

Séances 1 et 2 : La création de processus lourds

Exercice 1

1. On affiche l'arbre des processus à l'aide de la commande `ps tree`, pour afficher les PID des différents processus, on ajoute l'option `-p`. Le processus racine du système est nommé `systemd` et a pour PID 1.

```
s4my@LAPTOP-C540D56J: ~
$ ps tree -p
systemd(1)
├──agetty(224)
├──agetty(230)
├──cron(172)
├──dbus-daemon(180)
├──init-systemd(Ub(2))
│   ├──SessionLeader(1572)──Relay(1576)(1574)──zsh(1576)
│   ├──SessionLeader(1700)──Relay(1705)(1701)──zsh(1705)
│   └──SessionLeader(2788)──Relay(2790)(2789)──cpptools-srv(9720)
│       ├──cpptools-srv(9721)
│       ├──cpptools-srv(9722)
│       ├──cpptools-srv(9723)
│       ├──cpptools-srv(9724)
│       ├──cpptools-srv(9725)
│       ├──cpptools-srv(9726)
│       ├──cpptools-srv(9727)
│       ├──cpptools-srv(9728)
│       ├──cpptools-srv(9729)
│       ├──cpptools-srv(9730)
│       ├──cpptools-srv(9731)
│       ├──cpptools-srv(9732)
│       ├──cpptools-srv(9733)
│       ├──cpptools-srv(9734)
│       └──cpptools-srv(9735)
└──sh(2790)──sh(2791)──sh(2796)──node(2847)──node(2868)→
```

2. On trouve le `pid_shell` à l'aide de la commande `echo $$`
3. On trouve bien une feuille avec pour PID `pid_shell`
4. On affiche le sous-arbre enraciné en `pid_shell` à l'aide de la commande `ps tree ${pid_shell}`.
5. On compile le programme à l'aide de la commande `gcc -Wall -o concurrence concurrence.c`
6. En exécutant le fichier on remarque bien qu'un nouveau processus est apparu dans le `ps tree`, on remarque également qu'il est le fils du processus du terminal dans lequel nous l'avons lancé et il disparaît bien de l'arborescence lorsque l'exécution est finie.
7. Lorsque l'exécution est terminée, le processus lié à l'exécution de Shell 1 n'a plus de fils.
8. On lance deux fois l'exécutable à l'aide de la commande `./concurrence & ./concurrence`. On constate que deux processus sont créés dans l'arborescence, ils ont tous deux Shell 1 comme père.
9. Même chose, mais cette fois Shell 1 a trois fils identiques.

```
s4my@LAPTOP-C540D56J: ~
$ ps tree -c 1705
zsh
├──concurrence
├──concurrence
└──concurrence
```

10. On retrouve bien à l'aide de la commande `ps j` les deux processus liés à `concurrence` et d'après la colonne STAT, ils sont tous les deux endormis (S) (on constate par ailleurs que `ps` est en R)

```

s4my@LAPTOP-C540D56J: ~
$ ps j
[11:28:41]

