|  |  |
| --- | --- |
| D:\imagens\design\logos\uern.png | Governo do Estado do Rio Grande do Norte  Secretaria de Estado da Educação, da Cultural e dos Desportos – SECD  UNIVERSIDADE DO ESTADO DO RIO GRANDE DO NORTE – UERN  Pró-Reitoria de Ensino e Graduação – PROEG  Ciências da Computação  **Sistemas de Tempo Real – Tópicos de aula** |

**1. Introdução aos sistemas de tempo real**

1.1 Aspectos gerais

* Proliferação do uso dos sistemas computacionais na sociedade⭢ aparecimento de diversas aplicações com requisitos de tempo real
* Aplicações variam quanto a complexidade
  + Sistemas simples
    - Controladores inteligentes embutidos em aparelhos de utilidades domésticas (lavadoras, micro-ondas, aparelhos de DVD...)
  + Sistemas complexos
    - Sistemas militares de defesa
    - Sistemas de controle de plantas industriais
    - Controle de tráfego aéreo/ferroviário
* Algumas aplicações apresentam restrições de tempo mais rigorosas
  + Aplicações rigorosas
    - Monitoramento de pacientes em hospitais
    - Sistemas de supervisão e controle
    - Sistemas embarcados (robôs, aviões...)
  + Aplicações menos rigorosas
    - Videogame
    - Videoconferência
    - Aplicações multimídia
* Metodologias e ferramentas convencionais nem sempre garantem que os requisitos temporais sejam facilmente implementados
  + Os sistemas operacionaisou núcleos de tempo real que gerenciam interrupções e tarefas permitem então a programação de temporizadores que, na maioria dos casos, são ferramentas suficientes para a construção de STR
* As práticas de implementação atuais tem permitido atender de forma aceitável os requisitos da maioria dos STR
  + Técnicas e ferramentas convencionais apresentam limitações
* As exigências cada vez maiores sobre aspectos de segurança colocam em xeque as metodologias e ferramentas convencionais, sob pena de perdas sobre o aspecto financeiro, ambiental ou humano
* Nas aplicações mais críticas, são assumidas situações de operações extremas e pessimistas
  + Várias hipóteses são supostas
  + Consideram-se os piores cenários possíveis
  + Independente desses aspectos supostos, os STR devem atender todas as restrições temporais

1.2 Caracterização de um sistema de tempo real

* Dentre as diversas categorias de sistema, os sitemas de tempo real, de maneira geral, se encaixam na categoria de *Sistemas Reativos*
  + Sistemas reativos são aqueles em que reagem aos estímulos de entrada do ambiente, enviando respostas de maneira contínua
* **Definição:**Um *Sistema de Tempo Real* (STR) é um sistema computacional que deve reagir a estímulos oriundos do seu ambiente em prazos específicos
* O atendimento a esses prazos resulta em requisitos de natureza temporal sobre o comportamento desses sistemas
* Em cada reação o STR deve entregar um resultado correto dentro de um prazo específico, sob pena de ocorrer uma falha temporal
* O comportamento correto de um STR, portanto, não depende só da integridade dos dados obtidos (correção lógica ou *correctness*), mas também dos valores de tempo em que são produzidos (correção temporal ou *timeliness*)
* Uma reação que ocorra fora do prazo especificado pode se tornar sem utilidade ou até mesmo representar uma ameaça
* Um STR deve ser capaz de oferecer garantias de *correção temporal* para o fornecimento de todos os serviços que apresentem restrições temporais
  + Exemplo: Manuseio/Apresentação de vídeos em sistemas multimídia com áudio atrasado/adiantado com relação à imagem

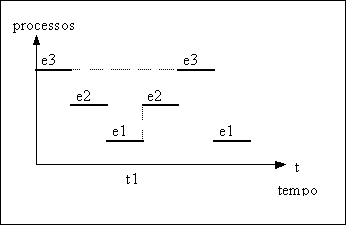
1.3 Classificação dos sistemas de tempo real

