Systèmes d'exploitation

Les processus

Thomas Ropars

Email: thomas.ropars@univ-grenoble-alpes.fr

Website: tropars.github.io

Dans ce cours

- Définition du concept de processus
- États et propriétés
- La mémoire des processus
- API pour manipuler des processus
    - `fork()`
    - `wait()`
    - `exec()`

Le concept de processus

Définition du concept de processus

```
Un processus est une abstraction créée par le système d'exploitation
qui représente un programme en cours d'exécution
```

- Le programme en lui même est un ensemble d'instructions (code exécutable)
    - Stockées dans un fichier (statique)
- Le processus est une instance d'un programme en cours d'exécution
    - Son état évolue au cours du temps
    - Il a un début et une fin

Un processus est défini par

- Le code exécutable du programme
- Les données qui lui sont associées
- Un contexte d'exécution
    - Valeur des registres
    - Information liées à la gestion de la mémoire
    - le compteur d'instructions (*program counter*)
    - Liste de descripteurs de fichiers
    - etc.

Pourquoi les OS introduisent ce concept?

Objectifs: mettre en oeuvre un système multi-tâches

Les processus permettent à l'OS:

- De savoir quels programmes s'exécutent à chaque instant sur le système
- De gérer la consommation de ressources de chacun

Commandes pour connaître les processus qui s'exécutent sur le système

- `ps`: Liste de tous les processus
- `top`: Liste des processus ordonnés en fonction de leur consommation de ressources

Isolation des processus

L'OS offre l'illusion à chaque processus qu'il s'exécute seul sur le système:

- Illusion d'un accès exclusif aux ressources matérielles (processeurs, mémoire)
    - Le système se comporte comme si le processeur exécutait les instructions du programme les unes après les autres sans jamais être interrompu
        - Contrôle de flot logique
    - Mémoire virtuelle privée
        - Privée = Illusion d'un accès exclusif
        - Virtuelle= Illusion d'une mémoire *infinie*
- Isolation entre processus
    - L'OS protège chaque processus des actions des autres processus

En pratique, de nombreux processus s'exécutent en même temps sur un système

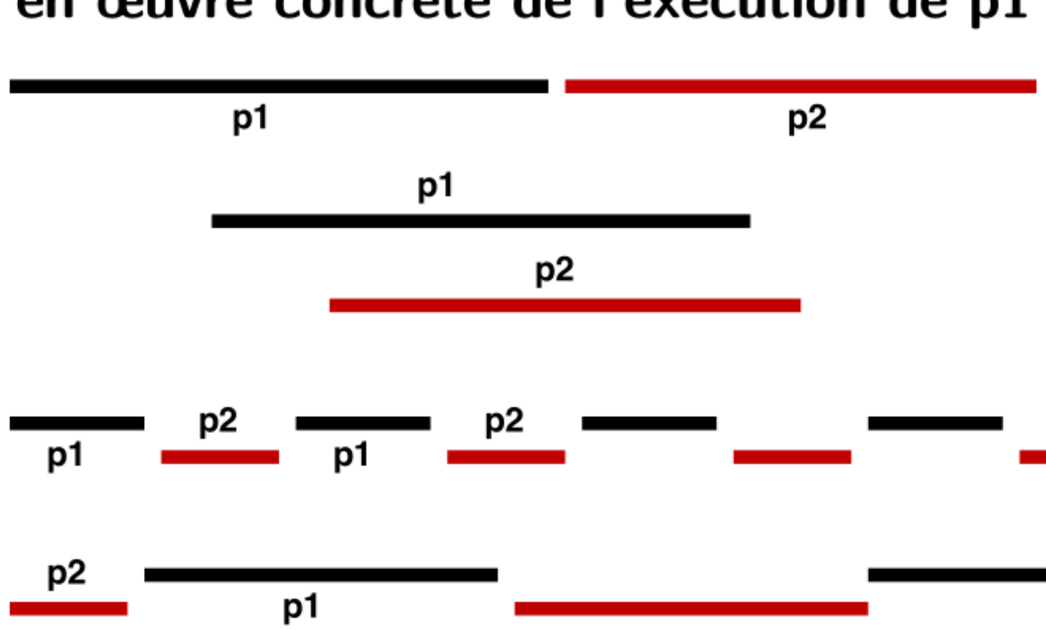
- Plusieurs programmes: un navigateur web + un éditeur de texte + un lecteur audio + des services systèmes (interface graphique, syslog, etc.)
- Plusieurs instances d'un même programme: `Emacs file1.txt` + `Emacs file2.txt`

