Lyon 1

Systèmes d'exploitation

Cahier de TD - printemps 2019

Table des matières

TD 1
Fichiers - système de fichiers

1.1 Exercices : mode utilisateur

EXERCICE 1 ▶ **Arborescence — Droits sous unix**
Sur le système considéré, il y a 4 utilisateurs :

- `fontaine` qui fait partie du groupe `prof` et `user`;
- `elise` qui fait partie des groupes `etu` et `user`;
- `hippolyte` qui fait partie du groupe `prof`.
- `root` qui est l'administrateur et fait partie du groupe `root`

```
drwxr-x--x 27      root  user   /bin/
-rwsr-xr-x  1      root   etu   /bin/visioneurPDF
drwxr-xr-x 80      root  user   /home/
drwxr-x--x 10  fontaine  prof   /home/fontaine/
drwx--x---  4  fontaine  prof   /home/fontaine/prive/
-rw-r-x---  1  fontaine  prof   /home/fontaine/sujet.pdf
-rw-r--r--  1 hippolyte  prof   /home/fontaine/prive/correction.pdf
-rw-rw----  1  fontaine  prof   /home/fontaine/prive/notes.ods
```

Attention, pour les fichiers vous devez tenir compte des droits des répertoires et sous répertoires.

1) Représenter l'arborescence.

2) Quelle commande peut faire `hippolyte` pour récupérer le fichier `correction.pdf` sur la racine de son compte, en a-t-il le droit?

3) Représenter les possibilités d'accès des 4 utilisateurs aux fichiers `visioneurPDF`, `sujet.pdf`, `correction.pdf` et `notes.ods`, par un tableau à double entrée.

4) Élise peut-elle copier le fichier `correction.pdf`? Peut-elle visualiser ce fichier?

EXERCICE 2 ▶ **Inœuds**

D'après T. Lavergne, Univ Paris Saclay. On considère un système de fichiers tel que l'information concernant les blocs de données de chaque fichier est accessible à partir du inœud de celui-ci (comme dans UNIX).

On suppose dans cet exercice :

- Le système de fichiers utilise des blocs de données de taille fixe 1 kio (1024 octets).
- L'inœud de chaque fichier (ou répertoire) contient 12 pointeurs directs sur des blocs de données, 1 pointeur indirect simple, 1 pointeur indirect double et 1 pointeur indirect triple.
- Chaque pointeur (numéro de bloc) est représenté sur 4 octets.

1) Quelle est la plus grande taille de fichier que ce système de fichiers peut supporter?

2) On considère un fichier contenant 100 000 octets. Combien de blocs de données faut-il (au total) pour représenter ce fichier sur disque?

EXERCICE 3 ▶ **Espace occupé par des fichiers sur le disque**
On travaille sur un système GNU/Linux, sur une partition formatée avec le système de fichiers ext4 (comme son ancètre ext2, il utilise des inœuds pour organiser les fichiers). La commande `fdisk` (utilisée avec précaution) donne les informations suivantes sur le disque dur :

```
Disque /dev/sda: 931,5 GiB, 1000204886016 octets, 1953525168 secteurs
Unités: secteur de 1 * 512 = 512 octets
Taille de secteur (logique / physique)~: 512 octets / 4096 octets
taille d'E/S (minimale / optimale)~: 4096 octets / 4096 octets
```

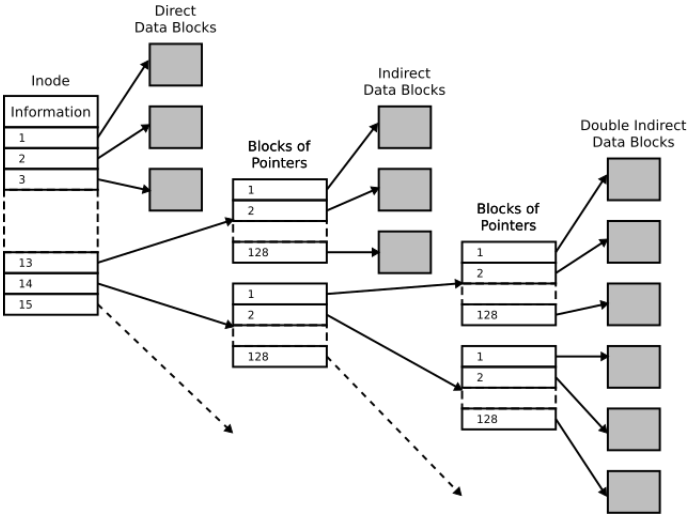


FIGURE 1.1 – ext2 inodes, timtjtim, CC by SA

1ère expérience : Dans un répertoire `test` initialement vide, on lance d'abord la commande suivante :

```
for i in $(seq 1 123); do printf "%03d" $i > tmp$i.fic; done
```

On lance ensuite :

```
nlouvet@dallol:~/test$ ls -lh
total 492K
-rw-r--r-- 1 nlouvet nlouvet 3 févr.  1 10:41 tmp100.fic
...
nlouvet@dallol:~/test$ du -sh .
496K    .
```

1) La commande `for` crée des fichiers dans le répertoire courant : combien, et quelle est leur taille en nombre d'octets ?
2) Quelle est la taille en kibioctets (kio) d'un bloc sur le disque dur ? Etayez votre hypothèse en vous référant au total renvoyé `ls -lh`, et en faisant un petit calcul.
3) La commande `du -sh` donne l'espace occupé sur le disque dur par le répertoire courant et son contenu : pourquoi est-ce cette valeur diffère du total donné par `ls -lh` ?
4) La commande `du --inode` donne le nombre d'inœuds occupés par le répertoire courant est son contenu : dans notre cas, combien va-t-elle afficher ?

