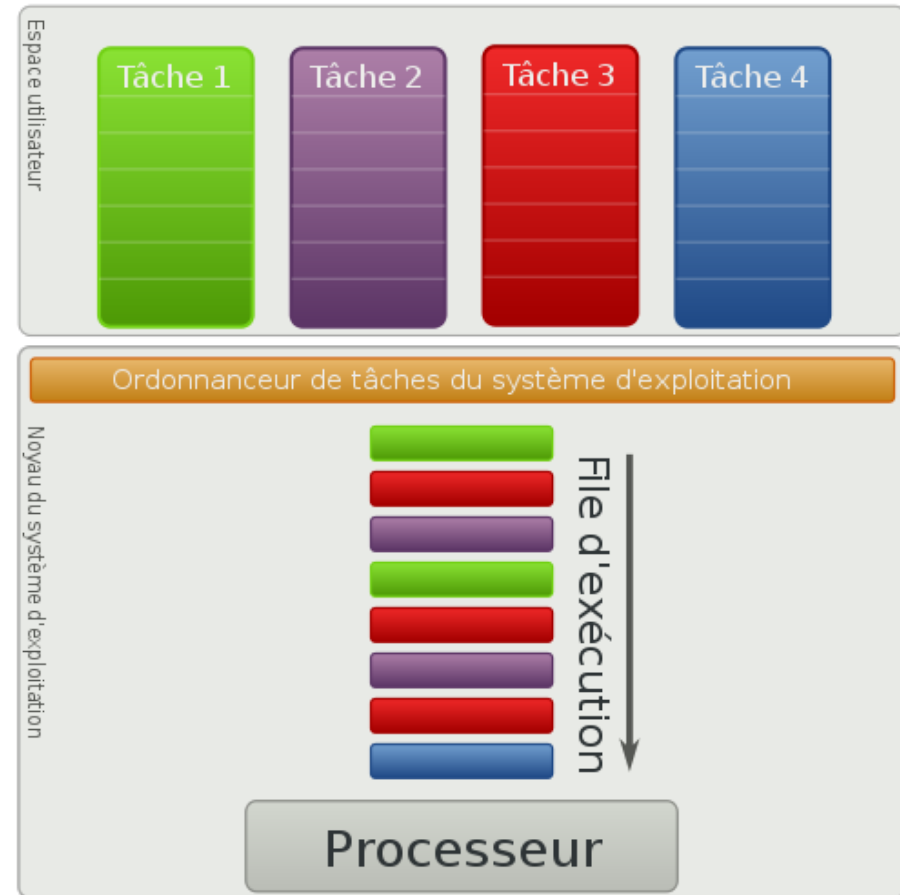


Processus

Processeur central: opération séquentielle, exécute une instruction à la fois (éventuellement avec des interruptions)

Processus: structure dans le système représentant une “unité d’exécution”; le système lui assure l’allocation de temps et non-interférence par d’autres processus



C'est quoi un processus ?

Chaque processus possède un **identifiant** numérique (*pid*). La liste des processus actuels du système peut être obtenue par `ps`, `ps tree` ou `top`.

Activité séquentielle poursuivi par l'ordinateur ; un processus possède son code, ses données et certaines attributs de plus.

Ne pas confondre avec un *programme* : un même programme peut être instancié par plusieurs processus en même temps.

Exemples: le processus "init", quelques programmes de service (démons), ligne de commande, éditeur de texte, ...

Pourquoi des processus ?

Facilitation de la programmation : on peut écrire ses programmes dans un style séquentiel comme si c'était la seule tâche du système, le système se chargera de gérer les différentes activités en même temps.

Les processus sont isolés l'un de l'autre : il ne peuvent pas s'espionner, une faute dans un processus ne gêne pas l'opération des autres.

Les processus peuvent interagir entre eux par des interfaces bien définies.

Exécution d'une commande dans le shell (grosso modo)

Lançons une commande simple dans le shell, p.ex. `echo bonjour`.
Qu'est-ce qui se passe ?

