

Programmation Système : introduction, généralités...

V. Risch

IUT, Aix-Marseille Univ., 2018, révision 2022

Remerciements à A. Dragut, J.-L. Massat, Alain Casali

Plan général du cours

Idée : parcourir les principes qui sous-tendent un système d'exploitation de type UNIX/LINUX au travers des possibilités existantes de Programmation Système...

Plan général du cours

Idée : parcourir les principes qui sous-tendent un système d'exploitation de type UNIX/LINUX au travers des possibilités existantes de Programmation Système...

Plan

- ① Cette introduction
- ② Système de gestion de fichiers
- ③ Signaux
- ④ Processus
- ⑤ IPC
- ⑥ ...

Plan général du cours

Idée : parcourir les principes qui sous-tendent un système d'exploitation de type UNIX/LINUX au travers des possibilités existantes de Programmation Système...

Plan

- ① Cette introduction
 - Catégories de systèmes
 - UNIX/LINUX, généralités
 - Le principe de la commutation de contexte
 - Commandes UNIX/LINUX
 - Outils pour les TPs
- ② Système de gestion de fichiers
- ③ Signaux
- ④ Processus
- ⑤ IPC
- ⑥ ...

Bibliographie

- Richard W. Stevens. *Advanced Programming in the Unix Environment*. Addison-Wesley, 2013.
- Michael Kerrisk. *The Linux Programming Interface : A Linux and UNIX System Programming Handbook.*, No Starch Press, 2010
- Graham Glass. *Unix for Programmer and Users*. Prentice Hall, 1993.
- Joëlle Delacroix. *Linux, Programmation système et réseau*. Dunod, 2003

Catégories de systèmes...

Définition... ?

Pas de définition universellement acceptée... Globalement, un *système d'exploitation* est un ensemble de programmes qui sert à

- *faire fonctionner le matériel*

- *maintenir un espace virtuel*

Définition... ?

Pas de définition universellement acceptée... Globalement, un *système d'exploitation* est un ensemble de programmes qui sert à

- *faire fonctionner le matériel*
 - comme « allocateur » de ressources
 - comme gestionnaire de ressources avec un minimum de sécurité
- *maintenir un espace virtuel*

Définition... ?

Pas de définition universellement acceptée... Globalement, un *système d'exploitation* est un ensemble de programmes qui sert à

- *faire fonctionner le matériel*
 - comme « allocateur » de ressources
 - comme gestionnaire de ressources avec un minimum de sécurité
 - arbitrer dans les cas conflictuels
 - superviser/contrôler l'exécution des programmes
 - prévenir les erreurs et toute utilisation non-conforme
- *maintenir un espace virtuel*

Catégories

On distingue

- SE « classiques »
- SE spécialisés
 - industriels - ex : gestion de chaînes de production
 - applications dédiées - ex : caisses enregistreuses
- servant à l'interrogation de grandes bases de données (bibliothèques, banques, hôpitaux,...)
- transactionnels (réservations aériennes, ...)

Catégories...

- embarqués (borne SNCF, distributeur de billets, terminal d'encaissement...)
- miniaturisés (téléphone cellulaire, lecteur MP3, ...)
- opérant en conditions extrêmes (plateformes pétrolières, sous-marins robotisés, ...)
- extrêmement robustes et sécurisés (centrales nucléaires, TGV, postes de commande de base militaire, objet volants à peu près identifiés, ...)

Suites logicielles

Les SE sont composés de suites de logiciels pour

- *effectuer/coordonner/sécuriser* l'accès aux ressources partagées (CPU, mémoire, ...)
- *piloter* des périphériques
- *organiser et retrouver* les données en mémoire de masse
- réaliser une bonne *interaction* homme-machine (interfaces graphiques, texte, ...) et machine-machine (réseaux)

Composantes d'un SE

Les principales composantes d'un SE sont :

- le *noyau* - partie centrale du SE contrôlant la plupart des ressources importantes
- des *modules* - étendant les fonctionnalités du noyau

D'autres composantes :

- *shells* - permettant le dialogue humain-SE
- systèmes graphiques d'affichage
- gestionnaires de fenêtres

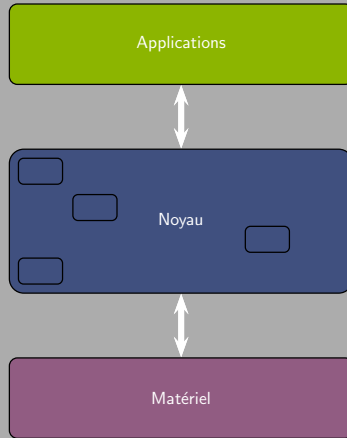
Architectures de noyau

Il existe différentes architectures possibles :

- *noyaux monolithiques*
 - non-modulaires
 - modulaires
- *micro-noyaux*
- *hybrides*
- *exo-noyaux*

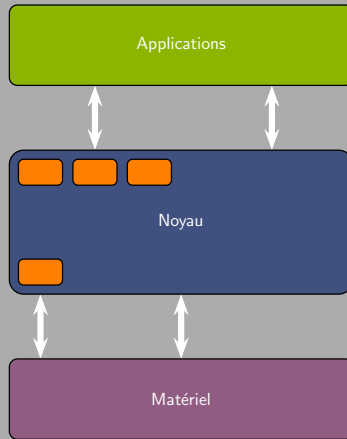
Architecture monolithique non-modulaire

La totalité du noyau réside en permanence en mémoire vive



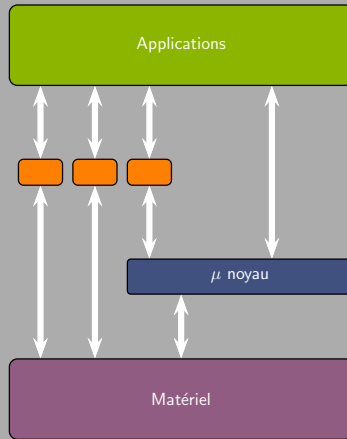
Architecture monolithique modulaire

Les modules, séparés, sont chargés dynamiquement



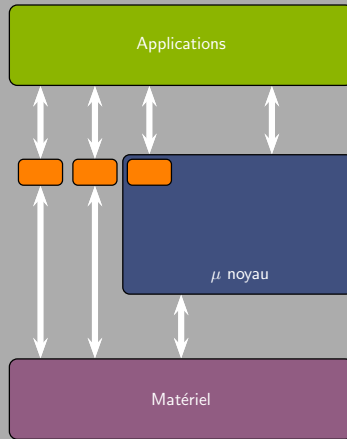
Micro-noyau

Seules les fonctions fondamentales sont conservées dans le noyau...



