

Introduction aux systèmes d'exploitation

Didier Puzenat dpuzenat@univ-ag.fr

Université Antilles Guyane

1 Introduction

- **Définition d'un système d'exploitation**
 - Le noyau du système
 - Les bibliothèques du système
 - Outils systèmes et applications de base

2 Principes des systèmes d'exploitation

- Notions de compte et de « droits »
- Équité et confidentialité
- Exemple d'Unix / de Linux

3 Gestion des processus

- Notion de processus
- Politiques d'ordonnancement des processus
- Manipulation des processus

Définition

Définition d'un **système d'exploitation** (SE),
en anglais Operating System (OS)

Définition

Définition d'un **système d'exploitation** (SE),
en anglais Operating System (OS) :

- ensemble de programmes assurant l'**interface** entre
les **ressources matérielles** d'un ordinateur et
les **applications** de l'utilisateur.

Définition

Définition d'un **système d'exploitation** (SE),
en anglais Operating System (OS) :

- ensemble de programmes assurant l'**interface** entre les **ressources matérielles** d'un ordinateur et les **applications** de l'utilisateur.
- un SE assure notamment le démarrage de l'ordinateur et fournit aux programmes applicatifs des points d'entrée génériques pour les périphériques.

Définition

Définition d'un **système d'exploitation** (SE),
en anglais Operating System (OS)

- ensemble de programmes assurant l'**interface** entre les **ressources matérielles** d'un ordinateur et les **applications** de l'utilisateur.
- un SE assure notamment le démarrage de l'ordinateur et fournit aux programmes applicatifs des points d'entrée génériques pour les périphériques.

⇒ **le SE permet de s'affranchir de la machine physique :**

- on écrit pas une application pour une machine donnée, mais pour un SE donnée (voire pour une famille de SE) ;
- on ne se forme pas à une machine mais à un SE...

Composition typique d'un SE

Un système d'exploitation est typiquement composé

Composition typique d'un SE

Un système d'exploitation est typiquement composé :

- d'un **noyau** (kernel en anglais) :
 - fournit des mécanismes d'abstraction du matériel notamment de la mémoire et des processeur(s) ;
 - permet l'abstraction des échanges entre logiciels et périphériques matériels ;
 - facilite la communication entre les processus. . .

Composition typique d'un SE

Un système d'exploitation est typiquement composé :

- d'un **noyau** (kernel en anglais) :
 - fournit des mécanismes d'abstraction du matériel notamment de la mémoire et des processeur(s) ;
 - permet l'abstraction des échanges entre logiciels et périphériques matériels ;
 - facilite la communication entre les processus. . .
- d'un **système de gestion de fichiers** ;

Composition typique d'un SE

Un système d'exploitation est typiquement composé :

- d'un **noyau** (kernel en anglais) :
 - fournit des mécanismes d'abstraction du matériel notamment de la mémoire et des processeur(s) ;
 - permet l'abstraction des échanges entre logiciels et périphériques matériels ;
 - facilite la communication entre les processus. . .
- d'un **système de gestion de fichiers** ;
- de **bibliothèques** dynamiques (chargées « à la volée ») ;

Composition typique d'un SE

Un système d'exploitation est typiquement composé :

- d'un **noyau** (kernel en anglais) :
 - fournit des mécanismes d'abstraction du matériel notamment de la mémoire et des processeur(s) ;
 - permet l'abstraction des échanges entre logiciels et périphériques matériels ;
 - facilite la communication entre les processus. . .
- d'un **système de gestion de fichiers** ;
- de **bibliothèques** dynamiques (chargées « à la volée ») ;
- d'un ensemble d'**outils système** ;

Composition typique d'un SE

Un système d'exploitation est typiquement composé :

- d'un **noyau** (kernel en anglais) :
 - fournit des mécanismes d'abstraction du matériel notamment de la mémoire et des processeur(s) ;
 - permet l'abstraction des échanges entre logiciels et périphériques matériels ;
 - facilite la communication entre les processus. . .
- d'un **système de gestion de fichiers** ;
- de **bibliothèques** dynamiques (chargées « à la volée ») ;
- d'un ensemble d'**outils système** ;
- de programmes **applicatifs de base**.

1 Introduction

- Définition d'un système d'exploitation
- **Le noyau du système**
- Les bibliothèques du système
- Outils systèmes et applications de base

2 Principes des systèmes d'exploitation

- Notions de compte et de « droits »
- Équité et confidentialité
- Exemple d'Unix / de Linux

3 Gestion des processus

- Notion de processus
- Politiques d'ordonnancement des processus
- Manipulation des processus

Le noyau du système

Rôles du noyau

Le noyau du système

Rôles du noyau :

- **gestion des périphériques** (au moyen de pilotes) ;

Le noyau du système

Rôles du noyau :

- **gestion des périphériques** (au moyen de pilotes) ;
- **gestion des processus** :
 - attribution de la mémoire aux processus ;
 - ordonnancement des processus ;
 - synchronisation et communication entre processus... .

Le noyau du système

Rôles du noyau :

- **gestion des périphériques** (au moyen de pilotes) ;
- **gestion des processus** :
 - attribution de la mémoire aux processus ;
 - ordonnancement des processus ;
 - synchronisation et communication entre processus... .
- **gestion des fichiers** (via le système de gestion de fichiers) ;

Le noyau du système

Rôles du noyau :

- **gestion des périphériques** (au moyen de pilotes) ;
- **gestion des processus** :
 - attribution de la mémoire aux processus ;
 - ordonnancement des processus ;
 - synchronisation et communication entre processus... .
- **gestion des fichiers** (via le système de gestion de fichiers) ;
- **gestion des protocoles réseau** (par exemple TCP/IP).

1 Introduction

- Définition d'un système d'exploitation
- Le noyau du système
- Les bibliothèques du système**
- Outils systèmes et applications de base

2 Principes des systèmes d'exploitation

- Notions de compte et de « droits »
- Équité et confidentialité
- Exemple d'Unix / de Linux

3 Gestion des processus

- Notion de processus
- Politiques d'ordonnancement des processus
- Manipulation des processus

Les bibliothèques dynamiques

Définition d'une bibliothèque dynamique

Les bibliothèques dynamiques servent à regrouper les opérations les plus utilisées dans les programmes informatiques.

Les bibliothèques dynamiques

Définition d'une bibliothèque dynamique

Les bibliothèques dynamiques servent à regrouper les opérations les plus utilisées dans les programmes informatiques.

- **réduction de la taille d'un exécutable :**
le code exécutable de la bibliothèque est chargé à la volée ;
- **utilisation de fonctions validées :**
fonctionnement, sécurité, performances...
- **efficacité des mises à jour :**
correction de bug, trous de sécurité, améliorations....

Les bibliothèques dynamiques

Définition d'une bibliothèque dynamique

Les bibliothèques dynamiques servent à regrouper les opérations les plus utilisées dans les programmes informatiques.

- **réduction de la taille d'un exécutable :**
le code exécutable de la bibliothèque est chargé à la volée ;
- **utilisation de fonctions validées :**
fonctionnement, sécurité, performances...
- **efficacité des mises à jour :**
correction de bug, trous de sécurité, améliorations....

Exemple sous :

- windows : fichiers .dll (pour *Dynamic Link Library*) ;
- Unix / Linux : fichiers .so (pour *shared object*).

