TP3 : Mémoire virtuelle et appel système `mmap()`

Le but de ce TP est de mettre en pratique les notions vues en cours dans le chapitre sur la mémoire virtuelle. Dans la première partie du sujet, vous allez utiliser `mmap()` pour demander au noyau l'allocation de nouvelles pages de mémoire virtuelle, qui seront partagées entre plusieurs processus concurrents. Dans la seconde partie du sujet, vous utiliserez `mmap()` pour demander au noyau de *projeter* dans votre espace d'adressage virtuel le contenu de fichiers existants. La dernière partie illustre cette même technique avec un exemple inspiré des systèmes de gestion de bases de données.

Au total, il y a assez peu de lignes de code à écrire dans ce TP, donc si vous avez bien compris les concepts de la mémoire virtuelle, vous arriverez rapidement dans la dernière partie qui est la plus ludique. Dans le cas contraire, n'hésitez pas à demander de l'aide et/ou des explications !

Exercice Téléchargez depuis moodle l'archive correspondant à ce TP, puis tapez la commande `tar -zxvf SYS-TP3.tgz`. Vous allez travailler dans le répertoire `SYS-TP3`. Commencez par lire le `Makefile`.

1 Le crible d'Ératosthène — mémoire partagée

Dans cette partie on va implémenter le crible d'Ératosthène, une méthode pour trouver tous les nombres premiers inférieurs à un certain entier *n*. Description tirée de Wikipédia :

> L'algorithme procède par élimination : il s'agit de supprimer d'une table des entiers de 2 à N tous les multiples d'un entier. En supprimant tous les multiples, à la fin il ne restera que les entiers qui ne sont multiples d'aucun entier, et qui sont donc les nombres premiers.
>
> On commence par rayer les multiples de 2, puis à chaque fois on raye les multiples du plus petit entier restant.

Et aussi, une visualisation animée : `https://fr.wikipedia.org/wiki/Crible_d'Eratosthene`.
Dans ce TP, on représentera le crible par un tableau de caractères de taille N, avec tab[i]=1 pour dire «i n'est pas barré» et tab[i]=0 pour dire «i est barré». Rayer les multiples de *k* consistera donc simplement à passer à zéro toutes les cases tab[k∗j] jusqu'à tab[N].

L'intérêt de cet algorithme est de se paralléliser naturellement, et surtout sans poser de problème de synchronisation. Autrement dit, les différentes valeurs de *k* peuvent être traitées en même temps, par différents processus accédant tous au même tableau. C'est le sujet de cette première partie.

Exercice Dans `crible.c` implémentez les fonctions `rayer_multiples()` et `afficher()`.
Vous devez obtenir environ le comportement ci-dessous :

```
% ./crible
nombres premiers jusqu'a 1000: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 1
63, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263,
269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379,
 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491
, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 61
7, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 7
43, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863,
877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997,

% ./crible 50
nombres premiers jusqu'a 50: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
```

Remarque Aidez-vous du `Makefile` fourni, et tapez donc `make crible && ./crible 50` plutôt que de jongler entre plusieurs commandes dans votre terminal.

Exercice On veut maintenant permettre à l'utilisateur de choisir la taille du crible depuis la ligne de commande. Pour cela, vous allez utiliser l'appel système `mmap()` pour demander au noyau d'allouer dynamiquement de l'espace.[1] Concrètement, il s'agit de supprimer le tableau `buffer` (et le `assert()` associé) et de le remplacer par une seule ligne `char * crible = mmap(...);`
Pour savoir comment remplir les « . . . » lisez les nombreuses remarques ci-dessous.

