- Como caracteriza os processsos em termos de interacção? Mostre como em cada categoria se coloca o problema da exclusão mútua.

• *processos independentes* – quando são criados, têm o seu 'tempo de vida' e terminam sem interactuarem de um modo explícito; a interacção que ocorre é implícita e tem origem na sua *competição* pelos recursos do sistema computacional; trata-se tipicamente dos processos lançados pelos diferentes utilizadores num ambiente interactivo e/ou dos processos que resultam do processamento de *jobs* num ambiente de tipo *batch*; é da responsabilidade do *SO* garantir que a atribuição do recurso seja feita de uma forma controlada para que não haja perda de informação; • isto impõe, em geral, que só um processo de cada vez pode ter acesso ao recurso (*exclusão mútua*).

• *processos cooperantes* – quando partilham informação ou comunicam entre si de um modo explícito; a *partilha* exige um espaço de endereçamento comum, enquanto que a *comunicação* pode ser feita tanto através da partilha de um espaço de endereçamento, como da existência de um canal de comunicação que interliga os processos intervenientes. é da responsabilidade dos processos envolvidos garantir que o acesso à região partilhada seja feito de uma forma controlada para que não haja perda de informação; • isto impõe, em geral, que só um processo de cada vez possa ter acesso á região partilhada(exclusão mútua).

- O que é a competição por um recurso? Dê exemplos concretos de competição em, pelos menos, duas situações distintas.

Competição por um recurso é no mesmo instante de tempo haver mais do que uma entidade(processo) a querer só para si um determinado recurso. Por exemplo o processador, ou um acesso de uma posição de memória.(???)

- Quando se fala em região crítica, há por vezes alguma confusão em estabelecer-se se se trata de uma *região crítica* de código, ou de dados. Esclareça o conceito. Que tipo de propriedades devem apresentar as primitivas de acesso com exclusão mútua a uma região crítica?

A região crtica de código é a execução por parte do processador do código que deseja aceder uma região critica de dados
Uma região critica é código que manipula recursos ou dados partilhados por vários processos ao mesmo tempo.
As propriedades das primitivas de acesso são:

- □□□□□□□□ Garantia efectiva de imposição da exclusão mútua. Só pode aceder um processo de cada vez á região crítica.
- □□□□□□□□ Um processo fora da região critica não pode impedir outros de lá entrar.
- □□□□□□□□ Os processos que aguardam acesso á região crítica não podem bloquear um processo que lhe acede no momento.
- □□□□□□□□ Não pode ser adiado indefinidamente a possibilidade de acesso à região crítica a um qualquer processo.
- □□□□□□□□ A solução não pode ser feita partindo do presuposto do tempo de execução de um processo. Independente da velocidade de execução.
- □□□□□□□□ O processo que tem a posse da região crítica não pode lá ficar indefinidamente. Tempo de permanência na região crítica tem que ser finito.

- Não havendo exclusão mútua no acesso a uma região crítica, corre-se o risco de inconsistência de informação devido à existência de *condições de corrida* na manipulação dos dados internos a um recurso, ou a uma região partilhada. O que são condições de corrida? Exemplifique a sua ocorrência numa situação simples em que coexistem dois processos que cooperam entre si.

São situações em que mais que um processo deseja aceder a uma região critica. Por exemplo dois processos que estão a alterar o mesmo ficheiro. (???)

- Distinga *deadlock* de adiamento indefinido. Tomando como exemplo o *problema dos produtores / consumidores*, descreva uma situação de cada tipo.

'deadlock'-quando dois ou mais processos ficam a aguardar eternamente a sua entrada nas respectivas regiões críticas, esperando acontecimentos, que se pode demonstrar, que nunca irão acontecer; o resultado é por isso um bloqueio das operações;

•***adiamento indefinido***-quando um ou mais processos competem pelo acesso às respectivas regiões críticas e, devido a uma

conjunção de circunstâncias em que surgem continuamente processos novos (de prioridade mais elevada, por exemplo) que o(s) ultrapassam nesse desígnio, o seu acesso é sucessivamente adiado; está-se por isso perante um impedimento real à sua continuação.

- A solução do problema de acesso com exclusão mútua a uma região crítica pode ser enquadrada em duas categorias: soluções ditas *software* e soluções ditas *hardware*. Quais são os pressupostos em que cada uma se baseia? Qual é a vantagem principal que boa parte das soluções 'hardware' apresenta sobre as soluções 'software'?

• *soluções software* – são soluções que, quer sejam implementadas num monoprocessador, quer num multiprocessador com memória partilhada, supõem o recurso em última instância ao *conjunto de instruções básico* do processador;
ou seja, as instruções de transferência de dados de e para a memória são de tipo *standard*: leitura e escrita de um valor; a única suposição adicional diz respeito ao caso do multiprocessador, em que a tentativa de acesso simultâneo a uma mesma posição de memória por parte de diferentes processadores é necessariamente serializada por intervenção de um árbitro;

• *soluções hardware* – são soluções que supõem o recurso a instruções especiais do processador para garantir, a algum nível, a atomicidade na leitura e subsequente escrita de uma mesma posição de memória; são muitas vezes suportadas pelo próprio sistema de operação e podem mesmo estar integradas na linguagem de programação utilizada.

