Programação Concorrente (ICP361)
Cap. 5: outras técnicas de programação concorrente (multiprocesso, I/O multiplexado, gerência cooperativa com co-rotinas, futuros)

Profa. Silvana Rossetto

[1]Instituto de Computação - UFRJ

1. Introdução

Nos capítulos anteriores exploramos a computação concorrente usando *multithreading*. Neste capítulo, vamos estudar outras técnicas ou mecanismos de computação concorrente, incluindo: **multiprocessos**, **I/O multiplexado**, **co-rotinas** e **futuros**.

Vimos anteriormente que fluxos de execução são concorrentes se eles se intercalam no tempo. Há diferentes formas da execução concorrente ser explorada em uma aplicação:

- **Acesso a dispositivos de I/O mais lentos**: sobreposição de computação com requisições de I/O (enquanto espera pelo I/O, faz uso do processador para outra computação).
- **Interação com usuários finais**: fluxos lógicos distintos para tratar cada ação solicitada pelo usuário (programação orientada a eventos).
- **Resposta a vários clientes**: em aplicações cliente/servidor, o servidor dispara um novo fluxo de execução independente para atender cada requisição de cliente.
- **Computação paralela**: permite explorar o paralelismo real do hardware (máquinas multicore) para execução simultânea.

Para implementar programas concorrentes, as seguintes técnicas podem ser implementadas:

- **Multiprocesso**: cada fluxo lógico de execução é um processo que é escalonado e mantido pelo sistema operacional. Os processos podem ser disparados a priori ou serem criados como processos filhos a partir do processo principal. O espaço de endereçamento é exclusivo de cada processo e por isso a comunicação e sincronização entre eles precisa ser feita com mecanismos explícitos de interação interprocessos. O escalonamento dos processos é feito pelo sistema operacional, normalmente de forma *preemptiva*.
- **I/O multiplexado**: em um outro extremo, com I/O multiplexado, a aplicação escalona seus próprios fluxos de execução, dentro do contexto de um único processo, com um espaço de endereçamento único. A aplicação é modelada como uma *máquina de estados* que o programa principal explicitamente controla, baseado nos eventos de I/O recebidos.
- **Multithreading**: fluxos lógicos independentes dentro de um mesmo processo, mas escalonados pelo sistema operacional, normalmente de forma *preemptiva* (podendo existir um "pre-escalonamento" no contexto da linguagem de programação). Compartilham o mesmo espaço de endereçamento.
- **Gerência cooperativa com co-rotinas**: construção de programação similar a uma *thread*. Cada co-rotina representa uma linha de execução distinta, com sua própria pilha, variáveis locais e um ponteiro de instrução. Como as *threads*, as co-rotinas

compartilham variáveis globais com outras co-rotinas. A principal diferença é que *threads* podem executar simultaneamente enquanto co-rotinas são **colaborativas**. Um programa com co-rotinas executa apenas uma co-rotina em cada momento, mesmo em um sistema multi-processador, e a execução de uma co-rotina só é suspensa quando a própria co-rotina faz essa solicitação. Em essência, co-rotinas são fluxos concorrentes onde a troca de controle é completamente especificada [1].

- **Futuros ou computação assíncrona**: primitiva de nível mais alto, oferecida pela linguagem de programação, que permite que uma computação seja executada em paralelo e seu resultado capturado mais adiante no programa. Normalmente é implementada fazendo uso do conceito de *pool de threads ou de processos*.

2. Multiprocesso

No caso mais comum, o processo principal cria processos filhos, atribuindo uma subtarefa para cada um deles. O processo filho herda uma cópia do código e do contexto de execução do processo pai e pode seguir sua execução de forma independente. No processo pai, é possível explicitamente aguardar a finalização do processo filho.

Um cenário de uso de programação concorrente multiprocesso é na implementação de aplicações *cliente/servidor*. O lado *servidor* aguarda requisições dos clientes. Quando uma nova requisição é recebida, um processo filho é criado para atender essa requisição e passa a interagir diretamente com a aplicação *cliente*, enquando o *servidor* volta a esperar por novas requisições. Os espaços de endereçamento do processo pai e do processo filho são distintos. Isso é vantajoso quando se pretende evitar que um processo tenha acesso aos dados do outro processo (indicado quando as subtarefas são independentes). Por outro lado, dificulta a possibilidade de comunicação via memória compartilhada.

3. I/O multiplexado

Continuando no contexto de aplicações *cliente/servidor*, considere agora o caso do servidor precisar responder também a comandos que o administrador solicita via entrada padrão. Nesse caso, o servidor precisará tratar dois eventos de I/O independentes: (i) um cliente fazendo requisição na conexão de rede; e (ii) o administrador digitando um comando no teclado. Por qual evento ele deverá esperar primeiro? O ideal é que seja ambos, pois não é possível saber qual chegará primeiro e qual será a frequência de cada evento. Para implementar esse tipo de tratamento, uma alternativa básica é usar a técnica de **I/O multiplexado**, fazendo uso de uma função especial chamada *select*. Essa função suspende o processo corrente, retornando o controle para ele apenas depois que um ou mais eventos de I/O (esperados) tenham ocorrido. A função *select* é complexa, manipula um *conjunto de descritores*, e pode ser usada em diferentes cenários.

