|  |  |
| --- | --- |
| 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 operacionais ou 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

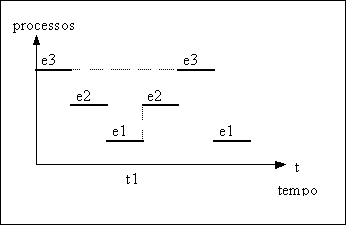
1.3 Classificação dos sistemas de tempo real

* 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

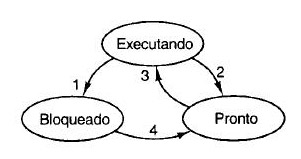
* Os SOs precisam assegurar de algum modo que todos os processos necessários à um propósito existam
* Em sistemas simples, tais como os fornos de micro-ondas, é possível criar todos os processos no momento da inicialização do sistema
* Já em sistemas complexos ou de propósito geral os SOs precisam ter 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 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
  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

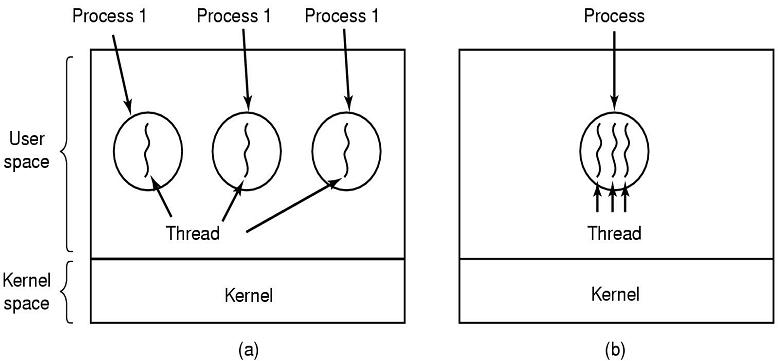


Figura 1 (a) Três processos, cada um com um *thread*. (b) Um processo com três *threads*.

* Observa-se na Figura 1(a) que são apresentados três processos tradicionais, cada um com sua *thread* de execução
* No caso da Figura 1(b), vê-se um único processo com três *threads*
* Em ambos os casos existem três *threads*
* Na Figura 1(a), cada um dos *threads* operam em um espaço de endereçamento diferente, enquanto que na Figura 1(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

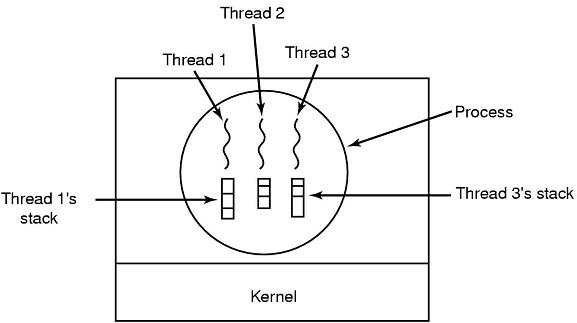


Figura 2 Cada *thread* com sua própria pilha.

* 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 conferi-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

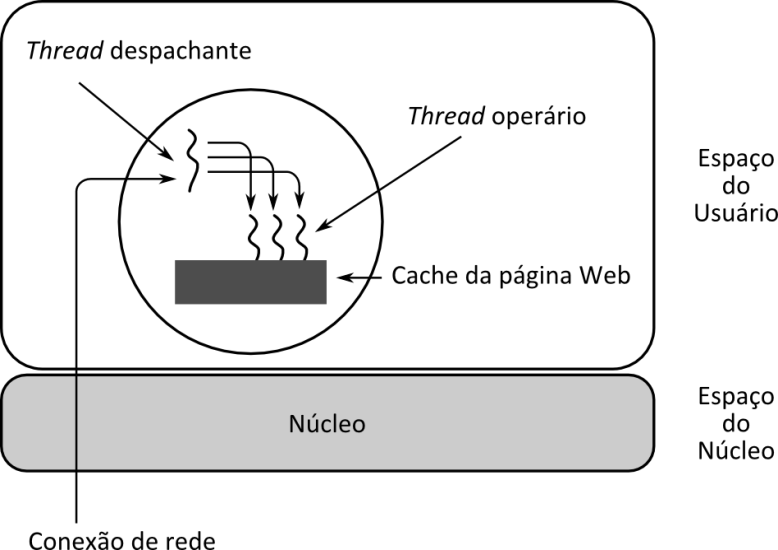


Figura 3 Um servidor web *multithread*.

