TP n° 9

Exercice 1. *Retour vers le futur*

1. Lapin est en L3IF à l'ENS de Lyon. Pour le cours d'ASR2, on lui demande d'écrire un programme `ex1_matmul` (sans `fork`s ni `pthread`s pour le moment) qui prend en entrée deux matrices sous la forme suivante :

```
2       // Nombre de matrices à suivre (ici toujours 2)
3 3     // taille de la première matrice
1 2 3   // entrées de la première matrice
4 5 6
7 8 9

3 3     // taille de la deuxième matrice
1 4 7   // entrées de la deuxième matrice
2 5 8
3 6 9
```

   et qui renvoie le produit des deux matrices (le traitement des commentaires n'est pas demandé).
   Lapin a rendu le code suivant : `https://ptpb.pw/RwKN/c`
   Ce code possède un problème de fond, qui apparait quand on essaye de multiplier des « grandes » matrices (300 par 300). Déterminez pourquoi ce problème apparait, et corrigez le.

Exercice 2. *For fun and profit*

1.
```c
#include <stdlib.h>
#include <stdio.h>

size_t read_file(const char *path, char *buf, size_t buf_size) {
    size_t i, size_read; FILE *file = fopen(path, "r"); int c;
    if (!file) exit(EXIT_FAILURE);
    size_read = buf_size;

    for (i=0; i<buf_size; i++) {
        if ((c = fgetc(file)) == EOF) {
            size_read = i;
            break;
        }
        buf[i] = (char) c;
    }
    buf[size_read-1] = '\0';

    /* Ligne cachée */

    fclose(file);
    return size_read;
}

void print_file(const char *path) {
```

```
    char foo[] = "abcdefg"; char buffer[10];
    size_t size_read = read_file(path, buffer, 10);
    printf("Read %zu bytes: %s\n", size_read, buffer);
    printf("abcdefg = %s\n", foo);
}

void main(void) {
    print_file("/etc/passwd");
}
```

C'est un programme tout à fait banal, à part qu'un lutin a ajouté une ligne dans le code de `read_file` avant la compilation (compilation pour laquelle il a utilisé gcc possiblement avec une option, elle aussi cachée) et vous ne pouvez malheureusement pas la voir. Quand vous exécutez le code, ceci s'affiche :

```
Read 10 bytes: root:x:0:
abcdefg = ghijklmn
```

Perplexe, vous l'exécutez dans valgrind, mais valgrind ne détecte pas d'erreur. Que s'est-il passé ?

2. Un gnome de passage vous suggère de remplacer `char foo[] = "abcdefg";` par `char *foo = calloc(9, sizeof(char)); strcpy(foo, "abcdefg");`
Cette fois, l'affichage de valgrind est :

```
==26831== Invalid read of size 1
...
==26831==    by 0x4008A5: print_file (file.c:34)
==26831==    by 0x4008DD: main (file.c:39)
==26831==  Address 0x6867666564636261 is not stack'd, malloc'd or (recently) free'd
```

Que remarquez-vous ?

3. Essayez de deviner la ligne ajoutée par le lutin. Il y a de nombreuses réponses possibles, essayez de justifier la ou les vôtres.

Exercice 3. *DAG*

Aujourd'hui nous allons nous intéresser à une problématique primordiale dans un système d'exploitation moderne : l'ordonnancement. Par exemple, on a 5 calculs à exécuter mais seulement 3 processeurs disponible, comment choisir l'ordre d'exécution de ces calculs ? Plus formellement :

- Soit $G = (V, E)$ un graphe orienté sans cycle. V est l'ensemble des tâches et E celui des contraintes de précédence des tâches ;
- On a $n = |V|$ tâches à exécuter $V = \{v_1, \cdots, v_n\}$;
- Pour exécuter la tâche v_i il faut que tous les prédécesseurs de v_i soit terminés.
- Chaque tâche prend $w(v_i)$ unités de temps à s'exécuter (on est ici dans un cas théorique : dans la vraie vie on connait rarement ces valeurs à l'avance, mais c'est très pratique de supposer qu'on les a) ;
- On a p processus sur lesquels on peut exécuter ces tâches.