Remarques

— La fonction `mmap()` attend six paramètres : 1. adresse, 2. longueur, 3. protection, 4. drapeaux, 5. descripteur de fichier, et 6. offset.
— La valeur renvoyée par `mmap()` est l'adresse de la nouvelle zone en cas de réussite, ou `MAP_FAILED` en cas d'échec. Testez toujours cette valeur de retour, sous peine d'avoir des bugs difficiles à trouver.
— Le premier paramètre permet d'indiquer à quelle *adresse* (virtuelle) on veut placer le nouveau bloc. Dans ce TP, vous le laisserez toujours à NULL pour laisser le noyau choisir.
— Le second paramètre *longueur* indique la taille (en octets) de la zone qu'on veut créer.
— Le paramètre *protection* est un champ de bits (bitfield) indiquant les droits d'accès qu'on souhaite pour la nouvelle zone. Dans cet exercice on veut pouvoir lire *et* écrire, il faudra donc écrire `PROT_READ|PROT_WRITE`. La barre verticale est le «ou bit-à-bit», i.e. bitwise or.
— Le paramètre *drapeaux* est également un champ de bits, et nous sert à indiquer *deux* informations (cf cours chapitre 3, diapo n° 33 et 34). Attention, il faut toujours préciser les deux !
  D'une part, il faut choisir entre «pages vierges» et «contenu venant d'un fichier».
  — avec `MAP_ANONYMOUS`, on demande l'allocation d'une nouvelle zone vierge, depuis le swap
  — avec `MAP_FILE`, on demande la «projection» d'un fichier dans notre espace d'adressage.
  D'autre part, il faut choisir entre «juste pour moi» et «mémoire partagée», ce qui change notamment le statut de la nouvelle zone en cas de duplication ultérieure du processus.
  — avec `MAP_PRIVATE` on demande une zone privée : lors d'un `fork()`, ces pages seront «dupliquées» comme tout le reste de notre espace d'adressage. [2]
  — avec `MAP_SHARED`, on demande une zone qu'on compte *partager* avec d'autres processus.
— Les paramètres *descripteur de fichier* et *offset* ne seront utiles qu'avec le drapeau `MAP_FILE`, dans la seconde partie du TP. Dans cette première partie, vous pouvez les laisser à zéro. [3]

Pour plus d'informations sur `mmap()` consultez Wikipedia, ou la documentation de la libc en cliquant ici : `https://www.gnu.org/software/libc/manual/html_node/Memory_002dmapped-I_002fO.html`

Vérifiez que votre programme marche toujours, en l'exécutant avec des valeurs différentes de n, par exemple 50, 1000, 5000.

Exercice On veut maintenant paralléliser le programme de façon à ce que `rayer_multiples()` ne soit plus exécutée par le processus principal. Au contraire, on veut que chaque appel à `rayer_multiples()` soit exécuté dans un processus «enfant». La fonction `main()` doit donc maintenant :

— allouer et initialiser le crible de taille n,
— créer $n-2$ enfants, chacun d'entre eux faisant un *unique* appel à `rayer_multiples()`,
— puis *attendre* la terminaison de tous les enfants avant de finalement afficher le crible.

Modifiez votre programme pour obtenir ce comportement. Vous aurez besoin des appels système `fork()`, `wait()`, et `exit()` pour le contrôle des processus (cf TP n° 1), mais surtout, vous devrez changer les paramètres de `mmap()` pour que la zone mémoire soit *partagée* entre tous nos processus.

1. Dans la vraie vie, on aurait fait plus simple, en écrivant par exemple `char * crible = malloc(n*sizeof(char));` mais le TP d'aujourd'hui porte explicitement sur les divers usages de `mmap()`.

2. En pratique, le noyau essayera de retarder la copie grâce à une optimisation nommée *copy-on-write*. cf diapo n° 40 du cours, ou `https://en.wikipedia.org/wiki/Copy-on-write`

3. Même si avec `MAP_ANONYMOUS` le paramètre *filedesc* est ignoré, la pratique courante est plutôt de mettre -1, car le n° zéro correspond à l'entrée standard. Le -1 est donc un peu plus explicite, ce qui facilite la relecture de votre code.

2 Trois petits chats — comment lire un fichier sans `read()`

Dans cette deuxième partie du TP, on s'intéresse à différentes API d'entrées/sorties en fichiers. Le prétexte sera de reproduire la commande unix `cat` qui permet de concaténer plusieurs fichiers et d'afficher le résultat sur la sortie standard.

Exercice Familiarisez-vous avec `cat` en tapant par exemple `cat Makefile`. Essayez aussi avec plusieurs fichiers, par exemple `cat monprogramme.c Makefile monfichier.txt`. Si par erreur vous affichez un fichier binaire qui vous pollue complètement votre terminal, tapez `reset` ou ouvrez simplement une nouvelle fenêtre.

Exercice Dans `cat-stdio.c` reproduisez ces fonctionnalités en utilisant uniquement l'API de la bibliothèque standard C :
- ouverture de fichier avec `fopen()`, fermeture avec `fclose()`
- lecture un caractère à la fois avec `fgetc()`
- affichage un caractère à la fois avec `putchar()`

Pour interrompre la boucle en fin de fichier, vous pouvez utiliser la fonction `feof()`, ou bien tester la valeur de retour de `fgetc()`.
Pour plus d'informations sur toutes ces fonctions, consultez votre poly de C §5.5 et §5.6 p51, et/ou la doc de la glibc : `https://www.gnu.org/software/libc/manual/html_node/I_002f0-on-Streams.html`

Exercice Dans `cat-readwrite.c` implémentez la même chose mais en passant par l'API syscall :
- ouverture de fichier avec `open()` et fermeture avec `close()`
- lecture et écriture, un caractère à la fois, avec `read()` et `write()`.

