# Projet OS

Réalisé par: Hamza BEQQI

# Table of Contents

Table of Contents ..... ii

List of Figures ..... iv

1 Introduction ..... 2

2 Architecture du système ..... 2

2.1 Description ..... 2

2.2 Modélisation ..... 3

3 Gestion de mémoire ..... 3

3.1 Pagination et virtualisation ..... 3

3.1.1 Structures données ..... 3

3.1.2 Get_page() ..... 4

3.1.3 release_page() ..... 4

3.1.4 read_addr() / write_addr() ..... 4

3.1.5 swap_in() ..... 4

3.1.6 swap_out() ..... 4

3.1.7 Algorithmes de déchargement de pages ..... 4

3.2 Allocation/Libération de mémoire ..... 5

3.2.1 My_malloc() ..... 5

3.2.2 my_free() ..... 5

4 Communication et Parallélisme ..... 6

4.1 Parallélisme ..... 6

4.2 Communication entre l'OS et les Pi ..... 6

4.3 Protocole réseau ..... 6

4.3.1 *NetListener* ..... 6

4.3.2 *NetResponder* ..... 6

5 Création des images et stéganographie ..... 7

6 Testbench ..... 7

7 Conclusions et améliorations..... 10

# List of Figures

Figure 1: architecture du système ..... 3

Figure 2: structure d'une requête distante ..... 6

Figure 3: structure d'une réponse réseau ..... 6

Figure 4: Code de P1 et P2 ..... 8

Figure 5: mémoire P1 ..... 8

Figure 6: mémoire P1 (2) ..... 9

Figure 7: mémoire P1(3) ..... 9

Figure 8: mémoire système (1) ..... 9

Figure 9: mémoire système (2) ..... 9

Figure 10: mémoire de P1 (4) ..... 10

Figure 11: mémoire de P1 (5) ..... 10

# 1 Introduction

Durant ce projet, on a créé un système de gestion de mémoire (allocation, pagination et virtualisation) pour plusieurs processus. Le système sert des requêtes TCP en retournant des images de l'état de mémoire du système ou d'un processus. Le rapport suivant présente le travail qu'on a effectué. On commencera par une description globale de l'architecture du système et de son fonctionnement, après on décrira comment l'allocation, la pagination et la virtualisation des pages ont été implémenté. Puis, on va parler de la communication entre les différentes parties du système (entre les processus, entre les processus et l'OS,...) et comment le parallélisme a été implémenté. Et avant de conclure, on va parler de comment les images TARGA (servies par le système à travers le réseau) sont créés.

# 2 Architecture du système

## 2.1 Description

Il s'agit d'un système ayant plusieurs processus Pi avec i ∈ 1..N, lesquels font des requêtes mémoire (allocation, lecture/écriture/libération). Le système d'exploitation (OS) gère la mémoire, en offrant deux fonctions de base : l'allocation/libération de zones mémoires pour un Pi et l'allocation/libération de pages dans le système. Le système supporte la virtualisation des pages mémoires, et l'OS effectue des opérations de type chargement déchargement de pages. Le système sert des requêtes sur socket TCP en retournant des images de l'état mémoire, soit par processus (l'espace d'adressage apparait linéaire), soit global (l'attribution et l'état des pages). Dans le premier cas, l'image renvoyée fait apparaitre les zones libres et occupées par des couleurs différentes, et les pages non présentes en mémoire sont plus sombres. Dans le deuxième cas, chaque processus possède sa couleur, et chaque page est coloriée en fonction de son processus (noir si non affectée).

L'ensemble des Pi et l'OS sont codés sous la forme de threads. L'OS également. L'espace d'adressage des processus est vide à l'origine et est modulé en fonction des demandes. Les primitives de réservation/libération mémoire sont my_malloc() et my_free(). Les primitives de swap sont swap_in() et swap_out(). Les primitives d'allocation/libération de pages sont get_page() et release_page().

## 2.2 Modélisation

La figure suivante décrit l'architecture du système qui a été implémenté :

