IUT de Montpellier, Dép. Informatique
M2101 - Archi 2 V. Poupet

Examen (2 heures)

Les documents, calculatrices, téléphones etc. ne sont pas autorisés.
Le sujet comporte 4 pages.

Remarque : Lorsqu'il vous est demandé d'écrire un programme, vous pouvez ignorer les inclusions de bibliothèques. Vous pouvez aussi négliger les cas d'erreurs, sauf s'il est explicitement demandé d'en tenir compte.

Exercice 1. *Questions de cours*

1 Écrivez une fonction `void echange(int *p1, int *p2)` qui échange les valeurs des entiers se trouvant aux adresses `p1` et `p2`. Pourquoi passe-t-on des pointeurs vers des entiers et non pas directement des entiers ?

Solution :

```
void echange(int *p1, int *p2) {
  int x = *p1;
  *p1 = *p2;
  *p2 = x;
}
```

2 L'instruction « `float t[10];` » permet d'allouer automatiquement (dans la pile) un tableau pouvant contenir 10 objets de type `float`. Donnez la commande permettant d'allouer un tableau semblable dynamiquement (dans le tas).

Solution :

```
    float *t = (float *)malloc(10 * sizeof(float));
```

3 Citez trois différences entre l'allocation dynamique et l'allocation automatique.

Solution : Quelques différences discutées en cours :
- Les variables allouées dans la pile sont locales à la fonction qui les déclare, tandis que les variables dans le tas sont accessibles de manière globale.
- Il faut libérer manuellement (à l'aide de la fonction `free`) l'espace des variables allouées dynamiquement (les variables allouées automatiquement sont libérées lorsque la fonction qui les a définies termine).
- L'espace dans la pile est limité à quelques Mo, tandis que l'espace dans le tas n'est limité que par la mémoire de l'ordinateur.
- En général les accès à la pile sont plus rapides que les accès au tas car la pile est automatiquement placée dans de la mémoire plus rapide (cache).
- La pile ne se fragmente pas.

Exercice 2. *Correction d'erreurs*

On veut écrire un programme qui calcule les 12 premières valeurs de la fonction factorielle, les place dans un tableau et les affiche à l'écran.

On utilise une fonction annexe `int *factorielle(int n)` qui prend en argument un entier n et créé un tableau de n cases, place les valeurs de la fonction factorielle dans le tableau et renvoie finalement un pointeur vers le tableau ainsi créé.

On écrit le programme suivant :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int *factorielle(int n) {
5   int t[n];
```

```
6   t[0] = 1;
7
8   for (i = 0; i < n; i++) {
9     t[i] = i * t[i-1];
10  }
11  return t;
12 }
13
14 int main() {
15  int t = factorielle(12);
16  int i;
17  for (i = 0; i < 12; i++) {
18    printf("%d! = %d\n", i, t[i]);
19  }
20 }
```

Lors de la compilation, le compilateur nous indique les erreurs et avertissements suivants :

```
prog.c:8:7: erreur: 'i' undeclared (first use in this function)
prog.c:11:2: attention : cette fonction retourne l'adresse d'une variable locale
prog.c:15:10: attention : initialization makes integer from pointer without a
cast [enabled by default]
```

1 Pour chacune des erreurs et avertissements indiqués par le compilateur, expliquez brièvement quel est le problème et indiquez les lignes à modifier pour résoudre le problème.

Solution :

a. la variable `i` n'a pas été déclarée. il faut ajouter « `int i;` » (ligne 7)

b. le résultat de la fonction `factorielle` est un pointeur vers une adresse de la pile. En effet, la variable `t` est allouée automatiquement dans la pile, et donc cette adresse sera réutilisée pour d'autres données lorsque la fonction aura terminé. Pour corriger ça, il faut allouer `t` dans le tas : « `int *t = (int *)malloc(n*sizeof(int));` » (ligne 5)

c. la fonction `factorielle` renvoie un pointeur vers un **`int`** (tableau d'entiers) et non pas un **`int`**. La variable `t` est donc de type **`int *`** : « `int *t = factorielle(12);` » (ligne 15)

2 Après avoir corrigé toutes les erreurs indiquées par le compilateurs, l'exécution du programme s'interromp avec le message d'erreur « `Erreur de segmentation` ». À quoi est dû cette erreur, comment la corriger ?

Solution : Dans la fonction `factorielle`, la boucle **`for`** commence à `i=0`, mais dans la boucle on accède à la valeur de `t[i-1]`. Ainsi, pour la première itération de la boucle, on demande la valeur `t[-1]` qui n'est pas disponible, ce qui provoque une erreur de segmentation. Pour corriger cette erreur, il faut commencer à `i=1` :

