Module R3.05

Ressource R3.05 — **Programmation système**

Informatique > Systèmes communicants en réseau > Prog. système

Descriptif détaillé

Objectif

L'objectif de cette ressource est de comprendre la structure d'une application client-serveur et de comprendre les mécanismes bas niveaux, mis en œuvre dans une application multitâches. Cette ressource permettra de découvrir le développement d'applications multi-processus, de comprendre et de traiter les problèmes de synchronisation et d'utiliser des outils de communication internes aux processus , mais aussi externes, via les interface de programmation (API) de transport.

Savoirs de référence étudiés

- Fonctionnement du système (par ex. : pagination, mémoire virtuelle, systèmes de fichiers…)
- Gestion de processus (par ex. : ordonnancement, synchronisation, threads…)
- Programmation client-serveur (par ex. : inter-process communication (IPC), interface socket, protocoles applicatifs…)
- Les différents savoirs de référence pourront être approfondis

Mécanismes bas niveaux | Processus | Client-serveur

Cursus	S3 tous parcours
Heures totales (30h)	**15h** TD et **15h** TP
programme national	10h TD et 10h TP
adaptation locale SAÉ	3h TD et 3h TP
adaptation locale non fléchée	2h TD et 2h TP
Exemple de contribution aux SAÉ	
S3.A.01 Développement appli	3h TD et 3h TP

Coefficients de pondération

UE	*Parcours*	Coeff.
UE 3.3	*parcours A*	22%
	parcours B	22%
	parcours C	22%
	parcours D	22%

Programmation Système

Les Processus sous Linux

Programmation système

I/Rappels et vocabulaire:

- ***Multitâche :***

Un système d'exploitation est dit multitâche (*multitasking*) **s'il est capable d'exécuter de façon apparemment simultanée plusieurs programmes informatiques.**

Le passage de l'exécution d'un programme à celle d'un autre est appelée ***Commutation de contexte.***

multitâche coopératif: commutation initiée par les programmes eux-mêmes

multitâche préemptif: commutation initiée par le système d'exploitation

Le multitâche n'est pas dépendant du nombre de processeurs présents physiquement dans la machine.

Programmation système

- ***Ordonnanceur :***

désigne le composant du noyau du système d'exploitation dont le rôle est de distribuer le temps du processeur entre les différents processus.

- ***Préemption :***

est la capacité d'un système d'exploitation à suspendre l'exécution d'un programme au profit de celle d'un autre.

- ***Le quantum :***

est la quantité de temps définie attribuée par l'ordonnanceur.

Dans un ordonnancement (statique à base de priorités) avec préemption, un processus peut être préempté (remplacé) par n'importe quel processus plus prioritaire qui serait devenu prêt.

Programmation système

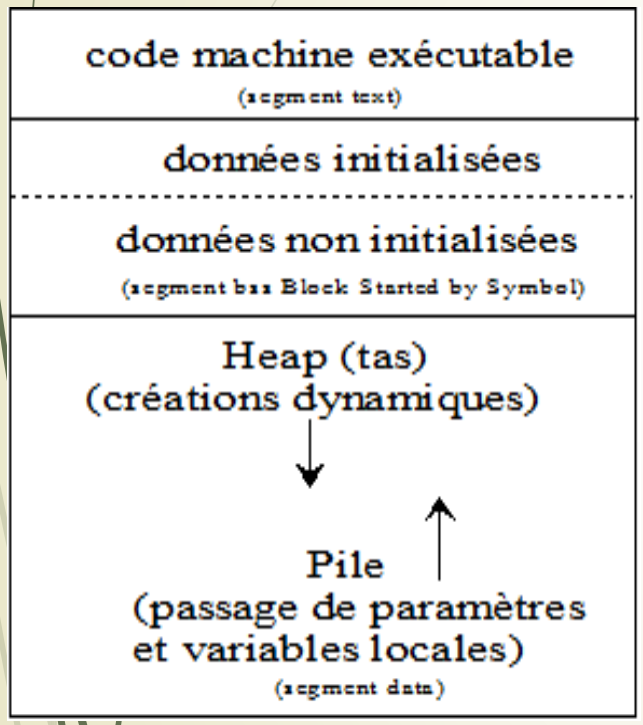
II/ Les processus

On appelle processus **une instance de programme**. C'est-à-dire **c'est un programme en cours d'exécution** par un système d'exploitation.

