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 7 pages.
Le barème est donné à titre indicatif et pourra être légèrement modifié.

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. *Fonctions, tableaux et pointeurs (4 points)*

1 Écrivez une fonction `int somme(int *t, int n)` qui prend en argument un tableau d'entiers t et le nombre de cases n de ce tableau et renvoie la somme des éléments contenus dans le tableau.

Solution :

```
int somme(int *t, int n) {
  int s = 0;
  int i;
  for (i = 0; i < n; i++) {
    s += t[i];
  }
  return s;
}
```

2 Pourquoi est-il nécessaire de passer en argument le nombre de cases du tableau ?

Solution : Les tableaux en C ne sont qu'un pointeur vers la première valeur du tableau. Il n'y a aucun moyen de savoir où s'arrête l'espace mémoire consacré au tableau si l'on ne connaît pas le nombre de cases.

3 Écrivez une fonction `int *range(int n)` qui renvoie un tableau de n cases contenant les entiers compris entre 0 et $(n-1)$ (inclus).
Attention : Le tableau renvoyé par la fonction `range` doit être alloué dans le tas lors de l'exécution de la fonction.

Solution :

```
int *range(int n) {
  int *t = malloc(n * sizeof(int));
  int i;
  for (i = 0; i < n; i++) {
    t[i] = i;
  }
  return t;
}
```

Exercice 2. *Débuggage (4 points)*

On veut écrire une fonction `int minuscule(char *s)` qui prend en argument une chaîne de caractères (dont la fin est indiquée par le caractère `'\0'`) et renvoie le nombre de lettres minuscules que contient cette chaîne. On écrit le programme suivant :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int minuscules(char *s) {
5   int nbMin = 0;
```

```
6   int i;
7   for (i = 0; s[i] != '\0'; i++) {
8     if (s[i] >= a && s[i] <= z) {
9       nbMin = nbMin + 1;
10    }
11  }
12  return *nbMin;
13 }
14
15 int main() {
16   char texte = "Ceci est un TEXTE.";
17   printf("Le texte contient %d minuscules.\n", minuscules(texte));
18 }
```

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

```
prog.c: In function ‘minuscules’:
prog.c:8:15: error: ‘a’ undeclared (first use in this function)
prog.c:8:28: error: ‘z’ undeclared (first use in this function)
prog.c:12:9: error: invalid type argument of unary ‘*’ (have ‘int’)
prog.c: In function ‘main’:
prog.c:16:15: warning: initialization makes integer from pointer without a cast
prog.c:17:2: warning: passing argument 1 of ‘minuscules’ makes pointer from integer without a cast
```

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 :

— Ligne 8 : on veut comparer aux caractères `'a'` et `'z'` et non pas à des variables `a` et `z` (qui n'existent pas et donc le compilateur se plaint qu'elles n'aient pas été déclarées).

```
8     if (s[i] >= 'a' && s[i] <= 'z') {
```

— Ligne 12 : `nbMin` est de type `int`. Ce n'est pas un pointeur donc il ne faut pas mettre « `*` ».

```
12   return nbMin;
```

— Ligne 16 : la valeur `"Ceci est un TEXTE."` est de type `char *` (c'est une chaîne de caractères, ou encore un tableau de caractères). Il faut donc déclarer la variable `texte` comme étant de type `char *`.