Parallélisme et pseudo-parallélisme

- **Soit deux processus p1 et p2 (exécution de deux programmes séquentiels P1 et P2)**

représentation abstraite
(p1 et p2 sont parallèles)

- **Mise en œuvre concrète de l'exécution de p1 et p2**



exécution séquentielle
(1 processeur)

exécution parallèle
2 processeurs

exécution pseudo-parallèle
1 processeur

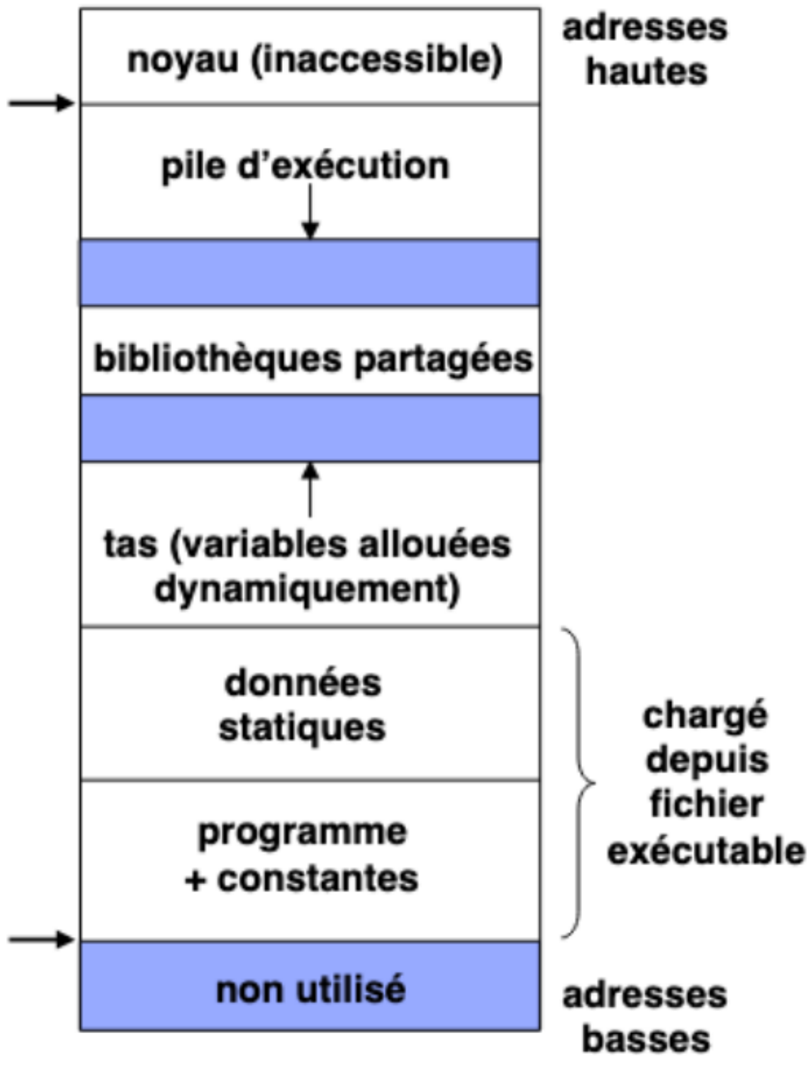
une autre exécution
pseudo-parallèle

Etats d'un processus

Informations associées à un processus (UNIX)

- Chaque processus est identifié par un PID (*Process Identifier*) unique
    - Assigné par l'OS
        - De manière *aléatoire*
        - Max PID par défaut: 32768 ou 4194304
            - `cat /proc/sys/kernel/pid_max`
    - La fonction `getpid()` permet d'obtenir l'identifiant du processus courant
- L'état du processus comprend:
    - L'état de la mémoire virtuelle
    - La valeur du compteur d'instruction
    - La valeur des registres
    - etc.

