Ordonnancement - Correction

Samy Jaziri

```c
#ifndef ORDONNANCEMENT_H
#define ORDONNANCEMENT_H
#include <stdbool.h>
```

1 Introduction

Préliminaires

On rappel le type de l'énoncé :

```c
typedef unsigned int uint;
typedef struct processus {
    uint id;
    uint cout;
} processus;
```

```c
processus* creer_processus(uint id, uint cout) {
    processus* p = (processus*) malloc(sizeof(processus));
    p->id = id;
    p->cout = cout;
    return p;
}
```

Pas besoin de fonction particulière pour détruire un processus. Un simple `free` suffira.

2 Files d'attente

On rappelle la structure de donnée :

```
typedef struct file {
    processus** contenu;
    uint capacite;
    uint debut;
    uint fin;
    uint charge;
    bool vide;
} file;
```

Pas de difficultés pour les deux premières fonctions :

```
file* creer_file_vide() {
    file* f = (file*) malloc(sizeof(file));
    f->capacite = 0;
    f->debut = 0;
    f->fin = 0;
    f->charge = 0;
    f->vide = true;
    f->contenu = NULL;
    return f;
}
```

```
void detruire_file(file* f) {
    if ( !f->vide )
        for (uint i = f->debut; i < f->fin + f->capacite; ++i)
            free(f->contenu[i % f->capacite]);
    free(f->contenu);
    free(f);
}
```

On met à jour le booléen `vide` si la file devient vide. On pense à gérer le cas où l'indice de fin ou de début est sur la limite du vecteur.

```
processus* retire(file* f) {
    if (f->vide)
        return NULL;
    f->fin = (f->fin + f->capacite - 1) % f->capacite;
    if (f->fin == f->debut)
        f->vide = true;
    f->charge -= f->contenu[f->fin]->cout;
    return f->contenu[f->fin];
}
```

Dans le cas où la capacité maximale n'est pas atteinte, la fonction de pose aucune difficulté. Sinon, pour assurer une complexité amortie constante, on double la taille du vecteur avant d'ajouter l'élément. Lors de la copie des éléments on doit faire attention à ajouter des cases libres *en dehors* de la file. C'est à dire, réarranger les éléments en refaisant partir la file du début du tableau ou insérer les cases vides entre les indice de début et de fin.

```
void ajoute(file* f, processus* p) {
    processus** tmp;
    if ( f->capacite == 0 ) {
        ++f->capacite;
        f->contenu = (processus**) malloc(sizeof(processus*));
    }
    else if ( f->fin == f->debut && !f->vide ) {
        tmp = (processus**) malloc(sizeof(processus*) * f->capacite * 2);
        for (uint i = 0; i < f->fin; ++i)
            tmp[i] = f->contenu[i];
        for (uint i = f->debut; i < f->capacite; ++i)
            tmp[i+f->capacite] = f->contenu[i];
        f->debut += f->capacite;
        f->capacite *= 2;
        free(f->contenu);
        f->contenu = tmp;
    }
    f->contenu[f->fin] = p;
    f->charge += p->cout;
    f->fin = (f->fin + 1) % f->capacite;
    f->vide = false;
}
```

Processuss aléatoire

```
file* F(uint u0, uint n) {
    uint u = u0;
    file* f = creer_file_vide();
    for (uint i = 0; i <= n; ++i) {
        if (u % 21)
            ajoute(f,creer_processus(u, (u % 10) * ( n % 10 + 1 ) ));
        else
            retire(f);
        u = ( 15091 * u ) % 64007;
    }
    return f;
}
```

```
uint taille_file(file* f) {
    return (f->fin + f->capacite) - f->debut;
}
```

```
uint milieu_file(file* f) {
    return f->contenu[(((f->fin + f->capacite) - f->debut) / 2) % f->capacite]->id;
}
```

3 Equilibrage de charge

Q. Soit $\mathcal{P}$ un ensemble de processus et notons $n = C(\mathcal{P})$. Soit une partition A_1, A_2 qui minimise la différence de charge. Notons $a_1 = C(A_1)$, $a_2 = C(A_2)$ Soit une partition B_1, B_2. Notons $b_1 = C(B_1)$, $b_2 = C(B_2)$. Supposons sans perte de généralité $a_1 \geq a_2$ et $b_1 \geq b2$.