* Os STR podem ser classificados quanto ao aspecto de proteção (*safety*):
  + STR brandos (*Soft Real-Time Systems*): quando as consequências de uma falha devida ao tempo é da mesma ordem de grandeza dos benefícios do sistema
    - Exemplos: sistema de comutação telefônico, sistema de processamento bancário
    - **Também conhecidos como Sistemas Não-Críticos de Tempo Real (SNCTR)**
  + STR rígidos (*Hard Real-Time Systems*):quando as consequências de pelo menos uma falha temporal excedam em muito os benefícios normais do sistema
    - Exemplos: sistema de controle de tráfego aéreo/ferroviário, sistema de controle de planta nuclear
    - Nos casos em que há falhas em sistemas rígidos, observa-se que na maioria dos casos as falhas são catastróficas
    - **Também conhecidos como Sistemas Críticos de Tempo Real (SCTR)**
* Os SCTR podem ainda ser subdivididos em
  + SCTR seguros
    - Nesse tipo de sistema, um ou vários estados seguros podem ser alcançados na ocorrência de uma falha
    - Exemplo: parada obrigatória de trens no caso de falhas no sistema de sinalização ferroviária
  + SCTR operacionais
    - Na presença de falhas, somente parte do sistema continuará funcionando
    - Exemplo: queima do sensor do nível de combustível de um avião em pleno voo

**2. Processos**

2.1 Modelos de processos

* O modelo de processos conceitual vem sendo desenvolvido ao longo dos anos por projetistas de Sistemas Operacionais (SOs)
* Nesse modelo, todos os softwares que podem executar em um computador (inclusive, algumas vezes, o próprio sistema operacional) são organizados em **processos**
* Um processo é apenas um programa em execução acompanhado dos valores atuais do contador de programa, dos registradores e das variáveis
* A alternância entre esses processos faz com que o usuário tenha a ilusão de que os processos estão sendo executados de maneira paralela
* Dá-se a essa ilusão o nome de pseudo-paralelismo
* Dá-se o nome de **multiprogramação**ao mecanismo de trocas rápidas que realiza a alternância entre os processos
* Funcionamento do pseudo-paralelismo



* A alternância da CPU entre os processos faz com que a taxa na qual um processo realiza a sua computação não seja uniforme
  + Por esse motivo os processos não devem ser programados com hipóteses predefinidas sobre a temporização
* Quando um processo tem restrições críticas de tempo real, ou seja, quando eventos específicos devem ocorrer dentro de um intervalo de tempo prefixado (normalmente muito pequeno), deve-se tomar certos cuidados para que esses eventos ocorram
* Uma última observação a ser feita é que não se deve confundir processo com programa
  + A diferença entre os dois termos é sutil, mas crucial
  + Exemplo: Padeiro assando um bolo
    - A receita do bolo contém todos os passos necessários para que o bolo seja feito (ingredientes e modo de preparo). Obviamente, nessa analogia, a receita seria o programa
    - O processador em questão é o padeiro, que irá pegar os ingredientes (dados de entrada) e processá-los de acordo com a receita
    - O processo em si é a atividade realizada pelo padeiro de ler a receita, buscar os ingredientes e assar o bolo
    - Imagine então que a filha do padeiro entra na sala dizendo que foi picada por uma abelha. O padeiro vai anotar em que ponto da receita ele parou (salvar o estado atual do processo), buscar um livro de primeiros socorros (outro programa) e começar a seguir as instruções nele contidas
    - Observa-se nesse último caso que o processador alternou de um processo com menor prioridade (assar o bolo) para um processo com maior prioridade (fornecer cuidados médicos)
    - Quando o processo de maior prioridade encerrar, a CPU alterna novamente para o processo de assar o bolo, continuando do ponto em que parou
  + A ideia principal é que um processo constitui uma atividade. Ele possui um programa, entrada, saída e um estado. Um único processador pode ser compartilhado entre vários processos através de algum algoritmo de escalonamento

2.2 Criação de processos

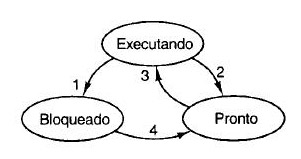
* Os SOs precisam assegurar de algum modo que todos os processos necessários à um propósito existam
* Em sistemas simples, tais como os fornos de micro-ondas, é possível criar todos os processos no momento da inicialização do sistema
* Já em sistemas complexos ou de propósito geral os SOs precisam ter mecanismos de criação e término de processos
* Há, basicamente, quatro possibilidades para criação de novos processos
  + Início do sistema
  + Execução de uma chamada de sistema de criação de processo a partir de um processo em execução
  + Requisição do usuário para criação de um novo processo
  + Início de um trabalho (*job*) em lote
* Tecnicamente, em todos esses casos, um novo processo é criado por um processo existente a partir da execução de uma chamada de sistema de criação de processos
* Nos sistemas Unix, há somente uma chamada para criar um novo processo, denominada fork
  + Depois da chamada os dois processos (pai e filho) possuem a mesma imagem de memória, as mesmas cadeias de caracteres no ambiente e os mesmos arquivos abertos
  + Normalmente o processo filho executa logo em seguida uma chamada da função execve (ou similar) para mudar a imagem de memória e executar um novo programa
* No Windows, uma única chamada de função (CreateProcess) trata tanto do processo de criação quanto da carga do programa correto no novo processo
  + Essa chamada possui 10 parâmetros, incluindo o programa a ser executado, os parâmetros da linha de comando, vários atributos de (in)segurança, bits de controle de arquivos abertos, informações de prioridade, especificações de janela, dentre outros
* Tanto no Unix quanto no Windows, após a criação de um processo, o pai e o filho possuem seus próprios espaços de endereçamento
  + Se um dos dois alterar uma palavra em seu espaço de endereçamento, essa mudança não será visível ao outro processo
  + No Unix, o espaço de endereçamento inicial do filho é uma cópia do espaço de endereçamento do pai, mas há dois espaços de endereçamento distintos
    - Nenhuma memória para escrita é compartilhada
    - Algumas implementações compartilham o código do programa entre os dois (já que não podem ser alterados)
  + É possível que os processos criados filhos compartilhem com seus pais alguns recursos, tais como os arquivos abertos
  + No Windows os espaços de endereçamento são diferentes desde o início

