fiabilidade do software no instante t, $r(t)$, é definida por:

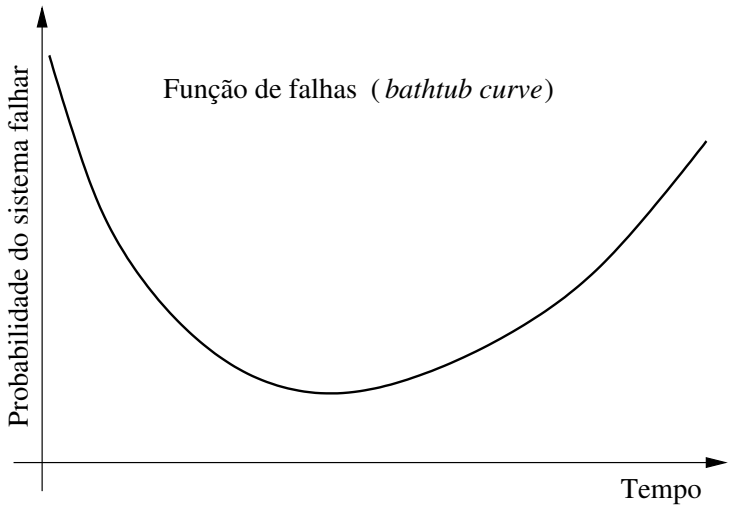
$$r(t) = P(T > t)$$

o que representa a probabilidade de um sistema de software funcionar sem falhas durante um determinado período de tempo. Um sistema com $r(t) = 1$ nunca falha.

• Função de falhas: probabilidade de que o software falhe no instante t. Dois modelos para esta função:

1. Função exponencial decrescente do tempo versus probabilidade de falha. Interpretação: à medida que o tempo passa, serão introduzidas novas versões do software e serão descobertos e reparados erros.

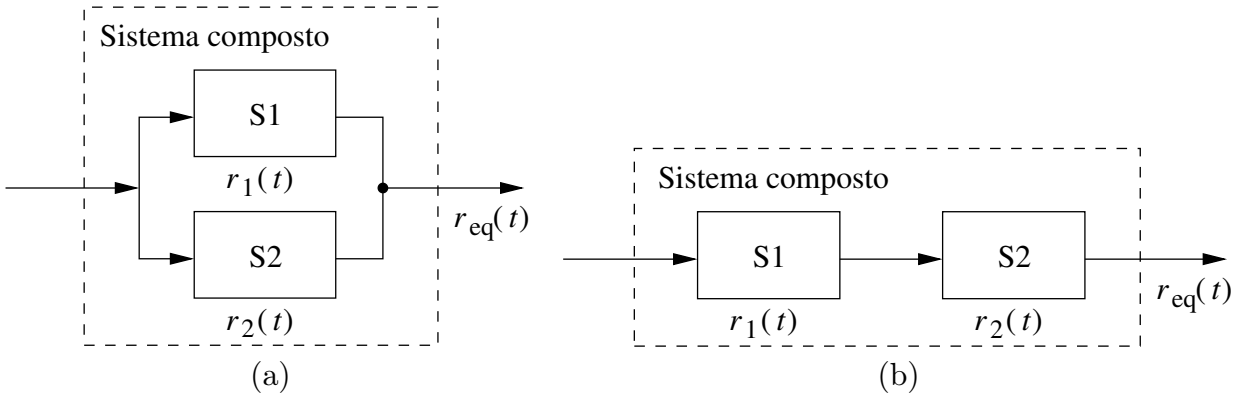
2. A "curva da banheira" é frequentemente utilizada para descrever a função de falhas de componentes de hardware - probabilidade elevada cedo devido a defeitos de fabrico e tarde devido a envelhecimento dos componentes. Também pode ser usada para descrever a probabilidade de falhas do S/W. Aqui → falhas elevadas para tempos elevados devido a erros introduzidos quando se efectuam pequenas correcções, modificações, acrescentos, ou devido à sujeição a um maior esforço por utilizadores mais experientes.



Sistemas Compostos: Fiabilidade de um Modelo de Blocos

• A fiabilidade de um sistema composto por vários blocos interligados em série ou paralelo pode ser calculada utilizando regras básicas de probabilidade.

• Utiliza-se normalmente a aproximação de que cada bloco tem uma função de falhas independente das restantes, ou seja, se um bloco falha isto não implica que qualquer outro deva falhar. Isto nem sempre é verdade pois por exemplo quando existem recursos partilhados, se um processo falha pode fazer com que outro com quem partilhe por exemplo uma zona de memória partilhada vá falhar também.



• Suponhamos dois sistemas com funções de fiabilidade $r_1(t)$ e $r_2(t)$. Se estes estiverem ligados em paralelo, o sistema composto apenas falhará se ambos falharem.

• Logo a função de falhas do sistema composto, $f_{\text{eq}}(t) = 1 - r_{\text{eq}}(t)$, será o produto das funções de falhas dos dois subsistemas:

$$1 - r_{\text{eq}}(t) = (1 - r_1(t))(1 - r_2(t))$$

• Resolvendo em ordem a $r_{\text{eq}}(t)$ vem:

$$r_{\text{eq}}(t) = r_1(t) + r_2(t) - r_1(t)r_2(t)$$

• Dado que as funções de fiabilidade assumem valores entre 0 e 1, a fiabilidade de um sistema composto por blocos paralelos é sempre igual ou superior à fiabilidade de cada bloco individual.

• No caso dos sistemas estarem ligados em série, o sistema composto falha se 1 ou ambos falharem:

$$1 - f_{\text{eq}}(t) = (1 - f_1(t))(1 - f_2(t))$$

ou seja

$$1 - r_{\text{eq}}(t) = 1 - r_1(t) + 1 - r_2(t) - (1 - r_1(t))(1 - r_2(t))$$

de onde vem

$$r_{\text{eq}}(t) = r_1(t)r_2(t)$$

• Neste caso $r_{\text{eq}}(t)$ é inferior à fiabilidade de ambos os blocos que compõem o sistema.

Métrica de McCabe

• Alguns estudiosos acreditam que a fiabilidade do software pode ser estimada *a priori* a partir das características do código.

• A métrica de McCabe é baseada na complexidade do fluxo de controlo do sistema.

• O software é descrito por um grafo direccionado onde:

  • cada bloco de código sequencial é representado por um nodo (node - nó);

  • cada mudança de fluxo síncrona é representada por um arco (edge) direccionado;

  • cada processo ou tarefa é representada por um grafo diferente.

• A métrica de McCabe não considera a possibilidade de fluxo de controlo assíncrono.

• Contabilizando o número de arcos, e, o número de nodos n, e o número de tarefas, p, a complexidade é definida como

$$C = e - n + 2p$$

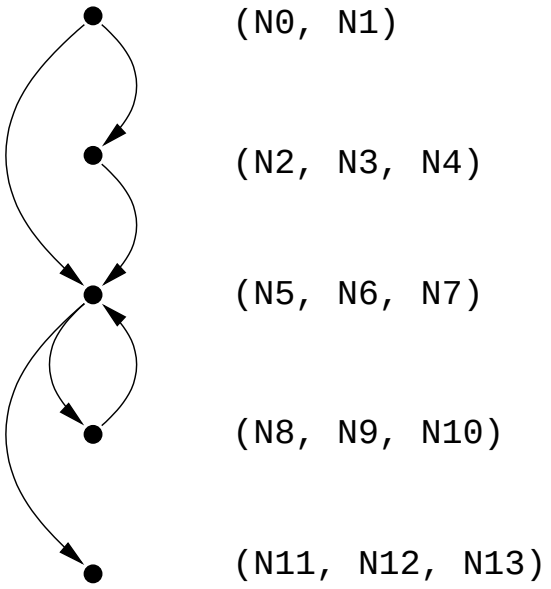
• Segundo McCabe a complexidade, C, do grafo de controlo de fluxo reflecte a dificuldade de compreender, testar e manter o software;

• Segundo McCabe módulos de software bem estruturados apresentam valores de complexidade entre 3 e 7.

• A partir de evidência empírica estabeleceu-se considerar 10 como o valor máximo aceitável para C em cada módulo.

• Exemplo com $C = 6 - 5 + 2 = 3$:

```
N0  euclid(int m, int n){
      int r;
N1    if(n>m){
N2      r=m;
N3      m=n;
N4      n=r;
N5    }
N6    r=m%n;
N7    while(r!=0){
N8      m=n;
N9      n=r;
N10     r=m%n;
N11   }
N12   return n;
N13 }
```



Grafo de fluxo de controlo

• Existem outras métricas que levam em linha de conta a possibilidade de fluxos de controlo assíncronos.

Tornar os Sistemas Confiáveis

• Podem-se distinguir duas aproximações de projecto que poderão ajudar a melhorar a fiabilidade dos sistemas:

• A prevenção de faltas tenta eliminar qualquer possibilidade da ocorrência de falhas antes do sistema se tornar operacional - e.g. durante o projecto e desenvolvimento.

• A tolerância a faltas consiste em fazer com que um sistema continue a funcionar mesmo na presença de faltas.

• Ambos os métodos se destinam à obtenção de sistemas cujo comportamento é bem definido em termos de falhas.

Prevenção de Faltas

• Existem duas etapas na prevenção de faltas: (1) **evitamento de faltas** e (2) **remoção de faltas** cuja introdução não tenha sido possível de evitar.

• O evitamento de faltas consiste em tentar limitar a introdução de componentes defeituosos durante a construção do sistema. No caso do hardware isto pode envolver:

- o uso de componentes mais fiáveis, o que pode ter implicações no custo ou mesmo no desempenho;
- o uso de técnicas apropriadas para obter a melhor forma de efectuar as ligações entre os componentes e a montagem do sistema;
- isolar o hardware de forma a evitar interferências conhecidas.

• Os componentes de software de grandes sistemas embebidos são hoje em dia muito mais complexos do que os componentes de hardware. Apesar do S/W não sofrer deterioração por fadiga, é praticamente impossível criar programas completamente livres de defeitos. No entanto a qualidade do S/W criado pode ser melhorada através de:

- especificação rigorosa dos requisitos;
- metodologias de projecto adequadas;
- utilização de linguagens de programação que possibilitem uma maior abstracção de dados e modularidade de programas.
- o uso de ambientes de engenharia de software para ajudar a gerir a complexidade.

• A utilização de todas estas técnicas que acabam de ser enunciadas permite obter S/W mais fiável. No entanto, vão estar sempre presentes defeitos no sistema construído. Em particular podem existir erros de projecto tanto em componentes de H/W como de S/W.

• É portanto necessário passar à segunda fase que é a da remoção de faltas ou dos defeitos que conduzem ao aparecimento destas.

• Embora existam técnicas como revisões de projecto, verificação de programas, e inspecção de código para detectar a presença de erros, a forma mais utilizada para a remoção de faltas é submeter o sistema a um conjunto de testes.

• Infelizmente a maior parte das vezes os testes de sistemas não podem ser exaustivos nem remover todas as potenciais faltas. Por isso existem as seguintes limitações na utilização de testes:

- Um teste permite apenas indicar a presença de faltas e não a sua ausência;
- Muitas vezes é impossível efectuar os testes em condições reais; Muitos testes são efectuados em simuladores e é difícil garantir que a simulação é precisa/realista;
- Se existirem erros nas especificações, estes só poderão ser detectados durante o funcionamento normal do sistema e não através de testes que sejam elaborados de acordo com as especificações.

• Em alguns sistemas é impossível efectuar a remoção de erros após a activação do sistema. Um exemplo disso são as sondas espaciais.

Tolerância a Faltas

• Apesar de todas as técnicas de teste e verificação, os componentes de hardware falham. Por isso a abordagem baseada em prevenir a ocorrência de falhas não tem sucesso quando:

- A frequência ou a duração das reparações é inaceitável;
- O sistema não está acessível para operações de manutenção e reparação.

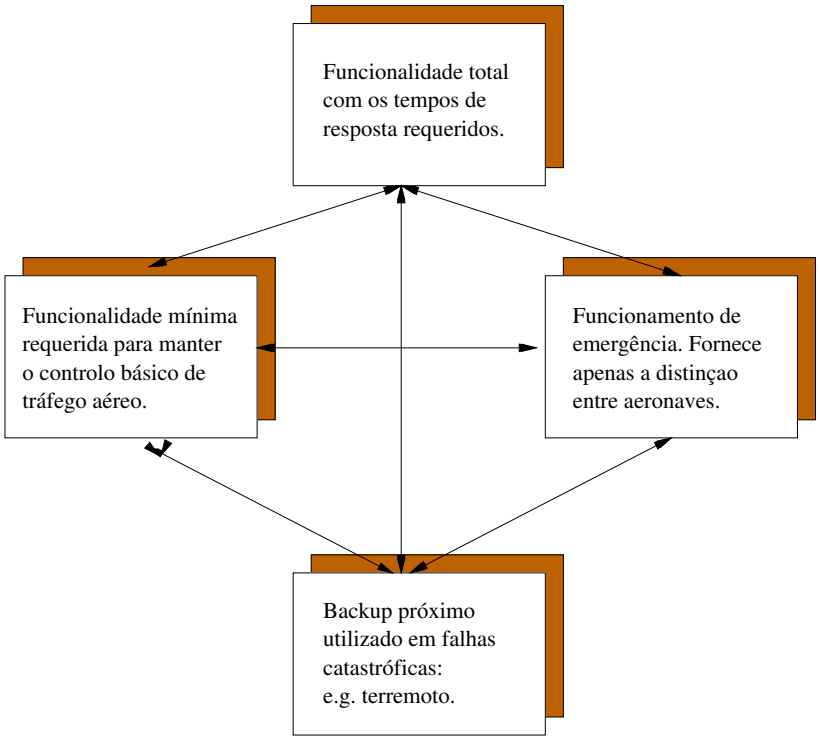
• A alternativa é a **tolerância a faltas**.