Noyau hybride

Tentative de compromis entre légèreté et efficacité



UNIX/LINUX, généralités

Historique

1964 Projet *Multics* de S.E. multi-tâche (MIT, General Electric, Bell Labs d'AT&T)

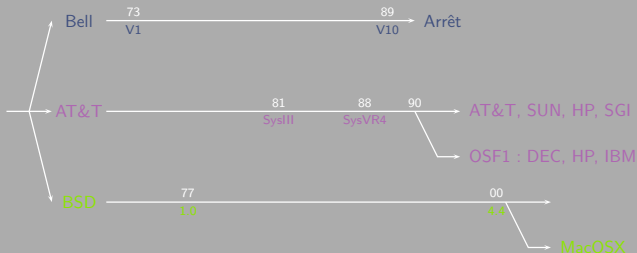
1969 Ken Thompson et Denis Ritchie (Bell Labs) écrivent un S.E. multi-tâche sur un ordi de récupération (DEC PDP 7 de 1964)

1970-71 V1 : Réécriture sur PDP11 (16 bits, 24K Ram, 512K DD) Ken Thomson crée le langage B (inspiré de *BCPL*) et réécrit Unix

1970-71 Denis Ritchie crée le langage C (inspiré de *B*) V2 : Ken Thomson réécrit entièrement Unix en C ;
les sources sont fournies à : Bell ; AT&T ; Univ. Californie à Berkeley

→ 3 branches principales de développement.

Branches principales d'UNIX



Historique complet et documents, voir <http://www.levenez.com>

Ces systèmes sont *propriétaires*, et *chers* d'où l'émergence d'un mouvement pour les logiciels *libres* (Richard Stallman) :

- Free Software Foundation
- Projet GNU → réécriture d'UNIX
- Licences GPL, CC, ...

Le 25 août 1991, Linus Thorvald publie le message suivant sur `comp.os.minix` :

« Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

Linus (torvalds@kruuna.helsinki.fi)

PS. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT portable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-).

– Linus Torvalds »

GNU/LINUX

GNU

LINUX

GNU/LINUX

GNU

→ GNU is Not Unix

LINUX

→ LINUX Is Not Unix

GNU/LINUX

GNU

→ GNU is Not Unix

LINUX

→ LINUX Is Not Unix

GNU/LINUX

GNU

→ GNU is Not Unix
→ GNU is Not Unix

LINUX

→ LINUX Is Not Unix
→ LINUX Is Not Unix

GNU/LINUX

GNU

→ GNU is Not Unix
→ GNU is Not Unix

LINUX

→ LINUX Is Not Unix
→ LINUX Is Not Unix

GNU/LINUX

GNU

- GNU is Not Unix
- GNU is Not Unix
- GNU is Not Unix

LINUX

- LINUX Is Not Unix
- LINUX Is Not Unix
- LINUX Is Not Unix

GNU/LINUX

GNU

- GNU is Not Unix
 - GNU is Not Unix
 - GNU is Not Unix

LINUX

- LINUX Is Not Unix
 - LINUX Is Not Unix
 - LINUX Is Not Unix

GNU/LINUX

GNU

- GNU is Not Unix
 - GNU is Not Unix
 - GNU is Not Unix
 - GNU is Not Unix

LINUX

- LINUX Is Not UniX
 - LINUX Is Not UniX
 - LINUX Is Not UniX
 - LINUX Is Not UniX

GNU/LINUX

GNU

- GNU is Not Unix
 - GNU is Not Unix
 - GNU is Not Unix
 - GNU is Not Unix

LINUX

- LINUX Is Not UniX
 - LINUX Is Not UniX
 - LINUX Is Not UniX
 - LINUX Is Not UniX

GNU/LINUX

GNU

- GNU is Not Unix
 - GNU is Not Unix
 - GNU is Not Unix
 - GNU is Not Unix

LINUX

- LINUX Is Not Unix
 - LINUX Is Not Unix
 - LINUX Is Not Unix
 - LINUX Is Not Unix

GNU/LINUX

GNU

- GNU is Not Unix
 - GNU is Not Unix
 - GNU is Not Unix
 - GNU is Not Unix

LINUX

- LINUX Is Not Unix
 - LINUX Is Not Unix
 - LINUX Is Not Unix
 - LINUX Is Not Unix

GNU/LINUX

GNU

- GNU is Not Unix
 - GNU is Not Unix
 - GNU is Not Unix
 - GNU is Not Unix
 - GNU is Not Unix
 - GNU is Not Unix...

LINUX

- LINUX Is Not UniX
 - LINUX Is Not UniX
 - LINUX Is Not UniX
 - LINUX Is Not UniX
 - LINUX Is Not UniX
 - LINUX Is Not UniX...

GNU/LINUX

GNU

- GNU is Not Unix
 - GNU is Not Unix
 - GNU is Not Unix
 - GNU is Not Unix
 - GNU is Not Unix
 - GNU is Not Unix...

LINUX

- LINUX Is Not UniX
 - LINUX Is Not UniX
 - LINUX Is Not UniX
 - LINUX Is Not UniX
 - LINUX Is Not UniX
 - LINUX Is Not UniX...

