Parte A (12 valores)

1. O diagrama abaixo procura ilustrar o papel desempenhado pelo sistema de operação no âmbito da funcionalidade geral apresentada por um sistema computacional. (3 valores)

   Neste contexto, explique
   a) que papel desempenha o sistema de operação na perspectiva do utilizador (*top-down*).
   b) que papel desempenha o sistema de operação na perspectiva do construtor (*bottom-up*).
   c) que papel desempenha o *ambiente de interacção com o utilizador*, entendido como a aplicação de carácter mais básico que está habitualmente presente.

2. Distinga *programa* de *processo*. Que propriedades são corporizadas pelo conceito de processo? Mostre como é que um mesmo programa pode dar origem a múltiplos processos. (3 valores)

3. Num ambiente de multiprogramação, a atribuição do processador aos diferentes processos que coexistem é multiplexada no tempo. Designa-se de política de *scheduling* do processador o modo como esta multiplexagem é operacionalizada. (3 valores)

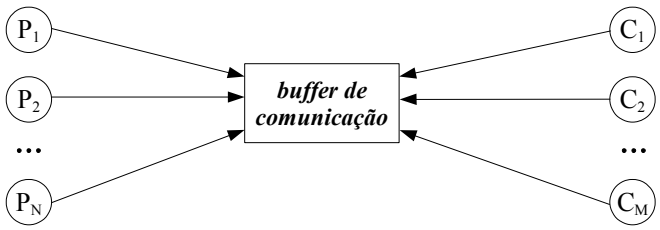
   Neste contexto, responda às questões seguintes.
   a) Distinga *preemptive scheduling* de *non-preemptive scheduling*.
   b) Construa um diagrama que descreva o *scheduling* de baixo nível, também designado de *gestão do processador*. Tenha em conta que deve identificar <u>todos</u> os estados e todas as transições que incluir e explicar o seu significado.
   c) Distinga políticas de prioridade *estática* de políticas de prioridade *dinâmica*. Dê um exemplo de cada uma.

4. A memória de massa de um sistema computacional é habitualmente dedicada à instalação de *sistemas de ficheiros* e da área de *swapping*. (3 valores)

   Neste contexto, responda às questões seguintes.
   a) Dê exemplos de três tipos de dispositivos que são comummente usados para implementar a memória de massa.
   b) Qual é o papel desempenhado pela área de *swapping* numa organização de memória real?
   c) O que são *sistemas de ficheiros*?

Parte B (8 valores)

A figura abaixo apresenta o diagrama de interacção de um grupo de processos cooperantes.



Os processos P_i, com $i = 1, \cdots, N$, designam-se de *produtores* e os processos C_i, com $i = 1, \cdots, M$, de *consumidores*. Os primeiros produzem informação que será consumida pelos segundos. A sincronização entre eles é obtida aqui por recurso a semáforos.

Admita que os semáforos são implementados pelo tipo de dados

```
typedef struct
        { unsigned int val;              /* estado do semáforo: verde
                                    em diferentes tonalidades ou vermelho */
          FIFO queue;       /* fila de espera dos processos bloqueados */
        } SEMAPHORE;
```

e que as operações para sua manipulação são

```
unsigned int semCreate (void);
void semDestroy (unsigned int semId);
void semDown (unsigned int semId);
void semUp (unsigned int semId);  .
```

A invocação de `semCreate` solicita ao sistema de operação a atribuição de um semáforo; a identificação do semáforo é devolvida. Após a realização da operação, o semáforo associado tem o campo `val` com o valor zero (está, portanto, no estado *vermelho*). As restantes operações são invocadas passando como parâmetro esta identificação. As invocações de `semDown` e `semUp` correspondem à invocação das operações convencionais de *down* e *up* sobre o dispositivo. Finalmente, a invocação de `semDestroy` alerta o sistema de operação que o semáforo em causa já não é necessário e pode ser libertado.

O *buffer* de comunicação, por seu lado, é implementado por uma memória com uma capacidade de armazenamento de *K* itens de informação e supõe as operações seguintes para a sua manipulação

```
void putVal (COMMBUFF *pBuff, INFO val);
void getVal (COMMBUFF *pBuff, INFO *pVal);  .
```

A invocação de `putVal` insere no *buffer* de comunicação referenciado por `pBuff` o item de informação passado em `val`. A invocação de `getVal` retira do *buffer* de comunicação referenciado por `pBuff` um item de informação e armazena-o na região de memória referenciada por `pVal`. Naturalmente, se o *buffer* de comunicação estiver *cheio*, a invocação de `putVal` conduz a erro; o mesmo se passa se `getVal` for invocado quando o *buffer* de comunicação estiver *vazio*.

Para que os processos intervenientes cooperem sem erros, é necessário garantir que o acesso à região crítica seja efectuado em regime de exclusão mútua e que haja sincronização entre eles.

Apresenta-se a seguir o código dos processos *produtores* e dos processos *consumidores* no dialecto de linguagem C usado nas aulas. O código supõe que os semáforos usados foram previamente criados e inicializados.

Processos produtores

```
shared COMMBUFF buff;                          /* buffer de comunicação */
shared unsigned int access; /* identificação do semáforo que garante o
                    acesso à região crítica em regime de exclusão mútua */
shared unsigned int bFull;              /* identificação do semáforo de
                     bloqueio quando o buffer de comunicação está cheio */
shared unsigned int bEmpty;             /* identificação do semáforo de
                     bloqueio quando o buffer de comunicação está vazio */

void main (void)
{ INFO val;
  forever
  { produceVal (&val);
    insertValOnCommBuff (&buff, val);
    doSomethingElse ();
  }
}
```

Processos consumidores

```
shared COMMBUFF buff;                          /* buffer de comunicação */
shared unsigned int access; /* identificação do semáforo que garante o
                    acesso à região crítica em regime de exclusão mútua */
shared unsigned int bFull;              /* identificação do semáforo de
                     bloqueio quando o buffer de comunicação está cheio */
shared unsigned int bEmpty;             /* identificação do semáforo de
                     bloqueio quando o buffer de comunicação está vazio */

void main (void)
{ INFO val;
  forever
  { retrieveValFromCommBuff (&buff, &val);
    consumeVal (val);
    doSomethingElse ();
  }
}
```

1. Em que consiste a região partilhada e o que é a região crítica neste caso? Justifique claramente a sua resposta. (2 valores)
2. Que processos bloqueiam, respectivamente, nos semáforos `bFull` e `bEmpty` e em que condições? Justifique claramente a sua resposta. (2 valores)
3. Que valor assume o campo `val` de cada um dos semáforos quando se efectua a sua inicialização? Justifique claramente a sua resposta. (2 valores)
4. Construa as primitivas `insertValOnCommBuff` e `retrieveValFromCommBuff`. (2 valores)