Mémoire virtuelle

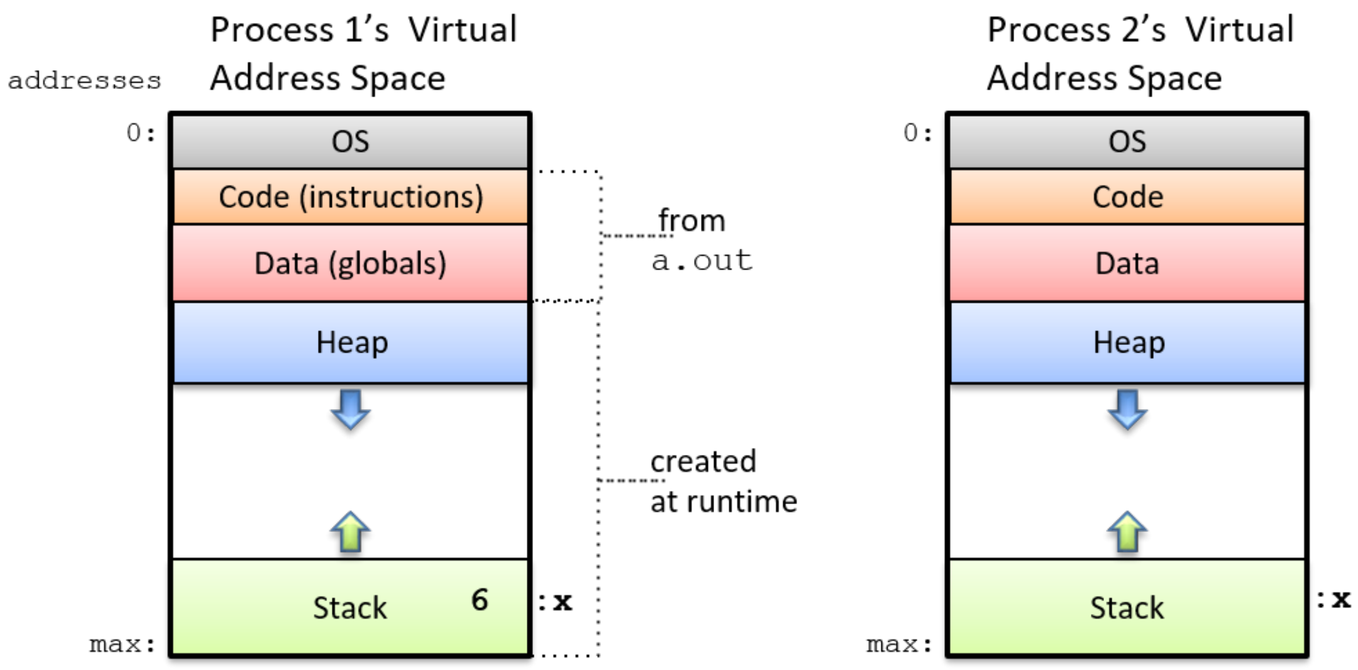


mémoire d'un processus dans Unix

Mémoire virtuelle

Le mémoire virtuelle de chaque processus est organisée de la même manière

- L'OS (avec l'aide du matériel) se charge de traduire les adresses virtuelles en adresses physiques
- La taille de la mémoire virtuelle = 2^n
    - $n = 32$ sur les systèmes 32 bits
    - $n = 64$ sur les systèmes 64 bits (en réalité $n = 48$ sur les systèmes actuels)



Crédits: Dive into Systems S. J. Matthews et al. -- (Attention: représentation inversée par rapport à la figure précédente)

Test

Environnement d'un processus (1)

- **Dans Unix, un processus a accès à un certain nombre de variables qui constituent son environnement.**
  - faciliter la tâche de l'utilisateur en évitant d'avoir à redéfinir tout le contexte du processus (nom de l'utilisateur, de la machine, terminal par défaut, etc.)
  - personnaliser différents éléments comme le chemin de recherche des fichiers, le répertoire de base (home), le shell utilisé, etc.
- **Certaines variables sont prédéfinies dans le système. L'utilisateur peut les modifier, et peut aussi créer ses propres variables d'environnement.**
  - La commande **env** (sans paramètres) affiche l'environnement courant
- **Pour attribuer la valeur <val> à la variable VAR,**
  - Shell tcsh : **setenv VAR <val>**
  - Shell bash : **export VAR=<val>**
- **La commande echo $VAR affiche la valeur courante de la variable VAR**