Le noyau LINUX

- Le noyau LINUX contient 27,8 millions de lignes de code en 2022 : il s'agit du projet libre qui en contient le plus. La première version du noyau en 1994 en contenait 176 250.
- Le développement du noyau est très actif : une nouvelle version est publiée toutes les 9 à 10 semaines.
- 95% des lignes de code du noyau sont écrites en C.
- 100% des 500 ordinateurs les plus rapides au monde fonctionnent sous LINUX.
- En 2022, LINUX est le noyau du système d'exploitation le plus utilisé au monde via les 3 milliards d'utilisateurs d'Android.

Pourquoi UNIX ?

UNIX :

- hérite des concepts SE généralistes précédents
- notion de noyau : partition virtuelle de la mémoire vive
- innovant par :
 - une simplification maximum en un ensemble d'éléments primitifs
 - la définition de leurs relations avec un ensemble réduit de règles
- but : avoir un schéma facile à maîtriser
- écrit en C, ouvert, et doté d'une aide en ligne extrêmement compréhensive, le **manuel**

Aide en ligne – le man

Section	Type de commandes
1	commandes et applications utilisateur
2	appels système, codes erreurs noyau
3	fonctions des bibliothèques
4	pilotes de périphériques et protocoles réseau
5	formats de fichiers standard
6	jeux et démos
7	divers fichiers et documents
8	commandes d'administration système
9	divers specs noyau et interfaces

- *man -k*
- plus d'informations : *man man*

Philosophie UNIX/LINUX

L'ordinateur exécute des programmes qui transforment des données, le tout reposant en mémoire...

Philosophie UNIX/LINUX

L'ordinateur exécute des programmes qui transforment des données, le tout reposant en mémoire...

- on appelle *processus* toute occurrence d'une exécution de programme

Philosophie UNIX/LINUX

L'ordinateur exécute des programmes qui transforment des données, le tout reposant en mémoire...

- on appelle *processus* toute occurrence d'une exécution de programme
- un *fichier* représente tout flux de données

Philosophie UNIX/LINUX

L'ordinateur exécute des programmes qui transforment des données, le tout reposant en mémoire...

- on appelle *processus* toute occurrence d'une exécution de programme
- un *fichier* représente tout flux de données

Le SE gère ces processus, leur permettant de « vivre heureux et s'épanouir dans un certain espace » ; ces processus consomment et/ou génèrent des données

Processus

On étudiera les processus pour comprendre comment ils

- sont gérés
- naissent, vivent, meurent et sont inhumés
- sont organisés ensemble au sein de familles
- sont pilotés par le système au travers de *signaux*
- communiquent grâce aux différentes possibilités d'*IPC*

Fichier

- Un fichier correspond à une collection de données. A ce titre, il
- est un point d'entrée/sortie système

Fichier

Un fichier correspond à une collection de données. A ce titre, il

- est un point d'entrée/sortie système
- correspond à une collection de données reposant *par exemple* en mémoire de masse

Fichier

Un fichier correspond à une collection de données. A ce titre, il

- est un point d'entrée/sortie système
- correspond à une collection de données reposant *par exemple* en mémoire de masse
- peut être sans taille, s'il est

Fichier

Un fichier correspond à une collection de données. A ce titre, il

- est un point d'entrée/sortie système
- correspond à une collection de données reposant *par exemple* en mémoire de masse
- peut être sans taille, s'il est
 - un *flot* de données (pipes, sockets...)

Fichier

Un fichier correspond à une collection de données. A ce titre, il

- est un point d'entrée/sortie système
- correspond à une collection de données reposant *par exemple* en mémoire de masse
- peut être sans taille, s'il est
 - un *flot* de données (pipes, sockets...)
 - associé à un *périphérique* permettant l'accès aux composantes matérielles ; par exemple `/dev/sda` est un accès au disque dur

```
brw-r-----1 root disk 8, 0 Oct 15 21:42 /dev/sda
brw-r-----1 root disk 8, 1 Oct 15 21:42 /dev/sda1
brw-r-----1 root disk 8, 2 Oct 15 21:42 /dev/sda2
brw-r-----1 root disk 8, 3 Oct 15 21:42 /dev/sda3
```


Fichier

Un fichier correspond à une collection de données. A ce titre, il

- est un point d'entrée/sortie système
- correspond à une collection de données reposant *par exemple* en mémoire de masse
- peut être sans taille, s'il est
 - un *flot* de données (pipes, sockets...)
 - associé à un *périphérique* permettant l'accès aux composantes matérielles ; par exemple `/dev/sda` est un accès au disque dur

```
brw-r-----1 root disk 8, 0 Oct 15 21:42 /dev/sda
brw-r-----1 root disk 8, 1 Oct 15 21:42 /dev/sda1
brw-r-----1 root disk 8, 2 Oct 15 21:42 /dev/sda2
brw-r-----1 root disk 8, 3 Oct 15 21:42 /dev/sda3
```

num majeur (type de périphérique : 8 est un port IDE)

Fichier

Un fichier correspond à une collection de données. A ce titre, il

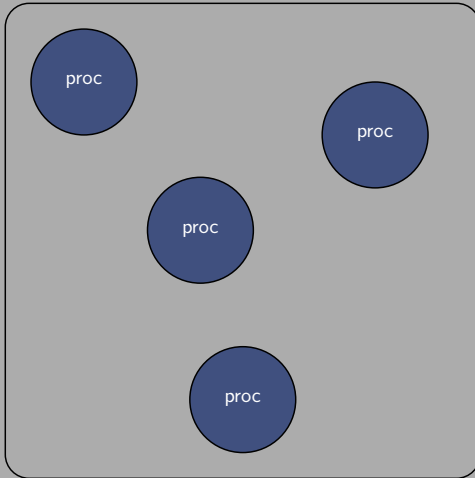
- est un point d'entrée/sortie système
- correspond à une collection de données reposant *par exemple* en mémoire de masse
- peut être sans taille, s'il est
 - un *flot* de données (pipes, sockets...)
 - associé à un *périphérique* permettant l'accès aux composantes matérielles ; par exemple /dev/sda est un accès au disque dur

```
brw-r-----1 root disk 8, 0 Oct 15 21:42 /dev/sda
brw-r-----1 root disk 8, 1 Oct 15 21:42 /dev/sda1
brw-r-----1 root disk 8, 2 Oct 15 21:42 /dev/sda2
brw-r-----1 root disk 8, 3 Oct 15 21:42 /dev/sda3
```