Le shell cherche un programme du nom `echo`, en utilisant la variable **PATH**.

Si un tel programme est trouvé, un nouveau processus est créé qui exécute le programme avec les paramètres données. Ces paramètres correspondent aux arguments de `main` dans C: `int main (int argc, char **argv)`

Le shell attend la fin du processus avant de continuer.

Hierarchie et création de processus

En POSIX, le seul moyen pour créer un processus est avec `fork`. (`fork.c`)

`fork` crée une copie du processus qui fait l'appel.

“Père” = processus original, “fils” = copie

La fonction renvoie l'identifiant du fils au père et 0 au fils. (-1 en cas d'échec)

Le fils peut obtenir l'identifiant de son père par `getppid` et son propre identifiant par `getpid`.

Du coup, les processus sont organisés dans une arborescence que l'on peut voir avec `pstree`.

fork et mémoire

Après l'appel `fork` nous avons deux processus identiques sauf deux aspects :

- leur identifiant (pid) (et l'identifiant parent)

- la valeur renvoyée par `fork`

Les mémoires des deux processus seront identiques au départ mais indépendantes ; ensuite, toute modification de mémoire n'affecte que le processus qui l'effectue.

Attributs d'un processus

Identifiant et identifiant du père

Contexte (compteur, registres)

Mémoire : code, données, pile

État (actif, en attente, bloqué, ...)

Environnement (des variables comme PATH etc)

etc, on en verra d'autres ...

Vie et mort d'un processus

Un processus se *termine* par l'appel `exit`.

(Dans C, si `main` fait un `return`, cela se traduit en un appel d'`exit`.)

Ce mécanisme permet au processus de renvoyer un **code de sortie**.

Le père peut appeler `wait` pour attendre la terminaison de l'un de ses fils.

(**wait.c**)

`wait` renvoie également le code de sortie du fils et certaines informations sur les circonstances de sa terminaison.

Exec

Il existe toute une famille de fonctions (`exec` etc) qui permettent de transformer un processus en remplaçant son code.

Exemple (on lance une commande dans le shell) : (`exec.c`)

1. L'utilisateur lance la commande.
2. Le shell fait appel à `fork`.
3. Le processus fils fait appel à `execvp` (p.ex.) pour devenir le programme souhaité par l'utilisateur.
4. Le shell attend la terminaison du fils, puis accepte d'autres commandes.

Combiner plusieurs commandes dans le shell

`cmd1 ; cmd2`: exécute d'abord `cmd1` puis `cmd2`

le shell crée un premier fils, attend sa terminaison, puis lance un deuxième fils et attend sa terminaison.

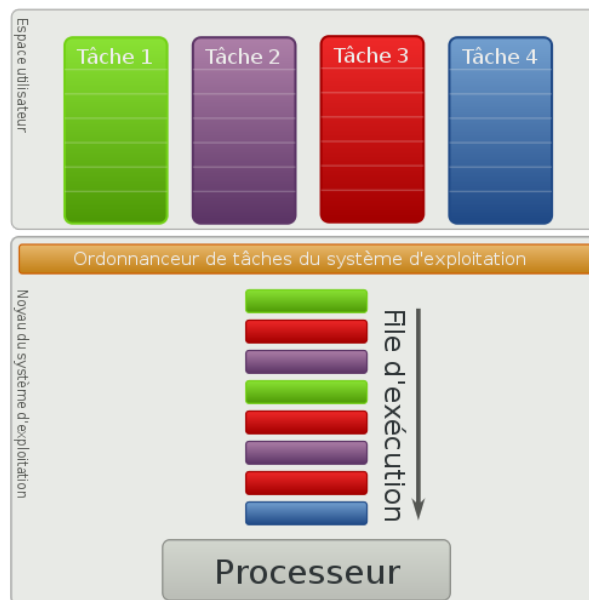
`cmd1 || cmd2`: exécuter `cmd1`, et si le code de sortie est non-zéro, exécuter `cmd2`

`cmd1 && cmd2`: exécuter `cmd1`, et si le code de sortie est zéro, exécuter `cmd2`

même principe, mais on regarde le code de sortie renvoyé par `wait` avant d'éventuellement lancer le deuxième fils.

