

Sistemas Operacionais

Ciência da Computação

Robson Costa, Dr.

`robson.costa@ifsc.edu.br`

Instituto Federal de Santa Catarina
Campus Lages

Versão: 27 de fevereiro de 2023

Sumário

Unidade 2 – Gerenciamento de processos e *threads*

2.1 - Tarefas

2.2 - Trocas de contexto

2.3 - Processos

2.4 - *Threads*

2.5 - Sincronização e Impasses

2.6 - Escalonamento

2.7 - Escalonamento por Prioridades

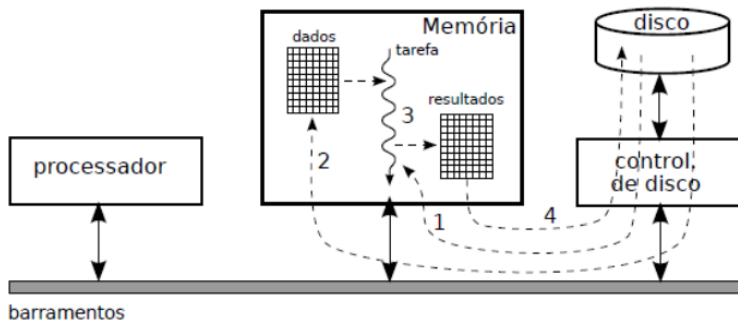
2.8 - Sistemas de Tempo Real

Tarefas

- As técnicas atuais de gerenciamento de processos realizam a multiplexação do processador entre as múltiplas tarefas existentes;
- O que é uma tarefa (*task*)?
 - A execução de um fluxo sequencial de instruções para atender uma finalidade específica;
- Um programa?
 - É o conjunto de uma ou mais sequências de instruções escritas para resolver um problema específico, constituindo assim uma aplicação ou um utilitário;

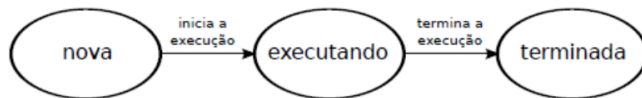
Tarefas

- Cada tarefa geralmente possui comportamento, duração e importância distintas;
- Cabe ao SO organizar a sequência de execução das tarefas;
- Os primeiros sistemas computadorizados eram **mono-tarefa**:
 - Somente era possível executar uma tarefa de cada vez;



Tarefas

- Como resultado, temos o seguinte diagrama de estado:



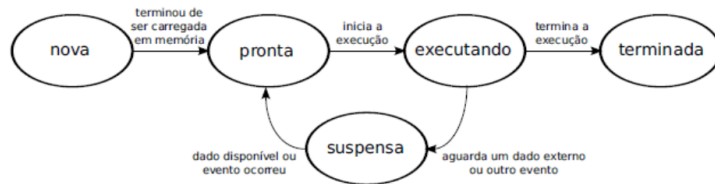
- A transição de uma tarefa para outra era realizada manualmente pelo programador ou pelo operado profissional;
- Posteriormente implementou-se o ***system monitor***, capaz de automatizar a transição do fluxo de execução de uma tarefa para a outra;
- Percebe-se claramente que a função do ***system monitor*** é gerenciar uma fila de tarefas a serem executada, otimizando assim a utilização do processador.

Tarefas

- Outros problemas ainda persistiam:
 - O tempo em que o processador ficava ocioso esperando a tarefa finalizar uma operação em algum periférico de entrada/saída;
- A solução encontrada foi permitir a suspensão temporária das tarefas que realizam operações de I/O e permitir que outras utilizem o processador em seu lugar;
- Este procedimento é chamado de **preempção de tarefas**, e os SO que o implementam são chamados de **sistemas preemptivos**;
- Isto deu origem aos sistemas **multi-tarefas**;

Tarefas

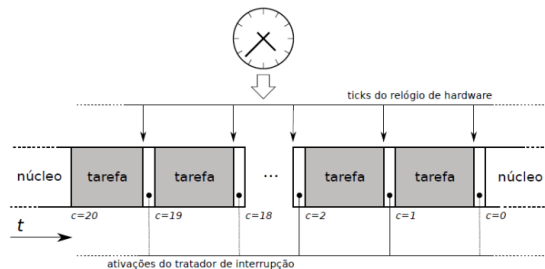
- Desta forma, o diagrama de estados passa a ser:



- Ainda existia um problema:
 - Como tratar uma tarefa cujo o tempo de execução era infinito (em *looping*), seja este acidentalmente ou intencionalmente?
 - Além de poder inviabilizar a utilização do computador no caso de *looping*, esta abordagem não permitia interatividade (ex.: uso do terminal de comandos);

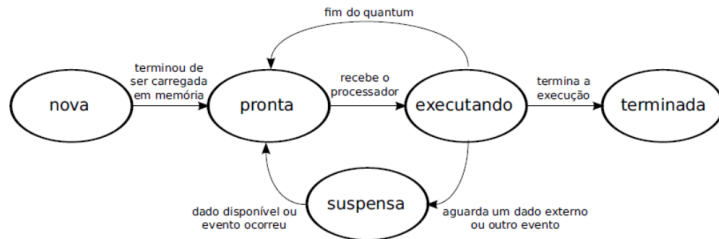
Tarefas

- Para solucionar isto foram criados os sistema de tempo compartilhado (*time-sharing*);
- Nesta abordagem, cada atividade que detém o processador recebe um limite de tempo de processamento chamado de **quantum**;
- Caso a tarefa não seja finalizada, esta retorna para a fila de tarefas prontas para execução (*ready-queue*) e aguarda uma nova oportunidade de utilizar o processador;



Tarefas

- Desta forma, o diagrama de estados pode ser:



Implementação de Tarefas

- Uma tarefa é uma unidade básica de atividade dentro de um sistema;
- Estas podem ser implementadas de várias formas:
 - Processos;
 - *Threads*;
 - *Jobs*;
 - Transações;
- As tarefas pode alternarem-se entre si utilizando o conceito de *chaveamento de contexto*;
- Contexto é o conjunto de dados que representam o estado atual da tarefa:
 - Arquivos abertos;
 - Valores de variáveis;
 - Valores de registradores;

Implementação de Tarefas

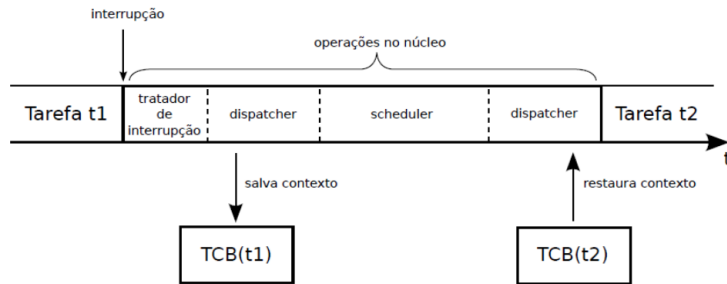
- Cada tarefa possui um descritor associado, ou seja, uma estrutura de dados onde as informações de contexto são armazenadas;
- Esta estrutura geralmente é chamada de TCB (*Task Control Block*) ou PCB (*Process Control Block*);
- O TCB possui tipicamente os seguintes dados:
 - ID (número, ponteiro, referência de objeto);
 - Estado (nova, pronta, executando, suspensa, finalizada, ...);
 - Valores dos registradores do processador;
 - Lista contendo as áreas de memória utilizadas pela tarefa;
 - Lista contendo os arquivos abertos, conexões de rede e outros recursos;
 - Informações de gerência e contabilização (prioridade, proprietário, data de início, tempo de processamento já decorrido, volume de dados lidos/escritos, etc.);

Trocas de Contexto

- O ato de gravar as informações de uma tarefa permitindo-a ser executada posteriormente é chamado de **chaveamento de contexto**;
 - Por ser uma operação complexa e envolver grande quantidade de manipulação de registradores normalmente esta rotina é implementada em linguagem de máquina;
- Em um chaveamento de contexto existem questões de ordem mecânica e de ordem estratégica a serem resolvidas:
 - Aspectos Mecânicos:
 - O **dispatcher** é responsável pela recuperação do contexto e atualização das informações contidas no TCB;
 - Aspectos Estratégicos:
 - O **scheduler** é responsável por decidir, com base em algum fator (prioridade, tempo de processamento, etc), a próxima tarefa que deverá ser executada pelo processador;

Trocas de Contexto

- A figura abaixo apresenta um diagrama temporal com os principais passos envolvidos em uma troca de contexto;
- É uma operação rápida (casa dos μs);
- A operação mais demorada é a de escalonamento.
 - Por este motivo alguns SO ativam o escalonamento esporadicamente, ou seja, somente quando há a necessidade de se ordenar a fila;

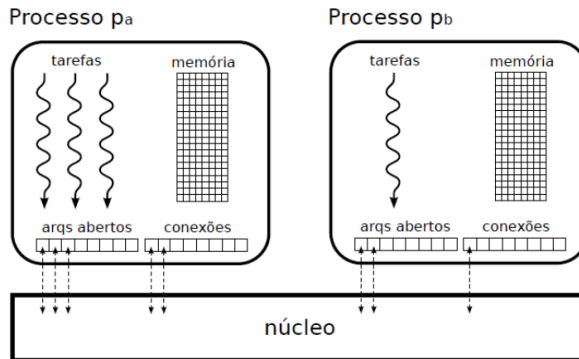


Processos

- Além do seu próprio código executável, cada tarefa ativa em um SO necessita de um conjunto de recursos para executar e cumprir os seus objetivos;
 - Este recursos são áreas de memória, arquivos abertos, conexões de rede, etc;
- O conjunto dos recursos alocados a uma tarefa para a sua execução é denominado **processo**;
- Historicamente o conceito de tarefa e processo se confundem;
 - Isto deve-se aos SO antigos, onde somente era suportado uma tarefa para cada processo;
- Algumas referências mais antigas mantêm este conceito até hoje;
- No entanto, os SO atuais suportam mais de uma tarefa por processo (ex. Linux, WinXP, Unix);
- Atualmente o processo deve ser visto como uma unidade de contexto, ou seja, um *contêiner* de recursos utilizados por uma ou mais tarefas para a sua execução;

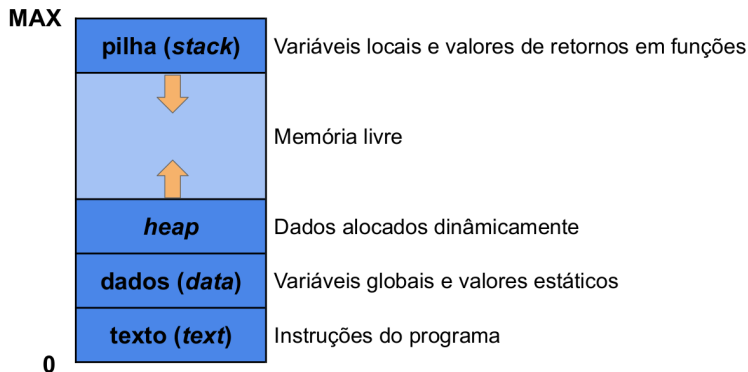
Processos

- O *kernel* do SO mantém descritores de processos chamados PCB (*Process Control Blocks*), onde cada um possui um PID (*Process Identifier*) único;
 - Desta forma o TCB pode ser simplificado;