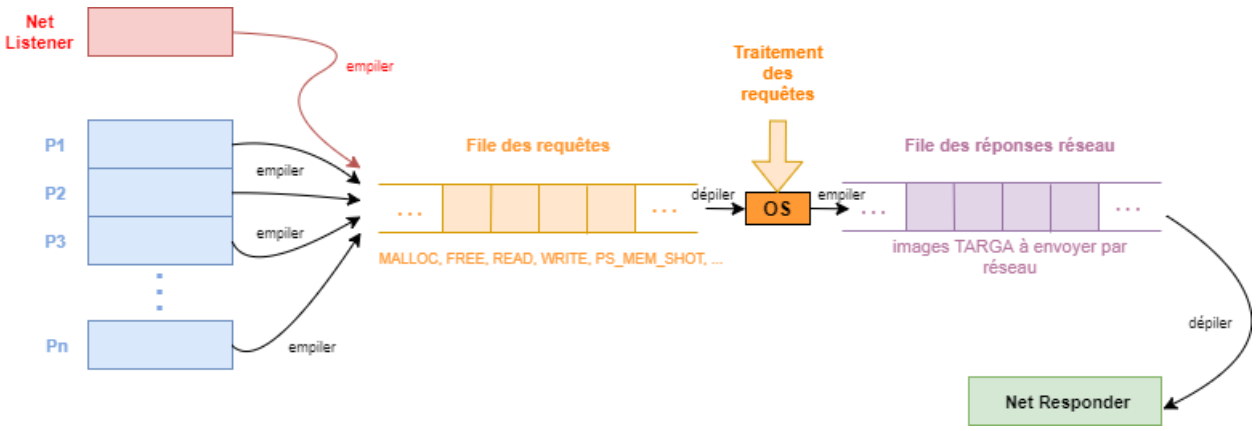


**Figure 1: architecture du système**

Le thread OS traite les requêtes qu'il reçoit à partir d'une file de requêtes. Cette dernière est remplie par les Pi qui demandent des opérations mémoires (malloc, free, read, write) et le NetListener qui se charge d'accepter les connexions réseau et de demander à l'OS de générer des images TARGA de l'état de mémoire que ça soit d'un processus ou du système lui-même. Après, l'OS retourne le résultat d'une requête locale (adresse mémoire) au Pi (partie qui sera détaillé davantage en dessous), et empile le résultat d'une requête réseau dans la file 'réponses réseau'. Le NetResponder dépile de cette dernière, et se charge d'envoyer la réponse au client.

# 3 Gestion de mémoire

## 3.1 Pagination et virtualisation

### 3.1.1 Structures données

Le système définit la taille d'une page, le nombre maximal de pages dans la mémoire physique du système (qui définit automatiquement la taille maximale de la mémoire physique de système) et le nombre maximale de pages virtuelles qu'un processus peut adresser.

L'OS garde l'état des frames (pages physiques) dans un tableau statique de taille maximale. L'état d'un frame se compose principalement du nombre de la page associée, l'identifiant du processus auquel appartient la page. L'état des pages est stocké dans un autre

tableau statique qui se compose principalement de booléens qui représentent si la page est libre, si elle en mémoire ou sur le disque dur, et du nombre de la frame associée. Ce tableau fait partie de l'état de chaque processus.

### 3.1.2 Get_page()

La fonction get_page se charge d'allouer une page. Il reçoit l'identifiant d'un processus en paramètre et retourne le nombre de la page allouée. La fonction fait une recherche linéaire dans la page table du processus pour trouver une page libre, et retourne son numéro.

### 3.1.3 release_page()

La fonction release_page se charge de libérer une page. Il reçoit en paramètre le numéro du processus et de la page concernés.

### 3.1.4 read_addr() / write_addr()

Ces fonctions permettent de toucher à une adresse particulière d'un processus. Elles identifient la page qui contient l'adresse en question, si elle est présente en mémoire, tant mieux, sinon un appel vers swap_in() est lancé.

### 3.1.5 swap_in()

Cette fonction permet d'allouer une frame à une page d'un processus. Elle reçoit le processus et la page en question. Elle effectue une recherche linéaire sur une frame libre et fait une mise à jour de la page table du processus. Si aucune frame n'est libre un appel vers swap_out() est lancé.

### 3.1.6 swap_out()

Cette fonction décharge une page de la mémoire physique et retourne le nombre de la frame qui lui a été associé. Le choix de la page à décharger se fait par deux algorithmes: LRU(Least Recently Used) ou vieillissement.

### 3.1.7 Algorithmes de déchargement de pages

- LRU: Cet algorithme a été implémenté en gardant, pour chaque frame, un nombre *last_used* qui indique la dernière instruction qui a fait référence à la page en question

(On garde en mémoire un compteur d'instructions d'accès mémoire effectuées). Lors du swap_out(), la page qui a le *last_used* minimal.

- Vieillissement: Cet algorithme repose sur un bit *referenced*, qui est mis à 1 si la page a été touchée depuis le dernier *tick* d'une horloge. À chaque *tick* de cette dernière, l'os reçoit une requête MMU_TICK, qui lui permet de remettre le bit *referenced* à zéro. Mais avant ceci, un compteur *ref_counter* est décaler à droite de 1bit et la valeur du bit *referenced* est mis dans le bit le plus fort. Le *ref_counter* permet de privilégier les pages qui ont été récemment utilisées.

## 3.2 Allocation/Libération de mémoire

L'OS implémente aussi un allocateur général de mémoire en gardant dans l'état d'un processus une liste chainée de block mémoire vide et une autre pour les bloques qui sont utilisés. Chaque block est représenté par une adresse et une taille.