• Podem-se definir/fornecer diversos **níveis de tolerância a faltas**:

- **Total tolerância a faltas:** o sistema continua a funcionar na presença de falhas, ainda que por um período de tempo limitado, sem perda significativa de funcionalidade ou desempenho.
- **Degradação controlada/graciosa ("fail soft")**: o sistema continua a funcionar na presença de erros, sendo aceitável uma degradação temporária de funcionalidade ou desempenho durante o período de reparação.
- **Seguro nas falhas:** o sistema mantém a sua integridade mesmo durante uma paragem temporária no funcionamento.

• O nível requerido de tolerância a faltas depende da aplicação. Embora em teoria a maioria dos sistemas de segurança crítica necessitem de total tolerância a faltas, na prática muitos implementam degradação controlada (e.g. aviões de combate podem sofrer danos físicos → vários níveis de degradação controlada).

• Com sistemas de operação contínua (requisito de disponibilidade elevada) é forçosa a degradação controlada dado que não se consegue alcançar total tolerância a faltas durante períodos infinitos.
• Exemplo: degradação controlada e recuperação num sistema de tráfego aéreo.



• Nalgumas situações pode ser simplesmente necessário forçar a paralisação do sistema num estado seguro. Exemplo: num Airbus A310 os computadores de controlo dos "slats" e "flaps" ao detectarem um erro na aterragem colocam valores simétricos em ambas as asas e depois desligam-se (só valores assimétricos seriam perigosos na aterragem).

• Tradicionalmente, o projecto de sistemas tolerantes a faltas baseava-se nos seguintes pressupostos:

- Os algoritmos não contêm erros de projecto.
- Todos os tipos de falhas dos diversos componentes são bem conhecidos.
- Todas as possíveis interacções entre o sistema e o ambiente que o rodeia são conhecidos, ou seja é possível fazer a sua antevisão.

Redundância

• Todas as técnicas de tolerância a faltas requerem a introdução de novos elementos no sistema, seja para detectarem a presença de faltas ou para efectuar a recuperação das mesmas.

• Estes componentes podem considerar-se redundantes pois não são necessários para o funcionamento de um sistema perfeito. Este mecanismo é chamado de **redundância protectiva**.

• Um dos objectivos das técnicas de tolerância a faltas é o de minimizar a redundância procurando maximizar a fiabilidade do sistema, de acordo com as restrições de custo e desempenho.

• Deve haver cuidado na estruturação de sistemas tolerantes a faltas → os componentes adicionais introduzidos aumentam inevitavelmente a complexidade do sistema o que pode levar ao efeito inverso do desejado, i.e. à diminuição da fiabilidade. Exemplo: o lançamento do primeiro "space shuttle" foi abortado devido a uma dificuldade de sincronização entre computadores replicados.

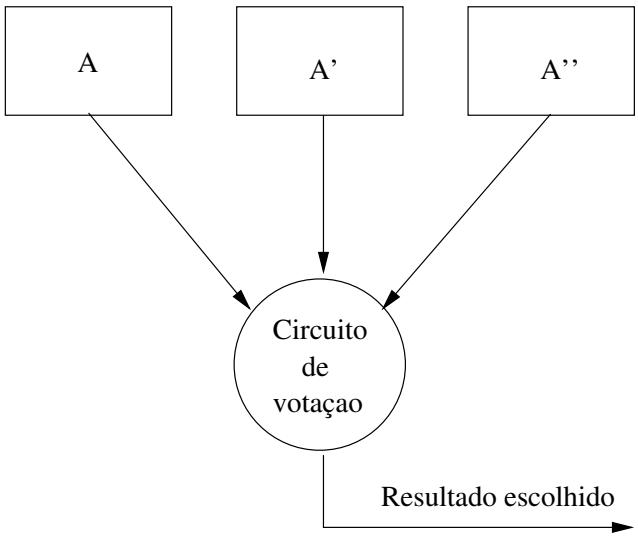
• Portanto: para reduzir os problemas associados com a interacção entre componentes redundantes, torna-se aconselhável separar os componentes tolerantes a falhas do resto do sistema.

Hardware Tolerante a Faltas

• Consideram-se duas formas básicas de redundância: redundância estática (dissimulante) e redundância dinâmica.

• No caso da redundância estática, são introduzidos novos componentes no sistema por forma a esconder os efeitos das faltas.

• Um exemplo de redundância estática é a redundância tripla modular (RTM) que consiste em três subcomponentes idênticos que recebem as mesmas entradas e as saídas estão ligadas a um circuito escrutinador de voto por maioria. O escrutinador compara as saídas de todos os componentes e se uma diferir das outras duas então é ignorada, sendo considerada válida a saída que seja comum a pelo menos dois módulos.



• Assume-se que a falta não é devida a um aspecto comum dos sub-componentes (e.g. um erro de projecto) mas é transitória ou devida à deterioração de um componente.

• Para dissimular faltas de mais do que um componente é necessária mais redundância. Usa-se portanto o termo redundância N Modular (RNM) para caracterizar este método.

• A redundância dinâmica é fornecida dentro dos componentes que indicam eles próprios se a sua saída está ou não errada. Desta forma, ao contrário da dissimulação de erros, trata-se de um mecanismo de detecção de erros.

• Exemplos:

  • Bits de paridade nas memórias;

  • CRCs nos discos magnéticos;

  • checksums utilizados nas comunicações.

Tolerância a Faltas de Software

• No caso da tolerância a faltas de software podem-se utilizar duas formas básicas.

• Uma semelhante à tolerância estática utilizada no hardware, a que se chama programação por N-versões.

• A segunda é baseada na detecção e recuperação de erros o que é semelhante à redundância dinâmica no sentido em que os procedimentos de recuperação entram em acção após a detecção de um erro.

Programação por N-Versões

• O sucesso no uso de RTM no hardware motivou o uso de uma aproximação semelhante no software.

• No entanto, como o software não envelhece, esta aproximação apenas pode servir para detectar erros de projecto.

• A programação por N-versões é definida como a geração independente de N ($N > 2$) programas funcionalmente equivalentes produzidos a partir das mesmas especificações.

• A geração independente de N programas significa que N indivíduos ou grupos produzem as necessárias N versões do software sem qualquer tipo de interacção. Por esta razão a programação por N-versões é frequentemente designada por diversidade de projecto.

• Uma vez projectados e escritos, os programas correm concorrentemente utilizando as mesmas entradas, e as suas saídas são analisadas por um processo driver. Em principio os resultados deveriam ser iguais, mas na prática pode haver alguma diferença, caso em que se assume como correcto um resultado que seja consensual, se um tal resultado existir.

• A programação por N-versões é baseada nas seguintes duas hipóteses:

(1) Um programa pode ser especificado de forma completa, consistente e não ambígua;

(2) Programas desenvolvidos independentemente falharão de forma independente, i.e. não existe qualquer relação entre as faltas de uma versão e as faltas de qualquer das outras versões. Para garantir esta independência dos erros é conveniente cada grupo utilizar:

- Diferentes ambientes de desenvolvimento de software (de diferentes "software houses");
- Linguagens de programação diferentes (C, ADA, PASCAL, etc);
- Se não for possível usar linguagens diferentes, pelo menos usar compiladores diferentes.

• Para proteger contra faltas físicas as N versões devem ser distribuídas por máquinas separadas, que comunicam através de linhas de comunicação tolerantes a faltas.

• Exemplo: no sistema de controlo de voo do Boeing 777 foi usado um único programa em ADA mas este foi compilado por 3 compiladores diferentes e cada versão é executada num computador independente para obter diversidade.

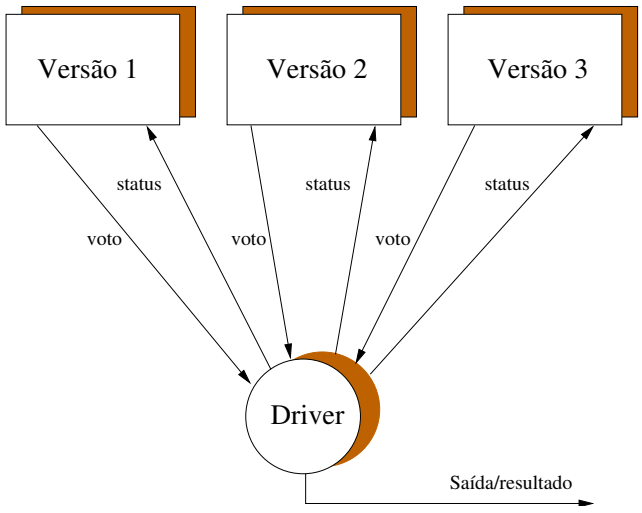
• A programação por N-versões é controlada por um processo driver que é responsável por:

- invocar cada uma das versões;
- esperar que estas completem as operações necessárias;
- comparar os resultados e escolher a acção a desencadear.

• Embora se considere que os programas são lançados e o driver espera que estes terminem para comparar os resultados, em alguns sistemas embebidos isto não é verdade pois todos estes processos executam continuamente. O driver e as N versões devem portanto comunicar durante o curso da sua execução.

• O modo de comunicação e sincronização com o driver dependerá da linguagem de programação e do respectivo modelo de concorrência. Se forem usadas diferentes linguagens para diferentes versões, então os modos de comunicação e sincronização serão usualmente fornecidos por um sistema operativo de tempo-real.

• A relação entre as N-versões e o driver para um sistema com $N = 3$ é ilustrada na figura seguinte:



• A interacção entre as versões e o programa driver é especificada nos requisitos para as versões e consiste de 3 componentes:

- Vectores de comparação: saídas (ou votos) produzidas pelas versões e atributos associados com o seu cálculo (e.g. como foi obtido o resultado? aquisição de dados recentes ou estimação recente?). Estas saídas devem ser comparadas pelo driver.
- Indicadores do estado da comparação: são comunicados do driver para as versões e indicam as acções que cada versão deve realizar como resultado da comparação. Estas acções dependem do resultado da comparação: se os votos estavam de acordo e se foram distribuídos a tempo. Algumas possibilidades são:
  – continuar,
  – terminar,
  – continuar após modificar o seu voto para o da maioria.
- Pontos de comparação: são pontos onde cada uma das versões deve indicar o seu voto ao processo driver. A frequência pela qual são efectuadas comparações é designada por **granularidade** da tolerância a falhas.

• Com granularidade elevada (i.e. comparações infrequentes) temos menores penalizações de desempenho provocados pelo processo de votação e a possibilidade de maior independência no projecto das versões; contudo teremos possivelmente uma maior divergência de resultados devido ao maior número de passos efectuados entre comparações.

• Com granularidade fina teremos possivelmente menor divergência de resultados nas comparações. Por outro lado existirão nos programas elementos estruturais comuns a um nível de detalhe mais fino (os programas têm que passar por um maior número de etapas de votação comuns), o que reduz o grau de independência entre as versões. Uma elevada frequência de comparações também aumenta as penalizações de desempenho associadas à votação.

Comparação de Votos

• Na programação com N-versões é essencial a eficiência e facilidade com a qual o escrutinador compara os resultados e estabelece se há ou não discordância.

• Em aplicações que manipulam texto ou realizem aritmética inteira, existe normalmente um resultado correcto. Nestes casos o driver pode facilmente comparar os votos das diferentes versões e escolher a decisão maioritária.

• No caso das votações envolverem (dados que resultem de cálculos com) números reais é improvável que diferentes versões produzam exactamente os mesmos resultados. Isto pode dever-se a: representação inexacta dos números pelo hardware e diferentes sensibilidades dos dados em cada um dos algoritmos. As técnicas usadas para comparar estes tipos de resultados são chamadas de votação inexacta. Uma técnica simples é realizar uma verificação de limites em torno de uma estimativa anterior ou de um valor mediano de todos os N resultados.

• Outro problema com a comparação de votos pode ocorrer quando existirem naturalmente múltiplas soluções para o mesmo problema. Por exemplo uma equação quadrática tem duas soluções. É possível discrepância apesar de não ter ocorrido nenhuma falta.