```
8   for (i = 1; i < n; i++) {
9     t[i] = i * t[i-1];
10  }
```

Exercice 3. *Modification de fichiers*

Dans cet exercice, nous allons manipuler des fichiers à l'aide des fonctions vues en cours et en TD :

— `FILE *fopen(const char *restrict filename, const char *restrict mode)`
— `int fclose(FILE *stream)`
— `int fgetc(FILE *stream)`
— `char *fgets(char * restrict str, int size, FILE * restrict stream)`
— `int fputc(int c, FILE *stream)`
— `int fputs(const char *restrict s, FILE *restrict stream)`

On considère le squelette de programme suivant :

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char tr(char c, char *s1, char *s2) {
  int i;
  for (i = 0; i < strlen(s1); i++){
    if (c == s1[i]) {
      return s2[i];
    }
  }
  return c;
}

int main(int argc, char **argv) {
  if (argc < 4) {
    printf("Erreur: pas assez d'arguments.\n");
    exit(1);
  }
  if (strlen(argv[2] != strlen(argv[3]) {
    printf("Les deux chaînes de caractères doivent avoir la même longueur.\n");
    exit(1);
  }

  /* [1] */
  // ouverture de fichiers

  /* [2] */
  while (1) {
    // Boucle principale terminée par un break
  }

  printf("\n");
}
```

1 Que fait la fonction `tr` ? Quel est le résultat de l'appel « `tr('b', "abcde", "fghij");` » ?
Indication : La fonction `strlen` de la bibliothèque `string.h` renvoie la longueur d'un tableau de caractères dont la fin est indiquée par le caractère `'\0'`.

Solution : La fonction remplace un caractère se trouvant dans `s1` en le caractère se trouvant à la même position dans `s2` (si le caractère n'est pas dans `s1` il n'est pas modifié). Dans l'exemple, la fonction renvoie `'g'` car `'b'` est à l'indice 1 dans `s1`.

On veut que notre programme prenne trois arguments en ligne de commande : un nom de fichier à ouvrir et deux chaînes de caractères qui définiront l'action à exécuter. Par exemple :

```
$ ./a.out fichier.txt "abcde" "fghij"
```

2 Écrivez les lignes à insérer en `[1]` pour que le programme ouvre en lecture le fichier correspondant au premier paramètre.

Solution :

```c
  FILE *fin = fopen(argv[1], "rb");
```

3 Ajoutez un test qui vérifie si l'ouverture du fichier s'est déroulée correctement et affiche un message d'erreur adapté (en utilisant `errno` et/ou `perror`) avant de quitter l'exécution en cas de problème.

Solution :

```c
  FILE *fin = fopen(argv[1], "rb");
  if (fin == NULL) {
    perror(argv[1]);
    exit(1);
  }
```

4 Complétez la boucle `while` du programme pour qu'il lise les caractères un par un du fichier ouvert à l'aide de la fonction `fgetc`, puis affiche à l'écran le caractère renvoyé par la fonction `tr` à laquelle on passe comme arguments le caractère lu ainsi que les deux chaînes de caractères reçues en argument du programme (`"abcde"` et `"fghij"` dans l'exemple).
Veillez à bien mettre une condition d'arrêt de la boucle lorsque le programme a fini de lire tous les caractères du fichier.
Indication : Si `x` est un caractère, vous pouvez utiliser l'appel « `printf("%c", x);` » pour l'afficher à l'écran.

Solution :

```
int c;
while (1) {
  c = fgetc(fin);
  if (c == EOF) {
    break;
  }
  printf("%c", tr(c, argv[2], argv[3]));
}
```

5 Modifiez la boucle `while` écrite à la question précédente pour que le programme lise maintenant les caractères du fichier par blocs de 256 caractères en utilisant la fonction `fgets`.
Indication : Vous devez déclarer de nouvelles variables en dehors de la boucle `while`, modifier la condition d'arrêt de la boucle et envoyer un par un les caractères lus par la fonction `fgets` à la fonction `tr`.
Extrait du manuel de `fgets` :

```
char *fgets (char *s, int size, FILE *stream);

fgets() lit au plus size - 1 caractères depuis stream et les place dans le
buffer pointé par s. La lecture s'arrête après EOF ou un retour-chariot '\n'.
Si un retour-chariot est lu, il est placé dans le buffer. Un caractère nul '\0'
est placé à la fin de la ligne.

fgets() renvoie le pointeur s si elle réussit et et NULL en cas d'erreur, ou si
la fin de fichier est atteinte avant d'avoir pu lire au moins un caractère.
```

6 Expliquez pourquoi la seconde version du programme (qui lit les caractères par blocs de 256) est beaucoup plus rapide que la première lorsqu'on manipule des gros fichiers.
Indication : Quelles sont les instructions du programme qui sont les plus longues à exécuter ?

Solution : Les appels systèmes sont des opérations très coûteuses. Il faut en effet interrompre l'exécution du programme en cours, sauvegarder son état, passer en mode noyau, exécuter l'appel système puis recharger l'état du programme initial et reprendre le fonctionnement.
Si on lit les caractères par blocs au lieu de les lire un par un, on ne fait qu'un appel système pour lire un bloc d'environ 256 caractères, ce qui est nettement plus rapide.
Remarque : La fonction `fgets` arrête de lire des caractères lorsqu'elle rencontre un retour à la ligne, et il est donc possible qu'elle lise nettement moins de 256 caractères à la fois. En particulier, si le fichier est constitué de lignes très courtes, on va faire tout de même beaucoup d'appels systèmes. Pour être sûrs de lire 256 caractères à la fois (tant qu'on n'est pas au bout du fichier) il faudrait utiliser l'appel `read`.