* + 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 inter-processos**

4.1 Condições de corrida

* Em alguns SOs, processos que trabalham juntos podem compartilhar algum armazenamento comum, a partir do qual cada um poderá ler e escrever
* Independente da natureza desse armazenamento (memória principal ou arquivo no disco), podem surgir diversos problemas
* Considere o caso de um *spool* de impressão
  + Se o *spool* possuir vagas enumeradas de a , e duas variáveis de controle, denominadas *in* e *out*, as quais apontam para a próxima vaga no *spool* e para o próximo arquivo a ser impresso, respectivamente
  + Imagine então que existem 2 processo A e B e que ambos desejam imprimir alguma coisa
  + A então lê a variável *in* e obtém, por exemplo, o valor 7
  + Nesse exato instante o processador decide que A já passou muito tempo na CPU e alterna para o processo B
  + B vai ler a variável *in* e obter o mesmo valor 7. Após isso, B escreve o nome do seu arquivo a ser impresso na posição 7 e sai da jogada...
  + A então é novamente escalonado para operar e, como já havia lido a posição 7, escreve lá o nome do seu arquivo a ser impresso, sobrescrevendo então o nome do arquivo do processo B
  + Dessa forma o diretório de *spool* estará internamente consistente, não percebendo nenhum tipo de erro
  + O arquivo de A será impresso enquanto B aguardará eternamente por seu arquivo
* Situações como essa, nas quais dois ou mais processo estão lendo ou escrevendo algum dado compartilhado e cujo resultado final depende das informações de quem e quando executa precisamente, são denominadas de **condições de disputa** ou **condições de corrida**
* A depuração deste tipo de programas não é nada divertida, pois os resultados da maioria dos testes não apresentam problemas

4.2 Regiões críticas

* Para evitar problemas como o apresentado na seção anterior deve-se ter cuidados para que mais de um processo não possa ler e escrever ao mesmo tempo na memória compartilhada
* Em outras palavras, precisa-se estabelecer critérios de **exclusão mútua** (*mutual exclusion*), assegurando, de algum modo, que outros processos sejam impedidos de usar uma variável ou um arquivo compartilhado que estiver em uso por um processo
* Esse tipo de problema pode ser formulado de uma outra maneira
  + Durante uma parte do tempo, um processo está realizando operações internas e outras coisas que não acarretarão em condições de disputa
  + Algumas vezes um processo precisa ter acesso à memória compartilhada ou a arquivos compartilhados, ou fazer alguma outra operação que gera condições de disputa
  + Aquela parte do programa em que há acesso a memória compartilhada é chamada de **região crítica** (*critical region*) ou **seção crítica** (*critical section*)
  + A ideia é fazer com que dois processos nunca estejam em sua região crítica simultaneamente
  + Embora essa solução impeça as condições de disputa, isso não é suficiente para que processos cooperem de maneira correta. Precisa-se então satisfazer as seguintes condições:
    - Nunca dois processos podem estar simultaneamente em suas regiões críticas
    - Nada pode ser afirmado sobre a velocidade ou sobre o número de CPUs
    - Nenhum processo executando fora de sua região crítica pode bloquear outros processos
    - Nenhum processo deve esperar eternamente para entrar em sua região crítica
  + De maneira geral, o comportamento desejado de um sistema pode ser representado pela Figura 4