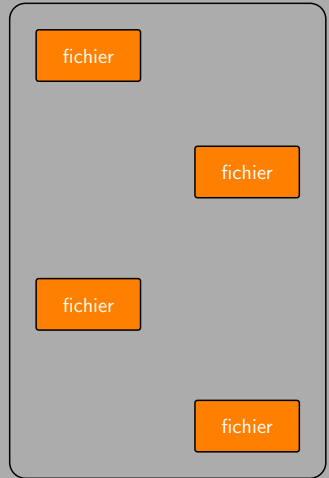
num majeur (type de périphérique : 8 est un port IDE)

num mineur (partition logique du périphérique)

Interaction Processus – Fichiers

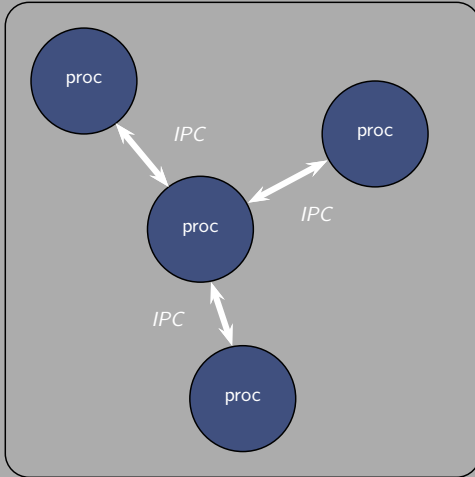


Ensemble de processus

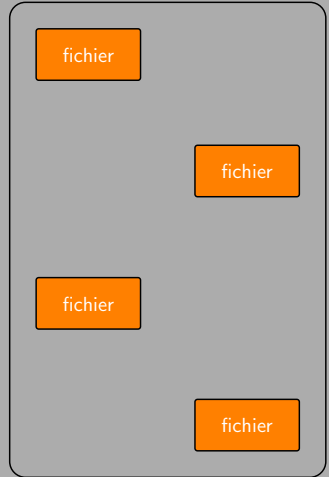


Système de fichiers

Interaction Processus – Fichiers

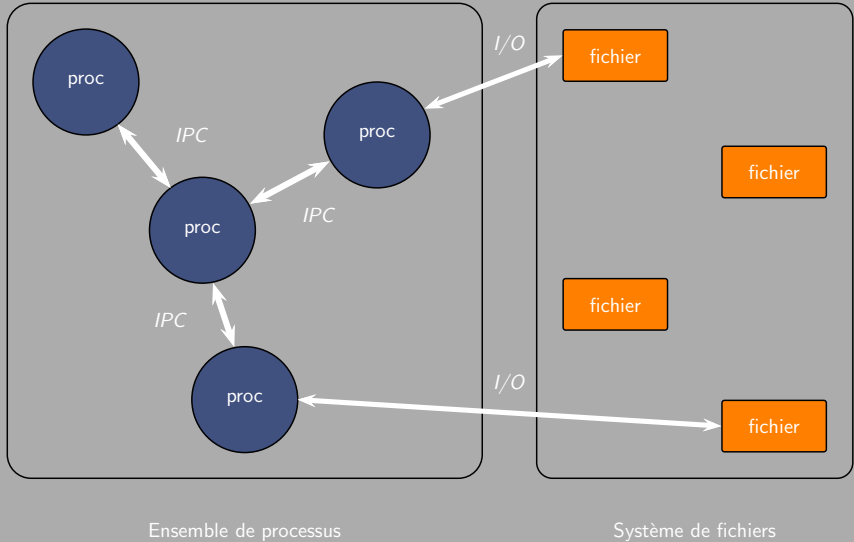


Ensemble de processus

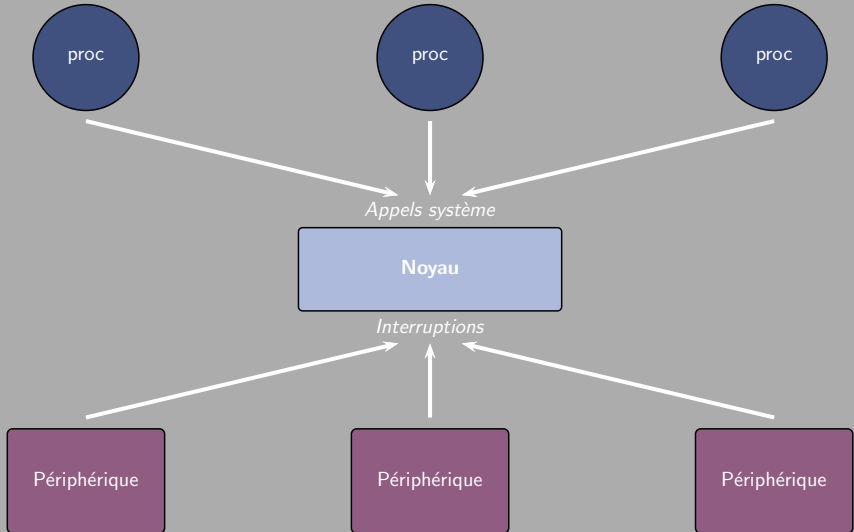


Système de fichiers

Interaction Processus – Fichiers



Interaction Processus – Fichiers : implémentation



Appels système et commutation de contexte

Modes d'exécution

- Pour les processus, le noyau fait office de gestionnaire capable de dispenser des *services*