```

PPID	PID	PGID	SID	TTY	TPGID	STAT	UID	TIME	COMMAND
	363	394	363	pts/1	394	S+	1000	0:00	-zsh
	1574	1576	1576	pts/0	1576	Ss+	1000	0:08	-zsh
	1701	1705	1705	pts/2	20774	Ss	1000	0:05	-zsh
	2789	2790	2790	pts/3	2790	Ss+	1000	0:00	sh -c "\$VSCODE_WSL_EXT_LOCATION/scripts/wslServer.sh" 31c37ee8f6349
	2790	2791	2790	pts/3	2790	S+	1000	0:00	sh /mnt/c/Users/Ychaa/.vscode-server/bin/31c37ee8f63491495ac49e43b8544550fbae4
	2791	2796	2790	pts/3	2790	S+	1000	0:00	sh /home/s4my/.vscode-server/bin/31c37ee8f63491495ac49e43b8544550fbae4
	2796	2847	2790	pts/3	2790	Sl+	1000	0:21	/home/s4my/.vscode-server/bin/31c37ee8f63491495ac49e43b8544550fbae4
	2859	2860	2860	pts/4	2860	Ssl+	1000	0:01	/home/s4my/.vscode-server/bin/31c37ee8f63491495ac49e43b8544550fbae4
	2847	2868	2790	pts/3	2790	Sl+	1000	0:00	/home/s4my/.vscode-server/bin/31c37ee8f63491495ac49e43b8544550fbae4
	2876	2877	2877	pts/5	2877	Ssl+	1000	0:03	/home/s4my/.vscode-server/bin/31c37ee8f63491495ac49e43b8544550fbae4
	2847	2943	2790	pts/3	2790	Sl+	1000	1:48	/home/s4my/.vscode-server/bin/31c37ee8f63491495ac49e43b8544550fbae4
	2993	2994	2994	pts/7	2994	Ss+	1000	0:00	/bin/sh -c cd '/home/s4my/ProgSys/TP_1-2' && /bin/sh
	2994	2995	2994	pts/7	2994	S+	1000	0:00	/bin/sh
	2995	3005	2994	pts/7	2994	Sl+	1000	0:00	/home/s4my/.vscode-server/bin/31c37ee8f63491495ac49e43b8544550fbae4
	2943	3016	2790	pts/3	2790	Sl+	1000	0:00	/home/s4my/.vscode-server/bin/31c37ee8f63491495ac49e43b8544550fbae4
	2943	3076	2790	pts/3	2790	Sl+	1000	0:02	/home/s4my/.vscode-server/bin/31c37ee8f63491495ac49e43b8544550fbae4
	2847	3089	2790	pts/3	2790	Sl+	1000	0:03	/home/s4my/.vscode-server/bin/31c37ee8f63491495ac49e43b8544550fbae4
	3089	3100	3100	pts/6	3100	Ss+	1000	0:01	/usr/bin/zsh -i
	2943	8543	2790	pts/3	2790	Sl+	1000	0:04	/home/s4my/.vscode-server/extensions/ms-vscode.cpptools-1.18.5.linu
	2789	9720	2790	pts/3	2790	Sl+	1000	0:00	/home/s4my/.vscode-server/extensions/ms-vscode.cpptools-1.18.5.linu
	1705	20773	20773	pts/2	20774	SN	1000	0:00	./concurrency
	1705	20774	20774	pts/2	20774	S+	1000	0:00	./concurrency
	20927	20931	20931	pts/8	21159	Ss	1000	0:00	-zsh
	20931	21157	20931	pts/8	21159	S	1000	0:00	-zsh
	20931	21158	20931	pts/8	21159	S	1000	0:00	-zsh
	20931	21159	20931	pts/8	21159	R+	1000	0:00	ps j

