

# Architecture et Systèmes

Stefan Schwoon

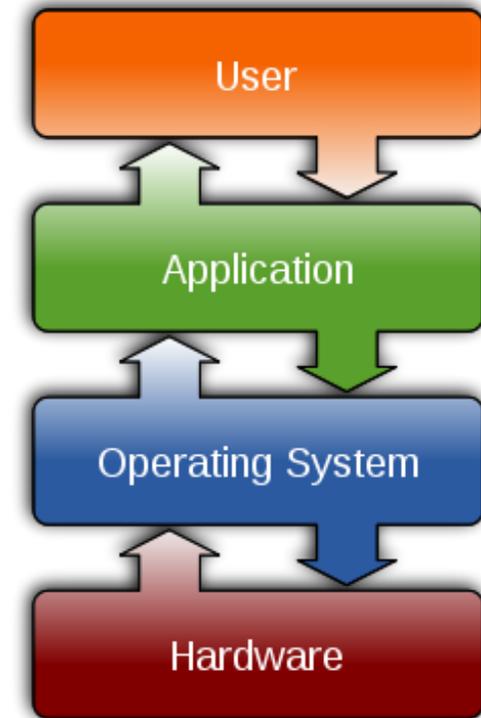
Cours L3, 2015/16, ENS Cachan

# Systeme d'exploitation

---

Caractérisation selon *Tanenbaum*:

*Software consists of two categories: the system programs, which enable the operation of the computer, and the application programs, which resolve the user's problems. The operating system is the most important of the system programs. It controls the resources of a computer and provides a basis for the application programs.*



Du coup le système intervient entre le matériel et les logiciels avec lesquels travaille l'utilisateur.

# Tâches d'un système d'exploitation

---

Fournir des fonctionnalités :

Simplifier la création des applications.

Abstraire les détails du matériel, permettre les mêmes logiciels de tourner sur de différents ordinateurs.

Fournir des structures de données abstraites : fichiers, processurs, ...

Permettre la concurrence entre applications.

# Tâches d'un système d'exploitation

---

Assurer la bonne opération de l'ordinateur :

Assurer qu'un utilisateur ne gêne pas un autre, soit par accident, soit par malveillance.

Pareil pour les applications d'un même utilisateur.

Assurer une distribution juste des ressources.

Protection contre les virus etc.

# Organisation d'un système d'exploitation

---

On distingue les composants suivants :

Le **noyau** (*kernel*): code “privilégié qui gère l'interaction entre les processus et entre les périphériques; réside en mémoire pendant l'opération de l'ordinateur.

Les **bibliothèques** qui donnent accès aux fonctions du noyau.

**Applications** de niveau bas (shell, ls, ...)

# Mode privilégié

---

La plupart des processeurs aujourd'hui permettent d'exécuter du code dans (au moins) deux modes différents :

**mode privilégié** (ou *mode noyau*) : toutes les opérations sont permises

**mode utilisateur** : certains opérations sont interdites, p.ex.

interaction avec les périphériques

opération d'interdiction d'interruptions

restrictions pour l'accès mémoire

Le mode est réalisé, p.ex. par un drapeau dans le registre statut.

# Noyau

---

L'ordinateur démarre en *mode privilégié*.

Le système charge son noyau en mémoire, établit toutes les structures nécessaires pour son opération puis passe en mode utilisateur pour lancer les applications.

Passer en mode privilégié : opération spéciale (appel système), p.ex.

*interruption* (matériel ou logiciel)

opération *syscall*.

# Les appels système

---

Interface pour demander un service au système :

Mécanisme typique :

tout service correspond à un numéro

l'identifiant du service et d'autres paramètres sont mis dans les registres

on exécute un appel système

Les détails dépend du système (et de sa version) – pas standardisé.

Typiquement, le programmeur n'utilise pas ce mécanisme directement.