7 Si le contenu du fichier `test.txt` est

```
Un deux trois
Quatre cinq six
```

qu'affiche le programme lorsque l'on éxécute la commande « `./a.out test.txt "abcd" "ABCD"` » ?

Solution : Le programme remplace les lettres `a`, `b`, `c` et `d` par leurs versions en majuscule. On devrait donc obtenir l'affichage

```
Un Deux trois
QuAtre Cinq six
```

Exercice 4. *Processus et threads*

1 Comment est organisée la mémoire occupée par un processus ? Décrivez brièvement le contenu de chaque zone.

Solution : Les principales parties de la mémoire occupée par un processus sont :
— instructions du programme (zone `text`) contenant les intrsuctions de l'exécutable en cours d'exécution ;
— variables globales (zone `data`) ;
— pile (variables locales à chaque fonction) ;
— tas (variables allouées dans la mémoire de l'ordinateur, accessible par toutes les fonctions du processus).

2 Que se passe-t-il au niveau de la mémoire d'un processus lorsque la fonction `exec` est appelée ?

Solution : La fonction `exec` remplace toute la mémoire du processus par un nouvel exécutable. Le segment `text` contient alors les instructions du nouveau programme et la pile le tas sont complètement réinitialisés.

3 Comment s'organise la mémoire lorsqu'un processus exécute plusieurs *threads* ? Quelles zones sont communes à tous les *threads*, et quelles zones contiennent des informations propres à chaque *thread* ?

Solution : Lorsqu'un processus exécute plusieurs *threads*, sa pile est divisée en plusieurs segments permettant à chaque *thread* d'avoir sa propre pile. Toutes les autres zones de la mémoire sont communes à tous les *threads* (ils ont les mêmes segments `text` et `data` et le même tas).

4 Écrivez un programme qui effectue les actions suivantes :
— le programme démarre un processus fils par argument passé en ligne de commande ;
— en utilisant une fonction de la famille `exec`, chaque fils doit remplacer son code d'exécution par celui de la commande `echo` en passant comme argument l'argument du programme de départ qui lui correspond ;
— le processus père doit ensuite attendre la terminaison de chacun des fils et afficher « `fin du programme` » avant de terminer.
Par exemple, si l'on exécute le programme avec l'appel « `./a.out toto titi` », deux processus fils sont lancés, le premier exécute la commande « `echo toto` » et le second la commande « `echo titi` », puis le père affiche « `fin du programme` » et termine.
Indication : Vous pouvez utiliser la fonction `execlp`, qui prend comme arguments le nom de l'exécutable à exécuter, suivi de la liste des arguments à passer à l'exécutable. La liste doit commencer par le nom de l'exécutable (une deuxième fois) et se terminer par la valeur `NULL`.
Par exemple, si vous voulez exécuter la commande « `cp a.txt b.txt` », il faut appeler « `execlp("cp", "cp", "a.txt", "b.txt", NULL);` ».

Solution :

```c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/wait.h>

int main(int argc, char **argv) {
  int nb_processus = argc - 1;
  int i, pid, pids[nb_processus];
  for (i = 0; i < nb_processus; i++) {
    pid = fork();
    if(pid == 0) {
      execlp("echo", "echo", argv[i+1], NULL);
    } else {
      pids[i] = pid;
    }
  }
  for (i = 0; i < nb_processus; i++) {
    waitpid(pids[i], NULL, 0);
  }
  printf("fin\n");
}
```

Le programme suivant essaie de faire sensiblement la même chose que ce qui était demandé à la question 4, mais en utilisant des *threads* au lieu des processus :

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void *f(void *arg) {
  char *s = arg;
  execlp("echo", "echo", s, NULL);
  return NULL;
}

int main(int argc, char **argv) {
  int nb_threads = argc - 1;
  pthread_t th[nb_threads];
  char *args[nb_threads];
  int i;
  for (i = 0; i < nb_threads; i++) {
    pthread_create(&th[i], NULL, f, argv[i+1]);
  }
  for (i = 0; i < nb_threads; i++) {
    pthread_join(th[i], NULL);
  }
  printf("fin\n");
}
```

5 À votre avis, que va-t-il se passer lors de l'exécution de ce programme ? Quel sera l'affichage obtenu ? Justifiez votre réponse.

Solution : Le premier *thread* « fils » qui s'exécute fait un appel à la fonction `execlp`. Lorsque cet appel est exécuté, la totalité du processus est remplacé par un processus exécutant la fonction `echo`. Ainsi, tous les autres *threads* disparaissent et seul le premier à appeler `execlp` exécute sa tâche.
Le programme affiche simplement le premier argument qui lui a été passé en ligne de commande (si l'on suppose que c'est le *thread* correspondant au premier argument qui démarre effectivement en premier).