```
12   char *texte = "Ceci est un TEXTE.";
```

— Ligne 17 : le compilateur se plaint car la fonction `minuscule` attend un argument de type `char *` alors que la variable `texte` est de type `char`. Ceci est dû à la mauvaise déclaration de la variable `texte` qui a été corrigée précédemment. Il n'est donc pas nécessaire de faire de modification supplémentaire pour corriger cette erreur.

Exercice 3. *Mémoire (4 points)*

On s'intéresse maintenant à un programme qui calcule les valeurs de la suite de *Fibonacci* définie par la relation de récurrence suivante :

$$u_0 = 1, \quad u_1 = 1$$
$$\forall n \geq 0, \quad u_{n+2} = u_n + u_{n+1}$$

Le programme va enregistrer toutes les valeurs calculées dans un tableau `fibo_tab` et compléter le tableau jusqu'à la valeur désirée.

On considère les trois variantes obtenues à l'aide du code suivant, selon que l'on décommente l'une des lignes 5, 9 ou 10 (il faut décommenter exactement l'une de ces trois lignes pour déclarer correctement la variable `fibo_tab`) :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define TABSIZE 3000000
4
5  // int fibo_tab[TABSIZE]; // variante 1
```

```
6
7  int fibo(int n) {
8    int i;
9    // int fibo_tab[TABSIZE]; // variante 2
10   // int *fibo_tab = malloc(TABSIZE * sizeof(int)); // variante 3
11   fibo_tab[0] = 1;
12   fibo_tab[1] = 1;
13   for (i = 0; i < n-2; i++) {
14     fibo_tab[i+2] = fibo_tab[i] + fibo_tab[i+1];
15   }
16   return fibo_tab[n-1];
17 }
18
19 int main() {
20   printf("fibo(3000000) = %d\n", fibo(3000000));
21 }
```

1 Pour chacune des trois variantes, indiquez dans quel espace de la mémoire du processus sera placé le tableau `fibo_tab`.

Solution : La première variante déclare le tableau de manière globale (hors de toute fonction). Le tableau est donc placé dans le segment *data* qui est alloué au démarrage du processus (ces variables ne sont donc ni dans la pile, ni dans le tas). Plus précisément comme le tableau n'est pas initialisé, il est mis dans le segment *BSS* de l'exécutable ce qui permet de ne pas occuper de mémoire dans le fichier exécutable (on écrit juste la taille que devrait occuper le tableau), et cet espace sera initialisé à 0 au démarrage du processus.

La seconde variante déclare le tableau comme une variable locale de la fonction `fibo`. La place allouée pour le tableau est donc dans la pile (allocation automatique).

Enfin, la troisième variante alloue l'espace du tableau à l'aide de la fonction `malloc`, qui prend donc l'espace dans le tas.

Chacune des trois variantes peut être compilée sans erreurs. Cependant lors de l'exécution lorsque l'on essaie de calculer trois millions de valeurs, comme c'est le cas dans le code présenté, l'une des variantes produit une erreur.

2 Quelle est la variante qui ne s'exécute pas correctement ? Quel est le problème ?

Solution : Des trois zones de mémoire considérées dans la question précédente, la pile est la plus limitée. Sa taille est dépendante du système d'exploitation mais en général de l'ordre de quelques Mo (8 Mo sur une installation typique d'*Ubuntu*). Le tas n'est limité que par la taille de la mémoire RAM (qui est donc de plusieurs Go) et le segment *data* du processus est alloué au démarrage du processus en fonction de ce qui est indiqué dans l'exécutable (la taille du segment *data* ne change pas pendant l'exécution). Ici encore, tant que ça tient dans la RAM, le processus peut être démarré.

Dans l'exemple, un tableau de 3 millions d'entiers occupe environ 12 Mo (avec des entiers codés sur 4 octets) ce qui dépasse la taille de la pile. C'est donc la deuxième variante qui provoque une erreur.

Lors de l'exécution des deux variantes qui ne génèrent pas d'erreurs, le message qui s'affiche est

```
fibo(3000000) = 768372992
```

Cette valeur est très inférieure à la valeur réelle du 30000000-ème élément de la suite de Fibonacci (qui est de l'ordre de $6,03 \times 10^{626962}$).

3 Expliquez pourquoi le résultat n'est pas correct.

Solution : La taille occupée par une variable de type *int* est limitée (4 octets, soit 32 bits sur une installation typique d'*Ubuntu*). Puisque la taille est limitée, la valeur maximale d'un `int` est également limitée ($2^{31} - 1$ sur 4 octets, pour des entiers signés, *cf.* cours du premier semestre).