Un processus s'exécute dans son contexte, quand il y a changement de processus courant il y a **commutation ou changement de contexte**. Le noyau s'exécute alors dans le nouveau contexte.

Programmation système

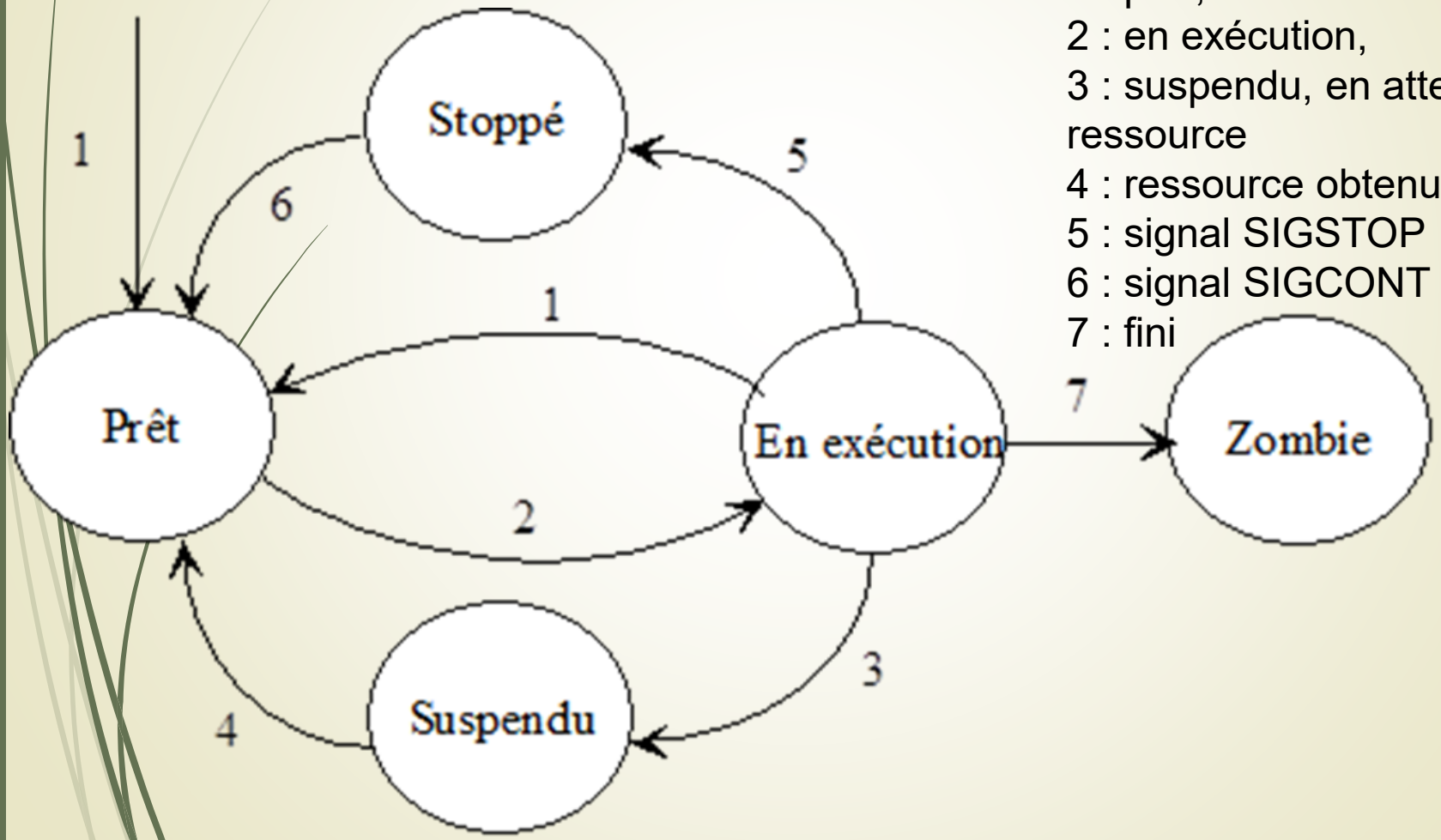
Un processus ne se résume pas au programme qu'il exécute.




```
int Globale1; /*donnée non initialisée */
int Globale2 = 1234; /*donnée initialisée */
static int Globale3; /*donnée non initialisée */
        /* … */
 int ma_fonction(paramètres)
{
    int Locale1;     /*pile d'exécution */
    static int Locale2 = 4321; /*donnée initialisée (static) */
    static int Locale3; /*donnée non initialisée (static)*/
            /*… */
}
```