2ème expérience : On efface le contenu du répertoire `test` (`rm tmp*.fic`), puis on reprend en créant des fichiers :

```
nlouvet@dallol:~/test$ for i in $(seq 1 250); do printf "%03d" $i > tmp$i.fic; done
nlouvet@dallol:~/test$ du -sh
1012K   .
nlouvet@dallol:~/test$ ls -lh
total 1000K
-rw-r--r-- 1 nlouvet nlouvet 3 févr.  1 14:07 tmp100.fic
...
```

Avec le raisonnement précédent, à quelle valeur pouvait-on s'attendre pour `du -sh` ? D'où vient la différence ?

1.2 Exercice : mode développeur

EXERCICE 4 ▶ **Écritures sur le disque avec write**
On reproduit ci-dessous un extrait de la page de manuel (release 4.15 of the Linux man-pages project, 2018-02-02) de la fonction C `write()`. Vous vous référerez à cet extrait pour la suite de l'exercice.

```
NAME
       write - write to a file descriptor

SYNOPSIS
```

```
       #include <unistd.h>

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

DESCRIPTION
       write()  writes  up  to count bytes from the buffer starting at buf to the file referred to by the
       file descriptor fd.

       The number of bytes written may be less than count if, for example, there is insufficient space on
       the  underlying  physical  medium,  or  the  RLIMIT_FSIZE resource limit is encountered (see setr-
       limit(2)), or the call was interrupted by a signal handler after having written  less  than  count
       bytes.  (See also pipe(7).)

       For  a  seekable  file  (i.e.,  one to which lseek(2) may be applied, for example, a regular file)
       writing takes place at the file offset, and the file offset is incremented by the number of  bytes
       actually  written.   If  the file was open(2)ed with O_APPEND, the file offset is first set to the
       end of the file before writing.  The adjustment of the file offset and  the  write  operation  are
       performed as an atomic step.

       [...]

RETURN VALUE
       On  success,  the number of bytes written is returned (zero indicates nothing was written).  It is
       not an error if this number is smaller than the number of bytes requested;  this  may  happen  for
       example because the disk device was filled.  See also NOTES.

       On error, -1 is returned, and errno is set appropriately.

       If  count is zero and fd refers to a regular file, then write() may return a failure status if one
       of the errors below is detected.  If no errors are detected, or error detection is not  performed,
       0  will  be  returned  without causing any other effect.  If count is zero and fd refers to a file
       other than a regular file, the results are not specified.

NOTES
       The types size_t and ssize_t are, respectively, unsigned and signed integer data  types  specified
       by POSIX.1.

       [...]

       If  a write() is interrupted by a signal handler before any bytes are written, then the call fails
       with the error EINTR; if it is interrupted after at least one byte has been written, the call suc-
       ceeds, and returns the number of bytes written.

       [...]
```

1) Ré-expliquez, avec vos propres mots ce que permet la fonction `write()` ? (détaillez les paramètres et la première phrase de la `DESCRIPTION`).

2) On suppose qu'un programme initialise un tableau de caractères déclaré par `char tab[64]`. Comment appeler `write` pour écrire (au plus) tout le contenu de `tab` sur le descripteur de fichier `fd`? Même question si le tableau a été déclaré par `long tab[64]`. Proposez des solutions avec ou sans `sizeof`.

3) Comment écrire une (potentiellement longue) chaîne de caractères de type `char*` sur le disque, via un descripteur de fichier `fd` **caractère par caractère**? Donnez une fonction `writech` pour cela (prototype et définition). Donnez un exemple d'appel de votre fonction. Combien de caractères (au plus) pourrez vous écrire via un appel à votre fonction (en faisant une hypothèse raisonnable) ?

TD 2
Fichiers (fin) et Mémoire

2.1 Fichiers

EXERCICE 1 ▶ **Lectures sur le disque avec read**
On reproduit ci-dessous un extrait de la page de manuel de la fonction C `read()`. Vous vous référerez à cet extrait pour la suite de l'exercice.

```
NAME
       read - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read()  attempts to read up to count bytes from file descriptor fd into
       the buffer starting at buf.

       On files that support seeking, the read operation commences at the file
       offset, and the file offset is incremented by the number of bytes read.
       If the file offset is at or past the end of file, no  bytes  are  read,
       and read() returns zero.

       If count is zero, read() may detect the errors described below.  In the
       absence of any errors, or if read() does not check for errors, a read()
       with a count of 0 returns zero and has no other effects.
[...]

RETURN VALUE
       On success, the number of bytes read is returned (zero indicates end of
       file),  and the file position is advanced by this number.  It is not an
       error if this number is smaller than the  number  of  bytes  requested;
       this  may happen for example because [...]

       On error, -1 is returned, and errno  is  set  appropriately.   In  this
       case,  it  is  left  unspecified  whether  the  file  position (if any)
       changes.
[...]
NOTES
       The types size_t and ssize_t are, respectively, unsigned and signed integer data types specified by POSIX.1.
```

1) Ré-expliquez, avec vos propres mots ce que permet la fonction `read()` ? (détaillez les paramètres et la première phrase de la `DESCRIPTION`).

