Instituto Superior de Engenharia de Lisboa

Programação II

2021/22 – 2.º semestre letivo

Teste de época especial (2022.09.06)

Nota importante: Valoriza-se a escrita de código que inclua comentários esclarecedores da implementação seguida e que contenha indentação legível.

1. [5,0 valores]

Pretende-se fazer o registo de processos a executar pelo processador. Para tal será usado o tipo **`Proc`** que irá conter a seguinte informação:

```
typedef struct {
  char app[MAX_NAME]; // nome da aplicação (Ex: "excel", "winword", "calc")
  char *args;   // argumentos a enviar à aplicação (Ex: "file.xls") ou NULL
  char priority;// '+' (High) ou '-' (Low)
  int  level;   // valor entre 1..5 com o nível do processo
                // dentro da mesma prioridade
  char *cmd;    // linha de comando a executar
} Proc;
```

Pretende-se representar uma lista para armazenar um conjunto de processos, suportada pela seguinte definição:

```
typedef struct {
  int count;          // N.º de elementos efetivamente na lista
  Proc arr[N_PROCS]; // Array com N_PROCS elementos criados estaticamente
} ProcList;
```

a) [2,0] Escreva a função

```
void fill_proc(Proc *proc, char *app, char *args, char pri, int lev);
```

que preenche uma estrutura já existente com os valores enviados à função:

- **`proc`** – referência para a estrutura (já existente) a preencher;
- **`app`** – *string* com o nome do processo a executar (Ex: "`mycopy`");
- **`args`** – *string*, a alojar dinamicamente, com os argumentos a enviar à aplicação ou NULL, caso não existam parâmetros para a app;
  (Exemplo: "`filein.txt fileout.txt`")
- **`pri`** – prioridade com que o processo deve ser executado '+' (*High*) ou '-' (*Low*);
- **`lev`** – nível associado ao processo (1 .. 5).

O campo **`cmd`** deve referenciar uma *string*, a alojar dinamicamente, com o comando a executar no sistema operativo.

Exemplos de comandos:

- "`mycopy +3 filein.txt fileout.txt`" [pri = '+', level = 3]
- "`calculator -5`" [pri = '-', level = 5, sem args]

b) [1,0] Escreva a função

```
int compare_proc(const void *p1, const void *p2);
```

que compara dois processos, retornando um valor negativo, zero ou um valor positivo, utilizável na função `qsort` para ordenar o *array* `arr` de `ProcList`, de modo que os processos a executar antes fiquem nos índices mais baixos. Os critérios de ordenação são os seguintes:

1. Todos os processos com `priority` '+' são executados antes dos processos com `priority` '-';
2. Com o mesmo sinal de prioridade, são executados antes os que têm valor de `level` mais baixo;
3. No caso de empate em `priority` e `level`, são executados antes os que têm a linha de comando a executar alfabeticamente mais baixa.

O exemplo da alínea seguinte ilustra também a ordem pretendida.

c) [1,0] Escreva a função

```
void print_proc_list(ProcList *list, int ord);
```

Que mostra no ecrã a lista de processos existentes em `list`. Caso `ord` represente o valor **verdade**, a lista deverá ser previamente ordenada usando obrigatoriamente a função `qsort` da biblioteca *standard* da linguagem.

Exemplo de possível output com uma lista de 10 processos:

```
0: calc +1
1: excel +3 dados.xls
2: zzz +4
3: excel +5 dados.xls
4: mycopy +5 filein.txt fileout.txt
5: calc -1
6: mycopy -1 filein.txt fileout.txt
7: calc -3
8: excel -3 dados.xls
9: mycopy -5 filein.txt fileout.txt
```

d) [1,0] Escreva a função

```
Proc *find_proc(ProcList *list, char *app, char *args, char pri, int lev);
```

que retorna uma referência para o processo existente na **lista ordenada** de processos `list`, ou NULL, caso não exista. Para tal deverá recorrer obrigatoriamente à função `bsearch` da biblioteca *standard* da linguagem.