11. On constate en effet que l'ordonnancement mis en oeuvre par le noyau système n'est pas prévisible de notre point de vue

```

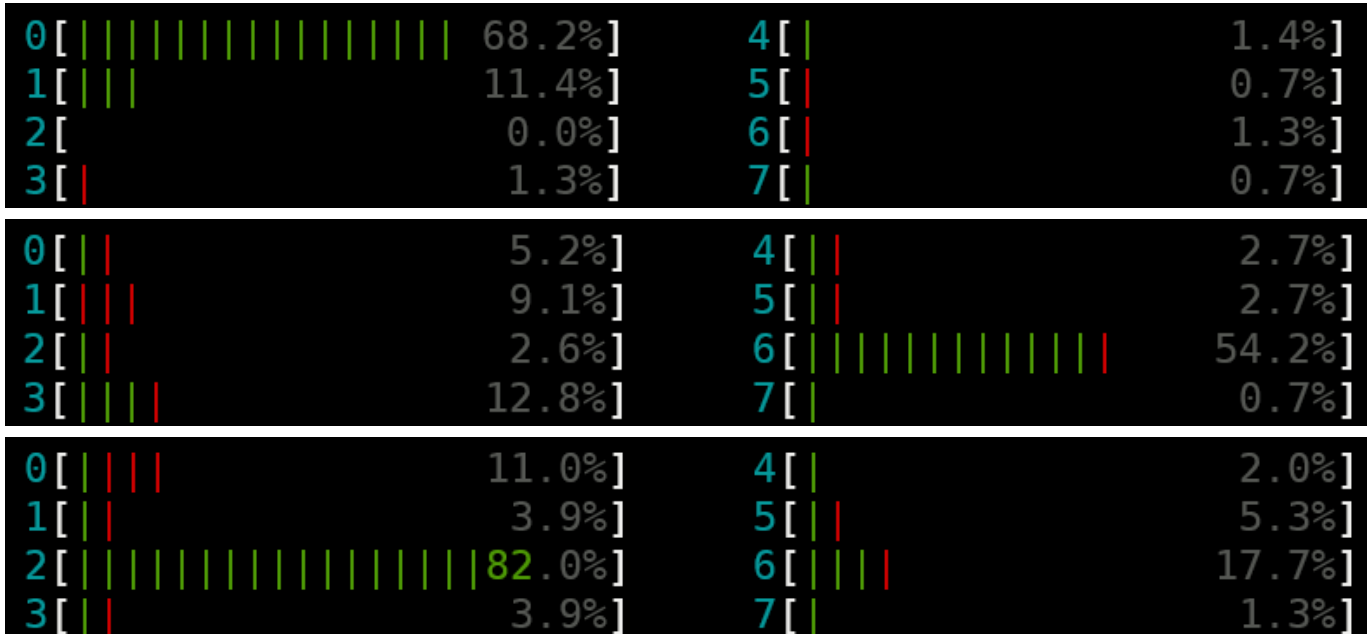
$ my@LAPTOP-C540D56J: ~
$ pstree -c 1705
zsh
├─concurrency_300
├─concurrency_300
├─concurrency_300
├─concurrency_300
├─concurrency_300
├─concurrency_300
└─concurrency_300

(25409) 29987 km ..., 29987 km à pied, ca use les souliers.
(25408) 29626 km ..., 29626 km à pied, ca use les souliers.
(25409) 29988 km ..., 29988 km à pied, ca use les souliers.
(25408) 29627 km ..., 29627 km à pied, ca use les souliers.
(25409) 29989 km ..., 29989 km à pied, ca use les souliers.
(25408) 29628 km ..., 29628 km à pied, ca use les souliers.
(25409) 29990 km ..., 29990 km à pied, ca use les souliers.
(25408) 29629 km ..., 29629 km à pied, ca use les souliers.
(25409) 29991 km ..., 29991 km à pied, ca use les souliers.
(25408) 29630 km ..., 29630 km à pied, ca use les souliers.
(25408) 29631 km ..., 29631 km à pied, ca use les souliers.
(25409) 29992 km ..., 29992 km à pied, ca use les souliers.
(25408) 29632 km ..., 29632 km à pied, ca use les souliers.
(25409) 29993 km ..., 29993 km à pied, ca use les souliers.
(25408) 29633 km ..., 29633 km à pied, ca use les souliers.
(25409) 29994 km ..., 29994 km à pied, ca use les souliers.
(25408) 29634 km ..., 29634 km à pied, ca use les souliers.
(25408) 29635 km ..., 29635 km à pied, ca use les souliers.
(25409) 29995 km ..., 29995 km à pied, ca use les souliers.
(25408) 29636 km ..., 29636 km à pied, ca use les souliers.
(25408) 29637 km ..., 29637 km à pied, ca use les souliers.
(25409) 29996 km ..., 29996 km à pied, ca use les souliers.
(25408) 29638 km ..., 29638 km à pied, ca use les souliers.
(25408) 29639 km ..., 29639 km à pied, ca use les souliers.
(25409) 29997 km ..., 29997 km à pied, ca use les souliers.
(25408) 29640 km ..., 29640 km à pied, ca use les souliers.
(25409) 29998 km ..., 29998 km à pied, ca use les souliers.
(25409) 29999 km ..., 29999 km à pied, ca use les souliers.
(25408) 29641 km ..., 29641 km à pied, ca use les souliers.
(25408) 29642 km ..., 29642 km à pied, ca use les souliers.
(25408) 29643 km ..., 29643 km à pied, ca use les souliers.
(25408) 29644 km ..., 29644 km à pied, ca use les souliers.

```

Exercice 2

Ma machine comporte 8 coeurs, on observe la consommation processeur à l'aide de htop. On constate que la mobilisation des coeurs est différentes pour plusieurs exécutions.



