INF2610
Noyau d'un système d'exploitation

Chapitre 2 - Processus

Département de génie informatique et génie logiciel

POLYTECHNIQUE MONTRÉAL
AFFILIÉE À L'UNIVERSITÉ DE MONTRÉAL

Sommaire

- Qu’est ce qu’un processus ?

- Gestion de processus (Linux/Unix)
  - Création de processus
  - Remplacement d’espace d’adressage
  - Terminaison de processus
  - Attente de la fin d’un processus fils
  - Partage de fichiers entre processus

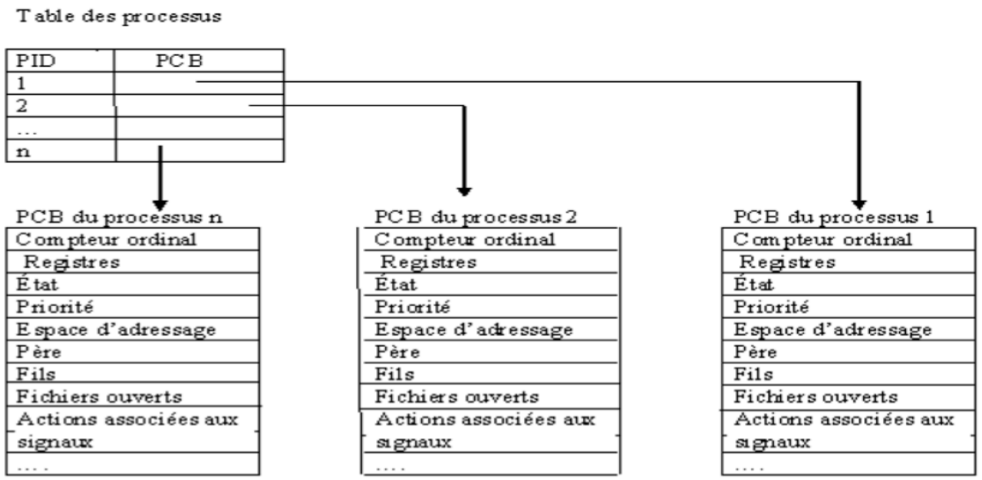
- Exercices

Qu’est-ce qu’un processus?

- Un processus est un programme en cours d’exécution.
- Au niveau du système d’exploitation (SE), un processus:
  - a un numéro d’identification unique appelé PID (Process Identifier) et
  - est représenté par une structure de données appelée bloc de contrôle du processus PCB (Process Control Bloc).
- Le PCB d’un processus regroupe toutes les informations nécessaires à la gestion de son exécution.
- Dans le cas de linux, le PCB est une structure de type ***task_struct.***

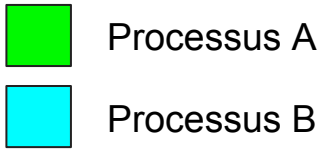
Qu'est-ce qu'un processus?

Dans le cas de linux, la table des processus est une liste doublement chainée de structures task_struct

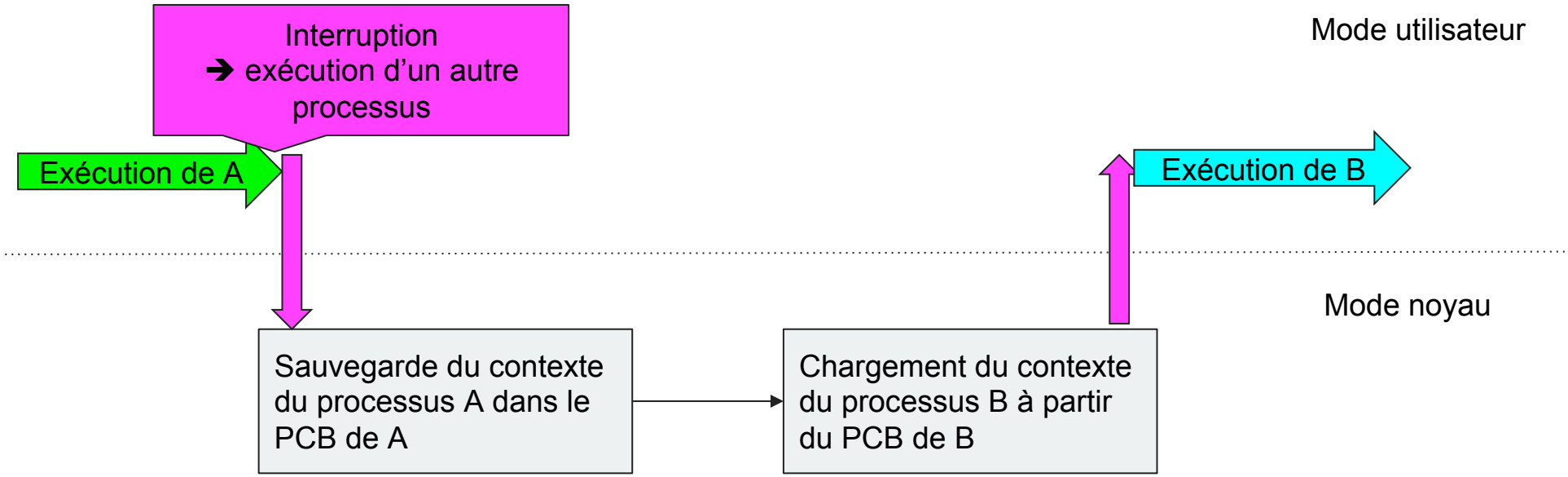


- **Compteur ordinal** : adresse de la prochaine instruction à exécuter.
- **Registres** : espace réservé à la sauvegarde des contenus des registres lors de la suspension de l'exécution du processus.
- **État :** prêt, en exécution, bloqué, etc.
- **Priorité** : sert à l'ordonnancement des processus (choisir le plus prioritaire).
- **Espace d'adressage** : regroupe le code, les données, la pile d'exécution, etc. du processus.
- **Fichiers ouverts** : accessibles au processus.
- etc.

Compteur ordinal & registres



Changement de contexte (statut)