# Les bibliothèques système

---

Les détails des appels systèmes sont susceptible de changer entre différents versions d'un même système.

Du coup, on y accède par les fonctions d'une bibliothèque.

Dans Linux : `strace` affiche les appels systèmes dans la forme des appels bibliothèque.

Pages man :

Section 1 : Applications de bas niveau

Section 2 : Bibliothèque système

Section 3 : Autres appels de bibliothèque (p.ex. `printf`)

etc

# La norme POSIX

---

Pour permettre la compatibilité entre différents systèmes, les vendeurs ont créé des **normes**.

Dans ce cours on traite la norme **POSIX** qui est réalisée par plusieurs systèmes importants :

OS X (complètement compatible)

Linux (largement compatible)

Windows (partiellement, p.ex. pour fichiers et réseaux)

# Applications de bas niveau

---

Ensemble de programmes pour interagir avec le système (sans devoir programmer).

Exemples: ligne de commande et d'autres programmes associés (`ls`, `cat`, ...)

On pourrait y rajouter l'interface graphique – distinction entre programmes “du système” et “applications” pas toujours précise.

# Domaines d'un système d'exploitation

---

Processus, signaux, ordonnancement

Fichiers, réseau, entrées/sorties en général

Gestion de mémoire

Gestion d'utilisateurs

(éventuellement interface graphique)

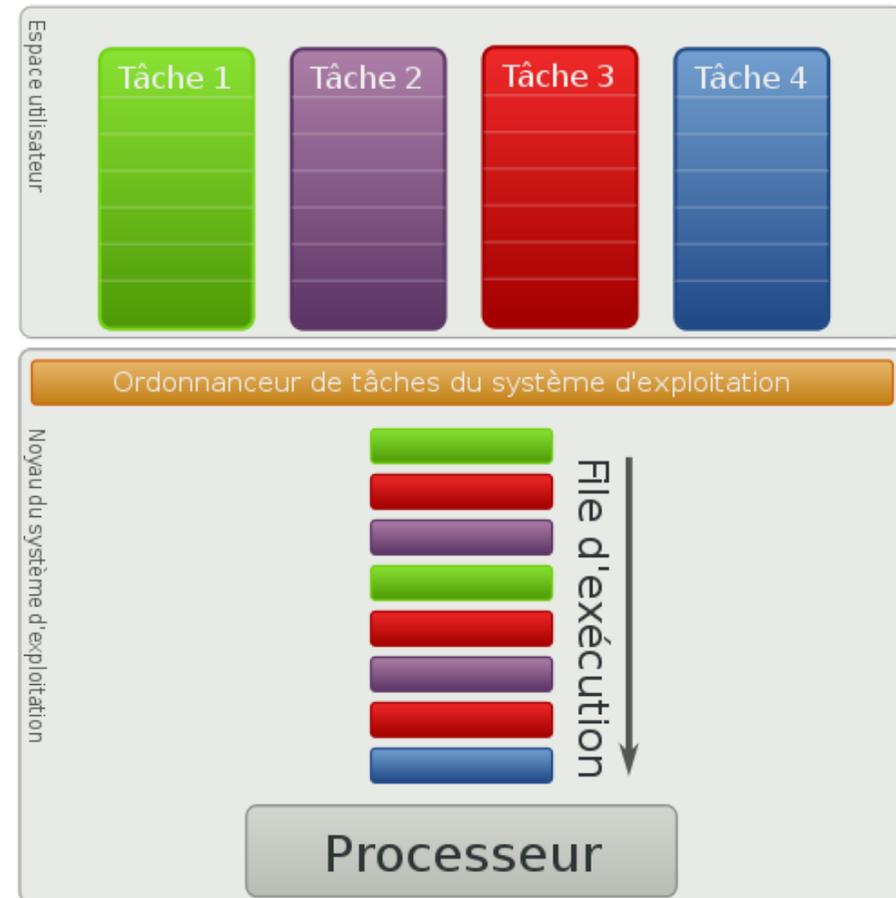
...