A principal vantagem das soluções de 'hardware' em relação ás de 'sofware' é o facto de os processos que aguardam entrada na região crítica ficam bloqueados, ou seja, não ficam em 'busy-waitng' (processos interveniente aguradam entrada na região crítica no estado activo) e por conseguinte não estão a consumir tempo de processador, isto faz com que as soluções de hardware aumentem a eficiência e eliminem os constrangimentos no estabelecimento no algoritmo de scheduling, porque um processo de maior prioridade não consegue bloquear um processo de menor prioridade que se encontre aceder á região crítica.

7.8.9 -> o borges acha de certeza k e mais genio k estes palermas por isso não sai de certeza

O que é o *busy waiting*? Porque é que a sua ocorrência se torna tão indesejável? Haverá algum caso, porém, em que não é assim? Explique detalhadamente.

Busy-Waiting é um problema comum às soluções *software* e ao uso de *flags de locking*, implementadas a partir de instruções de tipo *read-modify-write*, é que os processos intervenientes aguardam entrada na região crítica no estado activo Tem como inconvenientes a perda de eficiência, o processo pode ser calendarizado para execução mas não executar nada ou pode mesmo provocar constrangimentos no processo de shedulling, bloqueando em execução e entrar numa sistuação de deadlock.Em sistemas computacionais multiprocessador com memória partilhada, e mais concretamente no caso de multiprocessamento simétrico, o problema de *busy waiting* não é tão crítico.

- O que são *flags de locking*? Em que é que elas se baseiam? Mostre como é que elas podem ser usadas
para resolver o problema de acesso a uma região crítica com exclusão mútua.

Flags de locking é um processo em que a leitura e a escrita de uma posição de memoria seja uma acção atómica de forma a garantir exclusividade mutua no acesso a uma região critica. Ora, como as instruções convencionais do *conjunto de instruções* de um processador efectuam apenas a leitura ou a escrita de uma posição de memória na mesma instrução, a implementação de uma *flag de locking* não é possível com elas.(???)

- O que são semáforos? Mostre como é que eles podem ser usados para resolver o problema de acesso a
uma região crítica com exclusão mútua e para sincronizar processos entre si.

Um semaforo é um dispositivo que usa uma variavel tipo inteiro (exemplo: val) que não pode tomar valores negativos e sobre o qual só podem ser efectuadas duas operações atomicas, que são o sem_up (se existirem processos bloqueados são acordados , caso contrário a variavel val é incrementada) e o sem_down (se o valor de val for zero o processo que executou o sem_down é bloqueado, caso contrário a váriavel val é decrementada)

Para utilizar semaforos para aceder a uma região crítica, inicializa-se um semaforo (por exemplo: acesso) com o valor da váriavel val igual a 1. Se o processo P1 pretende aceder à RC executa a função sem_down(acesso) colocando a variavel val a zero sem ficar bloqueado executando o código que necessitar dentro da RC, se outro processo P2 quiser aceder à RC efectua a função sem_down(acesso), como o valor de val é zero o processo P2 fica bloqueado á espera que P1 liberte a região critica. Assim que o processo P1 já não necessitar da RC efectua a função sem_up(acesso), como o valor val é zero e existe o processo P2 em espera este é acordado e entra na RC, o valor de val continua a zero.

- O que são monitores? Mostre como é que eles podem ser usados para resolver o problema de acesso a

uma região crítica com exclusão mútua e para sincronizar processos entre si.

Um *monitor* é um dispositivo de sincronização, proposto de uma forma independente por Hoare e Brinch Hansen, que pode ser concebido como um módulo especial, suportado pela linguagem de programação [concorrente] e constituído por uma estrutura de dados interna, por código de inicialização e por um conjunto de primitivas de acesso.
Quando as estruturas de dados são implementadas com *monitores*, a linguagem de programação garante que a execução de uma primitiva do *monitor* é feita em regime de exclusão mútua. Assim, o compilador, ao compilar um *monitor*, gera o código necessário para impor esta situação. Um *thread* entra no *monitor* por invocação de uma das suas primitivas, o que constitui a única forma de acesso à estrutura de dados interna. Como a execução das primitivas decorre em regime de exclusão mútua, quando um outro *thread* está no seu interior, o *thread* é bloqueado à entrada, aguardando a sua vez.
A sincronização entre *threads* é gerida pelas *variáveis de condição*. Existem duas operações que podem ser executadas sobre uma *variável de condição*
wait – o *thread* que invoca a operação é bloqueado na *variável de condição* argumento e é colocado *fora do monitor* para possibilitar que um outro *thread* que aguarda acesso, possa prosseguir;
signal – se houver *threads* bloqueados na *variável de condição* argumento, um deles é acordado; caso contrário, nada acontece.

