

MICL : TD05 : Appels système

ABS – APA – DBO – EGR – HAL – NVS – SRE *



Année académique 2019 – 2020

Dans ce [TD¹](#), les appels systèmes sous GNU/Linux sont abordés. Concomitamment, la manière de représenter les immédiats chaînes de caractères avec `nasm` est exposée.

1 Définition

Les [appels système²](#) (*system calls*) sont des services offerts par le système d'exploitation pour effectuer diverses tâches, comme ouvrir ou lire le contenu d'un fichier. Chaque appel système est identifié par un numéro appelé *numéro de service*. On utilise les appels système comme les méthodes du cours d'algorithmique : ils ont des paramètres entrants et peuvent retourner une valeur. L'appel système en lui-même se fait au travers de l'instruction [syscall³](#). Celle-ci a comme effet de basculer le [CPU⁴](#) (*central processing unit*, processeur) en mode privilégié et passer la main au service système demandé, identifié par son numéro.

2 Mise en œuvre

Sous GNU/Linux 64 bits, un [appel système⁵](#) en langage d'assemblage se fait en quatre étapes :

*Et aussi, lors des années passées : BEJ – DWI – ELV – FPL – JDS – MBA – MCD – MHI – MWA.

1. https://poesi.esi-bru.be/pluginfile.php/5706/mod_folder/content/0/td05_micl.pdf (consulté le 21 février 2020).

2. https://fr.wikipedia.org/wiki/Appel_syst%C3%A8me (consulté le 21 février 2020).

3. <https://www.felixcloutier.com/x86/syscall> (consulté le 21 février 2020).

4. https://en.wikipedia.org/wiki/Central_processing_unit (consulté le 25 février 2020).

5. <https://blog.packagecloud.io/eng/2016/04/05/the-definitive-guide-to-linux-system-calls/#64-bit-fast-system-calls> (consulté le 21 février 2020).

1. Placer le numéro du service désiré dans `rax`.
2. Mettre les paramètres, s'il y en a, dans `rdi`, `rsi`, `rdx`, `rcx`, `r8` et `r9`.
3. Appeler le système par l'instruction `syscall`.
4. Consulter dans `rax` la valeur de retour, s'il y en a une, ou le statut d'erreur, si nécessaire ou utile.

Les étapes 1 et 2 peuvent se faire dans l'ordre qu'on veut. Ce qui compte, c'est que tout soit prêt avant d'exécuter l'instruction `syscall`. Ainsi, les assignations de l'étape 2 peuvent se faire dans l'ordre qu'on souhaite. Par contre, le numéro du service *doit* être renseigné via `rax`. De plus, si l'appel système attend des paramètres (voir la section 5), le premier *doit* se trouver dans `rdi`, le deuxième dans `rdx`, etc. en respectant l'ordre indiqué au point 2. En particulier, on peut déduire qu'un appel système a au plus six arguments.

3 Registres non préservés : `rcx`, `r11` et `rax`

Il est important de savoir que l'instruction `syscall` utilise les registres :

- `rcx` pour la sauvegarde de la valeur de `rip` : cela permet le retour au code appelant.
- `r11` pour la sauvegarde de la valeur du registre `rflags` et sa restauration lors du retour au code appelant.

Donc attention, si les contenus des registres `rcx` et `r11` sont importants, il doivent être sauvegardés par ailleurs avant l'utilisation de `syscall`.

Le registre `rax` est lui aussi modifié par `syscall`. La valeur de retour de l'appel système y est stockée. Comme ce registre est de toute façon utilisé pour indiquer le numéro de l'appel système, cela pose moins problème que les modifications de `rcx` et `r11`.

4 Numéro du service

Pour connaître la liste des services du système d'exploitation et les numéros correspondants, il faut consulter le fichier `/usr/include/asm-x86/unistd_64.h`⁶.