Exercice 3

On cherche la commande `fork` dans le manuel à l'aide de la commande `man fork`.

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

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.

Exercice 4

1. La primitive `fork` peut échouer si jamais la limite de threads autorisée a été atteinte : la table des PCB est pleine et on peut plus générer un nouveau PID ou si l'allocation mémoire a échouée à cause d'une mémoire trop étroite.

2. Après exécution on obtient l'arbre suivant :

```
s4my@LAPTOP-C540D56J: ~
$ pstree -c 35466 [12:07:32]
zsh—simple_fork—simple_fork
```

3. Ne pouvant pas prédire à l'avance le comportement de l'ordonnanceur du système, on ne peut pas savoir à l'avance si c'est le fils ou le père qui va se terminer en premier.

4. On modifie le code en ajoutant la ligne :

```
printf("My father's PID is : (%d) \n", getppid());
```

5. On obtient ainsi en sortie :

```
s4my@LAPTOP-C540D56J: ~/ProgSys/TP_1-2
$ ./bin/simple_fork [12:13:37]
(40024) All you need is love
My father's PID is : (35466)
(40025) All you need is love
My father's PID is : (40024)
```

6. On a donc l'arbre suivant :

```
s4my@LAPTOP-C540D56J: ~
$ pstree -cp 35466 [12:12:54]
zsh(35466)---simple_fork(40024)---simple_fork(40025)
```

7. On peut relancer l'exécution et on remarque que le processus lié à l'exécution du terminal n'a pas changé de PID (c'est normal, il est toujours en cours d'exécution) et que les PID des deux processus liés à l'exécution du fichier simple_fork sont nouveaux.

```
s4my@LAPTOP-C540D56J: ~
$ pstree -cp 35466 [12:16:44]
zsh(35466)---simple_fork(40636)---simple_fork(40637)
```

8. Les processus ont bien disparus à l'issue de l'exécution du programme : le pstree enraciné en Shell 1 n'a plus de fils.

Exercice 5

appel_fork1

```
flowchart LR
%%{ init : {"flowchart" : { "curve" : "stepBefore" }}}%%

SHELL[Terminal
pid_shell] --> P1[P1
pid_files : P2];
P1 --fork--> P2[P2
pid_files : 0]
```

On retrouve bien en sortie :

```
(P1) Sing it'
(P1) We will we will rock you (P2)
(P2) We will we will rock you (0)
```

appel_fork2

flowchart LR

```
SHELL[Terminal
pid_shell] --> P1[P1
pid_fils : P2];
P1 --fork---> P2[P2
pid_fils : 0]
```

Sortie:

```
(P1) We can light it up, up, up
(P2) So they can't put it out, out, out
```

appel_fork3

flowchart LR

```
%%{ init : {"flowchart" : { "curve" : "stepBefore" }}}%%

SHELL[Terminal
pid_shell] --> P1[P1
pid_fils1 : P2
pid_fils2 : P3];
P1 --fork---> P2[P2
pid_fils1 : 0
pid_fils2 : P4];
P2 --fork---> P4[P4
pid_fils1 : 0
pid_fils2 : 0];
P1 --fork---> P3[P3
pid_fils1 : P2
pid_fils2 : 0]
```

Sortie:

```
(P1) Do you do you Saint-Tropez (P2) (P3)
(P3) Do you do you Saint-Tropez (P2) (0)
(P2) Do you do you Saint-Tropez (0) (P4)
```

(P4) Do you do you Saint-Tropez (0) (0)

```
zsh(pid_shell)——appel_fork3(P1)——appel_fork3(P2)——appel_fork3(P4)
                                ↳appel_fork3(P3)
```

appel_fork4

```
flowchart LR
%%{ init : {"flowchart" : { "curve" : "stepBefore" }}}%%

SHELL[Terminal
pid_shell]

P1[P1
pid_fils1 : P2
pid_fils2 : P3]

P2[P2
pid_fils1 : 0
pid_fils2 : -1]

P3[P3
pid_fils1 : P2
pid_fils2 : 0]

SHELL-->P1
P1--fork--->P2
P1--fork--->P3
```