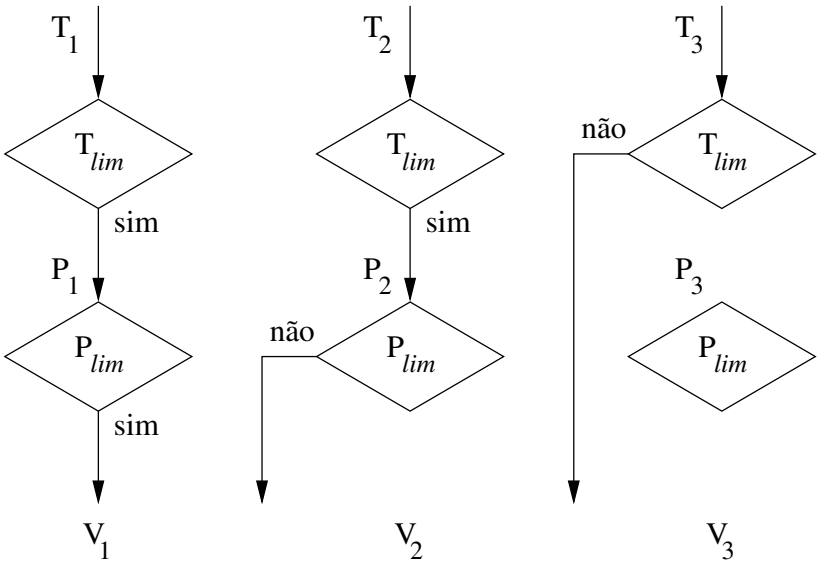
• O problema da comparação consistente: está associado à aritmética de precisão finita e ocorre quando uma aplicação tem de realizar uma comparação em relação a um certo valor finito dado na especificação; o resultado da comparação determina o caminho a seguir e em última instância o voto da versão.

• Exemplo: aplicação de controlo de processos com um sensor de temperatura e outro de pressão. Para manter a integridade do sistema, este toma acções apropriadas em função dos valores dos sensores. Suponhamos que quando qualquer destes sensores ultrapassa algum valor limiar deve ser desencadeada uma determinada acção correctiva.

• Suponhamos que existem três versões (V_1, V_2, V_3) cada uma das quais deve ler os sensores e votar uma saída (não existe comunicação entre as versões até que estas votem). Como resultado do uso de aritmética de precisão finita, cada versão irá calcular diferentes valores para a temperatura (T_1, T_2, T_3) e para a pressão (P_1, P_2, P_3). Se assumirmos que os valores limiar são T_{lim} para a temperatura e P_{lim} para a pressão, então o problema da comparação consistente ocorre quando ambas as leituras se situam em torno dos seus valores limite.

• Suponhamos que T_1 e T_2 estão ligeiramente abaixo de T_{lim} e T_3 ligeiramente acima. Neste caso V_1 e V_2 seguirão a sua execução normal e V_3 tomará alguma acção correctiva. Se as versões V_1 e V_2 seguirem agora para outro ponto de comparação é possível que P_1 esteja ligeiramente acima e P_2 ligeiramente abaixo de P_{lim}. Neste caso as 3 versões seguirão trajectórias de execução diferentes e produzirão portanto resultados diferentes mas cada um dos quais é válido.

• Surge então ao escrutinador o problema de qual resultado adoptar. À primeira vista parece apropriado usar técnicas de comparação inexacta e considerar que os valores são iguais se não diferirem mais do que uma certa tolerância Δ; mas o problema repete-se se os valores de diferença estiverem próximos de $\pm\Delta$.



Principais Questões em Programação com N-versões

• O sucesso da utilização de versões múltiplas para obter redundância estática e assim aumentar a fiabilidade de um sistema, depende de vários factores. Entre eles podemos apontar os seguintes:

• Especificação inicial: muitos dos defeitos presentes no software advêm de uma especificação inicial inadequada. Um erro de especificação irá manifestar-se em todas as N versões da implementação. As especificações devem ser completas, consistentes, compreensíveis e não ambíguas.

• Independência nos esforços de projecto: a utilização de esforços independentes mostrou em alguns casos que a maior parte das falhas não se repetiam nos vários módulos. Isto no entanto requer uma grande disciplina e uma especificação de requisitos muito completa. Este ponto tem sido algo controverso pois alguns autores advogam que se a especificação for muito complexa, poderá não ser correctamente compreendida por todas as equipas de desenvolvimento.

• Orçamento adequado: o custo de desenvolvimento de um sistema embebido é predominantemente do software. Por isso, se num sistema se pretender utilizar 3 versões isso implicará aproximadamente triplicar o custo final. Logo a utilização de versões múltiplas está muitas vezes dependente do orçamento previsto para o desenvolvimento do produto. Por outro lado, pode não ser claro se se poderia alcançar um sistema mais fiável se os recursos disponíveis para construir N versões fossem utilizados para produzir uma versão.

• Tem-se também mostrado que por vezes é difícil encontrar algoritmos de votação inexactos, e a menos que se tome cuidado com o problema da comparação consistente, os votos poderão diferir mesmo na ausência de faltas.

• A programação por N-versões tem sem dúvida um papel importante na produção de sistemas confiáveis, no entanto devem ser usados cuidados especiais no seu desenvolvimento e deve ser usada em conjunto com outras técnicas - como as que abaixo serão abordadas.

Redundância Dinâmica de Software

• A programação de N-versões é um método de redundância estática, em que as falhas no interior de cada módulo são escondidas do exterior. A designação estática advém do facto de serem fixas as relações entre versões e entre estas e o driver; e os diversos módulos são activados quer ocorram faltas ou não.

• Com a **redundância dinâmica** os componentes redundantes entram em operação apenas quando é detectado um erro. Esta técnica de tolerância a faltas tem quatro fases constituintes:

(1) Detecção de Erros: a maior parte das faltas manifestar-se-ão eventualmente na forma de um erro; nenhum esquema de tolerância a faltas pode ser utilizado até esse erro ser detectado.

(2) Avaliação e Confinamento de Danos: quando é detectado um erro tem que se decidir qual a extensão do sistema que foi corrompida (diagnóstico de erros); o atraso entre a ocorrência da falta e a manifestação do erro associado significa que o erro se pode ter espalhado pelo sistema.

(3) Recuperação de Erros: o objectivo das técnicas de recuperação de erros é transformar o estado do sistema corrompido levando-o a um estado onde se possa continuar com a operação normal (possivelmente com funcionalidade degradada). Este é um dos aspectos mais importantes da tolerância a faltas.

(4) Tratamento de Faltas e Funcionamento Contínuo: um erro é um sintoma de uma falta; mesmo que o dano já tenha sido reparado, a falta pode continuar a existir e, portanto, o erro pode reaparecer a menos que seja efectuada alguma forma de manutenção.

• Embora estas quatro fases sejam discutidas no contexto da redundância de software elas podem claramente ser aplicadas à programação por N-versões: a detecção de erros é fornecida pelo driver que realiza a verificação dos votos; a avaliação dos danos não é necessária porque as versões são independentes; a recuperação de erros envolve abandonar o resultado errado; o tratamento da falta consiste simplesmente em ignorar a versão que produziu o resultado errado. Contudo se todos os votos forem diferentes, então pode-se detectar o erro mas não existe método de o recuperar.

• Vamos ver de seguida cada uma das quatro fases da redundância dinâmica de software em mais detalhe.

Detecção de Erros

• A efectividade de qualquer sistema tolerante a faltas depende da efectividade das suas técnicas de detecção de erros. Podem-se identificar duas classes de técnicas de detecção de erros:

(1) Detecção de erros no ambiente onde a aplicação funciona. Estes incluem os que são detectados pelo hardware tais como "execução de instrução ilegal", "erros de funções aritméticas" tais como overflow, divisão por zero, etc, ou a "violação de mecanismos de protecção". Incluem também os erros detectados pelo sistema de suporte de execução da linguagem de programação usada: e.g. "violação dos limites da tabela", "referência a um ponteiro nulo", etc.

(2) Detecção pela aplicação: estes são os erros detectados pela própria aplicação. A maioria das técnicas que podem ser usadas pela aplicação enquadram-se dentro das seguintes categorias:

• Verificações por replicação: equivalente à programação por N-versões onde se verifica a consistência dos resultados (duas versões redundantes).

• Verificações de temporização: podem-se identificar dois tipos. (1) O primeiro é um mecanismo de temporização designado por "**watchdog timer**" pode por exemplo ser implementado como um processo que, se durante um certo período não receber um sinal de um componente, assume que ocorreu um erro nesse componente. Esse componente de software tem que enviar continuamente um sinal ao temporizador para indicar que está a funcionar correctamente. (2) Quando as metas temporais das respostas são importantes, pode-se verificar se ocorrem faltas por não cumprimento das metas. Quando o escalonamento de metas é efectuado pelo sistema operativo a verificação do cumprimento das metas pode-se considerar parte do ambiente de operação.
Como se compreende facilmente, os testes de temporização não garantem que a resposta de um módulo seja a correcta, apenas garantem que esta foi produzida no tempo certo. Terão portanto que ser usadas em conjunto com outras técnicas de detecção de erros.

• Verificação por reversibilidade: em casos em que existe uma relação de um-para-um entre a entrada de um componente e a sua saída, é possível calcular o valor de entrada que poderia ter dado origem à saída observada. Depois basta fazer a comparação entre o valor de entrada real e o valor calculado para verificar se coincidem.

• Verificação por codificação: os códigos de detecção e recuperação de erros são muitas vezes usados para verificar a corrupção de dados. Como exemplos temos os CRCs no armazenamento em discos e os ckecksums na transmissão de dados. Quando os dados são recebidos, os códigos são recalculados e comparados com os que foram armazenados ou transmitidos juntamente com os dados.

• Verificações de razoabilidade: são baseadas no conhecimento do projecto interno e construção do sistema. É verificado se o estado dos dados ou o valor de um objecto é razoável tendo em conta a sua utilização. Em algumas linguagens de programação muita da informação necessária para efectuar este tipo testes pode ser fornecida pelo programador como informação associada ao tipo dos objectos. Um destes testes poderá ser a verificação se se pretende calcular um número real que seja a raiz quadrada de um número de valor negativo. Outro exemplo: definição de objectos inteiros restringidos a certas gamas de valores e verificação de violações de gamas pelo sistema de suporte a execução da linguagem.
Por vezes são incluídas nos componentes de software verificações de razoabilidade explícitas. Estas são designadas de asserções que são expressões lógicas que durante a execução são avaliadas como verdadeiras se não houver erro.

• Verificações estruturais: são usadas para verificar a integridade de estruturas de dados tais como listas ligadas e filas. Podem consistir em contagens do número de elementos num objecto, ponteiros redundantes, ou informação adicional de estado.

• Verificações de razoabilidade da dinâmica: nas saídas de alguns sensores ou controladores digitais, existe usualmente uma relação entre duas saídas consecutivas. Pode-se assumir um erro se uma nova saída for demasiadamente diferente de uma anterior. Por exemplo uma leitura de velocidade de um automóvel amostrada com um período de 10ms não pode passar de 100km/h para 0km/h ou para 130km/h num único intervalo de amostragem.

Avaliação e Confinamento de Danos

• Como pode haver um atraso entre a ocorrência de uma falta e a detecção do erro correspondente, é necessário verificar qualquer dano que possa ter ocorrido. O tipo de erro detectado dará alguma ideia do dano causado. Interessa que os danos não se espalhem pelo sistema.
• O confinamento de danos (também conhecido por "firewalling") tem a ver com a estruturação do sistema de modo a minimizar os danos causados por um componente defeituoso.
• Existem duas técnicas que podem ajudar no confinamento de danos: a decomposição por módulos e as acções atómicas.
• A decomposição por módulos tem a vantagem de dividir o sistema em componentes sendo cada componente constituído por um ou mais módulos. A interacção entre os módulos ocorre através de interfaces bem definidos. Os detalhes internos de cada um dos módulos não são, em princípio, conhecidos ou acessíveis do exterior. Isto permite que um erro que ocorra num módulo não se propague facilmente a outro componente.
• Os módulos fornecem uma *estrutura estática* ao sistema de software. Igualmente importante para o confinamento dos danos é a *estrutura dinâmica* do software dado que facilita o raciocínio sobre o comportamento do software quando em execução.
• Uma forma de estruturação dinâmica é através de acções atómicas: *A actividade de um componente é considerada atómica se NÃO existirem quaisquer interacções entre a actividade e o sistema durante a acção.*
• Para o sistema, uma acção atómica parece instantânea e indivisível. Não pode passar nenhuma informação da acção atómica para o resto do sistema e vice-versa.
• As acções atómicas são também conhecidas por transacções ou transacções atómicas. São utilizadas para levar o sistema de um estado consistente para outro e restringir o fluxo de informação entre os componentes.
• Quando dois ou mais módulos partilham um recurso, a circunscrição de danos envolve o uso de restrições no acesso ao recurso. A implementação deste aspecto envolve primitivas de comunicação e sincronização entre componentes (semáforos, etc).