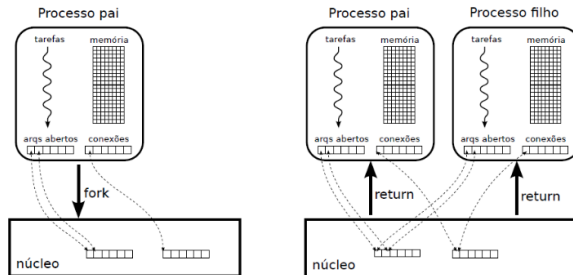
Processos

- A alocação da área de memória de um processo comumente se apresenta assim:



Processos - Criação de Processos

- Durante a operação do SO diversos processos podem ser criados e destruídos através de chamadas de sistema;
 - Cada SO possui sua chamada de sistema específica.
- No caso do Unix, a chamada de sistema **fork** cria uma réplica do processo solicitante;
 - Toda a informação contida nos descritores é copiada;



Processos - Criação de Processos

- Na operação de criação de processos no Unix fica clara a noção de **hierarquia entre processos**;
- À medida que os processos são criados estes formam uma **árvore de processos** (pais e filhos);
- Isto pode ser utilizado para gerenciar grupos de processos através de **sub árvores de processos**;
 - Por exemplo, caso o processo pai seja destruído, todos os processos filhos podem decidir entre encerrarem ou então continuarem a sua execução;

Processos - Comunicação entre Processos

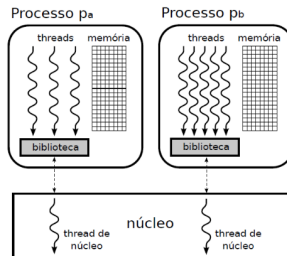
- Outro aspecto importante é a comunicação entre processos;
- Tarefas associadas ao mesmo processo podem trocar informações facilmente, pois compartilham a mesma área de memória;
- Porém, isto não é possível entre tarefas associadas a processos distintos;
- Para solucionar isto o *kernel* deve prover chamadas de sistema que permitam a comunicação entre processos (IPC – *Inter-Process Communication*);

Threads

- No início, os SOs suportavam apenas uma tarefa por processo, isto era um claro inconveniente;
 - Por exemplo, um editor de texto executa várias tarefas em simultâneo, como edição, formatação, paginação, verificação de ortografia. Em servidores isto torna-se ainda mais crítico;
- Assim, de forma geral, cada fluxo de execução do sistema, seja este associado a um processo ou ao *kernel* do SO, é denominado ***thread***;
- Existem dois tipos de *threads*:
 - ***user threads***: tarefas dos processos de usuário;
 - ***kernel threads***: tarefas do *kernel* do SO;

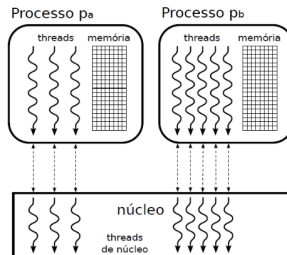
Threads

- Os primeiros SOs não suportavam *threads*;
- Para contornar isto os programadores criaram bibliotecas que permitiam criar e gerenciar *threads* de usuário sem o envolvimento do *kernel*;
- Isto fazia com que uma aplicação pudesse implementar múltiplas *threads* que eram vistas pelo *kernel* como uma única;
- Por isto o modelo recebeu o nome de **Modelo de Threads N:1**;



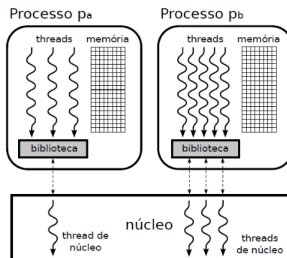
Threads

- Para solucionar este problemas criou-se o **Modelo de *Threads* 1:1**, onde cada *thread* de usuário possui uma referente no *kernel*;
- É adequada para a maioria das situações e atende muito bem aplicações interativas e servidores de rede;
- O seu problema é a sua pouca escalabilidade;
- A criação de muitas *threads* impõem uma carga significativa ao *kernel* do SO;



Threads

- Para resolver o problema da escalabilidade, alguns SO implementam um modelo híbrido, denominado **Modelo de *Threads* N:M**
 - Uma biblioteca gerencia um conjunto de *threads* de usuário que é mapeado em uma ou mais *threads* de núcleo;
 - O conjunto de *threads* de núcleo é geralmente composto por uma *thread* para cada tarefa bloqueada mais uma *thread* para cada processador disponível;
- Assim, N *threads* de usuário são mapeadas em $M \leq N$ *threads* de núcleo;



Threads

- No passado, cada SO implementava sua própria interface para a criação e gerência de *threads*, o que resultou em problemas de portabilidade das aplicações;
- Em 1995, foi definido o padrão IEEE POSIX 1003.1c, mais conhecido como **PThreads**;
 - Define uma interface padronizada para a criação e manipulação de *threads* na linguagem C;
- O conceito de *threads* também pode ser implementado em outras linguagens de programação como Java, Python, Perl, C++, C#, etc;

Threads

- Benefícios da utilização de *threads*:
 - **Capacidade de resposta:** o uso de vários *threads* em uma aplicação interativa pode permitir que o processo continue interagindo mesmo quando alguma tarefa estiver bloqueada pelo SO ou executando alguma operação demorada. Ex. 1: Receber um comando de “pausa” enquanto reproduz um vídeo ou música; Ex. 2: Decodificar um vídeo enquanto carrega o arquivo do disco;
 - **Compartilhamento de recursos:** os processos só podem compartilhar recursos através de técnicas como memória compartilhada ou troca de mensagens. Estas técnicas devem ser organizadas explicitamente pelo programador. Já as *threads*, por padrão, compartilham o mesmo espaço de memória do processo, podendo utilizar os mesmos recursos (ex.: arquivos, variáveis, etc...);

Threads

- Benefícios da utilização de *threads*:
 - **Economia:** a alocação de memória para a criação de processos é dispendiosa. Já que *threads* utilizam a mesma memória do processo, é mais econômico criar *threads* e alternar seus contextos. Exemplos apresentado por Silberschatz:
 - A criação de um processo no Solaris pode ser até 30 vezes mais demorada do que a criação de uma *thread*;
 - A troca de contexto entre processos pode ser até 5 vezes mais demorada do que a troca entre *threads* de um mesmo processo;
 - **Escalabilidade:** os benefícios do uso de várias *threads* podem ser muito maiores em arquiteturas com muitos processadores. Assim, cada processador estará ocupado executando uma determinada *thread* em um dado momento. Um processo com uma única *thread* poderá ser executado por apenas um processador. A combinação do uso de *threads* com múltiplos processadores aumenta o paralelismo;

Threads

- Devido a restrições tecnológicas para o aumento na velocidade de processadores, uma das soluções encontradas pelos fabricantes foi a de aumentar o número de núcleos por invólucro em um processador;
- Os Sistemas Operacionais vem ao longo do tempo implementando o recurso de multiprogramação e a possibilidade dos multiprogramas executarem, de fato ao mesmo tempo, trouxe mais força ainda para técnicas e tecnologias de paralelização;
- **Importante:**
 - 4 *threads* se revezando em 1 núcleo = concorrência;
 - 4 *threads* cada uma executando em 1 núcleo = paralelismo;

Threads

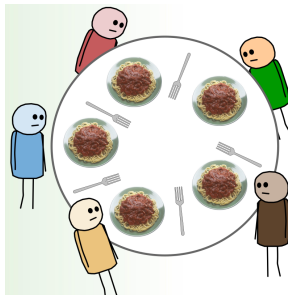
- Em geral, este paralelismo apresenta 5 principais desafios:
 - **Divisão de atividades:** a análise das aplicações em busca de atividades que possam ser divididas em tarefas separadas;
 - **Equilíbrio:** diferentes *threads* devem ter esforço parecido, para que existam paralelamente por mais tempo. Caso o esforço seja distribuído mal, as *threads* não usufruirão o máximo possível de paralelismo;
 - **Divisão de dados:** da mesma forma que as aplicações devem ser divididas em tarefas separadas, os dados acessados e manipulados por elas devem ser divididos para a execução em núcleos separados;
 - **Dependência de dados:** às vezes os dados de saída de uma *thread* servirão como dados de entrada para outra *thread*. Neste caso a sincronização deve ser planejada para resolver estas dependências;
 - **Teste e depuração:** quando um programa está sendo executado em paralelo por vários núcleos, há muitos caminhos de execução diferentes. O teste e a depuração desses programas concorrentes são inerentemente mais difíceis do que o teste e a depuração de aplicações com uma única *thread*;

Sincronização e Impasses

- A sincronização nada mais é do que a criação de estratégias, utilizando diferentes recursos, para evitar problemas que surgem devido à paralelização de tarefas;
- Impasses, também conhecidos como **deadlocks**, são situações onde a falta de sincronização pode ocasionar o travamento definitivo de duas ou mais tarefas paralelas;
- A seguir veremos os problemas:
 - Jantar dos filósofos;
 - Condição de corrida;
- Após a definição dos problemas, conheceremos ferramentas de sincronização;

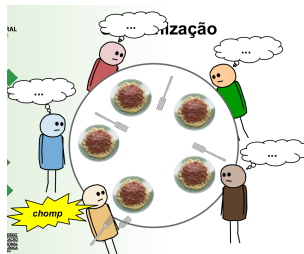
Sincronização e Impasses

- Jantar dos Filósofos:
 - Imagine uma mesa, em um restaurante de comida italiana;
 - 5 filósofos sentado à mesa;
 - São servidos 5 pratos de macarronada, com apenas um garfo para cada prato;
 - Em um determinado momento, ou os filósofos estão pensando, ou estão comendo;



Sincronização e Impasses

- Jantar dos Filósofos:
 - Quando está pensando, o filósofo permanece em silêncio, trabalhando apenas com seus próprios pensamentos;
 - Quando um filósofo sente fome e decide comer, ele percebe que precisará de dois garfos para comer e:
 - Ele pega primeiro o garfo da esquerda;
 - E pega depois o garfo da direita;



