HE2B ESI

MICL : TD05 : Appels système

ABS – BEJ – DBO – HAL – NVS – SRE – YVO *

Année académique 2021 – 2022

Dans ce TD, 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 système [1] (*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 `syscall` [2]. Celle-ci a comme effet de basculer le *CPU* [3] (*central processing unit*, processeur) en mode privilégié et passer la main au service système demandé, identifié par son numéro. À la fin de l'exécution du code système, l'exécution du code appelant reprend à l'instruction qui suit directement l'appel `syscall` [4].

2 Mise en œuvre

Sous GNU/Linux 64 bits, un appel système [5] en langage d'assemblage se fait en quatre étapes :

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

1. https://fr.wikipedia.org/wiki/Appel_syst%C3%A8me (consulté le 21 février 2020).
2. https://www.felixcloutier.com/x86/syscall (consulté le 21 février 2020).
3. https://en.wikipedia.org/wiki/Central_processing_unit (consulté le 25 février 2020).
4. Sauf bien entendu pour l'appel système `exit` (voir section 5.1).
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`, `r10`, `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 du registre `rip` : cela permet, à la fin de l'appel système, le retour au code appelant précisément à l'instruction qui suit `syscall` par la restauration de cette valeur sauvegardée.

— `r11` pour la sauvegarde 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 également 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/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. https://code.woboq.org/userspace/include/asm/unistd_64.h.html (consulté le 21 février 2020).

```
6  #define __NR_open 2
7  #define __NR_close 3
8  #define __NR_stat 4
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
```

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).

```
SYNOPSIS
       #include <unistd.h>

       void _exit(int status);

       #include <stdlib.h>

       void _Exit(int status);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       _Exit():
           _ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION
       The function _exit() terminates the calling process "immediately".  Any
       open file descriptors belonging to the process are closed.  Any chil-
       dren  of  the process are inherited by init(1) (or by the nearest "sub-
       reaper"  process  as  defined  through  the  use  of  the  prctl(2)
       PR_SET_CHILD_SUBREAPER  operation).  The  process's  parent  is sent a
       SIGCHLD signal.

       The value status & 0377 is  returned  to  the  parent  process  as  the
       process's  exit  status,  and can be collected using one of the wait(2)
       family of calls.

       The function _Exit() is equivalent to _exit().

RETURN VALUE
       These functions do not return.

CONFORMING TO
       POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD.  The  function  _Exit()  was
       introduced by C99.

NOTES
       For  a  discussion  on the effects of an exit, the transmission of exit
       status, zombie processes, signals sent, and so on, see exit(3).

       The function _exit() is like exit(3), but does not call  any  functions
       registered with atexit(3) or on_exit(3).  Open stdio(3) streams are not
       flushed.  On the other hand, _exit() does close open file  descriptors,
       and this may cause an unknown delay, waiting for pending output to fin-
       ish.  If the delay is undesired, it may be  useful  to  call  functions
       like  tcflush(3)  before  calling  _exit().  Whether any pending I/O is
       canceled, and which pending I/O may be canceled upon _exit(), is imple-
       mentation-dependent.

   C library/kernel differences
       In  glibc  up  to version 2.3, the _exit() wrapper function invoked the
       kernel system call of the same name.  Since  glibc  2.3,  the  wrapper
       function  invokes  exit_group(2),  in  order  to  terminate  all of the
```

```
       threads in a process.

SEE ALSO
       execve(2), exit_group(2), fork(2), kill(2), wait(2), wait4(2), wait-
       pid(2), atexit(3), exit(3), on_exit(3), termios(3)

COLOPHON
       This page is part of release 4.16 of the Linux man-pages project. A
       description of the project, information about reporting bugs, and the
       latest version of this page, can be found at
       https://www.kernel.org/doc/man-pages/.

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.

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).

5.2 open

`man 2 open` donne 5 prototypes :

```
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);
5
6 int openat(int dirfd, const char *pathname, int flags);
7 int openat(int dirfd, const char *pathname, int flags, 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. D'autre part, l'appel système `openat` (service de numéro 257) diffère d'`open` dans la détermination du répertoire de départ lorsqu'un chemin relatif est fourni. On ne s'en soucie pas ici.

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 *`.

Pour rappel, en langage d'assemblage, le premier argument d'un appel système est fourni via `rdi`.

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-linux.h` [17].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.
```

15. 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).

