Université Paris Diderot – Paris 7
L3 Informatique, C et Systèmes.
21 décembre 2017
Durée : 3 heures.

Documents autorisés :

(1) deux pages de notes personnelles (manuscrits ou imprimés) format A3 (ou A4), chaque page doit porter votre nom,
(2) le poly « fichiers »,
(3) le poly « processus »,
(4) le poly avec les fonctions de la bibliothèque de fonctions C standard.

Tous les autres documents interdits.
Le sujet comporte 6 pages.

Votre code doit être écrit de façon **lisible**, *avec des indentations et des accolades appropriées permettant de voir la fin de blocs de code (fin de boucles, etc.). Vous pouvez, si vous le trouvez utile, d'écrire de fonctions auxiliaires.*

Il est inutile d'écrire les includes. Par contre une gestion d'erreurs, même minimaliste, sera appréciée (mais ce n'est pas le point très important).

Le barème est donné à titre indicatif.

QUESTIONS

Question 1: [2 points] Qu'est-ce que affiche chaque `printf` dans le programme suivant ?

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stddef.h>
4
5  int main(void){
6    char *s = "abcdefghijkl";
7    char *d;
8    int i,j;
9    i=3;
10   printf("A=%c\n", *(s+i) );
11   printf("B=%c\n", s[i++] );
12   j=5;
13   printf("C=%c\n", s[++j] );
14   printf("longueur s=%lu\n", (unsigned long)strlen(s) );
15   d=s+7;
16   printf("longueur d=%lu\n", (unsigned long)strlen(d) );
17   printf("D=%c\n", d[-2] );
18
19   int tab[]={21,33,45,57,69,70,80,90,100};
20   ptrdiff_t g = &tab[7] - &tab[1];
21   printf("g=%ld\n", (long) g);
22
23
24   int *pi = &tab[7];
25   printf("expr = %d\n",  *(pi-5) + 5 );
26   return 1;
27 }
```

Question 2: [1 point] Soit `liste` le type qui implémente une liste simplement chaînée (sans tête) qui stocke des entiers,

```
struct elem{
  int data;
  struct elem *suivant;
};
typedef struct elem *liste;
```

Le fragment du code suivant est sensé de calculer la somme d'éléments de la liste `l`.

```
liste l;
.........
int somme;
liste p;
for(somme = ??? , p = ??? ; p != ??? ; somme += ??? , p = ???)
    ;
```

On suppose que

- à la fin de la boucle `for` la variable `somme` doit contenir la somme,
- la variable `p` sert à parcourir la liste,
- avant la boucle les variables `somme` et `p` ne sont pas encore initialisées.

En remplaçant les occurrences de `???` dans le code, écrire le code qui réalise la tâche demandée. (Le corps de la boucle est vide et il doit rester vide. Si votre solution n'est pas conforme à cette consigne vous aurez sensiblement moins de points)

Chaînes de caractères

Exercice 1 [2.5 points] Écrire une fonction

```
char *melange(char *x, char *y)
```

qui produit une chaîne de caractères qui est un mélange de deux chaînes `x` et `y`.

Un mélange de deux deux chaînes de caractères `x` et `y` est une chaîne de caractères qui

- alterne les caractères de `x` et de `y` et
- commence par le premier caractère de `x`,
- à la fin, si une chaîne est plus longue que l'autre alors on recopie les caractères de la chaîne plus longues sans mélange.

Par exemple si `x = "abcd"` et `y = "GHIJKL"` alors le résultat de `melange(x, y)` est la chaîne `"aGbHcIdJKL"`, par contre le résultat `melange(y, x)` est `"GaHbIcJdKL"`.

En cas de problèmes (`malloc` qui échoue) la fonction `melange` retournera `NULL`.

Systèmes

Exercice 2 [4 points] Écrire une fonction