Sortie :

```
Bonjour, je suis Léodagan (P1).
Bonjour, je suis Guenièvre (P2), mon père est P1.
Bonjour, je suis Yvain (P3), mon père est P1.

zsh(pid_shell)——appel_fork4(P1)——appel_fork4(P2)
                                ↳appel_fork4(P3)
```

appel_fork5

```
flowchart LR
%%{ init : {"flowchart" : { "curve" : "stepBefore" }}}%%

SHELL[Terminal
pid_shell]

P1[P1
```

```
pid_fils1 : P2
pid_fils2 : P3
pid_fils3 : P4]
```

```
P2[P2
pid_fils1 : 0
pid_fils2 : P5
pid_fils3 : P6]
```

```
P3[P3
pid_fils1 : P2
pid_fils2 : 0
pid_fils3 : P7]
```

```
P4[P4
pid_fils1 : P2
pid_fils2 : P3
pid_fils3 : 0]
```

```
P5[P5
pid_fils1 : 0
pid_fils2 : 0
pid_fils3 : P8]
```

```
P6[P6
pid_fils1 : 0
pid_fils2 : P5
pid_fils3 : 0]
```

```
P8[P8
pid_fils1 : 0
pid_fils2 : 0
pid_fils3 : 0]
```

```
P7[P7
pid_fils1 : P2
pid_fils2 : 0
pid_fils3 : 0]
```

```
SHELL-->P1
P1--fork--->P2
P1--fork--->P3
P1--fork--->P4
P2--fork--->P5
P2--fork--->P6
P3--fork--->P7
P5--fork--->P8
```

Sortie :

```
(P1) Alors on danse (P2) (P3) (P4)
(P4) Alors on danse (P2) (P3) (0)
```



```
(P2) Alors on danse (0) (P5) (P6)
(P3) Alors on danse (P2) (0) (P7)
(P5) Alors on danse (0) (0) (P8)
(P6) Alors on danse (0) (P5) (0)
(P7) Alors on danse (P2) (0) (0)
(P8) Alors on danse (0) (0) (0)
```

```
zsh(pid_shell)——appel_fork5(P1)——appel_fork5(P2)——appel_fork5(P5)——appel_fork5(P8)
                                                                    |
                                                                    └─appel_fork5(P6)
└─appel_fork5(P3)——appel_fork5(P7)
  └─appel_fork5(P4)
```

Exercise 6

```
flowchart LR
%%{ init : { "flowchart" : { "curve" : "stepBefore" } } }%%

SHELL[Terminal
pid_shell]

P1[P1
pid_fils1 : P2
pid_fils2 : P3
royaume : Carmélide]

P2[P2
pid_fils1 : 0
pid_fils2 : -1
royaume : Logres]

P3[P3
pid_fils1 : P2
pid_fils2 : 0
royaume : Carmélide]

SHELL---P1
P1--fork--->P2
P1--fork--->P3
```

Sortie :

```
(Léodagan, PID : P1, PPID : pid_shell) voici mon royaume : Carmélide.
(Guenièvre, PID : P2, PPID : P1) voici mon royaume : Carmélide.
(Guenièvre, PID : P2, PPID : P1) voici mon nouveau royaume : Logres.
(Yvain, PID : P3, PPID : P1) voici mon royaume : Carmélide.
```

```
zsh(pid_shell)——royaume(P1)——royaume(P2)
                                                                    └─royaume(P3)
```

Exercice 7

Affichage 1

```
Père - 1 Frère Jacques, Frère Jacques
Père - 1 Dormez-vous ? Dormez-vous ?
Fils - 2 Frère Jacques, Frère Jacques
Père - 1 Sonnez les matines ! Sonnez les matines !
Fils - 2 Dormez-vous ? Dormez-vous ?
Père - 1 Ding, daing, dong, Ding, daing, dong ...
Fils - 2 Sonnez les matines ! Sonnez les matines !
Fils - 2 Ding, daing, dong, Ding, daing, dong ...
```

Affichage possible.