1 Introduction

- Définition d'un système d'exploitation
- Le noyau du système
- Les bibliothèques du système
- **Outils systèmes et applications de base**

2 Principes des systèmes d'exploitation

- Notions de compte et de « droits »
- Équité et confidentialité
- Exemple d'Unix / de Linux

3 Gestion des processus

- Notion de processus
- Politiques d'ordonnancement des processus
- Manipulation des processus

Les outils du système et programmes applicatifs de base

Les **outils système** d'un OS permettent

Les outils du système et programmes applicatifs de base

Les **outils système** d'un OS permettent :

- de **configurer le système** :
 - gérer les comptes des utilisateurs ;
 - configuration des paramètres réseau ;
 - démarrage automatique des services... .

Les outils du système et programmes applicatifs de base

Les **outils système** d'un OS permettent :

- de **configurer le système** :
 - gérer les comptes des utilisateurs ;
 - configuration des paramètres réseau ;
 - démarrage automatique des services... .
- de **passer le relais** aux applications.

Les outils du système et programmes applicatifs de base

Les **outils système** d'un OS permettent :

- de **configurer le système** :
 - gérer les comptes des utilisateurs ;
 - configuration des paramètres réseau ;
 - démarrage automatique des services... .
- de **passer le relais** aux applications.

Les **applicatifs de base** ne font pas clairement partie du SE, mais les applications tierces comptent sur leur présence.

Exemples :

- éditeur, client ftp, outil d'archivage... .
- outils de visualisation (images, postscript, pdf...)
- client de courriel, navigateur web, suite bureautique... .

1 Introduction

- Définition d'un système d'exploitation
- Le noyau du système
- Les bibliothèques du système
- Outils systèmes et applications de base

2 Principes des systèmes d'exploitation

- Notions de compte et de « droits »
- Équité et confidentialité
- Exemple d'Unix / de Linux

3 Gestion des processus

- Notion de processus
- Politiques d'ordonnancement des processus
- Manipulation des processus

1 Introduction

- Définition d'un système d'exploitation
- Le noyau du système
- Les bibliothèques du système
- Outils systèmes et applications de base

2 Principes des systèmes d'exploitation

- Notions de compte et de « droits »
- **Équité et confidentialité**
- Exemple d'Unix / de Linux

3 Gestion des processus

- Notion de processus
- Politiques d'ordonnancement des processus
- Manipulation des processus

Systèmes multi-tâche et multi-utilisateur

Système multi-utilisateur

Un système multi-utilisateur est conçu pour que plusieurs utilisateurs puissent profiter des mêmes ressources **simultanément**.

Systèmes multi-tâche et multi-utilisateur

Système multi-utilisateur

Un système multi-utilisateur est conçu pour que plusieurs utilisateurs puissent profiter des mêmes ressources **simultanément**.

⇒ l'accès aux ressources et aux données est protégé par un « **compte utilisateur** » (et un mot de passe).

Systèmes multi-tâche et multi-utilisateur

Système multi-utilisateur

Un système multi-utilisateur est conçu pour que plusieurs utilisateurs puissent profiter des mêmes ressources **simultanément**.

- ⇒ l'accès aux ressources et aux données est protégé par un « **compte utilisateur** » (et un mot de passe).
 - le compte assure l'**équité** d'accès aux ressources ;

Systèmes multi-tâche et multi-utilisateur

Système multi-utilisateur

Un système multi-utilisateur est conçu pour que plusieurs utilisateurs puissent profiter des mêmes ressources **simultanément**.

- ⇒ l'accès aux ressources et aux données est protégé par un « **compte utilisateur** » (et un mot de passe).
- le compte assure l'**équité** d'accès aux ressources ;
 - le compte permet la **confidentialité** des données ;

Systèmes multi-tâche et multi-utilisateur

Système multi-utilisateur

Un système multi-utilisateur est conçu pour que plusieurs utilisateurs puissent profiter des mêmes ressources **simultanément**.

- ⇒ l'accès aux ressources et aux données est protégé par un « **compte utilisateur** » (et un mot de passe).
 - le compte assure l'**équité** d'accès aux ressources ;
 - le compte permet la **confidentialité** des données ;
 - le compte permet une bonne **protection du système**, les fichiers de configuration et les commandes sensibles ne sont accessibles qu'aux « **administrateurs** ».

1 Introduction

- Définition d'un système d'exploitation
- Le noyau du système
- Les bibliothèques du système
- Outils systèmes et applications de base

2 Principes des systèmes d'exploitation

- Notions de compte et de « droits »
- Équité et confidentialité
- Exemple d'Unix / de Linux

3 Gestion des processus

- Notion de processus
- Politiques d'ordonnancement des processus
- Manipulation des processus

Accès aux fichiers sous Unix / Linux / MacOS X / ...

Les fichiers (et répertoires) d'un système multi-utilisateur Unix :

- appartiennent tous à un utilisateur donné ;
- sont protégés par des **droits d'accès**, modifiables uniquement par le propriétaire du fichier (et par l'administrateur).

Accès aux fichiers sous Unix / Linux / MacOS X / ...

Les fichiers (et répertoires) d'un système multi-utilisateur Unix :

- appartiennent tous à un utilisateur donné ;
- sont protégés par des **droits d'accès**, modifiables uniquement par le propriétaire du fichier (et par l'administrateur).

Sous Unix, les droits possibles sont :

Accès aux fichiers sous Unix / Linux / MacOS X / ...

Les fichiers (et répertoires) d'un système multi-utilisateur Unix :

- appartiennent tous à un utilisateur donné ;
- sont protégés par des **droits d'accès**, modifiables uniquement par le propriétaire du fichier (et par l'administrateur).

Sous Unix, les droits possibles sont :

- pour un fichier (non répertoire) :
 - droit de lecture du fichier ;
 - droit d'écriture du fichier ;
 - droit d'exécution du fichier (script ou exécutable) ;

Accès aux fichiers sous Unix / Linux / MacOS X / ...

Les fichiers (et répertoires) d'un système multi-utilisateur Unix :

- appartiennent tous à un utilisateur donné ;
- sont protégés par des **droits d'accès**, modifiables uniquement par le propriétaire du fichier (et par l'administrateur).

Sous Unix, les droits possibles sont :

- pour un fichier (non répertoire) :
 - droit de lecture du fichier ;
 - droit d'écriture du fichier ;
 - droit d'exécution du fichier (script ou exécutable) ;
- pour un répertoire :
 - droit de listage du répertoire ;
 - droit d'écriture dans le répertoire,
donc le droit d'effacer ce qui s'y trouve ;
 - droit de traverser un répertoire ;

Accès aux fichiers sous Unix / Linux / MacOS X / ...

Les fichiers (et répertoires) d'un système multi-utilisateur Unix :

- appartiennent tous à un utilisateur donné ;
- sont protégés par des **droits d'accès**, modifiables uniquement par le propriétaire du fichier (et par l'administrateur).

Sous Unix, les droits possibles sont :

- pour un fichier (non répertoire) :
 - droit de lecture du fichier; noté **r**
 - droit d'écriture du fichier ;
 - droit d'exécution du fichier (script ou exécutable) ;
- pour un répertoire :
 - droit de listage du répertoire ; noté **r**
 - droit d'écriture dans le répertoire,
donc le droit d'effacer ce qui s'y trouve ;
 - droit de traverser un répertoire ;

Accès aux fichiers sous Unix / Linux / MacOS X / ...

