Desenvolvimento

Concorrência, Paralelismo, Processos, Threads, programação síncrona e assíncrona

Concorrência, paralelismo, processos, threads, programação síncrona e assíncrona, são assuntos que permeiam o dia a dia dos desenvolvedores. A ideia desse artigo é descomplicar um pouco o que esses conceitos significam e como eles se relacionam.

Kennedy Tedesco • há 2 anos

Concorrência, paralelismo, processos, threads, programação síncrona e assíncrona, são

Monotarefa versus Multitarefa

Os primeiros sistemas operacionais suportavam a execução de apenas uma tarefa por vez. Nesse modelo, o processador, a memória e os periféricos ficavam dedicados a uma única tarefa. Tínhamos um fluxo bem linear, como pode ser visto nesse diagrama:

Apenas no término da execução de uma tarefa que outra poderia ser carregada na memória e então executada.

O problema desse modelo é que enquanto o processo realizava uma operação de I/O para, por exemplo, ler algum dado do disco, o processador ficava ocioso. Ademais, uma operação do processador é infinitamente mais rápida que qualquer uma de leitura ou escrita em periféricos.

Para se ter ideia, quando falamos de uma operação que a CPU executa, lidamos com nanosegundos, enquanto em uma operação de rede consideramos mile segundos.

Se você "pingar" o Google observará isso:

Copiar

```
$ ping google.com.br
PING google.com.br (216.58.202.3): 56 data bytes
```

A solução encontrada para resolver esse problema foi permitir ao processador suspender a execução de uma tarefa que estivesse aguardando dados externos ou algum evento e passar a executar outra tarefa. Em outro momento de tempo, quando os dados estivessem disponíveis, a tarefa suspensa poderia ser retomada do ponto exato de onde ela havia parado. Nesse modelo, mais de um programa é carregado na memória. O mecanismo que permite a retirada de um recurso (o processador, por exemplo) de uma tarefa, é chamado de *preempção*.

Sistemas preemptivos são mais produtivos, ademais, várias tarefas podem estar em execução ao mesmo intervalo de tempo alternando entre si o uso dos recursos da forma mais justa que for possível. Nesse tipo de sistema as tarefas alteram de estado e contexto a todo instante.

Os estados de uma tarefa num sistema preemptivo:

- **Nova:** A tarefa está sendo criada (carregada na memória);
- **Pronta:** A tarefa está em memória aguardando a disponibilidade do processador para ser executada pela primeira vez ou voltar a ser executada (na hipótese de que ela foi substituída por outra tarefa, devido à preempção);
- **Executando:** O processador está executando a tarefa e alterando o seu estado;
- **Suspensa:** A tarefa não pode ser executada no momento por depender de dados externos ainda não disponíveis (dados solicitados à rede ou ao disco, por exemplo.);
- **Terminada:** A execução da tarefa foi finalizada e ela já pode sair da memória;

O diagrama de estado das tarefas com preempção de tempo:

Quantum pode ser entendido como o tempo que o sistema operacional dá para que os processos usem a CPU. Quando o quantum de um processo termina, mesmo que ele ainda não tenha terminado a execução de suas instruções, o contexto dele é trocado, é salvo na sua pilha de execução onde ele parou, os dados necessários e então ele volta pro estado de "pronto", até que o sistema operacional através do seu *escalonador de processos* volte a "emprestar" a CPU pra ele (e então ele volta pro estado de "executando"). Trocas de contexto (de pronto pra executando etc) acontecem a todo momento. Milhares delas. É uma tarefa custosa para o sistema operacional, mas que é necessária para que a CPU não fique ociosa.

Um core (núcleo) do processador executa uma tarefa por vez, cabendo ao escalonador do sistema operacional cuidar dessa fila de tarefas, decidir quem tem prioridade, quem tem o quantum disponível etc.

O que é um processo?

Um processo pode ser visto como um *container* de recursos utilizados por uma ou mais tarefas. Processos são isolados entre si (inclusive, através de mecanismos de proteção a nível de hardware), não compartilham memória, possuem níveis de operação e quais chamadas de sistemas podem executar. Como os recursos são atribuídos aos processos, as tarefas fazem o uso deles a partir do processo. Dessa forma, uma tarefa de um processo *A* não consegue acessar um recurso (a memória, por exemplo) de uma tarefa do processo *B*.

As tarefas de um processo podem trocar informações com facilidade, pois compartilham a mesma área de memória. No entanto, tarefas de processos distintos não conseguem essa comunicação facilmente, pois estão em áreas diferentes de memória. Esse problema é resolvido com **chamadas de sistema** do kernel que permitem a comunicação entre processos (IPC - Inter-Process Communication).

