Lista 2 - Sistemas Distribuídos - COS470

Aluno: Marcos Seefelder de Assis Araujo

Professor: Daniel Ratton Figueiredo

Questão 1

Os dois principais modelos de *Inter Process Communication* (**IPC**) são os de **memória compartilhada** e **troca de mensagens**.

- **Troca de mensagens**: Nesse modelo, os processos trocam mensagens através de chamadas de sistema `send()` e `receive()` que põe e retiram mensagens de uma fila armazenada em *kernel space*. O enlace pode ser direto ou indireto e a troca de mensagens pode ser síncrona ou assíncrona:
  - *Direto/Indireto*:
    - No caso do enlace **direto**, tanto a **origem** quanto o **destino devem ser identificados** nas chamadas de `send` e `receive`. Nesse caso os enlaces são **unidirecionais** e **criados automaticamente** (realizado com o uso de **sinais** por exemplo);
    - No caso **indireto**, as mensagens sao enviadas para e lidas a partir de uma **"caixa postal"**. Nesse caso, as caixas postais devem ser **criadas explicitamente** e são **bidirecionais** (realizado com **sockets** ou **pipes**, por exemplo);
  - *Síncrono/Assíncrono*:
    - No caso **síncrono**, as **chamadas são bloqueantes**, ou seja, o receptor espera a mensagem chegar e o emissor espera a mensagem ser recebida. Durante a espera, o processo fica em *Waiting*;
    - No caso **assíncrono**, as **chamadas são não-bloqueantes**, ou seja, após as chamadas os processos continuam executando normalmente. Nessa situação as mensagens são armazenadas em um *buffer* em *kernel space* de tamanho limitado, que pode ser controlado pelo processo.
- **Memória compartilhada**: Nesse modelo uma região de memória é mapeada para ser compartilhada entre dois processos. Diferente do caso de troca de mensagens, o sistema operacional não está envolvido nesse processo, e a região de memória compartilhada está **fora do *kernel space***. Isso acarreta numa desvantagem, que é a grande chance da ocorrência de *condições de corrida* no acesso dessa região, sendo necessária a **coordenação do acesso através de sincronização**.

As principais vantagens das trocas de mensagens é que o sistema operacional oferece as chamadas que facilitam sua realização e administra aspectos como os *buffers* da troca assíncrona, por exemplo. Uma desvantagem é que tudo é baseado em chamadas de sistema, que tomam bastante tempo de execução.

Questão 2

Em se tratando de pipes, uma escrita (`write()`) bloqueante bloqueia quando não há espaço no *buffer* do *pipe*, enquanto uma não-bloqueante simplesmente retorna um valor de erro e a execução continua. No caso da leitura (`read()`), uma leitura bloqueante bloqueia até que haja algo para ler, equanto a não-bloqueante retorna um valor de erro no caso de não haver o que ler e a execução continua.

No caso do código, temos as seguintes possíveis combinações:

- *Write* **bloqueante** & *Read* **bloqueante**: *P1* escreve continuamente no *pipe* e *P2* lê sempre que há algo para ser lido. Como a função `process()` pode demorar um pouco para retornar é improvável que não haja nada para ser lido por *P2*, mas pode ocorrer do *buffer* ficar cheio (por *P2* não dar conta da

