Test PCR : Programmation C et Réseau

Nicolas Bailluet et Rémi Hutin

Examen final du 13 décembre 2022 (2 h)

Tous documents interdits à l'exception d'un A4 recto manuscrit de votre main. Calculatrices, téléphones, et autres objets connectés interdits. La correction tiendra compte de la qualité de l'argumentaire et de la présentation. Un code illisible ou incompréhensible est un code faux. 1 point de barème ≈ 6 minutes.

Partie 1 : Programmation en C (11 points).

★ Exercice 1 : ROT13 (3 points).

Le ROT13 est un algorithme de chiffrement simpliste basé sur le chiffrement de César. Il consiste à effectuer une rotation de 13 places dans l'alphabet pour chaque caractère de la chaîne que l'on veut (dé)chiffrer. Ainsi on obtient la table de rotation suivante :

Avant rotation	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Après rotation	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

Implémentez un programme complet en C effectuant le chiffrement. Il devra répondre aux attentes suivantes :

- la chaîne à chiffrer doit être lue depuis les paramètres de ligne de commande (i.e `argv`),
- seuls les caractères compris dans les intervalles `a-z` et `A-Z` doivent être chiffrés, les autres restent inchangés,
- la chaîne chiffrée est affichée (via `printf` par exemple).

Ainsi, si on exécute votre programme, on désire obtenir ce type de sortie dans le terminal :

```
> ./rot13 "Le C c'est vraiment un super langage, j'adore !"
Yr P p'rfg ienvzrag ha fhcre ynatntr, w'nqber !
```

★ Exercice 2 : Malloc - *Back to the 90s* (8 points).

On souhaite développer en C sur une architecture embarquée exotique. On se rend compte rapidement qu'il y a un souci... on ne peut pas utiliser la `libc` ! Par conséquent, on n'a pas accès aux fonctions d'allocation de mémoire dynamique (telles que *malloc/free*).

On décide donc d'implémenter notre propre système d'allocation de mémoire dynamique. On suppose que l'on dispose d'un tableau `memory` de taille N défini comme ceci :

```
char memory[N];
```

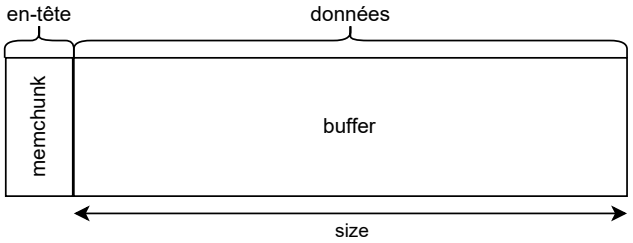
Ce tableau représente l'espace mémoire disponible que l'on peut utiliser pour nos allocations dynamiques.

Afin de gérer les régions allouées et libres de notre mémoire on dispose de deux structures :

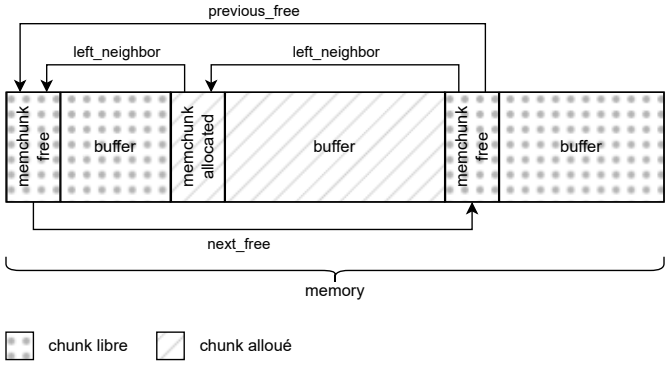
```
struct memchunk {
    bool free;                      // true si le chunk est libre, false si alloué
    struct memchunk* left_neighbor; // chunk adjacent à gauche
    unsigned size;                  // taille
    struct memchunk* previous_free; // chunk précédent dans la freelist
    struct memchunk* next_free:     // chunk suivant dans la freelist
}

struct heap {
    struct memchunk* freelist_head; // chunk en tête de la freelist
}
```

Pour gérer les régions allouées et libres de `memory`, on va compartimenter le tableau en *chunks*. Un *chunk* est constitué d'une en-tête (voir `struct memchunk`) suivi d'un buffer de `size` octets (`size` étant la taille donnée dans l'en-tête).