• Outras técnicas que tentam restringir o acesso a recursos são baseadas em mecanismos de protecção. De cada vez que um processo pretende fazer algum acesso ou operação a um recurso (leitura, escrita, execução, etc) o tipo de operação pretendida é comparada com as permissões de acesso do processo ao recurso, e se necessário o acesso é negado.

Recuperação de Erros

• Após os erros terem sido detectados e os danos avaliados, é necessário executar procedimentos de recuperação de erros. Esta é provavelmente a fase mais importante numa técnica de tolerância a faltas.

• O objectivo será transformar um estado erróneo num em que se possa continuar o funcionamento normal, embora possivelmente com serviço degradado.

• Têm sido propostos dois métodos para a recuperação de erros: recuperação directa ("forward recovery") e recuperação reversiva ("backward recovery").

• A **recuperação directa** tenta que o sistema recupere a partir de um estado de erro fazendo um conjunto de correcções (transições) ao estado do sistema.

• Pode ser importante para garantir a segurança num ambiente controlado por um sistema embebido.

• Este método apesar de ser eficiente, é específico a cada aplicação e depende da predição rigorosa da fonte e causa dos erros.

• Exemplos: ponteiros redundantes em estruturas de dados, códigos de autocorrecção tais como os códigos de Hamming.

• Se existir mais que um processo envolvido a fornecer serviços quando o erro ocorre pode ser necessário enviar uma excepção assíncrona a vários processos para realizar a recuperação.

• A **recuperação reversiva** consiste em restaurar o sistema num estado seguro prévio àquele em que ocorreu o erro.

• De seguida é executada uma secção alternativa àquela em que ocorreu o erro. Esta secção deve ter a mesma funcionalidade mas utilizar um algoritmo diferente. Tal como na programação por N-versões, espera-se que a secção alternativa não produza a mesma falta.

• O ponto para o qual o processo é recuperado, é chamado ponto de recuperação, e o acto de o estabelecer é designado de *checkpointing*. Para estabelecer um ponto de recuperação é necessário salvar informação apropriada durante a execução.

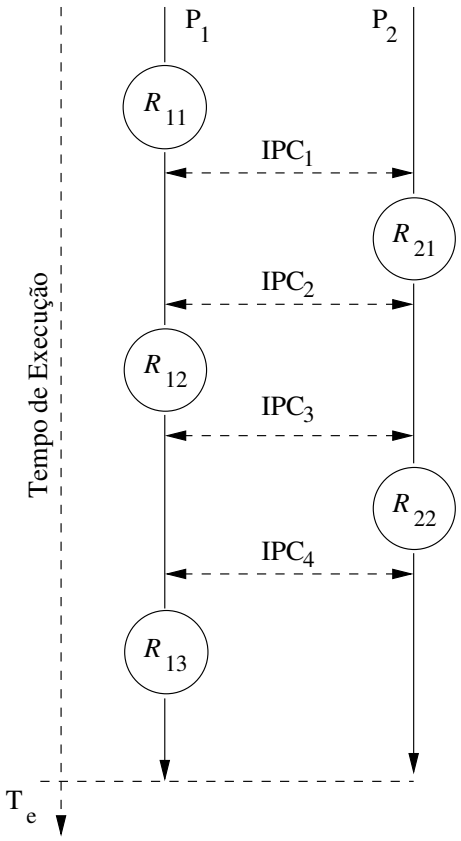
• Vantagem: o erro é apagado mas para isso não é necessário encontrar a localização ou a causa do erro. Este método pode portanto ser utilizado para recuperar de faltas não antecipadas incluindo erros de projecto.

• Desvantagens: (1) Não pode reverter os efeitos da falta no ambiente. (2) Pode consumir muito tempo o que pode impedir a sua utilização em aplicações de tempo-real.

• Para diminuir as penalizações de desempenho, podem-se implementar técnicas de checkpointing incremental. Um exemplo são os trilhos de auditoria. Nestes casos o sistema de suporte deve reverter as acções indicadas no trilho.

• Efeito dominó. A interacção entre processos concorrentes, pode dificultar o restauro do estado do sistema para um ponto de recuperação prévio. Nestes casos, se existir comunicação entre os processos, ao ser detectado um erro num estado de um processo pode ser necessário levar vários processos a estados anteriores uma vez que não podemos simplesmente desfazer e recomeçar as comunicações.

Exemplo: consideremos os dois processos ilustrados na figura ao lado. O processo P_1 estabelece pontos de recuperação R_{11}, R_{12}, R_{13}. O processo P_2 estabelece pontos de recuperação R_{21}, R_{22}. Os dois processos comunicam e sincronizam as suas operações através dos eventos de comunicação inter-processos IPC_1, IPC_2, IPC_3, IPC_4.



• Se P_1 detecta um erro no instante T_e, então recupera simplesmente para o ponto R_{13}. Contudo, consideremos o caso em que P_2 detecta um erro em T_e. Se P_2 for recuperado para R_{22}, então deve desfazer a comunicação IPC_4 com P_1.

• Mas isto obriga a que P_1 regresse a R_{12}. Mas se isso for feito, P_2 tem que regressar a R_{21} para desfazer a comunicação IPC_3, e assim sucessivamente. A consequência é que ambos os processos têm que recuperar para o início da sua fase de interacção mútua. Em muitos casos isto pode ser equivalente a abortar os processos! Este fenómeno é conhecido por efeito de dominó.

• Claramente, se não existir interacção entre processos não existirá efeito de dominó. Se houverem mais de dois processos a interactuar, a possibilidade de ocorrer o efeito dominó aumenta. Neste caso convém projectar pontos de recuperação consistentes em todo o sistema de forma a que a detecção de um erro num processo não resulte na recuperação total de todos os processos. Em vez disso, podem-se recuperar os vários processos para um conjunto consistente de pontos de recuperação. Estas linhas de recuperação como são frequentemente chamadas, estão relacionadas com a noção de acções atómicas atrás discutidas.

• Os sistemas embebidos devem ter capacidade de não só recuperar de erros não antecipados, mas também de responder em tempo finito. Podem portanto ter a necessidade de utilizar ambas as técnicas de recuperação de erros: directa e reversiva.

Tratamento de Faltas e Funcionamento Contínuo

• Um erro é uma manifestação de uma falta, e embora a fase de recuperação de erros possa ter reposto o sistema num estado consistente e sem erros, o erro pode voltar a ocorrer se a falta não for tratada/reparada. Assim...

• A fase final na tolerância a faltas é a erradicação da falta de forma a que a operação normal possa continuar.

• Tratamento automático de faltas é difícil e depende do sistema concreto. Por isso, alguns sistemas não implementam tratamento de faltas, assumindo que todas as faltas são transitórias. Outros sistemas assumem que as técnicas de recuperação de faltas são suficientes para lidar com faltas recorrentes.

• O tratamento de faltas pode ser dividido em duas fases: localização da falta e reparação do sistema/componente.

• Técnicas de detecção de erros podem ajudar a localizar o componente em falta.

• Para um componente de hardware isto pode ser suficientemente preciso e o componente pode ser simplesmente substituído.

• Uma falta de software pode ser removida numa nova versão do código.

• Em aplicações com funcionamento contínuo é necessário modificar o programa enquanto este está a ser executado. Trata-se de um grande problema técnico.

Blocos de Recuperação para Tolerância a Faltas de Software

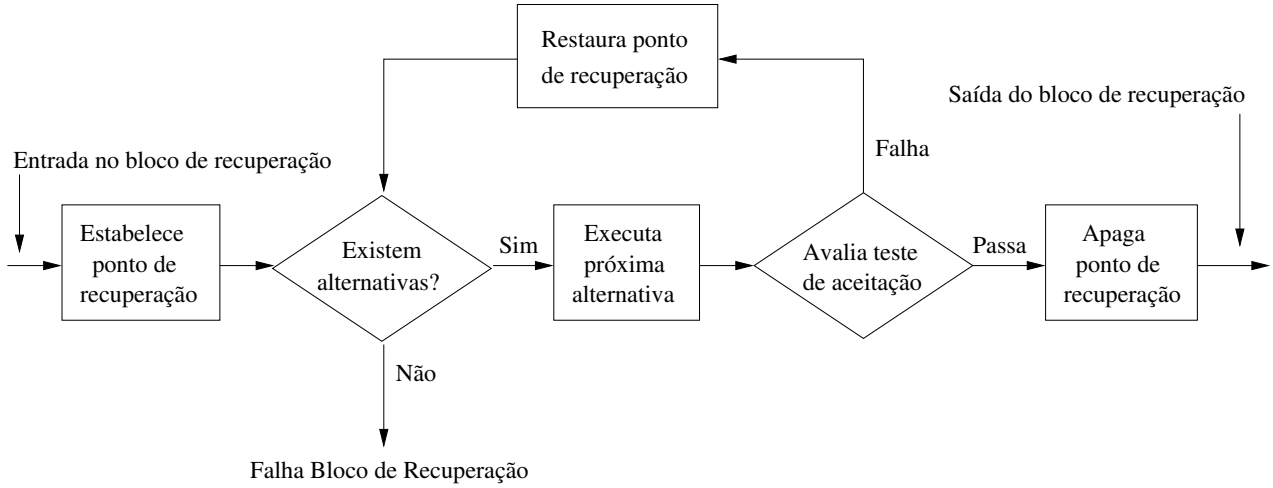
• Os blocos de recuperação de erros são uma forma de recuperação de erros reversiva. Podem ser implementados como um mecanismo da linguagem de programação.

• São blocos no sentido normal de uma linguagem de programação, excepto que é estabelecido automaticamente um ponto de recuperação à entrada do bloco e um teste de aceitação à saída.

• O teste de aceitação é usado para testar se o sistema está num estado aceitável após a execução do bloco (ou módulo primário como é frequentemente chamado).

• Se o teste de aceitação falhar, o programa é restaurado para o estado do ponto de recuperação do início do bloco sendo depois executado um módulo alternativo para esse bloco. Se o novo módulo também falhar o teste de aceitação, o programa é novamente restaurado no início do bloco e é executado um terceiro módulo.

• Se todos os módulos falharem, então o bloco falha e a recuperação tem que ser efectuada a um nível mais elevado. A execução de um bloco de recuperação é ilustrado na figura seguinte:



• Em termos das 4 fases da tolerância a faltas de S/W: (1) a detecção de erros é implementada pelo teste de aceitação, (2) a avaliação de danos não é necessária dado que se assume que (3) a recuperação reversiva apaga todos os estados erróneos, e (4) o tratamento de faltas é alcançado pelo uso de um módulo alternativo.

• Uma possível sintaxe para os blocos de recuperação é a seguinte:

```
ensure <teste de aceitação>
by
<módulo primário>
else by
  <módulo alternativo>
else by
  <módulo alternativo>
else by
  ...
else by
  <módulo alternativo>
else error
```

• Tal como nos blocos normais, os blocos de recuperação podem ser encadeados uns dentro dos outros. Se um bloco encadeado falhar os seus testes de aceitação e se todos os seus módulos alternativos também falham então será restaurado o ponto de recuperação do nível exterior, e será executado um módulo alternativo a esse bloco.

• O teste de aceitação fornece o mecanismo de detecção de erros que possibilita depois que a redundância do sistema seja explorada. O projecto do teste de aceitação é crucial para a eficácia do método dos blocos de recuperação.

• Tal como em todos os mecanismos de detecção de erros existe um compromisso entre fornecer um teste de aceitação muito completo e manter a sobrecarga de teste ao mínimo, de forma a que a execução normal sem faltas seja afectada o mínimo possível.

• Notemos que o termo empregue é aceitação e não correcção; isto permite que um componente forneça um serviço degradado mas aceitável.

• Todas as técnicas de detecção de erros vistas atrás podem ser usadas para construir o teste de aceitação. Contudo, deve haver cuidado no seu projecto dado que um teste de aceitação com faltas pode levar à continuação da operação do sistema com erros residuais não detectados.

Comparação Entre Programação por N-versões e Blocos de Recuperação

• Foram estudadas duas aproximações para fornecer software tolerante a faltas: programação por N-versões (PNV) e blocos de recuperação (BR). Seguidamente vamos fazer uma breve revisão e comparação destes dois processos.

• Redundância estática versus dinâmica: a programação por N-versões é baseada em redundância estática; todas as versões correm em paralelo independentemente da falta ocorrer ou não. Por outro lado, os blocos de recuperação são dinâmicos no sentido em que apenas quando é detectado um erro é que são executados módulos alternativos.

• Sobrecargas no projecto: tanto PNV como BR obrigam a custo de desenvolvimento adicional, dado que ambos requerem o desenvolvimento de algoritmos alternativos. Por outro lado, PNV requer um processo driver e os BR requerem o projecto de um teste de aceitação.