```
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  #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  // ...
35
```

Il faut savoir qu'un littéral entier qui commence par un zéro (0) dans un code en langage C [18] (ou en Java [19]) 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].

Pour rappel, en langage d'assemblage, le deuxième argument d'un appel système est fourni via `rsi`.

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`.

18. https://en.cppreference.com/w/c/language/integer_constant (consulté le 21 février 2020).
19. https://docs.oracle.com/javase/specs/jls/se13/html/jls-3.html#jls-OctalNumeral (consulté le 5 mars 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).

Pour rappel, en langage d'assemblage, le troisième argument d'un appel système est fourni via `rdx`.

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.

Pour rappel, en langage d'assemblage, la valeur retournée par un appel système l'est via `rax`.

5.2.3 Exemple

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

```
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.
23
```

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*.

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

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 caractères [26] sont déclarées :

```
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 caractères [27]. 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
```

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.
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).

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.

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.

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).

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

TABLE 1 – Descripteurs de fichier associés aux flux standards.

7 Hello, World!

Voici `Hello, World!` [33] en langage d'assemblage :

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

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

Service	Numéro (`rax`)	But	Paramètres (`rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`)	Retour (`rax`)	Notes
`exit`	60	quitter un programme	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	1er : chemin vers le fichier : chaîne zéro-terminée ; 2e : options d'ouverture : indicateurs à combiner avec | ; 3e : mode : si création d'un fichier	descripteur de fichier ou entier négatif en cas d'erreur	options d'ouverture : `/usr/include/bits/fcntl-linux.h` (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	1er : descripteur du fichier ; 2e : adresse où stocker le résultat de la lecture ; 3e : nombre de *bytes* à lire	nombre d'octets effectivement lus, −1 en cas d'erreur	— le descripteur de fichier est celui retourné par `open` ou 0 pour lire au clavier ; — la tête de lecture est avancée du nombre de *bytes* lus
`write`	1	écrire dans un fichier	1er : descripteur du fichier ; 2e : adresse de ce qui doit être écrit ; 3e : nombre de *bytes* à écrire	nombre d'octets effectivement écrits, −1 en cas d'erreur	— 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 *bytes* écrits

TABLE 2 – Cinq appels système.

```asm
; 01_open_exercice.asm

global    main

section .rodata
    nomFichier      DB      `brol`, 0   ; ne pas oublier le 0

section .text
main:
    ; ouverture de brol en écriture seule avec placement
    ; de la tête d'écriture en fin de fichier
    mov     rax, 2          ; open
    mov     rdi, nomFichier ; adresse du 1er caractère du nom
    mov     rsi, 1q | 2000q ; WRONLY + O_APPEND
    syscall

    mov     rax, 60
    mov     rdi,0
    syscall
```

1. Inspectez le contenu du registre `rax` juste après l'appel système d'ouverture du fichier de nom `brol` dans les cas suivants :
   a) le fichier `brol` existe avec droit en écriture ;
   b) le fichier `brol` existe sans droit en écriture ;
   c) le fichier `brol` n'existe pas.
2. Mettez en commentaire la ligne :

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

   et insérez les lignes :

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

   juste avant le premier 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`

USASCII code chart

b_4	b_3	b_2	b_1	Row ↓ / Column →	0 (0 0 0)	1 (0 0 1)	2 (0 1 0)	3 (0 1 1)	4 (1 0 0)	5 (1 0 1)	6 (1 1 0)	7 (1 1 1)
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

FIG. 1 – Table `ASCII` (Illustration Wikipedia [a]).

a. https://commons.wikimedia.org/wiki/File:USASCII_code_chart.png (consulté le 21 février 2020).

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 ?

Ex. 4 Écrivez un programme qui affiche à l'écran :
— `pair` si le contenu de `rcx` est pair ;
— `impair` si le contenu de `rcx` est impair.

Ex. 5 Produisez une variante de l'exercice Ex. 4. Il s'agit d'écrire le contenu du registre `rcx`. Cette écriture doit se faire soit dans un fichier nommé `pair`, lorsque le contenu de `rcx` est pair, soit dans un fichier nommé `impair`, lorsque ce contenu est impair. Le contenu de `rcx` ne doit pas être converti en chaîne de caractères, mais écrit tel quel dans le fichier, c'est-à-dire en petit-boutisme binaire complément à deux. Il n'y a donc rien de spécial à faire, au delà du transfert du contenu de `rcx` dans une variable avant écriture dans le fichier, puisqu'il s'agit des conventions standards de représentation des données.

Par exemple :
— si `rcx` contient la valeur décimale 77, il faut écrire cette valeur dans le fichier `impair`. Le contenu d'`impair` est alors la suite ordonnée des 8 *bytes* suivants, fournis en hexadécimal : `4D 00 00 00 00 00 00 00` ;
— si `rcx` contient la valeur $-1\,456\,777\,255\,814\,908$, le contenu de `pair` est : `04 79 26 9D 11 D3 FA FF`.

Aide : Pour vérifiez le contenu des fichiers `pair` et `impair`, utilisez la commande filtre `od`[34]. Par exemple :

```
od -tx1 file
```

affiche au format hexadécimal le contenu du fichier `file` *byte* par *byte*.

Ex. 6 Écrivez un programme qui stocke dans `rcx` la taille en *bytes* du fichier `brol`. 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.

34. http://man7.org/linux/man-pages/man1/od.1.html (consulté le 21 février 2020).
35. http://manpagesfr.free.fr/man/man2/lseek.2.html (consulté le 21 février 2020).

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.