2) Comment utiliser cette fonction pour lire 70 chars sur l'entrée standard en une unique fois, puis caractère par caractère ?

2.2 Exercices en mode utilisateur

Nous n'avons pas encore vu les processus, mais en première approximation, on peut dire qu'un processus est un programme en cours d'exécution

EXERCICE 2 ▶ **Mémoire paginée, version simple**
Exercice proposé par F. Rico. Dans cet exercice on considère 2 processus, qui s'exécutent sur un système à mémoire partagée décrite dans la Figure 2.1 : la table des répertoires de pages du processus 1 est à l'adresse 0o17 et celle du processus 2 à l'adresse 0o16.

De façon similaire à la mémoire des processeurs, la mémoire paginée décrite possède des caractéristiques suivantes :

— Chaque adresse est codée sur 6 bits (2 chiffres entre 0 et 7).

— Une adresse de 6 bits correspond à 4 octets (4 caractères).

— Chaque page contient 8 mots mémoires (1 ligne du tableau 2.1).

— Il y a une indirection : pour calculer l'adresse réelle à partir d'une adresse "virtuelle" données, on regarde la table des pages **du processus en cours** [1]).

— Une adresse logique est composée de 6 bits, de gauche à droite : 3 pour la page et 3 pour le décalage dans la page.

— Dans les tables de pages, pour chaque page, 3 bits sont utilisés pour détailler les propriétés de la page, dans l'ordre :

1. pour signaler l'existence de la page ;
2. pour signaler le droit d'écriture ;
3. pour signaler le droit d'exécution.

Dans le tableau de la Figure 2.1, les indices de lignes (`00`, ..., `17`) sont les bits de poids forts des adresses physiques.

Les indices des colonnes (`00`, ..., `07`) sont les (3) bits de poids faibles des adresses : ils donnent les décalages par rapport au début de page. Par exemple, si vous regardez ce qui se trouve dans la page physique `0o10`, au décalage `0o02`, on trouve le mot `supo` (de 4 octets).

1) Donnez les 16 octets de la chaîne de caractères stockée à l'adresse `0b011110` du processus 1. *On commencera par calculer la valeur de l'adresse en octal.*

2) Donnez les quatre premiers octets de la chaîne de caractères stockée à l'adresse `0b110010` du processus 1, puis les quatre premiers octets de la chaîne de caractères stockée à l'adresse `0b000010` du processus 2. Que remarquez-vous ?

	00	01	02	03	04	05	06	07
00	cta_	est_	B7K	mVf	nWs4	Nfi2	RRUU	7tE0
01	qSFc	dB90	uUou	cUcx	LkPT	8elm	Q7hU	mGRw
02	6csB	XnzW	WuzX	FoH4	gg2	j6Tp	eirC	6mWK
03	sbj2	6y75	7xng	CH81	4P0b	VISQ	ale_	a_ja
04	4Dyr	FVlL	hC6	FRxx	dYrM	K1Yv	uJqg	eg7H
05	MIt	PTs	EJCg	O7OW	zBcs	ct I	usu	A5Ag
06	dzPN	LmbP	rs0l	wnLg	jze0	M9tO	Z85	x p
07	qUK5	Dvs4	Ed5s	goF7	20Jk	9ZeN	tsE	tYxa
10	eOs7	Oki	supo	csan	ivid	S5gD	HkwC	l Yc
11	0Nqu	MxnY	AfAb	JCzq	RTkf	I P4	TzN4	CPp2
12	Md i	Lrba	NFCV	muth	JG5X	xO8g	qaJu	jT39
13	Nv0b	XX3q	n mF	eDxs	yHQv	nbib	tceT	FSW0
14	mQQf	N3mf	qOOj	XSX9	Cw B	n2cq	yw9	2PcV
15	tKgu	8GAT	N217	tz 6	wYxW	Hw44	WUWt	G8Hn
16	101.10	001.17	000.00	110.02	101.15	000.12	111.13	001.30
17	100.00	001.23	000.00	110.03	110.00	010.10	101.10	010.21

FIGURE 2.1 – Mémoire simplifiée

EXERCICE 3 ► **Mémoire paginée « à la Pentium »**

Exercice proposé par F. Rico

De façon similaire à la mémoire des processeurs INTEL PENTIUM, la table 2.2 représente une mémoire paginée :

— Chaque adresse est codée sur 9 bits.

— Une adresse de 9 bits correspond à 1 mot mémoire de 32 bits.

— Chaque page contient 8 mots mémoires (1 ligne du tableau 2.2).