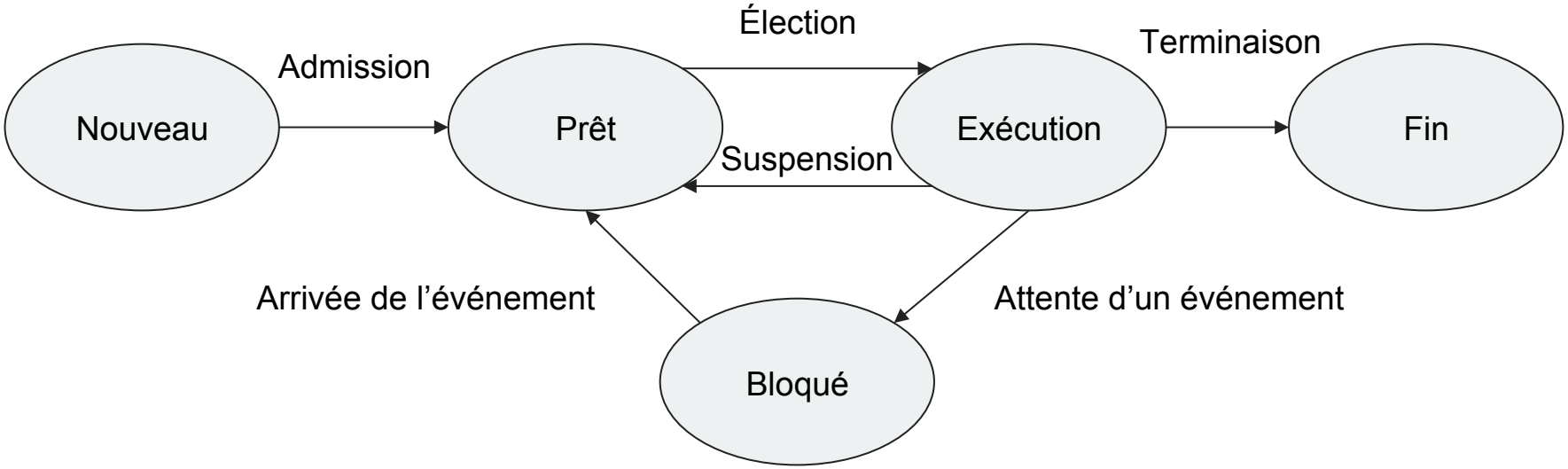


États d’un processus

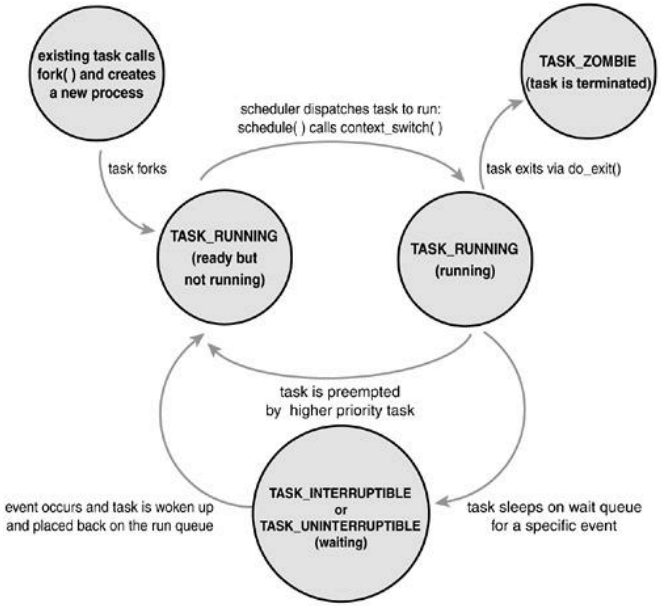
- L’état courant d’un processus permet au SE de savoir s’il est en exécution, en attente d’exécution ou en attente d’un événement (ex. fin d’E/S).

- Cette information est importante pour le gestionnaire de processeurs. Le SE va allouer un processeur à un processus en attente d’exécution.

États d’un processus (2)

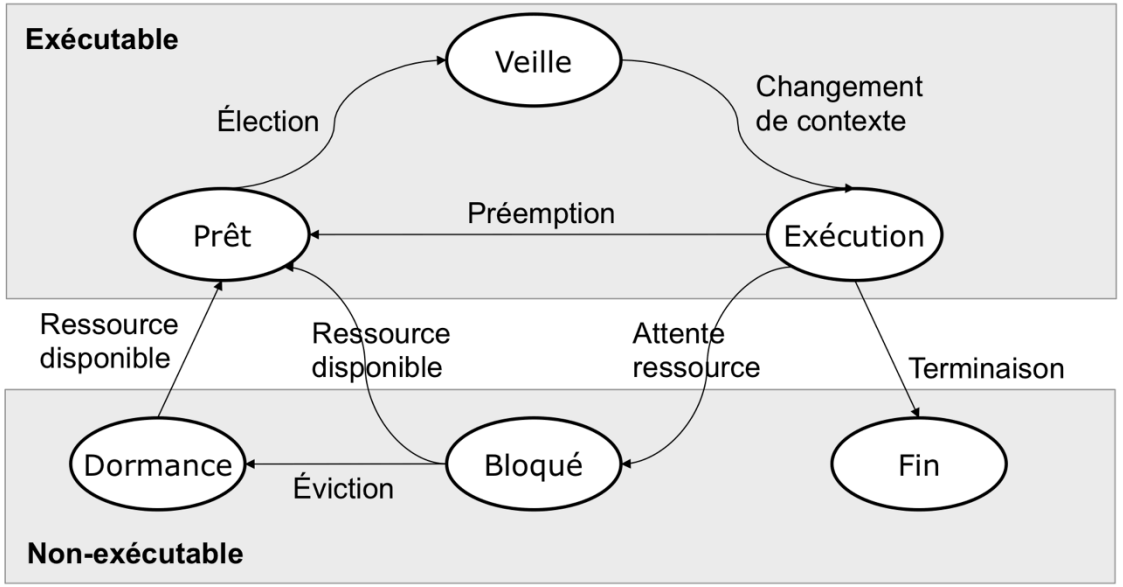


États d'un processus – LINUX (3)



- TASK_RUNNING – Tâche prête ou en exécution.
- TASK_INTERRUPTIBLE - Tâche en attente (événement, E/S…).
- TASK_UNINTERRUPTIBLE - Tâche en attente non interruptible par un signal.
- TASK_STOPPED – Tâche stoppée.
- TASK_ZOMBIE - Tâche terminée mais attend que son parent ait récupéré les informations sur sa terminaison (la tâche a toujours une entrée dans la table des processus).

États d'un processus – WINDOWS (4)



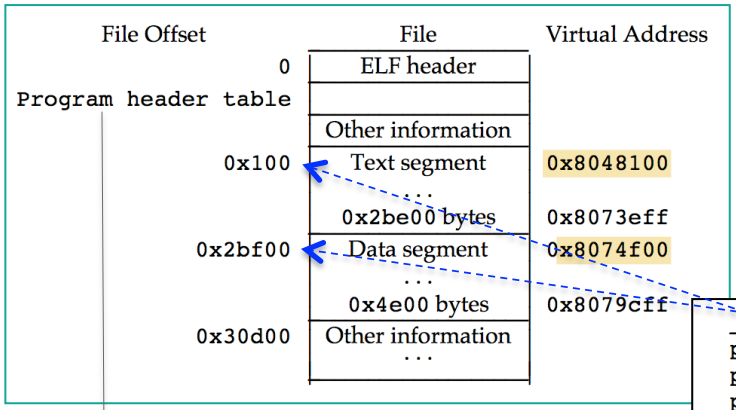
Espace d'adressage d'un processus

- Le fichier exécutable permet au SE de :
  - construire les structures de données représentant l'espace d'adressage du processus et
  - récupérer d'autres informations comme l'adresse de la première instruction à exécuter.

- Parmi les structures de données représentant l'espace d'adressage d'un processus, il y a la table des pages (TDP) où sont stockées les informations, notamment, de localisation et d'accès de chaque page de l'espace d'adressage.