2.3 Término de processos

* Após terminar seu trabalho, um processo será encerrado em virtude de alguma das seguintes condições
  + Saída normal (voluntária)
  + Saída por erro (voluntária) – Normalmente há exibição de avisos
  + Erro fatal (involuntário)
  + Cancelamento por um outro processo (involuntário)
* Na maioria das vezes os processos terminam porque fizeram seu trabalho
* A terceira razão para o término de um processo é um erro causado pelo processo, muitas vezes por um erro de programação
  + Entre os vários exemplos estão a execução de uma instrução ilegal, referência a regiões de memória inexistentes ou a divisão por zero
  + Em alguns sistemas o programador pode fazer com que tais erros sejam tratados no próprio processo
* A quarta razão ocorre, por exemplo, quando a função kill é chamada por um processo nos sistemas Unix. No caso do Windows, a função chamada seria a TerminateProcess
  + Em ambos os casos, o processo que fizer a chamada deve ter a autorização necessária para fazê-la

2.5 Estados de processos

* Embora cada processo seja uma entidade independente, com seu próprio contador de programa e estado interno, muitas vezes os processos precisam interagir com outros
* Quando essa interação é impossibilitada por algum motivo, o processo em questão bloqueia e espera até que as informações possam ser lidas
  + Exemplo: $cat capitulo1 capitulo2 capitulo3 | grep tree
* Um processo bloqueia, obviamente, porque não pode prosseguir
  + Porque está esperando alguma entrada que ainda não está disponível
  + Porque o SO decidiu alocar a CPU para um outro processo, mesmo estando o processo bloqueado conceitualmente pronto e capaz de executar
  + Observe que essas duas situações são completamente diferentes
* Estados possíveis de um processo



* As transições possuem o seguinte significado
  1. Processo em execução bloqueia aguardando uma entrada
  2. O escalonador seleciona um outro processo para execução
  3. O escalonador seleciona esse processo para execução
  4. A entrada torna-se disponível
* Exemplo: Player de vídeo + Leitura de DVD

**3. Threads**

3.1 Modelo clássico

* Em SOs tradicionais cada processo tem seu espaço de endereçamento e um único fluxo de controle (*thread*)
* Frequentemente necessita-se de múltiplos *threads* de controle no mesmo espaço de endereçamento, executando em quase-paralelo, como se fossem processos separados (exceto por possuírem espaços de endereçamento compartilhados)
* O modelo de um processo, conforme visto anteriormente, é baseado em dois conceitos: agrupamento de recursos e execução
* Um processo apresenta um espaço de endereçamento que contém o código e os dados do programa, bem como outros recursos
  1. Os recursos podem ser arquivos abertos, processos filhos, alarmes pendentes, tratadores de sinais entre outros
  2. Já com relação a execução, pode-se dizer que um processo possui uma *thread* de execução, normalmente abreviado para *thread*
* O *thread* contém
  1. Um contador de programa, que mantém o controle de qual instrução será executada, registra
  2. Registradores, com suas variáveis de trabalho
  3. Uma pilha, com a “história” da execução (procedimentos chamados)
* Apesar de um *thread* ter que executar em algum processo, os conceitos (*thread* e seu processo) são e podem ser tratados separadamente
  1. Processos são usados para agrupar recursos
  2. *Threads* são entidades escalonadas para a execução sobre a CPU
* A grande vantagem em se utilizar *threads* é que pode-se assim permitir que múltiplas execuções ocorram no mesmo ambiente do processo, com grande grau de independência de uma para com relação as outras
* Ter múltiplos *threads* executando em paralelo em um processo é análogo a ter múltiplos processos executando em paralelo em um computador
  1. No primeiro caso, os *threads* compartilham um mesmo espaço de endereçamento, os arquivos abertos e outros recursos
  2. No último, os processos compartilham um espaço físico de memória, discos, impressoras e recursos semelhantes
* Como os *threads* tem algumas propriedades dos processos, eles são algumas vezes chamados de processos leves (*lightweight process*)
* O termo ***multithread*** é também usado para descrever a situação em que se permite a existência de múltiplos *threads* no mesmo processo

