

Année	2024-2025	Type	Devoir Surveillé
License	Informatique		
Code UE	4TIN503U	Épreuve	Programmation Système
Date	23/10/2024	Documents	Non autorisés
Début	9h30	Durée	1h

Lisez l'intégralité du sujet avant de commencer à répondre aux questions. À la fin du sujet, vous trouverez un memento vous rappelant la syntaxe de quelques appels système utiles.

1 Questions de cours (échauffement)

Question 1 Rappelez le principe de fonctionnement de l'appel système `mmap`. Quels sont les principaux avantages par rapport à l'utilisation de `read/write` ?

Question 2 Qu'est-ce qu'un signal masqué ? Décrivez ce qui se passe lorsque plusieurs signaux identiques sont envoyés vers un processus qui les a masqués. Parmi ces signaux, lesquels ne peut-on pas (dé)masquer : SIGKILL, SIGINT, SIGZORRO.

2 Fichiers

On souhaite implanter une version simplifiée des entrées-sorties proposées par la bibliothèque C standard. L'objectif est de rendre transparente l'utilisation d'un cache de données lors des opération de lecture¹ sur les fichiers. À l'instar du type `FILE`, on décide donc de se doter d'une structure `my_file_t` qui contient les champs suivants :

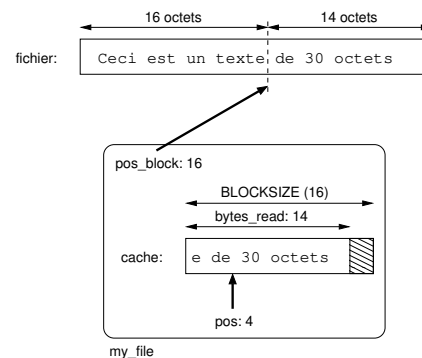
```
typedef struct {
    int fd;           // descripteur du fichier
    char cache[BLOCKSIZE]; // octets préchargés
    off_t pos;        // position tête de lecture au sein du bloc
    off_t pos_block;  // position du bloc au sein du fichier
    int bytes_read;   // remplissage du bloc (-1 si pas encore chargé)
} my_file_t;
```

```
my_file_t *my_file = my_open ("file.txt");
int n;
char c;

do {
    n = my_read (my_file, &c, 1);
    write (STDOUT_FILENO, &c, n);
} while (n == 1);

my_close (my_file);
```

(a) Exemple d'utilisation



(b) Exemple de contenu pour `my_file` après lecture de 20 octets

La figure 1a illustre la façon dont on souhaite utiliser les primitives associées à ce type. L'idée est d'utiliser le tableau `cache` pour précharger des tranches de `BLOCKSIZE` octets (ou moins si on est proche de la fin de fichier) au fur et à mesure des lectures de l'utilisateur. La figure 1b illustre le contenu de la structure `my_file` après que 20 octets aient déjà été lus. Le champ `bytes_read` indique le nombre d'octets chargés dans le cache. Ce champ vaut normalement `BLOCKSIZE` sauf dans le cas présent où il s'agit de la dernière tranche du fichier. Le champ `pos_block` indique la position du bloc au sein du fichier (ici 16). Enfin le champ `pos` indique la position courante du curseur de lecture au sein du cache (4 octets déjà lus).

1. On ne s'intéressera pas aux écritures dans cet exercice.

Question 1 Complétez le code de la fonction `my_file_open` ci-dessous. Notez que l'on ne cherche pas à précharger de données depuis le fichier à ce stade.

```
my_file_t *my_open (char *filename)
{
    my_file_t *f;
    ... // TODO
    f->bytes_read = -1;
    f->pos = f->pos_block = 0;
    return f;
}
```

Question 2 Donnez le code de la fonction `off_t my_file_size (my_file_t *f)` qui renvoie la taille du fichier préalablement ouvert. Vous vous aiderez de l'appel système `lseek`, mais en veillant bien à ce que la position courante dans le fichier soit inchangée à l'issue de l'appel.

Question 3 Complétez la fonction `my_read` qui doit lire `n` octets depuis le cache, éventuellement en plusieurs fois.

```
int my_read (my_file_t *f, char *buffer, int n)
{
    char *ptr = buffer;
    unsigned total_read = 0;

    for (;;) {
        // s'il ne reste plus d'octet à lire, on sort
        if (n == 0)
            break;
        // si on se trouve à la fin du bloc, on avance au suivant
        if (f->pos == BLOCKSIZE) {
            f->pos = 0;
            f->pos_block += BLOCKSIZE;
            f->bytes_read = -1;
        }
        // si le cache n'est toujours pas chargé, c'est le moment !
        if (...) {
            // TODO
        }
        // recopier les caractères depuis le cache vers buffer
        int nb = ...; // nombre de caractères extractibles de la tranche courante
        if (nb <= 0) // c'est fini
            break;
        // TODO

        f->pos += nb;
        ptr += nb;
        total_read += nb;
        n -= nb;
    }

    return total_read;
}
```

3 Processus

On souhaite disposer d'une fonction `void run_at (unsigned secs, char *cmd, char *arg)` qui crée un processus chargé d'exécuter le programme `cmd` (avec un seul paramètre `arg`) au bout de `secs` secondes. Voici typiquement comment on voudrait pouvoir l'utiliser :

```
int main ()
{
    run_at (2, "ls", "-l");
    run_at (4, "echo", "coucou");
    sleep (10);
}
```

Dans ce cas, on devrait voir apparaître le résultat de `ls -l` sur la sortie standard au bout d'environ 2 secondes, puis le résultat de `echo coucou` 2 secondes plus tard.

Question 1 Donnez le code de la fonction `run_at`. Note : on ne se soucie pas de la récupération des zombies.

Question 2 Ajoutez un 4^e paramètre `int fd` à la fonction `run_at` et redirigez la sortie d'erreur de la commande vers ce descripteur, supposé ouvert et accessible en écriture.

Question 3 On suppose qu'un *tube* est créé au début de la fonction `main` et que son descripteur en écriture est passé en paramètre à tous les appels à `run_at`.

Remplacez `sleep (10)` par une boucle qui lit depuis le tube et qui ajoute tous les caractères lus à la fin d'un fichier `"run.log"`. La boucle de lecture ne doit pas être infinie...

Memento

<pre>int open(const char *pathname, int flags, mode_t mode); ssize_t read(int fd, void *buf, size_t count); ssize_t write(int fd, const void *buf, size_t count); off_t lseek(int fd, off_t offset, int whence); /* whence = SEEK_SET ou SEEK_CUR ou SEEK_END */</pre>	<pre>int dup2(int oldfd, int newfd); int execlp(const char *file, const char *arg0, ...); int execvp(const char *file, char *const argv[]); int pipe(int fildes[2]); pid_t wait(int *stat_loc);</pre>
--	---