Voici un extrait d'une version de ce fichier :

```

1 #ifndef _ASM_X86_UNISTD_64_H
2 #define _ASM_X86_UNISTD_64_H 1
3
4 #define __NR_read 0
5 #define __NR_write 1
6 #define __NR_open 2
7 #define __NR_close 3
8 #define __NR_stat 4

```

6. https://code.woboq.org/userspace/include/asm/unistd_64.h.html (consulté le 21 février 2020).

```

9 // ...
10 #define __NR_clone 56
11 #define __NR_fork 57
12 #define __NR_vfork 58
13 #define __NR_execve 59
14 #define __NR_exit 60
15 #define __NR_wait4 61
16 #define __NR_kill 62
17 #define __NR_uname 63
18 // ...
19 #define __NR_pkey_free 331
20 #define __NR_statx 332
21 #define __NR_io_pgetevents 333
22 #define __NR_rseq 334
23
24 #endif /* _ASM_X86_UNISTD_64_H */

```

On y remarque, par exemple, que le service `exit`⁷ a le numéro 60, `open`⁸ est identifié par le chiffre 2 et `write`⁹ par le 1.

5 Paramètres et retour

Pour connaître les paramètres attendus par un appel système ou savoir s'il retourne une valeur, il faut consulter la [section 2](#)¹⁰ du [manuel GNU/Linux](#)¹¹.

5.1 exit

Pour le service `exit`, il faut lancer la commande :

```
man 2 exit
```

Cela provoque l'affichage du texte suivant :

```

1 _EXIT(2)                                Linux Programmer's Manual                                _EXIT(2)
2
3 NAME
4     _exit, _Exit - terminate the calling process
5
6 SYNOPSIS
7     #include <unistd.h>
8
9     void _exit(int status);

```

7. http://manpagesfr.free.fr/man/man2/_exit.2.html (consulté le 21 février 2020).

8. <http://manpagesfr.free.fr/man/man2/open.2.html> (consulté le 21 février 2020).

9. <http://manpagesfr.free.fr/man/man2/write.2.html> (consulté le 21 février 2020).

10. http://man7.org/linux/man-pages/dir_section_2.html (consulté le 21 février 2020).

11. <https://www.kernel.org/doc/man-pages/> (consulté le 21 février 2020).

```

10
11     #include <stdlib.h>
12
13     void _Exit(int status);
14
15     Feature Test Macro Requirements for glibc (see feature_test_macros(7)):
16
17     _Exit():
18         _ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
19
20 DESCRIPTION
21     The function _exit() terminates the calling process "immediately". Any
22     open file descriptors belonging to the process are closed. Any chil-
23     dren of the process are inherited by init(1) (or by the nearest "sub-
24     reaper" process as defined through the use of the prctl(2)
25     PR_SET_CHILD_SUBREAPER operation). The process's parent is sent a
26     SIGCHLD signal.
27
28     The value status & 0377 is returned to the parent process as the
29     process's exit status, and can be collected using one of the wait(2)
30     family of calls.
31
32     The function _Exit() is equivalent to _exit().
33
34 RETURN VALUE
35     These functions do not return.
36
37 CONFORMING TO
38     POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD. The function _Exit() was
39     introduced by C99.
40
41 NOTES
42     For a discussion on the effects of an exit, the transmission of exit
43     status, zombie processes, signals sent, and so on, see exit(3).
44
45     The function _exit() is like exit(3), but does not call any functions
46     registered with atexit(3) or on_exit(3). Open stdio(3) streams are not
47     flushed. On the other hand, _exit() does close open file descriptors,
48     and this may cause an unknown delay, waiting for pending output to fin-
49     ish. If the delay is undesired, it may be useful to call functions
50     like tcflush(3) before calling _exit(). Whether any pending I/O is
51     canceled, and which pending I/O may be canceled upon _exit(), is imple-
52     mentation-dependent.
53
54     C library/kernel differences
55     In glibc up to version 2.3, the _exit() wrapper function invoked the
56     kernel system call of the same name. Since glibc 2.3, the wrapper
57     function invokes exit_group(2), in order to terminate all of the
58     threads in a process.
59
60 SEE ALSO
61     execve(2), exit_group(2), fork(2), kill(2), wait(2), wait4(2), wait-

```

```

62     pid(2), atexit(3), exit(3), on_exit(3), termios(3)
63
64 COLOPHON
65     This page is part of release 4.16 of the Linux man-pages project. A
66     description of the project, information about reporting bugs, and the
67     latest version of this page, can be found at
68     https://www.kernel.org/doc/man-pages/.
69
70 Linux                                2017-05-03                                _EXIT(2)

```

Les parties qui nous intéressent sont sous les titres **SYNOPSIS**, **DESCRIPTION** et **RETURN VALUE**.

La partie **SYNOPSIS** contient, entre autre, le ou les **prototypes**¹² de la ou des fonctions C associées à l'appel système. Il s'agit ici de :

```

1 void _exit(int status);

```

5.1.1 Retour

Le mot clé **void** indique l'absence de valeur retournée.