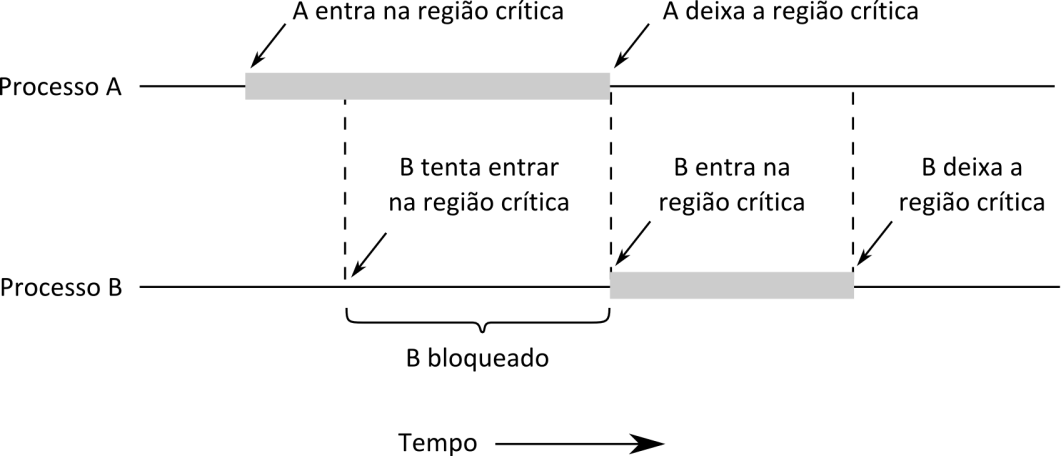


Figura 4 Exclusão mútua usando regiões críticas.

* Existem várias formas de se resolver os problemas gerados pela exclusão mútua, dentre elas, pode-se destacar os métodos com **espera ociosa**, os métodos **dormir e acordar** e os métodos com **semáforos e mutexes**

4.3 Semáforos

* Em 1965, E. W. Dijkstra sugeriu utilizar uma variável inteira para contornar os problemas dos sinais “dormir” e “acordar” (bloqueio e liberação de processos)
* De acordo com a proposta dele, foi introduzindo um novo tipo de variável chamado **semáforo**, que poderia assumir o valor 0 (zero) ou valores positivos de acordo com o número de sinais “acordar” que fossem emitidos pelos processos
* Dijkstra propôs a utilização de duas operações, denominadas *down* e *up* (generalizações de *sleep* e *wakeup*)
  + A operação *down* verifica se o valor do semáforo é maior que zero. Se for, o decrescerá de um (gasta um sinal de acordar armazenado) e prosseguirá. Se o valor for zero, o processo será posto para dormir, sem terminar o *down*, pelo menos por enquanto
    - Verificar o valor, alterá-lo e possivelmente ir dormir são tarefas executadas todas como uma única **ação atômica** e indivisível
    - Garanta-se que, uma vez iniciada a operação do semáforo, nenhum outro processo pode ter acesso ao semáforo até que a operação tenha terminado ou sido bloqueada
    - Essa atomicidade é absolutamente essencial para resolver os problemas de sincronização e evitar condições de disputa
  + A operação de *up* incrementa o valor de um dado semáforo
    - Se um ou mais processos estivessem dormindo naquele semáforo, incapacitados de terminar uma operação de *down*, um deles seria escolhido pelo sistema (por exemplo, aleatoriamente) e seria dada a permissão para terminar seu *down*
    - Portanto, depois de um *up* em um semáforo com processos dormindo nele, o semáforo permanecerá 0 (zero), mas haverá u processo a menos dormindo nele
    - A operação de incrementar o semáforo e acordar um processo é também indivisível
    - Nunca um processo é bloqueado a partir de um *up*
  + A garantia da atomicidade das operações é atribuída ao sistema operacional
    - Deve-se utilizar os semáforos fornecidos pela API do sistema

4.4 Mutex

* Os mutexes nada mais são que semáforos em que não se utiliza sua característica de contagem
* São normalmente utilizados para resolver problemas de exclusão mútua de recursos ou de memória compartilhada, daí seu nome **mutex** (abreviação de *mutual exclusion*)
* Um mutex só pode estar em um dos dois estados disponíveis: impedido ou desimpedido e, consequentemente, pode ser bem representado apenas com um bit
  + Na prática se utiliza um valor inteiro com 0 (zero) para desimpedido e qualquer outro valor para impedido
* Quando um *thread* ou processo precisa ter acesso a uma região crítica, ele chama uma função do tipo mutex\_lock.
  + Se o mutex estiver desimpedido (indicando que a região crítica está disponível), a chamada prosseguirá e o *thread* que chamou a função ficará livre para entrar na região crítica
  + Por outro lado, se o mutex estiver impedido, o *thread* que chamou a função ficará bloqueado até que o *thread* que está na região crítica chame o mutex\_unlock
  + Se múltiplos *threads* estiverem bloqueados, um deles será escolhido para adquirir o impedimento do mutex