velocidade com que *P1* o preenche). Nesse caso, *P1* vai bloquear até que haja espaço para a escrita no *pipe*. A função `process()` sempre receberá algum parâmetro que *P1* enviou;
- *Write* **bloqueante** & *Read* **não-bloqueante**: Temos uma situação parecida com a anterior, pois como `process()` pode demorar para retornar, é improvável que tente ler o *buffer* e não haja nada para ser lido. Caso isso acontecesse, o valor **m** lido em `process()` provavelmente seria preenchido com zeros, não garantindo que *P2* recebesse sempre valores enviados por *P1*;
- *Write* **não-bloqueante** & *Read* **não-bloqueante**: Nesse caso pode ocorrer de *P1* tentar escrever com o *buffer* cheio e não bloquear. Numa situação dessas, a escrita não será feita e a execução de *P1* continuará normalmente. *P2* iria provavelmente receber apenas valores enviados por *P1*, mas não necessariamente receberia todos os valores que P1 tentou enviar. Com pouquíssima probabilidade (devido à demora de `process()`) poderia ocorrer de P2 ler e o *buffer* estar vazio, acarretando numa situação descrita no tópico anterior;
- *Write* **não-bloqueante** & *Read* **bloqueante**: Nesse caso pode ocorrer de *P1* tentar escrever com o *buffer* cheio e não bloquear. Numa situação dessas, a escrita não será feita e a execução de *P1* continuará normalmente. *P2* irá receber sempre valores enviados por *P1*, mas não necessariamente receberia todos os valores que P1 tentou enviar.

Questão 3

Cada processo é associado a um **PCB** (Process Control Block), grande estrutura de dados mantida para armazenar todo o estado do processo, o que é bem custoso para o Sistema Operacional. Além disso a troca de contexto entre processos é uma operação extremamente pesada para o Sistema Operacional. Um sistema *multi-process* envolve todos esses problemas, pois pra cada processo temos um **PCB** e são diversas trocas de contexto.

Por outro lado, num sistema *multi-threaded*, múltiplas threads compartilham um único **PCB**, por sua vez tendo um custo menor para o Sistema Operacional em questões de consumo de memória. Dependendo da implementação, a troca de contexto também é consideravelmente mais leve (em *user-level threads*). Além disso, um mesmo processo pode aproveitar mais *cores* do processador e a administração de atividades concorrentes é facilitada.

Sistemas de com *threads* podem usá-las em dois modelos: *User-Level Threads* e *Kernel-Level Threads*:

- **User-Level Threads** rodam no espaço do usuário, fazendo com que as operações de criar e destruir threads sejam meramente operações de alocação e liberação de memória além da troca de contexto de threads poder ser realizada de forma barata, com poucas operações (pois a quantidade de dados armazenados específios de cada *thread* e pequena). Uma desvantagem é que no caso de uma *thread* bloquear em alguma instrução, o processo todo bloqueia, pois o Sistema Operacional não tem conhecimento da existência de diversas threads dentro do processo e o enxerga como um todo para questões de escalonamento;
- **Kernel-Level Threads** (ou *Lightweght Processes*) são implementações de *threads* visíveis como unidades individuais para o Sistema Operacional, pois são implemetadas a nível de *kernel*, fazendo com que uma *thread* bloqueada não bloqueie seu processo inteiro. Por outro lado, todas as operações de administração das *threads* passam a ser chamadas de sistema, tornando o custo da mudança de contexto entre threads alto;
- Para obter o melhor das duas opções, pode-se utilizar um modelo híbrido de *threads* que mistura *kernal-level threads* com *user-level threads*.

Uma **possível desvantagem** de sistemas *multi-threaded* é que, como as implementações de *threads* aramazenam o mínimo possível de informação que permite a divisão da CPU entre diferentes *threads*, proteger o acesso inapropriado de dados pelas diversas *threads* é uma tarefa deixada para o programador.

Questão 4

Um sistema *multi-threaded* baseado em *user-level threads* é um processo que só roda em um núcleo de processamento de cada vez, pois as ***threads* em nivel de usuário não são enxergadas pelo Sistema Operacional como unidades que podem ser escalonadas independentemente**. Portanto, é de se esperar que o desempenho de tal sistema sem um computador multi-processado seja igual ao de um mono-processado no sentido de que o mesmo **não aproveita a disponibilidade de múltiplos *cores* para a execução concomitante de *threads***.