5.1.2 Argument

L'**int** en argument doit être fourni via **rdi**.

Pour connaître la signification de l'argument, il faut lire la **DESCRIPTION**. Dans le cas présent, il s'agit de la valeur retournée par le programme au moment de sa fin à son processus parent. Cela correspond à un statut de fin qui décrit si le processus s'est bien terminé ou non. **Par convention**¹³, un statut de fin nul, c'est-à-dire égal à zéro, indique que la commande s'est bien terminée, tandis qu'une valeur non nulle signifie l'existence d'un problème.

Pour **récupérer dans le shell**¹⁴ le statut de fin d'un processus, il faut consulter la variable d'environnement `?` après la mort du processus :

```
echo $?
```

Vous devriez maintenant comprendre la signification des trois dernières lignes des codes sources assembleurs produits jusqu'à présent.

5.2 open

```
man 2 open
```

donne 3 prototypes :

12. https://en.wikipedia.org/wiki/Function_prototype (consulté le 21 février 2020).

13. https://en.wikipedia.org/wiki/Exit_status#Shell_and_scripts (consulté le 21 février 2020).

14. <https://thegeeksalive.com/how-to-check-exit-status-of-linux-commands/> (consulté le 21 février 2020).

```

1 int open(const char *pathname, int flags);
2 int open(const char *pathname, int flags, mode_t mode);
3
4 int creat(const char *pathname, mode_t mode);

```

La consultation du fichier `unistd_64.h` renseigne que l'appel système `creat` est le service 85. Comme signalé dans la page de manuel, en toute fin de `DESCRIPTION`, il peut être obtenu à l'aide d'`open`, avec un bon choix de paramètres. Concentrons-nous donc sur `open`.

5.2.1 Arguments

Premier argument Le type du premier argument est `const char *`. L'étoile `*` indique que la variable `pathname` est un *pointeur*¹⁵. `char *` signifie plus précisément *pointeur de caractère*. Et `const char *` *pointeur de caractère constant*. Les types se lisent de droite à gauche en langage C. Le premier argument attendu est donc l'*adresse* (`*`) d'un caractère (`char`) qui n'est pas modifié (`const`) par `open`. La lecture de la description signale que `pathname` est un chemin vers un fichier. Il s'agit d'une chaîne de caractères. En langage C, les chaînes de caractères sont des tableaux de caractères *zéro-terminés*¹⁶ : leur dernier caractère est la marque de fin de chaîne. Il s'agit du caractère de code nul (0). N'oubliez jamais de terminer par 0 tout argument de type `char *`.

Deuxième argument C'est un `int` permettant de fournir les valeurs d'une série d'indicateurs : `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_APPEND`, `O_CREAT`, `O_TRUNC`, etc. La signification de ces constantes est donnée dans la description d'`open`. Les valeurs de ces constantes sont disponibles dans le fichier `/usr/include/bits/fcntl.h`^{17 18}. En voici un extrait :

```

1 /* O_*, F_*, FD_* bit values for Linux.
2    Copyright (C) 2001-2018 Free Software Foundation, Inc.
3    This file is part of the GNU C Library.
4
5    The GNU C Library is free software; ... */
6
7 #ifndef _FCNTL_H
8 # error "Never use <bits/fcntl-linux.h> directly; include <fcntl.h> instead."
9 #endif
10
11 // ...
12
13 /* open/fcntl. */
14 #define O_ACCMODE      0003

```

15. [https://fr.wikipedia.org/wiki/Pointeur_\(programmation\)](https://fr.wikipedia.org/wiki/Pointeur_(programmation)) (consulté le 22 février 2020).

16. https://en.wikipedia.org/wiki/Null-terminated_string (consulté le 21 février 2020).

17. <https://code.woboq.org/gcc/include/bits/fcntl-linux.h.html> (consulté le 21 février 2020).

18. Ou `fcntl-linux.h`.

```

15 #define O_RDONLY      00
16 #define O_WRONLY      01
17 #define O_RDWR        02
18 #ifndef O_CREAT
19 # define O_CREAT      0100      /* Not fcntl. */
20 #endif
21 #ifndef O_EXCL
22 # define O_EXCL        0200      /* Not fcntl. */
23 #endif
24 #ifndef O_NOCTTY
25 # define O_NOCTTY      0400      /* Not fcntl. */
26 #endif
27 #ifndef O_TRUNC
28 # define O_TRUNC       01000     /* Not fcntl. */
29 #endif
30 #ifndef O_APPEND
31 # define O_APPEND      02000
32 #endif
33
34 // ...

