

# Programmation Système

Guillaume Chanel & Florent Glück

May 27, 2025

## Appels système et make

### Introduction

---

Nous sommes intéressés à comparer les performances des deux API permettant de manipuler les entrées/sorties (I/O) dans UNIX/Linux: les appels systèmes et les I/O bufferisées. Pour rappel, les appels systèmes utilisent les descripteurs de fichiers et comprennent les fonctions `open`, `read`, `close`, etc. Les I/O bufferisées utilisent le type opaque `FILE*` et comprennent les fonctions `fopen`, `fread`, `fclose`, etc. A savoir que à chaque appel aux fonctions `open`, `read`, `close` exécutent directement les appels systèmes des mêmes noms, alors que ce n'est pas toujours le cas pour les fonctions `fopen`, `fread`, etc.

Les objectifs de ce travail pratique sont les suivants :

- Copier des fichiers volumineux de manières différentes, réaliser des mesures de temps de ces copies et analyser/expliciter les résultats obtenus
- Se familiariser avec l'outil `make`
- Analyser l'impact des appels systèmes

### Cahier des charges

---

En premier lieu, écrivez deux fonctions C, `copy` et `copyf`, qui copient chacune le contenu d'un fichier représenté par le chemin `src` dans un fichier de destination `dst` en copiant par bloc(s) de `buf_size` byte(s). La fonction `copy` doit utiliser l'API portant sur les descripteurs de fichiers (fonctions `open`, `read`, etc.), alors que la fonction `copyf` doit utiliser l'API portant sur le type opaque `FILE*` (fonctions `fopen`, `fread`, etc.). Chaque fonction doit retourner le nombre de bytes copiés. Voici le prototype de chaque fonction :

```
int copy(char *src, char *dst, unsigned int buf_size);
int copyf(char *src, char *dst, unsigned int buf_size);
```

En deuxième lieu, on désire écrire en C le programme `copybench` qui prend exactement deux arguments selon la syntaxe suivante :

```
copybench SRC DST
```

Ce programme devra être dans un fichier `.c` différent des fonction `copy` et `copyf`. Votre programme aura donc au moins deux modules.

Voici le comportement attendu du programme :

- Copie le fichier `SRC` dans `DST` en utilisant les fonctions `read/write` et une taille de buffer de 1 byte

- Copie le fichier **SRC** dans **DST** en utilisant les fonctions `read/write` et une taille de buffer de 32KB
- Copie le fichier **SRC** dans **DST** en utilisant les fonctions `fread/fwrite` et une taille de buffer de 1 byte
- Copie le fichier **SRC** dans **DST** en utilisant les fonctions `fread/fwrite` et une taille de buffer de 32KB
- Pour chaque copie ci-dessus, les informations suivantes doivent être affichées :
  - le fichier source et destination
  - le nombre de KB copiés (1KB = 1024 bytes)
  - le temps mis pour effectuer la copie, en millisecondes (pour cela vous pouvez utiliser la fonction `clock_gettime` avec le paramètre `CLOCK_MONOTONIC`, voir ci-dessous)
- A titre d'exemple, si `copybench` est exécuté avec les arguments `bigfile tmp` alors, ceci sera affiché<sup>1</sup> sur la sortie standard :

```
Copied (read/write) "bigfile" into "tmp" (10240KB in chunks of 1 bytes) in 1234.0 ms
Copied (fread/fwrite) "bigfile" into "tmp" (10240KB in chunks of 1 bytes) in 5678.0 ms
Copied (read/write) "bigfile" into "tmp" (10240KB in chunks of 32768 bytes) in 8765.0 ms
Copied (fread/fwrite) "bigfile" into "tmp" (10240KB in chunks of 32768 bytes) in 4321.0 ms
```

Un bon moyen pour tester votre programme est de créer un fichier volumineux, par exemple de 10MB, contenant des données aléatoires. Utilisez donc l'outil `dd` et la source de données aléatoires `/dev/urandom` pour créer un tel fichier.

Enfin, un `Makefile` sera écrit afin de faciliter la création de l'exécutable et l'exécution du programme. Voici ce que `make` doit afficher si exécuté sans argument :

```
Available targets:
bigfile      generates a 10MB file of random data from /dev/urandom
copybench    generates the copybench executable
run          executes copybench
clean        deletes all generated files (including bigfile)
```

La cible `run` devra s'occuper de créer `bigfile`, puis d'exécuter `copybench` sur ce fichier afin de réaliser le test de performances.

