



# BUT 2 / R3.05

## PROGRAMMATION SYSTÈME

PROCESSUS



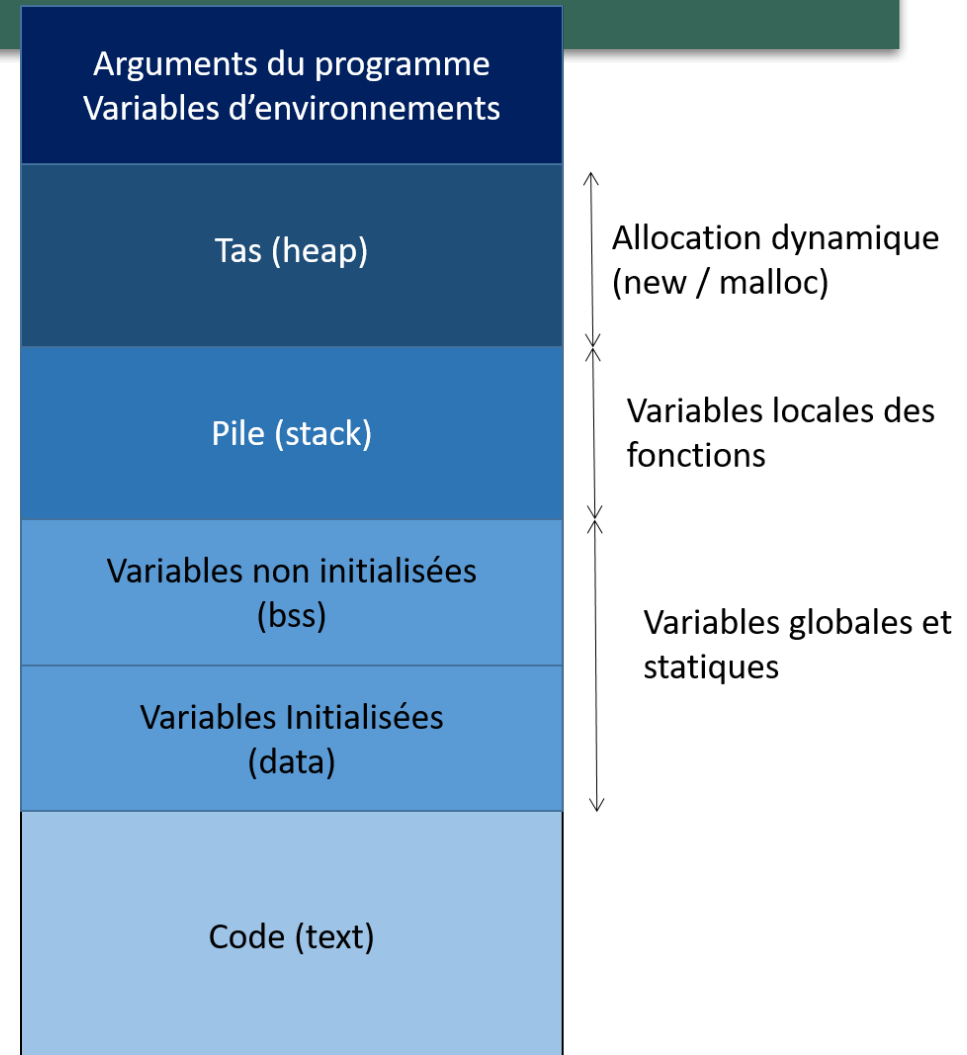
# APARTÉ

- Les appels systèmes (API système) sous Linux sont soit
  - SYSTEM V (5): implémentation historique du noyau Linux
  - POSIX: norme IEEE
- Sous windows
  - La plupart des fonctions POSIX sont supportées pour rendre les codes compatibles Linux/Windows
  - L'API système est la WIN-API (anciennement WIN-32)

# DEFINITION

- Un processus est un programme exécutable en cours d'exécution.
- Il utilise des ressources systèmes et matériels
  - Matériel
    - Entrée / sortie
    - Processeur
    - Mémoire
  - Systèmes
    - Appels systèmes (liés aux matériels ou au logiciel)
    - Gestion des processus par le SE