Pour cela on va utiliser, l'algorithme le plus simple : le *list-scheduling* [1]. L'idée étant de maintenir une liste des tâches prêtes à être exécuter, et de choisir la prochaine tâche à exécuter selon une priorité définie à l'avance.
Comme les temps d'exécutions des tâches sont des entiers, on va procéder à une simulation à événements discrets. Autrement dit, à chaque top de temps on va mettre à jour la liste des tâches prêtes et décider lesquelles on va exécuter et sur quelles ressources (processeurs ici).
Le makespan est alors défini comme le temps nécessaire pour exécuter le chemin critique du graphe G, autrement dit le temps de terminaison de la dernière tâche dans le graphe G.

1. https://en.wikipedia.org/wiki/List_scheduling

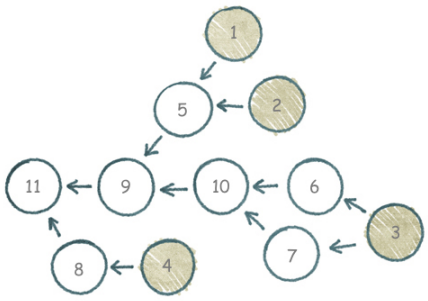

FIGURE 1 – Un exemple de graphe orienté sans cycle

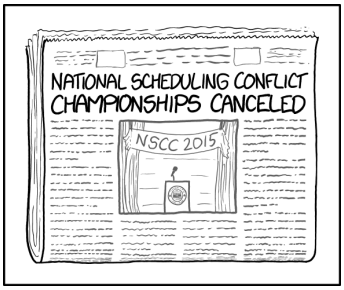

FIGURE 2 – Neither a spokesperson for the organization nor the current world champion could be reached for comment. *Source :* `https://xkcd.com/1542/`.

1. Dans un premier temps on va considérer des tâches indépendantes, on peut exécuter les tâches dans n'importe quel ordre. Plus formellement, $G = (V, \emptyset)$. On utiliser une structure de données `pool` qui va contenir des tâches. Une tâche étant représenté par 2 entiers, son identifiant i et son coût de calcul $w(v_i)$. On va maintenir deux structures `pool` :
   - `ready` qui va contenir la liste des tâches prêtes au top de temps k
   - `inflight` qui contenir la liste des tâches en cours de calcul au top k

   (a) Codez la structure de données `pool`. Sans contraintes de précédences, la liste va contenir toutes les n tâches au premier top de temps $k = 1$ (puisque on peut, en théorie, toutes les choisir).

   (b) Codez une fonction `void update(struct pool *ready, int k)` qui va mettre à jour la liste des tâches prêtes.

   (c) Codez une fonction `int get_next_task(struct pool *ready)` qui va renvoyez le numéro de la prochaine tâche à exécuter. Comme fonction de priorité on va utiliser un ordre aléatoire (parmi les tâches prêtes on choisi aléatoirement une tâche).

   (d) Codez la fonction `int allocate_proc(int *array_proc, int p)` qui choisi un processeur libre et met à jour le tableau `array_proc` pour signifier que ce processeur et occupé (et pour combien de temps, ça peut être utile).

   (e) Implémentez un *list-scheduling*.

2. Maintenant on veut considérer l'ordonnancement de tâches non-indépendantes.

   (a) Codez la représentation du graphe G (matrice ou liste d'adjacence). Pour représenter G vous pouvez utiliser une matrice d'adjacence ou une liste d'adjacence, je vous conseille la liste, plus efficace pour les graphes creux. Vous aurez aussi besoin de coder une fonction qui génère un DAG aléatoirement.

   Pour information, si une matrice d'adjacence M est triangulaire supérieure ou triangulaire inférieure stricte, alors G est acyclique. La réciproque est fausse.

$$\begin{pmatrix} 0 & e_{1,2} & \cdots & e_{1,n} \\ 0 & 0 & \cdots & e_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}$$

   (b) Modifiez votre implémentation du *list-scheduling* pour prendre en compte G.

3. *(Optionnelle)* Est-il possible d'obtenir, avec $G = (V, E)$ un makespan minimal en un temps polynomial pour $p > 2$? Si non, existe-t-il un facteur d'approximation ?

Exercice 4. *Corruption de free()*

1. ```c
   #include <stdlib.h>

   void main(void) {
       char *buf = calloc(5, sizeof(char));
       for (int i=0; i<= 10; i++) buf[i-5] = (char) ('a' + i);
       free(buf);
   }
   ```

   Ce programme devrait afficher une erreur du genre : `Error in ‘./a.out’: double free or corruption (out)` Essayez d'expliquer ce qu'elle signifie, et pourquoi c'est celle-ci qui est déclenchée.

2. Le même programme a un comportement différent quand il est exécuté avec valgrind. Pourquoi ?
   ```