Assurez-vous de gérer les dépendances correctement ! Par exemple, si la cible `copybench` est appelée, alors un fichier `.o` devra être généré par module et ces fichiers liés dans l'exécutable `copybench`. Autre exemple : si `make` est exécuté plusieurs fois, alors rien ne devrait être créé car les fichiers nécessaires seront déjà générés.

Finalement, pensez à gérer au mieux les erreurs qui pourraient survenir. Dans tous les cas, les messages d'erreurs doivent être écrits sur la sortie d'erreur standard (`stderr`). Si les fonctions utilisées mettent à jour `errno`, assurez vous d'utiliser les fonctions dédiées comme expliqué en cours. Dans le cadre de manipulations de fichiers, pensez toujours que les fonctions peuvent échouer, donc gérez **toujours** les erreurs<sup>2</sup> lors des appels à `open`, `fopen`, `read`, `fread`, etc. Le programme devra également afficher une erreur, et la syntaxe à utiliser, si le nombre d'arguments est incorrect.

<sup>1</sup>Les timings indiqués ici sont complètement fictifs et non-représentatifs de la réalité

<sup>2</sup>Consultez toujours le manuel (`man 2 open`, etc.), celui-ci est votre meilleur ami !

## Interprétation des résultats

---

Observez les timings obtenus lors de l'exécution de `copybench` et essayez d'expliquer les valeurs observées. En particulier, essayez d'expliquer :

- Les différences de temps obtenues entre l'utilisation de `fread/fwrite` vs `read/write`
- L'impact de la taille du buffer

## Remarques utiles

---

### make

Dans une instruction de production d'un Makefile, préfixer une commande avec `@` indique à `make` de ne pas afficher la commande sur la sortie standard. Exemple d'un Makefile :

```
msg:
    @echo "blah blah"
```

Puis son exécution :

```
$ make
blah blah
```

### Mesures de temps

La fonction `clock_gettime` du header `<time.h>` permet de réaliser des mesures de temps précises. Attention, si vous compilez votre code avec le standard C11 ou supérieur au lieu du standard GNU C (défaut de `gcc`), il faut spécifier `#define _POSIX_C_SOURCE 199309L` avant `#include <time.h>`. Voici un exemple d'utilisation de la fonction `clock_gettime` :

```
struct timespec start, finish;
clock_gettime(CLOCK_MONOTONIC, &start);
... // code à mesurer
clock_gettime(CLOCK_MONOTONIC, &finish);
double seconds_elapsed = finish.tv_sec-start.tv_sec;
seconds_elapsed += (finish.tv_nsec-start.tv_nsec)/1000000000.0;
```

Le code ci-dessus nécessite la librairie `librt` qui est automatiquement liée à l'exécutable sur la plupart des systèmes. Au cas où cela ne serait pas le cas, passez l'argument `-lrt` à l'édition des liens.

### Outil dd

L'outil `dd` permet de copier des blocs de bytes d'une source à une destination, selon la syntaxe suivante :

```
dd if=source of=dest bs=size count=n
```

où :

- `if` signifie *input file* ; `source` spécifie un fichier ou périphérique source
- `of` signifie *output file* ; `dest` spécifie un fichier ou périphérique de destination
- `bs` signifie *block size* ; `size` spécifie la taille d'un bloc (en octets) pour le transfert ; il est possible de postfixer la valeur avec "K" ou "M" pour spécifier des KB ou des MB

- `n` spécifie le nombre de blocs à copier de source à destination

A savoir que `bs` et `count` sont des paramètres optionels. Si aucun n'est spécifié alors `dd` copie l'entiereté de la source dans la destination.

Voici deux exemples qui copient les 2048 premiers octets d'une clé USB (exposée par le noyau dans `/dev/sdb`) dans le fichier `myfile` :

```
dd if=/dev/sdb of=myfile bs=512 count=4
dd if=/dev/sdb of=myfile bs=1K count=2
```

## Outil `strace`

L'outil `strace` pourra vous aider à analyser et comprendre les timings obtenus. Celui-ci affiche les appels système réalisés par tout programme passé en argument. Exemple d'utilisation :

```
strace ls
```

## Écriture sur la sortie d'erreur

Pour écrire sur la sortie d'erreur `stderr`, il est recommandé d'utiliser les fonctions vues en cours, notamment `perror` ou `strerror` (cette dernière en combinaison avec `fprintf`).

## Conseils

---

- Pensez à régulièrement compiler et tester votre code, ceci de manière incrémentale.
- Si besoin, utilisez un débogueur comme expliqué durant le cours plutôt que des `printf`.