Computação Concorrente (MAB-117)
Cap. II: Conceitos básicos para a computação concorrente*

Prof. Silvana Rossetto

[1]Departamento de Ciência da Computação (DCC)
Instituto de Matemática (IM)
Universidade Federal do Rio de Janeiro (UFRJ)
Agosto de 2011

1. Componentes arquiteturais dos sistemas de computação

Um sistema de computação consiste de elementos de *hardware* e *software* que funcionam juntos para executar programas/aplicações do usuário. Para entender o que acontece quando um programa executa na nossa máquina, precisamos entender um pouco como é a organização do hardware nos sistemas de computação modernos. A Figura 1 ilustra os componentes principais de um sistema de computação e suas interligações.

Os **barramentos** são condutores elétricos que carregam bytes de informações entre os componentes. Os barramentos são normalmente projetados para transferir conjuntos de bytes de tamanho fixo, conhecidos como *words*. O número de bytes em uma word é um parâmetro da arquitetura da máquina e tipicamente é de 4 ou 8 bytes.

Os **dispositivos de E/S** são a conexão do sistema com o mundo externo (ex., mouse, teclado, monitor, disco, interface de rede).

A **memória principal** é o dispositivo de armazenamento temporário que armazena o código dos programas e os dados que ele manipula. A memória é organizada logicamente como um *vetor linear de bytes*, cada um com um endereço único (índice do vetor), começando em zero. As instruções de máquina que constituem o programa podem conter um número variado de bytes. O tamanho dos itens de dados variam de acordo com o tipo de dado. Como exemplo, em uma máquina IA32 Linux um dado do tipo inteiro (*int*) ocupa 4 bytes, e um real grande (*double*) ocupa 8 bytes.

O **processador** (ou CPU — *central processing unit*) é onde a máquina interpreta, ou executa, as instruções de máquina armazenadas na memória principal. O processador usa um conjunto de dispositivos de armazenamento de acesso rápido, chamados **registradores**. Entre esses registradores está o *program counter* (PC) (ou ponteiro de programa). Sua função é armazenar o endereço da próxima instrução de máquina na memória que deverá ser executada pelo processador. Do momento em que a máquina é ligada até ela ser desligada, o processador repetidamente executa a instrução apontada pelo PC e o atualiza para apontar para a próxima instrução.

As instruções do código de uma aplicação são executadas em sequência, e a execução de uma instrução envolve a realização de uma série de passos: ler a instrução da memória, interpretar os bits da instrução, executar a operação definida e atualizar o ponteiro de programa para apontar para a próxima instrução (a qual pode ou não estar em uma posição continuada da memória em relação a instrução atual).

*Este texto foi extraído do livro "Computer Systems — a Programmer's Perspective", R. E. Bryant e D. R. O'Hallaron, capítulos 1.4-7, 1.9, 8.2, 12.1, 12.3[1]