# 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 ?

---

Activité séquentielle poursuivie par l'ordinateur ; possède son code, ses données et certains attributs.

Ne pas confondre avec un *programme* : un même programme peut correspondre à plusieurs processus.

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

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

# Pourquoi des processus?

---

Facilitation de la programmation : on peut écrire ses programmes dans un style séquentiel comme s'il était la seule tâche du système (sauf si une interaction est souhaitée), le système se charge des aspects de concurrence.

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

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

# Réalisation de la concurrence

---

Un seul processeur ne peut exécuter qu'un seul processus à un moment donné.

Le système met tout processus dans sa propre "machine virtuelle" et bascule régulièrement entre les processus (ordonnancement), en utilisant l'interruption de l'horloge.

Perspective d'utilisateur : illusion de concurrence

perspective du programmeur: illusion d'une exécution séquentielle

On regardera l'[ordonnancement](#) plus tard.

# Exécution d'une commande dans le shell

---

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 donnés. En C:

```
int main (int argc, char **argv)
```

Le shell attend la fin du processus avant de continuer.

# Hierarchie et création de processus

---

Strictement parlant, le seul moyen pour créer des processus est de les dupliquer par l'appel système `fork`.

`fork` crée un processus (le “père”) et en crée une copie (le “fils”).

Si la création réussit, la fonction renvoie l'identifiant du fils au père et 0 au fils. (En cas d'échec, c'est -1).

Détails: `man fork`

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 `ps tree`.

# 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 ; tout changement par la suite 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, ...)

etc, on va en étudier quelques uns ...

# Vie et mort d'un processus

---

Un processus se *termine* par l'appel `exit` ou par un `return` de la fonction `main`.

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 (c'est que fait le shell...).

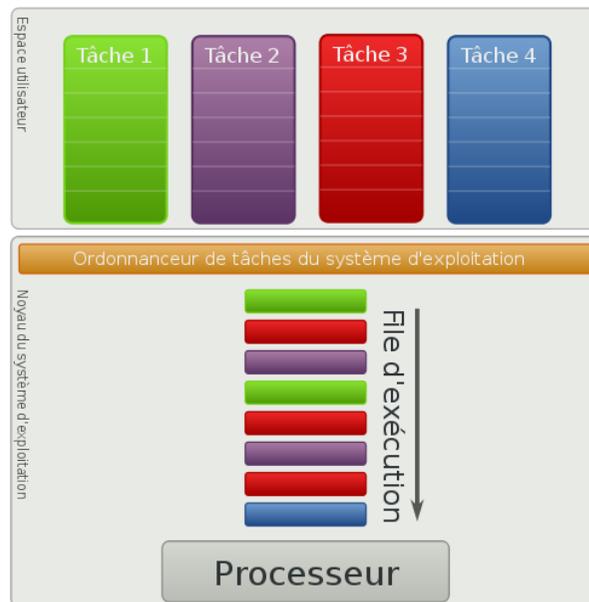
`wait` renvoie également le code de sortie du fils et certaines informations sur les circonstances de sa terminaison.

En particulier, le père obtient le code de sortie en utilisant le macro `WEXITSTATUS`.

# 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 sera pas ordonnancé pour l'instant.

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

# États spéciaux

---

Unix connaît certains états en plus :

**Crée:** il existe un créneau dans la liste de processus, mais le processus n'est pas encore prêt à exécuter.

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

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.

# Exec

---

Il existe toute une famille de fonctions (`exec` etc) qui permettent de remplacer le code du processus avec un autre programme.

Exemple (on lance une commande dans le shell) :

L'utilisateur donne la commande.

Le shell fait appel à `fork`.

Le processus fils fait appel à `execvp` (p.ex.) pour lancer le programme souhaité par l'utilisateur.

Le parent 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.