Modes d'exécution

- Pour les processus, le noyau fait office de gestionnaire capable de dispenser des *services*
- Un processus s'exécute par défaut en *mode utilisateur* : les actions entreprises par le programme sont limitées pour des raisons de sécurité

Modes d'exécution

- Pour les processus, le noyau fait office de gestionnaire capable de dispenser des *services*
- Un processus s'exécute par défaut en *mode utilisateur* : les actions entreprises par le programme sont limitées pour des raisons de sécurité
- Lorsque un processus souhaite accéder à un service, ou qu'une *interruption* se produit, un *appel système* a lieu pour exécuter une routine du noyau en *mode superviseur* (ou encore, *mode noyau*) : dans ce mode, il n'y a aucune restriction de droits

Modes d'exécution

- Pour les processus, le noyau fait office de gestionnaire capable de dispenser des *services*
- Un processus s'exécute par défaut en *mode utilisateur* : les actions entreprises par le programme sont limitées pour des raisons de sécurité
- Lorsque un processus souhaite accéder à un service, ou qu'une *interruption* se produit, un *appel système* a lieu pour exécuter une routine du noyau en *mode superviseur* (ou encore, *mode noyau*) : dans ce mode, il n'y a aucune restriction de droits
- Le passage d'un mode à l'autre s'appelle *commutation de contexte* : il s'accompagne d'une opération de *sauvegarde* du contexte utilisateur en passant en mode superviseur, et d'une *restauration* de ce contexte en repassant en mode utilisateur

Processeur et commutation de contexte

Le rôle de la *commutation de contexte* est d'assurer cohérence et protection dans un cadre multiprocesseur – multiutilisateur

- Le SE s'appuie généralement sur les niveaux de privilège définis au niveau des processeurs
- Ces niveaux attribuent aux objets connus par le processus (segments, tables, ...) une valeur qui définit des règles d'accès
- Dépendent de la commutation de contexte interruptions matérielles et logicielles ainsi qu'exceptions...

Processeur et commutation de contexte

Le rôle de la *commutation de contexte* est d'assurer cohérence et protection dans un cadre multiprocesseur – multiutilisateur

- Le SE s'appuie généralement sur les niveaux de privilège définis au niveau des processeurs
- Ces niveaux attribuent aux objets connus par le processus (segments, tables, ...) une valeur qui définit des règles d'accès
- Dépendent de la commutation de contexte interruptions matérielles et logicielles ainsi qu'exceptions...

Remarques :

- ① En passant en mode Noyau, le processeur accède à des zones de mémoire protégées
- ② Le nombre d'appels système disponibles est fixe, chaque type d'appel étant identifié par un entier unique
- ③ A chaque type d'appel système peut être associé un ensemble d'arguments précisant l'information à transférer de l'espace Utilisateur à l'espace Noyau et réciproquement

Etapas d'un appel système

- ① Un appel a lieu (par exemple par l'appel d'une fonction de la bibliothèque C par un programme)

Etapes d'un appel système

- ① Un appel a lieu (par exemple par l'appel d'une fonction de la bibliothèque C par un programme)
- ② Le *wrapper* de cette fonction rend disponible tous les arguments au *gestionnaire d'interruption*; ces arguments sont passés (*via la pile système*) au wrapper qui les copie dans certains registres

Etapes d'un appel système

- ① Un appel a lieu (par exemple par l'appel d'une fonction de la bibliothèque C par un programme)
- ② Le *wrapper* de cette fonction rend disponible tous les arguments au *gestionnaire d'interruption*; ces arguments sont passés (*via la pile système*) au wrapper qui les copie dans certains registres
- ③ Le wrapper copie le numéro de l'appel système correspondant dans un registre dédié

Etapes d'un appel système

- ① Un appel a lieu (par exemple par l'appel d'une fonction de la bibliothèque C par un programme)
- ② Le *wrapper* de cette fonction rend disponible tous les arguments au *gestionnaire d'interruption*; ces arguments sont passés (*via la pile système*) au wrapper qui les copie dans certains registres
- ③ Le wrapper copie le numéro de l'appel système correspondant dans un registre dédié
- ④ Le wrapper exécute une *interruption* (int 0x80) qui commute le processeur en mode Noyau, et exécute le code pointé par l'adresse 0x80 dans le *vecteur d'interruptions*

Etapes d'un appel système

- ① Un appel a lieu (par exemple par l'appel d'une fonction de la bibliothèque C par un programme)
- ② Le *wrapper* de cette fonction rend disponible tous les arguments au *gestionnaire d'interruption*; ces arguments sont passés (*via la pile système*) au wrapper qui les copie dans certains registres
- ③ Le wrapper copie le numéro de l'appel système correspondant dans un registre dédié
- ④ Le wrapper exécute une *interruption* (int 0x80) qui commute le processeur en mode Noyau, et exécute le code pointé par l'adresse 0x80 dans le *vecteur d'interruptions*
- ⑤ En réponse le noyau appelle la routine `system_call` pour gérer l'interruption

Exemple

mode Utilisateur

programme

```
...  
execve(path, arg, envp);  
...
```

wrapper glibc

```
execve(path, arg, envp) {  
    int 0x80  
    /*arguments : _NR_execve, path,  
                 argv, envp*/  
    return}
```

mode Noyau

routine système

```
sys_execve()  
{  
    ...  
    return error  
}
```

gestionnaire d'interruption

```
system_call:  
...  
call sys_call_table[_NR_execve]  
...
```