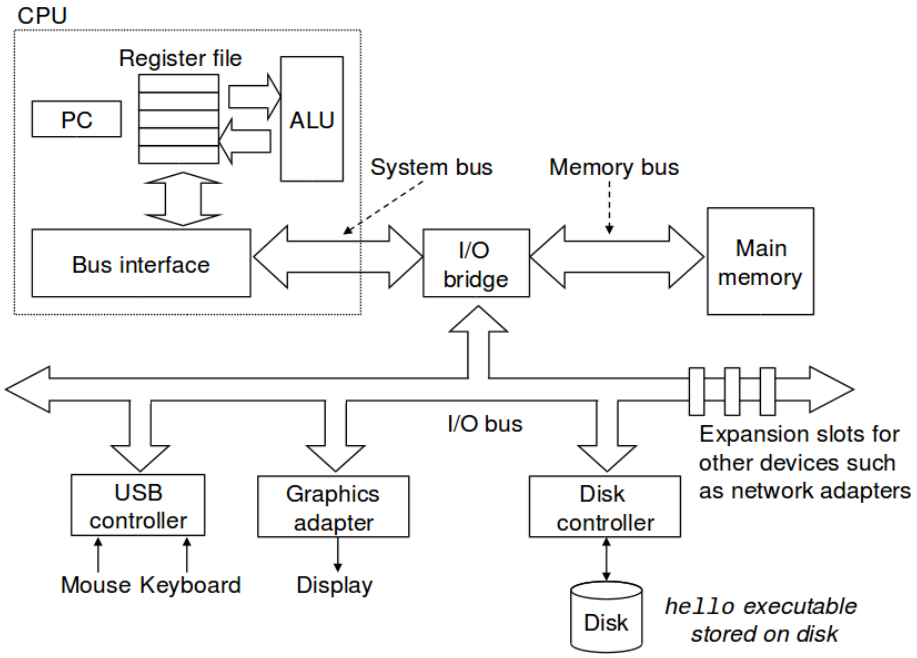

Figura 1. Organização típica de um sistema de computação [1].

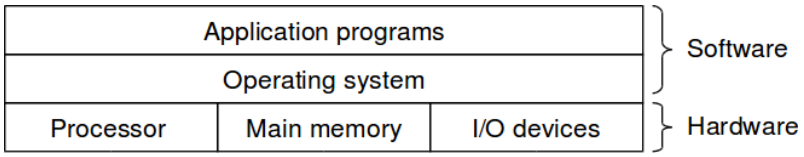

Figura 2. Visão em camada de um sistema de computação [1].

Um princípio arquitetural dirige todas as coisas: **processador e memória estão em lados opostos** — gasta-se um longo tempo para o processador ler/escrever dados na memória (similar à relação de tempo entre o envio de uma carta e um telefonema). Boa parte das informações que precisamos saber sobre arquitetura de computadores está relacionada ao esforço de "aliviar" a distância entre processador e memória [2].

2. O papel do Sistema Operacional

Os programas do usuário não acessam o hardware da máquina diretamente, ao invés disso eles fazem uso dos *serviços* oferecidos pelo Sistema Operacional (SO) que executa na máquina. O SO é uma camada de software interposta entre o hardware e as aplicações, como mostrado na Figura 2, e tem dois propósitos principais:

- proteger o hardware de erros (ou mau uso) das aplicações;
- prover mecanismos para as aplicações que simplifiquem o acesso ao hardware.

Esses propósitos são alcançados através da implementação das seguintes *abstrações*: **processos**, **memória virtual** e **arquivos**. A relação entre essas abstrações é mostrada na Figura 3. O detalhamento sobre como cada uma dessas abstrações é implementada pelo SO está fora do nosso escopo de estudo. Por ora, focaremos nossa atenção apenas no conceito de *processos*.

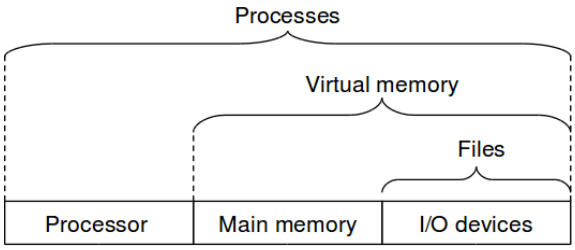

Figura 3. Abstrações providas pelo Sistema Operacional [1].

2.1. Processos

Quando um programa executa, o SO provê a "ilusão" de que o programa está executando sozinho no sistema, dispondo de uso exclusivo do processador, da memória e dos dispositivos de E/S. Essa ilusão é criada pelo conceito de *processo* — uma abstração de SO para um programa em execução. Assim, vários processos podem executar concorrentemente no mesmo sistema de computação e cada processo parece ter uso exclusivo do hardware.

Executar "concorrentemente" significa que se observarmos o comportamento do sistema ao longo do tempo, veremos que a execução das instruções de um processo alternam com a execução das instruções de outros processos, como ilustrado na Figura 4. Na maioria dos sistemas de computação há sempre mais processos em execução do que processadores disponíveis para executá-las. Um único processador consegue "aparentar" que está executando vários processos ao mesmo tempo, executando um pedaço de cada um até terminarem.