Ainsi, on va découper `memory` en plusieurs *chunks*. Pour un *chunk* donné, on va garder trace du *chunk "adjacent à gauche"* dans le champ `left_neighbor`. La position du *chunk "adjacent à droite"* peut être calculée grâce à `size`.



Afin de garder en mémoire quels *chunk* sont libres (et donc disponible pour allocation), on utilise une liste doublement chaînée. Le champ `freelist_head` pointe vers le premier *chunk* de la liste. Lorsqu'un *chunk* est libre, `previous_free` et `next_free` pointent respectivement vers le *chunk* précédent et suivant dans la *freelist* (lorsque le *chunk* est alloué, ces deux champs sont inutilisés).

▷ **Question 1 : Initialisation d'un *chunk*.** Écrivez une fonction `init_chunk` pour initialiser l'en-tête d'un *chunk*. On passera en argument sa taille et le pointeur vers son voisin de gauche. Par défaut, `free` vaut `true` et les pointeurs `previous_free` et `next_free` sont nuls.

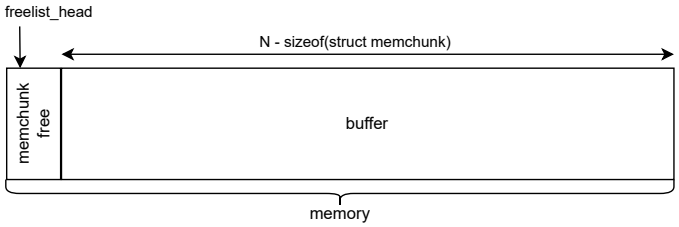
▷ **Question 2 : Insertion dans la freelist.** Écrivez une fonction `insert_freelist` qui insère un *chunk* en tête de la liste chaînée. On donne le prototype de la fonction :

```
void insert_freelist(struct heap* heap, struct memchunk* chunk)
```

Vous veillerez a bien mettre à jour les champs `previous_free` et `next_free` (pour `heap->freelist_head` et `chunk`).

▷ **Question 3 : Initialisation de la heap.** Pour initialiser notre *heap*, on procède comme suit :

1. on initialise un *chunk* de taille `N - sizeof(struct memchunk)` à l'indice 0 de `memory` ;
2. on insère le *chunk* nouvellement initialisé dans la *freelist* vide.



On rappelle que l'on peut obtenir un pointeur vers l'octet d'indice `n` dans `memory` de cette manière :

```
struct memchunk* chunk = (struct memchunk*)&memory[n];
```

Écrivez une fonction `init_heap` qui initialise la *heap* comme décrit ci-dessus.

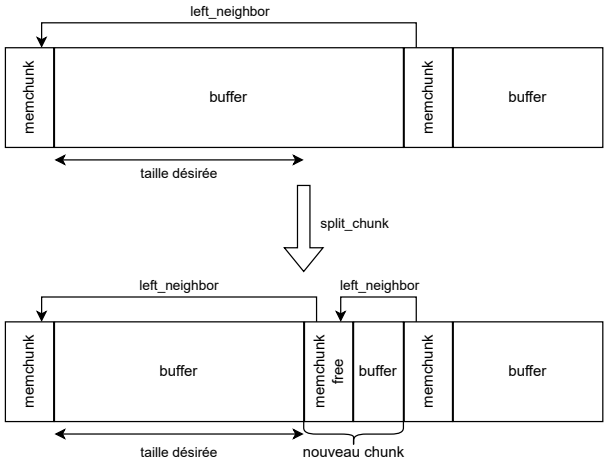
▷ **Question 4 : Recherche d'un chunk libre.** Lors d'une allocation, on veut pouvoir trouver un *chunk* libre dans la *freelist* assez grand pour contenir nos données.

Écrivez une fonction `find_freechunk` qui prend en argument une taille d'allocation et qui retourne le premier *chunk* de taille suffisante dans la *freelist* ou `NULL` s'il n'y en a pas.

