Alain DIAS

Rapport TP4

Temps estimé : 5 heures
Temps de réalisation : 7 heures 30

1 Code 1

Le code argc = 1 est toujours vrai, de plus on perd la capacité à connaître le nombre d'arguments en entrée du programme. En remplaçant « argc = 1 » par « argc == 1 », le code présente toujours le problème de ne pas arrêter son exécution lorsque le nombre d'arguments est incorrect. Si l'on ne passe aucun arguments au programme, on tentera toujours d'accéder à argv[1] qui ne devrait pas exister, il faut donc réaliser un exit ou return à l'intérieur du if. Finalement, la variable i n'est pas initialisée, ce qui ne permet pas au programme de convertir le premier argument du programme en long. Pour corriger cette erreur, on peut soit initialiser i à 0, ou bien remplacer la somme « i += strtol(argv[1], NULL, 10) ; » par « i = strtol(argv[1], NULL, 10) ; »

Code corrigé

```
void main (int argc, char **argv)
{
 long i = 0;

 if (argc == 1) {
        printf("Veuillez saisir un argument");
        return; //On quitte le code
 }
 i += strtol(argv[1], NULL, 10);
 printf("i = %d\n", i);
}
```

2 Code 2

La variable lg n'est pas initialisée à 0 alors que dans la suite du programme on incrémente cette variable, ce qui ne permettra pas d'obtenir ce que l'on souhaite puisqu'il est impossible de savoir ce que contient lg. On souhaiterai également utiliser IS_VALID_CHAR(x) à la place de IS_VALID_NUM(*chaine++). De plus en initialisant lg à 0, on peut utiliser chaine[lg] pour parcourir la chaîne car *chaine++ ne permet pas d'obtenir un caractère tout en parcourant la chaîne. Finalement il faut inverser les paramètre lg et chaine à l'intérieur du printf puisque dans l'était actuel, on tente d'afficher un int avec %s et une chaine de caractère avec %d.

Code corrigé

```
#define IS_VALID_CHAR(x) (((x)>='0' && (x)<='9') || \
 ((x)>='A' && (x)<='F') ?1:0)
void ma_fonction(char* chaine)
{
 int lg = 0;
 while(IS_VALID_CHAR(chaine[lg])) {
        lg++;
 }
 printf("%s commence par %d digits hexa\n",
 chaine, lg);
}
```

3 Code 3

Sur une machine 32 bits, un long est codé sur 4 octets alors que sur une machine 64 bits, un long est codé sur 8 octets. Ce programme peut donc stocker un plus grand nombre de secondes converties en micro secondes sur une machine 64 bits que sur une machine 32 bits. Cependant, le nombre de secondes que renvoie gettimeofday est depuis le 1er janvier 1970, sur un machine 32 bits il y aura un débordement. Le comportement ne sera donc pas le même.

4 Code 4

On a aucune garantie que le message "Myfunc" apparaitra à l'écran. En effet, le main peut se terminer avant que le thread crée puisse afficher son message. Il faut donc ajouter un pthread_join pour attendre que le thread crée se termine avant de terminer l'execution du programme principal. De plus, th est un pointeur, ce qui va poser problème lors de la création du thread.

Code corrigé

```
void * myfunc(void *myarg) {
 printf("Myfunc\n");
}
main() {
 pthread_t th; //On n'a plus de pointeur
 int res;
 res = pthread_create(&th, NULL, myfunc, NULL);//on passe l'adresse
 if (res = 0) printf("Thread create OK!\n");
 pthread_join(th, NULL);
}
```

5 Code 5

On utilise le paramètre lg pour allouer la taille de my_str, or lors de la copie de la chaîne buf dans my_str, on risque de déborder si lg est inférieur à la longueur de la chaîne + 1 (\0 est copié par strcpy). Il faudrait plutôt que d'utiliser un paramètre lg, faire appel à strlen pour connaitre la longueur de la chaine et ajouter 1 (strlen ne compte pas \0 dans la longueur).
Finalement, on ne teste pas une éventuelle erreur rencontrée par l'allocation mémoire.