$$b_1 + b_2 = a_1 + a_2 = n$$

Donc : $b_2 = n - b_1$
$a_2 = n - a_1$

Comme A_1, A_2 est optimale : b_1 - b_2 $\geq$ a_1 -a_2
2\,b_1 - n $\geq$ 2\,a_1 - n
b_1 $\geq$ a_1
*align}
Donc pour A_1, A_2 minimise aussi le maximum des charges. L'implication inverse se montre de la même fa'c'on.
Q. $\mu(0, 0)$ est la différence de charge de la partition optimale d'un ensemble de $\mathcal{P}$. $\mu(n, \Delta) = \Delta$.
Q. On remarque que

$$\mu(k, \Delta) = \min(\mu(k + 1, \Delta + r_{k+1}), \mu(k + 1, |\Delta - r_{k+1}|))$$

où r_{k+1} est le coût du processus p_{k+1}. Le premier cas correspond à une répartition où p_{k+1} est exécuté par le processeur qui avait déjà le plus de charge après la répartition des k premiers processus. Le second cas correspond à une répartition où p_{k+1} est exécuté par le processeur ayant le moins de charge.
Q. L'algorithme récursif utiliserai la formule de récurrence si dessus pour faire un calcul récursif de $\mu(k + 1, \Delta + r_{k+1})$ et $\mu(k + 1, |\Delta - r_{k+1}|)$. Le cas de base étant $\mu(n, \Delta')$. On aurait une complexité lors en $\mathcal{O}(2^n)$.

```
uint calcul_mu(file* p, uint indice, uint difference_charge, int** memoization) {
    uint cout = p->contenu[(p->fin+p->capacite-indice-1) % p->capacite]->cout;
    uint diff1 = cout + difference_charge;
    uint diff2 = cout < difference_charge ?
        difference_charge - cout : cout - difference_charge;
    int mu_diff1 = memoization[indice+1][diff1] ,
        mu_diff2 = memoization[indice+1][diff2];
    if(mu_diff1 == -1) {
        mu_diff1 = calcul_mu(p, indice+1, diff1, memoization);
    }
    if(mu_diff2 == -1) {
        mu_diff2 = calcul_mu(p, indice+1, diff2, memoization);
    }
    memoization[indice][difference_charge] =
        mu_diff1 < mu_diff2 ? mu_diff1 : mu_diff2;
    return memoization[indice][difference_charge];
}
```

```
uint difference_optimale(file* p) {
    if (p->vide)
        return 0;
    uint nb_processus = taille_file(p);
    uint charge_maximale = p->charge;
    int** mu = (int**) malloc(sizeof(int*) * (nb_processus + 1));
    for (uint i = 0; i < nb_processus; ++i) {
        mu[i] = (int*) malloc(sizeof(int) * ( charge_maximale + 1)) ;
        for (uint j = 0; j <= charge_maximale; ++j)
            mu[i][j] = -1;
    }
    mu[nb_processus] = (int*) malloc(sizeof(int) * ( charge_maximale + 1 ));
    for (uint j = 0; j <= charge_maximale; ++j) {
        mu[nb_processus][j] = j;
    }
    return calcul_mu(p,0,0,mu);
}
```

La complexité spatiale de l'algorithme est en grand $\mathcal{O}$ de la taille du tableau `mu`, c'est à dire en $\mathcal{O}(|\mathtt{p}|\mathcal{C}(p))$. La complexité temporelle est plus compliquée à calculer. Chaque appel de la fonction `calcul_mu` réalise des opérations en temps constant. Il reste à calculer le nombre de nœuds de l'arbre des appels récursifs de la fonction `calcul_mu`. On sait que la hauteur de l'arbre est bornée par le nombre de processus, $|{p}|$. On cherche à borner le nombre de nœuds à chaque niveau de l'arbre. Un niveau de l'arbre correspond aux calculs des $\mu(i, \Delta)$ pour i un indice fixé et $0 \leq \Delta \leq \mathcal{C}(\mathtt{p})$. L'utilisation d'une technique de mémoïzation assure que le calcul de $\mu(i,\Delta)$ est fait une seul fois, *i.e* que la fonction calcul_mu est appelée une seule fois récursivement sur les paramètres i et \Delta (p et memoization sont toujours