Les fichiers (et répertoires) d'un système multi-utilisateur Unix :

- appartiennent tous à un utilisateur donné ;
- sont protégés par des **droits d'accès**, modifiables uniquement par le propriétaire du fichier (et par l'administrateur).

Sous Unix, les droits possibles sont :

- pour un fichier (non répertoire) :
 - droit de lecture du fichier; noté **r**
 - droit d'écriture du fichier; noté **w**
 - droit d'exécution du fichier (script ou exécutable);
- pour un répertoire :
 - droit de listage du répertoire; noté **r**
 - droit d'écriture dans le répertoire, noté **w**
donc le droit d'effacer ce qui s'y trouve;
 - droit de traverser un répertoire;

Accès aux fichiers sous Unix / Linux / MacOS X / ...

Les fichiers (et répertoires) d'un système multi-utilisateur Unix :

- appartiennent tous à un utilisateur donné ;
- sont protégés par des **droits d'accès**, modifiables uniquement par le propriétaire du fichier (et par l'administrateur).

Sous Unix, les droits possibles sont :

- pour un fichier (non répertoire) :
 - droit de lecture du fichier ; noté **r**
 - droit d'écriture du fichier ; noté **w**
 - droit d'exécution du fichier (script ou exécutable) ; noté **x**
- pour un répertoire :
 - droit de listage du répertoire ; noté **r**
 - droit d'écriture dans le répertoire, noté **w**
donc le droit d'effacer ce qui s'y trouve ;
 - droit de traverser un répertoire ; noté **x**

Les droits sont décrits via 9 caractères :

- 3 pour le propriétaire ;
- 3 pour un groupe d'utilisateurs ;
- 3 pour les autres utilisateurs.

Les droits sont décrits via 9 caractères :

- 3 pour le propriétaire ;
- 3 pour un groupe d'utilisateurs ;
- 3 pour les autres utilisateurs.

Exemple : si un fichier a comme droits : **rw-r--r--**

Les droits sont décrits via 9 caractères :

- 3 pour le propriétaire ;
- 3 pour un groupe d'utilisateurs ;
- 3 pour les autres utilisateurs.

Exemple : si un fichier a comme droits : ***rw-r--r--***

- le **propriétaire** peut y lire et y écrire,
mais pas exécuter le fichier ;

Les droits sont décrits via 9 caractères :

- 3 pour le propriétaire ;
- 3 pour un groupe d'utilisateurs ;
- 3 pour les autres utilisateurs.

Exemple : si un fichier a comme droits : `rw-r--r--`

- le **propriétaire** peut y lire et y écrire,
mais pas exécuter le fichier ;
- les membres du **groupe** ne peuvent que lire,
mais ni y écrire ni l'exécuter ;

Les droits sont décrits via 9 caractères :

- 3 pour le propriétaire ;
- 3 pour un groupe d'utilisateurs ;
- 3 pour les autres utilisateurs.

Exemple : si un fichier a comme droits : **rw-r--r--**

- le **propriétaire** peut y lire et y écrire,
mais pas exécuter le fichier ;
- les membres du **groupe** ne peuvent que lire,
mais ni y écrire ni l'exécuter ;
- les “**autres**” utilisateurs peuvent également lire le fichier,
mais ni y écrire ni l'exécuter.

Commande **chmod** : changer les permissions d'accès

La commande **chmod** (abréviation de *change mode*) permet de changer les permissions d'accès sur un fichier (ou un répertoire).

Commande **chmod** : changer les permissions d'accès

La commande **chmod** (abréviation de *change mode*) permet de changer les permissions d'accès sur un fichier (ou un répertoire).

Syntaxe : **chmod options modes fichiers**

Commande **chmod** : changer les permissions d'accès

La commande **chmod** (abréviation de *change mode*) permet de changer les permissions d'accès sur un fichier (ou un répertoire).

Syntaxe : **chmod options modes fichiers**

① on précise à qui s'applique la modification des droits :

- au propriétaire du fichier
- aux utilisateurs dans le groupe du fichier
- à tous les autres utilisateurs
- à tous le monde

Commande **chmod** : changer les permissions d'accès

La commande **chmod** (abréviation de *change mode*) permet de changer les permissions d'accès sur un fichier (ou un répertoire).

Syntaxe : **chmod options modes fichiers**

① on précise à qui s'applique la modification des droits :

- au propriétaire du fichier, noté **u** pour *user*;
- aux utilisateurs dans le groupe du fichier, noté **g** pour *group* ;
- à tous les autres utilisateurs, noté **o** pour *other*;
- à tous le monde, noté **a** pour *all*;

Commande **chmod** : changer les permissions d'accès

La commande **chmod** (abréviation de *change mode*) permet de changer les permissions d'accès sur un fichier (ou un répertoire).

Syntaxe : **chmod options modes fichiers**

① on précise à qui s'applique la modification des droits :

- au propriétaire du fichier, noté **u** pour *user*;
- aux utilisateurs dans le groupe du fichier, noté **g** pour *group* ;
- à tous les autres utilisateurs, noté **o** pour *other*;
- à tous le monde, noté **a** pour *all*;

② on précise comment on modifie :

- opérateurs de changement **+**
- opérateurs de changement **-**
- opérateurs de changement **=**

Commande **chmod** : changer les permissions d'accès

La commande **chmod** (abréviation de *change mode*) permet de changer les permissions d'accès sur un fichier (ou un répertoire).

Syntaxe : **chmod options modes fichiers**

① on précise à qui s'applique la modification des droits :

- au propriétaire du fichier, noté **u** pour *user*;
- aux utilisateurs dans le groupe du fichier, noté **g** pour *group* ;
- à tous les autres utilisateurs, noté **o** pour *other*;
- à tous le monde, noté **a** pour *all*;

② on précise comment on modifie :

- opérateurs de changement **+** → ajout de droits ;
- opérateurs de changement **-** → retrait de droits ;
- opérateurs de changement **=** → affectation de droits ;

Commande **chmod** : changer les permissions d'accès

La commande **chmod** (abréviation de *change mode*) permet de changer les permissions d'accès sur un fichier (ou un répertoire).

Syntaxe : **chmod options modes fichiers**

① on précise à qui s'applique la modification des droits :

- au propriétaire du fichier, noté **u** pour *user*;
- aux utilisateurs dans le groupe du fichier, noté **g** pour *group* ;
- à tous les autres utilisateurs, noté **o** pour *other*;
- à tous le monde, noté **a** pour *all*;

② on précise comment on modifie :

- opérateurs de changement **+** → ajout de droits ;
- opérateurs de changement **-** → retrait de droits ;
- opérateurs de changement **=** → affectation de droits ;

③ on précise le droit : **r**, ou **w**, ou **x**, ou toute combinaison !

Principales options de `chmod` et exemples

Option les plus utilisées :

- **-R : récursivité**
→ traite récursivement les éventuels répertoires ;
- **-v : verbeux**
→ affiche la liste de tous les fichiers en cours de modification.

Principales options de **chmod** et exemples

Option les plus utilisées :

- **-R : récursivité**
→ traite récursivement les éventuels répertoires ;
- **-v : verbeux**
→ affiche la liste de tous les fichiers en cours de modification.

Exemples :

- `chmod u+rwx mon_fichier`

Principales options de **chmod** et exemples

Option les plus utilisées :

- **-R : récursivité**
→ traite récursivement les éventuels répertoires ;
- **-v : verbeux**
→ affiche la liste de tous les fichiers en cours de modification.