Questão 5

Uma condição de corrida ocorre quando a consistência de um resultado depende que uma determinada sequência de eventos ocorra e, devido ao paralelismo, podem ocorrer sequências alternativas, resultando em inconsistência e *bugs*.

```
saque (conta, valor) {
    saldo_anterior = get_saldo(conta);
    saldo_atual = saldo_anterior - valor;
    set_saldo(conta, saldo_atual);
}
```

Imagine que duas *threads* executam a função `saque()` para a mesma conta ao mesmo tempo. Caso a primeira *thread T1* execute `saldo_anterior = get_saldo(conta);` e depois ocorra uma mudança de contexto e a segunda *thread T2* execute a mesma linha, apenas um dos saques será registrado, porém ambos serão realizados, tornando o sistema inconsistente.

Questão 6

Pode ocorrer, por exemplo, um *deadlock* na seguinte situação:

1. A *thread T1* faz uma chamada `acquire()` e executa a linha `lock->interested[this_thread] = 1;`
2. Ocorre uma **troca de contexto** para a *thread T2*;
3. A *thread T2* faz uma chamada `acquire()` e executa a linha `lock->interested[this_thread] = 1;`

A partir de agora, **ambas as *threads* ficam presas no loop de `while`**, pois ambas as posições de `lock->interested[]` são verdadeiras (`== 1`).

Questão 7

```
//Se lock==false, está livre
//Se lock==true, está trancada
//Começa livre
bool lock = false;

acquire(bool lock) {
    bool acquired = true;
    while(acquired) {
        swap(&acquired, &lock);
    }
```

```
}

release(bool lock) {
    bool temp = false;
    swap(&lock, &temp);
}
```

Questão 8

Os padrões que podem ser impressos são sequências da combinação "ab", nessa ordem.

Não pode haver *deadlock*, uma vez que, para escrever o "a", há de se passar em um semáforo binário (`s1`). Uma vez que esse semáforo foi passado ele só é reaberto após se passar por um segundo semáforo binário (`s2`), imprimir "b" e abrir o semáforo `s2`. Ambas as chamadas de `print()` estão dentro da região crítica de `s1` é serializada para acesso de uma *thread* por vez.

Questão 9

O código ficaria similar ao apresentado a seguir:

Global:

```
semaphore mutex = 0;
semaphore empty = N;
semaphore full = 0;
```

Produtor:

```
void producer()
{
    wait(mutex)
    wait(empty)
        //região crítica:
        //adiciona produto
    signal(mutex)
    signal(full)
}
```

Consumidor:

```
void consumer()
{
    wait(full)
    wait(mutex)
        //região crítica:
        //retira produto
    signal(mutex)
    signal(empty)
    //processa produto
}
```

Nesse caso poderia ocorrer um *deadlock* na seguinte situação:

1. Um produtor *P[1]* entra nos dois semáforos `wait(mutex)` e `wait(empty)`, adiciona um produto e chama `signal(mutex)`;
2. Ocorre uma troca de contexto e entra o produtor *P[2]*;
3. *P[2]* e faz exatamente o que foi feito por *P[1]*;
4. Isso ocorre consecutivamente até *P[N]*, fazendo com que `empty == 0` e `full == 0`;
5. Mais um produtor *P[N+1]* entra, passa por `wait(mutex)` e bloqueia em `wait(empty)`.

Nesse caso nenhum consumidor vai conseguir passar de `wait(full)`, pois nunca houve uma chamada `signal(full)`. E se algum dos N produtores iniciais voltar a executar e chamar `signal(full)`, os produtores e consumidores vão travar no `wait(mutex)`, pois *P[N+1]* está na região crítica definida pelo mesmo.

Questão 10

O `signal(s)` de **semáforos** incrementa o semáforo `s`, premitindo que um semáforo bloqueado entre na *região crítica*.

Já no caso de **monitores**, a função `signal(vc)` acorda uma *thread* bloqueada num `wait()` na *variável de condição* `vc`.

Extra: `wait()` em Semáforos vs. Monitores

Em **semáforos**, a função `wait(s)` decrementa o semáforo `s` se o mesmo `> 0`, e bloqueia caso contrário, esperando até que o mesmo seja incrementado por alguma *thread*.

