Questão 1 -

a) O invariante do semáforo é uma expressão booleana cujo valor se mantém sempre verdadeiro, após a inicialização do semáforo e antes e após qualquer operacão P ou V sobre o semáforo.
Seja Vo o valor inicial do semáforo (inteiro não negativo)
NV – o número de operacoes V() já completadas
NP – o número de operacoes P() já completadas (isto é, o processo invocou e conseguiu prosseguir a execução, de imediato, ou após ter estado temporariamente bloqueado, mas o número NP só conta as operações que já se completaram)

então o invariante é a expressao I <--> (Vo + NV >= NP)

O significado: Vo+NV-NP representa o valor corrente do semáforo a cada momento e este valor não pode ser negativo, pelo que Vo+NV-NP >= 0
Ou de outro modo, numa interpretacao operacional:
se formos contando os números NV e NP,
e se num dado momento um processo fizer um V(), o numero NV será incrementado e a expressao I actualizada;
se um processo tentar fazer um P()
se Vo+NV > NP – o processo não se bloqueia e podemos incrementar NP na expressao
senao (Vo+NV==NP) – o processo bloqueia-se até que um V() permita fazer o incremento de NP de forma a manter I verdadeira.

b) Semáforos genéricos segunda a definição de Dijkstra correspondem a semáforos que podem ter valores inteiros não negativos. Semáforos binários só admitem os valores 0 ou 1. Assim num semáforo binário, a tentativa de fazer duas operações V() sucessivas sem que haja pelo menos uma operação P() entre elas, origina um problema pois que a implementação não consegue registar o efeito do segundo V(). Teríamos de ter isso em conta na implementação que se pedia. Outra exigência óbvia era que as operações P e V fossem indivisíveis ou atómicas, para isso servia o semáforo mutex que se inicializava a1. Para o bloqueio dos processos, havia o semáforo block, inicializado a 0. Mas havia que cuidar para evitar possiveis deadlocks, em casos em que um processo, tendo invocado um P, tivesse de se bloquear por o valor do semáforo não ser positivo, e ao fazer Pb(block), originava um deadlock se antes do Pb() não fizesse Vb(mutex).

O pseudo.código de uma solução é:

```
P(semaforo: s);
{ Pb(mutex);
  s = s -1;
  if (s<0) {
        Vb(mutex);
        Pb(block);
     };
 Vb(mutex);
}
```

```
V(semaforo:s);
{ Pb(mutex);
  s = s +1;
  if (s<=0)  /*menor ou igual a 0 */
        Vb(block);
  else
        Vb(mutex);
  }
```

Note que, quando s menor ou igual a 0, ou seja há pelo menos 1 processo que se bloqueou ou vai bloquear dentro do P(), o processo que faz V(), ao fazer Vb(block) sem libertar o mutex (pois que nao faz Vb(mutex)) garante que, enquanto um dos que estão em P() não fizer um Pb(block), nenhum outro processo poderá executar outroVb(block). E assim garante a restricao de semáforo binário.

c) Variáveis de tipo condition versus semáforos
1- conditions só se podem declarar e usar dentro de monitores ou pelo menos dentro de secções críticas onde se tenha obtido um lock sobre um mutex antes. Versus semaforos que se podem usar em qualquer ponto do programa
2- conditons representam apenas uma fila de processos. Não assumem valor algum, excepto o estado da fila. Inicializada automaticamente pela implementacao, quando se declara uma instancia de uma variavel deste

tipo. Versus semaforos que representam um valor inteiro nao negativo sendo suportados por: (i) uma variavel que representa o valor do semaforo -e que tem de ser explicitamente inicializado quando se cria o semaforo; e (ii) por uma fila de processos associada ao semaforo.
3- Conditions sao manipuladas por duas operacoes atomicas wait(condition) e signal(condition)
Versus semaforos operados por duas operacoes atomicas: P(s) e V(s)
Wait(c) -- nao faz teste algum – bloqueia obrigatoriamente o processo invocador na fila da condition c. Por isso qualquer teste que deva ser feito tem de ser explicitado no programa para decidir fazer o wait ou não.
Para que esse teste seja consistente, pressupoe que o processo invocador obteve, antes de invocar Wait(), um lock sobre um mutex. Isto pode ser automaticamente garantido se a linguagem tiver declaracao de monitores, pois so´dentro do monitor se pode usar conditions. Ou entao se se tratar da biblioteca de Pthreads, o processo tem de ter antes obtido um lock explicitamente fazendo lock_mutex e nesse caso a operacao Wait leva dois argumentos wait(condition, mutex). A operacao Wait faz duas accoes de forma atomica: liberta o mutex que estava.na posse do invocador e bloqueia este na fila da condition.
Versus P(s) – bloqueia o processo só no caso de s não ser positivo e ao completar-se, a operacao tem o efeito de decrementar o valor de s em 1 unidade. A operacao P() nao nenhum efeito sobre algum mutex adicional porque pode ser chamada em qualquer ponto do programa. A operacao wait nao tem teste algum nem decremento de valor algum.
Signal (c) – desbloqueia um dos processos na fila de condition, mas é uma operacao nula se nao houver processo algum bloqueado. A implementacao garante sempre que o processo bloqueado que seja acordado, recupera automaticamente o lock do mutex acima referido, antes de retomar a execucao.
Versus V(s) – nunca é uma operacao nula. Acorda um processo, se houver algum bloqueado, ou incrementa o valor do semaforo. Nao tem garantia alguma face ao estado do processo que seja acordado, apenas de que este retomará a execucao completando a operacao P().

