Programmation C et Système
Bufferisation des entrées/sorties de stdio

Régis Barbanchon

L2 Informatique

La couche `<stdio.h>` vs la couche système `<unistd.h>`

En C, on peut faire des entrées/sorties sur fichiers
en utilisant deux couches de fonctions, basse et haute :

- La couche basse des fonctions système POSIX de `<unistd.h>`
  qui manipule un entier comme descripteur de fichier (le `fd`) :
  - `open()`, `close()`, pour l'ouverture et la fermeture,
  - `read()`, `write()`, pour les entrées/sorties,
  - `lseek()`, pour le positionnement du curseur IO...
- La couche haute des fonctions standard ISO de `<stdio.h>`
  qui manipule une structure opaque pointée par un `FILE *`
  et qui **temporise les appels système via un buffer** :
  - `fopen()`, `fclose()`, pour l'ouverture et la fermeture,
  - `setvbuf()`, pour changer le mode de gestion initial du bufffer,
  - `fflush()`, pour vider le buffer et forcer l'appel système,
  - `fread()`, `fwrite()`, pour les entrées/sorties binaires,
  - `fputc()`, `fputs()`, `fprintf()`, pour l'écriture de caractères,
  - `fgetc()`, `fgets()`, `fscanf()`, pour la lecture de caractères.

Lecture/écriture avec <stdio.h> (voir les man pages)

Lecture/écriture de caractère et de ligne :

```
int fgetc (       FILE * stream); // returns EOR or the character read
int fputc (int c, FILE * stream); // returns EOF or c

char * fgets (char       s[], int size, FILE * stream); // returns NULL or s
int    fputs (char const s[],           FILE * stream); // returns EOF or >=0
```

Lecture/écriture texte formatée :

```
int fprintf (FILE * stream, char const format[], ...); // the # of printed chars
int fscanf  (FILE * stream, char const format[], ...); // the # of conversions
```

Lecture/écriture binaire :

```
size_t fread  (void       * ptr, size_t size, size_t count, FILE * stream);
size_t fwrite (void const * ptr, size_t size, size_t count, FILE * stream);
```

Positionnement dans un flux seekable :

```
int  fseek (FILE * stream, long offset, int whence); // -1 or 0
long ftell (FILE * stream);                          // the current offset
```

Contrôle d'erreur et de fin de fichier :

```
int  feof     (FILE *stream); // tests if end of file was met by a IO function
int  ferror   (FILE *stream); // tests if an IO error was met by a IO function
void clearerr (FILE *stream); // clears the flags of the two functions above
```

L'ouverture via `fopen()` vs `open()`

La fonction `fopen()` ouvre un fichier de chemin `path`, et retourne le fichier ouvert dans un `FILE *`, ou `NULL` en cas d'échec. C'est l'analogue de la fonction système `open()` qui renvoie un `fd`.

```
FILE * fopen (const char path[], char const mode[]);
int     open (const char path[], int flags, mode_t permissions);
```

Le `mode` est une chaîne qui contrôle les `flags` de `open()` :

- `"r"` : $\Leftrightarrow$ `open()` avec `O_RDONLY`,
- `"w"` : $\Leftrightarrow$ `open()` avec `O_WRONLY | O_CREAT | O_TRUNC`,
- `"a"` : $\Leftrightarrow$ `open()` avec `O_WRONLY | O_CREAT | O_APPEND`,
- `"r+"` : $\Leftrightarrow$ `open()` avec `O_RDWR`,
- `"w+"` : $\Leftrightarrow$ `open()` avec `O_RDWR | O_CREAT | O_TRUNC`,
- `"a+"` : $\Leftrightarrow$ `open()` avec `O_RDWR | O_CREAT | O_APPEND`.

Le fichier est créé avec `permissions= 0666`, c-à-d `[rw-rw-rw-]` : `S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH` moins les droits enlévés par le masque spécifié par `umask()`.

Pont entre les deux couches

La couche haute ISO fait appel à la couche basse POSIX :
tout `FILE *` manipule le `fd` sous-jacent ouvert par `open()`.

Des fonctions permettent de faire le pont entre les deux couches :

- `fileno()` permet de connaître le `fd` d'un `FILE *`.

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

- `fdopen()` permet d'obtenir un `FILE *` à partir d'un `fd`.

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

  Le mode doit être compatible avec les flags d'ouverture du `fd`.

Interruption lors d'un appel système

Le CPU a deux modes d'exécution des instructions :

- **user** : mode normal d'exécution des processus,
- **kernel** : mode privilégié d'exécution du sytème d'exploitation.