Em **monitores**, a função `wait(vc)` bloqueia a *thread* esperando na *variável de condição* `vc`, passa a região crítica para outra *thread* e espera até alguma *thread* a acordar com um `signal(vc)`.

Questão 11

Na **semântica de Mesa**, ao chamar `signal()` a *thread* que está esperando na variável de condição passa para o estado *ready*, mas a *thread* que chamou `signal()` continua executando. Portanto, não é garantido que quando a *thread* que passou para *ready* entra em execução, a condição ainda é verdadeira.

Extra: *Hoare Monitors*

Na **semântica de Hoare**, a chamada de `signal()` troca o contexto imediatamente, colocando em execução a *thread* que estava em wait. Dessa maneira, garante que a condição é verdadeira quando entra em execução.

Questão 12

Na arquitetura **cliente-servidor** existem dois papéis distintos do sistema, um dos quais é o *servidor* (normalmente uma só intância) ao qual múltiplos *clientes* se conectam. Cada um dos dois (*cliente* e *servidor*) executam programas diferentes.

Já na arquitetura **par-a-par** (P2P), todas as partes do sistema desempenham o mesmo papel (ou papéis semelhantes). Essas partes que conectam entre si de forma bidirecional, formando uma malha. Nesse tipo de sistema todos os participantes desempenham ao mesmo tempo o que seria o papel de cliente e servidor, e a demanda de um cliente pode ser suprida pelos demais, tornando o sistema mais escalável e distribuído.

Questão 13

Na **forma iterativa**, quando um *host* quer saber um endereço IP, pergunta para o *Local Name Server*. Se o mesmo não souber, vai perguntar para o *Root Name Server* e se o mesmo não souber, vai retonar um endereço para o qual o pode repetir a pergunta (um *TLD Name Server*, por exemplo). Isso se repete até que seja encontrada a resposta, que é repassada para o *host*.

Na **forma recursiva**, quando um *host* quer saber um endereço IP, pergunta para o *Local Name Server*. Se o mesmo não souber, passa a pergunta para outro *Name Server* e assim consecutivamente até que se chegue a uma resposta. Ao obter a resposta, a mesma é passada de volta até o *host* que originalmente havia feito a pergunta.

Enquanto o método **iterativo** põe a carga de resolução de nomes no *Local Name Server* que vai iterando pelos *Name Servers* até encontrar uma resposta, o método **recursivo** poria o peso da tarefa de resolução de nomes nos *Root Name Servers*, por onde passariam todos os requests do mundo.

Porém, o *DNS* é implementado com *caching*, ou seja, os *Name Servers* guardam respostas para perguntas que responderam recentemente. Nesse caso o **método recursivo é vantajoso** pois ao passar a resposta do respondente até o *host* (do qual partiu a pergunta), a mesma é armazenada nos *caches* em todo o caminho. Logo, a partir daquele momento, toda vez que essa chegar a um desses *Name Servers* do caminho, será respondida imediatamente. Assim, a sobrecarga nos *Root Name Servers* desaparece, pois a maioria dos pedidos de resolução vai encontrar uma resposta sem precisar chegar a um *Root Name Server*.

Questão 14

Apenas o *Authoritative DNS server* `cos.ufrj.br` precisa saber o endereço IP do *Authoritative Name Server* `lab.cos.ufrj.br`.

```
Root DNS Servers
|
|-- Top-level Domain (TLD) server: .br
|    |
|    |-- Authoritative DNS server: ufrj.br
|         |
|         |-- Authoritative DNS Server: cos.ufrj.br
|              |
|              |-- Authoritative DNS Server: lab.cos.ufrj.br
|
```

Questão 15

A estratégia das **CDNs** é disponibilizar o conteúdo mais próximo dos usuários, seja **dentro de redes de acesso** ou **perto de *POP*s** (*Points of Presence*, pontos nos quais *Internet Service Providers* se conectam), agilizando o envio do conteúdo aos mesmos.