Sincronização e Impasses

- Jantar dos Filósofos:
 - O que acontece enquanto o primeiro filósofo que pegou os garfos estiver comendo?
 - Os demais filósofos podem continuar apenas pensando;
 - Algum, ou todos, os filósofos podem querer comer também!
 - O problema surge quando algum filósofo quer comer, mas há algum filósofo ao lado que já está comendo...
 - Na combinação apresentada, é natural imaginar que um filósofo sem garfos irá esperar pela liberação dos mesmos para poder comer:
 - No cenário computacional, nem sempre isto acontece. Ao se tentar utilizar um recurso e ele não estar disponível, diferentes erros podem ocorrer;
 - Também existem casos onde um processo pode ficar sujeito à espera permanente e “morrer de fome”, o que chamamos de estado de **inanição** (*starvation*);

Sincronização e Impasses

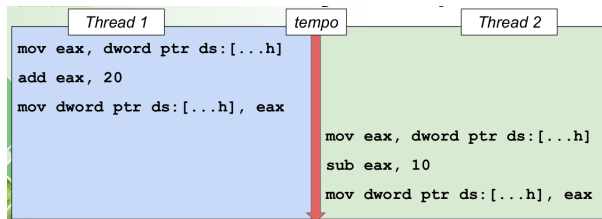
- Jantar dos Filósofos:
 - Agora vamos imaginar outro cenário, onde todos ficam com fome ao mesmo tempo;
 - Neste caso ocorre um impasse, ou *deadlock*:
 - Todos os processos/threads estão segurando um recurso enquanto aguardam por outro recurso, formando uma espera circular e sem fim;
 - Estes problemas são constantemente apresentados em softwares (incluindo os SOs), e são os SOs que devem dar subsídios para a resolução destes conflitos;

Sincronização e Impasses

- Condição de Corrida:
 - É quando processos que são executados em paralelo dependem, em um determinado momento, de um mesmo recurso compartilhado;
 - Conforme o escalonamento dos processos, há chance de ambos processos terem acesso primeiro ao recurso, ou ainda, terem uso do processador alternado durante o período de utilização de um recurso;
 - Exemplo: variável numérica
 - Imagine duas *threads* manipulando uma variável numérica paralelamente;
 - Uma *thread* está somando um valor à variável (T_1 soma 20);
 - A outra, está subtraindo um valor da variável (T_2 subtrai 10);

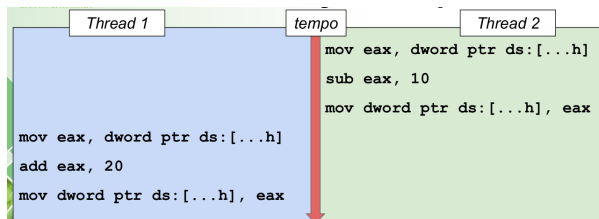
Sincronização e Impasses

- Condição de Corrida (Cenário 1):
 - Assumindo que a variável Total é inicializada com o valor 100;
 - T_1 carrega eax_1 com 100 de Total;
 - T_1 adiciona 20 à eax_1 que vira 120;
 - T_1 guarda 120 em Total;
 - T_2 carrega eax_2 com 120 de Total;
 - T_2 diminuiu 10 de eax_2 que vira 110;
 - T_2 guarda 110 em Total;



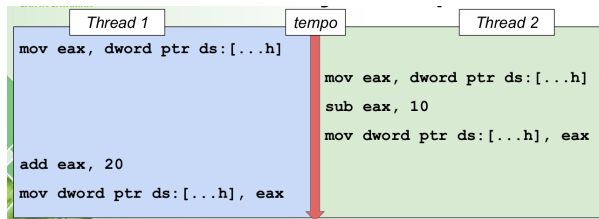
Sincronização e Impasses

- Condição de Corrida (Cenário 2):
 - Assumindo que a variável Total é inicializada com o valor 100;
 - T_2 carrega eax_2 com 100 de Total;
 - T_2 diminui 10 de eax_2 que vira 90;
 - T_2 guarda 90 em Total;
 - T_1 carrega eax_1 com 90 de Total;
 - T_1 adiciona 20 à eax_1 que vira 110;
 - T_1 guarda 110 em Total;



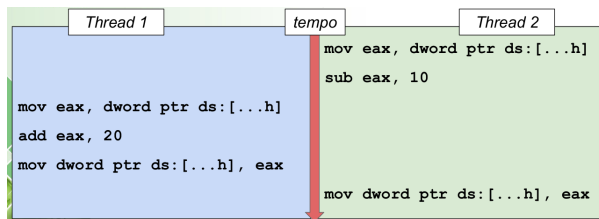
Sincronização e Impasses

- Condição de Corrida (Cenário 3):
 - Assumindo que a variável Total é inicializada com o valor 100;
 - T_1 carrega eax_1 com 100 de Total;
 - T_2 carrega eax_2 com 100 de Total;
 - T_2 diminui 10 de eax_2 que vira 90;
 - T_2 guarda 90 em Total;
 - T_1 adiciona 20 à eax_1 que vira 120;
 - T_1 guarda 120 em Total;



Sincronização e Impasses

- Condição de Corrida (Cenário 4):
 - Assumindo que a variável Total é inicializada com o valor 100;
 - T_2 carrega eax_2 com 100 de Total;
 - T_2 diminui 10 de eax_2 que vira 90;
 - T_1 carrega eax_1 com 100 de Total;
 - T_1 adiciona 20 à eax_1 que vira 120;
 - T_1 guarda 120 em Total;
 - T_2 guarda 90 em Total;

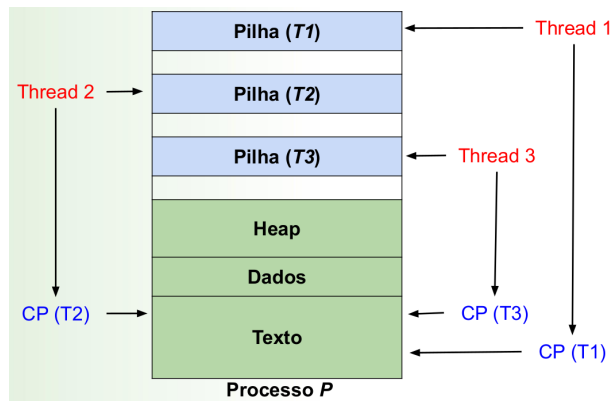


Sincronização e Impasses - Seção Crítica

- A **seção crítica** é um trecho da tarefa que utiliza algum recurso compartilhado e, por isto, não pode ser paralelizado com outras tarefas que utilizem o mesmo recurso (ex.: uma variável);
- Qualquer solução para implementar a seção crítica deve obedecer três regras fundamentais:
 - **Exclusão mútua:** enquanto um processo estiver executando sua seção crítica, outros processo não poderão executar suas seções críticas;
 - **Progresso:** quando não há processos em seção crítica, mas um conjunto deles quiserem entrar, a decisão deve levar em conta apenas os candidatos aptos;
 - **Espera limitada:** há um limite de vezes para que processos entrem em seção crítica após outro processo solicitar para entrar em seção crítica;

Sincronização e Impasses - Hardware de Sincronização

- Variáveis compartilhadas são exemplos de recursos que podem ser compartilhados entre diferentes *threads*;



Sincronização e Impasses - Hardware de Sincronização

- O que é compartilhado entre *threads*:
 - Variáveis globais e estáticas (dados);
 - Memória alocada dinamicamente (*heap*);
- O que não é compartilhado entre *threads*:
 - Dados de pilha!
 - Variáveis locais de *threads*;
 - Obs: nunca compartilhe ponteiros de variáveis locais entre diferentes *threads*;
- O hardware oferece algumas funcionalidades básicas que auxiliam na tarefa do SO:
 - **Execução atômica** é aquela que é capaz de realizar uma “leitura/operação/escrita” sem interrupção;
 - **lock**: é possível realizar uma operação atômica solicitando exclusividade sobre uma variável compartilhada;
 - Exemplo em Assembly: `lock cpxchg edx, end_var`

Sincronização e Impasses - Hardware de Sincronização

- **Lock e execuções atômicas** são recursos da CPU que suportam a construção de outros recursos de sincronização disponibilizados pelos Sistemas Operacionais;
- Veremos os recursos mais comuns a seguir:
 - Semáforos;
 - Monitores;

Sincronização e Impasses - Semáforos

- Um semáforo é um mecanismo que permite a exclusão mútua ao acesso de um determinado recurso;
- Com semáforos é possível controlar as seções críticas em tarefas paralelas;
- Semáforos são implementados pelo SO utilizando recursos de hardware previamente apresentados;
- Um semáforo é uma variável que só pode ter seu valor alterado através de três operações:
 - Criação do semáforo:
 - Define o valor inicial do semáforo (ex: 1);
 - Espera:
 - Enquanto o valor do semáforo for 0, a tarefa permanece em situação de espera;
 - Assim que se tornar 1, a tarefa que conseguir diminuir o valor do semáforo para 0 poderá entrar na seção crítica;
 - Sinaliza:
 - Quando a tarefa terminar sua seção crítica, avisará o semáforo adicionando 1 ao seu valor;

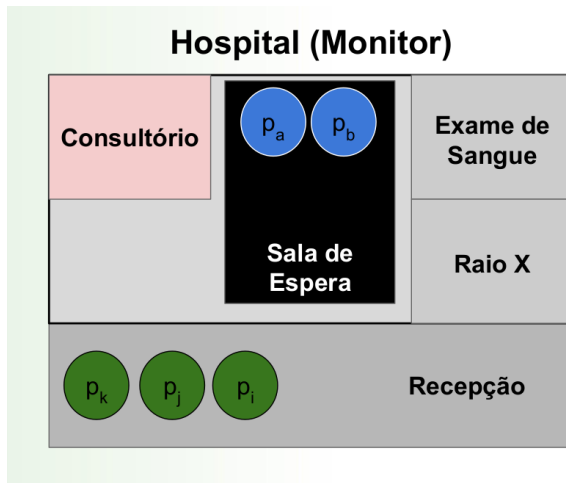
Sincronização e Impasses - Semáforos

- Vantagens da utilização de semáforos:
 - São simples;
 - Trabalham com vários processos;
 - Podem gerenciar diversas seções críticas diferentes, com diferentes semáforos (cada seção crítica deve ter seu próprio semáforo);
 - Pode permitir que a seção crítica seja acessada por mais do que um processo, se desejável;
- Limites na utilização de semáforos:
 - Inversão de prioridade é um limitador (um processo pode sofrer inanição);
 - O uso convencional (o processo não será impedido de utilizar um recurso compartilhado sem consultar o semáforo);
 - Uso inapropriado pode bloquear um processo indefinidamente (*deadlock*);

Sincronização e Impasses - Monitores

- São mecanismos de sincronização de alto nível;
- Utilizam um tipo de dado abstrato para encapsular dados privados com métodos públicos para operar sobre o recurso compartilhado;
- Contempla dados (recursos) e instruções;
- É um conceito abstrato, mas pode ser análogo ao exemplo de um hospital:
 - Quando precisa utilizar, entra em uma fila de recepção para ser atendido;
 - Pode ser FIFO ou pode contemplar prioridades;
 - Ao ser atendido, sai da fila de recepção e entra no consultório;
 - O consultório pode atender no máximo um paciente por vez;
 - Durante o atendimento o médico pode solicitar um exame:
 - Libera o consultório temporariamente, assim outra pessoa passará a ser atendida;
 - Aguarda o resultado do exame em uma sala espera de espera (fila de espera);
 - Todos que estão na sala de espera precisam ser notificados quando o consultório está novamente disponível;