Exemples :

- **chmod u+rwx mon_fichier**
donne au propriétaire les droits en lecture et écriture
sur **mon_fichier** ;

Principales options de `chmod` et exemples

Option les plus utilisées :

- **-R : récursivité**
→ traite récursivement les éventuels répertoires ;
- **-v : verbeux**
→ affiche la liste de tous les fichiers en cours de modification.

Exemples :

- `chmod u+rwx mon_fichier`
donne au **propriétaire** les droits en lecture et écriture
sur `mon_fichier` ;

Principales options de `chmod` et exemples

Option les plus utilisées :

- **-R : récursivité**
→ traite récursivement les éventuels répertoires ;
- **-v : verbeux**
→ affiche la liste de tous les fichiers en cours de modification.

Exemples :

- `chmod u+rw mon_fichier`
donne au propriétaire les droits en **lecture et écriture**
sur `mon_fichier` ;

Principales options de `chmod` et exemples

Options les plus utilisées :

- **-R : récursivité**
→ traite récursivement les éventuels répertoires ;
- **-v : verbeux**
→ affiche la liste de tous les fichiers en cours de modification.

Exemples :

- `chmod u+rwx mon_fichier`
donne au propriétaire les droits en lecture et écriture
sur `mon_fichier` ;
- `chmod -R a+rx mon_dossier`

Principales options de `chmod` et exemples

Options les plus utilisées :

- **-R : récursivité**
→ traite récursivement les éventuels répertoires ;
- **-v : verbeux**
→ affiche la liste de tous les fichiers en cours de modification.

Exemples :

- `chmod u+rwx mon_fichier`
donne au propriétaire les droits en lecture et écriture
sur `mon_fichier` ;
- `chmod -R a+rx mon_dossier`
pour tout ce qui est dans `mon_dossier` (récursivement) :
 - donne à tous les fichiers les droits en lecture et en exécution
pour tous les utilisateurs ;
 - donne à tous les répertoires les droits en listage et en traversée
à tous les utilisateurs.

Principales options de `chmod` et exemples

Options les plus utilisées :

- **-R : récursivité**
→ traite récursivement les éventuels répertoires ;
- **-v : verbeux**
→ affiche la liste de tous les fichiers en cours de modification.

Exemples :

- `chmod u+rwx mon_fichier`
donne au propriétaire les droits en lecture et écriture
sur `mon_fichier` ;
- `chmod -R a+rx mon_dossier`
pour tout ce qui est dans `mon_dossier` (récursivement) :
 - donne à tous les fichiers les droits en **lecture et en exécution**
pour tous les utilisateurs ;
 - donne à tous les répertoires les droits en **listage et en traversée**
à tous les utilisateurs.

Principales options de `chmod` et exemples

Options les plus utilisées :

- **-R : récursivité**
→ traite récursivement les éventuels répertoires ;
- **-v : verbeux**
→ affiche la liste de tous les fichiers en cours de modification.

Exemples :

- `chmod u+rwx mon_fichier`
donne au propriétaire les droits en lecture et écriture
sur `mon_fichier` ;
- `chmod -R a+rx mon_dossier`
pour tout ce qui est dans `mon_dossier` (récursivement) :
 - donne à tous les fichiers les droits en lecture et en exécution
pour **tous les utilisateurs** ;
 - donne à tous les répertoires les droits en listage et en traversée
à **tous les utilisateurs**.

1 Introduction

- Définition d'un système d'exploitation
- Le noyau du système
- Les bibliothèques du système
- Outils systèmes et applications de base

2 Principes des systèmes d'exploitation

- Notions de compte et de « droits »
- Équité et confidentialité
- Exemple d'Unix / de Linux

3 Gestion des processus

- **Notion de processus**
- Politiques d'ordonnancement des processus
- Manipulation des processus

Notion de processus

Processus = « programme en exécution »

Notion de processus

Processus = « **programme en exécution** », plus formellement :

Définition d'un processus

Un processus (en anglais, *process*), en informatique, est défini par :

- un **ensemble d'instructions à exécuter** (un programme) ;
- un **espace mémoire** pour les données de travail ;
- éventuellement, d'autres **ressources**
comme des « descripteurs de fichiers », des « ports réseau »... .

Le système gère une structure nommée « **contexte du processus** »
(ou « état du processus ») pour chaque processus en mémoire
contenant notamment l'état des registres du processeur
≡ *Process State Word* du cours d'architecture !

Notion de processus léger

La plupart des systèmes offrent la distinction entre :

- « processus lourds » (comme vu au transparent précédent), complètement isolés les uns des autres ;
- « **processus légers** » (ou *threads* en anglais), partageant des ressources (dont la mémoire) entre threads, au sein d'un processus dit « *multi-thread* ».

Notion de processus léger

La plupart des systèmes offrent la distinction entre :

- « processus lourds » (comme vu au transparent précédent), complètement isolés les uns des autres ;
- « **processus légers** » (ou *threads* en anglais), partageant des ressources (dont la mémoire) entre threads, au sein d'un processus dit « *multi-thread* ».

Intérêt des processus légers :

- plus **rapides à créer** (et détruire) qu'un processus lourd ;
- **collaboration entre threads** facile au sein d'un processus multi-thread, (sémaphores, mutex, sections critiques...).

Notion de processus léger

La plupart des systèmes offrent la distinction entre :

- « processus lourds » (comme vu au transparent précédent), complètement isolés les uns des autres ;
- « **processus légers** » (ou *threads* en anglais), partageant des ressources (dont la mémoire) entre threads, au sein d'un processus dit « *multi-thread* ».

Intérêt des processus légers :

- plus **rapides à créer** (et détruire) qu'un processus lourd ;
- **collaboration entre threads** facile au sein d'un processus multi-thread, (sémaphores, mutex, sections critiques...).

Dans le cas d'un processus *multi-thread*,
il existe un « contexte du processus » par thread.

Système multitâche

Un système d'exploitation est dit **multitâche** s'il permet d'exécuter plusieurs processus de façon apparemment simultanée.

Système multitâche

Un système d'exploitation est dit **multitâche** s'il permet d'exécuter plusieurs processus de façon apparemment simultanée.

En pratique, le système alloue le(s) processeur(s) disponible(s) aux différents processus pour de petites tranches de temps : on parle de « **temps partagé** » (en anglais « *time sharing* »).

Système multitâche

Un système d'exploitation est dit **multitâche** s'il permet d'exécuter plusieurs processus de façon apparemment simultanée.

En pratique, le système alloue le(s) processeur(s) disponible(s) aux différents processus pour de petites tranches de temps : on parle de « **temps partagé** » (en anglais « *time sharing* »).

Le multitâche est dit :

- « **coopératif** » (désuet),
si les processus se passent le processeur ;
- « **préemptif** »,
si le système confie le processeur aux processus,
et le reprend sans l'avis du processus le moment venu !

Système multitâche

Un système d'exploitation est dit **multitâche** s'il permet d'exécuter plusieurs processus de façon apparemment simultanée.

En pratique, le système alloue le(s) processeur(s) disponible(s) aux différents processus pour de petites tranches de temps : on parle de « **temps partagé** » (en anglais « *time sharing* »).