• Sobrecargas em execução: em execução PNV requer aproximadamente N vezes os recursos de uma versão. Embora BR apenas requeiram um único conjunto de recursos de cada vez, existem custos associados com o processo de estabelecimento de pontos de recuperação e de restauro do estado. Contudo, apenas é necessário restaurar o estado quando uma falta ocorre. Além disso, é também possível o desenvolvimento de hardware para suportar o estabelecimento de pontos de recuperação.

• Diversidade de projecto: Em ambos os métodos é explorada a diversidade de projecto para alcançar tolerância a erros não antecipados. Portanto, ambos os métodos são sensíveis a erros nas especificações de requisitos.

• Detecção de erros: PNV usa comparação de votos e BR usa um teste de aceitação. Quando for possível votação exacta ou inexacta esta tem provavelmente associada menos sobrecarga do que com teste de aceitação. Contudo, se for difícil encontrar uma técnica de votação inexacta, se existirem múltiplas soluções, ou se existir um problema de comparação consistente, os testes de aceitação podem fornecer uma solução mais flexível.

• Atomicidade: a recuperação de erros reversiva tem a desvantagem de não poder reverter qualquer dano que possa ter ocorrido no ambiente. PNV evita este problema porque se assume que as versões não interferem entre elas: são atómicas. Isto requer que cada versão comunique com o processo driver e não com o ambiente. Contudo é possível estruturar os programas de forma a que não sejam incluídas operações irrecuperáveis em blocos de recuperação.

• Deve-se notar que embora PNV e BR sejam aproximações alternativas, elas também são complementares. Por exemplo não existe nada que impeça de usar BR no projecto de cada versão de um sistema de PNV.

Redundância Dinâmica e Excepções

• Um erro é uma manifestação de uma falta. Uma falta é a manifestação de um erro num componente. Estes erros podem ser antecipados, como por exemplo um erro de leitura num sensor devido a um erro de hardware; ou não antecipados como no caso de um erro de projecto de um componente.

• Uma excepção pode ser definida como a ocorrência de um erro. Trazer uma excepção à consideração da entidade que invocou a operação que a causou é chamada sinalização (levantamento ou lançamento) da excepção. A resposta a essa excepção é designada de manuseamento (ou serviço).

• O manuseamento de uma excepção pode ser considerado um mecanismo de recuperação directa de erros, dado que quando uma excepção ocorre, o sistema não é reposto num estado prévio. Em vez disso é passado o controlo a uma rotina de resposta que inicia os procedimentos de recuperação.

• Contudo o manuseamento de excepções permite também implementar mecanismos de recuperação reversiva de erros.

• Os mecanismos de manuseamento de excepções não foram incluídos nas linguagens de programação para lidar com erros de projecto de programas. A motivação original para as execepções vinha da necessidade de lidar com condições raras e anormais que surjam no ambiente em que o programa executa.

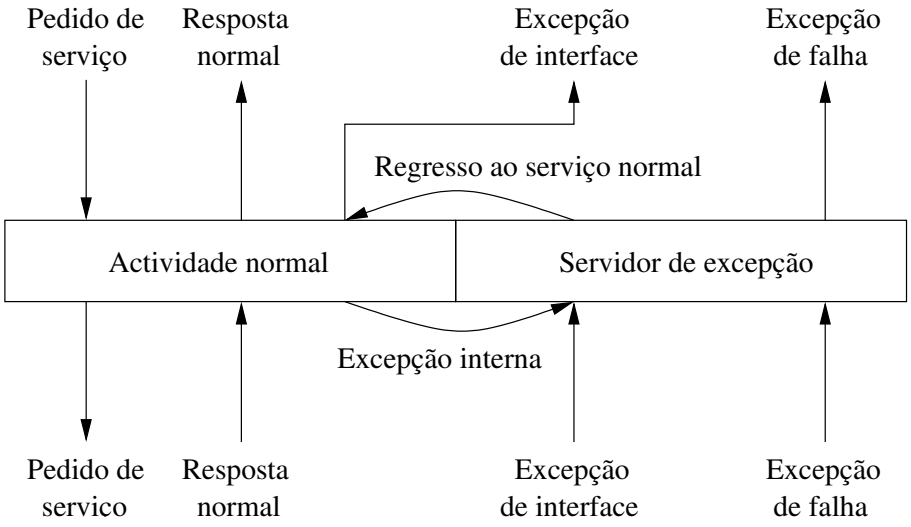
• Exemplo: uma válvula avariada ou um alarme de temperatura podem causar excepções. Estes são eventos raros que passado muito tempo podem perfeitamente ocorrer e devem ser tolerados.

• As excepções são um mecanismo geral para lidar com erros. Resumindo, as excepções podem ser usadas para:

- Lidar com condições anormais que surjam no ambiente;
- Possibilitar que sejam toleradas faltas de projecto do programa;
- Fornecer um mecanismo genérico de detecção e recuperação de erros.

Um Componente Tolerante a Faltas Ideal

• O componente recebe pedidos de serviço e, se necessário, solicita os serviços de outros componentes antes de fornecer uma resposta. Esta resposta pode ser normal ou ser uma excepção.



• Podem ocorrer dois tipos de faltas no componente ideal: as devidas a um pedido ilegal de um serviço, chamadas excepções de interface.

• O outro tipo de faltas são as devidas ao mau funcionamento do próprio componente ou nos componentes necessários para desempenhar o serviço originalmente pedido. Quando o componente, por recuperação de erros quer directa quer reversiva, não puder tolerar estas faltas, levanta uma excepção de falha para o componente que chamou o serviço.

• Antes de levantar qualquer excepção, o componente deve, se possível, repor-se a si próprio num estado consistente de forma a que possa servir qualquer pedido futuro.

Segurança e Fiabilidade

• Um sistema diz-se seguro se estiver *livre da ocorrência de condições que causem morte, ferimentos, doenças ocupacionais, danos em (ou perdas de) equipamento (ou propriedade).* Define-se um mau-acaso como a ocorrência de um evento (ou série de eventos) não planeados como aqueles que possam ter como consequência estes tipos de danos.

• A fiabilidade de um sistema pode-se definir como uma medida do sucesso pelo qual este está conforme com alguma especificação para o seu funcionamento.

• Embora a fiabilidade e a segurança sejam muitas vezes considerados como sinónimos, existe uma diferença de ênfase nos dois conceitos. A fiabilidade é frequentemente expressa em termos de probabilidade. A segurança, contudo, é a probabilidade de que não ocorram condições que possam levar a maus-acasos isto *independentemente de a função desejada seja ou não a realizada.*

• Estas duas definições podem ser conflituosas entre si. Por exemplo, métodos que aumentem a probabilidade de uma arma disparar quando é necessário podem perfeitamente aumentar a possibilidade da sua detonação acidental. Em muitos aspectos, o único avião seguro é aquele que nunca levanta voo; contudo, um tal avião não seria muito fiável.

• Tal como com a fiabilidade, para garantir requisitos de segurança de um sistema embebido, é necessário realizar análises de segurança do sistema ao longo de todas as fases de desenvolvimento do sistema.

• Existem técnicas para realizar análise de segurança de sistemas e em particular de (sistemas de) software.

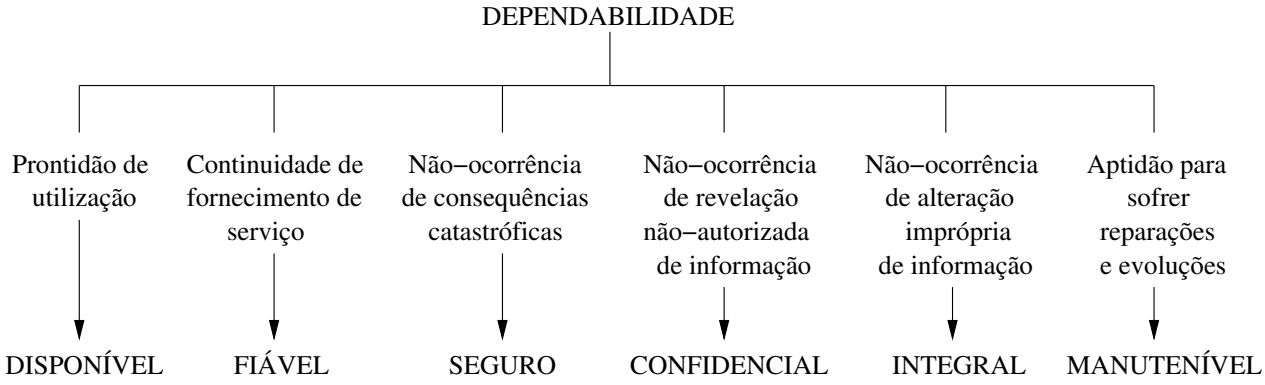
Dependabilidade

• Com o muito trabalho que se tem realizado na área da computação tolerante a faltas, os conceitos relativos à segurança e fiabilidade têm evoluído e tem-se procurado usar termos que melhor descrevam aspectos particulares que se queiram enfatizar.

• Tem-se portanto tentado produzir definições claras e largamente aceites para os conceitos básicos encontrados neste campo.

• Neste sentido, introduziu-se a noção de dependabilidade. A dependabilidade de um sistema é a propriedade que permite confiar justificadamente no serviço que este fornece.

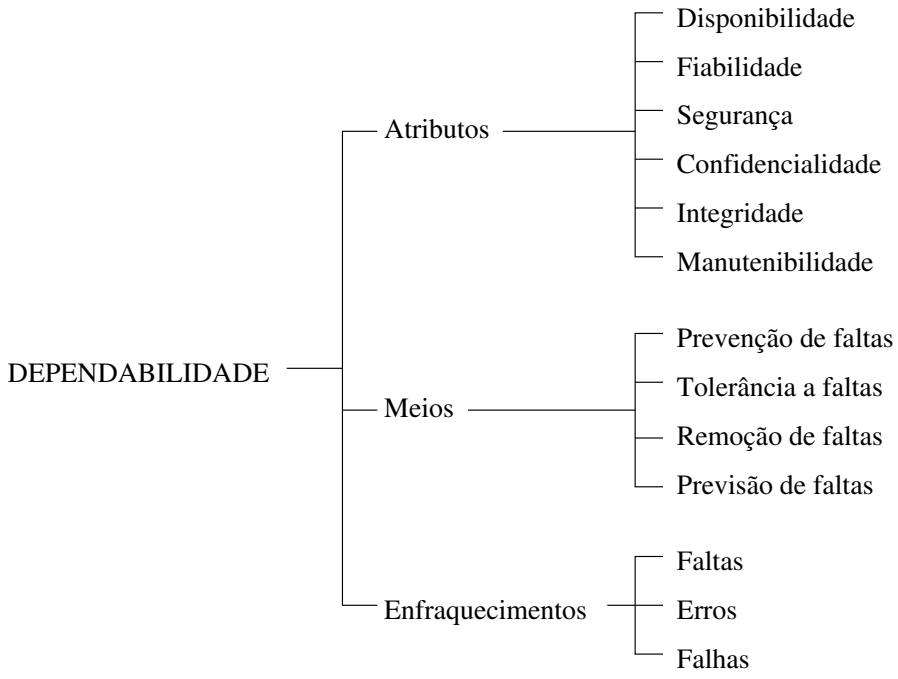
• A dependabilidade tem como casos especiais as noções de fiabilidade e segurança. A figura seguinte ilustra os diversos conceitos associados com a dependabilidade.



• A dependabilidade pode ser descrita em termos de três componentes:

- Enfraquecimentos: circunstâncias que sejam causa de, ou resultem de, não-dependabilidade;
- Meios: os métodos, ferramentas, e soluções necessários para um serviço dependável com o necessário grau de confiança;
- Atributos: a forma e as medidas pelas quais se possa avaliar/estimar a qualidade de um serviço dependável.

• A figura seguinte ilustra o conceito de dependabilidade em termos destes três componentes:



Excepções e Serviço a Excepções

• Excepções → método para implementar tolerância a faltas de *software*.

• Requisitos genéricos para um mecanismo de serviço a excepções:

(R1) Deve ser fácil de compreender e usar.

(R2) O código para o serviço a excepções não dever ser tão intrincado que obscureça a compreensão da operação normal sem erros. Um mecanismo que misture o código de operação normal e o de processamento a excepções será difícil de compreender e manter; Podendo perfeitamente levar a um sistema menos fiável.

(R3) Sobrecargas de execução só devem existir quando se realiza o serviço a uma excepção. A maioria das aplicações requer que o desempenho de um programa que usa excepções não seja adversamente afectado em condições normais de operação. Contudo, isto pode não ser sempre assim. Em algumas circunstâncias, em particular quando a velocidade de recuperação é de primordial importância, a aplicação poderá tolerar um pouco de sobrecarga em operação normal sem erros.