4.5 Problema do jantar dos filósofos

* Em 1965, Dijkstra resolveu um problema de sincronização que ele chamou de **problema do jantar dos filósofos**. Desde então, cada um que inventasse uma primitiva de sincronização via-se obrigado a demonstrar até que ponto ela funcionava, mostrando como tal problema era resolvido
* O problema é descrito de maneira simples. Suponha que cinco filósofos estão sentados à uma mesa, cada um com um prato de espaguete
* O espague é tão escorregadio que para comê-lo um filósofo deve utilizar dois garfos
* Entre cada par de prato está um garfo tal como mostra a Figura 5
* A vida de um filósofo é **comer** e **pensar**. Ou seja, um processo filósofo nada mais faz que passar um tempo comendo e um tempo pensando
* Quando um filósofo fica com fome ele tenta pegar os garfos à sua direita e à sua esquerda (em qualquer ordem)
* Se conseguir pegar os dois garfos ele comerá durante um tempo e então colocará os garfos na mesa novamente e continuará a pensar
* A questão fundamental é se você consegue escrever um programa em que todos os filósofos comam e nenhum fique “travado”
* A solução aparentemente é óbvia (Código 1), mas, na verdade, a solução óbvia está errada
  + Suponha que todos os filósofos tentem pegar os garfos à sua esquerda simultaneamente e logo em seguida tentem pegar à direita
  + Nenhum deles conseguirá pegar o garfo à direita e, assim, ocorrerá um ***deadlock***

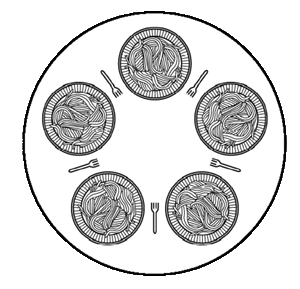


Figura 5 Hora do almoço do departamento de filosofia.

|  |
| --- |
| #define N 5  void filosofo(int i)  {  while(true)  {  pensar();  pegar\_garfo(i);  pegar\_garfo((i+1)%N);  comer();  liberar\_garfo(i);  liberar\_garfo((i+1)%N);  }  } |

Código 1 Uma solução errada para o problema do jantar dos filósofos.

* Podería-se fazer modificações para que o programa, depois de pegar o garfo esquerdo, verificasse se o garfo direito estaria disponível
* Se não estivesse, o filósofo devolveria o garfo que pegou, esperaria algum tempo e repetiria o processo
* Essa proposta, apesar de “menos mal” ainda não funciona
  + Suponha que todos os filósofos peguem seu garfo esquerdo simultaneamente
  + Ao tentar pegar o garfo direito não conseguirão, devolvendo o garfo esquerdo à mesa e esperando um tempo específico
  + Se o tempo de espera for o mesmo para cada um dos filósofos, esse procedimento de pegar o garfo esquerdo, devolvê-lo à mesa e pegar esperar um tempo será repetido eternamente
  + Uma situação como essa, em que todos os programas continuam executando indefinidamente, mas falham ao progredir, é chamada de **inanição** (*starvation*)
* Você deve estar pensando em adicionar um tempo de espera aleatório para resolver o problema, assim como é feita a retransmissão dos pacotes no algoritmo CSMA-CD
* De fato a probabilidade de tudo continuar intertravado seria muito pequena, mas em aplicações críticas (lembre-se das usinas nucleares, dos aviões ...) essa probabilidade tem que ser zero
* A solução ideal utiliza um arranjo de semáforos (um para cada filósofo)

4.6 Outros problemas de IPC

* Além do problema do jantar dos filósofos, outros problemas de comunicação inter-processos aparecem constantemente na literatura
* Dentre os principais problemas abordados, costuma-se mostrar o problema dos **leitores e escritores** e o problema do **barbeiro dorminhoco**
* As soluções encontradas para esses problemas ajudaram a solucionar diversos problemas práticos da vida real