▷ **Question 5 : Unlink.** Lorsqu'on trouve un *chunk* assez grand pour notre allocation, il faut le retirer de la *freelist*.
Écrivez une fonction `unlink_chunk` qui retire un *chunk* de la *freelist*. Vous veillerez a bien mettre à jour le champ `previous_free` du *chunk* suivant et `next_free` du *chunk* précédent.

▷ **Question 6 : Découpage de chunk.** Lorsque l'on trouve un *chunk* assez grand pour notre allocation, on veut pouvoir utiliser l'espace supplémentaire en fin de *chunk* pour de potentielles futures allocations. Pour cela on va découper le *chunk* trouvé en deux. On procède comme ceci :

- si l'espace restant est suffisant, on change la `size` du *chunk* trouvé, et on initialise un nouveau `memchunk` adjacent à droite du *chunk* trouvé ;
- sinon on ne découpe pas.



Écrivez d'abord une fonction `right_neighbor(struct memchunk* chunk)` qui renvoie un pointeur vers le *chunk* "adjacent à droite", on calcule sa position grâce à `chunk->size`.

Ensuite, écrivez une fonction `split_chunk` qui découpe un *chunk* en deux en suivant la procédure donnée précédemment et renvoie un éventuel pointeur vers le nouveau *chunk* créé.
Vous veillerez à bien initialiser le `left_neighbor` du nouveau *chunk* "ajdacent à droite" et à mettre à jour celui du précédent *chunk* "adjacent à droite".

▷ **Question 7 : Allocation.** Nous avons maintenant tous les outils pour effectuer une allocation ! On procède comme ceci :

1. on cherche un *chunk* assez grand dans la *freelist* ;
2. on le retire de la *freelist* et met à jour le champ `free` ;
3. on découpe le *chunk* ;
4. on ajoute l'éventuel *chunk* créé lors de la découpe dans la *freelist*.

Écrivez une fonction `alloc(struct heap* heap, unsigned size)` qui effectue une allocation et retourne un pointeur vers le *chunk* alloué ou `NULL` s'il n'y a pas de *chunk* assez grand.

Les questions ci-dessous sont optionnelles, à ne faire que si le reste du sujet à été traité.

▷ **Question 8 : (Bonus) Free naïf.** On veut pouvoir libérer un *chunk* alloué et le réinsérer dans la *freelist*. Pour cela, on peut simplement utiliser la fonction `insert_freelist`.

Considérons alors la séquence d'allocations et libérations suivante :

1. allocation de `chunk1` de taille `N/2 - sizeof(struct memchunk)` ;
2. allocation de `chunk2` de taille `N/2 - sizeof(struct memchunk)` ;
3. libération de `chunk1` et de `chunk2` ;
4. tentative d'allocation de `chunk3` de taille `N - sizeof(struct memchunk)`

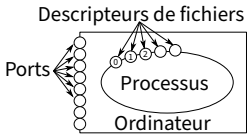
La dernière allocation échoue, expliquez pourquoi (un diagramme peut aider) et proposez une éventuelle solution.

▷ **Question 9 : (Bonus) Free intelligent.** Écrivez une fonction `merge_chunks` qui prend en paramètre un *chunk* et le fusionne avec les *chunks* adjacents si possible. Vous veillerez à bien mettre à jour la *freelist* et les champs `left_neighbor` de tous *chunks* impactés.

Partie 2 : Réseau (9 points).

★ **Exercice 3 : Sockets et connexion TCP (2 points).** ***Les questions sont indépendantes.***

On rappelle le formalisme vu en cours pour représenter des processus en cours d'exécution, repris ci-contre. Les machines sont représentées par des rectangles, avec des petits ronds sur le coté pour représenter les ports de la machine. Au sein des machines, chaque processus est représenté par une ellipse, et des petits ronds à la bordure de cette ellipse représentent les différents descripteurs de fichier du processus. Les trois premiers sont habituellement `stdin`, `stdout`, `stderr`.



▷ **Question 1 :** Dessinez en respectant ce formalisme les différentes étapes de l'établissement d'une connexion TCP entre deux processus distants. Vous donnerez les appels systèmes utilisés par chaque partie. Vous dessinerez l'état des choses après chaque appel système, en ajoutant les explications adéquates. En revanche, il ne vous est pas demandé de donner le code exact réalisant ces appels systèmes.