(R4) O mecanismo deve permitir o tratamento uniforme de excepções detectadas pelo ambiente e pelo programa. Por exemplo, uma excepção tal como o transbordo aritmético deve ser servida exactamente da mesma forma como uma excepção levantada pelo próprio programa como resultado de uma falha de asserção.

(R5) Mecanismo de excepções deve permitir programar acções de recuperação (de erros).

Serviço a Excepções em Linguagens de Tempo-Real Mais Antigas

• **Valor de retorno não usual**, ou **retorno de erro** de um procedimento:

• Exemplo:

```
if(chamada_a_função(parâmetros)==UM_ERRO){
    /* Codigo de serviço ao erro */
}else{
    /* Codigo para o regresso normal */
}
```

→ Cumpre o requisito de simplicidade (R1) e permite a implementação de acções de recuperação (R5).

→ Falha em cumprir o requisitos R2, R3, e R4: o código é obscuro, implica sobrecarga sempre que é usado, e não é claro como lidar com erros no ambiente.

• Outro exemplo: em POSIX, para cada chamada ao sistema `sys_call(param)` está definido um macro

```
#define SYS_CALL(A) if(sys_call(A)!=0) error();
```

em que `error()` → função que executa o processamento de erros.

• **Salto forçado** → saltar ou alterar o endereço de retorno de uma subrotina. Pode-se "saltar por cima" de uma ou mais instruções que imediatamente se sigam a uma subrotina para indicar a presença ou ausência de erro.

• Exemplo:

```
jsr pc, PRINT_CHAR       # Chamada a subrotina
jmp IO_ERROR             # Endereço de regresso:  caso 2
jmp DEVICE_NOT_ENABLED   # Endereço de regresso:  caso 3
# Processamento normal   # Endereço de regresso:  caso 1
```

→ Vantagens: pouca sobrecarga (R3), permite programar acções de recuperação (R5).

→ Desvantagem: estrutura de programação obscura $\Longrightarrow$ $\not R1$, $\not R2$.

→ Desvantagem: tratamento uniforme de excepções detectadas pelo ambiente e pelo programa (R4) também não pode ser satisfeito.

- **Goto não local:**
  - Exemplo:

```
svc data rrerr
  label erl; %uma variavel de label%
enddata
proc WhereErrorIsDeteted();
  ...
  goto erl;
  ...
endproc;
proc Caller();
  ...
  WhereErrorIsDetected();
  ...
endproc;
proc main();
  ...
  restart;
    ...
    erl := restart;
    ...
    Caller();
    ...
endproc;
```

→ Goto → "mais do que um jump" → implica regresso de um procedimento de forma anormal → a pilha deve ser restaurada e desenrolada para que o ambiente seja restaurado para o procedimento que contém a declaração do "label"-destino.

→ Sobrecarga de processamento excepcional da pilha só existe quando acontece um erro ⇒ o que está de acordo com o requisito R3.

→ Uso de goto's → flexível → de acordo com os requisitos R4 e R5.

→ Uso de goto's leva a programas obscuros: contraria os requisitos R1 e R2.

- **Variável de procedimento de erro:**

  - Exemplo:

```
svc data rrerr
  label erl;     % uma variavel de label%
  proc(int) erp; % erp e uma variavel de procedimento %
enddata
proc recover();
  ...
  ...
endproc;
proc WhereErrorIsDeteted();
  ...
  if recoverable then
    erp(n)
  else
    goto erl
  end;
  ...
endproc;
proc Caller();
  ...
  WhereErrorIsDetected();
  ...
endproc;
proc main();
  ...
  erl := fail;
  erp := recover;
  ...
  Caller();
  ...
fail:
  ...
endproc;
```

→ Novamente: principal desvantagem → com este método os programas podem tornar-se bastante difíceis de perceber e manter.

Técnicas de Serviço a Excepções Mais Recentes

• Métodos mais directos de serviço a excepções sem implicar mistura de código normal com código de serviço a excepções.

• Mais recentemente: introduzir mecanismos de serviço a excepções directamente na linguagem → fornecendo um mecanismo mais estruturado.

• Mecanismos particulares variam de linguagem para linguagem; Vamos de seguida descrever alguns traços comuns.

• Excepções e sua Representação

• Dois tipos de técnicas de detecção de erros:

→ Detecção no/pelo ambiente.

→ Detecção na/pela aplicação.

• Dependendo do atraso na detecção do erro, pode ser necessário levantar a excepção sincronamente ou assincronamente.

→ Uma excepção síncrona é levantada como resultado imediato de uma secção de código que tenta realizar uma operação inapropriada.

→ Uma excepção assíncrona é levantada num instante diferente (algum tempo depois p.ex.) da operação que resultou na ocorrência do erro.

• A excepção pode ser levantada no processo que originalmente executou a operação (que resultou no erro) ou num outro processo.

• Existem portanto quatro classes de excepções:

- Detectada pelo ambiente e levantada sincronamente: e.g. divisão por zero ou a violação dos limites de um array.
- Detectada pela aplicação e levantada sincronamente: e.g. falha de uma verificação de asserção definida pelo programa.
- Detectada pelo ambiente e levantada assincronamente: e.g. uma excepção levantada em resultado de uma falha de energia ou de uma falha de um mecanismo de monitorização de saúde.

- Detectada pela aplicação e levantada assincronamente: e.g. um processo pode reconhecer que ocorreu uma condição de erro que resultará que um outro processo não cumpra a sua meta temporal ou não termine correctamente. ⇒ excepção é depois levantada no processo que não cumprirá a meta.

• As excepções assíncronas são frequentemente designadas por sinais e são usualmente consideradas no contexto da programação concorrente.

• Domínio de um Servidor de Excepções

• Num programa podem haver vários servidores para uma excepção particular.

• Associado a cada servidor está um domínio que especifica a região de computação ao longo da qual, se uma excepção ocorrer, será activado esse servidor.

• A precisão/resolução com que pode ser especificado um domínio determina quão precisamente pode ser detectada a fonte da excepção.

• Numa linguagem estruturada por blocos, um domínio pode normalmente ser o bloco.

• Exemplo em ADA: variável fora de gama $\xrightarrow{\text{ADA}}$
é levantada uma excepção "constraint error".

```
declare
  subtype Temperature is Integer range 0..100;
begin
  - read temperature sensor and calculate its value
exception
  - handler for Constraint_Error
end;
```

• Quando os blocos formam a base de outras unidades tais como procedimentos e funções o domínio de um servidor de excepções é usualmente essa unidade.

• Em algumas linguagens nem todos os blocos podem ter servidores de excepções (internos). Alternativamente o domínio de um servidor de excepções deve ser explicitamente indicado e o bloco é considerado guardado.

• O domínio de um servidor de excepções especifica quão precisamente o erro pode ser localizado. → a unidade "bloco" pode não ter granularidade suficiente/adequada.

• Exemplo do problema da precisão de localização do erro:

```
declare
  subtype Temperature is Integer range 0..100;
  subtype Pressure is Integer range 0..50;
  subtype Flow is Integer range 0..200;
begin
  - read temperature sensor and calculate its value
  - read pressure sensor and calculate its value
  - read flow sensor and calculate its value
  - adjust temperature, pressure and flow according
  - to requirements
exception
  - handler for Constraint_Error
end;
```

• Com domínios de servidores de excepções baseados em blocos, uma solução pode ser diminuir o tamanho dos blocos ou encadeá-los ("nest"). Exemplo →

```
declare
  subtype Temperature is Integer range 0..100;
  subtype Pressure is Integer range 0..50;
  subtype Flow is Integer range 0..200;
begin
  begin
    - read temperature sensor and calculate its value
  exception
    - handler for Constraint_Error for temperature
  end;
  begin
    - read pressure sensor and calculate its value
  exception
    - handler for Constraint_Error for pressure
  end;
  begin
    - read flow sensor and calculate its value
  exception
    - handler for Constraint_Error for flow
  end;
  - adjust temperature, pressure and flow according
  - to requirements
exception
  - handler for other possible exceptions
end;
```

• Alternativamente → procedimentos contendo servidores de excepções podem ser criados para cada um dos blocos encadeados.

• Outra solução possível → servidores de excepções definidos ao nível de cada instrução → e.g. linguagem CHILL tem este mecanismo.

• Exemplo hipotético →

```
- NOT VALID Ada
declare
  subtype Temperature is Integer range 0..100;
  subtype Pressure is Integer range 0..50;
  subtype Flow is Integer range 0..200;
begin
  Read_Temperature_Sensor;
    exception - handler for Constraint_Error;
  Read_Pressure_Sensor;
    exception - handler for Constraint_Error;
  Read_Flow_Sensor;
    exception - handler for Constraint_Error;
  - adjust temperature, pressure and flow according
  - to requirements
end
```

→ Causa da excepção localizável mais precisamente, mas existe mistura de código de serviço de excepções com código de fluxo normal → pode resultar em programas menos claros ⇒ requisito R2 não verificado.

• A melhor solução para este problema poderá ser a passagem de parâmetros na invocação de servidores de excepções. E.g. em C++ é possível definir parâmetros quaisquer e definíveis; Ada → parâmetros fixos.

• Propagação de excepções

• Noção de propagação de excepções estreitamente relacionado com o conceito de domínio de uma excepção.

• Até aqui → quando um bloco ou procedimento levanta uma excepção, tem-se assumido que existe um servidor associado com esse bloco ou procedimento.

• Contudo isto pode nem sempre acontecer; e existem dois métodos possíveis para lidar com a situação em que não existe nenhum servidor de excepção imediato.

• 1ºmétodo → encarar a ausência de servidor de uma excepção como um erro de programação que deve ser detectado na compilação.

• Contudo, muitas vezes → excepção levantada num procedimento pode apenas (ser possível de) ser servida no contexto do procedimento "chamador" → neste caso não é possível ter um servidor local no procedimento.

→ E.g. 1 excepção levantada como resultado de uma asserção envolvendo os parâmetros de entrada de um procedimento apenas pode ser servida no procedimento que chamou.

• Contudo, pode nem sempre ser possível o compilador verificar se o contexto chamador inclui servidores de excepção apropriados.

→ Isto pode requerer uma complexa análise de fluxo de controlo principalmente quando um procedimento chama outros procedimentos que levantam excepções.

• Consequentemente → linguagens que requeiram a geração de erros em tais situações durante a fase de compilação requerem que um procedimento especifique que excepções pode levantar e que não sejam servidas localmente.

• O compilador pode então verificar se no contexto chamador existe um servidor apropriado e se necessário gerar uma mensagem de erro → E.g. CHILL, C++.

• 2ºmétodo – que pode ser usado quando não existe um servidor local → verificar se existe um servidor subindo acima na cadeia de invocação de procedimentos durante a execução

→ Este método é designado propagação de excepções. Ada, C++, Modula-2/3 permitem.

• Problema com propagação de excepções → ocorre quando a linguagem requer que as excepções sejam declaradas → e portanto seja fornecido um alcance de validade.

→ Problema: é possível uma excepção ser propagada fora do alcance do seu servidor, tornando portanto impossível encontrar um servidor.

→ Para resolver este problema → a maioria das linguagens têm um servidor "agarra tudo" ("catch all") → serve também para evitar que o programador enumere muitos nomes de excepções.

• Excepção não servida num programa sequencial ⇒ programa abortado.

→ Se programa contém mais que um processo e um deles não serve uma das suas excepções, então esse processo é abortado.

→ Contudo pode-se pôr o problema de "se a excepção deve ou não ser propagada para o processo pai".

• A propagação de excepções pode ser considerada em termos de se os servidores estão associados estaticamente ou dinamicamente com as excepções.

• Associação estática (e.g. CHILL) é realizada durante a compilação e não permite a propagação porque a cadeia de invocação é desconhecida.

• Associação dinâmica é realizada durante a execução e portanto permite a propagação.

⇒ mais flexível.

→ mas (-) mais sobrecarga de execução porque se tem que pesquisar o servidor (com associação estática pode-se gerar um endereço do servidor durante a compilação).

• Modelo de Recomeço (Reatamento) versus Modelo de Terminação

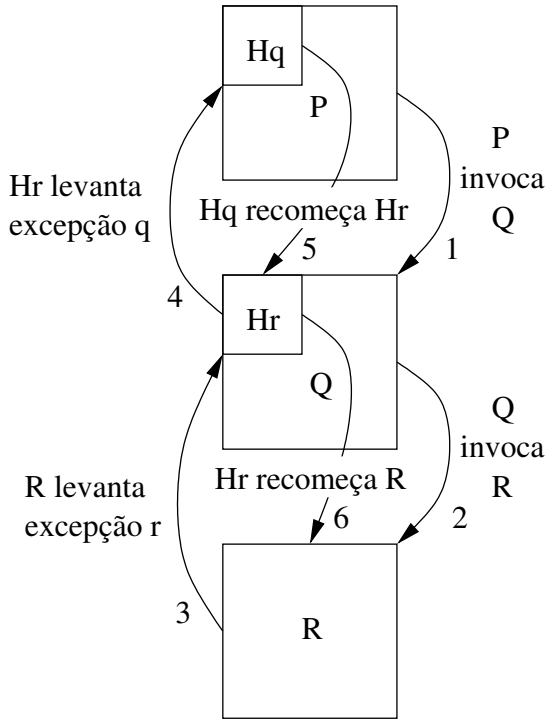
• Mecanismo de serviço de excepções → consideração crucial → o invocador da excepção deve ou não continuar a sua execução após a excepção ter sido servida?

