Compte rendu du TP1

Exercice 0

`valeur` est une variable, elle contient la valeur entière, `&valeur` représente l'adresse de la variable `valeur` Résultat de l'éxécution :

```
 valeur = 11.000000
&valeur = 0x7fffb7bc6698
```

- pv : poiteur de double, il contient l'addresse de `valeur`.
- &pv : représente l'adresse du pointeur pv.
- *pv : représente la valeur contenue dans la variable pointée. (ici cela représente la valeur de `valeur`).

Tailles des éléments :

Variable	Taille en mémoire
valeur	8 octets
pv	4 octets pour x32 et 8 octets pour x64
nombre	4 octets
pn	4 octets pour x32 et 8 octets pour x64

valeur et nombre étant des variables de type différents, leur taille est différente. En revanche pv et pn sont tout les deux des pointeurs, il contienent une adresse mémoire de 4 ou 8 octets, ils ont la même taille même s'il ne pointe pas sur le même type.

Echange de variables

En C, les paramètres des fonction sont des copies des arguments passés lors des appels. Cela signifie que l'on ne travaille jamais directement sur les éléments passé lors de l'appel mais avec des d'autres variables qui contiennent exactement les mêmes valeurs.

Dans la fonction initiale, les variables ne seront pas échangées. Mais si on refait une fonction avec des pointeurs, on pourra manipuler les variables via leurs addresses :

```c
void echange2(float *a, float *b) {
    float temp = *a;
    *a = *b;
    *b = temp;
}
```

On prend en paramètres les addresses des arguments et on manipule via les ces pointeurs les valeurs de pi et e. On stock dans `temp` la valeur de `a`(donc celle de `pi`), on donne à `a` la valeur de `b`(donc celle de `e`) et

on donne à `b` la valeur de `temp`(la valeur d'origine de `a`).

Exercice 1

Le but de ce problème est de détecter la présence d'un zéro dans un tableau de unsigned char de taille TABSIZE (par exemple, 10000) en partageant le travail entre plusieurs processus.

Question 1 - À quoi sert l'instruction srandom(time(NULL)) ?

Cette instruction permet de créer une graine pour la fonction random.

Question 2 - Combien de zéros, au minimum et au maximum, peuvent apparaitre dans le tableau à la fin de ce fragment de code ? Justifiez la réponse.

Aucun zéros ne peuvent apparaitre, en effet, `random() % 255` génère des nombres de 0 à 255 ainsi, l'ajout de 1 (`random() % 255)+1`) génère des nombres de 1 à 256.

Question 3 - De quel mécanisme peut-on se servir pour passer l'information du fils au père ?

Pour passer une information du fils au père, on peut se servir du status renvoyé par le fils (avec exit(n)) lors de sa mort, le père pourra le récupérer avec WEXITSTATUS.

Code Source

```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#define TABSIZE 1000

int fouiller(unsigned char tab[], int depart, int fin){
    int i = depart;
    while( i < fin){
        if (tab[i] == 0){
            return 1;
        }
        i++;
    }
    return 0;
}

int main(){

    //Initialisation du tableau
    unsigned char arr[TABSIZE]; //Entier de 0 à 255
    srandom(time(NULL));
    // entasser du foin
    for (int i = 0; i < TABSIZE; i++){
        arr[i] = (unsigned char) (random() % 255) + 1;
    }
    int index;
```

```c
    // cacher l'aiguille
    printf("Enter a number between 0 and %d: ", TABSIZE-1);
    scanf(" %d", &index);
    if (index >= 0 && index < TABSIZE){
        arr[index] = 0;
    }else{
        //Vérification de l'entrée
        while (!(index >= 0 && index < TABSIZE)){
            printf("You must a number between 0 and %d: ", TABSIZE-1);
            scanf(" %d", &index);
        }
        arr[index] = 0;
    }

    pid_t child = fork(); //Création du processus fils

    int rep_pere;
    int rep_fils;
    int status;

    if(child == -1){
        perror("fork() error");
        exit(1);
    }

    if(child > 0){
        //Exécuté par le père
        rep_pere = fouiller(arr, 0, TABSIZE/2);
        wait(&status);
    }else{
        //Exécuté par le fils
        exit(fouiller(arr, TABSIZE/2, TABSIZE));
    }

    rep_fils = WEXITSTATUS(status); //Le père récupére le résultat du fils
    if (rep_pere || rep_fils){
        printf("Needle found ! \n");
    }else {
        printf("No needle found... \n");
    }

    return 0;
}
```