```

Il faut savoir qu'un **littéral entier**¹⁹ qui commence par un zéro (0) dans un code en langage C est exprimé en **octal**²⁰. Il est possible de combiner plusieurs *flags* en composant ces constantes à l'aide du *ou bit à bit*. Cette combinaison peut se faire lors de l'exécution à l'aide de l'instruction **or** (voir TD02) ou, si possible et alors avantageusement, lors de l'assemblage en utilisant l'**opérateur** `|`²¹.

Troisième argument Il n'est attendu que si le *flag* `O_CREAT` apparaît dans le second argument. On trouve davantage de détails dans l'explication de l'indicateur `O_CREAT` dans la description de l'appel système **open**.

5.2.2 Retour

Si l'ouverture ou la création du fichier s'est bien passée, la *valeur retournée* par **open** est un petit entier positif. Cet entier porte le nom de *descripteur de fichier*²² (*file descriptor*). C'est par son biais qu'on peut ensuite accéder, écrire, lire ou fermer un fichier ouvert. Il s'agit donc de ne pas égarer cette information. Si l'ouverture échoue, un entier négatif est retourné.

5.2.3 Exemple

Voici un exemple d'utilisation d'**open** dans un source assembleur :

19. https://en.cppreference.com/w/c/language/integer_constant (consulté le 21 février 2020).

20. <https://en.wikipedia.org/wiki/Octal> (consulté le 21 février 2020).

21. <https://www.nasm.us/doc/nasmdoc3.html#section-3.5.1> (consulté le 21 février 2020).

22. https://en.wikipedia.org/wiki/File_descriptor (consulté le 21 février 2020).

```

1 ; 01_open_extrait.asm
2
3 ; etc.
4
5 section .rodata
6     nomFichier    DB    `brol`, 0    ; ne pas oublier le 0
7
8 ; etc.
9
10 section .text
11 main:
12
13 ; etc.
14
15     ; ouverture de brol en écriture seule avec placement
16     ; de la tête d'écriture en fin de fichier
17     mov     rax, 2        ; open
18     mov     rdi, nomFichier ; /adresse/ du 1er caractère du nom
19     mov     rsi, 1q | 2000q ; WRONLY + O_APPEND
20     syscall
21
22 ; etc.

```

5.2.4 Chaîne de caractères avec nasm

Comme il a été indiqué plus haut, en langage C, une chaîne de caractères est un tableau de caractères zero-terminé. En langage d'assemblage, une chaîne de caractères est un tableau de caractères, non nécessairement *zero-terminé*.

La notion de tableau en langage d'assemblage est détaillée dans le TD06. Dans le présent TD, on utilise uniquement des [immédiats de type chaîne de caractères](#)²³.

Avec `nasm`, les [littéraux caractères](#)²⁴ peuvent être renseignés entre apostrophes (*single quotes*, « `'` »), guillemets (*double quotes*, « `"` ») ou accents graves (*back quotes*, « ``` », ALT GR + μ). Si on désire utiliser les [séquences d'échappement](#)²⁵ semblables à celles du langage C, il faut utiliser les *back quotes*.

Les littéraux chaînes de caractères sont des variables immuables déclarées dans la `section .rodata`. La taille sous-jacente pour les caractères que nous conseillons d'utiliser est le *byte* via la pseudo-instruction `DB`.

Exemple Voici un source où plusieurs chaînes de caractères²⁶ sont déclarées :

23. <https://www.nasm.us/xdoc/2.14.02/html/nasmdoc3.html#section-3.4.4> (consulté le 22 février 2020).

24. <https://www.nasm.us/xdoc/2.14.02/html/nasmdoc3.html#section-3.4.2> (consulté le 22 février 2020).

25. <https://en.cppreference.com/w/c/language/escape> (consulté le 22 février 2020).