État d'un processus

En raison de l'*ordonnancement* un processus est dans un de deux états principaux :



Actif: il est actuellement exécuté par le processeur.

Passif: le processus est en vie sans être exécuté.

Certains processus sont passifs pour une raison “logique”, p.ex.:

ils attendent une donnée (ou la terminaison d'un fils);

ils se sont endormis (par `sleep`), etc

Pour permettre l'ordonnancer à éviter ces processus, on distingue deux types de processus passifs :

Bloqué: le processus attend quelque chose, il ne sera pas ordonnancé pour l'instant.

Prêt: le processus peut continuer mais n'est pas actuellement ordonnancé.

États spéciaux

Unix connaît certains états spéciaux dont le plus important pour nous est :

Zombie: le processus a terminé mais pas son créneau existe toujours. Son père doit appeler `wait` pour le libérer.

Remarque : quand un processus termine, ses fils sont rattachés ailleurs dans l'arborescence (dépend du système, p.ex. au processus `init`). Ceci est utilisé pour éliminer les zombies.

Signaux

Communication entre processus:

Fichiers, tubes, sémaphores, mémoire partagée, signaux, ...

Un **signal** est un *message simple* envoyé à un processus. Il est envoyé ...

- soit par le noyau.
- soit par un autre processus.

Interface primitif pour l'interaction entre les processus.

- ⇒ utilisé pour gérer des tâches de bas niveau, p.ex. signaler que des données sont disponibles, reveiller ou terminer un processus ;
- ⇒ pas vraiment conçu pour l'échange des données.

Liste de signaux

POSIX définit une trentaine de signaux. Les systèmes d'exploitation en utilisent souvent plus (pour usage interne).

Liste des signaux : `kill -l` ou `man 7 signal`

Il est préférable d'utiliser les noms plutôt que les valeurs numériques (p.ex. pour compatibilité).

Démonstration: `victim.c`

Quelques exemples de signaux:

SIGINT – généré par Ctrl+C dans la console

SIGTERM – pour terminer un processus

SIGKILL – tuer un processus (ne peut être ignoré/annulé)

SIGUSR1 – usage libre

SIGTSTP – généré par Ctrl+Z dans la console.

SIGALRM – utilisé par sleep(3), alarm(2)

On note qu'il existent plusieurs signaux pour terminer des processus, à utiliser dans des circonstances différentes.

Transfert d'un signal

Les signaux ne sont pas instantanés; leur transfert comprend deux étapes :

Envoi: Le signal est enregistré pour processus *A* (le signal est dit *en attente*).

Livraison: Le système fait réagir processus *A* à un signal en attente.

Envoi

Appel système: `kill`

p.ex., `kill(1000, SIGTERM)` envoie le signal `TERM` au processus 1000.

→ Le noyau enregistre un signal de type `TERM` en attente pour le processus 1000.

Restrictions:

Un processus envoyant un signal doit appartenir soit à l'administrateur (root) soit au même utilisateur que le processus ciblé.

Pour tout processus et tout type de signal, ce signal est en attente ou non. P.ex., si le signal `TERM` est déjà en attente pour un processus, un deuxième envoi de `TERM` au même processus ne change rien.

D'autres moyens pour envoyer un signal (qui reviennent à l'appel `kill`) :

`kill` sur la ligne de commande

Dans un terminal, certaines combinaisons de clavier (Ctrl+C, +Z, +S, +Q, ...) font que le terminal envoie un signal à son "processus actuel".

Certaines fonctions arrangent pour le système d'envoyer un signal après un délai (`sleep`, `alarm`).

Démonstration: `alarm.c`

Livraison

Fonctionnement similaire aux interruptions :

1. Le systeme décide d'ordonnancer un processus.
On vérifie d'abord l'existence d'un signal en attente.
2. Si c'est le cas, on procède avec la livraison du signal.
3. L'exécution normale du processus continue après cette livraison.