Affichage 2

```
Père - 1 Frère Jacques, Frère Jacques
Père - 1 Dormez-vous ? Dormez-vous ?
Fils - 2 Frère Jacques, Frère Jacques
Père - 2 Sonnez les matines ! Sonnez les matines !
Fils - 2 Dormez-vous ? Dormez-vous ?
Père - 2 Ding, daing, dong, Ding, daing, dong ...
Fils - 2 Sonnez les matines ! Sonnez les matines !
Fils - 2 Ding, daing, dong, Ding, daing, dong ...
```

Affichage impossible car la valeur de `i` est égale à 1 pour le père, en effet, `i` est incrémenté seulement si `pid_fils` est égal à 0 i.e. seulement pour le fils.

Affichage 3

```
Père - 1 Frère Jacques, Frère Jacques
Fils - 2 Frère Jacques, Frère Jacques
Père - 1 Dormez-vous ? Dormez-vous ?
Père - 1 Sonnez les matines ! Sonnez les matines !
Fils - 2 Dormez-vous ? Dormez-vous ?
Père - 1 Ding, daing, dong, Ding, daing, dong ...
Fils - 2 Sonnez les matines ! Sonnez les matines !
Fils - 2 Ding, daing, dong, Ding, daing, dong ...
```

Affichage impossible, en effet, le code du fils ne peut être appelé qu'une fois le fork réalisé, ce qui n'est pas le cas avant que le père chante les deux premières lignes.

Affichage 4

```
Père - 1 Frère Jacques, Frère Jacques
Père - 1 Dormez-vous ? Dormez-vous ?
Fils - 2 Frère Jacques, Frère Jacques
Père - 1 Sonnez les matines ! Sonnez les matines !
Fils - 2 Dormez-vous ? Dormez-vous ?
Père - 1 Ding, daing, dong, Ding, daing, dong ...
Fils - 2 Ding, daing, dong, Ding, daing, dong ...
Fils - 2 Sonnez les matines ! Sonnez les matines !
```

Affichage impossible, en effet, dans le code du fils, les deux dernières lignes sont inversées.

Affichage 5

```
Père - 1 Frère Jacques, Frère Jacques
Père - 1 Dormez-vous ? Dormez-vous ?
Fils - 2 Frère Jacques, Frère Jacques
Fils - 2 Dormez-vous ? Dormez-vous ?
Fils - 2 Sonnez les matines ! Sonnez les matines !
Fils - 2 Ding, daing, dong, Ding, daing, dong ...
Père - 1 Sonnez les matines ! Sonnez les matines !
Père - 1 Ding, daing, dong, Ding, daing, dong ...
```

Affichage possible.

Affichage 6

```
Père - 1 Frère Jacques, Frère Jacques
Père - 1 Dormez-vous ? Dormez-vous ?
Père - 1 Sonnez les matines ! Sonnez les matines !
Père - 1 Ding, daing, dong, Ding, daing, dong ...
Fils - 2 Frère Jacques, Frère Jacques
Fils - 2 Dormez-vous ? Dormez-vous ?
Fils - 2 Sonnez les matines ! Sonnez les matines !
Fils - 2 Ding, daing, dong, Ding, daing, dong ...
```

Affichage possible.

Exercice 8

Dans un système à temps partagé de type Linux, la création de processus n'est pas possible. En effet, tout nouveau processus est en réalité le fils d'un processus déjà existant (en cours d'exécution) qui va réaliser un fork. C'est pour cette raison qu'en exécutant la commande `ps tree` on obtient une structure arborescente des programme en exécution avec un seul père en commun : `systemd`.

Exercice 9

On introduit les deux fonctions suivantes :

```
void singBoy(int i)
{
    i++;
    printf("Fils - %d Frère Jacques, Frère Jacques \n", i);
    sleep(2);
    printf("Fils - %d Dormez-vous ? Dormez-vous ? \n", i);
    sleep(2);
    printf("Fils - %d Sonnez les matines ! Sonnez les matines !\n", i);
    sleep(2);
    printf("Fils - %d Ding, daing, dong, Ding, daing, dong ...\n",
i);sleep(2);
    exit(EXIT_SUCCESS);
}

void singMan(int i)
{
    wait(NULL);
    printf("Père - %d Sonnez les matines ! Sonnez les matines !\n", i);
    sleep(2);
    printf("Père - %d Ding, daing, dong, Ding, daing, dong ...\n", i);
    sleep(2);
}
```

