22 de julho de 2021

Universidade Federal do Rio de Janeiro

Sistemas Distribuidos

Lista 1

Bruno Dantas de Paiva
Rodrigo Mendes Palmeira

1 Escolhas de Implementação

1.1 Escolha da Linguagem de Programação

Decidimos utilizar a linguagem Go para o projeto por termos um interesse prévio de aprender a linguagem somado ao seu bom balanço de capacidade de realizar ações de baixo nível, necessárias para o projeto, e certas facilidades modernas como garbage collection. Ademais, Go foi feito pensando em sua aplicação em sistemas distribuídos com um tipo de primitiva especial chamada de *Go routines*(Rotinas GO) que possibilitam o uso de CSP(Communicating Sequential Processes)[1] que achamos que pode ser bastante interessante de utilizar para os trabalhos da disciplina.

1.2 Escolha do Sistema Operacional

Decidimos implementar nossos programas utilizando as *SysCalls* do Linux por ser o melhor sistema operacional para desenvolvimentos.

2 Implementações

2.1 Signals

Para Signals foi necessário desenvolver 2 programas, onde um seria responsável por enviar um sinal à qualquer outro processo e o outro seria responsável por capturar e tratar sinais diferentes, imprimindo uma mensagem específica para cada sinal.

Além disso, é importante introduzir um pouco melhor sobre o assunto ao falar sobre esses programas. Cada processo no UNIX possui um PID (Process Identification) que é um id único que define o processo de forma única enquanto este estiver em execução.

Cada sinal no UNIX também possui seu número identificador, no qual para esta implementação foram escolhidos 3 sinais, SIGHUP (1), SIGTERM (15), SIGQUIT (3), no qual o último foi definido como responsável por finalizar o programa.

[1] https://www.cs.cmu.edu/ crary/819-f09/Hoare78.pdf

2.1.1 Receiver

Sabendo que o Receiver deve responder a sinais específicos enviados pelo próprio sistema operacional ou por um outro programa, foi necessário inicialmente obter o PID do Receiver (o golang possui uma função própria do sistema operacional para isso) e receber um input do usuário para escolher em 2 tipos de espera, blocking wait e busy wait.

Em Go, para tratar o recebimento de signals ou comunicação entre processos por signal é necessário implementar uma goroutine, que é uma forma de go lidar com programação concorrente e a utilização de canais para permitir que haja uma sincronização dessas goroutines.

Cada um dos canais tratados acima, possui um tipo definido, deste modo foi necessário criar um canal do tipo signal (que serve somente para recebimento de sinais) e um canal auxiliar para representar a saída do processo.

Para a implementação dos modos de espera, o busy wait consiste num loop infinito aguardando por um sinal recebido, contudo, por estarmos utilizando go routines isto se torna um pouco mais traiçoeiro.

Para essa implementação, foi necessário utilizar o statement select (similar ao switch-case de outras linguagens) com um caso default contendo nada. A motivação para isto é que o método <- do golang significa que ele irá aguardar de forma bloqueante até que a variável receba o valor daquele canal, em nosso caso signalChan. Para contornar isso, foi necessário criar um caso default que não faz absolutamente nada, porém permite que o loop permaneça rodando infinitamente e que só entre no case quando tiver efetivamente algum sinal executando.

Já o Blocking wait possui um comportamento similar, com relação ao loop infinito, contudo não é mais necessário a utilização do select com um caso default. Neste caso, foi utilizado somente um switch-case, onde na definição anterior do switch é utilizado o método <- para receber efetivamente o sinal enviado para o signalChan.

Deste modo, como explicado no Busy Wait, o próprio processo ficará em estado waiting até ele receber o sinal e voltar ao estado running.

2.1.2 Sender

Este programa possui um processo bem simples, sendo necessário somente ler indefinidamente pelo os.Stdin (input do usuário) e caso tenha um input (assumindo que contenha um PID e inteiro representando o signal que deseja

enviar) este fará uma syscall.Kill para o pid designado com o signal definido.

Primeiramente é necessário definir se o processo existe ou não, para tal optamos por dar um Kill no pid com o signal 0 (sinal para checar acesso ao pid), caso não obtivessemos erro, o processo seria existente.

Para enviar o sinal recebido pelo input no stdin foi utilizado o mesmo processo Kill para enviar o sinal para o outro processo.

2.2 Pipes

Para essa implementação, foi desenvolvido um único programa que possui funções de cliente(produtor) e servidor(consumidor), para tal foi necessário utilizar a função fork e pipe para criar a comunicação entre os processos. Golang possui algumas funções nativas para executar tanto o fork quanto o pipe, contudo pela forma de que golang é implementada, algumas das funções nativas acabam abstraindo a utilização dessas funções, dificultanto todo o processo. Portanto, optamos por utilizar diretamente as syscalls para chamar tanto o Fork quanto o Pipe.

Para a leitura e a escrita dos dados, o filho foi definido como o produtor e o pai foi definido como o consumidor.

O input deste programa é feito dentro da função produtor, que faz o scan do dado inputado, gera números aleatórios com base no número anterior e escreve em um lado do pipe (passado como parâmetro da função).

Ao passo disso, conforme o Sistema Operacional escalona os processos, o consumidor irá ler os dados deste pipe em uma outra função, converter para inteiro e verificar se esse dado recebido é um primo ou não, imprimindo uma mensagem.

2.3 Sockets

Para o problema do Sockets, foram criados dois programas um trabalhando como servidor(consumidor) e outro como cliente(produtor).

2.3.1 Cliente

O programa cliente pede ao usuário o quantos números aleatórios ele deverá gerar e mandar para o servidor. Assim, ele inicia uma conexão TCP com o progama de servidor, manda os números, os gerando de forma randômica como especificado no trabalho utilizando uma biblioteca utilitária, e imprime

a resposta para cada um deles correspondente à se o número enviados é primo. No final, ele envia 0 para o servidor e encerra a execução. O cliente transmite os números utilizando um segmento TCP para cada e enviando o número seguido de um separador '
n'.

2.3.2 Servidor

O programa servidor escuta conexões TCPs na porta 8081, aceita a conexão que surgir(Ou seja lida apenas com um cliente por vez) e para cada mensagem recebida "parsea"(Sabendo que o separador para cada número é o '
n') o número e responde com uma mensagem indicando se esse é primo ou não, para decidir isso chama uma função da biblioteca utilitária. Se o número recebido for 0, o programa encerra a execução.

2.4 Funções utilitárias

Para auxiliar as partes comuns do nosso programa criamos uma pequena biblioteca de funções utilitárias. Nela, fizemos uma função que implementa a geração do próximo número como especificado na descrição do trabalho utilizando como seed para a parte randômica a hora do computador. Fizemos, também, uma função que verifica se um número é primo ao testar se ele é divisível por todos os números entre 2 e sua raiz quadrada.

2.5 Link da implementação

O código está disponível no github pelo link `https://github.com/DantasB/Distributed-Systems` acompanhado de um Readme com um breve explicação de como rodar os programas.