▷ **Question 2 :** On considère 2 machines A et B en connexion directe avec un débit de 1 Gbit/s. Une connexion TCP est établie entre A et B, et A envoie un fichier gigantesque à B. On suppose que A peut envoyer les données venant de l'application à un débit de 100 Mbit/s. En revanche, B ne peut traiter les données de son buffer de reception TCP qu'à un débit de 50 Mbit/s. Décrivez et expliquez le mécanisme de TCP permettant de ne pas surcharger le recepteur B.

★ **Exercice 4 : Contrôle de congestion TCP (3 points).**

Dans cet exercice, on va considérer une version simplifiée du contrôle de congestion de TCP. Les tailles des fenêtres de congestion seront mesurées en nombre paquets (et non en octets). On va supposer qu'on sera toujours en mode *congestion avoidance*, qui est en *Additive Increase, Multiplicative Decrease* :

— en *Additive Increase*, la fenêtre de congestion augmente de 1 par *RTT* (Round-Trip Time : le temps nécessaire pour envoyer un paquet et recevoir l'accusé de réception) ;

— en *Multiplicative Decrease*, la fenêtre de congestion est divisée par 2 (arrondi en dessous si besoin) quand une perte de paquet est détectée.

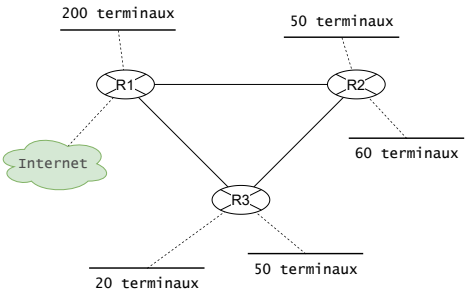
On suppose qu'on a deux connexions TCP, C_1 et C_2, qui partagent une liaison avec de la congestion, pouvant transmettre à un débit maximum de 30 paquets par secondes. La connexion C_1 a un RTT de 50ms, et la connexion C_2 a un RTT de 100ms. On fait enfin l'hypothèse que le débit de données dans la liaison dépasse le débit maximum, toutes les connexions TCP détectent une perte de paquets.

▷ **Question 1 :** À l'instant $t = 0$, C_1 et C_2 ont une fenêtre de congestion de 10 segments. Quelles seront leurs fenêtres de congestion après 1 seconde ? Justifier la réponse, par exemple à l'aide d'un graphique.

▷ **Question 2 :** Est-ce que ces deux connexions vont tendre vers une répartition équitable de la bande passante de la liaison congestionnée ? Expliquer.

★ **Exercice 5 : Routage (4 points).**

Le réseau ci-contre présente un réseau connecté à Internet, et comportant 3 routeurs $R1$, $R2$ et $R3$. Dans cet exercice, on souhaite configurer ce réseau, en prenant en compte le nombre de machines à adresser dans les différents sous-réseaux, tel qu'indiqué sur le schéma. Le fournisseur d'accès Internet (FAI) a attribué au réseau l'espace d'adresses IPv4 123.45.68.0/23.



▷ **Question 1 :** Combien ce réseau comporte-t-il de sous-réseaux ? Combien de machines peut-on adresser dans ce réseau ?

▷ **Question 2 :** Proposez un adressage complet du réseau. Chaque sous-réseau de votre proposition doit avoir un espace d'adressage assez grand pour accueillir les terminaux, tel qu'indiqué sur le schéma. Pour cela, recopiez le schéma sur votre copie, en précisant les adresses des sous-réseaux et les adresses des interfaces des routeurs. Vous expliquerez vos choix.

▷ **Question 3 :** Établissez les tables de retransmission des 3 routeurs, associant à chaque sous-réseau de destination le prochain saut à suivre (*next hop*). Les tables devront comporter une entrée 0.0.0.0/0, dirigeant les paquets vers le reste du réseau Internet. Vous veillerez à donner des tables comportant le moins d'entrées possible ; au besoin, modifiez l'adressage de la question précédente en justifiant.