Le multitâche est dit :

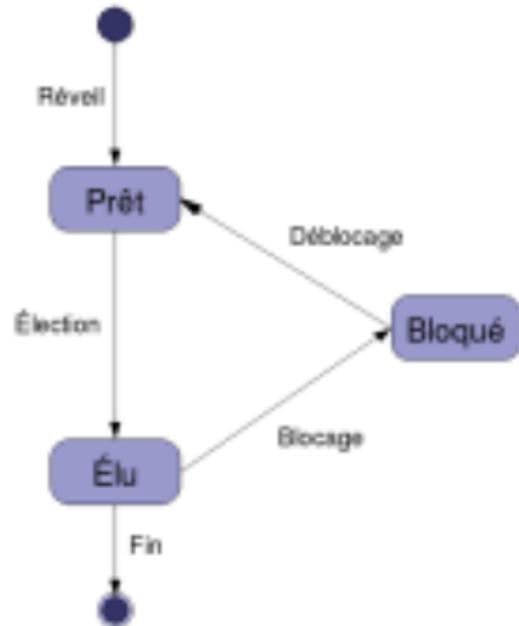
- « **coopératif** » (désuet),
si les processus se passent le processeur ;
- « **préemptif** »,
si le système confie le processeur aux processus,
et le reprend sans l'avis du processus le moment venu !

OS préemptifs : Windows **après** 3.11, MacOS depuis MacOS X,
Unix depuis que les processeurs sont préemptifs !

Principaux états d'un processus

Les **principaux états**
d'un processus sont :

- **prêt** ;



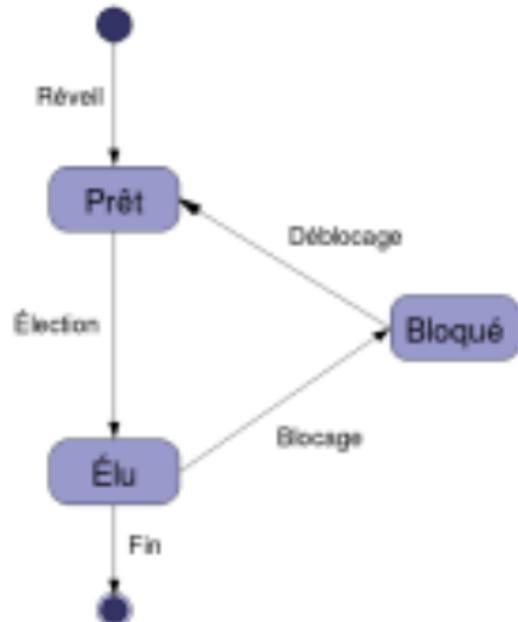
Principaux états d'un processus

Les **principaux états**
d'un processus sont :

- **prêt** ;
- **élu**, ou « en exécution » ;

Remarque

Temps partagé = prêt ⇔ élu.



Principaux états d'un processus

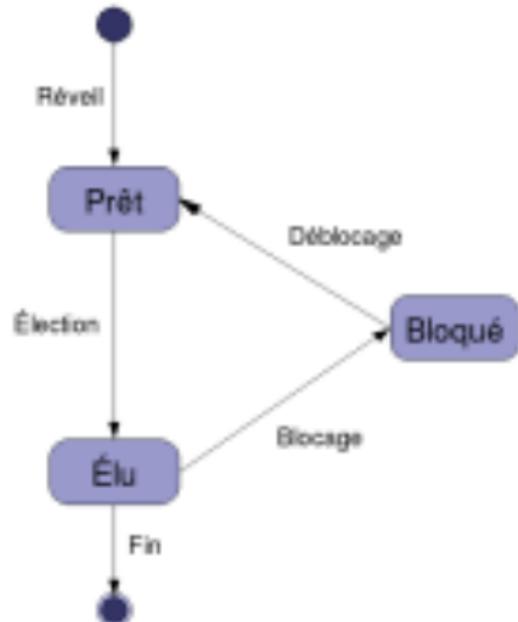
Les **principaux états**
d'un processus sont :

- **prêt** ;
- **élu**, ou « en exécution » ;
- **bloqué**,

en attente d'une ressource ;

Remarque

On a pas élu \leftrightarrow bloqué
mais élu \rightarrow bloqué \rightarrow prêt.



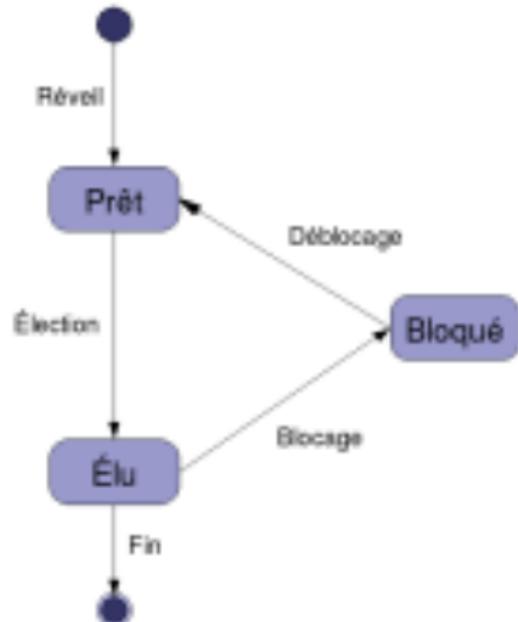
Principaux états d'un processus

Les **principaux états**
d'un processus sont :

- **prêt** ;
- **élu**, ou « en exécution » ;
- **bloqué**,
en attente d'une ressource ;
ou « **suspendu** » par
l'utilisateur (ctrl-z)
ou le système.

Remarque

On a pas élu \leftrightarrow suspendu
mais **élu** → **suspendu** → **prêt**.



1 Introduction

- Définition d'un système d'exploitation
- Le noyau du système
- Les bibliothèques du système
- Outils systèmes et applications de base

2 Principes des systèmes d'exploitation

- Notions de compte et de « droits »
- Équité et confidentialité
- Exemple d'Unix / de Linux

3 Gestion des processus

- Notion de processus
- **Politiques d'ordonnancement des processus**
- Manipulation des processus

1 Introduction

- Définition d'un système d'exploitation
- Le noyau du système
- Les bibliothèques du système
- Outils systèmes et applications de base

2 Principes des systèmes d'exploitation

- Notions de compte et de « droits »
- Équité et confidentialité
- Exemple d'Unix / de Linux

3 Gestion des processus

- Notion de processus
- Politiques d'ordonnancement des processus
- **Manipulation des processus**

Manipulation de processus depuis le shell

Il est facile de manipuler les processus depuis un interpréteur de commandes, par exemple le bash shell.

Manipulation de processus depuis le shell

Il est facile de manipuler les processus depuis un interpréteur de commandes, par exemple le bash shell.

Les commandes à connaître :

- **ps** : liste les processus (voir les options), donne notamment les **PID** (pour *Process ID*) ;
- **kill** : permet d'envoyer un signal à un processus, par exemple pour tuer un processus (signal 9) ;
- **top** : affiche un classement des processus, en fonction de leur gourmandise en temps processeur ;
- **pstree** : affiche les processus sous une forme arborescente.

Manipulation de processus depuis le shell

Il est facile de manipuler les processus depuis un interpréteur de commandes, par exemple le bash shell.

Les commandes à connaître :

- **ps** : liste les processus (voir les options), donne notamment les **PID** (pour *Process ID*) ;
- **kill** : permet d'envoyer un signal à un processus, par exemple pour tuer un processus (signal 9) ;
- **top** : affiche un classement des processus, en fonction de leur gourmandise en temps processeur ;
- **pstree** : affiche les processus sous une forme arborescente.