### 3.2.1 My_malloc()

Cette fonction reçoit en paramètre l'identifiant du processus et le nombre d'octets à allouer. Elle fait appel à *get_page* pour acquérir la mémoire puis opère sur les liste chainées de bloques vides et utilisés. Une première recherche d'un bloque vide à utiliser est effectué afin de maximiser l'utilisation des pages allouées. Le choix est fait selon deux stratégies :

- *Best-fit* : le bloque de taille minimale entre les bloques candidats est choisi.
- *Worst-fit* : le bloque de taille maximale entre les bloques candidats est choisi.

### 3.2.2 my_free()

Cette fonction transforme un block utilisé en un block vide. Elle effectue à la fin une fusion entre les blocks vides adjacents et libère les pages non utilisées en faisant appel à *release_page*.

# 4 Communication et Parallélisme

## 4.1 Parallélisme

Le système implémente 3 threads au-dessus des Pi: Un pour l'OS, le deuxième pour le *NetListener* et le troisième pour le *NetResponder*. Le système contient deux motifs de parallélisme bien reconnus. Un Plusieurs-Producteurs-Un-Consommateur pour la communication entre les Pi, le *NetListener* et l'OS, et un Un-Consommateur-Un-Producteur pour la communication entre l'OS et le *NetResponder*. Les deux sont implémentés avec un sémaphore qui signale l'existence de requêtes à consommer, et un *mutex* qui contrôle l'accès à la file (lors de l'ajout/suppression des entrés).

## 4.2 Communication entre l'OS et les Pi

Lors d'un malloc, l'OS doit retourner une adresse au processus. Ce mécanisme a été implémenté à l'aide d'un sémaphore qui bloque le processus jusqu'à ce que l'OS alloue la mémoire et met l'adresse à retourner dans un pointeur que le processus a fourni.

## 4.3 Protocole réseau

### 4.3.1 *NetListener*

Le *NetListener* reçoit des requêtes TCP sous la forme suivante:

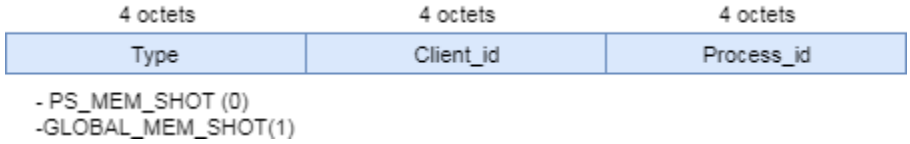


**Figure 2: structure d'une requête distante**

### 4.3.2 *NetResponder*

Le *NetResponder* répond avec des requêtes TCP sous la forme suivante:


**Figure 3: structure d'une réponse réseau**

# 5 Création des images et stéganographie

La création des images se fait en dessinant des rectangles dans un tampon qui représente l'image. Après, les informations suivantes: type de la requête, l'identifiant du client et l'identifiant du processus sont stockés dans les bits de poids faible de chaque couleur. Le type de la requête est représenté par un bit (1 pour *GLOBAL_MEM_SHOT*, 0 pour *PS_MEM_SHOT*). L'identifiant du client et du processus sont stocké sur 8bit chacun.

Un programme qui permet d'extraire ces informations à partir d'une image produite par le système a été créé.

[TODO: IMAGE]

# 6 Testbench

Le scénario de test contient deux processus qui s'exécutent sur un système avec des pages de taille 4Ko et deux frames physiques de même taille, avec LRU comme algorithme de choix de pages à décharger et Best-fit comme algorithme de choix du bloque vide où toute nouvelle allocation devra être faite.

Le code à exécuter par les deux processus est illustré sur les deux figures suivantes :

```
void* P1(void* d)
{
    int ps = 0;
    int ptr1 = my_malloc(ps, 3000);
    int ptr2 = my_malloc(ps, 3000);
    int ptr3 = my_malloc(ps, 2000);
    sleep(1);
    int ptr4 = my_malloc(ps, 1000);
    for (int i = 0; i<10; i++)
    {
        if (i % 2)
            read_addr(ps, ptr1);
        else
        {
            read_addr(ps, ptr3);
            read_addr(ps, ptr3);
        }
    }
    sleep(2);
    my_free(ps, ptr4);
    sleep(2);
    my_free(ps, ptr1);
}
```

```
void* P2(void* d)
{
    int ps = 1;
    int ptr1 = my_malloc(ps, 5000);
    int ptr2 = my_malloc(ps, 3000);
    int ptr3 = my_malloc(ps, 2024);
    sleep(3);
    for (int i = 0; i < 20; i++)
    {
        if (i % 2)
            read_addr(ps, ptr3);
        else
        {
            read_addr(ps, ptr1);
            read_addr(ps, ptr1);
        }
    }
    sleep(2);
    read_addr(ps, ptr2);
    sleep(2);
    my_free(ps, ptr1);
    my_free(ps, ptr2);
}
```

**Figure 4: Code de P1 et P2**

Les deux threads P1 et P2 sont lancés en même temps avec des *sleep* pour s'assurer de l'ordre d'exécution. Trois clients lancent des requêtes périodiquement pour avoir des images représentant la mémoire des deux processus et de la mémoire du système.