En mémoire :

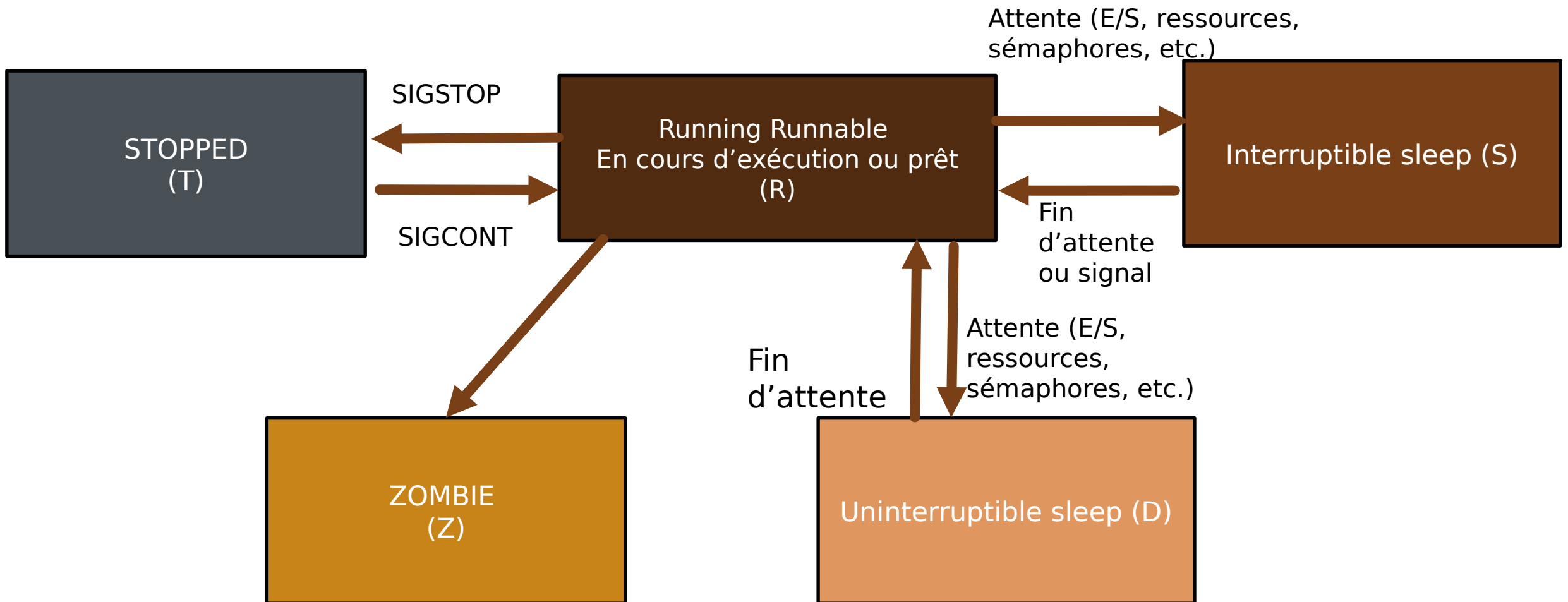




# PROCESSUS SOUS LINUX



# ETAT D'UN PROCESSUS SOUS LINUX



# INFORMATION NOYAU D'UN PROCESSUS

- Le système gère un certain nombre d'informations sur chaque processus

- PID, PPID, PGID, UID, GID

On peut accéder à certains d'entre eux grâce à des appels systèmes :

- `getpid()` → retourne le pid du processus en cours
- `getppid()` → retourne le pid du processus parent du processus en cours

- La table des descripteurs de fichiers
- Information sur les signaux, etc.
- Les informations sur son exécution: pile, registre, état/emplacement mémoire, etc.

# FILIATION SOUS LINUX

- Un processus est créé par un autre processus

- L'appel système est **fork()** ( ou clone() )

```
Prototype:  
pid_t fork(void);
```

- Si l'appel réussi, une copie du processus est créée dès l'exécution de ce fork

- Cela a pour conséquence que le code qui suit est exécuté par deux processus en « parallèle » :
      - } Le processus initial (père)
      - } Le nouveau processus (fils)

- fork() renvoie **deux valeurs différentes** selon le processus dans lequel on se retrouve !