Exemple

mode Utilisateur

programme

```
...  
execve(path, arg, envp);  
...
```

wrapper glibc

```
execve(path, arg, envp) {  
    int 0x80  
    /*arguments : _NR_execve, path,  
                  argv, envp*/  
    return}
```

mode Noyau

routine système

```
sys_execve()  
{  
    ...  
    return error  
}
```

gestionnaire d'interruption

```
system_call:  
...  
call sys_call_table[_NR_execve]  
...
```

Exemple

mode Utilisateur

programme

```
...  
execve(path, arg, envp);  
...
```

wrapper glibc

```
execve(path, arg, envp) {  
    int 0x80  
    /*arguments : _NR_execve, path,  
                  argv, envp*/  
    return}  
}
```

mode Noyau

routine système

```
sys_execve()  
{  
    ...  
    return error  
}
```

gestionnaire d'interruption

```
system_call: ←  
...  
call sys_call_table[_NR_execve]  
...
```

Exemple

mode Utilisateur

programme

```
...  
execve(path, arg, envp);  
...
```

wrapper glibc

```
execve(path, arg, envp) {  
    int 0x80  
    /*arguments : __NR_execve, path,  
                  argv, envp*/  
    return}
```

mode Noyau

routine système

```
sys_execve()  
{  
    ...  
    return error  
}
```

gestionnaire d'interruption

```
system_call: ←  
...  
call sys_call_table[__NR_execve]  
...
```

Exemple

mode Utilisateur

programme

```
...  
execve(path, arg, envp);  
...
```

wrapper glibc

```
execve(path, arg, envp) {  
    int 0x80  
    /*arguments : _NR_execve, path,  
                 argv, envp*/  
    return}
```

mode Noyau

routine système

```
sys_execve()  
{  
    ...  
    return error  
}
```

gestionnaire d'interruption

```
system_call: ←  
...  
call sys_call_table[_NR_execve]  
...
```

Exemple

mode Utilisateur

programme

```
...  
execve(path, arg, envp);  
...
```

wrapper glibc

```
execve(path, arg, envp) {  
    int 0x80  
    /*arguments : _NR_execve, path,  
                 argv, envp*/  
    return}
```

mode Noyau

routine système

```
sys_execve()  
{  
    ...  
    return error  
}
```

gestionnaire d'interruption

```
system_call: ←  
...  
call sys_call_table[_NR_execve]  
...
```


Exemple

mode Utilisateur

programme

```
...  
execve(path, arg, envp);  
...
```

wrapper glibc

```
execve(path, arg, envp) {  
    int 0x80  
    /*arguments : _NR_execve, path,  
                 argv, envp*/  
    return}
```

mode Noyau

routine système

```
sys_execve()  
{  
    ...  
    return error  
}
```

gestionnaire d'interruption

```
system_call:  
...  
call sys_call_table[_NR_execve]  
...
```

Exemple

mode Utilisateur

programme

```
...  
execve(path, arg, envp);  
...
```

wrapper glibc

```
execve(path, arg, envp) {  
    int 0x80  
    /*arguments : _NR_execve, path,  
                 argv, envp*/  
    return}
```

mode Noyau

routine système

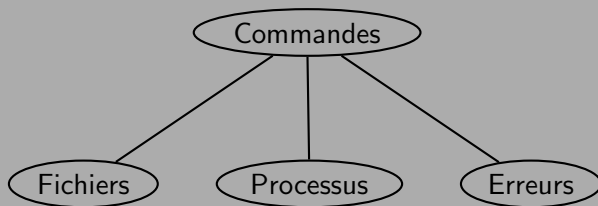
```
sys_execve()  
{  
    ...  
    return error  
}
```

gestionnaire d'interruption

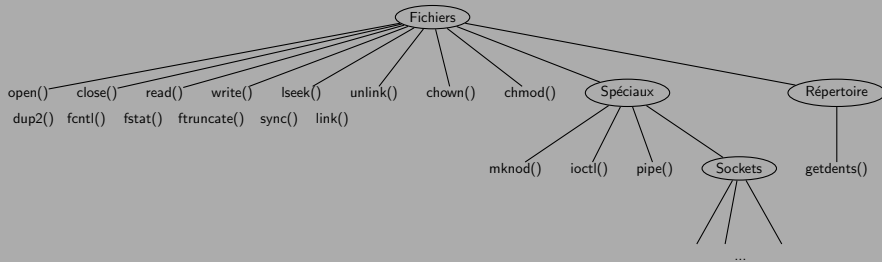
```
system_call:  
...  
call sys_call_table[_NR_execve]  
...
```

Commandes UNIX/LINUX

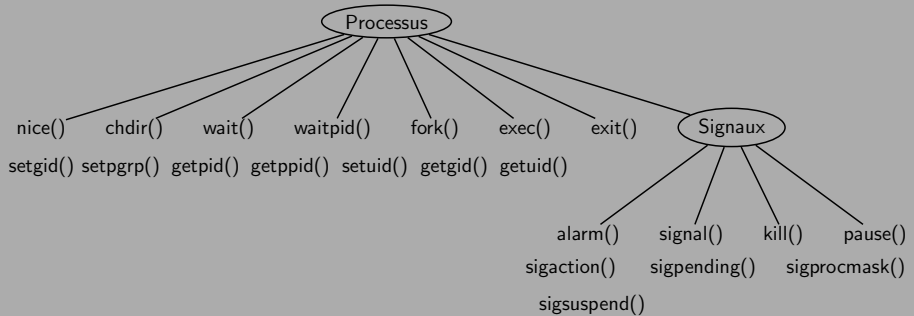
Arborescence des commandes



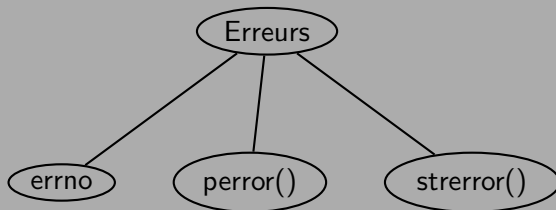
Fichiers



Processus



Erreurs



Mise en place des TPs

Idée Générale

- Etude du comportement des fonctions système étudiées en cours
- Au delà, conception d'utilitaires en ligne de commande, pouvant recevoir des arguments, et reposant sur l'emploi des fonctions système connues

Idée Générale

- Etude du comportement des fonctions système étudiées en cours
- Au delà, conception d'utilitaires en ligne de commande, pouvant recevoir des arguments, et reposant sur l'emploi des fonctions système connues

Exemple : La fonction « maison »

monitor [-t *délai*][-c *NbParcours*]{*fichiers*}+

monitor parcourt les *fichiers* spécifiés toutes les *délai* secondes (si l'option **-t** est précisée) et affiche des informations sur ceux des fichiers qui ont été modifiés depuis le dernier parcours. L'option **-c** permet de préciser le *NbParcours* à effectuer.

monitor : comment ?