Espace d'adressage d'un processus (2)

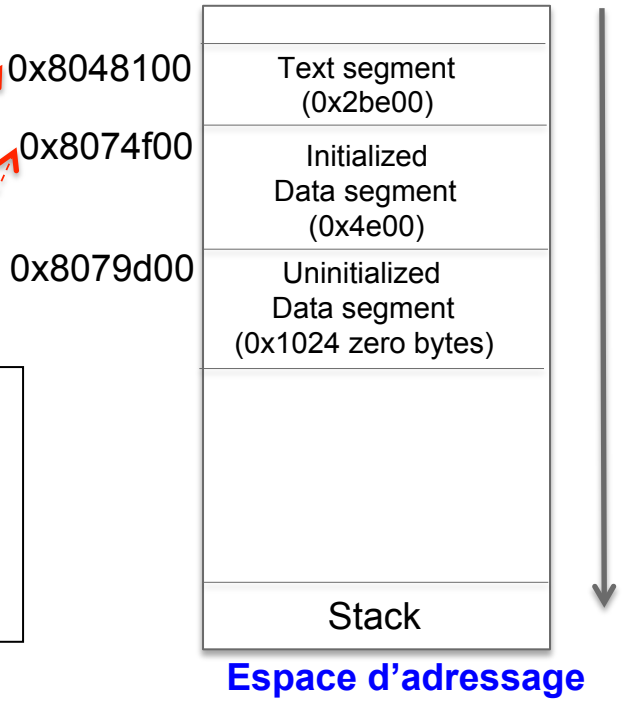
Fichier exécutable ELF (Executable and Linkable Format)



http://flint.cs.yale.edu/cs422/doc/ELF_Format.pdf

Program header table

Member	Text	Data
p_type	PT_LOAD	PT_LOAD
p_offset	0x100	0x2bf00
p_vaddr	0x8048100	0x8074f00
p_paddr	unspecified	unspecified
p_filesz	0x2be00	0x4e00
p_memsz	0x2be00	0x5e24
p_flags	PF_R+PF_X	PF_R+PF_W+PF_X
p_align	0x1000	0x1000



Espace d'adressage

Espace d'adressage d'un processus (2)

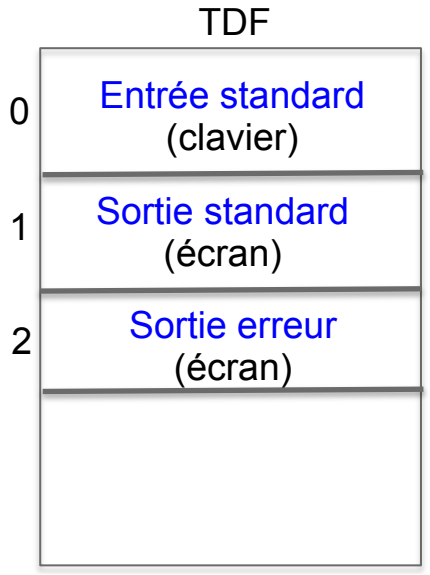
Program header table

Member	Text	Data
p_type	PT_LOAD	PT_LOAD
p_offset	0x100	0x2bf00
p_vaddr	0x8048100	0x8074f00
p_paddr	unspecified	unspecified
p_filesz	0x2be00	0x4e00
p_memsz	0x2be00	0x5e24
p_flags	PF_R+PF_X	PF_R+PF_W+PF_X
p_align	0x1000	0x1000

- **p_type** : type du segment (chargeable (PT_LOAD), chargeable dynamiquement, etc.).
- **p_offset** : début du segment dans le fichier.
- **p_vaddr** : adresse virtuelle du début du segment.
- **p_paddr** : ignoré pour les processus utilisateur.
- **p_filesz** : taille du segment dans le fichier exécutable.
- **p_memsz** : taille du segment dans l'espace d'adressage.
- **p_flags** : mode d'accès, etc.
- **p_align** : taille d'une page.

Fichiers ouverts

- Un processus peut accéder, durant son exécution, à certains fichiers ouverts (ordinaires ou spéciaux).
- Les fichiers accessibles au processus sont regroupés dans une table privée appelée table des descripteurs de fichiers (TDF) du processus.
- Trois fichiers sont accessibles au processus dès sa création :

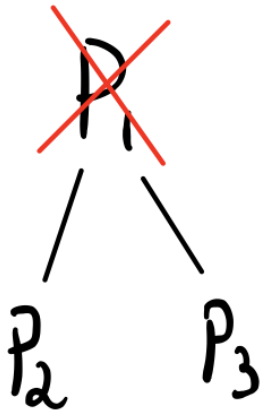


Père et fils

- Chaque processus a un processus père.
- Un processus peut créer d'autres processus qui peuvent, à leur tour, en créer d'autres (structure en arbre).
- Les processus sont créés et détruits dynamiquement et s'exécutent en concurrence.
- La gestion de la relation père-fils entre les processus diffère d'un système d'exploitation à un autre.

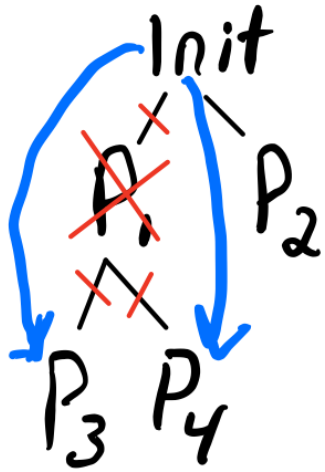
Père et fils (2)

- Que deviennent les processus fils lorsque leur processus père a terminé son exécution ?



Windows : Chaque processus a un pointeur vers son parent. Si le parent se termine, les processus fils ne sont pas adoptés par d'autres processus.

Linux/Unix : Chaque processus a un pointeur vers son parent. Si le parent se termine, les processus fils sont adoptés par un autre processus (le processus init de PID 1). Ce denier est la racine de l'arbre des processus gérés par le système d'exploitation.



Gestion de processus : appels système LINUX / UNIX (POSIX)

Appel système	Service
getpid()	Retourne le **PID** du processus **appelant.**
getppid()	Retourne le **PID** du **parent** du **processus appelant.**
fork()	Crée un processus fils du processus appelant.
_exit(value)	Termine le processus appelant.
wait(&status), etc.	Met le **processus appelant** en **attente** de la terminaison d'un processus fils avec possibilité de récupérer l'état de terminaison de ce dernier.
execl(chemin, nom, params), etc.	Remplace **le code exécutable** du processus appelant par celui d'un autre programme.

Création de processus - Fonction fork()

- La création d'un processus est réalisée par duplication avec le principe de « Copy On Write » (COW). Le processus fils créé est un clone du processus père (espaces d'adressage identiques, tables de descripteurs de fichiers identiques, etc.).

- Le processus fils exécute **le même code que le processus parent** à **partir de l'instruction qui suit l'appel à la fonction fork(). Cette fonction retourne donc une valeur au processus père et une valeur au processus fils créé :**
  - le processus **père** reçoit le **PID de son fils** et
  - le fils reçoit la valeur 0.