O que é uma thread?

Os processos podem ter uma série de threads associadas e as threads de um processo são conhecidas como threads de usuário, por executarem no modo-usuário e não no modo-kernel. Uma thread é uma "linha" de execução dentro de um processo. Cada thread tem o seu próprio estado de processador e a sua própria pilha, mas compartilha a memória atribuída ao processo com as outras threads "irmãs" (filhas do mesmo processo).

O núcleo (kernel) dos sistemas operacionais também implementa threads, mas essas são chamadas de threads de kernel (ou kernel-threads). Elas controlam atividades internas que o sistema operacional precisa executar/cuidar.

Concorrência e paralelismo

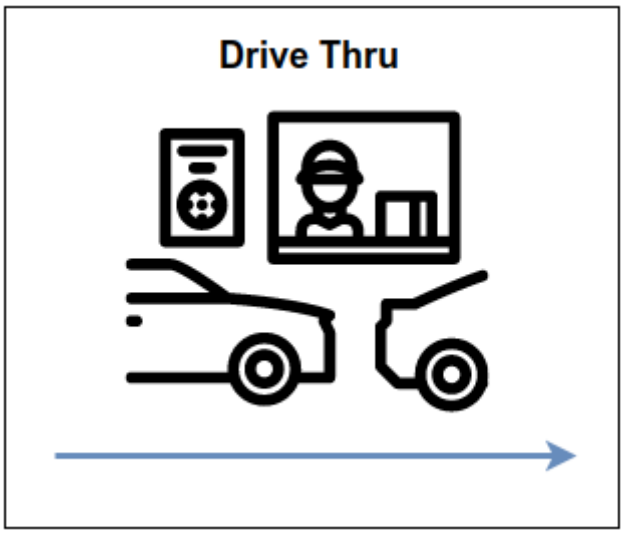
É comum achar que concorrência e paralelismo são a mesma coisa, mas não são. *Rob Pike* - um dos criadores da linguagem Go - em uma apresentação pontuou:

> *"Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once."*

Concorrência é sobre **lidar** com várias coisas ao mesmo tempo e paralelismo é sobre **fazer** várias coisas ao mesmo tempo. Concorrência é um conceito mais a nível de software e paralelismo mais a nível de hardware.

oportunidade de serem executadas.

Podemos fazer uma analogia em que concorrência (no contexto de um sistema operacional) é uma fila de drive thru em que carros estão disputando o recurso do atendimento, um por vez. Mas esse drive thru é especial, diferente do da vida real, ele é baseado num sistema preemptivo onde os carros possuem um tempo máximo em que podem ficar ali parados pelo atendimento, passando esse tempo, o carro tem que obrigatoriamente sair para dar oportunidade pro próximo carro da fila:

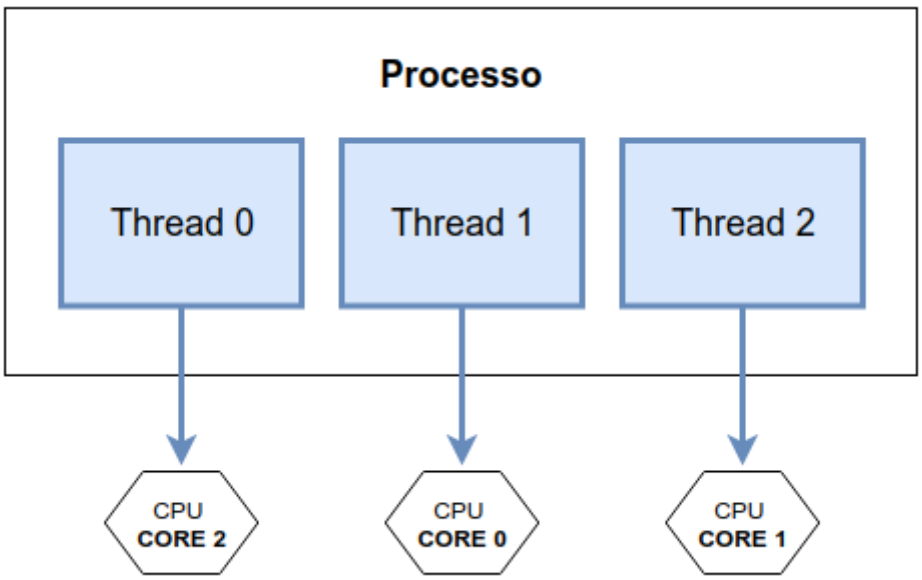


Paralelismo é sobre a execução paralela de tarefas, ou seja, mais de uma por vez (de forma simultânea), a depender da quantidade de núcleos *(cores)* do processador. Quanto mais núcleos, mais tarefas paralelas podem ser executadas. É uma forma de distribuir processamento em mais de um núcleo.