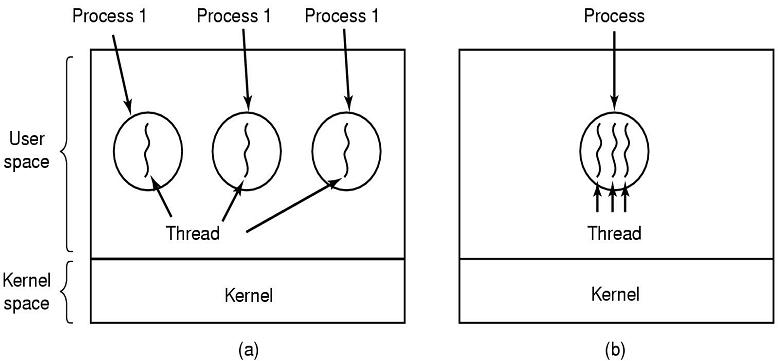


Figura 1 (a) Três processos, cada um com um *thread*. (b) Um processo com três *threads*.

* Observa-se na Figura 3(a) que são apresentados três processos tradicionais, cada um com sua *thread* de execução
* No caso da Figura 3(b), vê-se um único processo com três *threads*
* Em ambos os casos existem três *threads*
* Na Figura 3(a), cada um dos *threads* operam em um espaço de endereçamento diferente, enquanto que na Figura 3(b) os *threads* compartilham o mesmo espaço de endereçamento
* Assim como em processos tradicionais (isto é, um processo com apenas um *thread*), um *thread* pode estar em um dos vários estados: em execução, bloqueado, pronto ou finalizado. As transições entre esses estados são as mesmas exibidas anteriormente
* É importante destacar ainda que cada *thread* possui sua própria pilha de execução

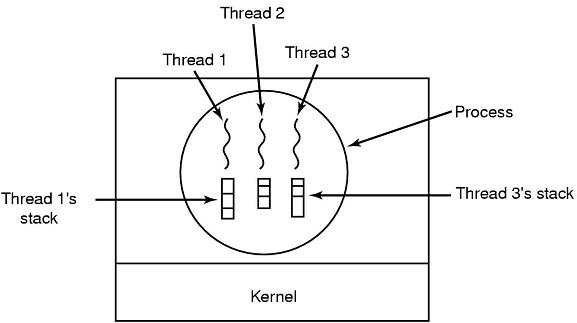


Figura 2 Cada *thread* com sua própria pilha.

* Quando ocorre a execução de múltiplos *threads*, os processos normalmente iniciam com um único *thread*
  + É possível se criar novos *threads* a partir do *thread* inicial chamando um procedimento tal como o thread\_create
    - Em geral uma chamada a esse procedimento tem como parâmetro um ponteiro para a função que deverá ser executada pelo *thread*
    - Não é necessário (nem possível) especificar qualquer detalhe sobre o espaço de endereçamento
* O método de encerramento é semelhante, chamando um procedimento tal como o thread\_exit
* Outra chamada comum é a thread\_yield, que permite que um *thread* desista voluntariamente da CPU para deixar outro *thread* executar
* Outras chamadas permitem que um *thread* aguarde o encerramento de outro ou que informe a finalização de alguma tarefa
* Um último aspecto importante a ser destacado é que os *threads* são mais fáceis de criar e destruir do que processos. Além disso, em alguns sistemas, criar um *thread* é cerca de cem vezes mais rápidos do que criar um processo