Sincronização e Impasses - Monitores

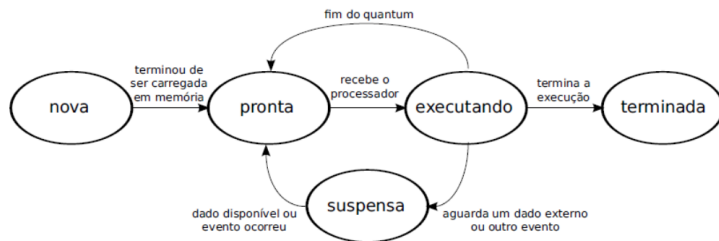


Escalonamento

- A **multiprogramação** surgiu para diminuir o tempo em que recursos computacionais, mas principalmente o processador, permaneciam ociosos;
- A simples busca de informações na memória principal leva um tempo longo em relação às operações do processador:
 - Em um sistema executando um único processo, o tempo gasto na espera de operações de E/S são um desperdício;
 - Por exemplo, um HD envolve movimentação mecânica para retornar resultados de solicitações!
- O grande desafio é tornar o sistema tão eficiente e justo quanto possível:
 - Eficiente e justo são termos subjetivos;

Escalonamento

- Tarefa (*task*) - unidade básica de processamento;
- Estado da tarefa - onde a tarefa se encontra no diagrama de execução;
- Fila de pronto (*ready queue*) - fila de tarefas prontas para serem alocadas no processador;
- Preempção - ato (decidido pelo S.O.) de suspender a execução de uma tarefa para a execução de outra;



Escalonamento

- **Escalonamento preemptivo**

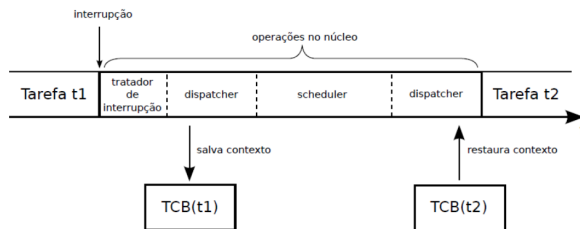
- Uma tarefa pode perder a CPU caso:
 - Termine o seu *quantum*;
 - Execute uma chamada de sistema;
 - Ocorra uma interrupção que acorde uma tarefa mais prioritária;
- A cada interrupção, exceção ou chamada de sistema o escalonador pode reavaliar todas as tarefas na fila de pronto para decidir se mantém ou substitui a tarefa em execução;

- **Escalonamento cooperativo**

- A tarefa de posse da CPU permanece de posse dela quanto tempo for necessário, somente liberando-a caso termine a sua execução, realize uma operação de I/O ou libere explicitamente a CPU;
- Esta liberação é feita pela chamada de sistema `_sched_yield()`;

Escalonamento

- Contexto - conjunto de informações necessárias para que uma tarefa continue a execução de onde parou;
- Chaveamento de contexto - operação de gravação e recuperação de contextos de tarefas que entram e saem do processador;
- *Dispatcher* - responsável pelo chaveamento de contexto;
- *Scheduler* - responsável por decidir qual tarefa será a próxima a alocar o processador;



Escalonamento

- O escalonador de tarefas (*task scheduler*) é um dos componentes mais importantes na gerência de tarefas;
- O algoritmo implementado pelo escalonador define o comportamento do sistema operacional;
- Ao implementar-se um algoritmo de escalonamento deve-se levar em consideração alguns comportamentos das tarefas;
- Neste contexto, algumas classificações pode ser definidas:
 - **Tarefas de tempo real:** geralmente estão associadas ao controle de sistemas críticos e exigem previsibilidade no tempo de resposta;
 - **Tarefas interativas:** necessitam obter tempos de resposta reduzidos, mas não existe uma limitação temporal;
 - **Tarefas orientadas a processamento:** utilizam intensivamente o processador na maior parte da sua execução (ex.: processamentos numéricos);
 - **Tarefas orientadas a entrada/saída:** dependem muito mais dos dispositivos I/O que do processador (ex.: comunicação de rede);

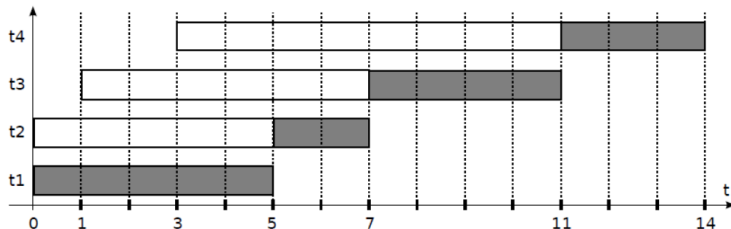
Escalonamento

- Algumas métricas utilizadas para sua avaliação são:
 - **Tempo de execução (*turnaround time*)**: tempo total da vida de uma tarefa; não deve ser confundida com o tempo de computação (*computation time*), o qual define o tempo de processamento;
 - **Tempo de espera (*waiting time*)**: tempo aguardando na fila de pronto (não inclui tempos de I/O);
 - **Tempo de resposta (*response time*)**: tempo entre a chegada de um evento ao sistema e o seu resultado;
 - **Justiça (*fairness*)**: define a distribuição dos tempos de computação entre as tarefas;
 - **Utilização (*utilization*)**: indica o grau de utilização da CPU na execução de tarefas do usuário;

Escalonamento - FCFS (*First Come First Served*)

- O algoritmo de escalonamento mais elementar;
- Também conhecido como FIFO (*First In First Out*);
- Consiste em atender as tarefas na ordem de chegada à fila de pronto;

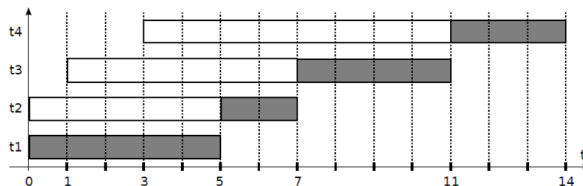
Tarefa	T1	T2	T3	T4
Ingresso	0	0	1	3
Duração	5	2	4	3



Escalonamento - FCFS (*First Come First Served*)

- $U = \frac{T_{CPU}}{\text{tempo}} = \frac{14}{14} = 1 = 100\%$
- $Prod = \frac{\text{num.proc.}}{\text{tempo}} = \frac{4}{14} = 0,2857$
- $WT_{avg} = \frac{\sum_{i=1}^N WT_i}{\text{num.proc.}} = \frac{0+5+6+8}{4} = \frac{19}{4} = 4,75$
- $TAT_{avg} = \frac{\sum_{i=1}^N TAT_i}{\text{num.proc.}} = \frac{5+7+10+11}{4} = \frac{35}{4} = 8,75$

Tarefa	T1	T2	T3	T4
Ingresso	0	0	1	3
Duração	5	2	4	3



Escalonamento - FCFS (*First Come First Served*)

- **Vantagens**

- Simples de implementar;

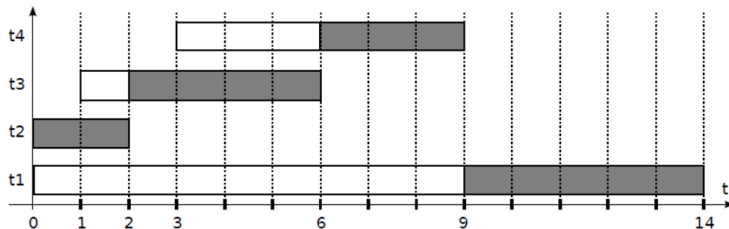
- **Desvantagens**

- Dependente da ordem de chegada dos processos;
- Altos tempos de espera;
- Tende a favorecer aos processos com muita carga de CPU, prejudicando aos processos intensivos de E/S;

Escalonamento - SJF (*Shortest Job First*)

- O processador é alocado ao processo com etapa de CPU mais breve;
- Em caso de empate utiliza-se FIFO;
- Não preemptivo;

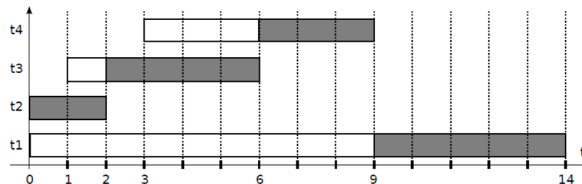
Tarefa	T1	T2	T3	T4
Ingresso	0	0	1	3
Duração	5	2	4	3



Escalonamento - SJF (*Shortest Job First*)

- $U = \frac{T_{CPU}}{tempo} = \frac{14}{14} = 1 = 100\%$
- $Prod = \frac{num.proc.}{tempo} = \frac{4}{14} = 0,2857$
- $WT_{avg} = \frac{\sum_{i=1}^N WT_i}{num.proc.} = \frac{9+0+1+3}{4} = \frac{13}{4} = 3,25$
- $TAT_{avg} = \frac{\sum_{i=1}^N TAT_i}{num.proc.} = \frac{14+2+5+6}{4} = \frac{27}{4} = 6,75$

Tarefa	T1	T2	T3	T4
Ingresso	0	0	1	3
Duração	5	2	4	3



Escalonamento - SJF (*Shortest Job First*)

- **Vantagens**

- Reduz o tempo de espera médio;
- Minimiza o efeito de priorizar processos do tipo *CPU-bound*;

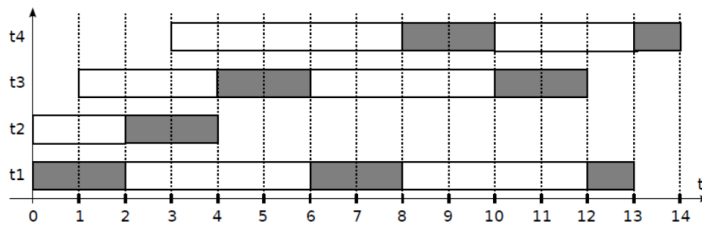
- **Desvantagens**

- Difícil determinar *a priori* qual será a duração da seguinte etapa de CPU dos processos;
- *Starvation* por sobrecarga de tarefas curtas;

Escalonamento - *Round Robin*

- Versão preemptiva do algoritmo FCFS;

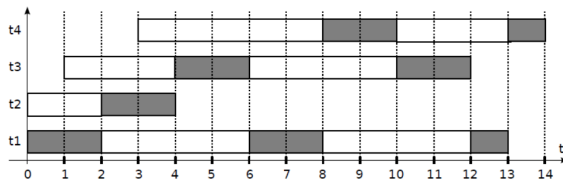
Tarefa	T1	T2	T3	T4
Ingresso	0	0	1	3
Duração	5	2	4	3
Quantum	2	2	2	2