Paralelismo é aquele pedágio que permite que carros progridam em diferentes fluxos simultaneamente:

Também podemos dizer paralelismo é uma forma de atingir concorrência e que cada linha de execução paralela também é concorrente, pois os núcleos estarão sendo disputados por várias outras linhas de execução e quem gerencia o que o núcleo vai executar em dado momento do tempo é o escalonador de processos. Paralelismo implica concorrência, mas o contrário não é verdadeiro, pois é possível ter concorrência sem paralelismo, é só pensar no caso de uso de uma única thread gerenciando milhares de tarefas, pausando e resumindo-as, esse é um modelo de concorrência sem paralelismo. Já o *scheduler* de corrotinas da linguagem Go é um exemplo de scheduler *multi threaded*, ele paraleliza a execução das corrotinas, ou seja, é um modelo de alta concorrência que faz uso dos núcleos do processador para paralelizar as execuções.

Do ponto de vista de um processador que possui mais de um núcleo (que é o padrão atualmente), um processo poderia ter, por exemplo, três threads rodando simultaneamente em três diferentes núcleos:



Copiar

```php
<?php

function tarefa1() {
    echo "tarefa1\n";
}

function tarefa2() {
    echo "tarefa2\n";
}

tarefa2();
tarefa1();
```

O resultado é previsível:

Copiar

```
tarefa2
tarefa1
```

Já no modelo assíncrono, uma operação não precisa esperar a outra ser finalizada, ao contrário disso, elas alternam o controle da execução entre si. É um modelo não previsível e que não garante a ordem da execução. É um modelo que favorece a concorrência.

Fazendo uma analogia, no modelo síncrono se você precisa colocar roupas para lavar e lavar louças, primeiro você poderia colocar as roupas para lavar e esperaria o tempo que fosse necessário até finalizar, para só depois lavar as louças. Enquanto que no modelo assíncrono você poderia colocar as roupas na máquina de lavar roupas e já começaria a lavar as louças no mesmo instante, pois você não precisa ficar ocioso esperando a máquina de lavar processar seu resultado para desempenhar outra tarefa, você pode pegar o “resultado” da máquina de lavar em um momento futuro.

Já no modelo assíncrono nós temos a alternância da execução das tarefas, elas concorrem entre si:

Em verde as tarefas que iniciaram, em cinza a alternância de execução durante o tempo e em vermelho a finalização da execução delas. Fiz questão de não colocar em ordem pois assincronismo não garante ordem, assincronismo é sobre a execução de tarefas independentes.

Observe que tanto no modelo síncrono quanto no assíncrono uma tarefa é executada por vez, a diferença é que no modelo assíncrono a tarefa não precisa esperar o resultado da outra para poder ter seu tempo de execução. Assincronismo não necessariamente implica em paralelismo (se pensar em assincronismo num modelo multi-thread, sim, aí atinge paralelismo), assincronismo é uma forma de atingir concorrência.

Existem diferentes formas de se atingir assincronismo, sendo que a principal é a orientada a eventos (*event loop* com o padrão *Reactor*), modelo usado por NodeJS, Nginx, Swoole, ReactPHP entre outros. No caso do NodeJS e ReactPHP, por exemplo, eles fazem isso de forma single-thread, mas também é possível atingir assincronismo num modelo orientado a threads (multi-thread).

Em termos gerais, usar o modelo assíncrono é vantagem para a maior parte dos aplicativos que fazem uso intensivo de operações de I/O (leitura e escrita de arquivos, acesso a rede etc). Enquanto que o modelo de paralelizar processamento é mais indicado quando as tarefas são mais orientadas à CPU, quando elas precisam de mais

exemplo, no caso do PHP que é síncrono por padrão e que (ainda, na versão 7.4) não possui nenhum mecanismo a nível da linguagem para operações assíncronas, você pode estudar ReactPHP ou Swoole. A linguagem Go é uma boa pedida para entender na prática concorrência e paralelismo. E para estudar I/O assíncrono, NodeJS pode ser uma boa pedida.

Se você se interessa pelos pormenores do assunto relacionado a processos e threads e como o sistema operacional gerencia isso, o livro *Sistemas Operacionais Modernos* do *Tanenbaum* é uma ótima referência de estudo.

Até mais!

#Sistemas Operacionais #síncrono #assíncrono #Threads #Processos

Autor(a) do artigo