# Programmation système : tampons

Juliusz Chroboczek

12 Octobre 2009

## 1 Tampons

Le programme de copie décrit au cours précédent e□ectue deux appels système pour chaque octet copié — ce qui est tragique. Une machine moderne est capable d'e□ectuer jusqu'à 3 000 000 d'appels système par seconde ; ce programme ne sera donc pas capable de copier plus d'1,5 Mo de données par seconde environ.

Pour résoudre ce problème de performances, il su□t d'e□ectuer des transferts de plus d'un octet pour chaque appel système ; il faudra pour cela disposer d'une zone de mémoire pour stocker les données en cours de transfert. Une telle zone s'appelle un *tampon* (*bu er* en anglais).

### 1.1 Tampon simple

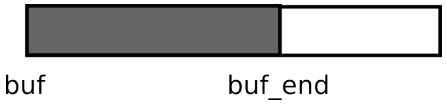


Figure 1: Tampon simple

Un tampon simple (□g. 1) est constitué d'une zone de mémoire `buf` et d'un entier `buf_len` indiquant la quantité de données dans le tampon :

```
void *buf;
int buf_end;
```

Les données valides dans le tampon se situent entre `buf` et buf + `buf_len` − 1.

Un tampon simple permet de lire les données de façon incrémentale ; cependant, les données doivent être écrites en une seule fois.

**Exemple** La version suivante du programme de copie utilise un tampon simple, et n'e□ectue que deux appels système tous les `BUFFER_SIZE` octets.

```
#define BUFFER_SIZE 4096
char buf[BUFFER_SIZE];
int buf_end;
...
    buf_end = 0;
    while(1) {
        rc = read(fd1, buf, BUFFER_SIZE);
        if(rc < 0) { perror("read"); exit(1); }
        if(rc == 0) break;
        buf_end = rc;
        rc = write(fd2, buf, buf_end);
        if(rc < 0) { perror("write"); exit(1); }
        if(rc != buf_end) { fprintf(stderr, "Écriture interrompue"); exit(1); }
        buf_end -= rc;
    }
...
```

## 1.2 Tampon avec pointeur de début des données


Figure 2: Tampon avec pointeur de début des données

```
    while(1) {
        rc = read(fd1, buf, BUFFER_SIZE);
        if(rc < 0) { perror("read"); exit(1); }
        if(rc == 0) break;
        buf_end = rc;
        while(buf_ptr < buf_end) {
            rc = write(fd2, buf + buf_ptr, buf_end - buf_ptr);
            if(rc < 0) { perror("write"); exit(1); }
            buf_ptr += rc;
        }
        buf_ptr = buf_end = 0;
    }
...
```

### 1.3 Tampon circulaire


Figure 3: Tampon circulaire

Avec un tampon avec pointeur, la variable `buf_end` ne revient au bord gauche du tampon que lorsque le tampon est vide ; lorsque la fin du tampon `buf_end` atteint le bord droit du tampon, une lecture n'est plus possible tant que le tampon n'est pas vidé.

Un *tampon circulaire* (fig. 3) consiste des mêmes données qu'un tampon avec pointeur, mais les deux pointeurs sont interprétés modulo la taille du tampon. Lorsque `buf_end` > `buf_ptr`, les données valides ne sont plus connexes, mais constituées des deux parties `buf_ptr` à `BUFFER_SIZE` et 0 à `buf_end`.

Il existe une ambiguïté dans cette structure de données : lorsque `buf_end` = `buf_ptr`, il n'est pas clair si le tampon est vide ou s'il est plein. On peut différencier entre les deux conditions soit en évitant de mettre plus de `BUFFER_SIZE` − 1 octets dans le tampon, soit en utilisant une variable booléenne supplémentaire.

## 2 La bibliothèque `stdio`

La bibliothèque `stdio` a un double rôle : elle encapsule la manipulation des tampons dans un ensemble de fonctions pratiques à utiliser, et combine les entrées/sorties avec l'analyse lexicale et le formatage. Des bibliothèques analogues existent dans la plupart des langages de programmation, par exemple la bibliothèque `java.io` en Java.

Pour des raisons d'efficacité, il est parfois souhaitable de contourner `stdio` et manipuler les tampons soi-même ; par exemple, les programmes ci-dessus utilisent un seul tampon pour les entrées et les sorties, ce qui est impossible avec `stdio`.