Programmation système

Etats d'un processus



1 : prêt,
2 : en exécution,
3 : suspendu, en attente de ressource
4 : ressource obtenue,
5 : signal SIGSTOP
6 : signal SIGCONT
7 : fini

Programmation système

Caractéristiques d'un processus:

Le contexte d'un processus *(Process Control Block)* est l'ensemble des informations :

- Son état
- Son identificateur et celui de son père
- L'identité des utilisateurs (réel et effectif) des groupes (réel et effectif)
- Des informations utilisées par le noyau pour procéder à l'ordonnancement (priorité, l'événement attendu …),
- Des informations concernant l'espace d'adressage
- Des informations concernant les entrées/sorties effectuées par le processus (descripteur de fichiers ouverts, répertoire courant …)
- Son entrée dans la table des processus
- Ses données privées
- Les piles *user et système*
- Les zones de code et de données

Développement Système

Informations d'un zombie

Un processus zombie n'a plus d'existence en mémoire.

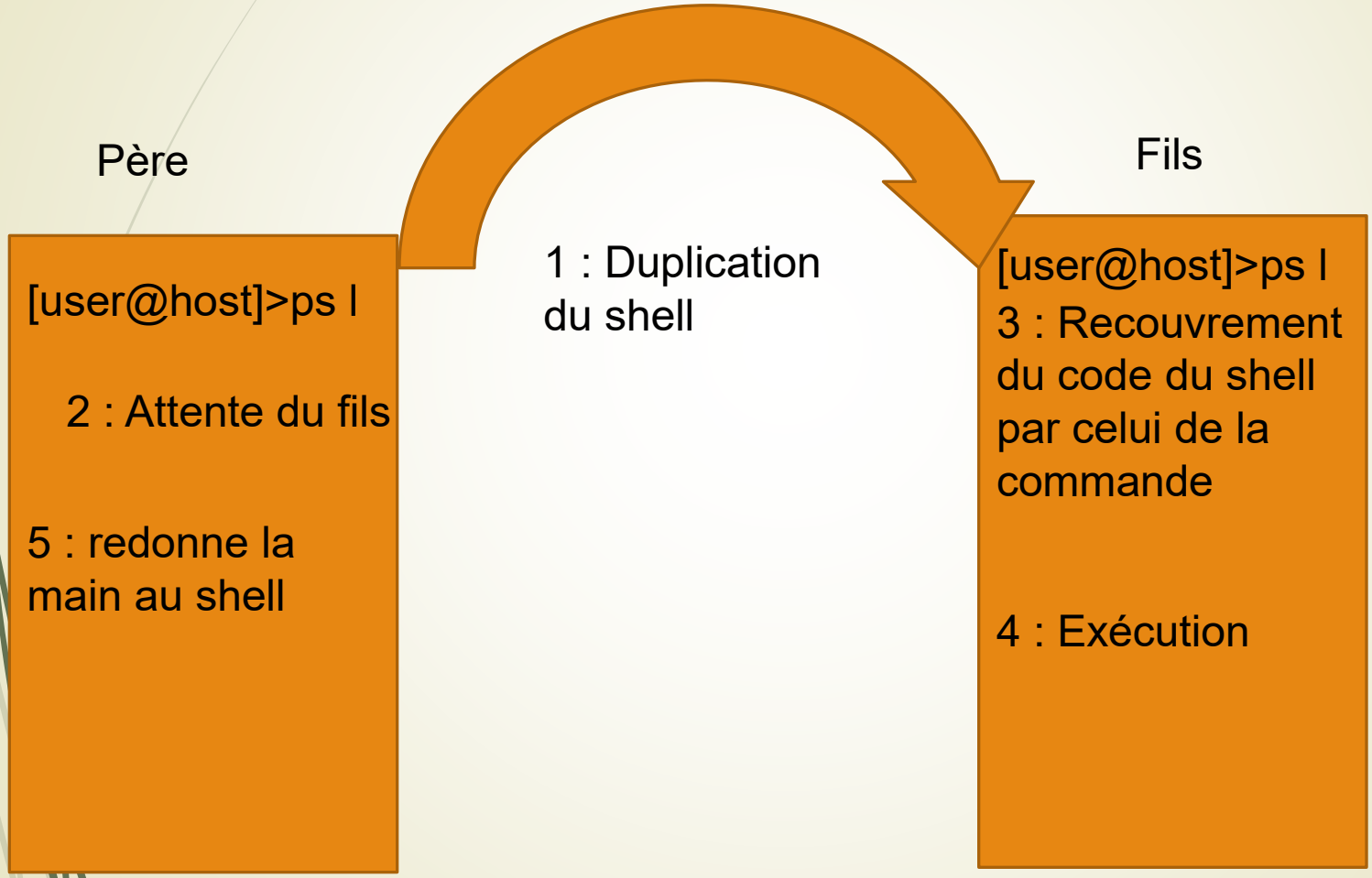
Le système garde des informations sur ce processus:

- Son PID (Processus IDentifier),
- Son PPID (Parent Processus IDentifier),
- Sa valeur de retour,
- Ses temps d'exécution (mode utilisateur et mode noyau).

Les informations sont conservées jusqu'à ce que le père prenne connaissance de la valeur de retour du fils ou se termine lui même.

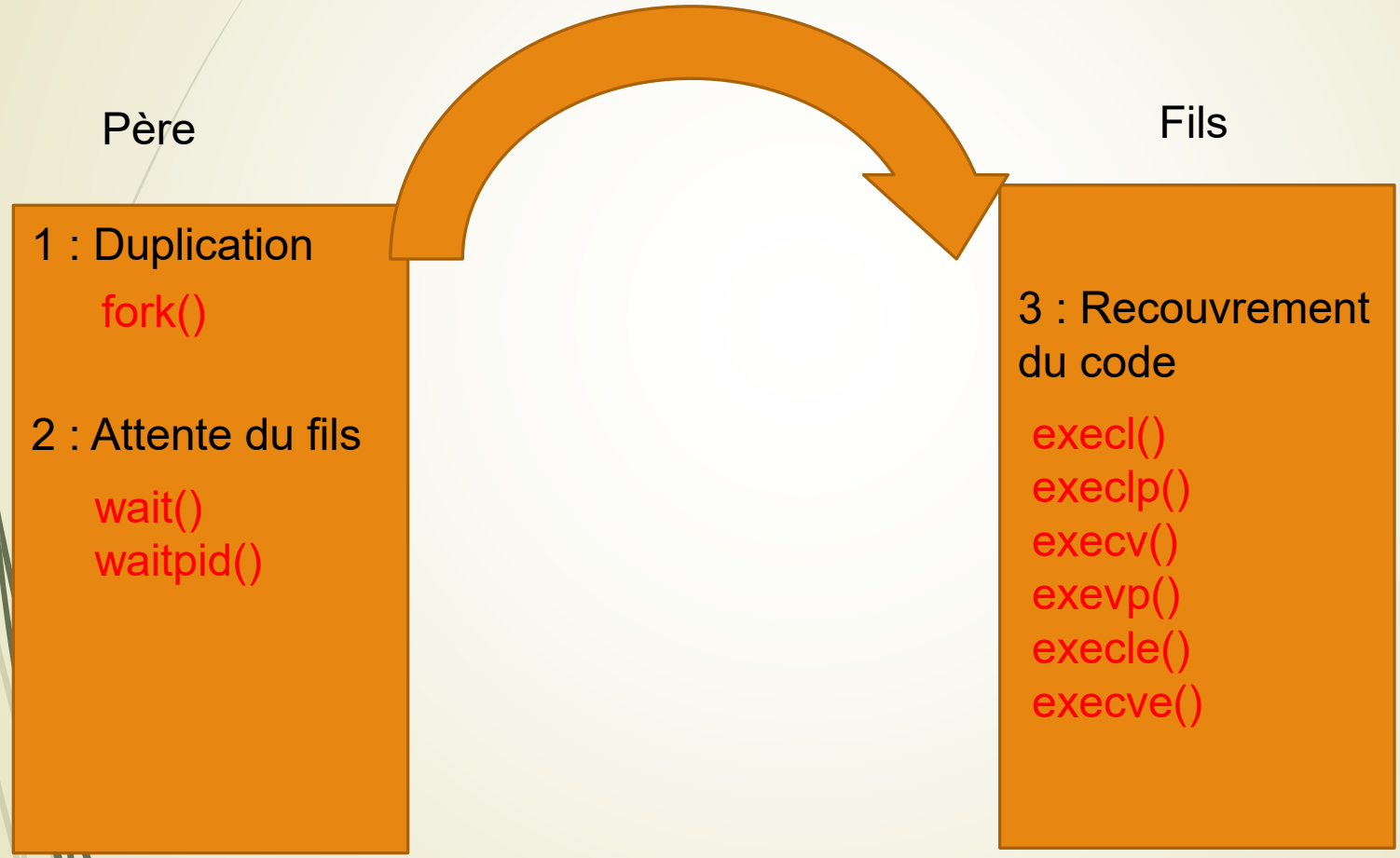
Programmation système

Que se passe-t-il lorsque vous exécutez une commande sous le shell ?



Programmation système

Comment réaliser ceci en C ?



Programmation système

Primitive de création d'un fils

- **pid_t fork(void);**

- **Valeur renvoyée** :

  En cas de succès, le PID du fils est renvoyé au processus père, et 0 est renvoyé au processus fils.

  En cas d'échec -1 est renvoyé dans le contexte du père, aucun processus fils n'est créé.

Programmation système

Un exemple simple

```
int main(int argc, char *argv[])
{
        pid_t pid;
        pid=fork();
        if (pid > 0)
        {
                printf("Je suis le pere : %d (de pere %d)\n",getpid(),getppid());
        }
        else
        {
                if (pid == 0)
                {
                        printf("\t\t\tJe suis le fils : %d (de pere %d)\n",getpid(),getppid());
                        return EXIT_SUCCESS;
                }
                perror("fork");
                return EXIT_FAILURE;
        }
        return EXIT_SUCCESS;
}
```

Je suis le fils : 20159 (de pere 20158)

Je suis le pere : 20158 (de pere 20038)

Les deux affichages ont bien lieu ce qui confirme qu'il y a bien deux processus. L'ordre d'affichage est quelconque et imposé par l'ordonnanceur du système. Les processus sont concurrents.

```
int main(int argc, char *argv[])
{
pid_t pid;