26. Attention, `str` (<https://www.felixcloutier.com/x86/str>) correspond à la mnémonique de l'instruction *store task register*, elle ne peut donc servir comme nom de variable.


```

1 section .rodata
2   ; ' et " sont équivalents; ` permet les séquences d'échappement
3   str1    DB      "abc"          ; 3 bytes initialisés
4   str2    DB      'abc', 0       ; 4 (3 + 1) bytes initialisés, str2 zéro-terminée
5   str3    DB      "abc\n", 0     ; 6 (5 + 1) bytes initialisés, str3 zéro-terminée
6   str4    DB      `abc\n`, 0     ; 5 (4 + 1) bytes initialisés, str4 zéro-terminée

```

Taille d'une chaîne de caractères On peut faire calculer par `nasm` la taille en *bytes* d'une chaîne de caractères²⁷. Au moment de l'assemblage, les étiquettes (*labels*) sont remplacées par les adresses où elles ont été collées. Par ailleurs `nasm` est capable de calculer des *différences*²⁸. Si on demande à `nasm` de calculer la différence de deux *labels*, on obtient le nombre d'emplacements adressables entre ces deux étiquettes, c'est-à-dire une taille en *bytes*²⁹. Signalons aussi, que le caractère `$`³⁰ est remplacé par `nasm` par l'adresse courante où il apparaît.

Exemple Voici un source où la taille de chaînes de caractères est calculée par `nasm` :

```

1 section .rodata
2   str0     DB      "abc"
3   lgrStr0  DQ      lgrStr0 - str0 ; taille en bytes de str0 : 3
4   str1     DB      `abc\n`, 0
5   lgrStr1  DQ      $ - str1       ; taille en bytes de str1 : 5

```

5.3 write

Le prototype de `write` est :

```

1 ssize_t write(int fd, const void *buf, size_t count);

```

Un exemple d'invocation de `write` est donné à la section 7.

5.3.1 Arguments

Premier Argument Il s'agit du descripteur de fichier du fichier où l'écriture doit avoir lieu.

27. On peut le faire pour n'importe quel tableau.

28. <https://www.nasm.us/xdoc/2.14/html/nasmdoc3.html#section-3.5.5> (consulté le 22 février 2020).

29. <https://fr.wikipedia.org/wiki/Byte> (consulté le 23 février 2020).

30. <https://www.nasm.us/xdoc/2.14/html/nasmdoc3.html#section-3.5> (consulté le 23 février 2020).

Flux	Fichier	Descripteur de fichier	Association
entrée standard	<code>stdin</code>	0	clavier
sortie standard	<code>stdout</code>	1	écran
sortie d'erreur standard	<code>stderr</code>	2	écran

TABLE 1 – Descripteurs de fichier associés aux flux standards.

Deuxième argument C'est un *pointeur générique*³¹ vers une zone constante. Le contenu de cette zone est ce qui doit être écrit dans le fichier. Il peut s'agir de données textuelles ou binaires. Contrairement aux chaînes de caractères (`char *`) qui sont zéro-terminées, une zone mémoire identifiée par un pointeur générique (`void *`) ne doit remplir aucune exigence particulière.

Troisième argument À défaut d'une valeur sentinelle indiquant la fin de la zone à écrire, un *troisième paramètre* est nécessaire : la taille, en *bytes*, de cette zone.

5.3.2 Retour

`write` retourne le nombre de *bytes* effectivement écrits dans le fichier.

6 Entrée et sorties standards

Au démarrage d'un programme, *trois flux*³² sont disponibles sans devoir être ouverts.

Le flux d'entrée standard, `stdin`, associé par défaut au clavier, peut être accédé comme un fichier avec 0 comme valeur de descripteur de fichier.

Le flux de sortie standard, `stdout`, associé par défaut à l'écran, peut être accédé comme un fichier avec 1 comme valeur de descripteur de fichier.

Le flux de sortie d'erreur standard, `stderr`, également associé par défaut à l'écran, peut être accédé comme un fichier avec 2 comme valeur de descripteur de fichier. La TABLE 1 résume ceci.

7 Hello, World!

Voici *Hello, World!*³³ en langage d'assemblage :

```

1 ; 02_hello_world.asm
2
3 global _start
4
```

31. <https://www.cquestions.com/2009/11/generic-pointer-in-c-programming.html> (consulté le 22 février 2020).

32. <http://manpagesfr.free.fr/man/man3/stdin.3.html> (consulté le 21 février 2020).