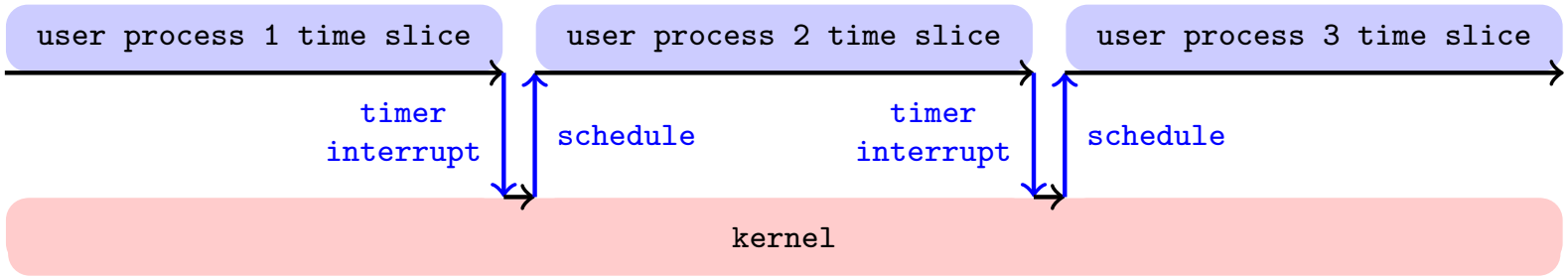
Les fonctions IO de la couche basse génèrent un appel au système, ce qui se traduit une **interruption** du programme pour donner le contrôle du CPU au système (avec changement de mode).

Le système utilisant les registres du CPU lors de son appel, leurs valeurs sont préalablement sauvées afin d'être restaurées lorsque le système rendra le CPU au processus (**context switch**).

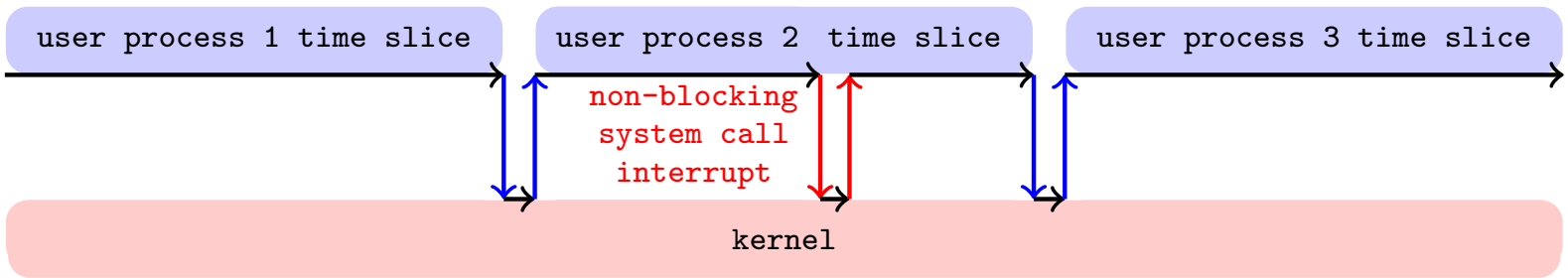
Si l'appel est **bloquant** (attente de disponibilité de la ressource), le processus perd la tranche de temps accordée par le *scheduler*, et le système donne alors le contrôle du CPU à un autre processus.

Interruption lors d'un appel système

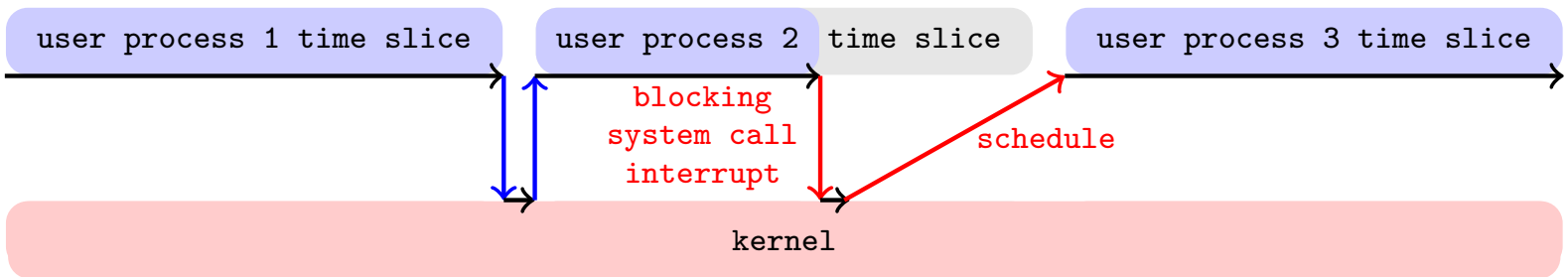
- Scheduling de 3 processus sans perte de tranche de temps :