Cependant, les opérations arithmétiques ne provoquent pas d'erreur si le résultat dépasse la valeur maximale (on peut toutefois demander au programme de vérifier qu'il n'y a pas eu de dépassement, mais ce n'est pas le cas ici), le résultat est simplement tronqué (on ne garde que les 32 bits les plus faibles). Ceci explique que le résultat affiché par le programme soit un entier, mais pas le résultat attendu.

Exercice 4. *Manipulation de fichiers (4 points)*

1 Écrivez un programme qui ouvre un fichier dont le nom est passé en argument lors de l'appel en ligne de commande et affiche à l'écran le nombre de lignes (délimitées par le caractère `'\n'`), ainsi que le nombre de caractères qu'il contient.

Par exemple, si le fichier `toto.txt` contient 3 lignes et 250 caractères, la commande « `./a.out toto.txt` » doit afficher le message « 3 lignes et 250 caractères. »

Remarque : Vous pouvez utiliser selon votre préférence les fonctions de la bibliothèque `stdio.h` telles que `fopen`, `fgetc`, etc. ou les appels système tels que `open`, `read`, etc.

Solution : Avec la bibliothèque `stdio.h`

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

int main(int argc, char **argv) {
  FILE *input = fopen(argv[1], "rb");
  int c, nbLignes = 0, nbChar = 0;
  while (1) {
    c = fgetc(input);
    if (c == EOF) { break; }
    if (c == '\n') { nbLignes++; }
    nbChar++;
  }
  printf("%d lignes et %d caractères.\n", nbLignes, nbChar);
}
```

Avec les appels système

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv) {
  int fd = open(argv[1], O_RDONLY);
  int i, nbread, nbChar = 0, nbLignes = 0;
  char buffer[512];

  while (1) {
    nbread = read(fd, buffer, 512);
    if (nbread == 0) { break; } // fin du fichier
    nbChar += nbread; // incrémenter le nombre de caractères lus
    for (i = 0; i < nbread; i++) {
      if (buffer[i] == '\n') {  // on cherche les fins de lignes
        nbLignes++;
      }
    }
  }
  printf("%d lignes et %d caractères.\n", nbLignes, nbChar);
}
```

2 Ajoutez à votre programme un test pour vérifier que l'ouverture de fichier s'est bien passée, et affiche un message d'erreur adapté en cas d'erreur.

Solution : Avec `stdio.h`

```
  FILE *input = fopen(argv[1], "rb");
  if (input == NULL) {
    perror(argv[1]); // ou : printf("Erreur d'ouverture.\n");
    exit(1);
  }
```

Avec les appels système

```
int fd = open(argv[1], O_RDONLY);
if (fd < 0) {
  perror(argv[1]); // ou : printf("Erreur d'ouverture.\n");
  exit(1);
}
```

Exercice 5. *Processus (4 points)*

On considère le programme suivant :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main() {
6    printf ("pid 0 = %d\n", getpid());
7    int pid;
8    pid = fork();
9    printf ("pid 1 = %d\n", pid);
10   if (fork() == 0) {
11     pid = fork();
12     printf("pid 2 = %d\n", pid);
13   } else {
14     sleep(10);
15   }
16 }
```

1 Représentez à l'aide d'un schéma l'ensemble des processus créés lors de l'exécution du programme, en précisant pour chacun quel est son PID et quel est son parent. On pourra supposer que le premier processus a comme PID 1000 (donc la commande `getpid` de la ligne 6 renvoie 1000) et que les différents processus créés ont des PID successifs 1001, 1002, etc.

Solution : Il y a trois lignes dans le programme qui appellent la fonction `fork` (lignes 8, 10 et 11). Au total, 6 processus seront créés lors de l'exécution du programme (cf. figure 1).

Remarque : l'ordre dans lequel les processus sont créés peut dépendre de l'exécution (puisque les processus existant à un instant donné s'exécutent en parallèle). Par rapport à ce qui est indiqué sur la figure 1, il est par exemple possible que le processus issu du processus 1001 soit créé avant le second issu de 1000 (ce qui échangerait les numéros 1002 et 1003). De même l'ordre de création des processus nommés 1004 et 1005 sur la figure n'est pas défini.

2 Donnez la suite de lignes qu'affiche le programme lorsqu'il est exécuté (on suppose toujours que le premier PID est 1000).

Remarque : L'ordre dans lequel les lignes sont affichées n'est pas totalement déterminé, il vous suffit donc de donner une sortie possible du programme.

Solution : D'après le schéma de la figure 1, une sortie possible du programme est