Para que um programa execute corretamente, ele precisa manter **informações sobre o estado da sua execução** ao longo do tempo, até terminar. Essas informações são mantidas dentro do **contexto do processo** associado à execução do programa, e incluem: código do programa, dados armazenados na memória, pilha de execução, conteúdo dos registradores, variáveis de ambiente e descritores de arquivos abertos.

Trocas de contexto O SO é responsável por gerenciar a alternância entre processos por meio de um mecanismo chamado **troca de contexto**. Considere o caso de uma máquina com um único processador (uniprocessador). O SO mantém as informações de estado (ou contexto) que os processos precisam para executar: valores armazenados nos registradores (ponteiro de programa, ponteiro da pilha, etc.), conteúdo da pilha, descritores de arquivos abertos, etc. Em qualquer instante de tempo, um sistema uniprocessador pode executar o código de apenas um processo.

Quando o SO decide transferir o controle do processo atual para algum novo processo, ele executa uma troca de contexto: salva o contexto do processo atual, restaura o contexto do novo processo e passa o controle para o novo processo. O novo processo retoma a sua execução exatamente do ponto onde ele parou anteriormente.

Fluxo de controle lógico Quando pedimos ao sistema para executar um programa, um novo processo é criado pelo SO e o arquivo executável do programa é executado no contexto desse processo. Duas abstrações importantes são providas para as aplicações através do conceito de processos:

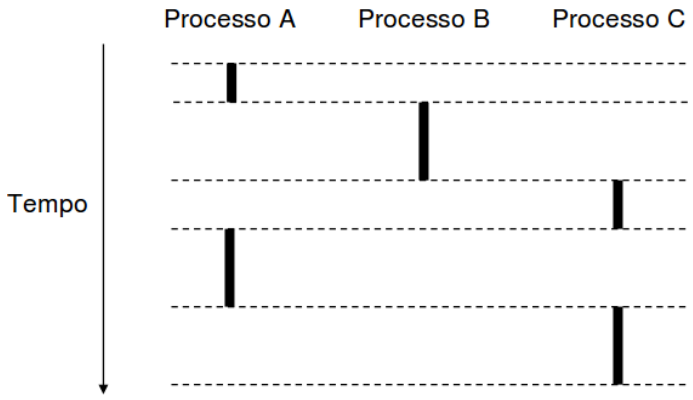

Figura 4. Alternância da execução das instruções dos processos ao londo do tempo [1].

- um **fluxo de controle lógico** independente que provê a ilusão de que o programa tem uso exclusivo do processador;
- um **espaço de endereçamento privado** que provê a ilusão de que o programa tem uso exclusivo da memória do sistema.

Um processo provê para o programa a ilusão de que ele tem acesso exclusivo ao processador, mesmo quando vários outros programas estão executando concorrentemente no sistema. Se usássemos um depurador para acompanhar a execução passo-a-passo de um programa, poderíamos observar uma série de valores para o PC (ponteiro de programa, registrador que aponta para o endereço da próxima instrução a ser executada) que correspondem exclusivamente aos endereços das instruções contidas no arquivo executável do programa (ou nos arquivos objeto das bibliotecas compartilhadas usadas pelo programa). Essa sequência de valores do PC é denominada **fluxo de controle lógico**, ou simplesmente *fluxo lógico*.

A Figura 4 ilustra essa idéia. O **fluxo de controle físico** do processador é dividido em três fluxos lógicos, um para cada processo em execução. A figura mostra que os processos se revesam no uso do processador. Cada processo executa uma porção do seu fluxo e então é "preemptado" (temporareamente suspenso) enquanto outro processo executa.

Fluxos concorrentes Fluxos lógicos podem aparecer de diferentes formas em um sistema de computação. Tratadores de exceção, processos, tratadores de sinais, threads são todos exemplos de fluxos lógicos. Um fluxo lógico cuja execução se sobrepõe/alterna no tempo com outro fluxo é chamado **fluxo concorrente** e os dois fluxos executam concorrentemente. Em outras palavras, dois fluxos lógicos X e Y são concorrentes entre si se e somente se X começa depois de Y começar e antes de Y terminar, ou vice-versa, Y começa depois de X começar e antes de X terminar.