- La création d'un processus pourrait échouer s'il n'y a pas suffisamment d'espace, la limite du nombre de processus créés par l'utilisateur est atteinte ou encore la limite du nombre total de processus dans le système est atteinte. La fonction **fork** retourne, dans ce cas, -1.

Terminaison d'un processus

- Un processus se termine normalement en invoquant l'appel système **_exit(value**), où **value** est la valeur de retour du processus (codée sur 8 bits).

- La fonction **exit** de la librairie C fait appel à **_exit** pour terminer le processus appelant.

- La terminaison peut être forcée par un signal (ex: SIGKILL). Il s'agit, dans ce cas, d'une terminaison anormale.

- Lorsqu'un processus se termine, son état de terminaison est enregistré dans le PCB du processus. Le processus bascule ensuite vers l'état **zombie**.

- L'état de terminaison comporte le type de terminaison **normale/anormale**, la valeur de retour du processus (si fin normale), le signal ayant causé sa terminaison (si fin anormale), etc.

Attente de la fin d'un processus fils

- Un processus père peut attendre la fin d'un fils et récupérer son état de terminaison via, par exemple, l'appel système **wait(&status)** ou encore **waitpid(pid, &status, options).** Cette attente peut être cependant interrompue par un signal (ex. SIGKILL).

- En cas de succès, ces appels système retournent le PID du fils terminé (après avoir copié l'état de terminaison du fils dans la variable **status**).

- En cas d'échec, ils retournent **-1** (par ex., **le processus appelant n'a pas ou n'a plus de fils**).

Attente de la fin d'un processus fils (2)

Appels système wait(&status) et waitpid(pid, &status, options)

- Macros sur **status :**
  - WIFEXITED(status): vrai si la terminaison est normale.
  - WIFSIGNALED(status): vrai si la terminaison est anormale.
  - WEXITSTATUS(status): valeur de retour du processus (_exit(valeur)).
  - WTERMSIG(status): numéro du signal ayant causé la terminaison anormale. …

- Pour vérifier si un processus fils de numéro pid est terminé sans le mettre en attente (sans le bloquer), il suffit d'utiliser l'option WNOHANG: **waitpid(pid, &status, WNOHANG).** Cet appel retourne :
  - 0, si le processus n'est pas terminé,
  - pid, si le processus est terminé et
  - -1 en cas d'erreur.

Exemple SimpleFork (fork(), wait(NULL), _exit(value))

```
$ ./run.sh ProcessusEtThreads/
SimpleFork
```

1. Que fait ce programme ?
2. Donnez le sous arbre des processus qu'il crée ?
3. Quel est le code exécuté / chemin d'exécution de chaque processus du sous arbre, y compris le processus principal ?

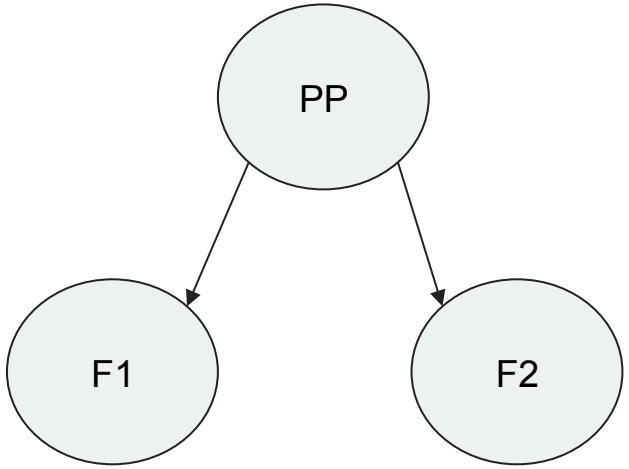
Exemple SimpleFork

```
1.  // programme deuxfils.c
2.  #include <sys/wait.h> // pour wait
3.  #include <stdio.h> // pour printf
4.  #include <unistd.h> // pour fork
5.  void fils(int i);
6.  int main() {
7.      if (fork()==0) { // premier fils
8.                          fils(1);
9.      } else if (fork()==0) { // second  fils
10.                         fils(2);
11.     } else  {
12.             if (wait(NULL) > 0)
13.                     printf("fin du fils\n");
14.             if (wait(NULL) > 0)
15.                     printf("fin du fils\n");
16.       }
17.     return 0;
18. }
19. void fils(int i) {
20.     sleep(2);
21.     _exit(i);
22. }
```

Exemple SimpleFork (2)

Réponses :

- Ce code crée deux fils à partir du processus principal PP.
- Chaque fils exécute la fonction fils(): Chaque fils va se bloquer pendant 2 secondes puis se terminer en retournant la valeur du paramètre de la fonction fils().
- Le processus principal attend qu'un processus se termine puis affiche à l'écran « fin d'un fils ».
- Le processus répète ensuite cette dernière étape pour attendre son autre fils.



Comment le père peut-il récupérer les états de terminaison de ses fils ?

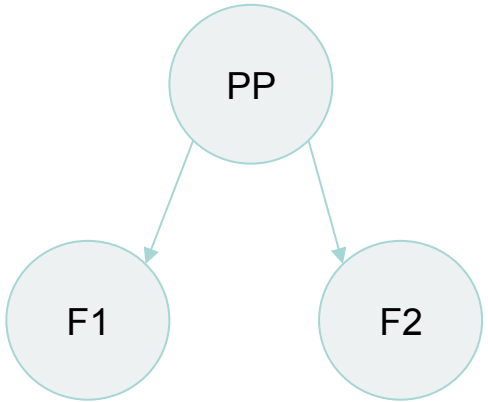
Exemple DeuxFils (fork(), wait(&status), _exit(value))

$ ./run.sh ProcessusEtThreads/DeuxFils

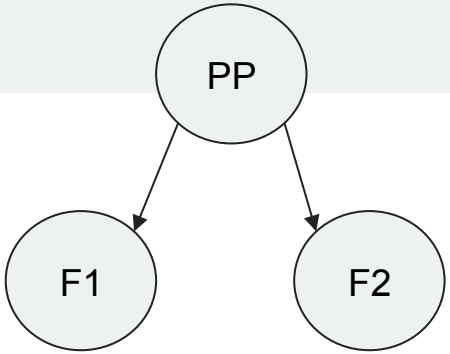
Il s’agit d’une extension de l’exemple précédent pour récupèrer les états de terminaison des processus fils.

Exemple DeuxFils

```
1.  // programme deuxfils.c
2.  #include <sys/wait.h> // pour wait
3.  #include <stdio.h>
4.  #include <unistd.h>
5.  void fils(int i);
6.  int main() {
7.      int status;
8.      if (fork()==0) { // premier filss
9.                              fils(1);
10.     } else if (fork()==0) { // second  fils
11.                             fils(2);
12.     } else  {    if (wait(&status) > 0 && WIFEXITED(status))
13.                             printf("fin du fils%d\n",  WEXITSTATUS(status));
14.                  if (wait(&status) > 0 && WIFEXITED(status))
15.                             printf("fin du fils%d\n", WEXITSTATUS(status));
16.         }
17.     return 0;
18. }
19. void fils(int i) {
20.     sleep(2);
21.     _exit(i);
22. }
```



Exemple DeuxFils (2)



Extension 1 :

Complétez/modifiez le code pour que :

- chaque processus (PP, F1 et F2) affiche à l'écran son PID suivi de celui de son père, juste avant de se terminer,
- les deux printf « fin du fils » affichent également le pid du fils terminé.