33. https://en.wikipedia.org/wiki/%22Hello,_World!%22_program (consulté le 22 février 2020).

```

5  section .rodata
6      msg      DB      `Hello, World!\n`
7      lgrMsg   DQ      lgrMsg - msg
8
9  section .text
10 _start:
11     ; affichage
12     mov     rax, 1      ; write
13     mov     rdi, 1      ; stdout, sortie standard
14     mov     rsi, msg    ; adresse du 1er caractère
15     mov     rdx, [lgrMsg] ; nombre de caractères
16     syscall
17
18 fin:
19     mov     rax, 60      ; exit
20     mov     rdi, 0      ; ok
21     syscall

```

8 Cinq appels système

La TABLE 2 montre les cinq appels système que nous utilisons le plus dans ce TD et les suivants.

9 Exercices

Ex. 1 Soit le code source :

```

1  ; 01_open_exercice.asm
2
3  global     main
4
5  section .rodata
6      nomFichier      DB      `brol`, 0      ; ne pas oublier le 0
7
8  section .text
9  main:
10     ; ouverture de brol en écriture seule avec placement
11     ; de la tête d'écriture en fin de fichier
12     mov     rax, 2      ; open
13     mov     rdi, nomFichier ; adresse du 1er caractère du nom
14     mov     rsi, 1q | 2000q ; WRONLY + O_APPEND
15     syscall
16
17     mov     rax, 60
18     mov     rdi, 0
19     syscall

```

Service	Numéro	But	Paramètres	Retour	Notes
exit	60	quitter un programme	un entier à retourner au processus parent, 0 si tout ok	aucun	à mettre en fin de tout programme
open	2	ouvrir ou créer un fichier	<ul style="list-style-type: none"> — chemin vers le fichier ; — chaîne zéro-terminée ; — options d'ouverture ; indicateurs à combiner avec <code> </code> ; — mode : si création d'un fichier 	descripteur de fichier ou entier négatif en cas d'erreur	options d'ouverture : <code>/usr/include/bits/fcntl.h</code> (voir p. 6)
close	3	fermer un fichier	descripteur du fichier	0 si ok, -1 si erreur	le descripteur de fichier est celui retourné par open
read	0	lire depuis un fichier	<ul style="list-style-type: none"> — descripteur du fichier ; — adresse où stocker le résultat de la lecture ; — nombre de <i>bytes</i> à lire 	nombre d'octets effectivement lus, -1 en cas d'erreur	<ul style="list-style-type: none"> — le descripteur de fichier est celui retourné par open ou 0 pour lire au clavier ; — la tête de lecture est avancée du nombre de <i>bytes</i> lus
write	1	écrire dans un fichier	<ul style="list-style-type: none"> — descripteur du fichier ; — adresse de ce qui doit être écrit ; — nombre de <i>bytes</i> à écrire 	nombre d'octets effectivement écrits, -1 en cas d'erreur	<ul style="list-style-type: none"> — le descripteur de fichier est celui retourné par open ou 1 pour écrire à l'écran ; — la tête d'écriture est avancée du nombre de <i>bytes</i> écrits

TABLE 2 – Cinq appels système.

1. Inspectez le contenu du registre `rax` dans les cas suivants :

- a) le fichier `bro1` existe avec droit en écriture ;
- b) le fichier `bro1` existe sans droit en écriture ;
- c) le fichier `bro1` n'existe pas.

2. Mettez en commentaire la ligne :

```
1  mov     rsi, 1q | 2000q
```

et insérez les lignes :

```
1  mov     rsi, 1q | 100q | 2000q    ; WRONLY + O_CREAT + O_APPEND
2  mov     rdx, 755q                ; droits du fichier créé
```

juste avant l'appel `syscall`.

Reproduisez alors les 3 cas répertoriés au point 1.

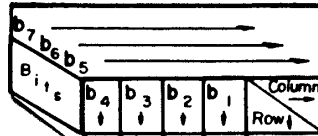
Ex. 2 Soit un fichier dont le nom est stocké dans une variable. Écrivez un programme qui :

1. tente d'ouvrir le fichier en lecture seule ;
2. si l'ouverture échoue :
 - a) affiche à l'écran le message :
 échec lors de l'ouverture du fichier
 - b) passe à la ligne ;
 - c) retourne le code d'erreur 1 ;
3. si l'ouverture réussit :
 - a) affiche à l'écran le message :
 fichier ouvert avec succès
 - b) passe à la ligne ;
 - c) ferme le fichier ;
 - d) retourne la valeur 0 pour indiquer un dénouement correct.