- L'exécutable résulte (à peu de choses près...) d'une compilation du genre

`g++ -o monitor MonProgrammeMonitor.cxx`

(avec l'option `-o` pour "output" après *compilation* et *édition de liens*)

monitor : comment ?

- L'exécutable résulte (à peu de choses près...) d'une compilation du genre

`g++ -o monitor MonProgrammeMonitor.cxx`

(avec l'option `-o` pour "output" après *compilation* et *édition de liens*)

- La possibilité de spécifier un *délai*, un *NbParcours*, sur quels *fichiers*, résulte de la possibilité de passer des *arguments* à `MonProgrammeMonitor.cxx`

monitor : comment ?

- L'exécutable résulte (à peu de choses près...) d'une compilation du genre

`g++ -o monitor MonProgrammeMonitor.cxx`

(avec l'option `-o` pour "output" après *compilation* et *édition de liens*)

- La possibilité de spécifier un *délai*, un *NbParcours*, sur quels *fichiers*, résulte de la possibilité de passer des *arguments* à `MonProgrammeMonitor.cxx`
- En « bonne programmation », on attend de **monitor** une gestion propre d'éventuelles *erreurs* pouvant (aussi) être générées par des appels système

monitor : comment ?

- L'exécutable résulte (à peu de choses près...) d'une compilation du genre

`g++ -o monitor MonProgrammeMonitor.cxx`

(avec l'option `-o` pour "output" après *compilation* et *édition de liens*)

- La possibilité de spécifier un *délai*, un *NbParcours*, sur quels *fichiers*, résulte de la possibilité de passer des *arguments* à `MonProgrammeMonitor.cxx`
- En « bonne programmation », on attend de **monitor** une gestion propre d'éventuelles *erreurs* pouvant (aussi) être générées par des appels système

D'où...

Outils pour les TPS...

- passage d'arguments *via* `argc` et `argv`
- `gcc` et `g++` pour la compilation séparée
- `make` et `makefiles`
- encapsulation (*wrapping*) des appels système et gestion des erreurs avec la classe C++ `CExc`

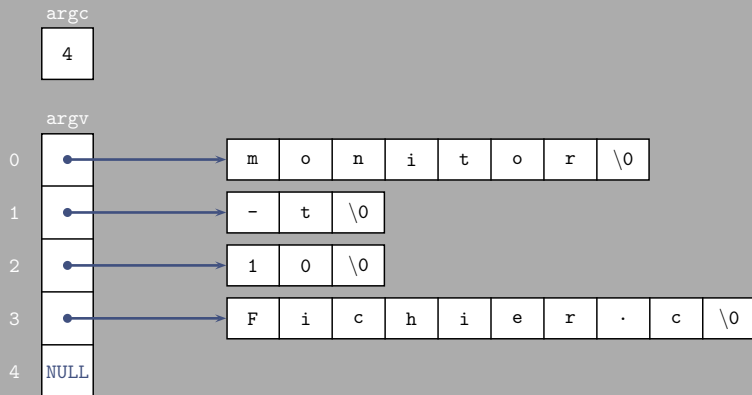
Passage d'arguments à un programme

```
main (int argc, char *argv[]) {  
    // ...  
    // analyse de argc et argv  
    // ...  
}
```

- *argc* désigne le nb total d'arguments passés à la fonction, y incluant le nom d'appel de la fonction elle-même
- *argv* est un tableau de pointeurs sur des tableaux de caractères `C` dont chacun contient un argument

argc et argv, exemple

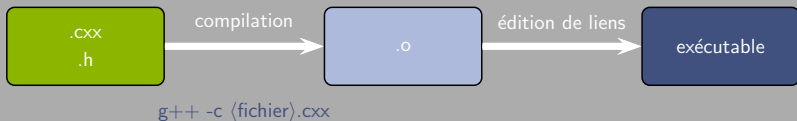
```
monitor -t 10 Fichier.c
```



gcc et g++ pour la compilation séparée



gcc et g++ pour la compilation séparée



gcc et g++ pour la compilation séparée



gcc et g++ pour la compilation séparée



En cas de *compilation séparée*, on souhaite ne pas avoir à tout recompiler lorsqu'on modifie un fichier parmi plusieurs (par ex. dans le cas d'un programme disséminé en plusieurs fichiers représentant plusieurs milliers de lignes de code...)

gcc et g++ pour la compilation séparée



En cas de *compilation séparée*, on souhaite ne pas avoir à tout recompiler lorsqu'on modifie un fichier parmi plusieurs (par ex. dans le cas d'un programme disséminé en plusieurs fichiers représentant plusieurs milliers de lignes de code...)