Extension 2 :

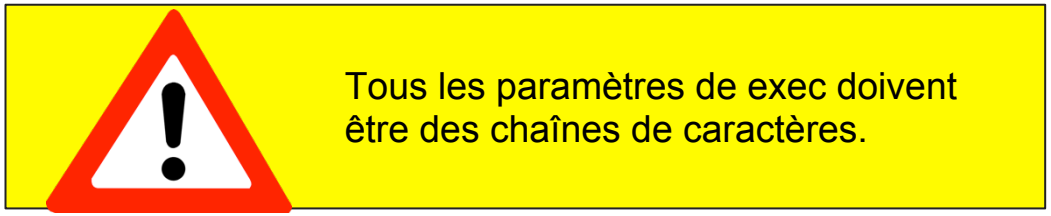
Ajoutez :

1. une variable entière globale v initialisé à *10* « int v=10; »,
2. une boucle « for » pour Incrémenter *10000* fois cette variable dans la fonction F juste après « sleep(2) », et
3. un affichage de la valeur de *v* ainsi que le *pid* du processus afficheur, juste après la bouche « for » et aussi juste avant « return 0; ».

Donnez pour chaque processus la valeur de v affichée à l'écran.

Remplacement du code d'un processus

- Il est possible de remplacer le code d'un processus par un autre à l'aide d'un appel système de la famille **exec : execl, execlp, execle, execv, execvp et execve**.
- **"l"** (resp. **"v"**) indique que les mots composant la ligne de commande du nouveau code exécutable sont passés séparément l'un à la suite de l'autre (resp. sont passés via un vecteur de mots).
- **"p"**: indique que le chemin d'accès au nouveau code exécutable est sauvegardé dans la variable PATH (echo $PATH ➔ ensemble de chemins d'accès) .
- "e" signifie qu'il est possible d'ajouter un vecteur de variables d'environnement (accessibles à partir du nouveau code).



Remplacement du code d'un processus (2)

```
int execl(const char* path, const char* arg, …, NULL)
int execlp(const char* file, const char* arg, …, NULL)
int execle(const char* path, const char* arg, …, NULL, char* const envp[])
int execv(const char* path, char* const argv[])
int execvp(const char* file, char* const argv[])
int execve(const char* path, char* const argv[], char* const envp[])
int execvpe(const char* file, char* const argv[], char* const envp[])
```

- En cas de succès, le processus abandonne le code courant pour débuter le nouveau code (**pas de retour à l'ancien code!**).
  ➔ L'espace d'adressage du processus est remplacé par un autre construit à partir de l'exécutable indiqué par le paramètre path ou file.

- En cas d'échec, le processus continue avec le code courant (pas remplacement de code).

Remplacement du code d'un processus (3)

Exemples :

- Pour le code courant par le code exécutable **ls** avec l'option **-a** :
  - **execl("/bin/ls", "ls", "-a", NULL);**
  - **execlp("ls", "ls", "-a", NULL);**
  - char* const params[] = { "ls", "-a", NULL }; **execv("/bin/ls", params);**
  - char* const params[] = { "ls", "-a", NULL }; **execvp("ls", params);**

- Pour exécuter le code exécutable **second** avec des variables d'environnement.
  - char* env[] = { "maVar=allo", NULL }; **execle("./second", "second", NULL, env);**
  - char* env[] = { "maVar=allo", NULL }; char* const params[] = { "second", NULL }; **execve("./second", params, env);**

Remplacement du code d'un processus avec exec()

$ ./run.sh ProcessusEtThreads/ForkExec

$ ./run.sh ProcessusEtThreads/Execle

- Le code forkExec.c dans ForkExec est une variante du code SimpleFork où les processus fils créés changent de codes exécutables.

- Le code first.c dans Execle affiche ses arguments et ses variables d'environnement. Il appelle ensuite execle pour remplacer le code exécutable de first.c par celui de second.c.

Exemple ForkExec

```
// programme forkExec.c
int main( ) {
    int status;
    if (fork()==0) { // premier fils
              printf("Le premier fils de pid=%d va se transformer \n",getpid());
              execl("/bin/ls","ls", "-l",NULL);
              printf("echec de execl de ls  -l./\n");
              exit(1);
    } else if (fork()==0) { // second  fils
               printf("Le second fils de pid=%d va se transformer \n",getpid());
               char* const params[] = { “ls”, “-a”, NULL };
               execv("/bin/ls”,params);
               printf("echec de execv de ls –a./\n");
               exit(1);
              }
```

```
else  {
                if ((status=wait(NULL)) > 0)
                   printf("fin du fils de pid=%d\n", status);
                if ((status=wait(NULL))>0)
                   printf("fin du fils de pid=%d\n", status);
                }
    return 0;
}
```

Remplacez execl et execv par respectivement execlp et execvp

Exemple Execle

```
1.  // programme first.c
2.  #include <stdio.h>
3.  #include <unistd.h>
4.  int main(int argc, char* argv[], char*env[]) {
5.    printf("ici first: argc = %d \n", argc);
6.    int i;
7.    for(i=0; i<argc;i++)
8.      printf("ici first: argv[%d]=%s \n", i, argv[i]);
9.    for(i=0;env[i]!=NULL;i++)
10.     printf("ici first: env[%d]=%s \n", i, env[i]);
11. printf("\nfirst va etre remplace par second \n");
12.   char *envbis[] = { "maVar=allo", NULL };
13.   execle("./second", "second", NULL, envbis);
14.   _exit(0);
15. }
```

```
1.  // programme second.c
2.  #include <stdio.h>
3.  #include <unistd.h>
4.  int main(int argc, char* argv[], char*env[]) {
5.   printf("ici %s: argc = %d \n", argv[0], argc);
6.   int i;
7.   for(i=0; i<argc;i++)
8.     printf("ici %s: argv[%d]=%s \n", argv[0], i, argv[i]);
9.   for(i=0;env[i]!=NULL;i++)
10.    printf("ici %s: env[%d]= %s \n", argv[0], i, env[i]);
11. _exit(0);
12. }
```