→ -1 si l'appel a échoué (problème de ressources), ou :

- 0 dans le processus fils
    - Le pid du fils dans le père

- Cela crée une filiation père/fils (parent/child)

# En mémoire...

➤ Lors d'un fork, tout l'espace mémoire du processus est copié :

- Variables statiques/globales
- pile/tas ( → donc toutes les variables définies )
- Descripteurs de fichier (nous verrons plus tard)
- Code à exécuter
- ...

Arguments du programme  
Variables d'environnements

Tas (heap)

Pile (stack)

Variables non initialisées  
(bss)

Variables Initialisées  
(data)

Code (text)



# Exemple

```
int main() {  
  
    int x = 100 ;  
    int retourFork ;  
  
    printf("Coucou\n") ;  
  
    retourFork = fork() ;  
  
    if (retourFork > 0) {  
        x++ ;  
        printf("je suis le père, x=%d\n", x) ;  
    } else {  
        x-- ;  
        printf("je suis le fils, x=%d\n", x) ;  
    }  
    printf("exécuté par les deux ! X=%d\n", x) ;  
  
    return 0 ;  
}
```

# Exercice

Qu'affiche le programme suivant ?

```
int main() {  
  
    int x = 100 ;  
  
    if ( fork() > 0 ) {  
  
        for (int i = 0 ; i < 100 ; i++){  
            x ++ ;  
            printf("x = %d", x) ;  
        }  
  
    } else {  
  
        for (int i = 0 ; i < 100 ; i++){  
            x -- ;  
            printf("x = %d", x) ;  
        }  
  
    }  
  
}
```

# FILIATION SOUS LINUX

- La valeur retournée par le fils est à destination du processus père.
- Le père peut la récupérer avec l'appel système `wait()` ou `waitpid()`.

# ATTENTE DE LA TERMINAISON CHEZ LE PÈRE

■ 2 choix :

**pid\_t wait(int \*wstatus);**

- Appel bloquant. Débloqué lorsque n'importe quel fils se termine
- Récupère dans wstatus ce qu'a retourné le main du fils
- Retourne le pid du fils en cas de succès, et -1 sinon (ex : aucun fils à attendre)

**pid\_t waitpid(pid\_t pid, int \*wstatus, int options);**

- Appel bloquant. Débloqué lorsque le fil de pid **pid** est terminé
- Récupère dans wstatus ce qu'a retourné le main du fils
- Options :
  - WNOHANG : l'appel n'est pas bloquant, et retourne 0 s'il n'y a pas de fils terminé

# ATTENTE DE LA TERMINAISON CHEZ LE PÈRE

## ■ **wstatus**

- Ce que retourne la fonction `main` est codé sur 32 bits
- La « vraie » valeur qui suit le « `return` » ne remplit que les 8 derniers bits de cette valeur
- Les autres bits contiennent d'autres informations
- Pour ne récupérer que ces 8 bits : fonction **`WEXITSTATUS(wstatus)`**

# ATTENTE DE LA TERMINAISON CHEZ LE PÈRE

```
int main()
{
    pid_t retourFork; int retourFils;

    retourFork=fork();
    switch(retourFork)
    {
        case -1: perror(" Error fork() ");
                break;
        case 0: printf(" Je suis le fils\n ");
                return(24);
        default: printf(" Je suis le père\n ");
                 wait(&retourFils);
                 printf(" La valeur est %d\n ",WEXITSTATUS(retourFils));
                 break;
    }
    return(0);
}
```