Escalonamento - *Round Robin*

- $U = \frac{T_{CPU}}{\text{tempo}} = \frac{14}{14} = 1 = 100\%$
- $Prod = \frac{\text{num.proc.}}{\text{tempo}} = \frac{4}{14} = 0,2857$
- $WT_{avg} = \frac{\sum_{i=1}^N WT_i}{\text{num.proc.}} = \frac{8+2+7+8}{4} = \frac{25}{4} = 6,25$
- $TAT_{avg} = \frac{\sum_{i=1}^N TAT_i}{\text{num.proc.}} = \frac{13+4+11+11}{4} = \frac{39}{4} = 9,75$

Tarefa	T1	T2	T3	T4
Ingresso	0	0	1	3
Duração	5	2	4	3
Quantum	2	2	2	2



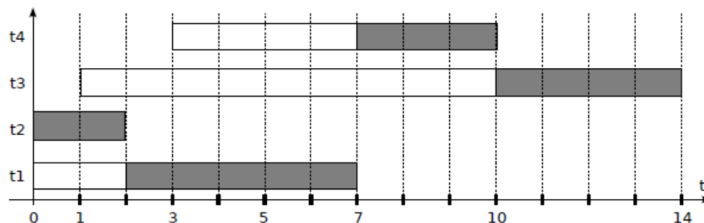
Escalonamento - SRTF (*Short Remaining Time First*)

- Versão preemptiva do algoritmo SJF;
- Resolve o problema de previsibilidade do tempo de computação das tarefas realizando uma média dos tempos de utilização anteriores;
- Mesmo assim, mudanças repentinas de comportamento podem desviar o valor calculado;

Escalonamento por Prioridades

- A cada tarefa é associada uma prioridade (ex.: número inteiro);
- Esta prioridade será utilizada pelo escalonador para organizar as tarefas;
- Exemplo de escalonamento **não preemptivo** por prioridades:

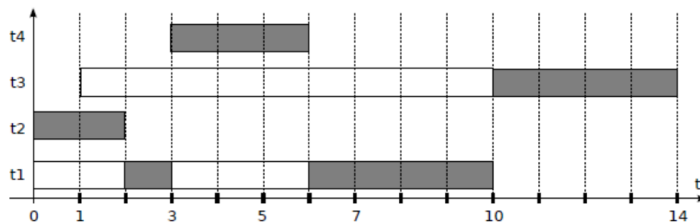
Tarefa	T1	T2	T3	T4
Ingresso	0	0	1	3
Duração	5	2	4	3
Prioridade	2	3	1	4



Escalonamento por Prioridades

- A cada tarefa é associada uma prioridade (ex.: número inteiro);
- Esta prioridade será utilizada pelo escalonador para organizar as tarefas;
- Exemplo de escalonamento **preemptivo** por prioridades:

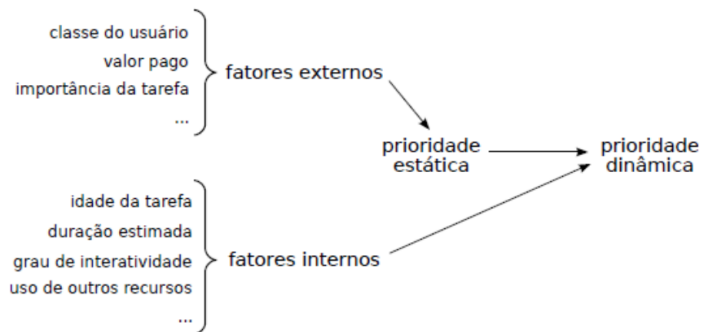
Tarefa	T1	T2	T3	T4
Ingresso	0	0	1	3
Duração	5	2	4	3
Prioridade	2	3	1	4



Escalonamento por Prioridades - Divisão de Prioridades

- Fatores externos:
 - Informações providas pelo usuário ou administrador do sistema;
 - O escalonador não consegue estimar sozinho;
 - Ex.: classe do usuário, importância da tarefa;
- Fatores internos:
 - Informações obtidas ou estimadas pelo escalonador;
 - Ex.: idade da tarefa, duração estimada, interatividade, uso de memória;
- A combinação destes fatores define um valor de prioridade para a tarefa;
- Os fatores externos são expressos por valores inteiros denominados prioridade estática (ou prioridade de base), resumindo assim a opinião do usuário ou do administrador sobre a tarefa;
- Os fatores internos mudam continuamente e devem ser recalculados periodicamente pelos escalonador;
- A combinação da prioridade estática com os fatores internos resulta na prioridade dinâmica (ou final) que será utilizada pelo escalonador;

Escalonamento por Prioridades - Divisão de Prioridades

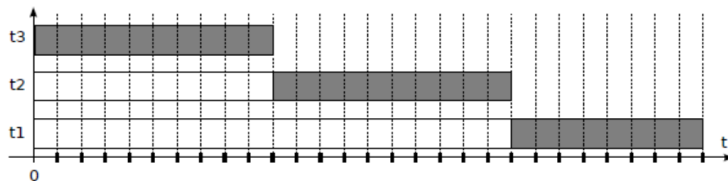


Escalonamento por Prioridades - *Starvation*

- É a incapacidade de uma tarefa de baixa prioridade receber a CPU devido a frequente entrada de tarefas de mais alta prioridade na fila de pronto;
- Em sistemas de tempo compartilhado, as prioridades estáticas estão intuitivamente relacionadas à proporcionalidade na divisão de tempo;
 - Ex.: duas tarefas iguais e com a mesma prioridade ocuparão 50% da CPU cada;

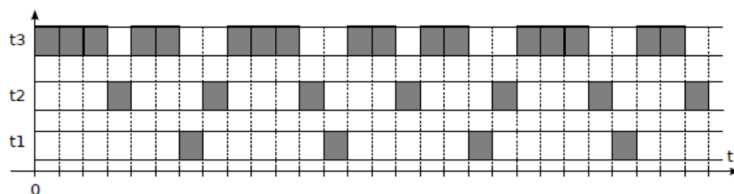
Escalonamento por Prioridades - *Starvation*

- Caso o sistema receba 3 tarefas no mesmo instante de tempo:
 - T_1 - prioridade 1;
 - T_2 - prioridade 2;
 - T_3 - prioridade 3;
- Isto resultará em uma execução sequencial, mesmo com a aplicação do *quantum*;



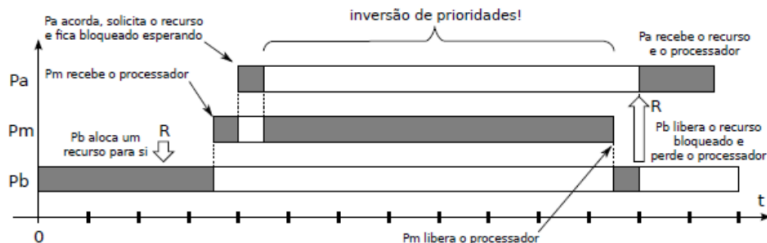
Escalonamento por Prioridades - *Task Aging*

- Para evitar *starvation* e garantir a proporcionalidade das prioridades estáticas, um fator interno chamado envelhecimento (*task aging*) é aplicado sobre a prioridade de cada tarefa;



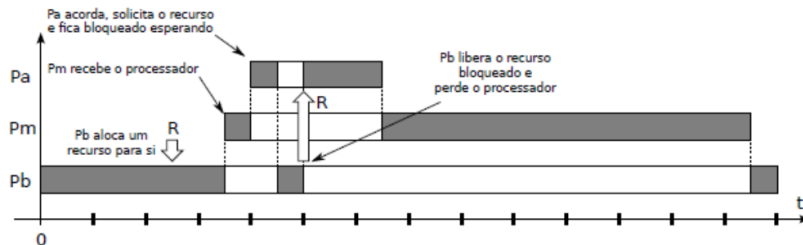
Escalonamento por Prioridades - Inversão de Prioridade

- Este problema ocorre quando uma tarefa qualquer bloqueio o acesso ao recurso de uma tarefa de mais alta prioridade;
- Este problema envolve o conceito de exclusão mútua;
 - A exclusão mútua é uma técnica utilizada para garantir consistência em sistemas multithread;



Escalonamento por Prioridades - Herança de Prioridade

- Uma solução elegante para o problema de inversão de prioridade é a utilização do protocolo de herança de prioridade;
- Consiste em aumentar temporariamente a prioridade de uma tarefa que detenha um recurso solicitado por outra tarefa de maior prioridade;
 - Caso P_b obtenha o recurso R e posteriormente P_a solicite este recurso, P_b então “herda” temporariamente a prioridade de P_a para que este possa voltar a executar e libere o recurso rapidamente à P_a ;



Sistemas de Tempo Real

- Tornam-se cada vez mais comuns;
- Variam entre a complexidade e necessidade de garantias temporais;
- Entre os mais simples estão:
 - Controladores embutidos em máquinas de lavar roupa, sistemas de comunicação de voz sobre IP (VoIP), etc...
- Entre os mais complexos estão:
 - Sistemas de defesa militar, sistemas de controle de plantas industriais, sistemas de controle de tráfego aéreo, sistemas de monitoramento de pacientes, etc...

Sistemas de Tempo Real

- Sistemas Transformacionais
 - Calculam os valores de saída a partir de valores de entrada e após terminam os seus processamentos;
 - Ex.: compiladores, programas de cálculo numérico;
- Sistemas Reativos
 - Reagem enviando respostas continuamente à estímulos de entrada vindos de seu ambiente;
 - Ex.: *kernel* de um SO, interfaces de comunicação;
-
- Sistemas de tempo real encaixam-se no conceito de sistemas reativos;

Sistemas de Tempo Real

O que é um Sistema de Tempo Real?

Um sistema de tempo real (*real-time system*) é um sistema computacional que deve reagir a estímulos oriundos do seu ambiente dentro de limites temporais (*deadlines*) previamente especificados.