```
pid 0 = 1000
pid 1 = 1001
pid 1 = 0
pid 2 = 1004
pid 2 = 0
pid 2 = 1005
pid 2 = 0
```

L'ordre dans lequel ces lignes sont affichées n'est pas fixé. Le schéma de la figure 1 représente les contraintes d'ordre : si une ligne se situe dans une branche issue d'une autre ligne, elle doit être affichée après (par exemple, la ligne « `pid 2 = 1004` » est nécessairement après la ligne « `pid 0 = 1000` ». Si par contre deux lignes ne sont pas sur le même chemin dans l'arbre, leur ordre d'affichage relatif n'est pas contraint (par exemple la lige « `pid 2 = 1004` » pourrait être affichée avant la ligne « `pid 1 = 0` »).

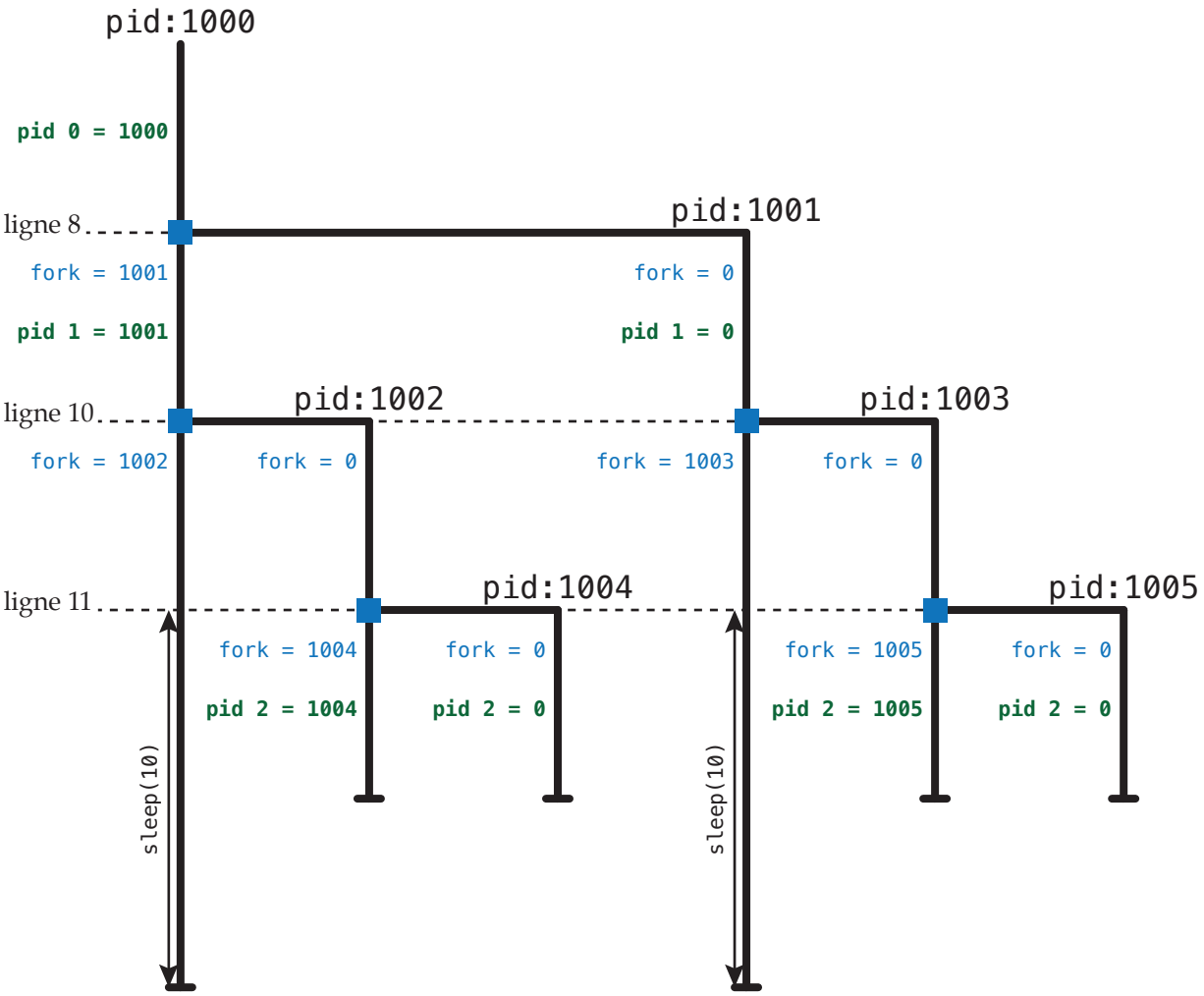

FIGURE 1 – Représentation des processus créés par le programme de l'exercice 5. Chaque processus est représenté par une ligne verticale, les appels à la fonction `fork` sont représentés par un carré bleu (le résultat de la fonction `fork` est indiqué en bleu pour chacun des processus concernés) et les messages affichés par les différents appels à `printf` sont représentés en vert.

3 Qu'est-ce qu'un processus *zombie* ? Lors de l'exécution du programme, quels processus deviennent des zombies pendant plus d'une seconde (on rappelle que l'instruction `sleep(10)` provoque un délai de 10 secondes) ?

Solution : Un processus *zombie* est un processus qui a terminé son exécution mais dont le parent n'a pas autorisé la suppression de la table des processus (par la fonction `wait` ou `waitpid` par exemple). Ces processus n'occupent plus le processeur ni même la mémoire (leurs ressources ont été libérées) mais continuent à exister dans la table des processus. Si trop de processus *zombie* co-existent, le système peut être à cours de numéro pour créer des nouveaux processus.