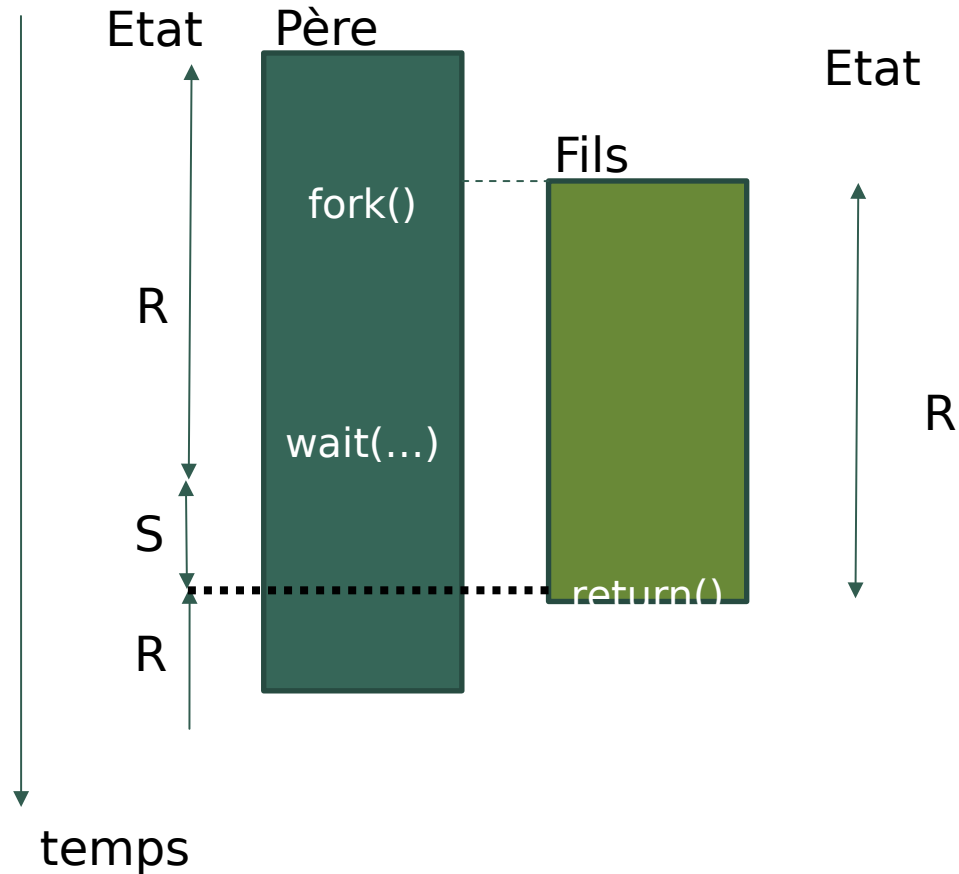
Création d'un nouveau processus.  
**Deux processus exécutent en parallèle le code qui suit.**

Le père attend jusqu'à la terminaison du fils.

Le code de retour est chiffré sur le dernier octet. Les octets de poids forts contiennent d'autres informations sur l'état du processus fils.