        pid=fork();
        if (pid > 0)
        {
                printf("Je suis le pere : %d (de
pere %d)\n",getpid(),getppid());
        }
        else
        {
                if (pid == 0)
                {
                        printf("\t\t\tJe suis le fils :
%d (de pere %d)\n",getpid(),getppid());
                        return EXIT_SUCCESS;
                }
                perror("fork");
                return EXIT_FAILURE;
        }
        return EXIT_SUCCESS;
}
```

Père : pid > 0

```
int main(int argc, char *argv[])
{
pid_t pid;
        pid=fork();
        if (pid > 0)
        {
                printf("Je suis le pere : %d (de
pere %d)\n",getpid(),getppid());
        }
        else
        {
                if (pid == 0)
                {
                        printf("\t\t\tJe suis le fils :
%d (de pere %d)\n",getpid(),getppid());
                        return EXIT_SUCCESS;
                }
                perror("fork");
                return EXIT_FAILURE;
        }
        return EXIT_SUCCESS;
}
```

Fils : pid = 0

Programmation système

Second exemple : état zombie: (le fils finit avant le père)

```
int main(int argc, char *argv[])
{
pid_t pid;
        switch(pid=fork())
        {
         case -1 : printf("Erreur de creation du processus fils\n");return EXIT_FAILURE;
        case 0 :        /* processus fils */
                printf("\t\t\tJe suis le fils : %d (de pere %d)\n",getpid(),getppid());
                return EXIT_SUCCESS;
        default :       /* processus pere */
                system("ps l");
                printf("Je suis le pere : %d (de pere %d)\n",getpid(),getppid());
        }
        return EXIT_SUCCESS;
}
```