- Portanto, o comportamento correto de um sistema de tempo real não depende somente da integridade dos resultados obtidos (*correctness*) mas também dos valores de tempo em que estes são produzidos (*timeliness*);

Sistemas de Tempo Real

- A crença mais comum é que este tipo de problema pode ser solucionado através do aumento da velocidade computacional;
 - Esta abordagem na verdade visa minimizar o tempo de resposta médio de um conjunto de tarefas, enquanto que o objetivo de um sistema de tempo real é o atendimento dos requisitos temporais de todas as tarefas caracterizadas como tempo real;
- Desta forma, o conceito de previsibilidade é mais importante do que a rapidez computacional;

Sistemas de Tempo Real

- Um sistema de tempo real é dito previsível (***predictable***) no domínio lógico e temporal quando, independentemente de variações à nível de *hardware*, da carga computacional e de falhas, o comportamento do sistema pode ser antecipado previamente à sua execução, ou seja, em tempo de projeto;
- Este conceito também é chamado de **determinismo**, resultando assim em **sistemas determinísticos**;

Sistemas de Tempo Real

- Dada esta dependência com o tempo, alguns dos aspectos mais importantes que caracterizam um sistema de tempo real são:
 - Tempo de execução dos processamentos;
 - Tempo de resposta dos eventos;
 - Regularidade na geração de eventos periódicos;

Sistemas de Tempo Real

- Estes aspectos são influenciados por diversos parâmetros internos:
 - Tempo de execução dos processamentos
 - Estrutura do código fonte (linguagem, condicionais, ciclos);
 - DMA, *caches*, instruções de *pipeline*;
 - Sistemas operacionais ou *kernels* (chamadas de sistema);
 - Tempo de resposta dos eventos
 - Interrupções de *hardware*;
 - *Multitasking*;
 - Acesso à recursos compartilhados (ex.: barramento, discos, portas);

Sistemas de Tempo Real

- Tipicamente, existem três tipos de requisitos impostos aos sistemas de tempo real:
 - Funcional (*Functional*);
 - Temporal (*Temporal*);
 - Confiança no funcionamento (*Dependability*);

Sistemas de Tempo Real

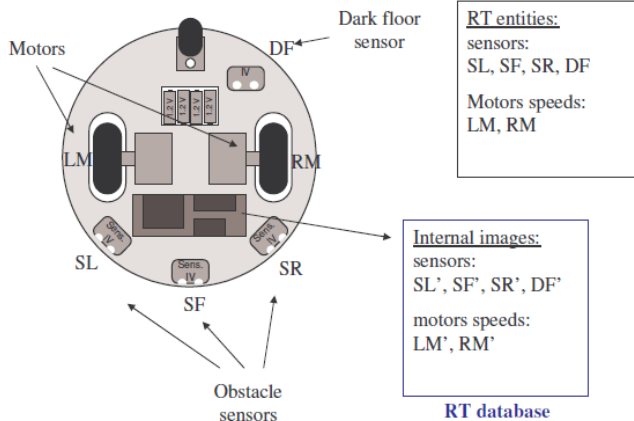
- Requisitos funcionais (exemplos):
 - Aquisição de dados do ambiente;
 - Amostragem das variáveis de processo (entidades de tempo real);
 - *Direct Digital Control* (DDC);
 - Acesso direto pelo controlador digital aos sensores/atuadores;
 - *Human-Machine Interface* (HMI)
 - Fornece informações sobre o estado do sistema, realiza log e configurações;

Sistemas de Tempo Real

- Requisitos funcionais (exemplos):
 - Aquisição de dados do ambiente;
 - As entidades dos processo de tempo real, no momento da aquisição dos dados, são internamente acessíveis ao sistema de computação através de uma imagem local (variáveis internas);
 - Cada imagem local de uma entidade de tempo real possui uma validade temporal limitada devido à dinâmica do processo;
 - Um grupo de imagens locais de entidades de tempo real formam uma banco de dados tempo real;
 - Este banco de dados precisa ser atualizado sempre que ocorra alguma modificação em um estado/valor de uma entidade de tempo real;

Sistemas de Tempo Real

- Requisitos funcionais (exemplos):
 - Aquisição de dados do ambiente;

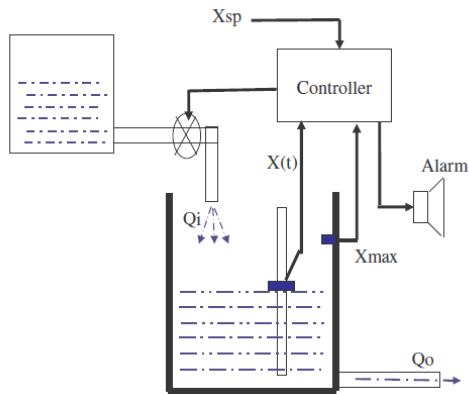


Sistemas de Tempo Real

- Requisitos temporais:
 - Surgem a partir da dinâmica do processo que deve ser controlado/monitorado;
 - Grupo de restrições:
 - Atraso de observação do estado do sistema;
 - Atraso de computação dos valores de controle/atuação;
 - Variação no atraso (*jitter*) de ambos;
 - Estas restrições precisam ser cumpridas em todas as instâncias (incluindo o pior caso) e não somente em sua média;

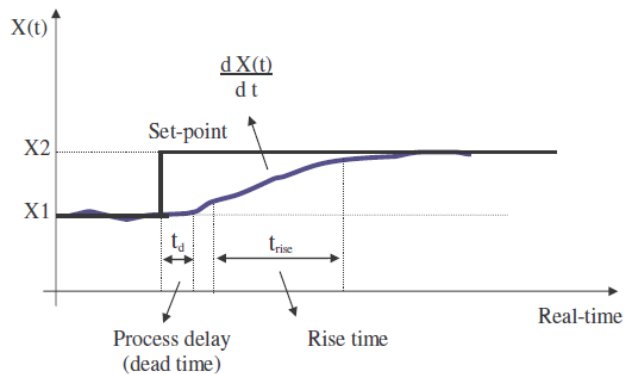
Sistemas de Tempo Real

- Requisitos temporais (exemplo): controle de nível de um tanque



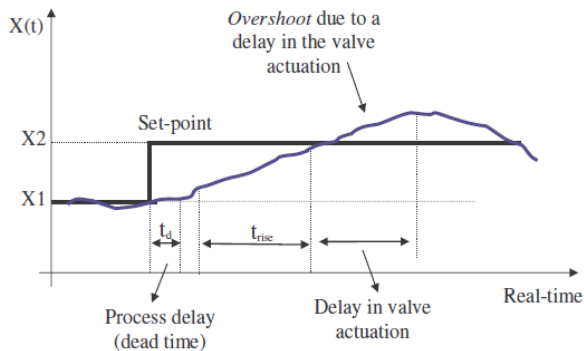
Sistemas de Tempo Real

- Requisitos temporais (exemplo): dinâmica do controle



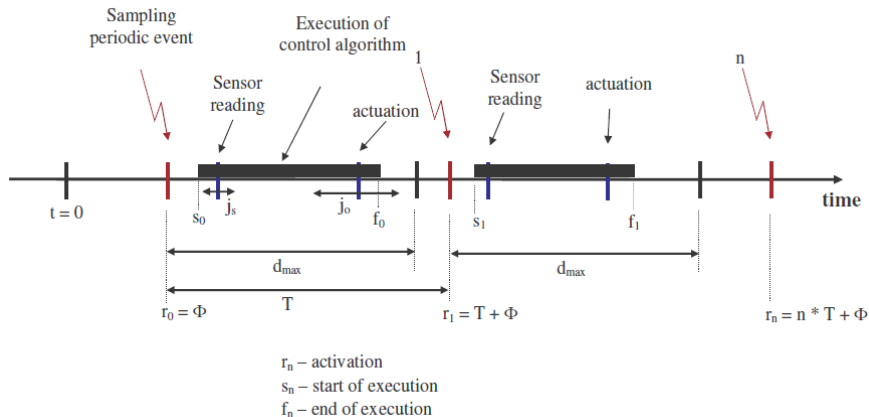
Sistemas de Tempo Real

- Requisitos temporais (exemplo): dinâmica do controle
 - Atraso na atuação resulta em degradação do controle;



Sistemas de Tempo Real

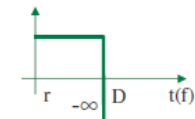
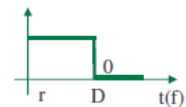
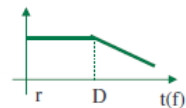
- Requisitos temporais (exemplo): dinâmica do controle



Sistemas de Tempo Real - Classificação

Em função da utilidade do resultado para a aplicação

- **Soft:** o resultado da tarefa ainda possui algum valor mesmo após o *deadline*. A sua qualidade degrada-se com o passar do tempo.
- **Firm:** o resultado da tarefa não possui qualquer validade após o *deadline*.
- **Hard:** uma falha catastrófica poderá ocorrer caso alguma tarefa perca o seu *deadline*.



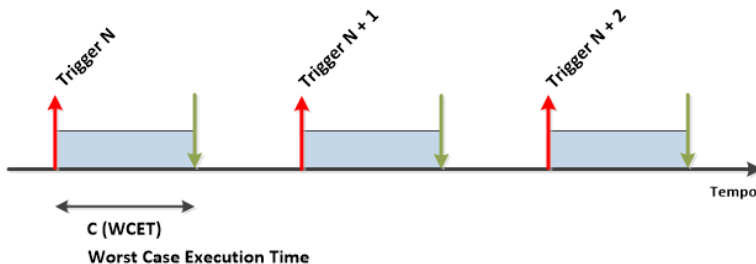
Sistemas de Tempo Real - Classificação

Em função do tipo de restrição temporal da aplicação

- ***Soft Real-Time***: possuem somente as restrições temporais do tipo *firm* e *soft* (ex.: simuladores, sistemas multimídia);
- ***Hard Real-Time***: possui somente restrições temporais do tipo *hard*. São considerados sistemas seguros ou sistemas críticos (ex.: controle de armas, sistemas de transporte);

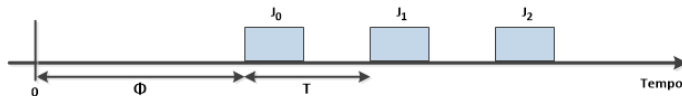
Sistemas de Tempo Real - Tarefas

- Sequência de ativações (instâncias ou *jobs*), cada um executando um conjunto de instruções que, na ausência de outras atividades, é levado a cabo sem interrupção;

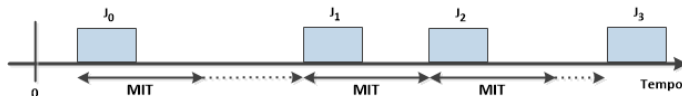


Sistemas de Tempo Real - Tarefas

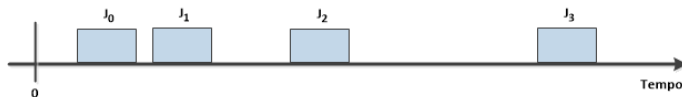
- **periódica:** a tarefa n é ativada a cada $n \times T + \phi$;



- **esporádica:** define um intervalo de tempo mínimo entre ativações consecutivas (*Minimum Interarrival Time* – MIT);

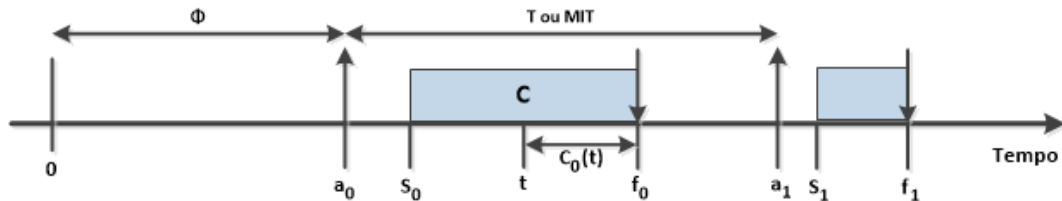


- **aperiódica:** caracterizada somente de forma estocástica;