2. [5,0 valores]

Considere a organização do *array* da figura seguinte. No exemplo ilustrado por esta figura o *array* contém quatro elementos. Cada elemento contém um campo "U" e um campo "Data". O campo "U" tem a dimensão de um *byte* e o campo "Data" tem uma dimensão adequada à aplicação em causa, sendo dimensionado pelo utilizador (todos os campos "Data" do *array* têm a mesma dimensão). Os campos "U" indicam se o respetivo campo "Data" contém dados (pertinentes para a aplicação em causa), caso em que o valor de "U" deverá ser *true* (em linguagem C). Caso contrário, se "U" for *false*, os *bytes* que compõem o campo `Data` são considerados como não contendo informação (pertinente para a aplicação em causa).

U	Data	U	Data	U	Data	U	Data

Considere as definições das funções `insert_elem` e `delete_elem`, que estão incompletas.

```
1   void * insert_elem(void * a, int n_elem, void * elem, size_t el_size) {
2     void * b = realloc(...);
3     if (!b) return NULL;
4     char * t = ... ;
5     char * s = ... ;
6     *t++ = ... ;          // Marcação de que este elemento contém dados.
7     while (el_size--)
8       *t++ = *s++;        //Cópia, byte a byte, do novo elemento para
9                           //o novo espaço do array.
10    return ... ;
11  }
12  void delete_elem(void * a, int del_index, size_t el_size){
13    ... = 0;
14  }
```

a) [1,0] Reescreva as definições destas funções completando-as nos troços com reticências e comentando-as com clareza. Os parâmetros da função `insert_elem` são, respetivamente: um ponteiro para o *array* já existente (ou `NULL` caso o *array* ainda não exista), o número de elementos que já existem no *array* (no exemplo anterior seriam quatro), um ponteiro para o elemento a inserir (apenas o novo campo `Data`, a inserir no fim do *array*) e a dimensão dos campos `Data`. Os parâmetros da função `delete_elem` são, respetivamente: o endereço inicial do *array* já existente (é da responsabilidade do utilizador não chamar esta função com este parâmetro a `NULL`), o índice do elemento a marcar como não contendo dados (elemento "apagado"), e a dimensão dos campos `Data`.

Considere o programa seguinte, inscrito num ficheiro de nome **`mod1.c`**, no qual se usam as funções `insert_elem` e `delete_elem`. Admitindo que as linhas que contêm reticências estão vazias, qual o *output* gerado? Justifique com clareza.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define MSG_SIZE 10
4  void *  insert_elem(void * a, int n_elem, void * elem, size_t el_size);
5  void delete_elem(void * a, int del_index, size_t el_size);
6  typedef struct data {
7    int i;
8    char c;
9    char s[MSG_SIZE];
10 } Data;
11 void print_array(void *a,int n_elem,size_t el_size,void(*print_func)(void *)){
12   int i;
13   char * p = (char *)a;
14   for(i=0; i<n_elem; i++, p+=1+el_size)
15     if(*p)
16       print_func(p + 1);
17     else
18       printf("---\n");
19   printf("\n");
20 }
21 void print_elem(Data * p){
22   printf("%d, %c, %s\n", p->i, p->c, p->s);
23 }
24 int main() {
25   int n_elem = 0;
26   Data d0 = {3, 'A', "Ana"}; Data d1 = {6, 'C', "Carlos"}; Data d2 =
27   {5, 'D', "David"}; Data d3 = {4, 'E', "Elsa"}; Data d4 = {7, 'F', "Filinto"};
28   void * a = NULL;
29   a = insert_elem(a, n_elem++, &d0, sizeof (Data));
30   a = insert_elem(a, n_elem++, &d1, sizeof (Data));
31   a = insert_elem(a, n_elem++, &d2, sizeof (Data));
32   a = insert_elem(a, n_elem++, &d3, sizeof (Data));
33   a = insert_elem(a, n_elem++, &d4, sizeof (Data));
34   print_array(a, n_elem, sizeof (Data), print_elem);
35   delete_elem(a, 0, sizeof(Data));
36   delete_elem(a, 2, sizeof(Data));
37   delete_elem(a, 4, sizeof(Data));
38   print_array(a, n_elem, sizeof (Data), print_elem);
39   ...
40   ...
41   return 0;
42 }
```

b) [1,0] Descreva a funcionalidade das funções `print_array` e `print_elem`.

c) [2,0] Após a execução do exemplo anterior, o *array* "a" contém alguns elementos "apagados", ou seja, cujo respetivo *byte* "U" tem valor `false`. Se se souber que aqueles elementos não serão doravante necessários na aplicação em causa, pode-se "compactar" o *array*, libertando alguma memória ocupada pelo atual *array*. Realize a função `compress` que executa esta "compactação", escolhendo o tipo de retorno da função e os seus parâmetros. A função `compress` deverá libertar toda a memória não utilizada no *array*, mantendo a ordem relativa dos elementos. Indique como usaria esta função na linha 39 do troço de código anterior como se indica na caixa seguinte, substituindo nesta as reticências pelo que entender necessário.