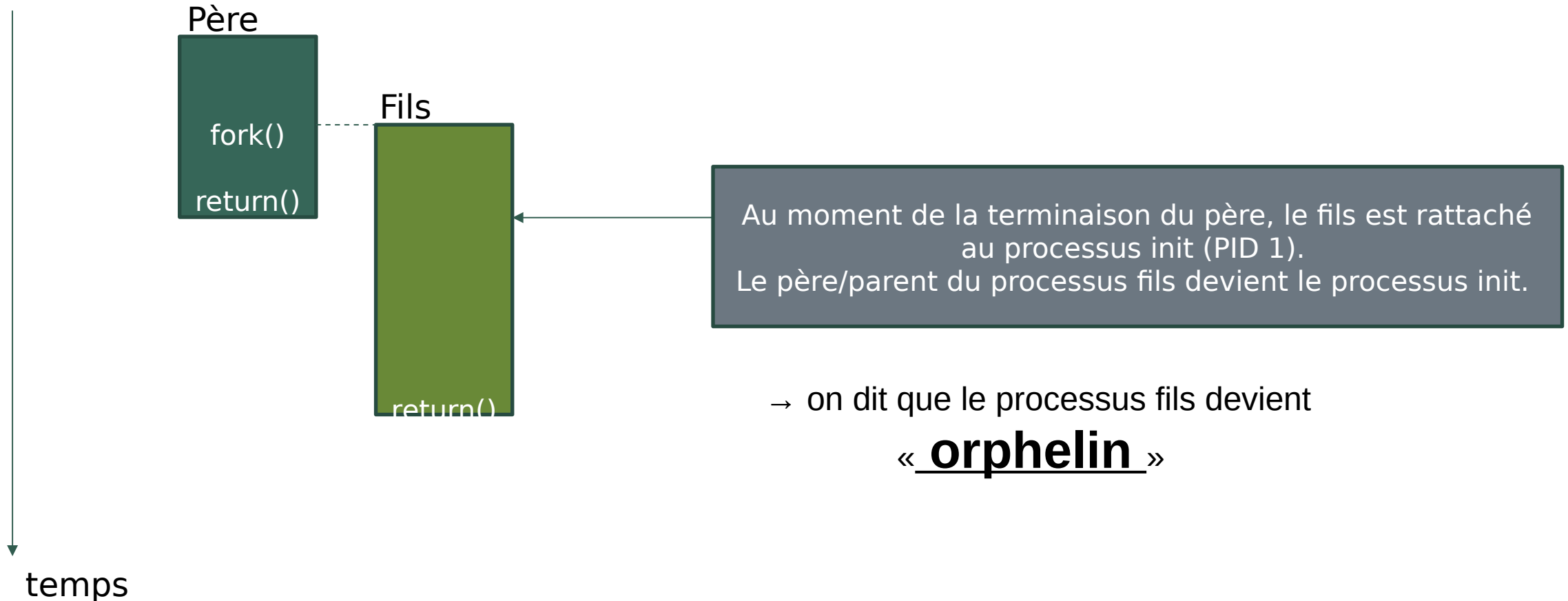
# FILIATION SOUS LINUX: SCENARIO PÈRE/FILS

- Cas 1: le père effectue son wait() avant la terminaison du fils



# FILIATION SOUS LINUX: SCENARIO PÈRE/FILS (2)

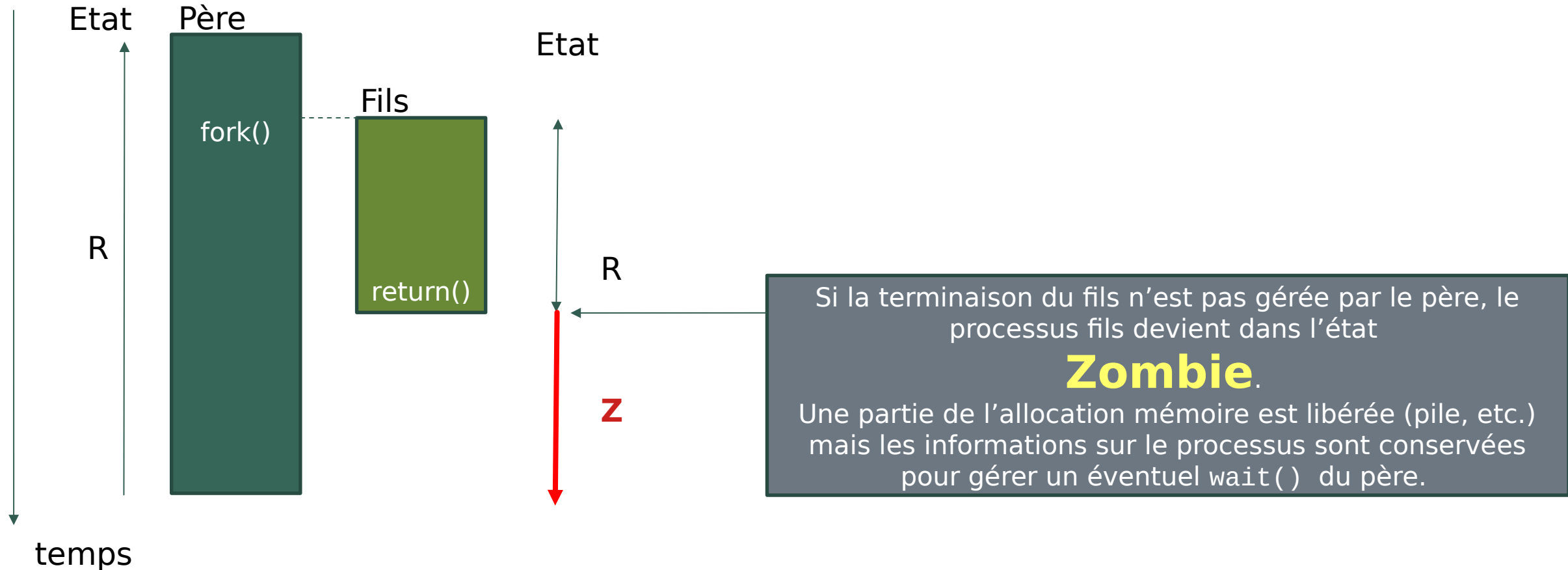
- Cas 2: le père se termine avant son fils