Environnement d'un processus (2)

On peut aussi consulter et modifier les variables d'environnement par programme

```
#include <stdlib.h>
char * getenv (const char *nom);
int putenv (const char *chaine);
int setenv (const char *nom, const char *valeur, int écraser);
int unsetenv (const char *nom);
```

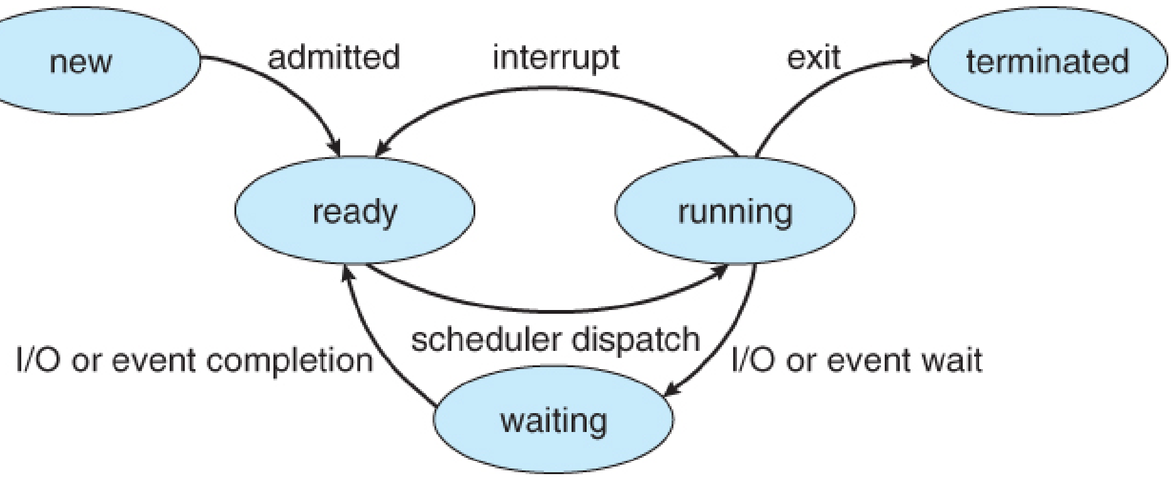
Exemples :

```
char *var = getenv("USER");
if (var != NULL) printf("utilisateur: %s", var);

putenv("MAVARIABLE=toto");
setenv("MAVARIABLE", "toto", 1);

unsetenv("MAVARIABLE"); // équivalent à putenv("MAVARIABLE") !!!
```

État d'un processus



- **New**: Le processus est en train d'être créé
- **Ready**: Le processus est prêt à s'exécuter mais l'OS a choisi d'exécuter un autre processus
- **Running**: Le processus est en train de s'exécuter sur le processeur
- **Waiting**: Le processus est bloqué en attente (par exemple d'une entrée au clavier)
- **Terminated**: Le processus est terminé

Crédits: Figure by Silberschatz et al.

API pour manipuler des processus

Manipuler des processus dans les systèmes UNIX

3 ensembles de fonctions

- *fork()* pour la création
- *exec()* pour le chargement d'un nouvel exécutable
- *wait()* pour attendre la terminaison d'un processus