```
39    ... compress( ... );
40    print_array(a, n_elem, sizeof (Data), print_elem);
```

O *output* gerado pela função `print_array` chamada na linha 40 deverá ser o seguinte:

```
6, C, Carlos
4, E, Elsa
```

Sobre a realização da função `compress`:

i) Descreva num troço de texto, com clareza e sem usar linguagem C, o algoritmo que vai utilizar na realização da função `compress`. Não utilize fluxogramas (mas pode usar pseudocódigo se considerar útil).

ii) Depois de realizar a parte i) desta alínea, escreva a definição da função `compress`, seguindo estritamente a descrição que fez do algoritmo utilizado. Escreva o código desta função com uma indentação correta, usando, se necessário, comentários para complementar a descrição anterior. Nota: desaconselha-se veementemente a escrita da definição da função `compress` pedida nesta parte ii) sem a realização da parte i) porque penalizará significativamente a avaliação de toda a alínea c) deste grupo.

d) [1,0] Admita que todo o código acima referido consta num ficheiro `mod1.c`. Ao executar o comando

```
gcc -c -g mod1.c
```

são geradas mensagens de aviso (*warnings*). Porquê? Mostre o que se deveria alterar naquele código para evitar a geração daquelas mensagens.

Qual a utilidade das opções `-c` e `-g` ?

3. [4,0 valores]

Pretende-se representar os dados de um conjunto de pessoas, pertencentes a uma organização, em listas ligadas baseadas nos tipos seguintes:

```
typedef{
    int number;  // Número mecanográfico da pessoa, na organização
    char *name;  // ponteiro para o nome, em string alojada dinamicamente
} Person;

typedef struct lNode{
    struct lNode *next; // ponteiro de ligação na lista
    Person *data;       // ponteiro para os dados da pessoa
} LNode;
```

Existe uma lista principal, construída na fase inicial do programa, a partir de dados gravados num ficheiro, inserindo sucessivamente à cabeça da lista ligada, pelo que a ordem da lista depende da ordem no ficheiro.

a) [1,0] Escreva a função

```
int listVerifyOrder( LNode *head );
```

que verifica se a lista indicada por head está ordenada por valor crescente do campo `number` dos dados associados aos nós. Se a verificação for bem sucedida, retorna 1; caso contrário, retorna 0.

b) [1,0] Escreva a função

```
void listInvert( LNode **headPtr );
```

que inverte a ordem dos elementos na lista ligada cujo ponteiro cabeça é indicado por `headPtr`.

c) [2,0] Escreva a função

```
LNode *listBuildRef( LNode *head );
```

Que recebe o ponteiro cabeça da lista principal, em `head`, e cria uma nova lista ligada cujos nós apontam, através do campo `data`, para o mesmos elementos `Person` da lista principal, indicada por. A nova lista é ordenada, alfabeticamente crescente, pelo campo `name` dos dados referenciados. A função retorna o ponteiro cabeça da nova lista ligada. A lista principal permanece sem alterações. Se necessitar de funções auxiliares, escreva-as também.

4. [3,0 valores]

Considere uma árvore binária de pesquisa com dados genéricos, formada por nós com o tipo **`TNode`**, e a função **`tAdd`** para adicionar dados.