A faire en TP : lire le manuel (**man**) de ces commandes et tester !

Manipulation de processus depuis le C

Unix est écrit en C (et C a été créé pour écrire Unix !)
⇒ on peut bien sûr manipuler les processus en C.

Manipulation de processus depuis le C

Unix est écrit en C (et C a été créé pour écrire Unix !)
⇒ on peut bien sûr manipuler les processus en C.

Nous allons voir comment :

- créer un processus en C,
en fait « dupliquer un processus »,
puis le cas échéant remplacer le code du nouveau processus ;
- synchroniser des processus en C ;
- tuer un processus en C.

Manipulation de processus depuis le C

Unix est écrit en C (et C a été créé pour écrire Unix !)
⇒ on peut bien sûr manipuler les processus en C.

Nous allons voir comment :

- créer un processus en C,
en fait « dupliquer un processus »,
puis le cas échéant remplacer le code du nouveau processus ;
- synchroniser des processus en C ;
- tuer un processus en C.

Il restera à voir :

- comment communiquer entre processus ;
- tout ce que sont processus légers.

Création d'un processus en C

Sous Unix :

création d'un processus = **bifurcation** (ou embranchement)
soit « **fork** » en anglais.

Création d'un processus en C

Sous Unix :

création d'un processus = **bifurcation** (ou embranchement)
soit « **fork** » en anglais.

Le **processus appelant** à une création est appelé le **père**,
le **processus créé** est appelé le **fils**
⇒ forme une arborescence de processus.

Création d'un processus en C

Sous Unix :

création d'un processus = **bifurcation** (ou embranchement)
soit « **fork** » en anglais.

Le **processus appelant** à une création est appelé le **père**,
le **processus créé** est appelé le **fils**
⇒ forme une arborescence de processus.

Remarques :

- un processus n'a qu'un parent (direct), son père ;
- un processus peut avoir plusieurs fils ;
- le père de tous les processus est le processus init,
et init adopte les processus dont le père meurt.

L'appel système fork()

L'appel système **fork** (de unistd.h).

L'appel système fork()

L'appel système **fork** (de unistd.h) :

- ➊ **duplique le processus appelant,**
à l'identique pour ce qui est du code (copie),
avec les mêmes descripteurs de fichiers (copies)...

L'appel système fork()

L'appel système **fork** (de `unistd.h`) :

- ① **duplique le processus appelant**,
à l'identique pour ce qui est du code (copie),
avec les mêmes descripteurs de fichiers (copies)...
- ② **renvoie un entier** :
 - **le PID du fils si on est dans le père** ;
 - **la valeur 0 si on est dans le fils** ;
 - **la valeur -1 en cas d'échec**.

L'appel système fork()

L'appel système **fork** (de unistd.h) :

- ① **duplique le processus appelant**,
à l'identique pour ce qui est du code (copie),
avec les mêmes descripteurs de fichiers (copies)...
- ② **renvoie** un entier :
 - le PID du fils si on est dans le père ;
 - la valeur 0 si on est dans le fils ;
 - la valeur -1 en cas d'échec.

Remarque :

il y a tout de même de petites différences en père et fils,
par exemple les variables renseignant sur le temps d'exécution.

Exemple d'appel système fork()

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main (void)
{ pid_t pid;
    printf("Avant le fork") ;
    pid=fork();
    if (pid>0)
        printf("père : mon fils a pour PID %d.",pid) ;
        else printf ("fils : hello world") ;
    return 0; }
```

Autres fonctions bien utiles

Deux fonctions permettent de connaître :

- le PID du processus courant,
renvoyé par `getpid()` ;
- le PID du processus père,
renvoyé par `getppid()`.

Autres fonctions bien utiles

Deux fonctions permettent de connaître :

- le PID du processus courant,
renvoyé par `getpid()` ;
- le PID du processus père,
renvoyé par `getppid()`.

Remarques :

- ces appels réussissent toujours ;
- si le PID du père vaut 1,
le processus est fils de `init`
⇒ le processus est probablement orphelin.

Synchronisation de processus père et fils (wait)

Un père peut attendre la fin (la mort) d'un fils,
en appelant la fonction **wait** (de sys/wait.h)

Synchronisation de processus père et fils (wait)

Un père peut attendre la fin (la mort) d'un fils,
en appelant la fonction **wait** (de sys/wait.h)

⇒ suspend l'exécution du processus appelant,

- jusqu'à ce qu'un enfant se termine,
- ou jusqu'à ce qu'un signal à interceppter arrive.

Synchronisation de processus père et fils (wait)

Un père peut attendre la fin (la mort) d'un fils,
en appelant la fonction `wait` (de `sys/wait.h`)

⇒ suspend l'exécution du processus appelant,

- jusqu'à ce qu'un enfant se termine,
- ou jusqu'à ce qu'un signal à interceppter arrive.

Remarques :

- si un processus fils est déjà mort lors de l'appel (on parle de processus « zombie »),
la fonction `wait` revient immédiatement ;
- toutes les ressources utilisées par le fils sont libérées.

Synchronisation de processus père et fils (wait)

Syntaxe : `pid_t wait(int *status)`

Synchronisation de processus père et fils (wait)

Syntaxe : `pid_t wait(int *status)`

Si `status` est non NULL,
`wait` et `waitpid` y stockent des informations sur la mort du fils.

Synchronisation de processus père et fils (wait)

Syntaxe : `pid_t wait(int *status)`

Si `status` est non `NULL`,
`wait` et `waitpid` y stockent des informations sur la mort du fils.

La variable `status` peut être analysée
avec les macros suivantes (non exhaustif !) :

- **WIFEXITED(status)**,
non nul si le fils s'est terminé normalement ;
- **WEXITSTATUS(status)**,
code de retour tel qu'il a été mentionné dans l'appel `exit()`
ou dans le `return` de la routine `main`.

Synchronisation de processus père et fils (waitpid)

Il existe une variante permettant d'attendre un fils précis, en appelant la fonction **waitpid** (de sys/wait.h).

Syntaxe :

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

Synchronisation de processus père et fils (waitpid)

Il existe une variante permettant d'attendre un fils précis, en appelant la fonction **waitpid** (de sys/wait.h).

Syntaxe :

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

La valeur de **pid** peut être :

- < -1 : attendre la fin de n'importe quel fils appartenant à un groupe de processus d'ID pid ;
- -1 : attendre la fin de n'importe quel fils, soit le même comportement que wait ;
- 0 : attendre la fin de n'importe quel fils du même groupe que l'appelant ;
- > 0 : attendre la fin du processus de PID pid.

Synchronisation de processus père et fils (waitpid)

L'argument options est un OU binaire des constantes suivantes :

- WNOHANG : ne pas bloquer si aucun fils n'est mort ;
- WUNTRACED : recevoir l'information concernant également les fils bloqués si on ne l'a pas encore reçue.

Synchronisation de processus père et fils (waitpid)

L'argument options est un OU binaire des constantes suivantes :

- WNOHANG : ne pas bloquer si aucun fils n'est mort ;
- WUNTRACED : recevoir l'information concernant également les fils bloqués si on ne l'a pas encore reçue.

Retour de la fonction :

- en cas de réussite,
le PID du fils qui s'est terminé est renvoyé ;
- en cas d'échec,
-1 est renvoyé et errno contient le code d'erreur.

Exercices

Exercice 1

Écrire un programme C qui crée un nombre aléatoire de processus, chaque processus fils devra attendre un temps aléatoire compris entre 1 et 100 secondes (fonction sleep(int nb_sec)) avant de se terminer. A chaque mort, le processus père devra afficher le PID du défunt fils, jusqu'à ce que tous ses fils soient morts.

Exercices

Exercice 1

Écrire un programme C qui crée un nombre aléatoire de processus, chaque processus fils devra attendre un temps aléatoire compris entre 1 et 100 secondes (fonction sleep(int nb_sec)) avant de se terminer. A chaque mort, le processus père devra afficher le PID du défunt fils, jusqu'à ce que tous ses fils soient morts.

Exercice 2

Ecrire un programme C qui crée 10 processus fils, en attendant la mort d'un fils créé pour en créer un nouveau.

Différenciation du père et du fils

Si on en reste à ce qu'on a vu,
init devrait contenir tous les exécutables du système,
même les applications !

Différenciation du père et du fils

Si on en reste à ce qu'on a vu,
init devrait contenir tous les exécutables du système,
même les applications !

⇒ il est possible de charger un nouveau code.

Différenciation du père et du fils

Si on en reste à ce qu'on a vu,
init devrait contenir tous les exécutables du système,
même les applications !

⇒ il est possible de charger un nouveau code.

La famille de fonctions **exec** (de `unistd.h`)
charge un fichier dans la zone de code du processus qui l'appelle,
remplaçant ainsi le code courant !

La famille exec au complet