Usando a técnica de I/O multiplexado é possível implementar **aplicações baseadas em eventos**. No caso de uma aplicação cliente/servidor, essa técnica torna-se uma alternativa quando o tratamento das requisições são simples e diretos, não compensando a criação de processos filhos ou threads. Como o processo executa no contexto de um único processo, é possível compartilhar dados entre os fluxos lógicos de execução (tratadores de eventos). A principal desvantagem está na complexidade do código que, de forma geral, precisa implementar uma máquina de estados. Quando as transições de estado se tornam

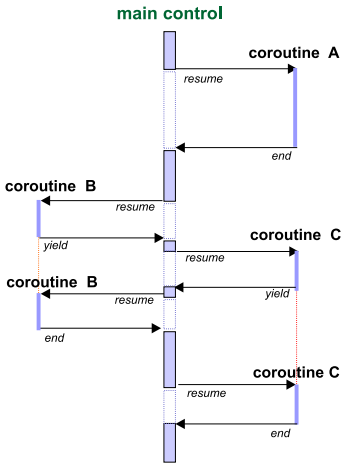

Figure 1. Exemplo de interação entre co-rotinas.

mais complexas e requerem a manutenção de contextos de execução entre as ocorrências dos eventos, a alternativa de I/O multiplexado se torna inviável.

4. Co-rotinas

O conceito de co-rotinas apareceu como uma construção de linguagem de programação nos anos 70 com Simula,mas não é uma construção comum nas linguagens de programação. Como discutido em [2], o motivo para essa ausência foi a falta de uma visão uniforme do conceito, e a complexidade das primeiras implementações. Recentemente, entretanto, o interesse por co-rotinas tem reaparecido nas aplicações multi-tarefa, em linguagens de *scripting*, e também no contexto de adaptação dinâmica de sistemas distribuídos. Na adaptação dinâmica, a gerência cooperativa de tarefas implementada por meio de co-rotinas, garante a atomicidade das operações de adaptação sem a necessidade de mecanismos adicionais de sincronização. A Figura 1 mostra um exemplo de interação entre co-rotinas de uma aplicação.

A gerência *cooperativa* entre linhas de controle distintas é considerada uma alternativa às dificuldades encontradas tanto com a gerência preemptiva, da programação *multithreading*, como com a gerência serial, da programação dirigida a eventos (caso mais geral da programação dirigida a I/O) [3] [4]. Nesse caso, a aplicação pode manter diferentes linhas de execução e a transferência de controle entre elas é explicitamente embutida dentro do próprio algoritmo da aplicação. A abordagem cooperativa é adequada quando o controle do processador precisa ser alternado para evitar a espera ocupada por informações de outros processos, e o paralelismo real não é crucial para um bom desempenho da aplicação. Os pontos de transferência são pré-definidos e a troca de contexto ocorre apenas quando é de fato necessária, minimizando o custo computacional requerido. Ao mesmo tempo, a espera por informações de outros processos não impede a evolução de computações distintas.

Entretanto, uma dificuldade ligada à gerência cooperativa, é que a transferência de controle entre as tarefas deixa de ser implícita, como ocorre na gerência preemptiva, precisando ser explicitamente codificada dentro da aplicação.

5. Futuros ou computação assíncrona

Permite que uma parte da aplicação, cuja execução pode ocorrer de forma paralela, possa ser antecipadamente iniciada e seu resultado recebido mais adiante, quando ele se torna necessário. Não necessariamente essa execução ocorre em paralelo (depende da disponibilidade de recursos), mas é uma forma de indicar que ela pode ser antecipada e executada em paralelo a partir do ponto em que é invocada na aplicação. Esse mecanismo é tipicamente usado em aplicações *IO-bound*, fazendo uso do conceito de *pool de threads* (ou de processos).

Um *pool de threads* é um conjunto de threads previamente criadas cuja lógica central consiste em aguardar a submissão de uma tarefa para execução, executar a tarefa submetida e voltar a esperar por nova tarefa, até que seja finalizada. A ideia central é que uma vez criadas, as threads ficam disponíveis até o final da aplicação, minimizando os custos de criação e finalização dessas threads, e o tempo de escalonamento das tarefas. Outra vantagem é oferecer uma interface de programação concorrente de nível mais alto, que facilita a alocação dinâmica de tarefas entre as threads e reduz a complexidade de lidar com decisões como quantidade ideal de threads e dependências de plataforma [5].

Linguagens como Java e Python implementam classes de *pool de threads* que permitem submeter tarefas, aguardar o término da execução dessas tarefas e cancelar uma tarefa incompleta. No caso de Java, a tarefa é submetida passando um objeto que implementa a interface `Runnable`. A classe que implementa o *pool de threads* retorna outro objeto que implementa a interface `Future` que conceitualmente é a *promessa* de entrega mais adiante do resultado de uma **computação assíncrona**. A interface *Future* provê o método `get()` que retorna o resultado da computação, bloqueando se necessário o fluxo que o invocou (no caso da computação assíncrona ainda não ter sido concluída).

6. Exercício

Liste exemplos de problemas ou aplicações conhecidas onde cada uma das formas de uso da concorrência apresentadas acima se aplicaria. Em cada caso descreva em linhas gerais como a aplicação seria projetada.

References

[1] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, 1983.

[2] A. L. Moura. *Revisitando co-rotinas*. PhD thesis, PUC-Rio, Rio de Janeiro, Brasil, 2004.

[3] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track USENIX Annual Technical Conference*, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.

[4] R. Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, May 2003.

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