Programmation système

Adoption du fils par le : (le père finit avant le fils)

```
int main(int argc, char *argv[])
{
pid_t pid;
        switch(pid=fork())
        {
            case -1 : perror("fork");return EXIT_FAILURE;
            case 0 :  /* processus fils */
                system("ps l");
                printf("\t\t\tJe suis le fils : %d (de pere %d)\n",getpid(),getppid());
                return EXIT_SUCCESS;
            default :          /* processus pere */
                printf("Je suis le pere : %d (de pere %d)\n",getpid(),getppid());
        }
        return EXIT_SUCCESS;
}
```

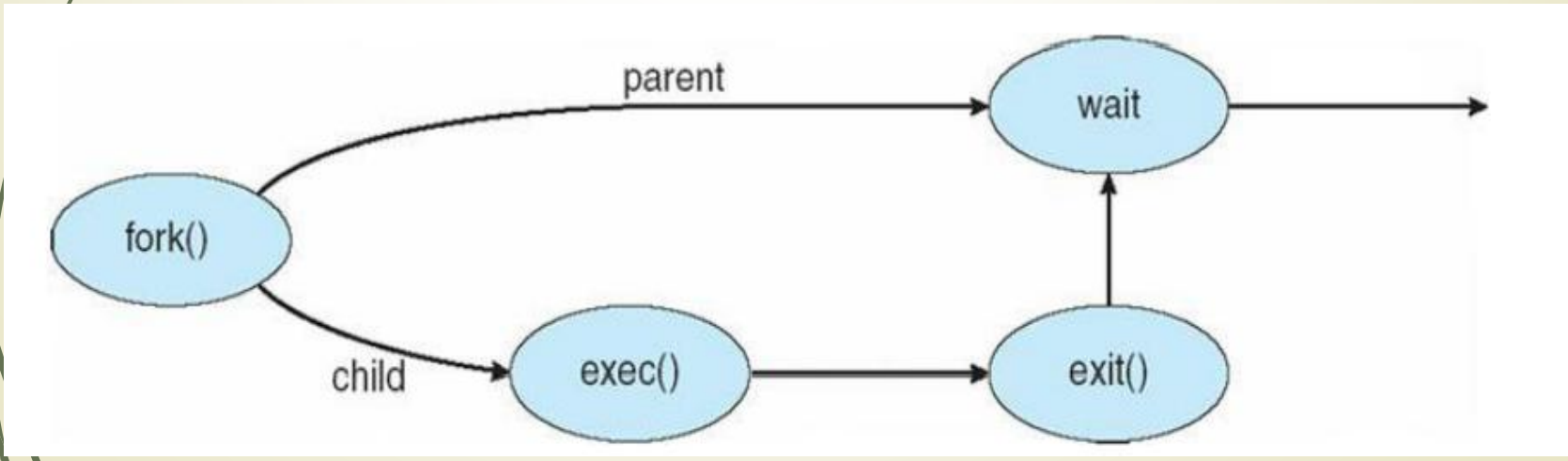
le processus fils ayant perdu son père est automatiquement adopté par le processus INIT de PID 1

Programmation système

Les processus créés par fork s'exécutent de façon concurrente avec leur père. On ne peut présumer l'ordre d'exécution de ces processus. C'est l'ordonnanceur qui attribue le processeur selon un algorithme donné.

Il est donc impossible de savoir quels processus se terminent avant tels autres, même pas entre père et fils.
Il existe donc dans certains cas un problème de synchronisation entre processus.

Les primitives **wait** permettent de résoudre ce problème en provoquant la suspension du processus appelant jusqu'à ce que l'un de ses processus fils se termine.