Jeu de test

- *Indice dans la première moitiée* : 250
- *Indice dans la deuxième moitiée* : 750
- *Indice hors de la liste* : 1, 9999 (Géré par la boucle while de vérification)
- *Indice autour de la coupure* : 499, 500, 501

Exercice 2

On modifie le programme de l'exercice précèdent pour qu'il prenne sur la ligne de commande un nombre N entre 1 et 100 qui désigne le nombre de processus entre lesquelles on partage le tableau pour faire la recherche.

Code Source

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdbool.h>
#define TABSIZE 1000

int fouiller(unsigned char tab[], int depart, int fin){
    int i = depart;
    while( i < fin){
        if (tab[i] == 0){
            //printf("Child process: %d has found %d at %d\n", getpid(),
tab[i], i);
            return 1;
        }
        i++;
    }
    return 0;
}

int main(int argc, char *argv[]){

    //Initialisation du tableau
    unsigned char arr[TABSIZE]; //Entier de 0 à 255
    srandom(time(NULL));

    int N = 1; //correspond au nombre de processus allant fouiller le
tableau
    if (argc > 1) N = atoi(argv[1]);
    if (N < 1 || N > 100) N = 1;

    printf("%d will share the task.\n", N);
    // entasser du foin
    for (int i = 0; i < TABSIZE; i++){
        arr[i] = (unsigned char) (random() % 255) + 1;
    }
    int index;
    // cacher l'aiguille
    printf("Enter a number between 0 and %d: ", TABSIZE-1);
    scanf(" %d", &index);
    if (index >= 0 && index < TABSIZE){
        arr[index] = 0;
    }else{
        //Vérification de l'entrée
        while (!(index >= 0 && index < TABSIZE)){
```

```
            printf("You must a number between 0 and %d: ", TABSIZE-1);
            scanf(" %d", &index);
        }
        arr[index] = 0;
    }

    //affiche(arr, TABSIZE);

    int rep_fils[N];
    int status;

    pid_t child = getpid();
    int i = 0;
    while (i < N && child > 0){
        child = fork();
        if(child == -1){
            perror("fork() error");
            exit(1);
        }
        if(child == 0){
            //Exécuté par le fils
            exit(fouiller(arr, (TABSIZE/N)*i, (TABSIZE/N)*i+TABSIZE/N));
        }
        i += 1;
    }

    //Le père attends tout les enfants
    for (i = 0; i < N; i++){
        wait(&status);
        rep_fils[i] = WEXITSTATUS(status);
    }

    i = 0;
    bool found = false;
    while (!found && i<N){
        if (rep_fils[i]){
            printf("Needle found ! \n");
            found = true;
        }
        i += 1;
    }
    if(!found){
        printf("No needle found ! \n");
    }

    return 0;
}
```

Jeu de test

- *Indice dans la nieme moitiée* : (N*n)+x
- *Indice hors de la liste* : 1, 9999 (Géré par la boucle while de vérification)

- *Indice autour de la coupure* : (999/N)+1, (999/N)-1, 999/N, etc...

Exercice 3

Reprenons le programme de l'exercice 2. Supposons qu'un des pro- cessus fils trouve zéro très vite et en avertit le père. Dans ce cas, le père peut afficher la réponse et terminer sans attendre la fin de calcul des autres fils.

Question 1 - Lisez les pages de man correspondantes et expliquez ce que font ces commandes.
`putc('.', stdout); fflush(stdout);`

`putc('.', stdout);` écris un caractère sur la sortie standard, `fflush(stdout);` rafraichi la sortie standard.

Question 2 - Faites plusieurs tests avec le programme modifié. Que constatez-vous ?

Un point s'affiche à chaque parcours de la boucle de recherche, on remarque que lorsqu'un fils à trouvé, lui s'arrête mais les autres continuent leur recherche malgré que l'aiguille est déjà été trouvée.

Voici le code rajouté à cette étape :