```
→ int exec1 (const char *path,  
              const char *arg, ... );  
→ int execlp (const char *file,  
              const char *arg, ...);  
→ int execle (const char *path,  
              const char *arg, ...,  
              char * const envp[] );  
→ int execv (const char *path, char *const argv[] );  
→ int execvp (const char *file, char *const argv[] );
```

Détaillons les arguments de ces fonctions...

La famille exec au complet

```
→ int execl (const char *path,  
                const char *arg, ... );  
→ int execlp (const char *file,  
                const char *arg, ...);  
→ int execle (const char *path,  
                const char *arg, ...,  
                char * const envp[] );  
→ int execv (const char *path, char *const argv[] );  
→ int execvp (const char *file, char *const argv[] );
```

Le **chemin du nouvel exécutable**
pour **execl**, **execle** et **execv**.

La famille exec au complet

```
→ int exec1 (const char *path,  
              const char *arg, ... );  
→ int execlp (const char *file,  
              const char *arg, ...);  
→ int execle (const char *path,  
              const char *arg, ...,  
              char * const envp[] );  
→ int execv (const char *path, char *const argv[] );  
→ int execvp (const char *file, char *const argv[] );
```

Simplement le **nom** du nouvel exécutable
pour **execlp** et **execvp**,
auquel cas on fait « comme le shell » pour la recherche de l'exécutable.

La famille exec au complet

```
→ int exec1 (const char *path,  
                const char *arg, ... );  
→ int execlp (const char *file,  
                const char *arg, ...);  
→ int execle (const char *path,  
                const char *arg, ...,  
                char * const envp[] );  
→ int execv (const char *path, char *const argv[] );  
→ int execvp (const char *file, char *const argv[] );
```

Les arguments à passer au nouvel exécutable,
pour **exec1**, **execlp** et **execle** :
liste de pointeurs sur chaînes se terminant par NULL.

La famille exec au complet

```
→ int exec1 (const char *path,  
              const char *arg, ... );  
→ int execlp (const char *file,  
              const char *arg, ...);  
→ int execle (const char *path,  
              const char *arg, ...,  
              char * const envp[] );  
→ int execv (const char *path, char *const argv[] );  
→ int execvp (const char *file, char *const argv[] );
```

Les arguments à passer au nouvel exécutable,

pour execv et execvp :

un tableau de pointeurs sur des chaînes de caractères
dont le dernier pointeur est NULL.

La famille exec au complet

```
→ int exec1 (const char *path,  
                const char *arg, ... );  
→ int execlp (const char *file,  
                const char *arg, ...);  
→ int execle (const char *path,  
                const char *arg, ...,  
                char * const envp[] );  
→ int execv (const char *path, char *const argv[] );  
→ int execvp (const char *file, char *const argv[] );
```

Un **environnement spécifique** à passer au nouvel exécutable,
pour **execle** :
un tableau de pointeurs sur des chaînes de caractères
dont le dernier pointeur est NULL.

La famille exec au complet

```
→ int exec1 (const char *path,  
                const char *arg, ... );  
→ int execlp (const char *file,  
                const char *arg, ...);  
→ int execle (const char *path,  
                const char *arg, ...,  
                char * const envp[] );  
→ int execv (const char *path, char *const argv[] );  
→ int execvp (const char *file, char *const argv[] );
```

Mise à part **execle**,
les fonctions fournissent au nouvel exécutable
l'environnement constitué par la variable externe **environ** :
extern char **environ ;

Le poids des conventions par l'exemple

Prenons un exemple :

```
#include <stdio.h>
#include <unistd.h>

int main (void)
{ execl ("/bin/ls", "ls", "-l", NULL) ; return 0 ; }
```

Le poids des conventions par l'exemple

Prenons un exemple :

```
#include <stdio.h>
#include <unistd.h>

int main (void)
{ execl ("/bin/ls", "ls", "-l", NULL) ; return 0 ; }
```

⇒ fait ce que ferait un ls -l en ligne de commandes.

Le poids des conventions par l'exemple

Prenons un exemple :

```
#include <stdio.h>
#include <unistd.h>

int main (void)
{ execl ("/bin/ls", "ls", "-l", NULL) ; return 0 ; }
```

⇒ fait ce que ferait un ls -l en ligne de commandes.

Détaillons :

- ① chemin du nouvel exécutable ;

Le poids des conventions par l'exemple

Prenons un exemple :

```
#include <stdio.h>
#include <unistd.h>

int main (void)
{ execl ("/bin/ls", "ls", "-l", NULL) ; return 0 ; }
```

⇒ fait ce que ferait un ls -l en ligne de commandes.

Détaillons :

- ① chemin du nouvel exécutable ;
- ② les arguments passés au nouvel exécutables :
 - ① le **nom de l'exécutable** (par convention !) ;
 - ② les argument(s) à passer, ici un seul ;
 - ③ le pointeur NULL.

Le poids des conventions par l'exemple

Prenons un exemple :

```
#include <stdio.h>
#include <unistd.h>

int main (void)
{ execl ("/bin/ls", "ls", "-l", NULL) ; return 0 ; }
```

⇒ fait ce que ferait un ls -l en ligne de commandes.

Détaillons :

- ① chemin du nouvel exécutable ;
- ② les arguments passés au nouvel exécutables :
 - ① le nom de l'exécutable (par convention !) ;
 - ② les **argument(s)** à passer, ici un seul ;
 - ③ le pointeur NULL.

Le poids des conventions par l'exemple

Prenons un exemple :

