Time: Sarapatel

Alunos: Giancarlo, Flavimar, Luca e Lucas Oliveira

Atividade Parte 2

Questão 1:
Basicamente o problema anterior (Produtor/Consumidor) é um problema de buffer, onde o buffer (message) não pode ser preenchido quando já está cheio e não pode ser esvaziado quando já está vazio. Portanto, a message deve ser bloqueada para acesso, dependendo da condição. Utilizando a implementação da forma como está, o problema que pode ocorrer é a espera ocupada, pois se tiverem mais consumidores do que produtores, o consumidor ficará esperando até que o produtor produza, mas se não tiver mais produtores para produzir, o consumidor ficará esperando infinitamente. Uma solução para resolver o problema da espera ocupada, seria utilizando semáforo, onde cada thread só será autorizado entrar na região crítica, se ela tiver livre e com isso, não haverá espera ocupada

Questão 2:
A figura demonstra o funcionamento de um mutex.
Há dois processos, A e B, compartilhando o acesso à região crítica.
Em T1, o processo A entra na região crítica, que se torna bloqueada.
Em T2, o processo B tenta acessar a região crítica, que se encontra bloqueada pelo processo A.
Em T3, o processo A deixa a região, liberando-a para que o processo B entre na região crítica.
Em T4, o processo B deixa a região crítica.

Questão 3:
Os prós:
- Protege a região crítica
- É simples

Os contras:
- A cpu é mal utilizada executando um processo que não faz nada
- Se o algoritmo é mal codificado, pode causar loop infinito
- Possibilidade de bloqueio indefinido em ambos os processos

Algumas soluções para a espera ocupada são:
- sleep e wake up: faz o processo ser bloqueado ou desbloqueado,
- semáforos: Um array que agrupa vários sleep e wake up
- monitores: é um conjunto de procedimentos

Pois todas elas fazem o processo dormir em vez de ocupar processamento desnecessariamente

Questão 4:

Uma thread aloca tempo no processador , sendo ela uma linha de execução de um processo, ou seja, uma parte de um processo. Já um processo é basicamente um programa em execução. Se tratando de programação concorrente, podem existir N threads trabalhando de forma concorrente dentro de um processo.

Questão 5:
Safety está relacionado a que nada de errado aconteça, e caso aconteça, seja tratado. Por sua vez, Liveness está relacionado a sempre manter a aplicação funcionando.
Por exemplo, um deadlock representa uma quebra ao "liveness" da aplicação, pois impede essa de seguir sua execução corretamente. Safety já adiciona uma lógica de tratamento de exceção em deadlock.

Questão 6:

- **a)** Sim, existem dois processos executando, um de realizar o download, e outro de atualizar a porcentagem, com isso, para que o processo de atualizar a porcentagem aconteça, é necessário ter informações do processo de download, fazendo com que tenha que existir o compartilhamento de um recurso, sendo assim concorrente.
- **b)** Não é necessário concorrência, uma vez que só existe a ação de download, e o recurso não é editado, portanto, as requisições podem ser realizadas simultaneamente sem que exista competição de recurso. Sim, existe um arquivo em algum repositório, recebendo várias requisições, fazendo com que exista uma concorrência neste recurso
- **c)** Sim, n-requisições, com isso existem várias threads executando.

Questão 7:

- **a)** Sim, pois várias threads ou processos executam uma atividade independente de um mesmo objetivo
- **b)** Sim, pois na concorrência o recurso só pode ser acessado por um processo de cada vez para não gerar inconsistência
- **c)** Não, pois os processos são iguais porém independentes um do outro e dependentes do mesmo recurso
- **d)** Não, pois no paralelismo os processos não compartilham o mesmo recurso

Questão 8:

a) Muitas vezes, a concorrência quando mal implementada, leva a um aumento de uso dos recursos que não é traduzido em eficiência, pois podem ocorrer interdependência entre os processos ou há uma necessidade das instruções serem sequenciais. Isso também pode desencadear em lentidão da aplicação.
b) O projeto muda totalmente ao se usar programas concorrentes, uma vez que é necessário levar em consideração que um recurso pode ser compartilhado, e isso deve ser pensado e tratado.
c) Em muitos casos, a sincronização dos processos para evitar condições como por exemplo deadlocks pode ser bem complicado.

Questão 9:
Situação 1: X = 10

a)

Situação 1: X = 10

p1 é escalado

p1 = 10

p2 é escalado

p2 = 10

Situação 2: X = 11

p1=12 e p2=12

p1=12 e p2=13

Situação 3: X = 12

p1=13 e p2=13

p1=14 e p2=13

b) Usando monitores temos o seguinte código:

```
public class X{
        private int lastIdUsed;
        public synchronized int getNextId(){
                return ++ lastIdUsed;
        }
}
```

Situação 1: X = 10

p1 é escalado

p1=10

p2 é escalado

p2=11

como p1 é 10 então p2 vai ser 11 por conta que com o synchronized a variável lastIdUsed é atualizada antes que seja atribuída ao processo 2 por conta da solução dos monitores que trava a região crítica e define um invariante que define a execução das operações anteriores ante que outra operação seja executada'