— Il y a 2 niveaux d'indirection (table de répertoires de pages et table de pages).

— Une adresse logique est composée de 9 bits, de gauche à droite : 3 pour le répertoire de pages, 3 pour la page et 3 pour le décalage dans la page.

— Dans les tables de pages, pour chaque page, 7 bits sont utilisés pour détailler les propriétés de la page :

1. pour signaler l'existence de la page
2. pour signaler la présence de la page en mémoire
3. pour signaler le droit d'écriture
4. pour signaler le droit d'exécution

1. Donc pour les deux processus, une même donnée peut être à des adresses différentes.

5. pour signaler le « copy-on-write »
6. pour le bit d'accès
7. pour le dirty bit

— Dans les tables de répertoire de pages, seul le premier bit d'information est utilisé (celui de l'existence du répertoire correspondant).

On considère 2 processus, la table des répertoires de pages du processus 1 est à l'adresse `0o01` et celle du processus 2 à l'adresse `0o17`.

Dans cet exercice et dans le cadre du cours, on ignore les informations apportées par le bit "copy on write" et par le "dirty bit".

1) Quel est la capacité d'adressage ?

2) Pour chacun des deux processus, que contient la case d'adresse `0b101111001` ?

3) Que se passe-t-il si le processus 1 essaye de lire la valeur de la case `0b101101101` ?

4) Que remarque-t-on pour l'adresse 0b101010000 du processus 1 et l'adresse 0b000101000 du processus 2 ?

adresse de la page	Décalage dans la page							
mem	0	1	2	3	4	5	6	7
00	70	48	22	142	32	126	215	128
01	1100000.11	0000000.13	0000000.07	0000000.02	0000000.21	1100000.33	0000000.23	0000000.11
02	107	63	71	241	171	91	93	223
03	29	167	27	139	154	87	59	79
04	33	54	197	180	13	112	71	168
05	130	15	233	70	8	206	76	139
06	178	227	4	159	116	23	58	200
07	204	28	123	120	61	173	51	11
10	185	63	13	130	24	18	117	87
11	1101010.05	1101000.06	1100110.25	1110110.37	0010100.57	0010010.77	0001100.66	0010000.74
12	157	115	237	19	159	219	30	187
13	0000000.16	0010100.37	0001111.67	0000000.37	0000000.03	0000000.34	0000000.34	1110000.20
14	197	248	206	145	47	28	197	249
15	129	126	122	137	205	70	220	127
16	1101010.05	1101000.06	1100110.25	1110110.37	0011000.46	1110000.14	0001010.06	1110000.27
17	1100000.16	0000000.07	0000000.25	0000000.20	0000000.25	1100000.13	0000000.07	0000000.06
20	122	17	153	217	242	151	150	209
21	205	201	173	237	183	208	31	78
22	147	148	236	235	243	118	195	249
23	145	42	235	41	148	181	30	83
24	193	36	2	240	167	207	147	192
25	69	69	216	223	163	41	58	141
26	41	93	176	16	99	10	178	207
27	223	135	55	26	205	211	248	24
30	193	26	8	3	42	191	181	232
31	165	162	122	104	67	191	78	123
32	219	55	208	116	148	253	55	143
33	0001010.57	0010100.32	1110010.14	0011000.31	0000000.63	0001111.31	0001010.06	1110011.23
34	191	142	215	171	241	121	112	180
35	1	94	177	206	225	148	86	2
36	111	5	242	120	32	44	143	201
37	0	0	0	0	0	0	0	0

FIGURE 2.2 – Exemple de mémoire

TD 3

Processus

3.1 Processus (utilisateur)

EXERCICE 1 ▶ **Petites manipulations/démos**

Exercice fortement inspiré d'un TD de J. Seinturier, univ. Aix-Marseille.

1) Donner une commande qui permet de lister tous les processus du système.