Sistemas de Tempo Real - Tarefas

- C – *Computation Time* (baseado no WCET);
- T – período (periódica);
- MIT – *Minimum Interarrival Time* (esporádicas);
- ϕ – *offset* relativo = instante da 1ª ativação (periódicas);
- a_n – instante de ativação da tarefa n ;
- S_n – início da execução da tarefa n ;
- f_n – final da execução da tarefa n ;
- $C_n(t)$ – tempo máximo de execução restante da tarefa N no instante t ;



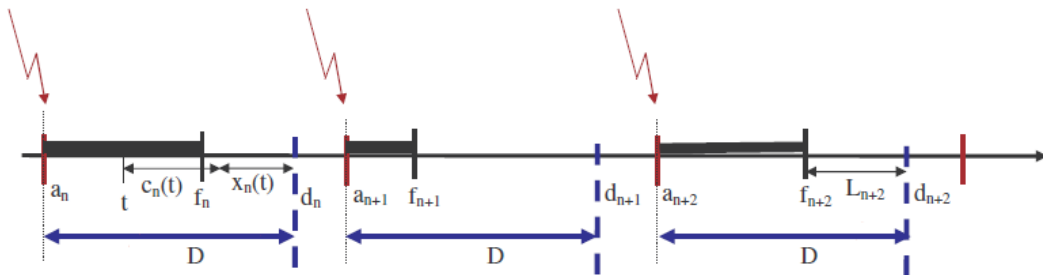
Sistemas de Tempo Real - Requisitos Temporais

- **Deadline:** define um limite superior para o instante de finalização de uma tarefa;
- **Window:** define os limites inferior e superior para o instante de finalização de uma tarefa;
- **Synchronism:** define um limite superior para o intervalo entre dois eventos de saída;
- **Distance:** define um limite superior para o intervalo entre os instantes de finalização de duas tarefas consecutivas;

Sistemas de Tempo Real - Requisitos Temporais

Deadline

- **Deadline Relativo** - $\forall n, f_n - a_n < D$;
- **Deadline Absoluto** - $\forall n, f_n < d_n$;

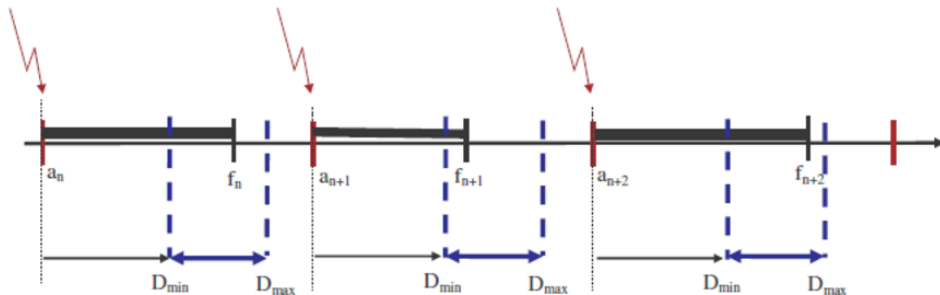


Sistemas de Tempo Real - Requisitos Temporais

Window

- $\forall n, D_{min} < f_n < D_{max};$

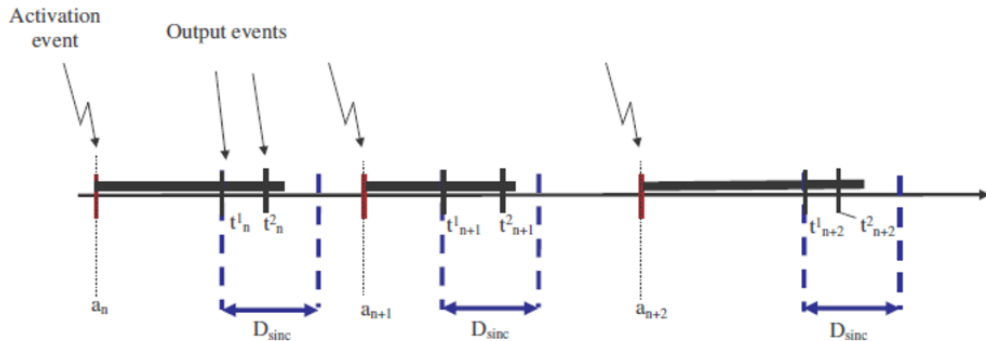
Activation
event



Sistemas de Tempo Real - Requisitos Temporais

Synchronism

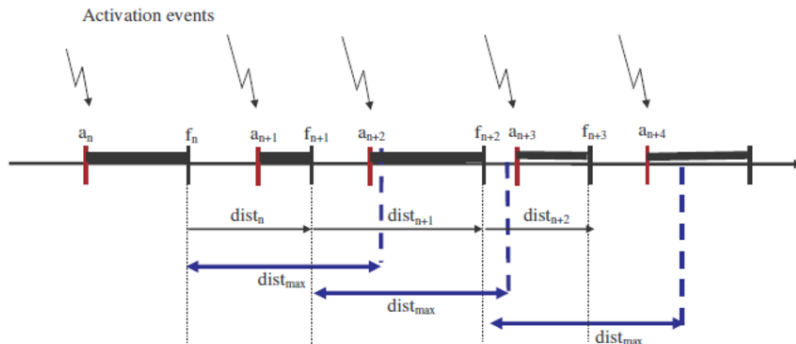
- $\forall n, t_n^2 - t_n^1 < D_{sync};$



Sistemas de Tempo Real - Requisitos Temporais

Distance

- $\forall n, dist_n = f_{n+1} - f_n < dist_{max};$



Sistemas de Tempo Real - Requisitos Temporais

Exemplo de caracterização de tarefas

- Periódica – $\tau_i(C_i, \phi_i, T_i, D_i)$:
 - $\tau_1(2, 5, 10, 10)$;
 - $\tau_2(3, 10, 20, 20)$;
- Esporádica – $\tau_i(C_i, MIT_i, D_i)$:
 - $\tau_1(2, 5, 5)$;
 - $\tau_2(3, 10, 7)$;

Sistemas de Tempo Real - Controles Lógicos e Temporais

- **Controle Lógico**

- Controle de fluxo do programa (ex.: uma sequência de operações que devem ser executadas);
- Fundamental para determinar o $C_i - WCET_i$;

- **Controle Temporal**

- Controle dos instantes de execução das operações do programa (ex.: ativações e verificações de restrições temporais);

Sistemas de Tempo Real - Gatilhos de Tarefas

- **Pelo tempo (*time-triggered*)**
 - A execução das tarefas é disparada por um sinal de controle baseado na progressão do tempo (ex.: interrupção periódica);
- **Por eventos (*event-triggered*)**
 - A execução das tarefas é disparada por um sinal de controle assíncrono gerado por uma mudança no estado do sistema (ex.: interrupção externa);

Sistemas de Tempo Real - Gatilhos de Tarefas

- ***Time-triggered systems***
 - Tipicamente utilizados em sistemas de controle automático (avaliação contínua de variáveis);
 - Possui uma base de tempo comum;
 - A utilização da CPU é constante mesmo ocorrendo variações no estado do sistema;
 - Possui uma situação de pior caso (*worst case*) bem definida;

Sistemas de Tempo Real - Gatilhos de Tarefas

- ***Event-triggered systems***

- Tipicamente utilizados para monitorar condições esporádicas de estado de um sistemas (ex.: alarmes ou requisições de serviço assíncronas);
- A utilização da CPU é variável de acordo com a frequência de ocorrência dos eventos;
- Possui uma situação de pior caso (*worst case*) fracamente definida;
 - Ou são utilizados argumentos probabilistas ou então uma frequência máxima de eventos deve ser definida;

Sistemas de Tempo Real - Escalonamento

- O termo **escalonamento** (*scheduling*) identifica o procedimento de ordenar tarefas na fila de pronto;
- Uma **escala de execução** (*schedule*) é então uma ordenação ou lista que indica a ordem de ocupação do processador por um conjunto de tarefas disponíveis na fila de pronto;
- O **escalador** (*scheduler*) é o componente do sistema responsável em tempo de execução pela gestão do processador;
 - É o escalador que implementa a **política de escalonamento** ao ordenar para a execução sobre o processador um conjunto de tarefas;

Sistemas de Tempo Real - Escalonamento

- As políticas de escalonamento definem critérios ou regras para a ordenação das tarefas de tempo real;
- Os escalonadores utilizando então estas políticas produzem escalas que se forem **realizáveis (*feasible*)**, garantem o cumprimento das restrições temporais impostas às tarefas de tempo real;
- Uma escala é dita ótima se a ordenação do conjunto de tarefas, de acordo com os critérios pré-estabelecidos pela política de escalonamento, é a melhor possível no atendimento das restrições temporais;

Sistemas de Tempo Real - Escalonamento

- **Preemptivo vs. Não-Preemptivo:**
 - Onde as tarefas podem ou não, respectivamente, serem interrompidas por outras tarefas de maior prioridade;
- **Estático vs. Dinâmico:**
 - Se estático, a escala de execução de tarefas é realizada tomando como base parâmetros atribuídos às tarefas do conjunto em tempo de projeto. Caso estes parâmetros sejam atribuídos em tempo de execução (com a evolução do sistema) o escalonamento é dito dinâmico;
- **Off-line vs. On-line:**
 - Se off-line, a escala de execução foi gerada em tempo de projeto. Caso seja online, a escala de execução será produzida em tempo de execução;

Sistemas de Tempo Real - Escalonamento

- **Ótimo vs. Sub-ótimo (heurístico):**
 - O algoritmo ótimo consegue encontrar uma escala realizável sempre que outro algoritmo da mesma classe também consegue. Se o algoritmo ótimo falhar, todos os outros também falharão;
- **Com garantias de WCET vs. Best-effort:**
 - WCET trata sistemas deterministas e *best-effort* trata sistemas dinâmicos (onde a carga e o instante de chegada da tarefa não são conhecidos);

Sistemas de Tempo Real - Escalonamento

Teste de Escalonabilidade

- Tem como objetivo determinar se um conjunto de tarefas é escalonável, ou seja, se existe para este conjunto de tarefas uma escala realizável no sistema;
- Os teste de escalonabilidade variam conforme o modelo de tarefas e políticas definidas em um problema de escalonamento;