**5. Deadlocks**

* Os sistemas computacionais tem inúmeros recursos adequados ao uso de somente um processo a cada vez (impressoras, unidades de fita, tabelas internas do SO ...)
  + Por exemplo, se dois processos quiserem escrever simultaneamente na mesma impressora haverá um impasse
* Como consequência disso, todos os SOs devem ser capazes de garantir, temporariamente, o acesso exclusivo de um processo a certos recursos
  + Algumas vezes isso pode ser feito, por exemplo, desabilitando as interrupções ao processador
* Para muitas aplicações, um processo necessita de acesso exclusivo não somente a um recurso, mas também a vários
  + Por exemplo, suponha que dois processos irão tentar gravar um cd a partir de uma imagem obtida por um *scanner*
  + O processo A requisita primeiro o *scanner*, enquanto que o processo B requisita primeiro o gravador de CD-ROM
  + Uma nova solicitação de A para o gravador de CD-ROM será negada até que o processo B o libere. Situação semelhante acontece com o *scanner* com relação ao processo B
  + Nesse ponto, ambos os processo ficam bloqueados e assim permanecerão para sempre
  + Tal situação é chamada de **deadlock**
* Um **deadlock** pode ser formalmente definido assim:
  + *Um conjunto de processos estará em situação de deadlock se todo processo pertencente ao conjunto estiver esperando por um evento que somente um outro processo desse mesmo conjunto poderá fazer acontecer*
  + Como todos os processos estão esperando, nenhum deles desencadeará qualquer um dos eventos que poderia acordar algum(ns) outro(s) membro(s) do conjunto e, assim, todos os processos continuam a esperar para sempre

5.1 Aquisição de recursos

* Para alguns tipos de recursos – como registros em um sistema de BD –, cabe aos processos de usuário gerenciar a utilização dos recursos
* Uma maneira de permitir ao usuário o gerenciamento de recursos é associar um semáforo (ou mutex) a cada recurso
* Em alguns casos, como o caso do *scanner* e do gravador de CD-ROM citado acima, os processos podem precisar de mais de um recurso
  + Nesses casos é essencial perceber que a ordem de aquisição dos recursos é realmente importante
  + O ideal é fazer com que os processos adquiram os recursos na mesma ordem, evitando assim os deadlocks

5.2 Condições para ocorrência de deadlocks

* Existem quatro condições para que ocorra um deadlock
  + Condição de exclusão mútua
    - Em um determinado instante, cada recurso está em uma de duas situações: ou associado a um único processo ou disponível
  + Condição de posse e espera
    - Processos que, em um determinado instante, retêm recursos concedidos anteriormente podem requisitar novos recursos
  + Condição de não preempção
    - Recursos concedidos previamente a um processo não podem ser forçosamente tomados desse processo – eles devem ser explicitamente liberados pelo processo que os retém
  + Condição de espera circular
    - Deve existir um encadeamento circular de dois ou mais processos; cada um deles encontra-se à espera de um recurso que está sendo usado pelo membro seguinte dessa cadeia
* Todas as quatro condições devem estar presentes para que um deadlock ocorra. Se faltar uma delas, não ocorrerá deadlock
* É importante notar que cada condição está relacionada a uma política que o sistema pode ou não adotar
  + Um certo recurso pode ser associado a mais de um processo por vez?
  + Um processo pode ter um recurso e requisitar outro?
  + Os recursos podem sofrer preempção (serem retirados de um processo proprietário sem gerar prejuízos)?
  + É possível que exista espera circular?
* Ao negar alguma dessas condições pode-se fazer com que os deadlocks deixem de existir

5.3 O problema do produtor consumidor

* Também conhecido como problema do **buffer limitado**
* Ocorre quando dois processos compartilham um buffer comum e de tamanho fixo
* Um deles, o produtor, põe informações no buffer, enquanto que o outro, o consumidor, retira essas informações
  + Pode-se generalizar o problema para *n* produtores e *m* consumidores
* O problema se origina quando o produtor quer colocar um item no buffer e ele já está cheio
  + A solução é por o produtor para dormir e só despertá-lo quando o consumidor retirar algum item do buffer
  + Da mesma maneira, se o consumidor quiser remover um item do buffer e este estiver vazio, o consumidor deverá dormir até que o produtor insira algum item no buffer
* Imagine o que poderia acontecer quando
  + O consumidor lê a variável que determina o tamanho do buffer e percebe que está vazio
  + Imediatamente antes de “dormir”, o escalonador coloca o produtor para executar
  + O produtor gera um novo item e o coloca no buffer, incrementando o valor para 1 e mandando o consumidor (que deveria estar dormindo, pois o valor estava zero) acordar
* Percebe-se então que o sinal de acordar vai ser “perdido”, e quando o escalonador colocar o consumidor para executar novamente este irá dormir
* O produtor continuará produzindo itens até encher o buffer e dormirá também
* **Exercício:** Resolva este problema utilizando semáforos

**6. Modelagem de sistemas de tempo real por Redes de Petri**

* De acordo com o material do Prof. Adelardo Medeiros (DCA/UFRN), disponível em <ftp://adelardo:web@users.dca.ufrn.br/DCA409/ModuloII.pdf.zip>