Du fait de leur portabilité, les fonctions de la bibliothèque `stdio` présentent un certain nombre de limitations. En particulier, comme certains systèmes différencient entre fichiers binaires et fichiers texte, `stdio` ne permet pas de mélanger les données textuelles et les données binaires au sein d'un même fichier (cette limitation peut être ignorée sur les systèmes POSIX).

### 2.1 La structure `FILE`

Les fonctions de `stdio` n'opèrent pas sur des descripteurs de fichiers, qui sont une notion spécifique à Unix, mais sur des pointeurs sur une structure qui représente un flot, la structure `FILE`. La définition de `FILE` dépend du système, mais sur un système POSIX elle pourrait être définie comme suit :

```
#define BUFSIZ 4096

#define FILE_ERR 1
#define FILE_EOF 2

typedef struct _FILE {
    int fd;             /* descripteur de fichier */
    int flags;          /* flags */
    char *buffer        /* tampon */
    int buf_ptr;        /* position courante dans le tampon */
    int buf_end;        /* fin des données du tampon */
} FILE;
```

Le champ `fd` contient le descripteur de fichier associé à ce flot. Le champe `buffer` contient un pointeur sur un tampon de taille `BUFSIZ`. Le champ `buf_ptr` indique la position courante dans le tampon, et le champ `buf_end` indique la fin du tampon.

Le champ `flags` indique l'état du flot. Il vaut normalement 0, `FILE_ERR` si une erreur a eu lieu sur ce flot, et `FILE_EOF` si la fin du fichier a été atteinte. Dans une vraie implémentation de `stdio`, le champ `flags` contiendra aussi de l'information qui indique la politique de gestion du tampon associé au flot (voir ci-dessous).

Une structure `FILE` est normalement créée par la fonction `fopen`(3) :

```
FILE *fopen(const char *path, const char *mode)
```

Cette fonction ouvre le fichier nommé par `path` et construit une structure `FILE` associée au descripteur de fichier résultant.

Le paramètre `mode` indique le but de l'ouverture de fichier. Il s'agit d'une chaîne de caractères dont le premier élément vaut « `r` » pour une ouverture en lecture, « `w` » pour une ouverture en écriture, et « `a` » pour une ouverture en mode ajout. Ce caractère peut être suivi de « `+` » pour une ouverture en lecture et écriture simultanées, et « `b` » si le fichier ouvert est un fichier binaire plutôt qu'un fichier texte (cette dernière information est ignorée sur les systèmes POSIX).

Une structure `FILE` est détruite à l'aide de la fonction `fclose`(3) :

```
int fclose(FILE *file);
```

Cette fonction ferme le descripteur de chier, libère le tampon, puis libère la structure `FILE` elle-même.

## 2.2 Entrées/sorties de haut niveau

Les entrées/sorties `stdio` utilisent toujours un tampon ; celui-ci peut-être vidé lorsqu'il est plein (c'est ce qui se passe lorsqu'un `FILE` est associé à un chier), à la n de la ligne (c'est ce qui se passe lorsqu'un `FILE` est associé à un terminal), ou après chaque appel de fonction d'entrée/sortie. La politique de gestion du tampon tampon associé à un `FILE` peut être changée à l'aide de la fonction `setvbuf`(3).

Les fonctions d'entrée/sortie fondamentales sont `getc`(3), qui retourne un caractère lu sur un ot, et `putc`(3), qui écrit un caractère sur un ot. La fonction `fflush`(3) permet de vider le tampon.

En ignorant les complications liées aux chiers ouverts en entrée et en sortie simultanément ainsi que celles liées aux tampons « par ligne », les fonctions `getc` et `putc` pourraient être dénies comme suit :

```
int getc(FILE *f)
{
    if(f->buf_ptr >= f->read_end)
        _filbuf(f);
    if(f->flags & (FILE_ERR | FILE_EOF))
        return -1;
    return (unsigned char)fp->buf[fp->buf_ptr++];
}

int putc(char ch, FILE *f)
{
    if(f->buf_ptr >= BUFSIZ)
        _flshbuf(f);
    if(f->flags & FILE_ERR)
        return -1;
    fp->buf[fp->buf_ptr++] = ch;
    return ch;
}
```

*Exercice :* écrivez les fonctions `_filbuf` et `_flshbuf`.