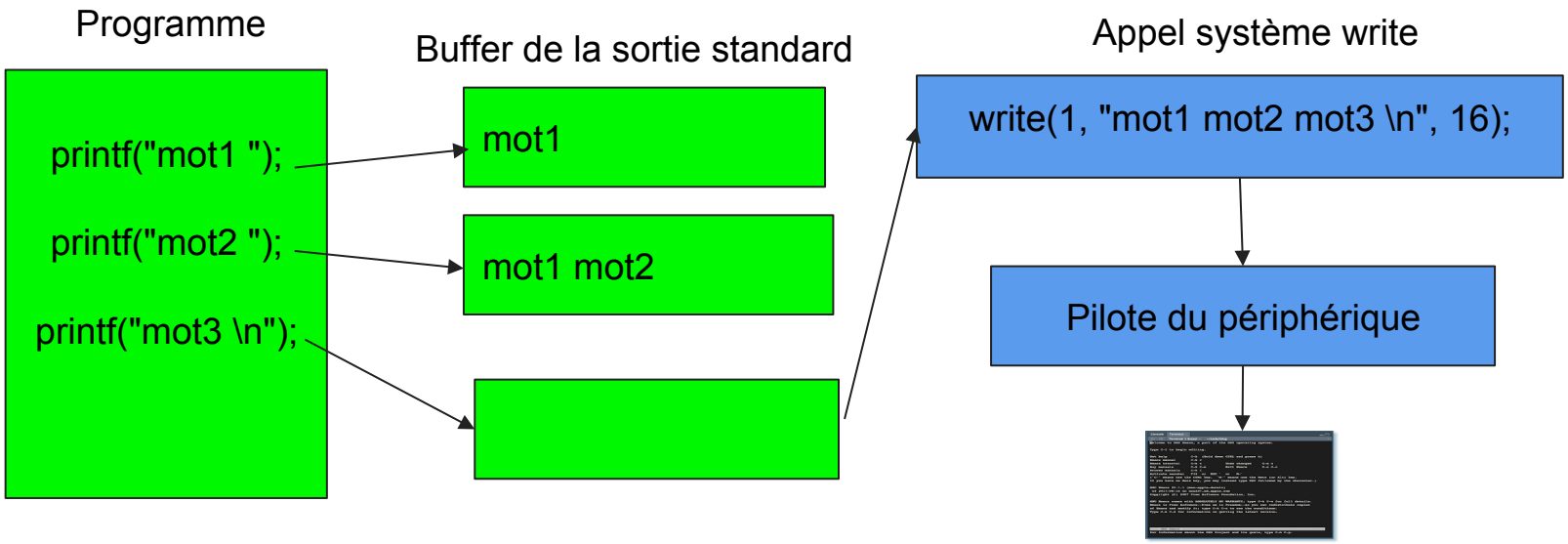
Exemple Execle (2)

```
$ ./run.sh ProcessusEtThreads/Execle
gcc -o second second.c; gcc -o a.out first.c
ici first: argc = 1
ici first: argv[0]=./a.out
ici first: env[0]=HOSTNAME=22210b2199d1
ici first: env[1]=HOME=/root
ici first: env[2]=OLDPWD=/
ici first: env[3]=TERM=xterm
ici first: env[4]=ARGV=ProcessusEtThreads/Env2
ici first: env[5]=PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
ici first: env[6]=DIR=ProcessusEtThreads/Env2
ici first: env[7]=PWD=/codes/ProcessusEtThreads/Env2
ici first: env[8]=ARGC=1
```

```
first va etre remplace par second
ici second: argc = 1
ici second: argv[0]=second
ici second: env[0]= maVar=allo
```

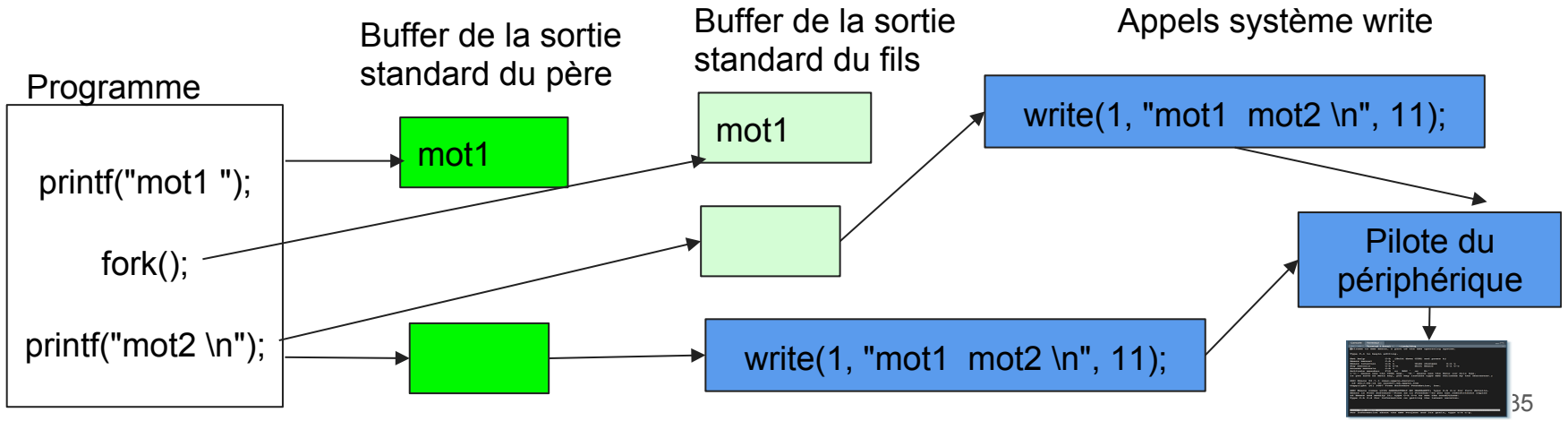
Partage de fichiers entre processus père et fils

- Par défaut, la sortie standard d'un processus est bufférisé par ligne.

