Aluno: João Victor da Silva Prado
Sistemas Operacionais
Avaliação 6

1) R: Pois um semáforo representa uma seção crítica cujo conteúdo não é acessível ao programador. Sendo assim, o semáforo não usa operações de 'read' ou 'write'. Na verdade, internamente, um semáforo possui um contador interno e uma fila de tarefas inicialmente vazia. Assim no lugar de read e write ele terá 2 operações primitivas:

- Up(s): Que é invocado para liberar a seção crítica associada ao semáforo, onde haverá um incremento no contador interno e este contador será testado (um contador negativo ou nulo indica que há tarefas suspensas naquele semáforo)
- Down(s): Usado para solicitar acesso à seção crítica associada ao semáforo. Decrementa o contador e o testa (se ele for negativo, a tarefa solicitante vai ser adicionada na fila do semáforo e suspensa.

2) R:

t: tarefa que invocou a operação
s: semáforo contendo um contador e uma fila
v: valor (inteiro)

```
fun down (t, s) {
    s.counter --;
    if(s.counter <0){
        acrescentar(t, s.queue); // coloca t no fim da fila
        suspender(t); // tarefa t é suspensa
    }
}
fun up(s){
    s.couter ++;
    if(s.counter ≤0){
        v= first(s.queue); // retira 1ª tarefa da fila
        awake(v); // devolve v à fila de tarefas prontas
    }
}
fun aux() {
    // função para inicializar semáforo
    s.counter = v;
    s.queue = [];          // fila começa vazia
}
```

D S T Q Q S S

3) R: Na situação em questão um impasse não poderá ocorrer, pois a preempção é permitida, o que quer dizer que um processo ao solicitar um recurso em qualquer momento (mesmo que um processo bloqueado tenha reservado esse recurso) terá esse recurso transferido para ele. Desse forma o novo processo não vai precisar esperar a execução do processo bloqueado para acessar o recurso.

4) R: Para que um deadlock ocorra, 4 condições precisam ocorrer simultaneamente:

- Exclusão mútua: Em um determinado momento, cada um dos recursos estará ou associado a um único processo ou disponível.
- Posse e espera: cada processo pode solicitar um recurso, ter esse recurso alocado para si e ficar bloqueado esperando por outro recurso.
- Não preempção: recursos concedidos previamente não podem ser tomados forçosamente.
- espera circular: Deve haver uma cadeia circular com dois ou mais processos, onde cada um encontra-se à espera de um recurso que está sendo usado pelo membro seguinte dessa cadeia.

Portanto, como podemos ver nas condições citadas, não é possível haver um deadlock envolvendo um único processo, para sua ocorrência são necessários 2 processos ou mais.

5) a) Incorreta. A detecção e recuperação de impasses, apesar de ser uma abordagem interessante, é relativamente pouco utilizada porque o custo de detecção pode ser elevado (visto que algoritmos de busca de ciclos em grafos tem custo computacional alto) e as alternativas de resolução sempre implicam perder tarefas ou parte das execuções já realizadas.

b) Incorreta. Na verdade essa técnica consiste em fazer o sistema retornar a um estado seguro anterior ao impasse e operações envolvendo a rede ou interações com o usuário podem ser difíceis ou impossíveis de retroceder.

c) Correta

d) Incorreta. Como dito anteriormente algoritmos de busca de ciclos em grafos tem custo computacional elevado, o que faz com que sua ativação com muita frequência possa prejudicar o desempenho do sistema.

6) R: Há um problema no "count==n", pois se o n-esimo encadeamento for interrompido neste ponto e o n-esimo encadeamento vem do mutex ambos vão descobrir "count==n" e vão sinalizar para destravar a barreira

7) R:

```
int numero_clientes = 0;
sem mutex = semaphore(1);
sem cliente = semaphore(0);
sem barbeiro = semaphore(0);
int num_cadeiras;
cliente(){
    down(mutex); //entra na região crítica
    if(numero_clientes != num_cadeiras){
        numero_clientes = numero_clientes + 1;
        up(cliente); // há um novo cliente para cortar o cabelo
        up(mutex); //sai da região crítica
        down(barbeiro); //aguarda o barbeiro cortar o cabelo
        ObterCorte(cabelo);
    }else{
        volta();
        up(mutex); //sai da região crítica
    }
}
barbeiro(){
    while(1){
        down(cliente); // dorme sem cliente
        down(mutex); // entra na região crítica
        numero_clientes = numero_clientes - 1;
        up(mutex); // barbeiro pronto
        up(barbeiro);
        cortarCabelo();
    }
}
```