Disposition

L'effet d'un signal sur un processus est déterminé par sa **disposition**.

Dispositions possibles :

Ign (bloquer le signal) ;

Term (terminer le processus) ;

Core (terminer et créer un *core*) ;

Stop/Cont (suspendre/continuer le processus) ;

gestionnaire de signal individuel.

Voir aussi `signal(7)`.

Modifier la disposition d'un signal

Un processus peut modifier sa disposition pour un signal avec `signal(2)` (déconseillé) ou `sigaction(2)`.

Pour certains signaux, la disposition ne peut être modifiée (notamment SIGKILL).

Un *gestionnaire* individuel est un pointeur vers une fonction qui sera exécutée lors d'une livraison.

Démonstration: `sleep.c`

Attention

La livraison d'un signal peut interrompre certaines fonctions avant leur terminaison normale.

Exemples : `sleep`, `wait`

→ tester le code renvoyé par ces fonctions,
notamment quand on joue avec des signaux !

Note : pas d'interruption si disposition `lgn`.

Hérédité des dispositions

Lors d'un `fork`, le fils hérite les dispositions de son père.

Cependant, `exec` remet les dispositions pour tout gestionnaire individuel à la disposition standard pour ce signal.

Signaux diverses

SIGCHLD : Envoyé lorsqu'un processus fils termine
(aussi lorsqu'il est stoppé dans certains cas, voir ci-dessus).
Peut servir comme alternative pour `wait`, lire attentivement la documentation de sigaction !

SIGALRM : Utilisé par `alarm` (et, selon la réalisation du système) par `sleep`.

Stop/Continue

SIGSTOP / **SIGCONT** : arrêter (temporairement) et continuer l'exécution d'un processus. La disposition de SIGSTOP ne peut pas être modifiée.

SIGTSTP : comme SIGSTOP, mais peut être ignoré. Envoyé par Ctrl+S dans la console (Ctrl+Q pour SIGCONT).

Groupes de processus

Chaque processus appartient à un **groupe de processus**.

Les groupes possèdent un identifiant numérique.
(typiquement identique au PID d'un membre)

`setpgid` peut changer le groupe d'un processus. Exemples :

`setpgid(p, g)` – rajoute *p* au groupe *g*

`setpgid(0, 0)` – équivalent à `setpgid(p, p)`, où *p* est le PID du processus

Groupe de processus et signaux

Un signal peut être envoyé à tous les membres d'un groupe.

P.ex., `kill` interprète un argument négative comme le numéro d'un groupe. (voir aussi `killpg`)

Exaeples: `kill(SIGINT, -100)` – envoyer SIGINT au groupe 100

Groupes dans le shell

Un shell est exécuté dans un **terminal**.

Le terminal possède la notion d'un *groupe de premier plan*.

Les entrées de l'utilisateur (et les signaux en raison de Ctrl+C etc) sont envoyés par le terminal à ce groupe.

Quand le shell lance une commande, il crée un nouveau groupe pour ce processus et en fait le groupe de premier plan.

Foreground/background jobs

Ctrl+Z dans le terminal envoie SIGTSTP au groupe *de premier plan*.

Comportement par défaut : Les processus s'arrêtent, le shell redevient le *processus de premier plan*.

bg et **fg** envoient SIGCONT à ce groupe ce qui leur permettra de continuer en premier plan ou en arrière plan.

Blockage des signaux

Un processus peut retarder la livraison d'un signal à l'aide de son **masque de blockage** $\Rightarrow$ c'est pour bloquer *temporairement*, sinon utiliser SIG_IGN.

Ce masque peut être altéré par **sigprocmask(2)**.

Un signal dans le masque reste en attente et sera livré lorsque son type est enlevé du masque.

En plus, des signaux peuvent être bloqués pendant une livraison, voir **sigaction**.

sigsuspend(2) permet de remplacer le masque en attendant un signal, puis restaurer (atomiquement) le masque.