Note que a idéia de fluxos lógicos concorrentes é independente do número de processadores na máquina. Se dois fluxos se sobrepõem no tempo, então eles são concorrentes, mesmo que eles estejam alternando o uso do mesmo processador. Quando dois fluxos executam concorrentemente em diferentes processadores, então eles são **fluxos paralelos** (executam de fato ao mesmo tempo).

3. Concorrência no nível das aplicações

A concorrência dentro dos programas/aplicações do usuário final é interessante por vários aspectos:

1. **Acesso concorrente aos dispositivos de E/S**. Quando a aplicação está esperando por dados de dispositivos E/S de baixa velocidade, o SO mantém a CPU ocupada com outra aplicação. A própria aplicação pode explorar essa possibilidade de concorrência sobrepondo processamento com requisições E/S.
2. **Interação com humanos**: As pessoas ao interagirem com o computador demandam a possibilidade de executarem várias tarefas ao mesmo tempo. Por exemplo, redimensionar uma janela enquanto imprime um documento ou receber um stream de audio. Os sistemas de interfaces modernos usam a concorrência para prover essa capacidade. Sempre que o usuário requisita alguma ação (ex., clique do mouse), um fluxo lógico concorrente é criado para executar a ação.
3. **Redução da latência por antecipação de tarefas**: As aplicações podem usar concorrência para reduzir a latência de certas tarefas, antecipando a execução de algumas operações. Por exemplo, uma aplicação que requer a execução de uma operação em um dispositivo periférico ou requer a entrada de dados do usuário, pode antecipar a solicitação da operação, de modo que quando os dados forem de fato requeridos para o processamento poderão ser acessados de forma mais rápida.
4. **Atendimento a vários clientes simultâneos**: Aplicações servidoras, que esperam por requisições de centenas o milhares de clientes por segundo, não podem tratar um cliente de cada vez pois postergará o acesso de todos os demais clientes. A abordagem mais comum é construir um servidor concorrente que cria um fluxo lógico de execução para cada cliente.
5. **Computação em paralelo em máquinas multiprocessador**: A maioria dos sistemas de computação modernos são equipados com processadores multi-núcleo. Aplicações que são divididas em fluxos concorrentes têm mais oportunidades de usufluir desse hardware pois os fluxos podem executar em paralelo.

Os programas que usam concorrência explícita no nível de aplicação são chamados **programas concorrentes**. Há diferentes caminhos para construir programas concorrentes:

- **Aplicações com vários processos**: Cada fluxo lógico dentro da aplicação é criado e mantido como um novo processo. Como o espaço de endereçamento de cada processo é separado, se houver a necessidade de interação entre os fluxos de execução é preciso usar algum mecanismo de comunicação inter-processo.
- **Aplicações com várias threads**: Cada fluxo lógico dentro da aplicação é criado e mantido dentro do contexto do mesmo processo, e é escalonado implicitamente pelo SO ou por uma biblioteca de suporte. Como o espaço de endereçamento é compartilhado entre os fluxos, a interação entre eles pode ser feita via acesso a memória compartilhada (menos custosa que os mecanismos típicos de comunicação inter-processos).
- **Aplicações com várias co-rotinas**: Similar ao caso anterior (co-rotinas são fluxos lógicos de execução dentro do contexto de um mesmo processo), mas o escalonamento (alternância entre fluxos) é feito pela própria aplicação (e não pelo SO ou biblioteca). Por isso apenas uma co-rotina pode estar executando a cada instante de tempo (embora todas possam estar ativas, i.e., aptas a executar).

- **Aplicações com multiplexação de E/S**: Forma de concorrência onde as aplicações escalonam, de forma explícita, fluxos lógicos dentro do contexto do mesmo processo. Os **fluxos lógicos são modelados como máquinas de estado** e o programa principal transita de um estado para outro em resposta ao recebimento de dados que chegam de diferentes descritores de arquivos.