```
int copie_massive(char *source[], char *dest[], size_t n)
```

où `source` et `dest` sont deux tableau de chaînes de caractères de même taille `n`. On suppose que les deux tableaux contiennent des noms de fichiers. La fonction doit copier chaque fichier `source[i]` dans `dest[i]`, $i = 0, \ldots, n-1$. Pour effectuer chaque copie la fonction doit créer un processus enfant (un enfant distinct pour chaque l'opération de copie).

Chaque enfant doit copier un fichier en exécutant, à l'aide d'une fonction de la famille `exec`, la commande UNIX `/bin/cp source destination` avec les arguments appropriés.

Après avoir lancé tous les processus enfants la fonction attendra leurs terminaison.

On dira qu'un enfant a terminé correctement si et seulement si il a terminer par `exit(0)` (ou `return 0;` dans `main`). Si l'enfant a terminé tué par un signal ou avec la valeur `exit` différente de 0 alors on dira que l'enfant a échoué.

La fonction doit retourner 1 si tous les processus enfants ont terminé correctement, 0 s'il y a un ou plusieurs enfants qui ont échoué (mais tous les enfants ont été créés sans erreur de `fork`), et -1 si un des `fork()` a échoué.

Exercice 3 [2.5 points] Écrire une fonction

```
int ajouter(const char *fichier, unsigned int i, int c)
```

On suppose que `fichier` est un nom d'un fichier ordinaire contenant des entiers de type `int`. La fonction ajoutera la valeur `c` au $i$ème entier dans le fichier. On suppose que les entiers dans le fichier sont numérotés à partir de 0.

Par exemple, si un fichier `"toto"` contient quatre entiers `3 7 -4 8`, alors après

```
ajouter("toto", 0, -5);
ajouter("toto", 2, 9);
```

le contenu du fichier `"toto"` est `-2 7 5 8`. Dans les deux cas la fonction retournera 1.

L'appel `ajouter("toto", 10, 11)` où on demande d'ajouter 11 à 10ème entiers et le fichier `"toto"` n'a pas autant d'entiers doit échouer et la fonction retournera -1.

Dans le fichier, les entiers sont stockés en tant que ~~les~~ entiers de type `int`, chaque entier occupant `sizeof(int)` octets, l'un immédiatement après l'autre sans aucun octet de séparation entre deux entiers consécutifs.

Il est possible de faire cette exercice avec les fonctions de bas niveau ou avec les fonctions de C standard. Les deux solutions seront acceptées.

Problème : FIFO

Le but des exercices dans cette section est d'implémenter une file (FIFO). Vous pouvez faire ces exercices dans n'importe quel ordre.

Rappelons qu'une file est une suite d'éléments du même type telle que les éléments sont retirés de la file dans l'ordre d'arrivée (first in first out). La file `fifo` sera implémentée à l'aide d'un tableau qui contient les éléments de la file et d'une structure auxiliaire qui permet d'implémenter la gestion de la file. La structure auxiliaire et la file (fifo) sont définies comme :

```
struct file{
  char *first;      /*pointeur premier element*/
  char *last;       /*pointeur apres dernier element*/
  size_t te;        /*taille d'un element*/
  char *occupe;     /*pointeur premier element*/
  char *libre;      /*pointeur vers le premier element libre*/
};