```
typedef struct tNode{
    struct tNode *left, *right;  // ponteiros de ligação na árvore
    void *data;                  // ponteiro para o bloco de dados
} TNode;

void tAdd( TNode **rp, void *d, int (*comp)( void *d1, void *d2 ),
           void * (*new)( void *d ) ){
   if( *rp == NULL ){
      TNode *n = malloc( sizeof *n );
      n->data = new( d );
      n->left = n->right = NULL;
      *rp = n;
      return;
   }
   int cmp = comp( d, (*rp)->data);
   if( cmp == 0 )
      return;
   if( cmp < 0 )
      tAdd( &(*rp)->left, d, comp);
   else
      tAdd( &(*rp)->right, d, comp);
}
```

a) [1,0] Escreva a função

```
void * tSearch( TNode *r, void *d, int (*comp)( void *d1, void *d2 ) );
```

que pesquisa, na árvore com raiz `r`, os dados indicados por `d` e retorna o ponteiro para o bloco de dados encontrado ou NULL, no caso de não existir.

Considere a função e **`tSearchAdd`** seguinte, também para pesquisar um bloco de dados, mas que o adiciona se este ainda não existir.

```
void * tSearchAdd( TNode **rp, void *d, int (*comp)( void *d1, void *d2 ),
                   void * (*new)( void *d ) ){
   void *res = tSearch( *rp, d, comp);
   if( res == NULL ){
      tAdd( rp, d, comp, new );
      res = tSearch( *rp, d, comp);
   }
   return res;
}
```

b) [1,0] Admita que a função `tSearchAdd` é chamada recebendo em `rp` o endereço de um ponteiro raiz de uma árvore com 7 nós, perfeitamente balanceada, e em `d` um bloco de dados que não existe na árvore. Tendo em conta a chamadas às outras funções e os seus algoritmos, determine e justifique a quantidade total de vezes que é chamada a função de comparação, indicada pelo parâmetro `comp`.

c) [1,0] Escreva uma nova versão, mais eficiente, da função anterior, agora com o nome `tSearchAdd2`, a qual executa as operações equivalentes sem usar as funções `tSearch` e `tAdd` chamando a função de comparação (`comp`) a quantidade de vezes estritamente necessária.

5. [3,0 valores]

Considere um conjunto de módulos de programa fonte, escritos em linguagem C, dos quais foram produzidos os módulos compilados com o comando

```
gcc -c *.c
```

A caixa ao lado contém as listas de símbolos dos módulos referidos e a respetiva classificação: **`T`** (*public text*), **`t`** (*private text*) ou **`U`** (*undefined*), em resultado do comando

```
nm *.o
```

a) [1,0] Note que há funções com o nome "**`aux`**" em vários módulos.
Indique se os módulos que contêm a função "**`aux`**" podem fazer parte, em simultâneo, do conjunto de módulos para produzir um executável. Justifique.

b) [1,0] Suponha que é desenvolvido um módulo de aplicação com o nome "**`aplic.c`**", o qual, além de usar de usar funções de biblioteca, chama apenas as funções **`setEmpty`**, **`setInsert`** e **`setVerify`**.
Indique, justificando, quais são os módulos fonte (**`.c`**) estritamente necessários para produzir o executável da aplicação.
Escreva uma sequência de comandos para compilar estes módulos e para os ligar, produzindo o executável "**`aplic`**".

c) [1,0] Admita que existe um *header file* "**`set.h`**", contendo todas as definições e assinaturas relativas aos módulos referidos, nos quais é incluído.
Escreva um *makefile* para gerar, de forma eficiente, o executável "**`aplic`**" a partir dos módulos necessários.

```
setcomplem.o:
0000000000000000 T setComplement

setempty.o:
0000000000000000 T setEmpty

setinsert.o:
0000000000000000 t aux
000000000000003b T setInsert
                 U setReunion
                 U setSingle

setintersect.o:
0000000000000000 T setIntersect

setprint.o:
                 U printf
0000000000000000 t aux
0000000000000028 T setPrint
                 U setVerify

setremove.o:
0000000000000000 t aux
                 U setComplement
                 U setIntersect
0000000000000047 T setRemove
                 U setSingle

setreunion.o:
0000000000000000 T setReunion

setsingle.o:
0000000000000000 T setSingle

setverify.o:
                 U setEmpty
                 U setIntersect
                 U setSingle
0000000000000000 T setVerify
```