Questão 16

Ao solicitar a um conteúdo de uma página que está disponível em uma **CDN** o **Authoritative Name Server** que serve a página retorna a URL do conteúdo na **CDN**. Então essa **URL é resolvida pelo Authoritative Name Server da CDN** e o cliente usa o endereço resolvido para requisitar o conteúdo.

A resolução da URL do conteúdo pelo **Authoritative Name Server da CDN** pode ser feita escolhendo o nó da **CDN** mais próximo geograficamente do cliente ou com o menor atraso até o mesmo, por exemplo. Para a segunda opção os nós da **CDN** periodicamente fazem *ping* para os pontos de acesso dos **ISPs** (*Internet Service Providers*) e mantém os resultados no **DNS da CDN**.

Outra opção é responder ao cliente uma **lista de servidores CDN** da qual o mesmo vai escolher qual usar baseando-se, por exemplo, no resultado de *pings* para os seridores da lista.

Questão 17

Ao entrar no *swarm*, um cliente pode receber *peers* com partes muito distintas do arquivo, o que faz com que demore mais tempo para que ele tenha uma parte completa para começar a fazer o *upload*, consequentemente arriscando que alguns *peers* parem de enviar para o mesmo (*choking*).

Além disso, em *swarms* muito grandes, há a possibilidade dos *peers* aleatórios serem geograficamente muito distantes, atrapalhando as taxas de transmissão e consequentemente podendo levar ao *choke* do cliente.

Questão 18

Sim, é possível, porém o cliente alterado depende de *optimistic unchokes* e/ou de *seeders* para receber seus *chunks*:

- Da maneira que o protocolo BitTorrent funciona, de forma resumida, os *peers* que recebem *chunks* (pedaços dos arquivos solicitados) são aqueles que enviam *chunks* numa taxa mais alta para seus *peers* (a taxa de *upload* é o importante para manter uma conexão). Porém, a cada 30 segundos um *peer* aleatório é escolhido e passa a receber *chunks* (*optimistic unchoke*), o que só continua caso o mesmo corresponda enviando *chunks* a uma taxa alta.
- Uma excessão é o caso de clientes que já possuem o arquivo inteiro (*seeders*), seus *peers* são escolhidos de acordo com a velocidade de *upload* que o cliente consegue manter com o *peer*.

De qualquer maneira o download provavelmente seria muito mais lento do que se fosse utilizado um cliente com o protocolo BitTorrent original.

Questão 19

Através da *função de hash* o *peer* entrante descobre qual sua posição na **DHT**, se conecta com algum nó da **DHT** e pergunta quais seriam os antecessor e sucessor de sua posição. Quando recebe a resposta o *peer* entrante atualiza seu sucessor e notifica o antecessor para que atualize o sucessor para si próprio (o entrante).

Questão 20

Em uma **DHT** (**Distributed Hash Table**), no momento em que um par **A** busca o valor de uma chave k que não conhece, passa a busca para o par seguinte e assim consecutivamente até que chegue em um par **B** que sabe a resposta e o valor associado a k para **A**. Caso **A** busque o mesmo valor posteriormente, o mesmo mecanismo é repetido.

Ao utilizarmos *caching*, após a primeira busca o resultado (ou o endereço de quem sabe aquela faixa de resultados) pode ser armazenado por **A**. Assim, quando a mesma busca for repetida, o resultado é obtido de maneira muito mais rápida.

Extras:

1 - Hierarquia DNS:

```
Root DNS Servers
|
|-- Top-level Domain (TLD) server: .br
|    |
|    |-- Authoritative DNS server: ufrj.br
|         |
|         |-- Local Name Server: cos.ufrj.br
|
|-- Top-level Domain (TLD) server: .com
|    |
|    |-- Authoritative DNS server: yahoo.com
|
|-- Top-level Domain (TLD) server: .edu
|    |
|    |-- Authoritative DNS server: umass.edu
|
...
```