- Appel système non-bloquant (sans perte de la tranche) :



- Appel système bloquant (et perte de la tranche) :



Bufferisation de `<stdio.h>`

Les appels système sont coûteux en temps. Il faut donc en limiter la fréquence. La couche haute limite le nombre d'appels systèmes :

- 1 appel à `fputc()`, `fputs()`, `fprintf()`, `fwrite()`, ... n'implique pas nécessairement 1 appel à `write()`, et
- 1 appel à `fgetc()`, `fgets()`, `fscanf()`, `fread()`, ... n'implique pas nécessairement 1 appel à `read()`.

Les sorties de la couche haute sont **bufferisées**. *Grosso modo* :

- En sortie, chaque appel à `fputc()` *and co* met les caractères dans un **buffer**, c-à-d, une file d'attente de caractères. Lorsque le buffer est plein (entre autres), le flux subit un **flush** : `write()` est appelé, et le buffer est vidé.
- En entrée, un appel à `fgetc()` *and co* prend ses caractères depuis un **buffer** : `read()` n'est appelé que si le buffer est vide à ce moment, pour tenter de le remplir au max de sa capacité.

Modes de bufferisation des flux de sortie

Il y a 3 modes de bufferisation pour les flux de sorties.

- **unbuffered**. Pas de buffer associé au flux de sortie :
  - 1 `write()` pour chaque `fgetc()`, `fgets()`, `fprintf()`.

- **fully-buffered**. Le flux de sortie subit un **flush** lorsque :
  - la capacité du buffer est atteinte,
  - la fonction `fflush()` est invoquée sur le flux,

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

  - le flux est fermé avec `fclose()`, qui invoque `fflush()` (des écritures en attente s'effectuent donc à la fermeture).

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

- **line-buffered**. Le flux de sortie subit un **flush** lorsque :
  - une des 3 conditions ci-dessus est rencontrée,
  - le caractère `\n` est écrit sur le flux de sortie,
  - on saisit depuis un flux d'entrée **line-buffered**/**unbuffered**.

Changement du mode de bufferisation initial

`setvbuf()` change le mode de bufferisation initial d'un `FILE *`.
La fonction doit être appelée avant la première IO sur le fichier.

```
int setvbuf(FILE *stream, char * buffer, int mode, size_t size);
```

Le mode de bufferisation `mode` est l'une des 3 valeurs...

- `_IONBF` : unbuffered,
- `_IOLBF` : line-buffered,
- `_IOFBF` : fully-buffered.

On peut spécifier son propre `buffer` de caractères de taille `size`,
mais il est conseillé d'utiliser `buffer= NULL` et `size= 0`
pour une gestion de mémoire automatique et de taille optimale.

En effet, ce buffer doit survivre jusqu'à la fermeture du fichier,
ce qui peut parfois arriver **après** la fin de la fonction `main()`,
si le fichier est fermé automatiquement à la mort du processus.

Modes de bufferisation de `stdout` et de `stderr`

La norme ISO requiert que :

- `stderr` n'est jamais **fully-buffered**,
- `stdout` est **fully-buffered** SSI ce n'est pas un terminal.

En pratique, avec les implémentations usuelles, on trouve que :

- `stderr` toujours **unbuffered**,
- `stdout` est **fully-buffered** si ce n'est pas un terminal. Il en va de même pour tous les flux de sortie.
- `stdout` est **line-buffered** si c'est un terminal. Il en va de même pour tous les flux de sortie.

Remarque :

- les tailles des buffers sont souvent 1024 ou 4096 bytes.
- Le mode de bufferisation et la taille du buffer d'un flux est paramétrable en invoquant `setvbuf()` avant sa 1ère IO.