• Modelo de recomeço (reatamento) ou notificação: Se o invocador pode continuar, então pode ser possível para o servidor resolver o problema que causou o levantamento da excepção e o invocador continuar como se nada tivesse acontecido.

• Por outro lado → o modelo onde o controlo não é devolvido ao invocador/causador da excepção é chamado modelo de terminação ou de escape.

• Modelo híbrido: se for possível ao servidor decidir sobre se deve continuar a operação que causou a excepção, ou terminar a operação.

• Modelo de recomeço: Exemplo:



• P $\overset{\text{invoca}}{\longrightarrow}$ Q $\overset{\text{invoca}}{\longrightarrow}$ R

• R $\overset{\text{levanta}}{\longrightarrow}$ excepção r $\overset{\text{servida por}}{\longrightarrow}$
→ servidor Hr em Q
→ (não existe servidor local para r em R)

• Hr $\overset{\text{levanta}}{\longrightarrow}$ q $\overset{\text{servida}}{\longrightarrow}$ Hq em P (rotina chamadora de Q)

• Hq termina → Hr continua

• Modelo de recomeço → mais facilmente compreendido se encararmos o servidor como um procedimento implícito que é invocado quando a excepção é levantada.

• Um problema com este método (em algumas linguagens) é que se torna frequentemente difícil de reparar erros que são levantados pelo ambiente de execução. E.g. "arithmetic overflow" no meio de uma sequência de expressões complexas pode resultar em vários registos contendo avaliações parciais. Como consequência de chamar o servidor → escrever por cima destes registos.

• Linguagens Pearl e Mesa → fornecem mecanismos pelos quais um servidor pode regressar para o contexto que levantou a excepção → Mas também suportam o modelo de terminação.

• Modelo de recomeço estrito → difícil de implementar → solução-compromisso: re-executar (apenas) o bloco associado com o servidor de excepções.

• E.g. linguagem Eiffel contém este mecanismo como parte do modelo de serviço a excepções → designado por retry.

→ O servidor escreve uma flag para indicar que ocorreu um erro e o bloco pode testar essa flag.

→ Para este método funcionar, as variáveis locais do bloco não devem ser reinicializadas num retry. Caso contrário poderá ser desencadeado o mesmo erro/excepção.

• Vantagem do modelo de recomeço: manifesta-se quando a excepção foi levantada assincronamente e portanto tem pouco a ver com a execução presente do processo.

• Modelo de terminação: após uma excepção ter sido levantada e o servidor já ter sido executado, o controlo não regressa depois ao ponto onde a excepção ocorreu.

→ Ao contrário: o bloco ou procedimento contendo o servidor é terminado e o controlo é passado para o bloco ou procedimento que chamou.

• Um procedimento invocado pode portanto terminar numa de várias possíveis condições → uma destas condições é a condição normal e as outras são condições de excepção.

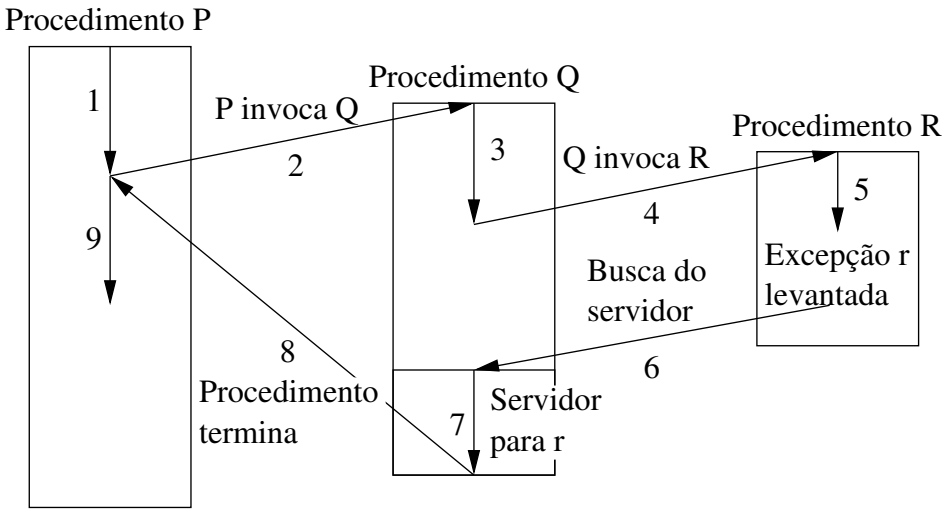
• Quando o servidor está dentro de um bloco, o controlo é transferido para a 1ªinstrução que se segue ao bloco após a excepção ter sido servida.

→ Exemplo com blocos:

```
declare
  subtype Temperature is Integer range 0..100;
begin
  begin
    - read temperature sensor and calculate its value
    - may result in an exception being raised
  exception
    - handler for Constraint_Error for temperature,
    - once handled this block terminates
  end;
  - code here executed when block exits normally or
  - when an exception has been raised and handled.
exception
  - handler for other possible exceptions
end;
```

• Exemplo com procedimentos: ao contrário do que acontece com blocos, o fluxo de controlo pode alterar-se dramaticamente como se ilustra na figura seguinte:

• P $\overset{\text{invoca}}{\longrightarrow}$ Q $\overset{\text{invoca}}{\longrightarrow}$ R; • excepção levantada em R $\overset{\text{e servida em}}{\longrightarrow}$ Q



• Com o modelo híbrido cabe ao servidor decidir se um erro é recuperável ou não. Se for, o servidor pode devolver um valor e a semântica é a mesma que no caso do modelo de recomeço. Se o erro não for recuperável, o invocador da excepção é terminado.

Execução Concorrente

• Existem três mecanismos básicos para representar execução concorrente:

→ fork e join

→ cobegin/coend

→ declaração explícita de processos

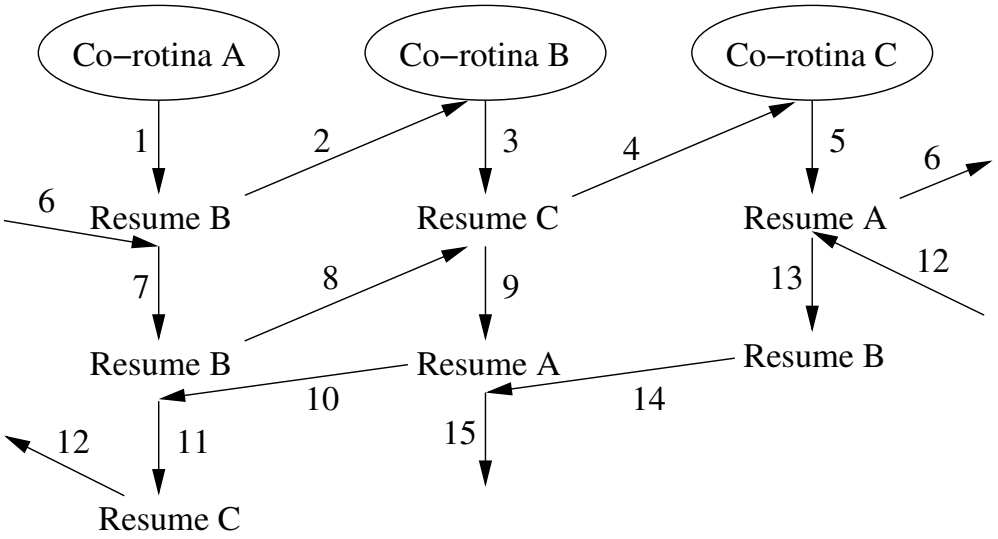
→ co-rotinas são também por vezes incluídas como mecanismo para expressar execução concorrente.

• Co-rotinas: são subrotinas mas que permitem a passagem explícita de controlo entre elas de uma forma simétrica e não apenas de uma forma hierárquica.

• O controlo é passado de uma co-rotina para outra através de uma instrução de resume que nomeia a rotina a continuar de seguida.

• Quando uma co-rotina executa 1 resume, pára a sua execução mas mantém a informação de estado local de forma a que se uma outra subrotina "a resume" depois, a co-rotina poderá continuar a sua execução.

• Exemplo: (fluxo de controlo identificado por setas numeradas)



• Cada co-rotina pode ser encarada como implementando um processo.

• Contudo, não é necessário suporte do sistema de execução dado que as co-rotinas ordenam elas próprias a sua ordem de execução.

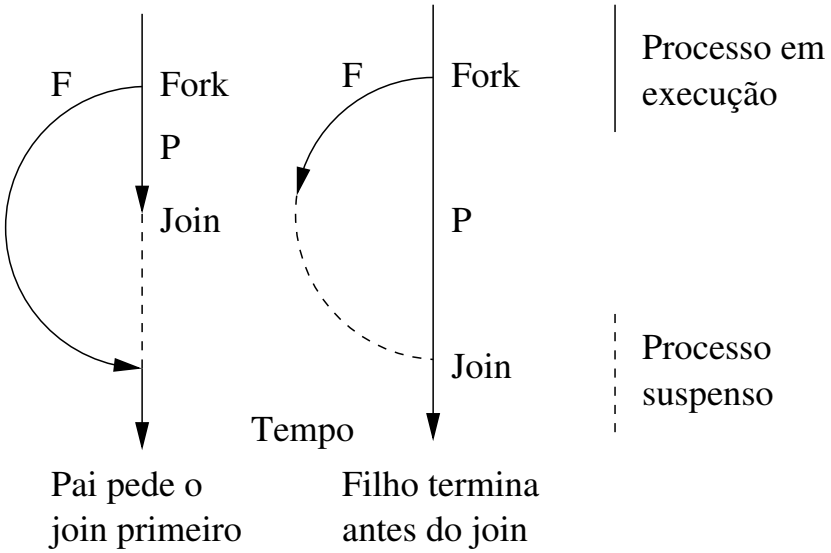
• Co-rotinas → não adequadas para processamento verdadeiramente paralelo dado que a sua semântica apenas permite uma rotina de cada vez.

• Fork e Join: A instrução/função `fork` especifica uma rotina que deve começar a execução concorrentemente com a rotina invocadora.

• A instrução/função `join` permite que o chamador se sincronize com o fim da rotina especificada. Exemplo:

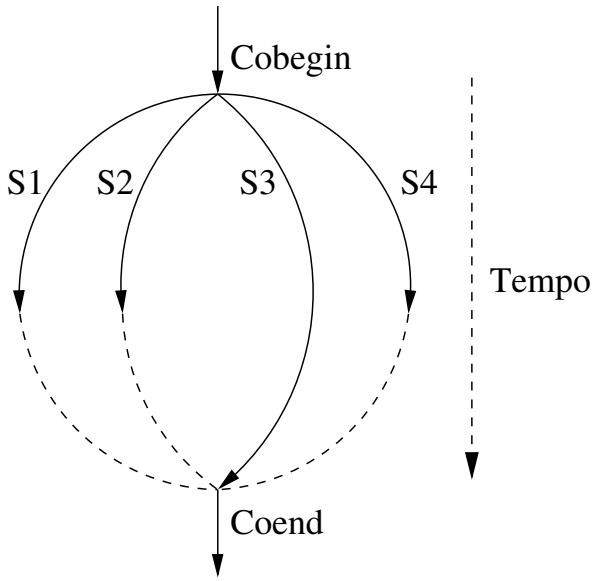
```
function F return ...;
⋮
end F;
procedure P;
  ...
  C := fork F;
    ⋮
  J := join C;
  ...
end P;
```

• Entre a execução do fork e do join, o procedimento P e a função F executam concorrentemente.
• Ao chegar ao join o procedimento espera até que a função termine (se tal ainda não aconteceu).
• Exemplo: ilustração da execução de fork e join.



• Fork e join permitem a criação dinâmica de processos e fornecem meios para passar informação para o processo-filho através de parâmetros.
• Embora sendo flexíveis, não fornecem um método estruturado para a criação de processos e são muito sujeitos a erros de utilização.
• Cobegin/coend: o cobegin (ou parbegin ou par) é uma forma estruturada de representar a execução concorrente de uma colecção de instruções.

```
cobegin
  S1;
  S2;
  S3;
  :
  Sn;
coend
```



• Este código executa as instruções S1,. . . ,Sn concorrentemente.

• A instrução cobegin termina quando todas as instruções concorrentes tiverem terminado.