Code corrigé

```
char * save_str(char * buf) //On n'utilise plus de parametre lg
{
 char *my_str;
 unsigned int lg = strlen(buf) + 1;//on recupere la longueur de buf
 if((my_str = malloc(lg)) == NULL) return NULL;
 strcpy(my_str, buf);
 return(my_str);
}
```

6 Code 6

Ce programme ne va pas créer 20 processus. En effet, chaque fils va continuer la boucle et créer des processus qui vont à leur cours créer des processus. On peut de cette façon très vite atteindre la limite de processus pouvant être créer. Le nombre de processus créer sera de 2^{20}- 1. Il faut donc faire en sorte que le fils quitte la boucle for après sa création (à l'aide d'un break).

7 Code 7

Dans ce code, i n'est pas initialisé et est utilisé tel quel dans la boucle for du thread. De plus, on remarque que même si i était initialisé à 0 dans le for, on créerait un infinité de threads puisque chaque threads va recommencer la boucle à 0. Une solution pour résoudre ce problème serait de rendre i global et de l'initialiser à 0. Il faut de plus, utiliser un mutex pour s'assurer qu'un autre thread ne crée pas d'autre thread avant la fin de l'exécution de la boucle for. En effet, cela aurai pour effet de produire plus de 20 threads au total.

8 Code 8

Ici, le compilateur va essayer de faire des optimisations au niveau de l'accès à la mémoire. Or, ce n'est pas ce que nous voulons avec la variable check puisque sa valeur change à travers un handler de signal SIGALARM. Pour s'assurer qu'on lise toujours en mémoire la valeur de check, il faut qu'on declare cette variable comme **volatile**. De plus, il faut initialiser check à 0 pour être sûr de bien compter tous les signaux SIGALARM reçus.

Code corrigé

```
volatile int check = 0;
```

9 Code 9

Ici, le père peut ne pas traiter le signal SIGCHLD au moment souhaité si le fils se termine avant que le père ne se mette en pause pour attendre de recevoir un signal. Si le père s'exécute avant le fils, ce code va fonctionner comme attendu.

1 - Ceci n'est pas lié à la fiabilité des signaux puisque si le fils se termine avant, le père va bien recevoir un signal SIGCHLD, mais va se mettre en attente de signal avec pause par la suite... Or il ne recevra plus de signal puisque son fils lui a déjà envoyé le signal SIGCHLD.

2 - Pour palier à ce problème, il faut bloquer le signal SIGCHLD jusqu'a ce que le père soit intéressé par la gestion de ce signal. On remplacera ensuite le pause par un sigsuspend pour se mettre en attente d'un signal SIGCHLD venant du fils

10 Code 10

Le problème dans ce code est que **pt_s** est une copie locale d'un pointeur. Lorsque l'on souhaite faire une allocation mémoire, c'est cette copie qui va pointer vers l'adresse que nous avons alloué. Or, à la fin de l'exécution de la fonction, cette copie locale sera supprimée et donc on n'aura plus accès à la mémoire allouée. On doit donc recevoir en paramètre un pointeur de pointeur de **my_struct**.
Par ailleurs, on ne teste la réussite de l'allocation mémoire qu'après avoir utilisé la structure allouée **my_struct**, ce qui peut rendre le programme instable.

Code corrigé

```
void my_struct_init(struct my_struct ** pt_s, int val) {
 *pt_s = malloc(sizeof(struct my_struct));
 if (*pt_s == NULL) {
        perror("malloc failed!");
        exit(EXIT_FAILURE);
 }
 (*pt_s)->value = val;
 (*pt_s)->time = time(NULL);
}
```

11 Code 11

Une mauvaise utilisation de l'opérateur sizeof est faite. Il suffit de réorganiser les parenthèses pour régler ce problème.
De plus, la fonction colimacon doit renvoyer un char * mais ne renvoie rien, ce qui va provoquer un problème lors de l'exécution du main. Pour pouvoir renvoyer le tableau remplis comme souhaité, il faut retirer le free et ajouter un return. De cette manière le code dans le main va pouvoir afficher correctement le tableau ou bien gérer une éventuelle erreur d'allocation mémoire. Finalement, il faut modifier le code permettant l'affichage du tableau car la fonction colimacon renvoie un tableau à une dimension alors que le code actuel d'affichage travaille avec un tableau à deux dimensions.