```
#include <stdio.h>
#include <unistd.h>

int main (void)
{ execl ("/bin/ls", "ls", "-l", NULL) ; return 0 ; }
```

⇒ fait ce que ferait un ls -l en ligne de commandes.

Détaillons :

- ① chemin du nouvel exécutable ;
- ② les arguments passés au nouvel exécutables :
 - ① le nom de l'exécutable (par convention !) ;
 - ② les argument(s) à passer, ici un seul ;
 - ③ le **pointeur NULL**.

Valeur renvoyée par les fonctions exec

Par définition, **si il y a retour, il y a erreur !** et le retour sera -1
⇒ voir *errno* pour le code d'erreur.

Valeur renvoyée par les fonctions exec

Par définition, **si il y a retour, il y a erreur !** et le retour sera -1
⇒ voir *errno* pour le code d'erreur.

Comportement « exotique » :

- pour `execlp` et `execvp`,
si aucun PATH n'est défini ⇒ on prend « /bin :/usr/bin »;
- si l'en-tête d'un fichier n'est pas reconnu,
les fonctions (toutes) exécuteront un shell
avec le chemin d'accès au fichier comme premier argument
(en cas d'échec, abandon).

Accès aux arguments passés lors de l'appel

Des arguments peuvent être passés au processus :

- via l'interpréteur, par exemple : ls -l /home ;
- via les fonctions exec.

Accès aux arguments passés lors de l'appel

Des arguments peuvent être passés au processus :

- via l'interpréteur, par exemple : ls -l /home ;
- via les fonctions exec.

Considérons l'entrée `int main(int argc, char *argv[])` :

- `argc` = nombre d'arguments de l'appel du programme;
- `argv[]` = tableau des arguments passés au programme.

Accès aux arguments passés lors de l'appel

Des arguments peuvent être passés au processus :

- via l'interpréteur, par exemple : `ls -l /home`;
- via les fonctions exec.

Considérons l'entrée `int main(int argc, char *argv[])` :

- `argc` = nombre d'arguments de l'appel du programme;
- `argv[]` = tableau des arguments passés au programme.

Attention, `argv[0]` correspond au nom du programme,
avec l'exemple ci-dessus (`ls`), on a :

- `argv[0] = "ls";`
- `argv[1] = "-l";`
- `argv[2] = "/home";`

Intéraction avec l'environnement

La variable `environ` et la fonction `getenv2` (de `stdlib.h`) sont utilisées pour accéder aux valeurs définies dans l'environnement.

Déclaration (syntaxe) :

- `extern char **environ ;`
- `char *getenv (const char *name) ;`

Intéraction avec l'environnement

La variable `environ` et la fonction `getenv2` (de `stdlib.h`) sont utilisées pour accéder aux valeurs définies dans l'environnement.

Déclaration (syntaxe) :

- `extern char **environ ;`
- `char *getenv (const char *name) ;`

La fonction `getenv()` :

- recherche dans la liste des variables d'environnement une chaîne correspondant à celle pointée par `name`, les chaînes sont de la forme `nom=valeur` ;
- renvoie un pointeur sur la valeur correspondante, dans l'environnement du processus, ou `NULL` si pas de correspondance.

Intéraction avec l'environnement (suite)

Les fonctions `putenv` et `setenv` (de `stdlib.h`) permettent de « positionner » des variables d'environnement :

```
→ int putenv (const char *string) ;  
→ int setenv (const char *name, const char *value,  
               int overwrite) ;
```

Intéraction avec l'environnement (suite)

Les fonctions **putenv** et **setenv** (de `stdlib.h`) permettent de « positionner » des variables d'environnement :

```
→ int putenv (const char *string) ;  
→ int setenv (const char *name, const char *value,  
               int overwrite) ;
```

La fonction **putenv()**

- ① ajoute ou modifie la valeur d'une variable d'environnement, `string` est de la forme `nom=valeur`,
 - si `nom` n'existe pas, la variable est ajoutée ;
 - si `nom` existe, la variable est modifiée à `valeur`.

Intéraction avec l'environnement (suite)

Les fonctions **putenv** et **setenv** (de `stdlib.h`) permettent de « positionner » des variables d'environnement :

```
→ int putenv (const char *string) ;  
→ int setenv (const char *name, const char *value,  
               int overwrite) ;
```

La fonction **putenv()**

- ① ajoute ou modifie la valeur d'une variable d'environnement, `string` est de la forme `nom=valeur`,
 - si `nom` n'existe pas, la variable est ajoutée ;
 - si `nom` existe, la variable est modifiée à `valeur`.
- ② renvoie :
 - -1 en cas d'échec (pas assez de mémoire) ;
 - 0 en cas de succès.

Intéraction avec l'environnement (suite)

Les fonctions `putenv` et `setenv` (de `stdlib.h`) permettent de « positionner » des variables d'environnement :

```
→ int putenv (const char *string) ;  
→ int setenv (const char *name, const char *value,  
               int overwrite) ;
```

La fonction `setenv()`

- ① ajoute la variable `name` dans l'environnement, en lui attribuant la valeur `value` ; si `name` existe déjà dans l'environnement, alors sa valeur est modifiée en `value` si `overwrite` $\neq 0$;

Intéraction avec l'environnement (suite)

Les fonctions `putenv` et `setenv` (de `stdlib.h`) permettent de « positionner » des variables d'environnement :

```
→ int putenv (const char *string) ;  
→ int setenv (const char *name, const char *value,  
               int overwrite) ;
```

La fonction `setenv()`

- ① ajoute la variable `name` dans l'environnement, en lui attribuant la valeur `value` ;
si `name` existe déjà dans l'environnement, alors sa valeur est modifiée en `value` si `overwrite` $\neq 0$;
- ② renvoie :
 - -1 en cas d'échec (pas assez de mémoire) ;
 - 0 en cas de succès.

Navigation dans le système de fichiers

Les références relatives utilisées par un processus sont liées à son « **répertoire de travail** ».

Navigation dans le système de fichiers

Les références relatives utilisées par un processus sont liées à son « **répertoire de travail** » :

- le répertoire initial est le répertoire à partir duquel le processus a été exécuté ;
- il est possible de modifier ce répertoire de travail, soit de « se déplacer dans le système de fichiers » ;
- il est possible de connaître le répertoire de travail courant.

Navigation dans le système de fichiers

Les références relatives utilisées par un processus sont liées à son « **répertoire de travail** » :

- le répertoire initial est le répertoire à partir duquel le processus a été exécuté ;
- il est possible de modifier ce répertoire de travail, soit de « **se déplacer dans le système de fichiers** » ;
- il est possible de connaître le répertoire de travail courant.

La fonction **chdir** :

- remplace le répertoire courant par celui indiqué dans le chemin path :
`int chdir(const char *path);`
- renvoie 0 en cas de succès,
renvoie -1 en cas d'échec.

Navigation dans le système de fichiers

Les références relatives utilisées par un processus sont liées à son « **répertoire de travail** » :

- le répertoire initial est le répertoire à partir duquel le processus a été exécuté ;
- il est possible de modifier ce répertoire de travail, soit de « se déplacer dans le système de fichiers » ;
- il est possible de **connaître le répertoire de travail courant**.

La fonction `getcwd()` :

La fonction `getcwd()` :

- copie le chemin d'accès absolu du répertoire de travail courant dans la chaîne pointée (de longueur `longueur`) :

```
long *getcwd ( char *tampon,  
                unsigned long longueur );
```

La fonction `getcwd()` :

- copie le chemin d'accès absolu du répertoire de travail courant dans la chaîne pointée (de longueur `longueur`) :

```
long *getcwd ( char *tampon,  
                unsigned long longueur );
```

- renvoie :
 - NULL, si le chemin du répertoire en cours nécessite un tampon plus long que `longueur` octets ;
 - une valeur négative en cas d'échec, par exemple si le répertoire en cours n'est pas lisible.