Questão 2 -
a) Sim, garante exclusão mútua. O programa só tem 2 processos e cada um deles teste uma condição antes de tentar entrar na sua região crítica. Esta condição é uma conjunção de duas componentes: só se ambas forem verdadeiras, é que o processo fica em ciclo de espera activa sem conseguir entrar. Basta que uma das componentes seja falsa, para que o respectivo processo possa entrar.
Por exemplo, em P1:

1a componente: teste de flag[1] – se esta for 0, significa que P2 está fora da região crítica e nem sequer ainda executou a acção 1, ou seja não está interessado em entrar. Então neste caso P1 entra, sem problemas. Note que se P1 entra, neste ponto (accao 3) é porque ele já antes colocou flag[0] a 1, pelo que esta componente, que é testada no código de P2, estará sempre verdade enquanto P1 estiver na regiao critica.

2a componente: teste de turn – note que P1, antes de chegar `a accao 3, executou as accoes 1 e 2. Ora, na accao 2, P1 colocou turn = 1. A ideia é P1 comecar, por uma questao de cortesia, digamos, a dar a prioridade a P2, caso P2 estiver interessado. De facto, se P2 já tiver posto flag[1] = 1 (accao 1 de P2) e se agora P1 puser turn = 1, verifica-se que P1 ficará em espera activa no seu ciclo 3. Entao em que caso 'e que turn pode ser 0, quando P1 chega à accao 3? Se turn estiver inicialmente a 0 entao P1 pode avancar, mesmo que P2 já tenha posto flag[1] =1. O mesmo sucede a P1, caso P2 já tenha executado a sua accao 2 pois ele aqui coloca turn = 0 , dando prioridade a P1, ou seja P1, mesmo que ficasse um pouco em espera activa em 3 (por testar flag[1] == 1 e turn == 1, esta posta por ele proprio), logo vai sair, quando P2 executar a accao 2, pondo turn = 0, o que torna a condicao 3 falsa no processo P1 e ao mesmo tempo coloca a condicao 3 verdadeira em P2, que ficaria em espera.
Argumentos simétricos podem aplicar-se a P2.
A conclusão é:
- nos cenários em que ambos os processos estejam a competir na entrada ou em que um deles já esteja na regiao criticca e o outro esteja a querer entrar, tem-se flag[0] == 1 e flag[1] == 1
- nos cenários em que ambos os processos estejam a competir na entrada ou em que um deles já esteja na regiao criticca e o outro esteja a querer entrar, tem-se turn == 0 ou turn == 1, mas nunca se pode ter ambas.
- nos cenários em que ambos os processos estejam a competir na entrada ou em que um deles já esteja na regiao criticca e o outro esteja a querer entrar, tem-se (admitindo X <-> (turn ==1):

3 em P1: (flag[1] == 1) && (turn == 1) <-> (VERDADE) && (X)
3 em P2: (flag[0] == 1) && (turn == 0) <-> (VERDADE) && (NÃO X)

Nesse caso, se X == VERDADE ou seja turn == 1, então:

3 em P1 <-> VERDADE e P1 fica em espera activa
3 em P2 <-> FALSA e P2 avança

ou vice-versa: se X == FALSA ou seja turn == 0, então:

3 em P1 <-> FALSA e P1 avança
3 em P2 <-> VERDADE e P2 fica em espera activa

Donde: só um dos processos consegue avancar para alem do ponto 3 e nunca ambos. Daí a garantia de exclusão mutua.