Le résultat de la commande précédente donne :

```
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.2  0.0 168288  6624 ?        Ss   08:36   0:02 /sbin/init splash
root          2  0.0  0.0      0     0 ?        S    08:36   0:00 [kthreadd]
root          3  0.0  0.0      0     0 ?        I<   08:36   0:00 [rcu_gp]
root          4  0.0  0.0      0     0 ?        I<   08:36   0:00 [rcu_par_gp]
root          6  0.0  0.0      0     0 ?        I<   08:36   0:00 [kworker/0:0H-events_highpri]
[...]
root       1832  0.0  0.0 255300  5740 ?        Ssl  08:37   0:00 /usr/lib/upower/upowerd
root       2524  0.0  0.0 175200  5680 ?        Sl   08:37   0:00 gdm-session-worker [pam/gdm-passwo
laure      2573  0.1  0.0  19984  5196 ?        Ss   08:37   0:01 /lib/systemd/systemd --user
laure      2580  0.0  0.0 175596  4456 ?        S    08:37   0:00 (sd-pam)
laure      2717  0.0  0.0 2747956 8700 ?        S<sl 08:37   0:00 /usr/bin/pulseaudio --daemonize=no
laure      2768  0.0  0.0 538160  5112 ?        SLl  08:37   0:00 /usr/bin/gnome-keyring-daemon --da
laure      2874  0.0  0.0   9028  4528 ?        Ss   08:37   0:01 /usr/bin/dbus-daemon --session --a
root       2876  0.0  0.0      0     0 ?        S<   08:37   0:00 [krfcommd]
laure      2878  0.0  0.0 166596  3540 tty2     Ssl+ 08:37   0:00 /usr/lib/gdm3/gdm-x-session --regi
laure      2880  2.0  0.6 458688 98520 tty2     Sl+  08:37   0:26 /usr/lib/xorg/Xorg vt2 -displayfd
laure      2892  0.0  0.0   4848  1172 tty2     S+   08:37   0:00 /bin/sh /usr/lib/gnome-session/run
laure      3000  0.0  0.0   8072   484 ?        Ss   08:37   0:00 /usr/bin/ssh-agent /usr/bin/im-lau
laure      3164  0.0  0.0  16352   888 tty2     S+   08:37   0:00 systemctl --user start --wait unit
[...]
laure      3507  6.7  1.9 3368744 315620 ?      Sl   08:37   1:27 /usr/lib/firefox/firefox
laure      3510  0.0  0.0 275752  9340 ?        Sl   08:37   0:00 /usr/lib/policykit-1-gnome/polkit-
laure      3512  0.0  0.0 502304 11820 ?        Sl   08:37   0:00 a11y-profile-manager-indicator
laure      3513  0.3  0.1 636048 28292 ?        Sl   08:37   0:04 /usr/bin/python /usr/bin/solaar
laure      3523  0.0  0.1 582488 19352 ?        Sl   08:37   0:00 /usr/lib/x86_64-linux-gnu/libexec/
[...]
laure      6992  1.0  0.3 749340 61880 pts/1    Sl   08:51   0:07 emacs td3.tex
```

2) À quelle heure le système a-t-il été démarré?

3) À quelle heure l'utilisatrice `laure` a-t-elle commencé à utiliser l'ordinateur?

4) Que fait l'utilisatrice au moment de la commande?
Un peu de lecture de documentation fournit : [1]

> **VSZ (Virtual Set Size)** *The Virtual Set Size is a memory size assigned to a process (program) during the initial execution. The Virtual Set Size memory is simply a number of how much memory a process has available for its execution.*
> **RSS (Resident Set Size)** *As oppose to VSZ (Virtual Set Size), RSS is a memory currently used by a process. This is a actual number in kilobytes of how much RAM the current process is using.*

5) Quelle est la quantité de RAM utilisée par solaar au moment de la commande?
Maintenant on désire dessiner l'arborescence de quelques processus, on utilise donc la commande `ps -ef` et cela donne le résultat tronqué suivant :

```
UID         PID  PPID  C STIME TTY          TIME CMD
root          1     0  0 08:36 ?        00:00:03 /sbin/init splash
root          2     0  0 08:36 ?        00:00:00 [kthreadd]
root       1538     1  0 08:36 ?        00:00:00 /usr/sbin/gdm3
root       2524  1538  0 08:37 ?        00:00:00 gdm-session-worker [pam/gdm-password]
laure      2573     1  0 08:37 ?        00:00:01 /lib/systemd/systemd --user
laure      3193  2573  0 08:37 ?        00:00:00 /usr/lib/gnome-session/gnome-session-binary --syste
laure      3507  3193  3 08:37 ?        00:02:25 /usr/lib/firefox/firefox
laure      3513  3193  0 08:37 ?        00:00:07 /usr/bin/python /usr/bin/solaar
laure      4887  2573  0 08:38 ?        00:00:07 /usr/libexec/gnome-terminal-server
laure      5112  4887  0 08:39 pts/1    00:00:00 bash
laure      7481  5112  0 09:02 pts/1    00:00:03 evince cahier_TD_LIFASR5_2020-solution.pdf
laure      6992  5112  0 08:51 pts/1    00:00:12 emacs td3.tex
```

6) Dessiner le début de l'arborescence des processus.

7) Que se passe-t-il si l'utilisatrice `laure` tape la commande `kill -9 2524`?

8) Que se passe-t-il si l'utilisateur `root` tape la commande `kill -9 7481`?

1. `https://linuxconfig.org/ps-output-difference-between-vsz-vs-rss-memory-usage`)