L'encapsulation est réalisée grâce à un `wait(NULL)` avant que le père ne chante et d'un `exit(EXIT_SUCCESS)` après que le fils ait fini (comme les sémaphores). On obtient donc bien la sortie suivante :

```
Père - 1 Frère Jacques, Frère Jacques
Père - 1 Dormez-vous ? Dormez-vous ?
Fils - 2 Frère Jacques, Frère Jacques
Fils - 2 Dormez-vous ? Dormez-vous ?
Fils - 2 Sonnez les matines ! Sonnez les matines !
Fils - 2 Ding, daing, dong, Ding, daing, dong ...
Père - 1 Sonnez les matines ! Sonnez les matines !
Père - 1 Ding, daing, dong, Ding, daing, dong ...
```

Exercice 10

Code complexe 1

```
flowchart LR
%%{ init : {"flowchart" : { "curve" : "stepBefore" }}}%%

SHELL[Terminal
pid_shell]
```

```

P1[P1
pid_fils1 : P2
i : 3]

P2[P2
pid_fils1 : 0
i : 2]

P3[P3
pid_fils1 : P2
i : 3]

P4[P4
pid_fils1 : 0
i : 2]

SHELL---P1
P1--fork--->P2
P1--fork--->P3
P2--fork--->P4

```

Sortie :

```

Je suis P3, et la valeur de i est : 3
Je suis P2, et la valeur de i est : 2
Je suis P4, et la valeur de i est : 2
Je suis P1, et la valeur de i est : 3

zsh(pid_shell)——complex1(P1)——┐——complex1(P2)——complex1(P4)
                                └——complex1(P3)

```

Code complexe 2

```

flowchart LR
%%{ init : {"flowchart" : { "curve" : "stepBefore" }}}%%

SHELL[Terminal
pid_shell]

P1[P1
pid_fils1 : P2
pid_fils2 : P3
i : 3]

P2[P2
pid_fils1 : 0
pid_fils2 : -1
i : 0]

```

```
P3[P3
pid_fils1 : P2
pid_fils2 : 0
i : 3]
```

```
P4[P4
pid_fils1 : 0
pid_fils2 : -1
i : 0]
```

```
SHELL---P1
P1--fork--->P2
P1--fork--->P3
P2--fork--->P4
```

Sortie :

```
Je suis P4
pid_fils1 : 0
pid_fils2 : -1
i : 0
```

```
Je suis P2
pid_fils1 : 0
pid_fils2 : -1
i : 0
```

```
Je suis P1
pid_fils1 : P2
pid_fils2 : P3
i : 3
```

```
Je suis P3
pid_fils1 : P2
pid_fils2 : 0
i : 3
```

```
zsh(pid_shell)——complex2(P1)——┐complex2(P2)——complex2(P4)
                               └─┬─complex2(P3)
```

Exercice 11

1. Ce programme a pour effet de réaliser des fork à l'infini. En effet, à chaque nouveau `fork()`, l'entier `fork_return_value` prend la valeur 0, la condition de sortie n'est donc jamais vérifiée, il s'agit donc d'une boucle infinie d'où le nom de fork bomb.
2. À partir d'un certain nombre de fork, les suivants vont échouer. Cela est dû à une erreur système : on ne peut plus créer de nouveau processus et l'erreur renvoyée nous fait sortir de la boucle.
3. En rajoutant l'instruction

```
sleep(1);
```

entre chaque appel de fork, on a le temps d'observer l'effet du programme sur l'arbre des processus du système :

```
zsh(41479)——bomb(104071)└─bomb(104072)
                               │└─bomb(104097)
                               │└─bomb(104109)
                               │└─bomb(104131)
                               │└─bomb(104133)
                               │└─bomb(104165)
                               │└─bomb(104166)
                               │└─bomb(104180)
                               │└─bomb(104206)
                               │└─bomb(104213)
                               │└─bomb(104214)
                               │└─bomb(104245)
                               │└─bomb(104246)
                               │└─bomb(104253)
                               │└─bomb(104278)
                               │└─bomb(104280)
                               │└─bomb(104286)
                               └─bomb(104294)
```