A necessidade de implementar aplicações concorrentes tornou-se mais evidente com o desenvolvimento de aplicações Web. Em um programa navegador (ex., Firefox, Internet Explorer, etc.) há tipicamente várias linhas de execução distintas ativas simultaneamente, cada uma delas fazendo comunicação com um servidor remoto. Quando os dados são recebidos, eles devem ser formatados para serem exibidos na tela. O uso de várias linhas de execução garante que operações rápidas (ex., exibição de texto) não precisem esperar por operações mais lentas (ex., exibição de imagens).

O desenvolvimento de aplicações concorrentes usando threads é um dos caminhos mais usuais. É mais fácil compartilhar dados entre threads do que entre processos, e a troca de contexto entre elas é menos custosa do que a troca de contexto entre processos. Continuaremos nosso estudo sobre programação concorrente focando no conceito de threads.

3.1. Programação concorrente com threads

Como dito anteriormente, um processo pode consistir de várias unidades de execução, escalonadas diretamente pelo SO ou por biblioteca de suporte, chamadas **threads**. Cada thread tem um conjunto de informações de **contexto da thread**, incluindo um identificador único, pilha de execução e valores nos registradores de uso geral. Todas as threads que executam dentro do contexto de um mesmo processo compartilham código e dados globais com as outras threads, como ilustrado na Figura 5.

Aplicações com mais de uma thread são normalmente chamadas **aplicações multithreading**. Mesmo em máquinas com um único processador, o modelo multithreading pode ser usado para simplificar a estrutura do programa que é logicamente constituído por diferentes funcionalidades [3]. Alguns exemplos de uso de threads em máquinas monoprocessador são:

- **Execução em background**: em aplicações com interface visual, uma thread pode ser responsável por exibir os menus e capturar os eventos de entrada e outra thread pode ser responsável por executar os comandos e atualizar a interface. Essa organização normalmente melhora a percepção de velocidade da aplicação, permitindo que o programa apresente os próximos comandos enquanto o comando anterior ainda está sendo executado.
- **Processamento assíncrono**: elementos assíncronos do programa podem ser implementados por threads distintas, ex., uma thread é responsável por periodicamente fazer um backup da aplicação enquanto outra thread é responsável pelo programa principal.
- **Sobreposição de processamento e comunicação**: Um processo com várias threads pode computar um lote de dados enquanto lê o próximo lote de um dispositivo.
- **Estrutura modular**: Programas que envolvem uma variedade de atividades ou uma variedade de fontes e destinos de entrada e saída pode ser mais fácil de projetar e implementar usando threads.

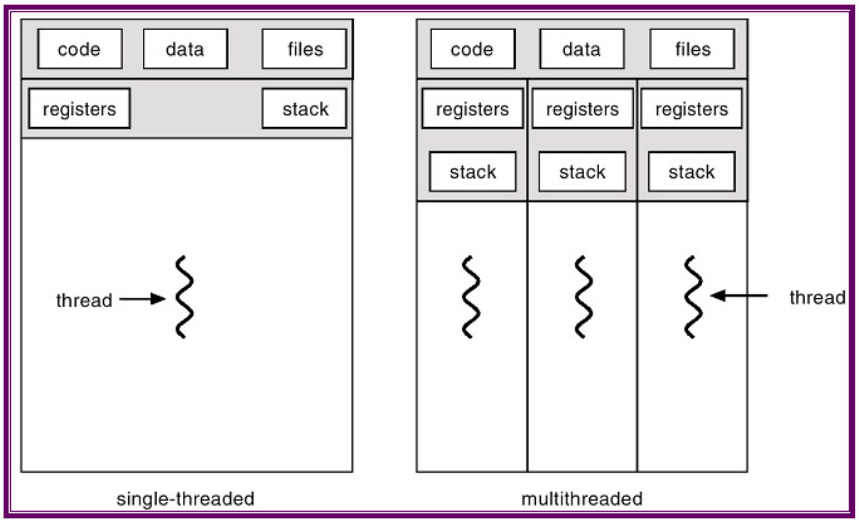

Figura 5. Processos com uma única thread e com várias threads.