Créer un nouveau processus

- Appel système `fork()`
  - `pid_t fork(void)`
  - Crée un nouveau processus (processus fils) **identique** au processus appelant (processus père)
  - Valeur de retour:
    - `0` pour le fils
    - `pid du fils` pour le père (ou `-1` si le processus n'a pas pu être créé)

```c
pid_t pid = fork();
if (pid == 0) { /* Le père et le fils exécutent ceci */
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Attention: `fork()` est appelé une fois et retourne deux fois (une fois dans le père et une fois dans le fils)

Informations en plus sur fork()

- Le père et le fils exécutent le même code
    - La seule chose qui diffère est la valeur retournée par `fork()`
- L'état du fils est une copie de l'état du père. Il a sa propre copie (privée):
    - de la mémoire virtuelle
    - des variables d'environnement
    - de la liste de fichiers ouverts
    - etc.
- L'ordre d'exécution est non déterministe
    - Une fois le fils crée, le père et le fils sont prêt à s'exécuter
    - Qui s'exécute en premier après l'appel à `fork()` (le père? le fils? les deux?)
        - Par d'ordre prédéfini

Terminaison d'un processus

Auto destruction

- Appel explicite à `exit()` par le processus
    - `void exit(int status)`
    - Par convention, `status == 0` signifie *succès*
    - Renvoyer une valeur différente de 0 signifie *erreur*

Tuer un processus

- On peut envoyer un signal à un processus pour le tuer (`SIG_TERM` ou `SIG_KILL`)
    - commande `kill`
    - `Ctrl+C`

Processus démons

- Certains processus ne se terminent jamais
    - Processus démons réalisant des fonctions du système

Attente d'un processus

- `pid_t wait(int *child_status)`
  - Bloque le processus appelant jusqu'à ce que l'un de ses processus fils termine
  - La valeur retournée est le `pid` du fils terminé
  - Si `child_status != NULL`, la variable pointé se voit attribuée un statut décrivant la manière dont le fils s'est terminé
  - Si plusieurs fils ont terminé, un est pris de manière arbitraire

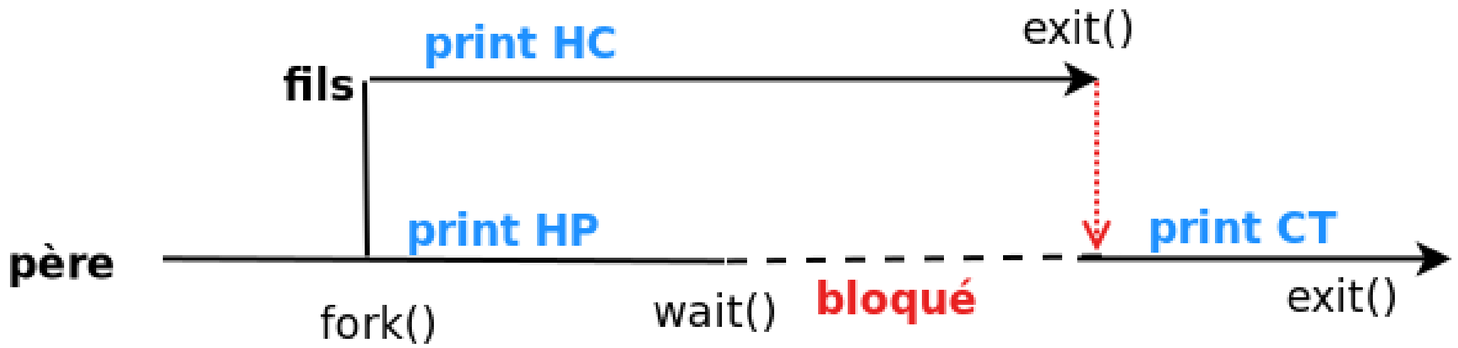
- `pid_t waitpid(pid_t pid, int *child_status, int options)`
  - Bloque le processus appelant en attente d'un processus spécifique
  - Plusieurs options (voir les pages de `man`)

Exemple d'attente d'un processus fils

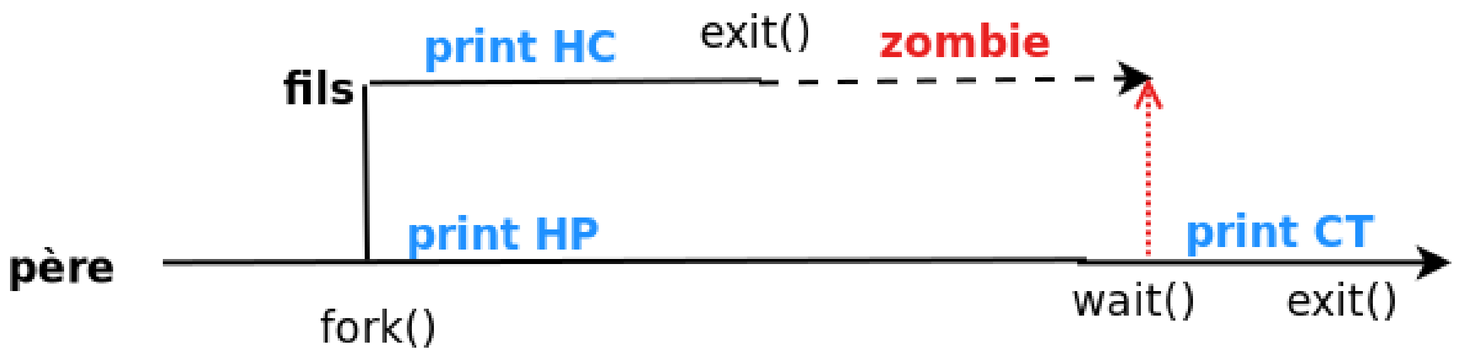
```c
int main(void) {
    int child_status;
    if (fork() == 0) {
        printf("HC: hello from child\n");
    }
    else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    exit(0);
}
```

Exemple d'attente d'un processus fils

Scénario 1



Scénario 2



Processus zombie

Idée

- Quand un processus est dans l'état `terminated`, il consomme encore des ressources
    - L'OS conserve des informations sur le processus
- Le processus est appelé `zombie`

Supprimer un zombie

- Attente explicite par le processus père (wait() ou waitpid())

Que se passe-t-il si un père se termine sans attendre un processus fils

- Le processus non attendu est reparenté avec un processus *spécial* (souvent appelé `init`)
    - Ce processus va supprimer les zombies
- Conséquence: Attente explicite indispensable seulement pour les applications ayant une durée de vie très longue (shell, serveur web, etc.)

Analyser le status d'un fils terminé

Utilisation de macros

- `WIFEXITED()` et `WEXITSTATUS()` (voir `man 2 wait`)