typedef struct file *fifo;
```

La structure `struct file` possède les champs suivants :

(1) `first` est le pointeur vers le premier élément du tableau dans lequel on stockera les éléments de la file.

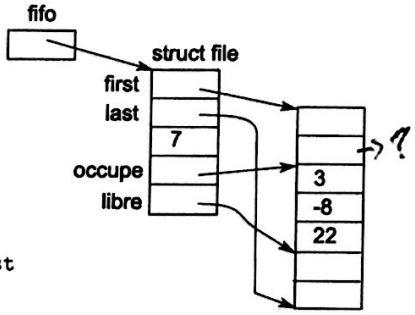
(2) `last` est le pointeur vers le premier octet juste **après** le dernier élément du tableau, ce pointeur sert à marquer la fin du tableau.

(3) `te` est la taille d'un élément en octets. Notre but est d'implémenter une pile générique, la taille d'un élément sera fixée à la création de la pile.

(4) `occupe` est l'adresse du premier élément dans la file, c'est l'adresse de l'élément qui sera retiré de la file par la prochaine opération `get`.

(5) `libre` le pointeur vers l'octet juste après le dernier élément de la file, la prochaine opération `put` mettra un nouveau élément à cette adresse.

Le fait que tous les pointeurs dans `struct file` sont de type `char *` ne veut pas dire que la file sert à stocker des caractères, on pourra stocker des éléments de n'importe quel type à condition que tous les éléments ont la même taille `te` en octets. Nous avons choisi d'utiliser `char *` au lieu de pointeur générique `void *` uniquement parce qu'il est impossible de faire l'arithmétique de pointeurs avec les pointeurs génériques. Et faire chaque fois un *cast* quand on veut faire une arithmétique de pointeurs devient vite très pénible.



La figure ci-contre représente une file de nombres entier. Dans la file il y a actuellement trois éléments : $3, -8, 22$. Et la capacité maximale de la file est ~~de 7 éléments~~. La prochaine opération `get` obtiendra le premier élément 3 qui sera retirer de la file. La prochaine opération `put` mettra un nouveau élément après 22.

A chaque moment les pointeurs utilisés doivent satisfaire la condition suivante :

```
first <= occupe <= libre <= last
```

De cette description il découle que :

(a) `occupe == libre` si et seulement si il n'y a aucun élément dans la file,

(b) `( occupe - libre )` divisé par `te` donne le nombre d'éléments actuellement dans la file,

(c) `( last - first )` divisé par `te` donne le nombre maximal d'éléments que nous pouvons stocker dans la file quand toutes les places sont occupées,

(d) `libre == last` s'il n'y a plus de place pour le nouveau élément (l'adresse `libre` n'est plus une adresse valide d'un élément du tableau, il faudra réorganiser la file pour pouvoir insérer un nouveau élément).

Exercice 4 [1.5 point] Écrire la fonction

```
fifo create_fifo( size_t capacite_init, size_t taille_elem)
```

qui crée une file vide (sans éléments), i.e. crée la structure et le tableau. Le paramètre `capacite_init` c'est la taille initiale du tableau qui stocke les éléments de la file (la taille mesurée en nombre d'éléments et non pas en octets). Le paramètre `taille_elem` est la taille d'un élément en octets. Par exemple une file de 7 `int` (comme celle sur la figure ci-dessus) sera créée par `create_fifo( 7, sizeof(int) )`.

Initialement les trois pointeurs seront égaux : `first == libre == occupe` La fonction `create_fifo` retourne NULL si la création échoue (par exemple problème de `malloc`), sinon elle retournera `fifo`, c'est-à-dire l'adresse de `struct file`.

Exercice 5 [0.25 point] Écrire la fonction

```
void delete_fifo(fifo f)
```

qui supprime la file.

Exercice 6 [0.25 point] Écrire la fonction

```
int empty_fifo(fifo f)
```

qui retourne 1 si la file est vide et 0 sinon.

Exercice 7 [2 points] Écrire la fonction

```
void *get_fifo(fifo f, void *element)
```

qui récupère le premier élément de la file. La fonction doit copier cet élément à l'adresse `element` et elle doit le supprimer de la file (ce qui revient à changer la valeur de pointeur `occupe`).

La fonction retourne NULL si la file est vide (et rien ne sera écrit à l'adresse `element`), sinon elle retourne `element`.

Exercice 8 [2.5 points] Écrire une fonction auxiliaire

```
static void *put_fifo_no_shift( fifo f, void *pelem )
```

qui mets un éléments dans la file uniquement quand il y a de la place à la fin de la file.

Le paramètre `pelem` contient l'adresse d'élément qui sera recopié à l'adresse `libre`). Cette fonction permet d'ajouter un nouveau élément dans la file uniquement si `libre < last`, i.e. `libre` donne l'adresse à l'intérieur de la file. Dans ce cas la fonction retourne l'adresse du nouveau élément dans la file.

Si `libre == last` c'est-à-dire il n'y a plus de place à la fin de la file alors la fonction retournera `NULL` et le nouveau élément ne sera pas mis dans la file.

Exercice 9 [4 points] Écrire la fonction