Traçage de la bufferisation de `stdout` avec `strace` (1/3)

Soit le prog `outbuf` qui affiche 1000 fois `"Hello"` sur `stdout` :

```c
// File: outbuf.c
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

int main (int argc, char * argv[]) {
  if (argc != 1+1) {
    fprintf (stderr, "Usage: "); // no newline here on purpose
    fprintf (stderr, "%s -split|*\n", argv[0]);
    exit (1);
  }
  bool split= (strcmp (argv[1], "-split") == 0);
  for (int k = 0; k < 1000; k++) {
    fprintf (stdout, "Hello");
    if (split) fprintf (stdout, "\n");
  }
  if ( ! split) fprintf (stdout, "\n");
  return 0;
}
```

Sa sortie est, selon les options :

```
$ ./outbuf
Usage: ./outbuf -split|*
```

```
$ ./outbuf -nosplit
Hello...Hello
```

```
$ ./outbuf -split
Hello
...
Hello
```

Traçage de la bufferisation de `stdout` avec `strace` (2/3)

`strace -e write -o file cmd` exécute la commande `cmd` tout en écrivant dans `file` ses appels à `write()`.

En utilisant `strace` sur notre prog `outbuf`, on voit que...

- avec `-nosplit`, il y a 5 appels à `write()` de taille ≤ 1024 :

```
$ strace -e write -o nosplit-to-term.log ./outbuf -nosplit
HelloHello...Hello
```

```
$ cat nosplit-to-term.log | nl
     1  write(1, "HelloHelloHelloHelloHelloHelloHe"..., 1024) = 1024
     2  write(1, "oHelloHelloHelloHelloHelloHelloH"..., 1024) = 1024
     3  write(1, "loHelloHelloHelloHelloHelloHello"..., 1024) = 1024
     4  write(1, "lloHelloHelloHelloHelloHelloHell"..., 1024) = 1024
     5  write(1, "elloHelloHelloHelloHelloHelloHel"...,  905) =  905
     6  +++ exited with 0 +++
```

- avec `-split`, il y a 1000 appels à `write()` de taille $= 6$:

```
$ strace -e write -o split-to-term.log ./outbuf -split
Hello
Hello
...
Hello
```

```
$ cat split-to-term.log | nl
     1  write(1, "Hello\n", 6) = 6
     2  write(1, "Hello\n", 6) = 6
...
  1000  write(1, "Hello\n", 6) = 6
  1001  +++ exited with 0 +++
```

Traçage de la bufferisation de `stdout` avec `strace` (3/3)

Quand on redirige `stdout` de `outbuf` vers un pipe, on voit que...

- avec `-nosplit`, il y a 2 appels à `write()` de taille ≤ 4096 :

```
$ strace -e write -o nosplit-to-pipe.log ./outbuf -nosplit | cat
HelloHello...Hello
```

```
$ cat nosplit-to-pipe.log
     1  write(1, "HelloHelloHelloHelloHelloHelloHe"..., 4096) = 4096
     2  write(1, "elloHelloHelloHelloHelloHelloHel"...,  905) =  905
     3  +++ exited with 0 +++
```

- avec `-split` aussi, 2 appels à `write()` de taille ≤ 4096 :

```
$ strace -e write -o split-to-pipe.log ./outbuf -split | cat
Hello
Hello
...
Hello
```

```
$ cat split-to-pipe.log
     1  write(1, "Hello\nHello\nHello\nHello\nHello\nHe"... , 4096) = 4096
     2  write(1, "o\nHello\nHello\nHello\nHello\nHello\n"..., 1904) = 1904
     3  +++ exited with 0 +++
```

On aurait le même résultat avec une redirection vers fichier, donc :

- `stdout` est **line-buffered** à 1024 sur le terminal,
- `stdout` est **fully-buffered** à 4096 sur un pipe ou un fichier.

Traçage de la bufferisation de `stderr` avec `strace` (3/3)

Exécutons `outbuf` sans arg pour qu'il affiche l'usage sur `stderr`...

- avec `stderr` sur le terminal, on a :

```
$ strace -e write -o usage-to-term.log ./outbuf
Usage: ./outbuf -split|*
```

```
$ cat usage-to-term.log | nl
     1  write(2, "Usage: "              ,  7) =  7
     2  write(2, "./outbuf -split|*\n", 18) = 18
     3  +++ exited with 1 +++
```

- et avec `stderr` vers un fichier, on a aussi :

```
$ strace -e write -o usage-to-file.log ./outbuf 2> usage.txt
$ cat usage.txt
Usage: ./outbuf -split|*
```

```
$ cat usage-to-file.log | nl
     1  write(2, "Usage: "              ,  7) =  7
     2  write(2, "./outbuf -split|*\n", 18) = 18
     3  +++ exited with 1 +++
```

Dans les deux cas, on a 1 `write()` par `fprintf()`, indépendamment de la présence d'un `\n`, donc on voit que :

- `stderr` est **unbuffered** sur le terminal,
- `stderr` est **unbuffered** aussi sur fichier.