```
//Hors du main
void handleSIGTERM(int sig){
    printf("PID %d has terminated \n", getpid());
    exit(0);
}

//Dans le main
for (i = 0; i < N; i++){
        wait(&status);
        rep_fils[i] = WEXITSTATUS(status);
        if (rep_fils[i] == 1){
            kill(0, SIGTERM); //AJOUT : si un fils à trouvé, on envoie un
signal aux autres.
        }
    }

//et un affichage du PID du père et des PIDs fils à la création des
processus fils
```

On remarque que le processus est tué aussi quand le signal est lancé, le programme se termine donc de façon prématuré. Il faudrait vérifier au moment de la gestion du signal que le processus qui va etre tué n'est pas le processus père :

```
//Avant le main
volatile pid_t parentPID; //Pour contenir le PID du père

void handleSIGTERM(int sig){
    if(getppid() == parentPID){ //Verification du PID
        printf("PID %d has stop because of parent's order \n", getpid());
```

```
        exit(0);
    }
}

//Dans le main
parentPID = getpid();
```

Code source

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdbool.h>
#include <signal.h>
#define TABSIZE 1000

volatile pid_t parentPID;

void affiche(unsigned char tab[], int size){
    printf("Affichage du tableau : ");
    for (int i = 0; i < size; i++){
        printf("%d ", tab[i]);
    }
    printf("\n");
}

int fouiller(unsigned char tab[], int depart, int fin){
    int i = depart;
    while( i < fin){
        //putc('.', stdout); fflush(stdout); //Affiche un point (fflush
permet de recharger la sortie), nous verrons s'afficher un point à chaque
parcours de la boucle
        if (tab[i] == 0){
            return 1;
        }
        i++;
    }
    return 0;
}

void handleSIGTERM(int sig){
    if(getppid() == parentPID){
        printf("PID %d has been stop by signal \n", getpid());
        exit(0);
    }
}

int main(int argc, char *argv[]){

    signal(SIGTERM, handleSIGTERM);
```

```
    parentPID = getpid();

    //Initialisation du tableau
    unsigned char arr[TABSIZE]; //Entier de 0 à 255
    srandom(time(NULL));

    int N = 1; //correspond au nombre de processus allant fouiller le
tableau
    if (argc > 1) N = atoi(argv[1]);
    if (N < 1 || N > 100) N = 1;

    printf("%d will share the task.\n", N);
    // entasser du foin
    for (int i = 0; i < TABSIZE; i++){
        arr[i] = (unsigned char) (random() % 255) + 1;
    }
    int index;
    // cacher l'aiguille
    printf("Enter a number between 0 and %d: ", TABSIZE-1);
    scanf(" %d", &index);
    if (index >= 0 && index < TABSIZE){
        arr[index] = 0;
    }else{
        //Vérification de l'entrée
        while (!(index >= 0 && index < TABSIZE)){
            printf("You must a number between 0 and %d: ", TABSIZE-1);
            scanf(" %d", &index);
        }
        arr[index] = 0;
    }

    int rep_fils[N];
    int status;

    printf("Parent : %d \n", getpid());

    pid_t child = getpid();
    int i = 0;
    while (i < N && child > 0){
        child = fork();
        if(child == -1){
            perror("fork() error");
            exit(1);
        }
        if(child == 0){
            //Exécuté par le fils
            printf("Child : %d \n", getpid());
            exit(fouiller(arr, (TABSIZE/N)*i, (TABSIZE/N)*i+TABSIZE/N));
        }
        i += 1;
    }

    //Le père attends tout les enfants
    for (i = 0; i < N; i++){
```

```
            wait(&status);
            rep_fils[i] = WEXITSTATUS(status);
            if (rep_fils[i] == 1){
                kill(0, SIGTERM);
            }
        }

        i = 0;
        bool found = false;
        while (!found && i < N){
            if (rep_fils[i]){
                printf("Needle found ! \n");
                found = true;
            }
            i += 1;
        }
        if(!found){
            printf("No needle found ! \n");
        }

        return 0;
    }

    //L'appel de kill(0, SIGTERM) tue tout les processus du groupe de
    l'appellant, y compris l'appellant lui-même, le programme est donc stoppé.
```