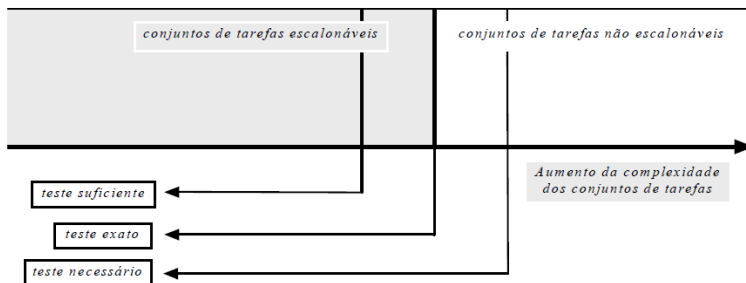


Figura 2.4 : Tipos de testes de escalonabilidade

Sistemas de Tempo Real - Escalonamento

Teste de Escalonabilidade

- **Suficiente**

- São mais simples na execução porém apresentam o custo do descarte de conjuntos de tarefas escalonáveis;
- É um teste mais restritivo onde conjuntos de tarefas aceitos nesses testes certamente são escalonáveis; porém entre os descartados podem existir conjuntos escalonáveis;

- **Necessário**

- Corresponde também a análises simples porém não tão restritivas;
- O fato de um conjunto ter passado por um teste necessário não implica que o mesmo seja escalonável;
- A única garantia que esse tipo de teste pode fornecer é que os conjuntos descartados de tarefas certamente não são escalonáveis;

- **Exato**

- Não afastam conjuntos de tarefas que apresentam escalas realizáveis;
- São precisos na medida em que identificam também conjuntos não escalonáveis;
- Em muitos problemas são impraticáveis os testes exatos;

Sistemas de Tempo Real - Escalonamento

Teste de Escalonabilidade Baseado na Utilização

- A utilização (U_i) de uma tarefa periódica i é dada por:

$$U_i = \frac{C_i}{T_i} \quad (1)$$

- C_i é o tempo de computação da tarefa i ;
- T_i é o intervalo de ativação da tarefa i ;
- A utilização (U_i) de uma tarefa esporádica i é dada por:

$$U_i = \frac{C_i}{MIT_i} \quad (2)$$

- C_i é o tempo de computação da tarefa i ;
- MIT_i é o intervalo mínimo de ativação da tarefa i ;

Sistemas de Tempo Real - Escalonamento

Teste de Escalonabilidade Baseado na Utilização

- A utilização do sistema (U) é dada por:

$$U = \sum_i^n U_i \quad (3)$$

- m é o número de processadores;
- Neste caso:

$$U \leq m \quad (4)$$

Sistemas de Tempo Real - Rate Monotonic (RM)

- Proposto em 1973 por Liu e Layland;
- Utilizado para escalonadores preemptivos dirigidos a prioridades;
- Utiliza um esquema de prioridade fixa, sendo definido então como um escalonamento estático e online;
- É ótimo entre os escalonadores de prioridade fixa na sua classe de problema;
- Premissas:
 - As tarefas são periódicas e independentes;
 - O *deadline* de cada tarefa coincide com o seu período ($D_i = T_i$);
 - O tempo de computação (C_i) de cada tarefa é conhecido e constante (WCET);
 - O tempo de chaveamento entre as tarefas é assumido como nulo;
- As prioridades das tarefas decrescem em função do aumento dos períodos, ou seja, quanto maior a frequência da tarefa maior a sua prioridade no conjunto;
- Como o período das tarefas não mudam, o RM define uma atribuição estática de prioridade;

Sistemas de Tempo Real - Rate Monotonic (RM)

- O teste de escalonabilidade (baseado no cálculo de utilização) do RM define que a seguinte condição suficiente deve ser satisfeita:

$$U = \sum_i^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \quad (5)$$

- onde n é o número de tarefas, C_i e T_i são o tempo de computação e o período de ativação da tarefa i , respectivamente;

$$\lim_{n \rightarrow \infty} n(\sqrt[n]{2} - 1) = \ln 2 \approx 0.6931471805... \quad (6)$$

- Desta forma, conforme n cresce, a utilização do processador converge para 0.69, ou seja, uma utilização de $\approx 70\%$;

Sistemas de Tempo Real - Rate Monotonic (RM)

- Exemplo:

Tarefa	C	T	D
1	20	100	100
2	40	150	150
3	100	350	350

$$U = \sum_i^3 \frac{C_i}{T_i} \leq 3(2^{1/3} - 1) \quad (7)$$

$$U = \frac{20}{100} + \frac{40}{150} + \frac{100}{350} \leq 3(2^{1/3} - 1) \quad (8)$$

$$U = 0,2 + 0,267 + 0,286 \leq 0,779 \quad (9)$$

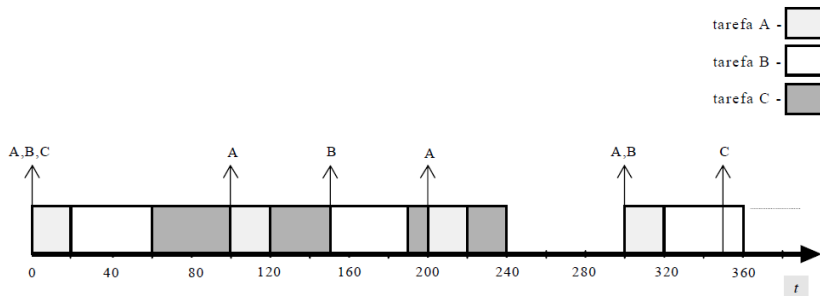
$$U = 0,753 \leq 0,779 \quad (10)$$

- sendo assim, o conjunto testado é escalonável;

Sistemas de Tempo Real - Rate Monotonic (RM)

- Exemplo:

Tarefa	C	T	D
1	20	100	100
2	40	150	150
3	100	350	350



Sistemas de Tempo Real - Deadline Monotonic (DM)

- Proposto em 1982 por Leung e Whitehead;
- Estende o modelo RM;
- A premissa do RM que limita os valores de *deadlines* relativos aos respectivos valores de períodos das tarefas, em muitas aplicações, pode ser considerada muito restritiva;
- Assim como no RM, o DM também assume a existência de tarefas periódicas, independentes entre si, que possuem como tempo de computação (C_i) o seu pior caso de processamento (WCET) e que o tempo de chaveamento entre as tarefas é nulo;
- No entanto, este modelo assume que os *deadlines* relativos podem ser menores ou iguais aos períodos de ativação das tarefas ($D_i \leq T_i$);
- A política do DM define uma atribuição estática de prioridades, baseada nos *deadlines* relativos das tarefas;

Sistemas de Tempo Real - Deadline Monotonic (DM)

- As prioridades são atribuídas na ordem inversa dos valores de seus *deadlines* relativos;
- A produção da escala, portanto, é feita em tempo de execução por um escalonador preemptivo dirigido a prioridades;
- O esquema de prioridades fixas do DM também define um escalonamento estático e *online*;
- O DM é também um algoritmo ótimo na sua classe de problema;

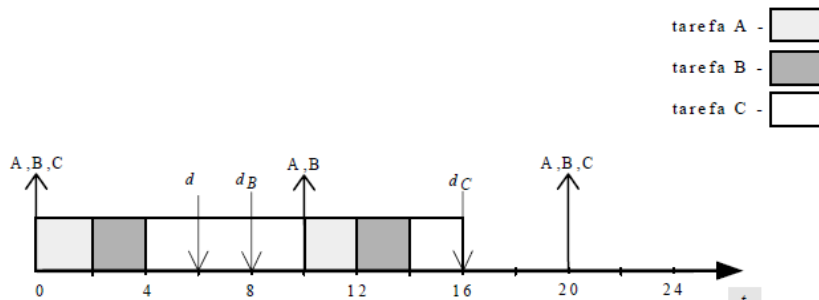
Sistemas de Tempo Real - Deadline Monotonic (DM)

- Na introdução deste artigo os autores não apresentaram um teste de escalonabilidade correspondente;
- Em 1991, Audsley, Burns and Richardson, definiram um teste suficiente e necessário ao DM;
- Diferentemente do RM, onde o teste de escalonabilidade é baseado na utilização, o teste proposto baseia-se no **tempo de resposta**;
- Este teste será tratado posteriormente.

Sistemas de Tempo Real - Deadline Monotonic (DM)

- Exemplo:

Tarefa	C	T	D
1	2	10	6
2	2	10	8
3	8	20	16



Sistemas de Tempo Real - Earliest Deadline First (EDF)

- O EDF define um escalonamento baseado em prioridades: a escala é produzida em tempo de execução por um escalonador preemptivo dirigido a prioridades;
- É um esquema de prioridades dinâmicas com um escalonamento *online* e dinâmico;
- O EDF é um algoritmo ótimo na sua classe de problema;
- As premissas que determinam o modelo de tarefas no EDF são idênticas às do RM:
 - As tarefas são periódicas e independentes;
 - O *deadline* de cada tarefa coincide com seu período de ativação ($D_i = T_i$);
 - O tempo de computação (C_i) de cada tarefa é conhecido e constante (WCET);
 - O tempo de chaveamento entre as tarefas é considerado nulo;

Sistemas de Tempo Real - Earliest Deadline First (EDF)

- A política de escalonamento no EDF corresponde a uma atribuição dinâmica de prioridades que define a ordenação das tarefas segundo os seus *deadlines* absolutos (d_i);
- A tarefa mais prioritária é a que tem o *deadline* absoluto mais próximo do tempo atual;
- A cada chegada de tarefa a fila de pronto é reordenada, considerando a nova distribuição de prioridades;
- A cada ativação de uma tarefa, seguindo o modelo de tarefas periódicas, um novo valor de *deadline* absoluto é determinado considerando o número de períodos que antecedem a atual ativação ($k : d_{ik} = kT_i$);
- No EDF, a escalonabilidade é também verificada em tempo de projeto, tomando como base a utilização do processador;

Sistemas de Tempo Real - Earliest Deadline First (EDF)

- Um conjunto de tarefas é dito escalonável se, e somente se, satisfaça a seguinte premissa:


$$U = \sum_i^n \frac{C_i}{T_i} \leq 1 \quad (11)$$


- Esse teste é suficiente e necessário na classe de problema definida para o EDF pelas premissas a , b , c e d ;
- Se qualquer uma dessas premissas é relaxada (ex.: assumindo $D_i \neq T_i$) o teste continua a ser necessário porém não é mais suficiente;

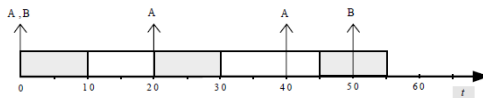
Sistemas de Tempo Real - Earliest Deadline First (EDF)

- Exemplo:

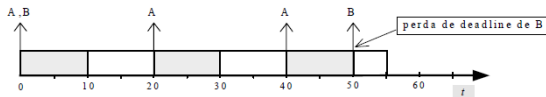
Tarefa	C	T	D
1	10	20	20
2	25	50	50

tarefa A - 

tarefa B - 



(a) Escalonamento EDF



(b) Escalonamento RM