3.2 Programmation de processus (programmation système)

Pour cette section on donne quelques éléments de manuel :

```
NAME   fork - create a child process
SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>
       pid_t fork(void);
DESCRIPTION fork() creates a new process by duplicating the calling process. The new process is referred to as the
       child process. The calling process is referred to as the parent process. The child process and the parent
       process run in separate memory spaces. At the time of fork() both memory spaces have the same content.
RETURN VALUE
       On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On
       failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.
------------------------------------------------------------------------------------------------------------------------
NAME   waitpid - wait for process to change state
SYNOPSIS
       #include <sys/types.h>
       #include <sys/wait.h>
       pid_t waitpid(pid_t pid, int *wstatus, int options);
DESCRIPTION The waitpid() system call suspends execution of the calling process until a child specified by pid
       argument has changed state. A state change is considered to be: the child terminated; the child was stopped
       by a signal; or the child was resumed by a signal.

       By default, waitpid() waits only for terminated children, but this behavior is modifiable via the options
       argument.

       The value of pid can be:
          -1   meaning wait for any child process.
           0   meaning  wait  for  any  child  process  whose  process  group ID is equal to that of the calling
               process.
         > 0   meaning wait for the child whose process ID is equal to the value of pid.

       The value of options is an OR of zero or more of the following constants:
         WNOHANG     return immediately if no child has exited.
         [...]

       If  wstatus  is  not  NULL, waitpid() stores status information in the int to which it points.
RETURN VALUE
       waitpid(): on success, returns the process ID of the child whose  state has changed. On error, -1 is
       returned.
------------------------------------------------------------------------------------------------------------------------
(wait)
    The wait() system call suspends execution of the calling  thread  until one  of its children terminates. The
    call wait(&wstatus) is equivalent to:

            waitpid(-1, &wstatus, 0);
```

EXERCICE 2 ▶ **Deux frères**

Dans cet exercice on utilisera `fork` et `wait`.

1. Écrire un programme C++ qui fait créer un fils au "processus `main`", ce fils affichant les entiers de 1 à 50 avant de retourner.
2. Modifier ce programme pour créer un autre fils au "processus `main`", affichant les entiers de 51 à 100. Modifier votre programme pour qu'il soit certain que l'affichage des nombres se produise dans l'ordre.

EXERCICE 3 ▶ **Croissez et multipliez - (facultatif)**

D'après un examen de Système M2CCI, 2015 et J. Solanski pour l'ENSEIRB.

Combien de processus sont créés par le code suivant ? Expliquer et commenter. (On donnera des exemples d'exécutions possibles avec des pid cohérents). *Commentez aussi l'usage du nom de variable "pid".*

```
  int pid, i;

  for (i=0; i<3;i++) {
    pid = fork();
    if (pid < 0 ){
      /* ici code echec*/
    } else if (pid==0) {
      printf("(i : %d) je suis le processus : %d, mon pere est : %d\n", i, getpid(), getppid()); }
    else {
      printf("(i : %d) je suis le processus : %d, mon pere est : %d\n", i,getpid(), getppid()); }
  }//fin du "for"
  return 0;
}
```

TD 4

Processus et communication

4.1 Communication par tuyaux (pipe)

Les *pipes* (en mancunien ; en français, on parle de tubes ou de tuyaux) sont des canaux de communication *unidirectionnels* manipulés à l'aide de *descripteurs de fichiers*. Tout processus ayant accès à ces descripteurs peut lire ou écrire dans un tube : seuls les processus descendant du créateur d'un tube, et le créateur lui-même pourront l'utiliser.

Un tube se comporte comme une file FIFO (First In, First Out) : les premières données entrées dans le tube seront les premières à être lues. Toute lecture est destructive : si un processus lit une donnée dans le tube, celle-ci n'y sera plus présente, même pour les autres processus.

Il y a quatre primitives importantes pour les tubes :

— `pipe(pipefd)` : crée un tube et retourne 0 en cas de succès, -1 en cas d'échec. Après appel, `pipefd[0]` est le descripteur du tube créé en lecture, `pipefd[1]` est le descripteur du tube créé en écriture.

— `write(pipefd[1], buf, count)` permet d'écrire au plus `count` octets depuis l'adresse `buf` dans le pipe dont le descripteur pour l'écriture est `pipefd[1]`. L'appel retourne le nombre d'octets écrits, ou -1 en cas d'échec.

— `read(pipefd[0], buf, count)` lit au plus `count` octets dans le tube ayant `pipefd[0]` pour descripteur en lecture et place ces octets en mémoire à partir de l'adresse `buf`. L'appel renvoie le nombre d'octets lus, 0 si le tube est vide et qu'il n'y a plus d'écrivain (un processus qui a gardé le descripteur pour l'écriture ouvert) sur le pipe, et -1 en cas d'échec. *Tant qu'il reste un écrivain sur le pipe, même si le pipe est vide, l'appel à `read` sur le pipe est bloquant.*

— `close(pipefd)` : comme d'habitude, permet de fermer un descripteur de fichiers.

EXERCICE 1 ▶ **Échange d'entiers (`int`) via des tuyaux**

Dans cet exercice, on suppose l'existence de trois fonctions :

- `int write_int(int fd, int x);` permet d'écrire exactement un entier `x` de type `int` sur le descripteur de fichier `fd`. La fonction retourne 1 en cas de succès, -1 en cas d'échec.
- `int read_int(int fd, int &x);` permet de lire exactement un entier `x` de type `int` sur le descripteur de fichier `fd`. La fonction retourne 1 en cas de succès, 0 si le tube est vide et qu'il n'y a plus d'écrivain dessus, et -1 en cas d'échec.
- `void die(void);` qui affiche un message d'erreur conforme à ce qu'indique la variable `errno`, et termine le processus depuis lequel elle est appelée.

