Architecture et Système

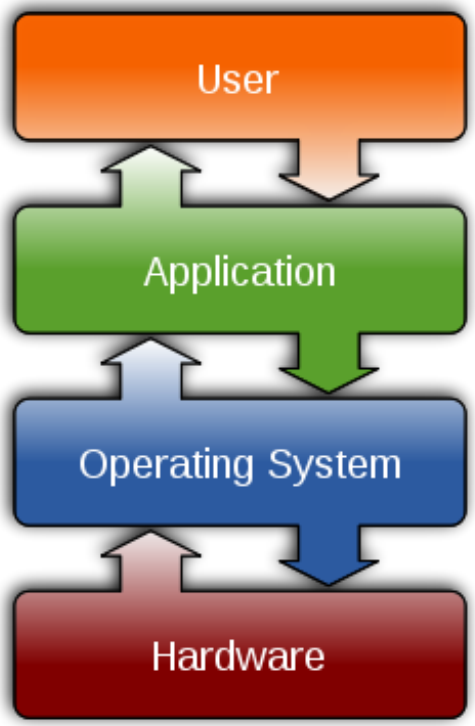
Stefan Schwoon

Cours L3, 2018/19, ENS Cachan

Système 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 systéme intervient entre le matériel et les logiciels avec lesquels un utilisateur travaille.

Tâches d'un système d'exploitation

Facilitation, par exemple :

Simplifier la création des applications

Permettre la concurrence entre applications

Abstraire les détails du matériel, permettre aux mêmes logiciels de fonctionner sur divers ordinateurs

$\Longrightarrow$ bibliothèques, structures de données abstraites : fichiers, processus, ...

Tâches d'un système d'exploitation

Contrôle, par exemple :

Assurer qu'aucun utilisateur ne gêne un autre (par accident ou malveillance)

Protéger les applications d'un utilisateur en cas l'une parmi elles a une faute

Assurer une distribution juste des ressources

Protection contre les virus etc.

Modes utilisateur et privilégié

Les processeurs courants typiquement possèdent au moins deux modes différents : mode utilisateur et mode privilégié (ou *mode noyau*).

Dans le processeur, ce mode peut être indiqué par un bit dans le registre de statut qui sera pris en compte pour certains opérations.

En mode utilisateur :

l'interaction avec les périphériques est interdite

l'accès mémoire est restreint ; certains adresses sont 'lecture seulement' ou 'hors limite' (chapitre gestion mémoire à venir)

on ne peut pas interdire les interruptions (ni modifier les vecteurs d'interruption)

Modes utilisateur et privilégié

En mode privilégié :

toute opération est permise

accès mémoire non-restreint

modification de la configuration mémoire (p.ex. droits d'accès)

Modes utilisateur et privilégié

L'ordinateur démarre en mode privilégié.

Lors du démarrage de l'ordinateur, le système fait le suivant :

Mise en place du code pour le système (le noyau).

Basculer les vecteurs d'interruption vers les bonnes adresses dans le système (notamment interruption horloge pour l'ordonnancement)

Configurer les accès mémoire pour les processus utilisateur

Basculer en mode utilisateur et lancer un premier processus utilisateur (p.ex. un login)

Modes utilisateur et privilégié

Comment basculer du mode utilisateur vers mode privilégié ?

Plusieurs réalisations possibles selon le type de processeur, p.ex.:

- par une interruption (matériel ou logiciel) qui renvoye donc vers le code du système
- par une instruction syscall

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 bas niveau (shell, ls, . . .)

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 (p.ex. une interruption logiciel)

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 des bibliothèques.

Dans Linux :

- `ltrace` affiche les appels bibliothèque d'un processus
- `strace` affiche les appels systèmes dans la forme des appels bibliothèque.

Pages man :

- Section 1 : Applications de bas niveau
- Section 2 : Biliothèque système
- Section 3 : Autres appels de bibliothèque (p.ex. `printf`)
- etc

La norme POSIX

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

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

- macOS (complètement compatible)
- Linux (largement compatible)
- Windows (partiellement, p.ex. pour fichiers et réseaux)

Liens statiques et dynamiques

La liaison entre le code d'un programme et les bibliothèques peut être . . .

. . . statique :

le code des appels bibliothèque est inclus dans un fichier exécutable lors de la compilation

Drapeau `-static` dans gcc

. . . dynamique :

code bibliothèque non inclus dans fichier exécutable ; bibliothèques partagées

le fichier exécutable contient des informations sur les bibliothèques requises (`ldd`)

lors du lancement, un *éditeur de liens* assure que tout appel pointe vers la bonne adresse

Applications de bas niveau

Ensemble de programmes pour interagir avec le système.

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

On pourrait y rajouter l'interface graphique – la distinction entre programmes "du système" et "applications" n'est pas toujours précise.