P1 effectue 3 allocations de 3000 octets, 3000 octets et 2000 octets respectivement. La mémoire de P1 après ces allocations est montrée sur la figure suivante :


**Figure 5: mémoire P1**

Après une allocation de 1000 octets est effectué afin de tester la stratégie Best-fit de choix de bloque vide. L'algorithme Best-fit choisi le bloque minimal entre les bloques vide qui peuvent contenir l'espace a alloué. C'est ce que montre la figure 6. On peut remarquer que les pages sont

sombres, cela revient au faite qu'elles ne sont pas encore chargées en mémoire physique qui ne se fait pas jusqu'à ce qu'on écrit ou on lit à partir d'une adresse qui appartient à la page.


**Figure 6: mémoire P1 (2)**

Après les instructions de lecture, les deux pages concernées sont chargé en mémoire. La figure 7 représente la mémoire de P1 et la figure 8 représente la mémoire du système à ce moment.


**Figure 7: mémoire P1(3)**

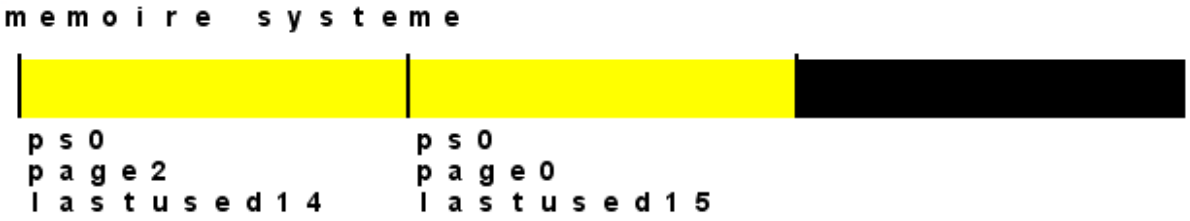


**Figure 8: mémoire système (1)**

Après, P2 se réveille et lit à partir de la première et la troisième page. Ce qui force le système à décharger une page de la mémoire physique suivant l'algorithme LRU (qui décharge la frame avec la valeur *last_used* minimale). La figure 9 montre l'état de mémoire du système après cette opération.

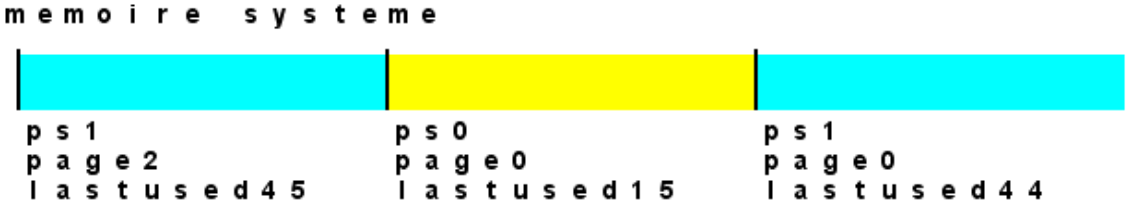


**Figure 9: mémoire système (2)**

P1 continue en libérant la mémoire pointé par *ptr4* suivi de la libération de la mémoire pointé par *ptr1*. La figure 10 montre la mémoire de P1 après la première opération, et la figure 11 après la deuxième. On remarque que la première page a été marqué comme libre.


**Figure 10: mémoire de P1 (4)**


**Figure 11: mémoire de P1 (5)**

# 7 Conclusions et améliorations

Le projet implémenté répond parfaitement au cahier des charges. Il implémente un système de gestion de mémoire (pagination, virtualisation, allocation,…) et répond à des requêtes TCP en envoyant des images TARGA représentant des états de mémoire. Par contre, le projet est loin d'être parfait. Afin d'avoir un système avec une bonne qualité de production, un changement des structures de données utilisées, pour stocker l'état des différents objets, est nécessaire afin de gagner en espace et en vitesse. Par exemple, l'utilisation des tables de *hashage* pour représenter l'état des frames et pages est souvent implémentée pour accélérer le *mapping* entre les pages et les frames. Un autre exemple est l'utilisation des arbres binaires pour accélérer la recherche d'un block mémoire ou encore des listes doublement chainées pour améliorer le mécanisme de fusion de bloques vide adjacents.