3.2 O uso do *Thread*

* A maneira mais interessante de se observar a utilidade dos *threads* talvez seja vislumbrando exemplos concretos de suas utilizações
* Como exemplo, considere um processador de textos
  + A maioria dos processadores de texto mostra em sua interface o texto formatado de maneira semelhante ao que será impresso
    - Todas as posições das quebras de linha e de páginas estão na posição “correta” para que o usuário possa conferí-las
  + Imagine que um autor está escrevendo um livro com 600 páginas e todas elas estão em um único arquivo (para facilitar edições)
  + Se o usuário fizer uma alteração na primeira página, removendo, por exemplo, alguma sentença. Depois de verificar se o posicionamento das sentenças está correto, o usuário decide então alterar uma outra sentença na página 450, digitando um comando para que o processador de texto vá até aquela página
  + Dessa forma o processador é obrigado a processar todas as páginas, da primeira até a 450ª, para reposicionar todas as linhas necessárias, pois até o momento não há como saber qual será a primeira linha da página referida
  + Haverá então uma demora substancial até que essa página seja exibida para o usuário. Nesse caso, se o processador fosse programado com dois *threads*, poder-se-ia fazer com que o primeiro interagisse com o usuário enquanto que o segundo fizesse a reformatação
  + Um terceiro *thread* poderia ainda ser adicionado para fazer o *backup* automático do arquivo de texto de tempos em tempos
  + Se o programa tivesse apenas um *thread* e se o *backup* fosse iniciado, os comandos de teclado e mouse seriam ignorados até que o *backup* fosse encerrado. Dessa forma, o usuário perceberia uma nítida queda de desempenho do sistema
  + Deve estar claro que três processos separados não funcionariam no exemplo dado, pois todos os três *threads* precisam operar sobre o mesmo documento. Ao invés de três processos, são três *threads* que compartilham uma memória comum e, desse modo, têm todo o acesso ao documento que está sendo editado
* Um outro exemplo seria o de um servidor web
  + Um servidor web funciona de tal forma que as requisições por páginas chegam a ele, e a página solicitada é enviada de volta ao cliente
  + Um modo de organizar um servidor web pode ser visto na Figura 3

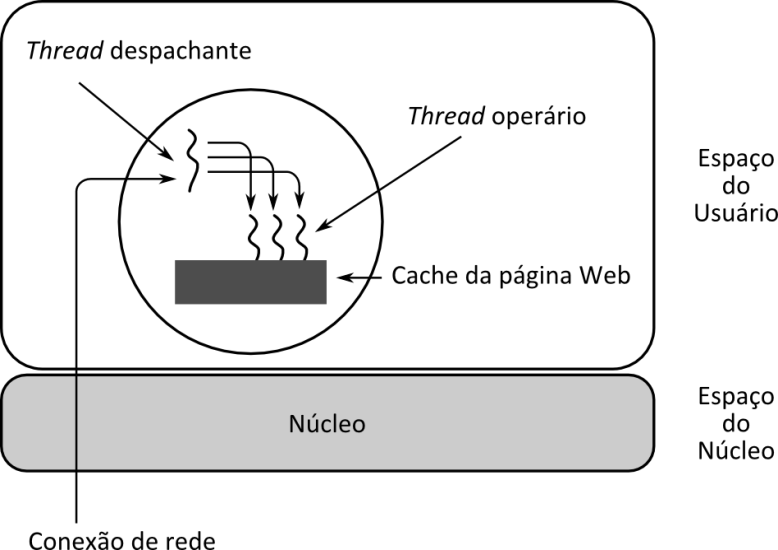


Figura 3 Um servidor web *multithread*.

* + Observa-se na figura que existe um *thread* despachante e vários *threads* operários
  + O *thread* despachante é encarregado de receber as requisições de trabalho que chegam pela rede e escolher um dos *threads* operários ociosos (bloqueados) para fazer a busca da página
  + Dessa maneira o *thread* despachante irá acordar o operário, enviando-lhe uma mensagem, tirando-o do estado de bloqueado e colocando-o no estado pronto
  + Quando desperta o operário verifica se a requisição pode ser satisfeita pela cache. Se isso não ocorrer, o operário fará a leitura da página solicitada no disco e responderá de maneira adequada
* Além dos aspectos aqui abordados, poder-se-ia falar ainda sobre *Threads* de usuário, *Threads* de núcleo, implementações híbridas de *threads* e conversão de código *monothread* para *multithread*. Tais assuntos não foram abordados por não serem necessários até esse momento

**4. Comunicação entre processos**

4.1 Condições de corrida

4.2 Regiões críticas

4.3 Exclusão mútua e espera ociosa

4.4 Dormir e acordar

4.5 Semáforos

4.6 Mutex

4.7 Monitores

4.8 Troca de mensagens

4.9 Barreiras

**5. Escalonamento**

5.1 Introdução ao escalonamento

5.2 Escalonamento em sistemas em lote

5.2 Escalonamento em sistemas interativos

5.2 Escalonamento em sistemas de tempo real

5.3 Escalonamento de threads

6. Problemas clássicos

8.Modelagem de sistemas de tempo real por Redes de Petri

9. Desenvolvimento de sistemas de tempo real em Qt