Dans le programme étudié les processus 1002 et 1003 (d'après la numérotation indiquée par la figure 1) deviennent des *zombies* pendant quelques secondes (le temps que leurs parents respectifs (1000 et 1001) terminent, ce qui prend environ 10 secondes à cause de l'instruction `sleep`). Lorsque les processus 1000 et 1001 terminent, les processus 1002 et 1003 sont automatiquement rattachés au processus `init` (dont le PID est 1), qui se chargera de les enlever de la table de processus.
Notons que les processus 1004 et 1005 ne deviennent pas des *zombies* car ils sont automatiquement récupérés par `init` dès que leurs parents (1002 et 1003) terminent.

Annexe

Voici les signatures de quelques fonctions de manipulations de fichier dans la bibliothèque `stdio.h` ainsi que des fonctions d'appels système d'entrée-sortie. Si vous ne vous souvenez plus des arguments exacts pour certaines fonctions (en particulier `open` et `fopen`) mettez des arguments approximatifs.

```
FILE *fopen(const char *restrict filename, const char *restrict mode);
char *fgets(char * restrict str, int size, FILE * restrict stream);
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
char *fgets(char * restrict str, int size, FILE * restrict stream);
int fputs(const char *restrict s, FILE *restrict stream);
int fseek(FILE *stream, long offset, int whence);
int fclose(FILE *stream);

int open(const char *path, int oflag, ...);
ssize_t read(int fildes, void *buf, size_t nbyte);
ssize_t write(int fildes, const void *buf, size_t nbyte);
off_t lseek(int fildes, off_t offset, int whence);
int close(int fildes);
```