# FILIATION SOUS LINUX: SCENARIO PÈRE/FILS (3)

- Cas 3: le père s'exécute sans s'occuper de la terminaison du fils




# QUESTIONS

- Que retourne la fonction fork ?


- Quelle est la sortie du code ci-contre

- Toto
- Toto Toto
- Toto Toto Toto
- Toto Toto Toto Toto
- Aucun affichage



```
int main()
{
    fork(); fork();
    printf(" Toto ");
    return(0);
}
```

- Y-a-t-il un zombie avec le code ci-contre?



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

# ANALYSE...

```
int main()
{
    int i, retourExit, retourFork=1, pid;

    printf("Le shell a pour PID %d\n",getppid());
    for(i=0; i < 3; i++)
    {
        retourFork=fork();
        printf("Mon PID est %d mon Pere est %d et i=%d\n",

                getpid(),getppid(),i);

    }

    while( (pid=wait(&retourExit))>0 )
        printf("Code de retour du fils %d: %d\n",pid,WEXITSTATUS(retourExit));

    exit(i);
}
```

# Exercice

## Exercice 1

Écrivez un programme `zombie` qui crée un processus qui reste zombie pendant 30 secondes.

Pour cela, il faut d'abord effectuer un `fork`. Puis, le père et le fils devront exécuter deux codes différents :

- le père se met en sommeil pendant 30 secondes (grâce à la fonction `sleep`)
- le fils se termine (avec un `return`)

Pour obtenir la liste des processus en mode zombie ("`defunct`" sous Linux) en cours d'"exécution", vous pouvez par exemple exécuter la commande suivante (depuis un autre terminal) :

```
ps -e | grep "defunct"
```

# Exercice

## Exercice 2

Écrivez un programme `orphelin` qui crée un processus qui devient orphelin. Votre programme devra montrer que le processus fils perd effectivement à un moment donné son processus père, et afficher quel est son nouveau père.

Pour cela, il faut d'abord effectuer un `fork`. Puis, le père et le fils devront exécuter deux codes différents :

- le père affiche son process id (avec la fonction `getpid()`), se met en sommeil 5 secondes, puis se termine
- le fils affiche son parent id (avec la fonction `getppid()`), se met en sommeil 10 secondes, puis affiche à nouveau son parent id

# Exécuter un autre code

Dupliquer un processus, c'est très bien mais ça ne fait pas démarrer d'autres programmes...

```
int execvp(const char *file, char *const argv[]);
```

- C'est en fait une famille de fonctions (voir le man)
- Permet de changer le code du programme en cours d'exécution par celui d'un autre programme
  - **file** est le nom de l'exécutable à lancer
  - **argv** est la liste des arguments à lui passer
    - Tableau de chaînes de caractères
    - La première case reprend le nom de l'exécutable
    - La dernière case doit être NULL

# Exécuter un autre code

Dupliquer un processus, c'est très bien mais ça ne fait pas démarrer d'autres programmes...

**`int execvp(const char *file, char *const argv[]);`**

- Retourne -1 en cas d'échec
- En cas de succès, on ne saura jamais ce qu'elle retourne, puisque le processus en cours ne s'exécute plus !
- Tout le code qui suit n'est donc exécuté que si l'appel n'a pas fonctionné !

# Exécuter un autre code

Lancez la commande « `ls -l` » grâce à `execvp` !