Programmation système

Le père attend son fils : wait

```
int main(int argc, char *argv[])
{
pid_t pid;
        switch(pid=fork())
        {
                case -1 : perror("fork");return EXIT_FAILURE;
                case 0 :        /* processus fils */
                        system("ps l");
                        printf("\t\t\tJe suis le fils : %d (de pere %d)\n",getpid(),getppid());
                        return EXIT_SUCCESS;
                default :
                        /* processus pere */
                        printf("Je suis le pere : %d (de pere %d)\n",getpid(),getppid());
        }
        if (wait(NULL) == -1)
        {
                perror("wait");
                return EXIT_FAILURE;
        }
        printf("Le fils vient de se terminer\n");
        return EXIT_SUCCESS;
}
```

Programmation système

wait – waitpid

pid_t wait(int **status*)

pid_t waitpid(pid_t *pid*, **int ****status*, **int** *options*);

La valeur de *pid* peut être l'une des suivantes :

- -1 attendre la fin de n'importe quel fils. C'est le même comportement que **wait**,
- > 0 attendre la fin du processus numéro *pid*.

La valeur de l'argument *options* est un OU binaire entre les constantes suivantes :

- **WNOHANG** ne pas bloquer si aucun fils ne s'est terminé.
- **WUNTRACED** recevoir l'information concernant également les fils bloqués si on ne l'a pas encore reçue.

Programmation système

wait – waitpid (suite)

pid_t wait(int *****status*)**

pid_t waitpid(pid_t *pid*, **int *****status*, **int** *options*);

Si *status* est non **NULL**, **wait** et **waitpid** y stockent l'information sur la terminaison du fils.

Cette information peut être analysée avec les macros suivantes :

- **WIFEXITED(***status***)** non nul si le fils s'est terminé normalement,
- **WEXITSTATUS(***status***)** donne la valeur de retour. Cette macro ne peut être évaluée que si **WIFEXITED** est non nul.
- **WIFSIGNALED(***status***)** indique que le fils s'est terminé à cause d'un signal non intercepté.
- **WTERMSIG(***status***)** donne le numéro du signal qui a causé la fin du fils. Cette macro ne peut être évaluée que si **WIFSIGNALED** est non nul.
- **WIFSTOPPED(***status***)** indique que le fils est actuellement stoppé. Cette macro n'a de sens que si l'on a effectué l'appel avec l'option **WUNTRACED.**
- **WSTOPSIG(***status***)** donne le numéro du signal qui a causé l'arrêt du fils. Cette macro ne peut être évaluée que si **WIFSTOPPED** est non nul.

Programmation système

Récupérer la valeur de retour du fils

```
int main(int argc, char *argv[])
{
pid_t pid;
int retour;
        switch(pid=fork())
        {
                case -1 : perror("fork");return EXIT_FAILURE;
                case 0 :  printf("\t\t\tJe suis le fils : %d (de pere %d)\n",getpid(),getppid());
                        return 49;
                default : printf("Je suis le pere : %d (de pere %d)\n",getpid(),getppid());
        }
        if (wait(&retour) == -1)
        {
                perror("wait");
                return EXIT_FAILURE;
        }
        if(WIFEXITED(retour))
        {
              printf("Le fils vient de se terminer en retournant la valeur : %d\n",WEXITSTATUS(retour));
        }
        return EXIT_SUCCESS;
}
```

Programmation système

Le recouvrement:

Faisons exécuter une archive par un processus fils mais en utilisant la commande tar :

tar zcvf archive.tar.gz /home/user1/Processus

Quelle est la commande ?

Où est-elle située ?

Combien y a-t-il d'arguments ?

Quels sont les arguments ?

Programmation système

Recouvrement (les fonctions)

- **int execl (const char *path, const char *arg, ...);**
- **int execlp (const char *file, const char *arg, ...);**
- **int execv (const char *path, char *const** *argv**[]);**
- **int execvp (const char *file, char *const** *argv**[]);**
- **int execle (const char *path, const char *arg , ..., char * const** *env**[]);**
- **int execve (const char *** *fichier*, **char * const** *argv* **[], char * const** *env***[]);**

Programmation système

```
int main(int argc, char* argv[])
{
	switch (fork())
	{
	case -1: perror("fork"); break;
	case 0: /*     fils      */
		execl("/bin/tar", "tar", "zcvf", "archive.tar.gz", "/home/user1/Processus/", NULL);
		break;
	default:/*pere*/
		wait(NULL);
	}
	return EXIT_SUCCESS;
}
```

*Le premier paramètre de la primitive **execl** est le chemin complet de l'exécutable, le second paramètre est le nom de la commande. Ensuite on trouve les paramètres passés à cette commande. La fin des paramètres est signalée par un pointeur NULL.*