Remarques
- Contrairement à l'API stdio où un fichier ouvert correspond à un pointeur `FILE*`, les appels système travaillent uniquement en termes de *descripteurs de fichier*.
- Un *descripteur de fichier* est un petit entier, choisi par le noyau au moment du `open()`, et qui sert à identifier le fichier ouvert. Par la suite, lors de chaque appel système, le processus doit indiquer sur quel descripteur il veut travailler.
- Par défaut, chaque processus commence avec trois descripteurs de fichiers :
  - le n° 0 correspond à l'entrée standard (i.e. le clavier).
  - le n° 1 correspond à la sortie standard (i.e. l'écran où vous devez afficher).
  - le n° 2 correspond à la «sortie d'erreur». Il n'est pas utile dans ce TP.
- Pour interrompre la boucle en fin de fichier, testez si la valeur de retour de `read()` est zéro.

Pour plus d'informations, consultez `https://en.wikipedia.org/wiki/File_descriptor` ou la documentation en ligne de la glibc :
open/close → `https://www.gnu.org/software/libc/manual/html_node/Opening-and-Closing-Files.html`
read/write → `https://www.gnu.org/software/libc/manual/html_node/I_002f0-Primitives.html`

Exercice Implémentez dans `cat-mmap.c` une troisième version du programme, cette fois sans utiliser du tout l'appel système `read()`. Au contraire, vous allez utiliser `mmap()` pour demander au noyau de «projeter» tout le fichier dans votre espace d'adressage. Vous pourrez ensuite l'afficher en faisant un unique appel à `write()`. L'exercice consiste essentiellement à trouver quels arguments passer à la fonction `mmap()`. Pour cela, aidez-vous des remarques ci-dessous.

Remarques
- On veut maintenant non plus allouer des pages vierges, mais projeter le contenu d'un fichier existant. Il faut donc passer le drapeau `MAP_FILE`, ainsi que le bon descripteur de fichier.
- Les paramètres *longueur* et *offset* permettent de spécifier quelle portion du fichier on veut projeter.
- Pour connaître la taille du fichier, vous pouvez utiliser par exemple l'appel système `fstat()`, en écrivant `struct stat buf; fstat(fd, &buf);` (où `fd` est le descripteur de fichier). La taille du fichier sera indiquée dans `buf.st_size`

— Utilisez `close()` pour refermer le fichier, et `munmap()` pour supprimer la projection.
— Rappel : posez des questions, lisez Wikipedia, et/ou la documentation en ligne de `mmap()`
`https://www.gnu.org/software/libc/manual/html_node/Memory_002dmapped-I_002fO.html`

3 Le jeu des prénoms — mémoire virtuelle VS bases de données

Dans cette dernière partie, on se penche d'un peu plus près sur la projection de fichiers en mémoire, à la manière des systèmes de gestion de bases de données. Les exercices sont moins guidés que dans le début du TP, donc n'hésitez pas à demander de l'aide si vous en avez besoin.

Vous trouverez sur Moodle un fichier contenant des infos sur les prénoms attribués aux enfants nés en France depuis 1900. Il s'agit d'un fichier binaire, destiné à être projeté en mémoire avec `mmap()` puis accédé comme un tableau de struct, à l'aide de la déclaration ci-dessous :

```
typedef struct {
    int sexe; // 1=Garcon, 2=Fille
    char prenom[25];
    int annee; // 1900..2021
    int nombre; // d'enfants nés cette année avec ce prénom
} tuple;
```

Note : pour simplifier votre travail, les prénoms sont écrits sans accent. Les seuls caractères utilisés sont les majuscules ASCII (c.à.d. les valeurs entre `'A'` et `'Z'`) et le trait d'union `'-'`.

Exercice Dans `prenoms.c`, commencez par charger le fichier en mémoire, puis affichez-en le contenu de façon lisible, un `tuple` par ligne. Attention, après la fin du fichier, notre zone mémoire contiendra des données invalides, i.e. des `tuple` avec `sexe`=0.

Exercice Parcourez la table pour répondre à des questions de type «bases de données», par exemple : quel est le prénom le plus long ? ou encore : en quel année *votre* prénom a-t-il été le plus populaire ?

Exercice Avec les données ainsi projetées en mémoire, on peut aussi les modifier directement, et compter sur le noyau pour sauvegarder les changements dans le fichier. Remplacez dans tous les prénoms l'écriture en CAPSLOCK par du CamelCase, par exemple JEAN-KEVIN $\rightarrow$ Jean-Kevin. Assurez-vous que vos modifications sont bien sauvegardées sur le disque.

Bonus, pour la culture générale, un peu de lecture sur les joies de mmap dans la vraie vie :
`https://www.sublimetext.com/blog/articles/use-mmap-with-care`