```c
int main()
{
    pid_t pid;
    int child_status;
    if ((pid = fork()) == 0)
        exit(EXIT_SUCCESS); /* Child */

    pid_t wpid = wait(&child_status);
    if (WIFEXITED(child_status)){
        printf("Child %d terminated with exit status %d\n",
        wpid, WEXITSTATUS(child_status));
    }
    else
        printf("Child %d terminated abnormally\n", wpid);
}
```

Exécuter un nouveau programme dans un processus

Utilisation d'un primitive exec

- Appel système

```c
int execve(char *path, char * argv[], char * envp[]);
```

- Reconfigure complètement l'état du processus appelant
  - Charge l'exécutable pointé par `path`
  - Avec la liste d'arguments `argv` (terminée par une entrée `NULL`)
    - Par convention `argv[0]` est le nom de l'éxécutable
  - Avec la liste de variables d'environnement `envp`
  - La mémoire virtuelle du processus est écrasé
    - Mais le pid est conservé

Attention: cet appel de retourne pas (à moins d'une erreur)

Les différentes variantes d'Exec

- **Il existe 6 versions différentes de la primitive Exec:**
  - `execv`, `execve`, `execvp`, `execl`, `execle`, `execlp`
- **Nuances**
  - Passage des paramètres
    - Sous forme de tableau de pointeurs (`v`) : `execv`, `execve`, `execvp`
    - Sous forme de liste (`l`) : `execl`, `execle`, `execlp`
  - Passage de l'environnement
    - Explicite : paramètre pointeur vers tableau de pointeurs (e) : `execve`, `execle`
    - Implicite (héritage de l'environnement courant) : les autres primitives
  - Chemin de l'executable
    - Recherche dans le `PATH` (`p`) : `execlp`, `execvp`
    - Nécessité de spécifier le chemin complet : les autres primitives
- **En général, une seule de ces primitives est fournie par le système d'exploitation (`execve`) et les autres sont construites au dessus dans des bibliothèques**

Exemple avec execl

```c
int main(void) {
    int res;
    if (fork() == 0) { /* le fils charge le programme cp */
        res = execl("/usr/bin/cp", "cp", "foo", "bar", NULL);
        if (res < 0) {
            printf("error: execl failed\n");
            exit(-1);
        }
    }
    wait(&res); /* le père attend le fils */
    if (WIFEXITED(res) && (WEXITSTATUS(res) == 0)) {
        printf("copy completed successfully\n");
    } else {
        printf("copy failed\n");
    }
    exit(0);
```

Résumé

- La notion de processus
    - Informations associées
    - État
    - Mémoire virtuelle
- Manipuler des processus
    - Création
    - Attente
    - Exécuter un nouveau programme