Modes de bufferisation des flux d'entrée

On retrouve les 3 modes de bufferisation pour les flux d'entrée.
Pour un buffer de taille `BUFSIZ`, le mode a les effets suivants :

- **unbuffered**. Un `read()` ne lit que jusqu'a un caractère,
- **fully-buffered**. Un `read()` lit jusqu'à `BUFSIZ` caractères,
- **line-buffered**. Un `read()` lit jusqu'à `BUFSIZ` caractères, en s'arrêtant au premier `\n` rencontré.

La norme ISO requiert que :

- `stdin` est **fully-buffered** SSI ce n'est pas un terminal.

En pratique, avec les implémentations usuelles, on trouve que :

- `stdin` est **fully-buffered** si ce n'est pas un terminal.
  Il en va de même tous les flux d'entrée.
- `stdin` est **line-buffered** si c'est un terminal.
  Il en va de même tous les flux d'entrée.

Traçage de la bufferisation de `stdin` (1/3)

Soit le prog `inbuf` qui affiche le nb de caractères lus sur `stdin` :

```c
// File: inbuf.c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

int main (int argc, char *  argv[]) {
  bool unbuffered= (argc == 1+1 && strcmp (argv[1], "-unbuffered") == 0);
  if (unbuffered) setvbuf (stdin, NULL, _IONBF, 0);
  int count= 0;
  for (;;) {
    int character= fgetc (stdin);
    if (character == EOF) break;
    count++;
  }
  fprintf (stdout, "counted %d characters.\n", count);
  return 0;
}
```

Un exemple de sortie lorsqu'on saisit depuis le terminal :

```
$ ./inbuf
Hi
Bye
[CTRL-D]
counted 7 characters
```

Un autre exemple lorsqu'on lit 1000 lignes `Hello` depuis un pipe :

```
$ yes Hello | head -n 1000 | ./inbuf
counted 6000 characters.
```

Traçage de la bufferisation de `stdin` (2/3)

`strace -e read -o file cmd` exécute la commande `cmd` tout en écrivant dans `file` ses appels à `read()`.

En utilisant `strace` sur notre prog `inbuf`, on voit que...

- depuis le terminal, il y a 3 appels à `read()` de taille ≤ 1024 :

```
$ strace -e read -o stdin-from-term.log ./inbuf
Hi
Bye
[CTRL-D]
counted 7 characters.
```

```
$ cat stdin-from-term.log | nl
     1  read(3, "\177ELF\2\1\1\3\0"...,  832) = 832 # (loading of libc)
     2  read(0, "Hi\n" , 1024) = 3
     3  read(0, "Bye\n", 1024) = 4
     4  read(0, ""     , 1024) = 0 # (end of file)
     5  +++ exited with 0 +++
```

- depuis un pipe, il y a 3 appels à `read()` de taille ≤ 4096 :

```
$ yes Hello | head -n 1000 | strace -e read -o stdin-from-pipe.log ./inbuf
counted 6000 characters.
```

```
$ cat stdin-from-term.log | nl
     1  read(3, "\177ELF\2\1\1\3\0"...,  832) =  832 # (loading of libc)
     2  read(0, "Hello\nHello\nHello\nHello\nHello\nHe"... , 4096) = 4096
     3  read(0, "o\nHello\nHello\nHello\nHello\nHello\n"..., 4096) = 1904
     4  read(0, "", 4096) = 0 # (end of file)
     5  +++ exited with 0 +++
```

Traçage de la bufferisation de `stdin` (3/3)

On voit donc que par défaut :

- `stdin` est **line-buffered** à 1024 sur le terminal,
- `stdin` est **fully-buffered** à 4096 sur un pipe.

Enfin, via l'option `-unbuffered`, déclenchons `setvbuf()` avec l'argument `_IONBF` qui indique le mode **unbuffered** :

```
$ strace -e read -o stdin-unbuffered.log ./inbuf -unbuffered
Hi
Bye
[CTRL-D]
counted 7 characters
```

Chaque caractère lu est alors obtenu par un `read()` de taille = 1 :

```
$ cat stdin-unbuffered.log | nl
     1  read(3, "\177ELF\3\1\1\3\0"..., 832) = 832 # (loading of libc)
     2  read(0, "H",  1) = 1
     3  read(0, "i",  1) = 1
     4  read(0, "\n", 1) = 1
     5  read(0, "B",  1) = 1
     6  read(0, "y",  1) = 1
     7  read(0, "e",  1) = 1
     8  read(0, "\n", 1) = 1
     9  read(0, "",   1) = 0 # (end of file)
    10  +++ exited with 0 +++
```