les mêmes dans l'exécution). Donc le nombre de nœud au niveau i est borné par $\mathcal{C}(\mathrm{p})+1$. Le nombre de nœud est donc borné par $\mathcal{O}(|\mathrm{p}|\,\mathcal{C}(p))$. Ce qui correspond à la complexité temporelle au pire de l'algorithme.
Q. Pour l'équilibrage, on rempli la matrice mu avec la fonction calcul_mu puis on choisi reparti le processus i en regardant la valeur minimale de μ à l'indice $i+1$, selon qu'on place le processus dans a1 ou a2. mu étant déjà calculé, la complexité de la répartition est linéaire en la taille de la file p, ce qui n'augmente pas la complexité au pire dejà obtenu pour le calcul du \mu.

```
void equilibrage(file* p, file* a1, file* a2) {
    if (p->vide)
        return;
    uint nb_processus = taille_file(p);
    uint difference_charge = a1->charge < a2->charge ?
        a2->charge - a1->charge : a1->charge - a2->charge;
    uint charge_maximale = p->charge + difference_charge;
    uint cout, diff1, diff2;
    file* f_tmp;
    processus* p_tmp;
    int mu_diff1, mu_diff2;
    int** mu = (int**) malloc(sizeof(int*) * ( nb_processus + 1 ));
    for (uint i = 0; i < nb_processus; ++i) {
        mu[i] = (int*) malloc(sizeof(int) * (charge_maximale+1));
        for (uint j = 0; j <= charge_maximale; ++j)
            mu[i][j] = -1;
    }
    mu[nb_processus] = (int*) malloc(sizeof(int) * ( charge_maximale + 1 ));
    for (uint j = 0; j <= charge_maximale; ++j) {
        mu[nb_processus][j] = j;
    }

    calcul_mu(p,0,difference_charge,mu);

    for (uint indice = 0; !p->vide; ++indice) {
        p_tmp = retire(p);
        cout = p_tmp->cout;
        diff1 = cout + difference_charge;
        diff2 = cout < difference_charge ?
            difference_charge - cout : cout - difference_charge;
        mu_diff1 = mu[indice+1][diff1];
        mu_diff2 = mu[indice+1][diff2];
        if(mu_diff1 < mu_diff2) {
            ajoute(a1,p_tmp);
            difference_charge = diff1;
        }
        else {
            ajoute(a2,p_tmp);
            difference_charge = diff2;
        }
        if(a1->charge < a2->charge) {
            f_tmp = a2;
            a2 = a1;
            a1 = f_tmp;
        }
    }
}
```

4 Affichage des résultats de la fiche réponse

```c
#include <stdio.h>
#include <stdlib.h>
#include "ordonnancement.h"

uint somme_id(file* f) {
    uint taille = taille_file(f);
    uint somme = 0;
    for (uint i = 0; i < taille ; ++i) {
        somme += f->contenu[(f->debut + i) % f->capacite]->cout;
        somme %= 12000;
    }
    return somme;
}
```

```
int main( int argc, char** argv ) {
    if (argc != 2) {
        fprintf(stderr, "Usage : %s u0\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    int u0 = atoi(argv[1]);
    file* f1212 = F(u0, 1212);
    file* f1000 = F(u0, 1000);
    file* f1256 = F(u0, 1256);
    printf("Question 4 : a) (%i, %i, %i)  b) (%i, %i, %i)  c) (%i, %i, %i)\n",
           f1212->capacite, taille_file(f1212), milieu_file(f1212),
           f1000->capacite, taille_file(f1000), milieu_file(f1000),
           f1256->capacite, taille_file(f1256), milieu_file(f1256));
    printf("Question 6 : a) %i  b) %i  c) %i\n",
           difference_optimale(f1212),
           difference_optimale(f1000),
           difference_optimale(f1256));
    file* a1_1212 = creer_file_vide();
    file* a2_1212 = creer_file_vide();
    file* a1_1000 = creer_file_vide();
    file* a2_1000 = creer_file_vide();
    file* a1_1256 = creer_file_vide();
    file* a2_1256 = creer_file_vide();
    equilibrage(f1212,a1_1212,a2_1212);
    equilibrage(f1000,a1_1000,a2_1000);
    equilibrage(f1256,a1_1256,a2_1256);
    printf("Question 8 : a) (%i, %i)  b) (%i, %i)  c) (%i, %i)\n",
           somme_id(a1_1212), somme_id(a2_1212),
           somme_id(a1_1000), somme_id(a2_1000),
           somme_id(a1_1256), somme_id(a2_1256));
```

```
    detruire_file(f1212);
    detruire_file(f1000);
    detruire_file(f1256);
    detruire_file(a1_1212);
    detruire_file(a2_1212);
    detruire_file(a1_1000);
    detruire_file(a2_1000);
    detruire_file(a1_1256);
    detruire_file(a2_1256);
    return EXIT_SUCCESS;
}
```