b) A resposta é sim: garante que nao ocorrem situacoes de impasse: tal ocorreria se ambos os processos pudessem ficar eternamente bloqueados nos respectivos ciclos 3, quando ninguem estivesse na regiao critica. Note que só há 2 processos. Assim, tal nunca pode acontecer, pois a variavel turn garante a assimetria, ou seja há sempre um que encontra a condicao 3 a falso e avanca.

c) A resposta é sim: evita situacoes de injustica: o algoritmo seria injusto e poderia prejudicar um dos processos, se houvessem cenários em que, estando por exemplo P2 a tentar entrar (portanto tendo posto flag[1] =1 e turn = 0, nas accoes 1 e 2), ficasse eternamente no ciclo 3 de espera por a condicao (flag[0] == 1) && (turn == 0) ser eternamente verdadeira. Mas tal nao pode acontecer.
Admita que nesse caso (com P2 em espera em 3), P1 estava na regiao critica (com (flag[0] == 1): logo que P1 sai, vai colocar flag[0] == 0 (accao 5). Entao, se nesse ponto, P2 conseguir de imediato reavaliar a condicao 3, verifica que esta é agora FALSA e P2 entraria logo. Mas pode acontecer que, por causa do escalonamento de processos do SO, que P2 nao tenha oportunidade de reavaliar 3, antes de P1 voltar ao inicio do seu ciclo e tentar entrar de novo. Nesse caso, P1 poria de novo (flag[0] == 1) e P2 nao conseguiria entrar. No entanto, logo a seguir P1 executa a accao 2, pondo turn =1 e segue para 3 ficando em espera, pois (flag[1] == 1) && (turn == 1). Eventualmente, num monoprocessador P1 esgotaria o seu intervalo de TIME-SLICE e o SO daria a oportunidade a P2 para reavaliar a condicao 3, agora com (flag[0] == 1) && (turn == 1), e P2 avancaria.
Assim, uma situacao de espera eterna nunca pode acontecer (admitindo que as regioes criticas têm duracao limitada).

Questao 3 - sugestão de resolução – há outras soluções possiveis

Declaração e inicialização de Semaforos: neste caso são todos binários:

```
semaforo mutex = 1; /* exclusao mutua no acesso ao bloco*/

semaforo Scheia = 1;
/*Scheia indica zona do Bloco ocupada: inicialmente livre=1 para o processo P1 pôr*/

semaforo Svazia[2..N] = 0;
/*vector de semaforos,  para cada Pi:2..N aguardar, inicialmente 0 indicam zona vazia;
 o processo P1 faz N-1 operacoes V sobre este vector, após ter posto um bloco*/
```

```
porBloco (char *buf);
/*no Processo i=1
{P(Scheia);
 P(mutex);
   colocar_bloco(buf);
 V(mutex);
 Para i = 2 até N
    V(Svazia[i]);
}
```

```
lerBloco(char *buf);
/*em cada Processo i:2..N*/
{ P(Svazia[i]);
 P(mutex); /*ver nota2 abaixo*/
    obterBloco(buf);
 V(mutex);
 barreira(); /*barreira de N-1 processos – ver nota1 abaixo*/
 if (Pi == 2) V(Scheia); /*admitir Pi = identifica o processo;*/
          /* qualquer um podia fazer este V()
         serve para avisar P1 de que já todos Pi:2..N leram*/
```

Nota1:
Acrescentar a implementação da função barreira(); {...como dado nas aulas...}. Ë necessario que todos Pi:2..N chamem a barreira para confirmar que já todos leram um bloco antes

de avisar P1 e de os Pi:2..N voltarem a tentar ler o próximo bloco.

Nota2:
Na verdade as operacoes colocar/obterBloco() neste problema poderiam ser feitas sem terem de estar protegidas pelo P(mutex) e V(mutex) porque os processos Pi:2..N só estão a ler o bloco, e entretanto temos a garantia de que o processo P1 não pode tentar escrever pois este, antes de ter posto o bloco, teve de fazer P(Scheia) que deixou este semáfaro a 0, pelo que P1 não pode voltar a escrever enquanto não avisado por um V sobre Scheia, o qual P2 só faz após todos terem lido (após passada a barreira). Portanto o pseudo-código poderia ser muito simplificado, eliminando o mutex.

Nota3:
Em vez do vector Svazia pode ter-se um único semáforo inicializado a 0 e
´´P(Svazia[i]);´´ seria substituido por ´´P(Svazia);´´ em lerBloco
e em porBloco ficaria ´´Para (i = 2 até N) V(Svazia); ´´