Ex. 3 Écrivez un programme qui affiche à l'écran le contenu *converti en caractère* de `rsi` dont on garantit qu'il est compris entre 0 et 9. N'hésitez pas à consulter la table ASCII fournie à la FIG. 1.

Que se passe-t-il si le contenu de `rsi` n'est pas entre 0 et 9 ?

USASCII code chart



	b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	b ₁	b ₀	
	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	1	0	1
	0	0	0	0	0	1	0	0	0
	0	0	0	0	0	1	0	1	1
	0	0	0	0	1	0	0	0	0
	0	0	0	0	1	0	0	1	0
	0	0	0	0	1	0	1	0	1
	0	0	0	0	1	1	0	0	0
	0	0	0	0	1	1	0	1	0
	0	0	0	0	1	1	1	0	0
	0	0	0	0	1	1	1	1	0
	0	0	1	0	0	0	0	0	0
	0	0	1	0	0	0	0	1	0
	0	0	1	0	0	0	1	1	0
	0	0	1	0	0	1	0	0	0
	0	0	1	0	0	1	0	1	0
	0	0	1	0	0	1	1	0	0
	0	0	1	0	0	1	1	1	0
	0	0	1	0	1	0	0	0	0
	0	0	1	0	1	0	0	1	0
	0	0	1	0	1	0	1	0	0
	0	0	1	0	1	0	1	1	0
	0	0	1	0	1	1	0	0	0
	0	0	1	0	1	1	0	1	0
	0	0	1	0	1	1	1	0	0
	0	0	1	0	1	1	1	1	0
	0	1	0	0	0	0	0	0	0
	0	1	0	0	0	0	0	1	0
	0	1	0	0	0	0	1	1	0
	0	1	0	0	0	1	0	0	0
	0	1	0	0	0	1	0	1	0
	0	1	0	0	0	1	1	0	0
	0	1	0	0	0	1	1	1	0
	0	1	0	0	1	0	0	0	0
	0	1	0	0	1	0	0	1	0
	0	1	0	0	1	0	1	0	0
	0	1	0	0	1	0	1	1	0
	0	1	0	0	1	1	0	0	0
	0	1	0	0	1	1	0	1	0
	0	1	0	0	1	1	1	0	0
	0	1	0	0	1	1	1	1	0
	0	1	0	1	0	0	0	0	0
	0	1	0	1	0	0	0	1	0
	0	1	0	1	0	0	1	0	0
	0	1	0	1	0	0	1	1	0
	0	1	0	1	0	1	0	0	0
	0	1	0	1	0	1	0	1	0
	0	1	0	1	0	1	1	0	0
	0	1	0	1	0	1	1	1	0
	0	1	0	1	1	0	0	0	0
	0	1	0	1	1	0	0	1	0
	0	1	0	1	1	0	1	0	0
	0	1	0	1	1	0	1	1	0
	0	1	0	1	1	1	0	0	0
	0	1	0	1	1	1	0	1	0
	0	1	0	1	1	1	1	0	0
	0	1	0	1	1	1	1	1	0
	0	1	1	0	0	0	0	0	0
	0	1	1	0	0	0	0	1	0
	0	1	1	0	0	0	1	0	0
	0	1	1	0	0	0	1	1	0
	0	1	1	0	0	1	0	0	0
	0	1	1	0	0	1	0	1	0
	0	1	1	0	0	1	1	0	0
	0	1	1	0	0	1	1	1	0
	0	1	1	0	1	0	0	0	0
	0	1	1	0	1	0	0	1	0
	0	1	1	0	1	0	1	0	0
	0	1	1	0	1	0	1	1	0
	0	1	1	0	1	1	0	0	0
	0	1	1	0	1	1	0	1	0
	0	1	1	0	1	1	1	0	0
	0	1	1	0	1	1	1	1	0
	0	1	1	1	0	0	0	0	0
	0	1	1	1	0	0	0	1	0
	0	1	1	1	0	0	1	0	0
	0	1	1	1	0	0	1	1	0
	0	1	1	1	0	1	0	0	0
	0	1	1	1	0	1	0	1	0
	0	1	1	1	0	1	1	0	0
	0	1	1	1	0	1	1	1	0
	0	1	1	1	1	0	0	0	0
	0	1	1	1	1	0	0	1	0
	0	1	1	1	1	0	1	0	0
	0	1	1	1	1	0	1	1	0
	0	1	1	1	1	1	0	0	0
	0	1	1	1	1	1	0	1	0
	0	1	1	1	1	1	1	0	0