Le but est d'écrire deux programmes : dans l'un le père envoie des données à son fils ; dans l'autre, un fils envoie des données à un autre fils. Dans les deux cas, veillez à *fermer les descripteurs de fichiers qui ne sont pas utilisés par le processus courant dès que possible.* **Ce programme est très similaire à celui du cours, pour que l'apprentissage soit efficace, il est recommandé de ne pas avoir celui-ci sous les yeux**.

1) Écrire (en utilisant la primitive système `fork`) un programme dans lequel :
- le processus père : crée un pipe, lance un processus fils, écrit un-à-un les entiers (`int`) de 1 à 100 inclus dans le pipe, puis attend la fin de son fils (primitive `waitpid`).
- le processus fils : lit un-à-un les entiers reçus dans le pipe (sans les compter…), affiche seulement ceux qui sont multiples de 3, puis se termine proprement.

2) Imaginez que vous n'ayez pas respecté la consigne « fermer les descripteurs de fichiers qui ne sont pas utilisés par le processus courant dès que possible » dans le programme de la question précédente : que se passe-t-il si le fils « oublie » de fermer son descripteur en écriture du pipe avant de commencer à lire les données dessus ?

EXERCICE 2 ▶ **(supplément) 1 père 2 fils**

On vous fournit le programme suivant :

```
int main(void) {
  int pid1, pid2;

  if((pid1 = fork()) == -1) die();

  if(pid1 > 0) { // processus père
```

```
    if((pid2 = fork()) == -1) die();

    if(pid2 > 0) { // // processus .(a)...

      if( waitpid(pid1, NULL, 0) == -1) die();
      if( waitpid(pid2, NULL, 0) == -1) die();

    }
    else { // processus .(b)...
      sleep(1);
    }

  }
  else { // processus ..(c)...
    sleep(1);
  }

  return 0; // exécuté par ?
}
```

1) Que fait ce programme?

2) Modifier le programme pour que :

- le processus `fils1` : écrit les entiers (`int`) de 0 à 100 dans le pipe puis se termine proprement.
- le processus `fils2` : lit un-à-un les entiers dans le pipe, affiche seulement ceux qui sont multiples de 7, se termine proprement.

3) Jouer avec les fermetures et non fermetures de descripteurs.

4.2 Communication par sockets

Le but de cet exercice est d'étudier la mise en place d'un serveur réseau (TCP/IP) Un tel serveur met en place une socket d'écoute sur une certaine adresse IP de la machine hôte, et sur un certain port. Ensuite, lorsqu'un client demande à se connecter (sur cette adresse et ce port), le serveur crée une socket de dialogue pour échanger avec le client. Voici un exemple du comportement attendu pour le serveur :

```
./server 4000
create_server_socket: tentative de création d'une socket serveur sur le port 4000
create_server_socket: la socket a été créée
create_server_socket: la socket 3 est maintenant en attente à l'adresse 0.0.0.0 sur le port 4000
accept_connection: connexion depuis localhost, 41980
(PERE) une connexion vient d'être acceptée...
(PERE) je viens de créer un fils de PID 24116
(FILS de PID 24116) je prends en charge un client.
[...]
```

Ce comportement a été obtenu en lançant la commande suivante dans un autre terminal :

```
telnet localhost 4000
```

Fonctions utilitaires données Ici, on ne se préoccupe pas de la phase de dialogue, et on va supposer que cela est géré par une fonction, dans laquelle tout est pris en compte :

```
void dial(int s); // s est une socket de dialogue
```

On fait l'hypothèse que la fonction `dial` ne crée pas de nouveau fils (cela aura de l'importance, mais plus tard seulement). On suppose que vous disposez des fonctions suivantes pour faciliter la mise en place des sockets :

```
// Tente de créer une socket côté serveur (localhost), à l'écoute sur toutes les
// interfaces, sur le port passé en paramètre. Affiche des messages sur stderr.
// Retourne la socket d'écoute créée en cas de succès, -1 en cas d'échec.
int create_server_socket(const char* port);

// Côté serveur, se met en attente bloquante d'une connexion sur la socket
// d'écoute s. Affiche des messages sur stderr.
// Retourne la socket de dialogue créée en cas de succès, -1 en cas d'échec.
int accept_connection(int s);
```

Le dialogue est donc géré avec `dial()`, une fois la demande de connexion acceptée avec l'appel `accept_connection()` [1].

Rappelons qu'*il est toujours judicieux de fermer un descripteur de fichier devenu inutile dès que possible, afin d'éviter des situations de blocage.*

EXERCICE 3 ▶ Serveur Simple

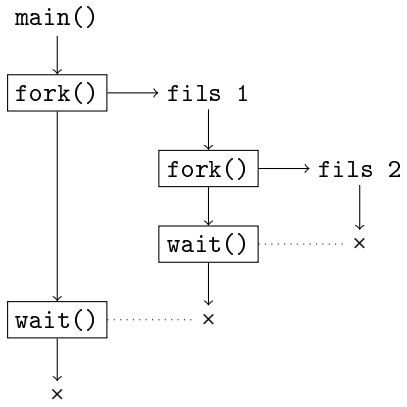
1) Écrivez tout d'abord un serveur « simple » qui se contente de traiter les clients qui arrivent les uns à la suite des autres.