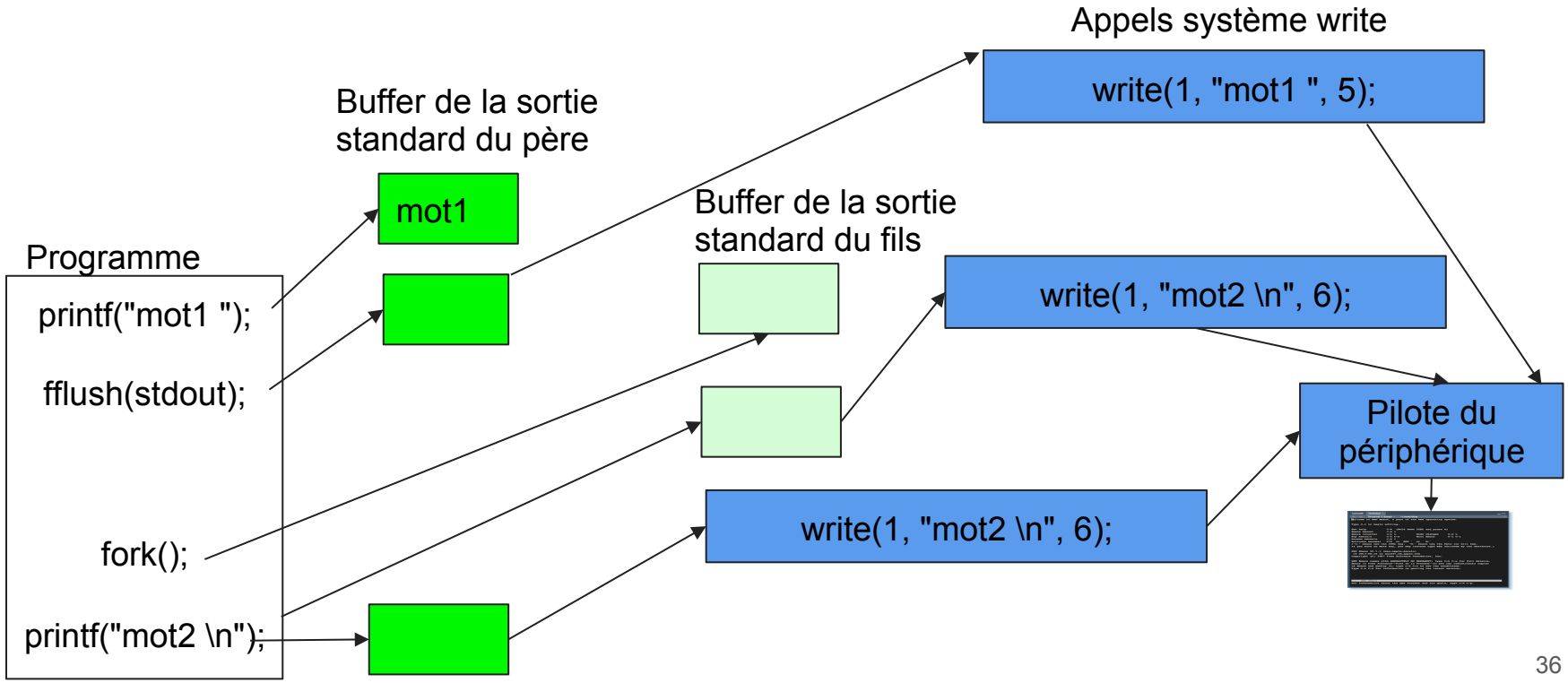


Partage de fichiers entre processus père et fils (2)

- Rappel : la fonction fork crée un processus fils par duplication.
- À la création du fils, sa table de descripteurs de fichiers est une copie de celle de son père. Le père et le fils ont donc les mêmes fichiers ouverts (la même entrée standard, la même sortie standard et la même la sortie erreur, etc.).
- Aussi, le buffer de sa sortie standard est une copie de celui de son père.

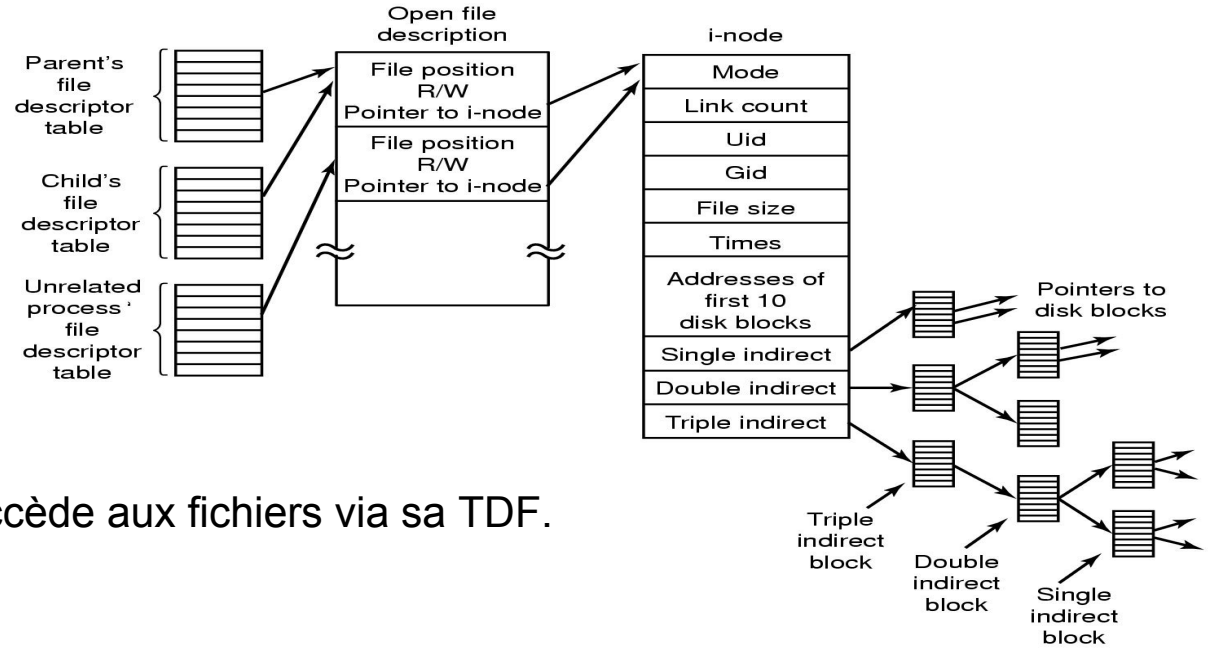


Partage de fichiers entre processus père et fils (3)



Partage de fichiers entre processus père et fils (4)

- Rappel : la fonction fork crée un processus fils par duplication.
- À la création du fils, la table des descripteurs de fichiers (TDF) est une copie de celle de son père.



Tannenbaum

- Un processus accède aux fichiers via sa TDF.

printf & write

$ ./run.sh ProcessusEtThreads/PrintfWrite

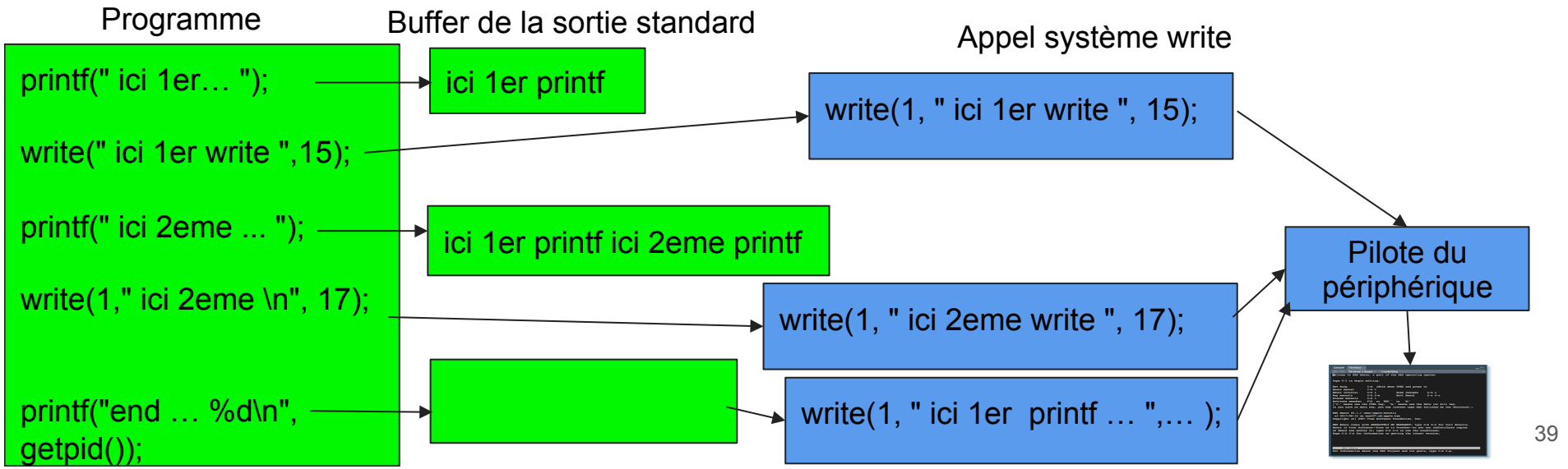
Expliquez l'ordre d'affichage du programme printfWrite.c :