Code corrigé

```
char * colimacon(unsigned int rows, unsigned int cols)
{
  char * pt;
  pt = malloc(sizeof((unsigned int)) * rows * cols); //Parentheses
  if (pt == NULL) {
    perror("malloc failed!");
    return NULL;
  }
  fill_colimacon(pt); // remplit et rien d'autre
  //free(pt); <- On supprime le free
  printf("colimacon OK");
  return pt; //On renvoie le tableau alloue
}
main()
{
  char * tab;
  int i, j;
  tab = colimacon(4, 5);
```

```
  if (tab == NULL) exit(1);
  for (i = 0; i < 4; i++) {
    for (j = 0; j < 5; j++) {
    //On calcule l'indice car ce n'est pas un tableau a
    //2 dimensions
    printf("%d\n", tab[i*5+j]);
    }
  }
}
```

12 Code 12

Ce code traitant d'un code similaire au précédent, on pourra regarder les corrections apportées à la fonction colimacon dans la question précédente.
Dans le main, le return de pt n'a aucun sens également puisque pt n'existe pas.
Cependant, le réel problème de ce code est qu'entre chaque vérification, aucun free n'est effectué. On a des fuites mémoires qui vont devenir de plus en plus importante lorsque i augmente. Il faut donc à la fin de la boucle for libérer la mémoire afin de palier à ce problème.
Il faut également tester le retour de la fonction colimacon afin de detecter un eventuel problème d'allocation mémoire du tableau colimacon afin de pouvoir effectuer le test check_colimacon
On supposera que check_colimacon permet effectivement de bien vérifier que le tableau est rempli correctement à l'aide de tests écris à la main (**Bien que cela semble étonnant pour de si grands tableaux**)

Code corrigé

```
char * colimacon(unsigned int rows, unsigned int cols) {
        //Voir code 11
}
main()
{
 char * tab;
 int i, res;
 static int ok, ko;
 for (i = 1; i < 10000; i++) {
   tab = colimacon(4*i, 5*i);
   if(tab == NULL) { //test du retour de la fonction colimacon
        fprintf(stderr, "Erreur allocation colimacon\n");
        return EXIT_FAILURE;
   }
```

```
    res = check_colimacon(tab);
    (res == 0)? ok++ : ko++;
    free(tab); //On libere la memoire
  }
  printf("Passed %d, Failed %d\n", ok, ko);
  return EXIT_SUCCESS;
}
```

13 Code 13

Dans ce code, on remarque que le contenu de la chaîne **fmt** est d'abord écrit dans le buffer **res** avant de tester si cette écriture n'a pas débordé de la mémoire du tableau. Ce code va fonctionner tant que la longueur de fmt ne dépasse pas **MAX_STRING_LENGTH - 1**. Mais dans le cas où la chaîne de départ est plus longue, on pourra obtenir un segmentation fault lors de l'exécution.
Pour éviter d'avoir a bricoler une solution fausse de gestion de débordement, il vaudrait mieux utiliser directement la fonction **vsnprintf** qui permet de spécifier la longueur maximale à écrire. Il est à noter que vsnprintf renvoie la longueur de la chaine qui a été écrite ou qui aurait pu être écrite (si la chaîne est plus longue que ce que l'on a demandé).
Je vais supposer que ce code ne doit pas renvoyer un pointeur vers le buffer lorsque la chaine à écrire dépasse la longueur MAX_STRING_LENGTH, c'est pourquoi, il faut libérer la mémoire avant de renvoyer NULL.

Code corrigé

```
char * printStringInBuffer (char * fmt, va_list pa)
{
  char * res;
  res = (char *)calloc(MAX_STRING_LENGTH, sizeof(char));
  if (res == NULL)
    return NULL;

  //On va utiliser vsnprintf pour eviter un debordement de res
  if (vsnprintf(res, MAX_STRING_LENGTH, fmt, pa)
        >= MAX_STRING_LENGTH) {
    //res[MAX_STRING_LENGTH-1] = '\0'; <-inutile (free juste apres)
    free(res); //On libere la memoire
    return NULL;
  }
  return res;
}
```