2) Maintenant, votre serveur doit créer un fils à l'arrivée de chaque nouveau client, et chaque fils se chargera du dialogue avec un client. Pour l'instant, ne vous préoccupez pas de la mort des fils.

3) Que se passe-t-il à la mort des fils ? On verra en TP une solution à base de signaux.

EXERCICE 4 ▶ Un mode de représentation de l'exécution des processus

1) On considère le diagramme suivant qui représente l'exécution de 3 processus : le processus principal `main()` et deux procesus fils, `fils1` et `fils2`.



En ce qui concerne l'appel à `wait()` : il a été représenté sur le diagramme à l'instant où le processus fils attendu se termine, et donc où le processus parent sort de cet appel. Il vous est demandé d'écrire la fonction `main()` qui réalise à l'exécution le comportement décrit sur ce diagramme.

2) En vous inspirant du diagramme de la 1ère question, donnez une représentation de l'exécution du programme suivant.

```
int main(void) {
  if(fork() > 0) return 0;
  if(fork() > 0) return 0;
  wait(NULL);
  wait(NULL);
  return 0;
}
```

Que devient votre diagramme si l'on oublie l'un ou l'autre des `wait(NULL)` ?

3) Même question avec le programme suivant. Vous noterez sur votre diagramme, pour chaque processus créé, les valeurs prisent par la variable `i`, ce qui vous permettra de dérouler plus facilement l'exécution du programme.

```
int main(void) {
  for(int i = 0; i < 3; i++) {
    if(fork() == 0) {
      wait(NULL);
      break;
    }
  }
  return 0;
}
```

EXERCICE 5 ▶ Synchronisation avec des tubes

1. Point orthographe : en français connexion ; en anglais *connection*

1ère partie. On considère le morceau de programme suivant :

```
#define NB_CHLD 4

int main(void) {
  int id, i;

  for(id = NB_CHLD; id > 0; id--) {
    if(fork() == 0) break;
  }

  cout << "je suis le processus dont l'id est " << id << endl;

  // MODIFIER ICI

  return 0;
}
```

On va supposer pour simplifier que tous les appels systèmes effectués réussissent, et qu'il n'y a jamais d'erreur (en pratique, il faudrait être plus vigilant...)

1) Combien de processus fils sont créés par le processus principal (celui qui exécute `main()`) dans ce programme ?

2) Après l'exécution de la boucle `for`, chaque processus a une variable `id` avec une valeur distincte : expliquez pourquoi. Quel est l'`id` attribué au processus principal ? Par la suite, on dira que `id` est l'identifiant du processus.

3) Représentez l'arborescence des processus créés (avec `NB_CHLD` égal à 4), en notant pour chaque processus la valeur de `id`. Donnez une trace possible (affichage sur la sortie standard) de l'exécution du programme).

4) On veut que le père attende chacun de ces fils avec `wait()` (ou `waitpid()`) : modifier le programme en conséquence, au niveau du commentaire `// MODIFIER ICI`.

2ème partie. On souhaiterait forcer l'ordre dans lequel les processus exécutent l'affichage de leur `id`, de façon à ce qu'ils l'affichent par ordre croissant :

```
je suis le processus dont l'id est 0
je suis le processus dont l'id est 1
je suis le processus dont l'id est 2
je suis le processus dont l'id est 3
je suis le processus dont l'id est 4
```

L'idée générale est que le processus d'identifiant `id` doit :

— attendre que le processus d'identifiant `id-1` ait affiché son identifiant,

— afficher son identifiant,

— prévenir le processus d'identifiant `id+1` qu'il peut afficher son identifiant.

Il faut bien entendu faire attention à ce qui se passe « aux bords », pour les processus d'identifiants `0` et `NB_CHLD`. Notez que l'on ne veut pas que le processus d'identifiant `id` s'éxécute entièrement avant le processus d'identifiant `id+1` : il s'agit de synchroniser les processus au cours de leur exécution.

On vous demande de réfléchir à une solution à se problème en utilisant des tubes ; en effet un processus reste bloqué en attente de lecture sur un tube tant que le tube est vide et qu'il reste au moins un processus possédant un descripteur de fichier ouvert en écriture sur le tube.

1) Représentez graphiquement le problème, en dessinant un graphe dans lequel : chaque sommet est un processus identifié par son `id` ; il existe un arc du processus i au processus j si i doit s'exécuter avant j ; le nombre arcs est minimal (un tel graphe s'appelle un graphe de dépendances de tâches). Commencez avec `NB_CHLD` égal à 4, puis généralisez.

2) Expliquez comment faire à l'aide d'un tube pour que, si dans le graphe précédent il existe un arc de i à j alors i effectuent son affichage avant j ? Réfléchissez aussi au cas un peu particulier du processus d'identifiant `NB_CHLD`.

3) Pour synchroniser tous les processus, de combien de tubes avez-vous besoin ?

4) Exprimez votre solution en langage algorithmique. N'oubliez pas de prévoir la fermeture des descripteurs de fichiers non-utilisés !