ici 1er write ici 2eme write

ici 1er printf ici 2eme printf end of line printf de 3226

Exemple PrintfWrite

```
1.  // programme printfWrite.c
2.  int main( ) {
3.     printf(" ici 1er printf ");
4.     write(1," ici 1er write ",15);
5.     printf(" ici 2eme printf ");
6.     write(1," ici 2eme write \n", 17);
7.     printf("end of line printf de %d\n", getpid());
8.     return 0;
9.  }
```



printf avant fork

$ ./run.sh ProcessusEtThreads/PrintfFork

Exemple PrintfFork

```
1.  int main() {
2.      printf("bonjour je suis le processus principal de pid %d", getpid());
3.      int cpid = fork();
4.      if (!cpid) {
5.          printf("Ici le fils!\n");
6.          exit(0);
7.      }
8.      printf("Fin du programme\n");
9.      return 0;
10. }
```

```
$ ./run.sh ProcessusEtThreads/PrintfFork
gcc -o a.out printfFork.c
bonjour je suis le processus principal de pid 13Fin du programme
bonjour je suis le processus principal de pid 13Ici le fils!
```

Comment pourrait-on corriger le code pour que le fils n'affiche pas le message du père ?

Exemple PrintfFork

```
1.  // programme printfFork.c
2.  int main() {
3.      printf("bonjour je suis le processus principal de pid %d", getpid());
4.      fflush(stdout);
5.      int cpid = fork();
6.      if (!cpid) {
7.          printf("Ici le fils!\n");
8.          exit(0);
9.      }
10.     printf("Fin du programme\n");
11.     return 0;
12. }
```

```
$ ./run.sh ProcessusEtThreads/PrintfFork
gcc -o a.out printfFork.c
bonjour je suis le processus principal de pid 13Fin du programme
Ici le fils!
```

Exercices

Exercice 1

```
1.  // programme chp2Ex1.c dans Chp2Ex1
2.  //./run.sh ProcessusEtThreads/Chp2Ex1
3.  int main() {
4.      int i;
5.      int n = 2;
6.      pid_t fils_pid;
7.      for (i = 0; i < n; ++i) {
8.          fils_pid = fork();
9.          if (fils_pid > 0) {
10.             wait(NULL);
11.             break;
12.         }
13.     }
14.     printf("Processus %d de pere %d\n", getpid(), getppid());
15.     return 0;
16. }
```

Quel est le sous arbre des processus créé le programme ci-contre ?

Quels sont les ordres d'affichages possibles ?

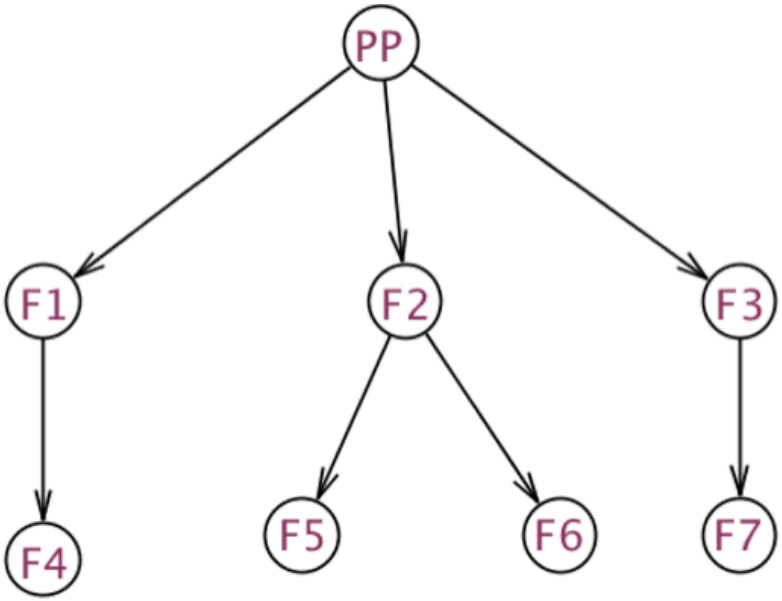
Exercice 2

```
1.  // programme chp2Ex2.c dans Chp2Ex2
2.  // ./run.sh ProcessusEtThreads/Chp2Ex2
3.  int main() {
4.      printf("message0\n");
5.      if (fork()) {
6.          printf("message1\n");
7.          if (fork()) {
8.              printf("message2\n");
9.          }
10.         else {
11.             exit(0);
12.         }
13.     }
14.     else {
15.         printf("message3\n");
16.     }
17.     exit (0);
18. }
```

```
$ ./run.sh ProcessusEtThreads/Chp2Ex2
gcc -o a.out chp2Ex2.c
message0
message1
message3
message2
$ ./run.sh ProcessusEtThreads/Chp2Ex2
gcc -o a.out chp2Ex2.c
message0
message1
message2
message3
$ ./run.sh ProcessusEtThreads/Chp2Ex2
gcc -o a.out chp2Ex2.c
message0
message3
message1
message2
```

- **Ajouter une ligne au programme suivant pour forcer l'ordre suivant d'affichage des messages :**
  message0, message3, message1, message2

Exercice 3



Écrivez le code nécessaire pour reproduire l'arbre de processus suivant :

- Chaque processus doit écrire à l'écran son pid ainsi que le pid de son parent

- Les processus F4 à F7 se transforment pour exécuter respectivement ls, pwd, cd ../ et ls.

Exercice 4

```
const int n=3, m=2;
int main(){
        int i,j=0;
        for(i=0; i<n&& j<m; i++)
        {       if(fork()==0)
                {   i=0;
                    j=j+1;
                }
        }
        printf(" j=%d\n",j);
        while(wait(NULL)>0);
        exit (0);
}
```

Donnez l'arborescence des processus créés par ce programme.

Donnez la valeur de j affichée par chacun des processus, y compris le processus principal. À quoi correspond-elle ?

Lecture suggérée

- **Chapitre 2.2 : Processus**

Introduction aux systèmes d'exploitation - Cours et exercices en GNU/Linux, Hanifa Boucheneb & Juan-Manuel Torres-Moreno, 216 pages, édition ellypses, 2019, ISBN : 9782340029651.

Initialisation de LINUX

- Le BIOS (Basic Input/Output System) vérifie l'intégralité du système et exécute le programme d'amorçage MBR (Master Boot Record).

- MBR charge une partie du système d'exploitation en mémoire principale et lance son exécution.

- Le processus 0 (MBR) lance d'autres initialisations (comme le système de fichiers). Il crée également le processus init (PID 1) et le démon des pages (PID 2). Les autres processus sont créés à partir du processus init.

- **Le démon des pages** a pour objectif de suivre l'état de la mémoire. Un démon est un processus qui s'exécute en arrière plan et qui s'active régulièrement pour réaliser un traitement.

https://www.mageialinux-online.org/wiki/le-processus-de-demarrage-d-un-systeme-linux