• Cada Si pode ser uma qualquer instrução permitida pela linguagem, incluindo chamadas a procedimentos → podem ser usados parâmetros a transmitir ao procedimento ou ao processo.

• Uma instrução de cobegin pode mesmo incluir outras que, elas próprias, tenham um cobegin dentro delas.

• Declaração explícita de processos:

• Execução concorrente possível por meio de cobegin e fork.

• Mas a estrutura de um programa concorrente pode ser tornada mais clara se as próprias rotinas especificarem se devem ser executadas concorrentemente.

• Este é o caso da declaração explícita de processos

• Exemplo em Modula-1: um controlador de robot que usa três processos semelhantes cada um dos quais controla uma dimensão de movimento.

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

Serviços de Tempo-Real

• Se um programa interage de alguma forma consistente com a referência temporal do seu ambiente então tem que ter acesso a algum método de "conhecer o tempo" ou pelo menos alguma forma de medir a passagem do tempo.

• Isto pode ser feito de duas formas:

1. Ter acesso directo à referência temporal do ambiente. Exemplos:
   → Interrupção "ambiente → sistema";
   → Sinal de rádio emitido com informação de tempo (e.g. Alemanha);
   → GPS também tem referência do tempo.
2. Usar um relógio interno de hardware que dê uma aproximação adequada da passagem do tempo no ambiente.

• Do ponto de vista do programador, o acesso ao tempo pode ser feito através de uma primitiva de relógio da linguagem ou de um device driver para o relógio interno, relógio externo ou receptor de radio.

Atrasar um Processo

• Atraso relativo: permite a um processo esperar por um evento futuro em vez de fazer chamadas de "busy wait" ao relógio. Exemplo em Ada (atraso relativo com "busy wait"):

```
Start := Clock; - from calendar
loop
  exit when (Clock - Start) > 10.0;
end loop;
```

• Alternativa em Ada → primitiva `delay`; POSIX sleep, nanosleep (+ granularidade):

```
delay 10.0;
```

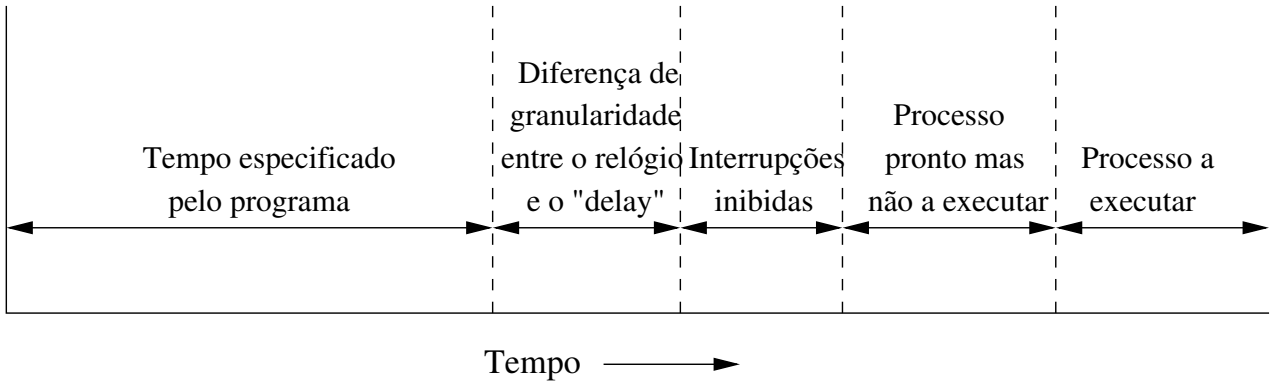
• Delay apenas garante que o processo passa a pronto depois de expirar o período:

→ o atraso real até entrar em execução também depende de outros processos que estão a competir pelo processador;

→ Também notar: granularidade do `delay` e granularidade do relógio → não necessariamente as mesmas;

→ Além disso o relógio interno pode ser implementado com interrupções que podem ser inibidas por curtos períodos.

• Ilustração:



• Atrasos absolutos: se for necessário um atraso até (à chegada de) um instante de tempo absoluto → duas hipóteses:

1. O programador calcula o atraso relativo necessário;
2. Primitiva adicional necessária.

• Exemplo por atraso relativo: 2 acções com inícios separados por 10 seg.

```
Start := Clock;
First_Action;
delay 10.0 - (Clock - Start);
Second_Action;
```

- Para o atraso ser exacto, a instrução de `delay` anterior deveria ser não interrompível.
- Mas se First_Action demora 2 seg., então `10.0 - (Clock - Start)` → 8 seg.; mas se a tarefa é interrompida (e.g. durante 3 seg.) por outra tarefa após o cálculo do tempo de `delay`, então após o reatamento temos um `delay` de 8 e não de 5 como se pretendia nesta situação.

• Solução em Ada → instrução `delay until`:

```
Start := Clock;
First_Action;
delay until Start + 10.0;
Second_Action;
```

• Tal como no "delay", o `delay until` tem precisão apenas no seu limite inferior. A tarefa envolvida não será activada antes do tempo corrente atingir aquele que é especificado na instrução, mas poderá ser activada depois.

• O excesso de tempo associado tanto com o atraso relativo como com o atraso absoluto é designado de **desvio local**.

• Contudo com o atraso absoluto é possível eliminar o **desvio cumulativo** que pode surgir se se permitir que os desvios locais se sobreponham numa sequência de atrasos relativos. Exemplo de uma acção em Ada realizada a cada 7 seg. →

```
declare
  Next :  Time;
  Interval :  Constant Duration := 7.0;
begin
  Next := Clock + Interval;
  loop
    Action;
    delay until Next;
    Next := Next + Interval;
  end loop;
end;
```

Programação de Timeouts

• Programação de timeouts: possivelmente a restrição temporal mais simples que pode ocorrer num sistema embebido é reconhecer e actuar em resposta a uma não ocorrência de algum evento externo.

• Exemplo: 1 sensor de temperatura que tenha que fazer leituras todos os segundos; se não chegar uma leitura até 10 seg. → pode-se definir que ocorreu uma falta.

• Em geral: timeout → restrição sobre o tempo que um processo se permite esperar por uma comunicação.

• Timeouts → também podem ser aplicados para impor restrições ao tempo de execução de acções. Exemplo: um programador pode pretender que uma secção de código execute dentro de um certo tempo. Se tal não acontecer, então pode ser necessário algum procedimento de recuperação de erros.

• Timeouts têm aplicação no contexto de:

- Comunicação por variáveis partilhadas (e.g. semáforos, ou monitores) no contexto de secções críticas ou de condições de sincronização (e.g. buffer cheio no contexto de um problema produtor/consumidor).
- Passagem de mensagens: nenhuma mensagem para receber ou buffer de envio cheio no contexto de recepção e envio síncronos de mensagens, respectivamente.
- Em acções: permite detectar pedaços de código que excedam o tempo alocado para a sua execução. Outro exemplo: tarefa que envolva parte obrigatória e parte opcional e que tenha que terminar dentro de um certo tempo. Depois da obtenção de um resultado adequado para a parte obrigatória, o tempo restante até à ocorrência de um *timeout* pode ser utilizado para melhorar incrementalmente a primeira solução.

Especificação de Requisitos Temporais

• Para muitos sistemas de tempo-real não é suficiente que os programas estejam logicamente correctos; é necessário a satisfação de requisitos temporais determinados pelo sistema subjacente. Estas restrições temporais podem ser mais complexas que simples timeouts.

• Para projectar o sistema de acordo com os requisitos temporais são muitas vezes usadas aproximações *ad hoc*; procedendo-se depois à implementação do sistema, e à verificação da implementação acompanhada se necessário por refinamentos para atingir o objectivo.

- Aproximações mais rigorosas:
  - Uso de linguagens com mecanismos semânticos formalmente definidos: propriedades temporais podem ser definidas e analisadas:

    → Técnicas pouco maduras;

    → Pouca experiência conhecida;

  - Análise de desempenho de sistemas de tempo-real em termos da verificação da possibilidade de escalonamento de uma certa carga de trabalho nos recursos de execução (processadores).

- Verificação de um sistema de tempo-real → 2 fases:

  (1) Verificação de requisitos – dado um computador suficientemente rápido e fiável, são os requisitos temporais coerentes e consistentes? E têm potencial de ser satisfeitos?
  E.g. se um evento A deve ser completado antes do evento B mas está dependente de algum evento C que ocorre após B, então independentemente da velocidade de processamento, não é possível satisfazer estes requisitos.

  (2) Verificação da implementação – com um conjunto finito de recursos de hardware, podem os requisitos temporais ser satisfeitos?

Alcances Temporais ("Temporal Scopes")

• Um alcance temporal (AT) identifica uma colecção de instruções que tem associada uma determinada restrição temporal.

• Atributos possíveis para associar com um alcance temporal:

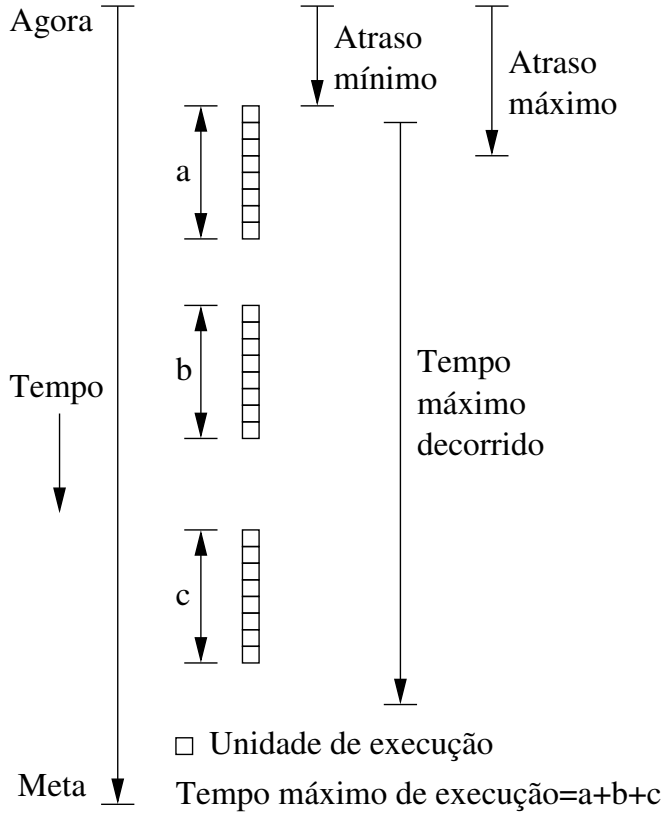
  • Meta (deadline) – o instante até ao qual o AT deve ser/ter terminado.

  • Atraso mínimo (no início) – mínima quantidade de tempo que deve decorrer até começar a execução do alcance temporal.

  • Atraso máximo (no início) – máxima quantidade de tempo que pode decorrer até começar a execução do alcance temporal.

  • Tempo de execução máximo – de um alcance temporal.

  • Tempo máximo decorrido – num alcance temporal.



• Alcances temporais podem-se dividir em periódicos e aperiódicos

• Exemplo de AT periódico → ciclo de amostragem e controlo com metas fixas.

• Alcances temporais aperiódicos ou esporádicos surgem usualmente de eventos assíncronos.

• Em geral, alcances temporais aperiódicos são encarados como sendo activados aleatoriamente de acordo com uma certa distribuição de probabilidades (e.g. de Poisson).

• Para permitir cálculos de tempos de resposta no pior caso, define-se frequentemente um período mínimo entre dois eventos (da mesma fonte). O processo associado designa-se esporádico.

• Em muitas linguagens de tempo-real, os alcances temporais estão associados aos processos que os contêm. Os processos podem ser descritos como periódicos, aperiódicos, ou esporádicos dependendo das propriedades do seu alcance temporal interno.

• Muitos dos atributos temporais de um AT definidos na lista acima podem ser satisfeitos por:

- correr os processos a uma frequência correcta;
- completar todos os processos dentro da sua meta temporal.

• O problema de satisfazer as restrições temporais torna-se portanto num de escalonar processos para cumprir metas → escalonamento de metas.

• Notemos que já estudámos métodos para analisar tempos de resposta e a possibilidade de escalonamento.

• O requisito de atraso máximo antes do início de execução de um AT, pode ser alcançado/modelado separando o alcance em dois processos com uma relação de precedência:

- O primeiro que representa a fase inicial do AT, pode ter uma meta que garanta que não seja violado o atraso máximo no início. A necessidade deste tipo de estrutura surge por exemplo de uma aplicação que primeiro leia um sensor e produza depois uma saída a partir dessa leitura. Para ter um controlo fino do instante em que o sensor é lido é necessária uma meta apertada para esta acção inicial.
- A saída pode depois ser produzida por uma acção que tenha associada uma meta posterior

• Estas variações relativas ao instante em que o sensor é lido ou quando um valor de actuação é produzido na saída são designadas de ruído de entrada ("input jitter") e ruído de saída ("output jitter") respectivamente.