Comme `stdio` utilise des tampons, il est possible de faire toutes les entrées/sorties avec ces fonctions. Cependant, `stdio` fournit aussi des fonctions d'entrée/sortie « par lots », `fread`(3) et `fwrite`(3), modélisées sur `read`(2) et `write`(2). Comme elles sont prévues pour fonctionner sur des systèmes où les chiers ne sont pas forcément des suites d'octets, ces fonctions opèrent sur des tableaux d'éléments de taille quelconque plutôt que sur des tableaux d'octets :

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *file);
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
```

```
                FILE *file);
```

*Exercice :* écrivez les fonctions `fread` et `fwrite`, d'abord en termes de `getc` et `putc`, ensuite en manipulant directement les tampons.

### 2.2.1 Interaction entre `stdio` et le système

Au démarrage du programme, `stdio` crée trois flots associés aux descripteurs de fichier 0, 1 et 2, et les affecte aux variables globales `stdin`, `stdout` et `stderr` respectivement.

Une extension POSIX à `stdio`, `fdopen`(3), permet de créer un `FILE` sur un descripteur de fichier existant :

```
FILE *fdopen(int fd, const char *mode);
```

Dualement, la fonction `fileno`(3) permet d'obtenir le descripteur de fichier associé à un `FILE` :

```
int fileno(FILE *file);
```

Enfin, la fonction `fflush`(3) permet de vider le tampon associé à un `FILE` :

```
int fflush(FILE *file);
```

Appelée avec un pointeur nul, cette fonction vide les tampons de tous les `FILE` du système.

Comme `stdio` utilise des tampons, il n'est pas facile d'utiliser simultanément les appels système et les fonctions `stdio` sur le même descripteur de fichier. Dans le cas des sorties, il faut utiliser la fonction `fflush`(3), qui vide le tampon associé à un `FILE`, avant chaque appel système ; dans le cas des entrées, il faut faire un appel à `fseek`(3) à chaque transition entre les appels système et les fonctions `stdio`.

## 2.3 Entrées/sorties formatées

Pour présenter des données à l'utilisateur, il faut en construire une *représentation textuelle*, une suite de caractères qui correspond à une présentation culturellement acceptée de ces données. Inversément, les données entrées par l'utilisateur doivent être *analysées*.

### 2.3.1 Formatage et analyse

La suite `stdio` contient la fonction `snprintf`(3) qui effectue le formatage des données :

```
int snprintf(char *buf, size_t size, const char *format, ...);
```

Cette fonction prend en paramètre un tampon `buf` de taille `size`, et y stocke une représentation des paramètres guidée par `format`.

Si le formatage est réussi, `snprintf` retourne le nombre de caractères stockés dans `buf` (sans compter le `\0` final). Lorsque `buf` était trop court, le comportement dépend de la version : dans les implémentations C89, elle retourne -1, dans les implémentations C99, elle retourne la taille du tampon dont elle aurait eu besoin. Cette différence entre les versions doit être gérée par le programmeur :

```
char *format_integer(int i)
{
    char *buf;
    int n = 4, rc;
    while(1) {
        buf = malloc(n);
        if(buf == NULL) return NULL;
        rc = snprintf(buf, n, "%d", i);
        if(rc >= 0 && rc < n) return buf;
        free(buf);
        if(rc >= 0)
            n = rc + 1;
        else
            n = 2 * n;
    }
}
```

Dualement, la fonction `sscanf`(3) e□ectue l'analyse :

```
int sscanf(const char *str, const char *format, ...);
```

### 2.3.2 Entrées/sorties formatées

Les fonctions de formatage et d'analyse et les fonctions d'entrées/sorties sont combinées en des fonctions d'entrées/sorties formatées. Ainsi, la fonction `fprintf`(3) formate ses arguments avant de les écrire sur un □ot, et `fscanf`(3) lit des données qu'elle analyse ensuite. Les fonctions `printf`(3) et `scanf`(3) sont des abbréviations pour `fprintf` et `fscanf` dans le cas où l'argument `file` vaut `stdout` et `stdin` respectivement.