14, na interessa

- Que vantagens e inconvenientes as soluções baseadas em monitores trazem sobre soluções baseadas em semáforos?

Conceptualmente, o principal problema com o uso dos *semáforos* é que eles servem simultaneamente para garantir o acesso com exclusão mútua a uma região crítica e para sincronizar os processos intervenientes. Assim, e porque se trata de primitivas de muito baixo nível, a sua aplicação é feita segundo uma perspectiva *bottom-up* (os processos são bloqueados antes de entrarem na região crítica, se as condições à sua continuação não estiverem reunidas) e não *top-down* (os processos entram na região crítica e bloqueiam, se as condições à sua continuação não estiverem reunidas).
A primeira abordagem torna-se logicamente confusa e muito sujeita a erros, sobretudo em interacções de alguma complexidade, porque as primitivas de sincronização podem estar dispersas por todo o programa.
Uma solução é introduzir *ao nível da própria linguagem de programação* uma construção [concorrente] que trate separadamente o acesso com exclusão mútua a uma dada região de código e a sincronização dos processos.

- Em que é que se baseia o paradigma da passagem de mensagens? Mostre como se coloca aí o problema do acesso com exclusão mútua a uma região crítica e como ele está implicitamente resolvido?

O princípio em que se baseia é muito simples:
• sempre que um processo PR, dito *remetente*, pretende comunicar com um processo PD, dito *destinatário*, envia-lhe uma mensagem através de um canal de comunicação estabelecido entre ambos (*operação de envio*);
• o processo PD, para receber a mensagem, tem tão só que aceder ao canal de comunicação e aguardar a sua chegada (*operação de recepção*).

A *troca de mensagens* só conduzirá, porém, a uma comunicação fiável entre os processos *remetente* e *destinatário*, se for garantida alguma forma de sincronização entre eles.
O grau de sincronização existente pode ser classificado em dois níveis
• *sincronização não bloqueante* - quando a sincronização é da responsabilidade dos processos intervenientes; a *operação de envio* envia a mensagem e regressa sem qualquer informação sobre se a mensagem foi efectivamente recebida; a *operação de recepção*, por seu lado, regressa independentemente de ter sido ou não recebida uma mensagem;

• *sincronização bloqueante* - quando as operações de envio e de recepção contêm em si mesmas elementos de sincronização; a *operação de envio* envia a mensagem e bloqueia até que esta seja efectivamente recebida; a *operação de recepção*, por seu lado só regressa após receber uma mensagem.

O resto na interessa

- Distinga as diferentes políticas de *prevenção de deadlock no sentido estrito*. Dê um exemplo ilustrativo de cada uma delas numa situação em que um grupo de processos usa um conjunto de blocos de disco para armazenamento temporário de informação.

A politica de prevenção de deadlock no sentido estrito tem como finalidade a negação de uma das 4 condições que originam deadlock para deste modo eliminá-lo:

• *negação da condição de exclusão mútua* – só pode ser aplicada a recursos passíveis de partilha em simultâneo;

• *negação da condição de espera com retenção* – exige o conhecimento prévio de todos os recursos que vão ser necessários, considera sempre o pior caso possível (uso de todos os recursos em simultâneo);

• *imposição da condição de não libertação* – ao supor a libertação de todos os recursos anteriores quando o próximo não puder ser atribuído,atrasa a execução do processo de modo muitas vezes substancial;

• *negação da condição de espera circular* – desaproveita recursos eventualmente disponíveis que poderiam ser usados na continuação do processo.

Sendo a prevenção no sentido estrito muito radical, recorre-se ás politicas de prevenção no sentido lato que se baseiam numa politica de monitorização de atribuição de recursos, tendo como base o número de recursos que cada um pode vir a precisar e os recursos que se encontram disponíveis).(???)

- As políticas de *prevenção de deadlock no sentido lato* baseiam-se na transição do sistema entre estados ditos *seguros*. O que é um estado seguro? Qual é o princípio que está subjacente a esta definição?

Define-se-se neste contexto *estado seguro* como uma qualquer distribuição dos recursos do sistema, livres ou atribuídos aos processos que coexistem, que possibilita a terminação de todos eles. Por oposição, um estado é *inseguro* se não for possível fazer-se uma tal afirmação sobre ele.
Convém notar o seguinte
• *é necessário o conhecimento completo de todos os recursos do sistema e cada processo tem que indicar à cabeça a lista de todos os recursos que vai precisar* – só assim se pode caracterizar um estado seguro;
• *um estado inseguro não é sinónimo de deadlock* – vai, contudo, considerar-se sempre o pior caso possível para garantir a sua não ocorrência.

- Que tipos de custos estão envolvidos na implementação das políticas de *prevenção de deadlock nos sentidos estrito e lato*? Descreva-os com detalhe.

  (???)