	1	0	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	1	0
	1	0	0	0	0	0	1	0	0
	1	0	0	0	0	0	1	1	0
	1	0	0	0	0	1	0	0	0
	1	0	0	0	0	1	0	1	0
	1	0	0	0	0	1	1	0	0
	1	0	0	0	0	1	1	1	0
	1	0	0	0	1	0	0	0	0
	1	0	0	0	1	0	0	1	0
	1	0	0	0	1	0	1	0	0
	1	0	0	0	1	0	1	1	0
	1	0	0	0	1	1	0	0	0
	1	0	0	0	1	1	0	1	0
	1	0	0	0	1	1	1	0	0
	1	0	0	0	1	1	1	1	0
	1	0	0	1	0	0	0	0	0
	1	0	0	1	0	0	0	1	0
	1	0	0	1	0	0	1	0	0
	1	0	0	1	0	0	1	1	0
	1	0	0	1	0	1	0	0	0
	1	0	0	1	0	1	0	1	0
	1	0	0	1	0	1	1	0	0
	1	0	0	1	0	1	1	1	0
	1	0	0	1	1	0	0	0	0
	1	0	0	1	1	0	0	1	0
	1	0	0	1	1	0	1	0	0
	1	0	0	1	1	0	1	1	0
	1	0	0	1	1	1	0	0	0
	1	0	0	1	1	1	0	1	0
	1	0	0	1	1	1	1	0	0
	1	0	0	1	1	1	1	1	0
	1	0	1	0	0	0	0	0	0
	1	0	1	0	0	0	0	1	0
	1	0	1	0	0	0	1	0	0
	1	0	1	0	0	0	1	1	0
	1	0	1	0	0	1	0	0	0
	1	0	1	0	0	1	0	1	0
	1	0	1	0	0	1	1	0	0
	1	0	1	0	0	1	1	1	0
	1	0	1	0	1	0	0	0	0
	1	0	1	0	1	0	0	1	0
	1	0	1	0	1	0	1	0	0
	1	0	1	0	1	0	1	1	0
	1	0	1	0	1	1	0	0	0
	1	0	1	0	1	1	0	1	0
	1	0	1	0	1	1	1	0	0
	1	0	1	0	1	1	1	1	0
	1	0	1	1	0	0	0	0	0
	1	0	1	1	0	0	0	1	0
	1	0	1	1	0	0	1	0	0
	1	0	1	1	0	0	1	1	0
	1	0	1	1	0	1	0	0	0
	1	0	1	1	0	1	0	1	0
	1	0	1	1	0	1	1	0	0
	1	0	1	1	0	1	1	1	0
	1	0	1	1	1	0	0	0	0
	1	0	1	1	1	0	0	1	0
	1	0	1	1	1	0	1	0	0
	1	0	1	1	1	0	1	1	0
	1	0	1	1	1	1	0	0	0
	1	0	1	1	1	1	0	1	0
	1	0	1	1	1	1	1	0	0
	1	0	1	1	1	1	1	1	0
	1	1	0	0	0	0	0	0	0
	1	1	0	0	0	0	0	1	0
	1	1	0	0	0	0	1	0	0
	1	1	0	0	0	0	1	1	0
	1	1	0	0	0	1	0	0	0
	1	1	0	0	0	1	0	1	0
	1	1	0	0	0	1	1	0	0
	1	1	0	0	0	1	1	1	0
	1	1	0	0	1	0	0	0	0
	1	1	0						

Ex. 6 Écrivez un programme qui stocke dans `rcx` la taille en *bytes* du fichier `bro1`. Si ce fichier ne peut être ouvert, le code d'erreur 3 est retourné ; sinon, n'oubliez pas de bien le fermer !

Aide : Utilisez `lseek`³⁵ sachant que `SEEK_SET` vaut 0, `SEEK_CUR` 1 et `SEEK_END` 2.

Notions à retenir

Appel système, chaîne de caractères.

Références

- [1] Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4, octobre 2017. <https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf>.
- [2] Igor Zhirkov. *Low-Level Programming*. Apress, 2017. <https://www.apress.com/gp/book/9781484224021>.

35. <http://manpagesfr.free.fr/man/man2/lseek.2.html> (consulté le 21 février 2020).