```
void *put_fifo(fifo f, void *pelem)
```

qui met un nouveau élément à la fin de la file. Contrairement à la fonction précédente, cette fonction doit insérer un élément même s'il n'y a plus de place après le dernier élément dans la file.

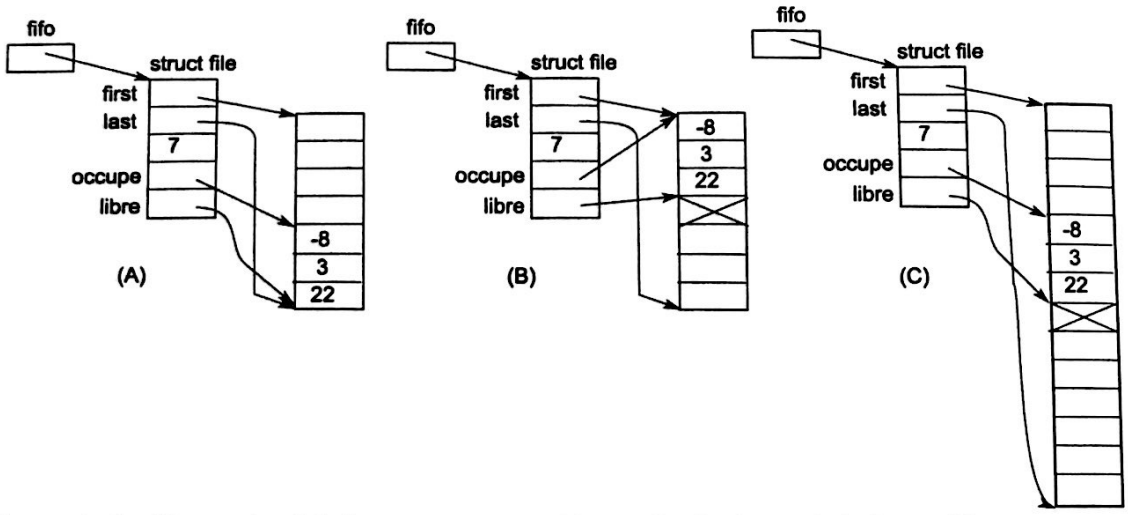
Dans `put_fifo` vous pouvez utiliser librement la fonction de l'exercice précédent (même si vous ne l'avez pas écrite).

S'il y a de la place à la fin de la file alors on procède comme dans l'exercice précédent.

Supposons maintenant que `libre == last` c'est-à-dire il n'y a pas de place à la fin de la file, comme dans la figure (A) dessous. Il y a deux algorithmes possibles pour insérer un nouveau élément dans la file dans cette situation.

(B) Le premier algorithme déplace (`memcpy`) tous les éléments au début du tableau comme dans la figure (B) ci-dessus. Maintenant le nouveau éléments sera inséré dans la case marquée, juste après le dernier élément 22. Cet algorithme est applicable uniquement si le nombre de cases occupées est inférieur au nombre total de cases dans le tableau. Noter que, une fois le déplacement des éléments et ajustement de pointeurs effectué, l'insertion peut se faire avec la fonction de l'exercice précédent.

(C) Le deuxième algorithme agrandit le tableaux deux fois (`realloc`) ce qui donne des nouvelles cases libres après le dernier élément dans la file, voir la figure (C) ci-dessous. Une fois le tableau agrandi et les pointeurs ajustés, l'insertion peut se faire avec la fonction de l'exercice précédent dans la case après le dernier élément.



Dans votre implémentation de la fonction `put_fifo`, s'il n'y a plus de place après le dernier élément, vous devez utiliser le deuxième algorithme quand le pourcentage de cases libres est inférieur à 25%, c'est-à-dire quand

```
(libre - occupe) < (last - first)*0.25
```

Dans le cas contraire vous devez appliquer le premier algorithme.