Sempre que uma linha de execução (*thread*) bloqueia esperando por E/S (ex., mensagens da rede), a implementação da aplicação permite que uma troca de contexto ocorra automaticamente, escalonando uma outra thread para execução. Em uma **biblioteca de threads preemptiva**, as trocas entre threads ocorrem também em intervalos de tempo pré-definidos, de forma a garantir a interatividade da aplicação e evitar que alguma thread monopolize o uso da CPU (ex., threads com muito processamento e pouca E/S).

Modelo de execução das threads O modelo de execução de várias threads é similar ao modelo de vários processos. Todo processo começa a executar a partir de uma *thread principal* (em C e Java implementada pela função *main*). Em algum ponto do código, a thread principal cria uma nova thread. A partir desse ponto as duas threads podem executar concorrentemente. A execução de threads difere da execução de processos em alguns pontos importantes:

- Como o contexto de uma thread é menor que o contexto de um processo (conjunto de informações que devem ser salvas e restauradas quando há troca de contexto), a troca de contexto entre threads é mais rápida do que a troca de contextos entre processos.
- As threads não são organizadas de forma hierárquica, como no caso de processos (processos pai e processos filhos). Todas as threads associadas a um processo formam um grupo de threads de mesmo nível, independente da ordem de criação ou de quais threads foram criadas por quais outras threads.
- A thread principal se distingue das demais apenas no sentido de que é sempre a primeira thread a executar e, dependendo da implementação da biblioteca de gerência de threads usada, o processo encerra automaticamente quando a thread principal termina.

Bibliotecas de threads Uma **biblioteca de threads** fornece ao usuário/programador uma API (conjunto de funções) para criação e gerenciamento de threads. Ela pode ser

implementada inteiramente no espaço do usuário, ou inteiramente no espaço do kernel (implementação do SO). Três bibliotecas de threads de uso comum são [4]:

- **POSIX PThreads**: pode ser oferecida/implementada como biblioteca no nível do usuário ou no nível do kernel.
- **Win32**: biblioteca de kernel do Windows.
- **Java**: acompanha a linguagem Java, normalmente é implementada pela JVM (por isso pode variar de uma JVM para outra) usando uma biblioteca de threads disponível no sitema hospedeiro. Cada implementação da JVM decide como associar as threads Java com as threads do SO.

4. Ganhos dos sistemas multiprocessadores

Inicialmente, a execução concorrente era apenas *simulada*, com um único processador alternando rapidamente entre os processos/threads em execução. Essa forma de concorrência permite vários usuários interagindo com o sistema ao mesmo tempo (ex., várias pessoas acessando a página de um mesmo servidor Web); ou um único usuário interagindo com várias aplicações concorrentemente (ex., um navegador em uma janela, um editor de texto em outra janela e um stream de audio em execução).

Quando o sistema de computação consiste de vários processadores (arquiteturas *multicore* ou *hyperthreaded*), todos sob o controle de um mesmo SO, temos um **sistema multiprocessador**. Os sistemas multiprocessadores podem melhorar o desempenho das aplicações de duas formas:

- reduzindo a necessidade de simular a concorrência quando executa várias tarefas;
- diminuindo o tempo de execução de uma aplicação (requer que o programa seja dividido em várias threads que podem de fato executar em paralelo).

Referências

[1] R. E. Bryant and D. R. O'Hallaron. *Computer Systems - A Programmer's Perspective*. Prentice-Hall, 2 edition, 2010.

[2] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan-Kaufmann, 2008.

[3] W. Stallings. *Operating Systems – Internals and Design Principles*. Pearson - Prentice Hall, 6 edition, 2009.

[4] A. Silberschatz, P.B. Galvin, and G. Gagne. *Sistemas operacionais com Java*. Elsevier-Campus, 7 edition, 2008.