Domaines d’un système d’exploitation

Processus, signaux, ordonnancement

Fichiers, réseau, entrées/sorties en général

Gestion de mémoire

Gestion d’utilisateurs

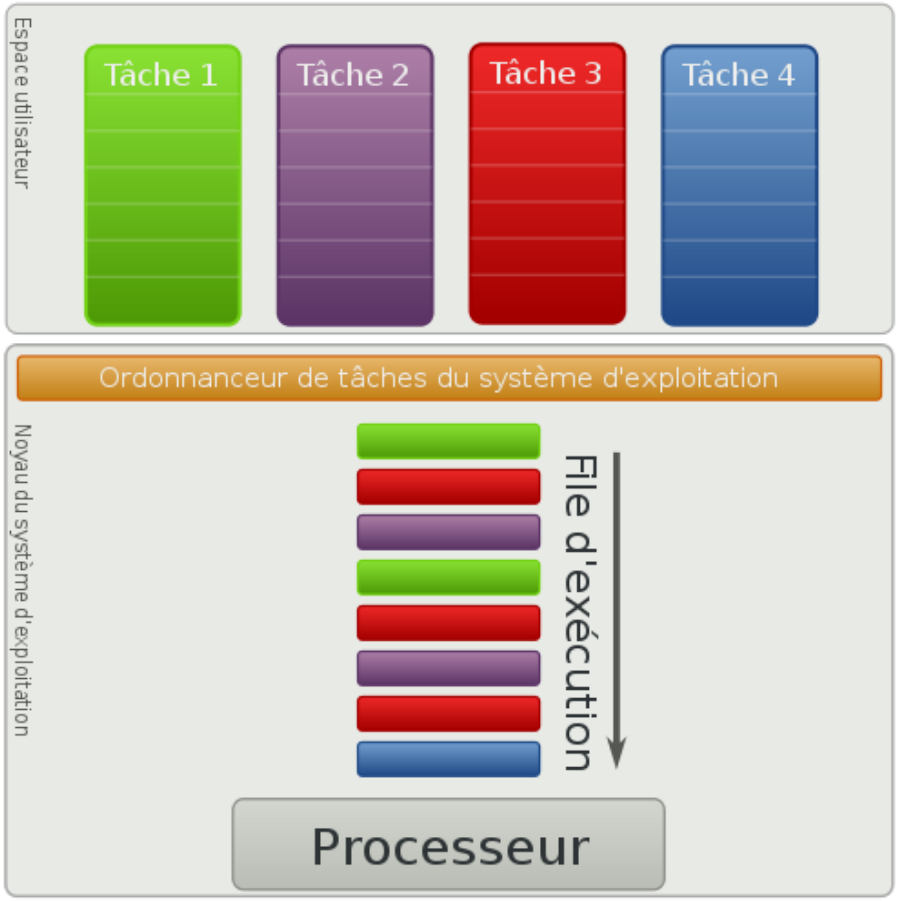
(éventuellement interface graphique)

...

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 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 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`, `pstree` ou `top`.

Pourquoi des processus ?

Facilitatation 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.

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 "mémoire 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 et la gestion de mémoire 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 crée qui exécute le programme avec les paramètres données. 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'échèc, 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 `pstree`.

fork et mémoire

Aprés 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 identiant 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` 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 ce 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`.

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.

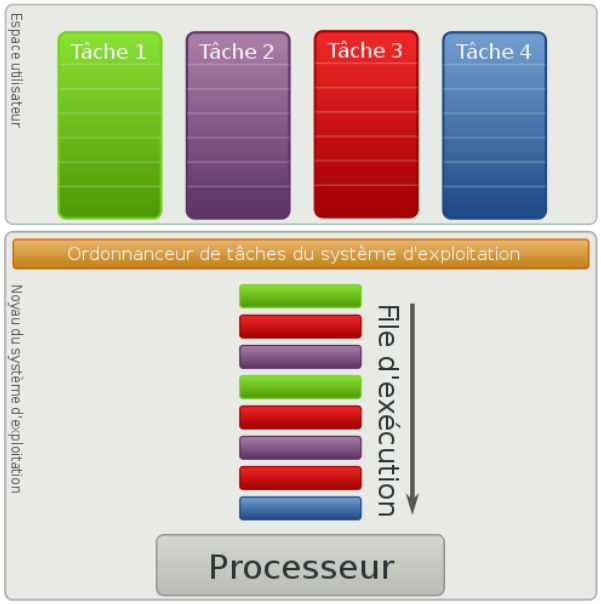
`cmd1 || cmd2`: exécuter cmd1, et si le code de sortie est non-zéro, exécuter cmd2

`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.

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 toujour. 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.