Universidade Federal de Uberlândia (UFU)

Faculdade de Computação (FACOM)

Disciplina: GBC045 - Sistemas Operacionais

Prof. Rivalino Matias Jr.

Exercícios de Fixação (Unidade III)

1) De acordo com a teoria de SO estudada na Unidade III desta disciplina, correlacione os três conceitos a seguir: **exclusão mútua**, **região crítica** e ***deadlock***.

2) Qual a vantagem de se utilizar semáforo em relação a um algoritmo de espera ocupada?

3) Marque a(s) alternativa(s) correta(s), e justifique sua resposta para as assertivas consideradas incorretas.
   a) O uso de semáforos é necessário em aplicações *multithreaded*.
   b) O acesso concorrente a recursos dentro do *kernel* é controlado por meio de semáforos.
   c) O acesso concorrente a recursos dentro do *kernel* é controlado por meio de mecanismos do tipo *busy wait*.
   d) Algoritmos de espera ocupada (*busy wait*) não causam *deadlock*, apenas aqueles baseados em semáforos.
   e) Algoritmos de espera ocupada não devem ser usados e *user-level*, apenas em *kernel-level*.

4) Marque V (verdadeiro) ou F (falso). No caso das assertivas marcadas como **F**, explique o que está errado:
   ( ) Para a ocorrência de *deadlock* é necessário apenas que dois recursos sejam compartilhados entre dois processos.
   ( ) Mecanismos de exclusão mútua são usados para resolver o problema de *deadlock*.
   ( ) Um processo faz uso de um recurso compartilhado. Neste caso, existe uma única região crítica.
   ( ) Três processos compartilham dois recursos, portanto cada processo possui no mínimo duas regiões críticas.

5) Marque a(s) alternativa(s) correta(s) e justifique sua resposta para as assertivas consideradas incorretas.
   a) O uso de *pipes* somente é possível entre processos executando no mesmo sistema operacional.
   b) O uso de *pipes* somente é possível entre processos filhos do mesmo processo.
   c) Somente é possível o uso de *pipes* em processos com múltiplas *threads*.
   d) O uso de semáforos somente é possível entre processos no mesmo sistema operacional.

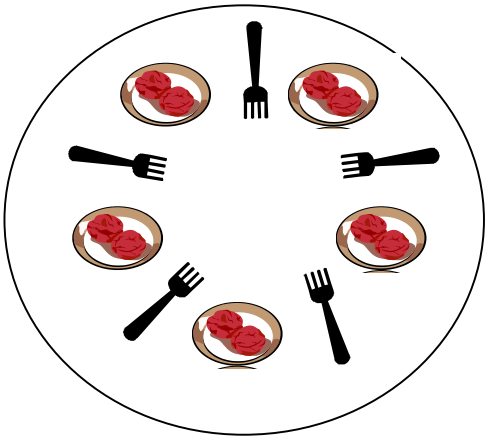
6) Faça um programa que ao ser executado crie dois processos filhos, P2 e P3. P2 deve utilizar um *pipe* para enviar todos os valores pares de 0 – 100 para P3. Já P3, deve usar um segundo *pipe* para enviar os valores ímpares de 0 – 100 para P2. Tanto P2 quanto P3 devem escrever na tela todos os valores recebidos.

7) Altere o programa acima para usar threads e não processos. Neste caso, o processo principal cria duas threads, T2 e T3, as quais realizam as mesmas tarefas dos processos P2 e P3 do exercício anterior.

8) Você está desenvolvendo uma aplicação de controle de um VANT (veículo aéreo não tripulado). Essa aplicação é composta de 6 processos. Cinco processos fazem interface (E/S) com um conjunto de 50 portas de comunicação (10 por processo). A comunicação com cada porta é feita por meio de threads. A arquitetura da aplicação é assim composta. Um processo (P1) é o pai dos demais processos (P2 - P6). Cada processo filho (P2 - P6) possui 10 threads, onde cada thread comunica-se com uma porta de comunicação. As portas são identificadas como COM1 - COM50, onde as 10 primeiras são gerenciadas por P2 e assim sucessivamente, onde P6 gerencia as últimas 10 portas (COM41-COM50). Os processos P2 até P6 trabalham lendo valores (int) das suas respectivas portas, sempre na ordem COMi, COMi+1, ..., COMi+9. Após ler um valor de todas as portas, cada processo filho envia o somatório desses valores para P1, o que define um ciclo de operação da aplicação. A cada ciclo, P1 verifica os cinco valores recebidos. Se os cinco valores forem iguais a 0, então a aplicação encerra sua execução, caso contrário, executa um novo ciclo. Para ler de uma porta de comunicação a aplicação deve usar a função *v*=read_com(*p*), onde *p* é um valor inteiro que indica o número da porta (ex. 23) e *v* é uma variável que recebe o valor de retorno do sensor associado com a respectiva porta *p*. A comunicação entre os processos deve ser realizada usando o mecanismo de *pipe* estudado na disciplina. Abaixo a implementação da função *read_com*().

```
#include <stdlib.h>
int read_com(int port){    v=rand()%5; return ++v; }
```

9) Fazer um programa, usando *threads*, para ilustrar o problema clássico do **Jantar dos Filósofos** (*ver livro texto da disciplina*). O desenho abaixo ilustra este cenário.



Em frente de cada prato existe um filósofo sentado, ou seja, existem 5 filósofos sentados na mesa. Para comer o seu espaguete, cada filósofo precisa de dois garfos. O comportamento do filósofo se resume a **comer** e **pensar**. Desta forma, aleatoriamente, o filósofo deve comer e pensar, indefinidamente. A ação de **comer** significa pegar dois (2) garfos, primeiro o do lado esquerdo e depois do lado direito, um em cada mão, e aguardar um determinado intervalo de tempo aleatório (rand(3)). Posteriormente, o filósofo deve deixar os garfos sobre a mesa, novamente primeiro o do lado esquerdo e depois do lado direito, a fim de pensar um pouco. A ação de **pensar** também deve ser implementada como uma inatividade por um determinado período de tempo aleatório.

Nesta implementação, propositalmente, você não deve usar qualquer mecanismo de exclusão mútua com o objetivo de experimentar o problema de inconsistência no uso dos recursos compartilhados..

10) Alterar o programa do exercício #9 para incluir o uso de exclusão mútua entre *threads* (filósofos) vizinhas que compartilham o mesmo garfo. Utilizar o mecanismo de semáforos.

11) Como pode ser observado no resultado do exercício #10, a implementação com semáforo resolverá o problema de compartilhamento de recursos, mas causará o problema de *deadlock*. Neste exercício, você deverá fazer mudanças no seu código a fim de eliminar as condições de *deadlock*.