→ d'où l'intérêt de l'utilitaire *make* et des *makefiles*

Makefile

Un *makefile* est constitué de *règles* qui disent comment, et dans quel ordre effectuer une compilation. Chaque règle définit trois entités :

- le fichier construit par la règle, matérialisé par une *cible*
- la liste des *dépendances* nécessaires à la construction de cette cible
- les *opérations* à réaliser pour la construction de la cible

Makefile

Un *makefile* est constitué de *règles* qui disent comment, et dans quel ordre effectuer une compilation. Chaque règle définit trois entités :

- le fichier construit par la règle, matérialisé par une *cible*
- la liste des *dépendances* nécessaires à la construction de cette cible
- les *opérations* à réaliser pour la construction de la cible

Format de règle :

$$\langle \text{cible} \rangle : \quad \langle \text{dépendances} \rangle \\ \quad \quad \langle \text{opérations} \rangle$$

Makefile

Un *makefile* est constitué de *règles* qui disent comment, et dans quel ordre effectuer une compilation. Chaque règle définit trois entités :

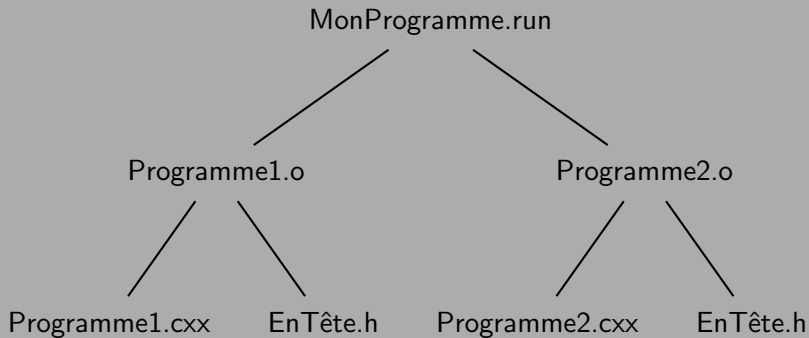
- le fichier construit par la règle, matérialisé par une *cible*
- la liste des *dépendances* nécessaires à la construction de cette cible
- les *opérations* à réaliser pour la construction de la cible

Format de règle :

⟨cible⟩ : ⟨dépendances⟩
↔ ⟨opérations⟩

Attention à la tabulation *obligatoire* !

Exemple



Makefile correspondant

```
MonProgramme.run: Programme1.o Programme2.o  
    g++ -o MonProgramme.run Programme1.o Programme2.o
```

```
Programme1.o: Programme1.cxx EnTete.h  
    g++ -c Programme1.cxx
```

```
Programme2.o: Programme2.cxx EnTete.h  
    g++ -c Programme2.cxx
```

Makefile correspondant

```
MonProgramme.run: Programme1.o Programme2.o  
    g++ -o MonProgramme.run Programme1.o Programme2.o
```

```
Programme1.o: Programme1.cxx EnTete.h  
    g++ -c Programme1.cxx
```

```
Programme2.o: Programme2.cxx EnTete.h  
    g++ -c Programme2.cxx
```

Remarque : comment gérer le cas fichiers disséminés dans des répertoires distincts ?

Makefile correspondant (variation)

Avec `EnTete.h` dans un répertoire `include` voisin...

```
MonProgramme.run: Programme1.o Programme2.o
    g++ -o MonProgramme.run Programme1.o Programme2.o
```

```
Programme1.o: Programme1.cxx ../include/EnTete.h
    g++ -c Programme1.cxx -I ../include
```

```
Programme2.o: Programme2.cxx ../include/EnTete.h
    g++ -c Programme2.cxx -I ../include
```

Makefile correspondant (variation)

Avec `EnTete.h` dans un répertoire include voisin...

```
MonProgramme.run: Programme1.o Programme2.o
    g++ -o MonProgramme.run Programme1.o Programme2.o
```

```
Programme1.o: Programme1.cxx ../include/EnTete.h
    g++ -c Programme1.cxx -I ../include
```

```
Programme2.o: Programme2.cxx ../include/EnTete.h
    g++ -c Programme2.cxx -I ../include
```

→ L'option `-I` dans `g++ -I ../include` permet d'ajouter le répertoire include aux chemins explorés par `g++`

Gestion des erreurs système

De manière standard

- un appel système échoué positionne la variable globale `errno`
- la fonction `perror()` permet d'afficher un message ainsi que la cause de l'erreur obtenue à partir du dernier positionnement de `errno`
- la fonction `strerror()` permet d'afficher la cause de l'erreur à partir de toute valeur de `errno` passée en paramètre

Exemple

```
...
#include <errno.h> // valeurs admises pour errno
#include <string.h> // pour strerror()
#include <iostream>
...
struct stat buf;
...
const int e = stat("Fichier.txt", &buf);
if (e == -1) {
    std::cerr << "Erreur en ouverture de fichier: "
               << strerror(errno) << "\n";
}
```

stat() renvoie -1 en erreur, auquel cas `errno` peut valoir par exemple `ENOENT` (fichier inexistant), `EACCESS` (pas d'accès autorisé)...

CExc, classe de gestion des erreurs

- Les erreurs système modifient la variable globale `errno` : si une seconde erreur apparaît avant de pouvoir prendre connaissance de la valeur d'`errno`, on perd la première erreur...

CExc, classe de gestion des erreurs

- Les erreurs système modifient la variable globale `errno` : si une seconde erreur apparaît avant de pouvoir prendre connaissance de la valeur d'`errno`, on perd la première erreur...
- On souhaite donc interrompre l'exécution d'un programme précisément à l'endroit où une erreur système surgit, afin de transmettre un maximum de renseignements au code appelant.

CExc, classe de gestion des erreurs

- Les erreurs système modifient la variable globale `errno` : si une seconde erreur apparaît avant de pouvoir prendre connaissance de la valeur d'`errno`, on perd la première erreur...
- On souhaite donc interrompre l'exécution d'un programme précisément à l'endroit où une erreur système surgit, afin de transmettre un maximum de renseignements au code appelant.
- Idée : une classe `CExc` dérivée de la classe standard `exception`, et dotée d'une fonction d'affichage adéquat.

CExc, classe de gestion des erreurs

- Les erreurs système modifient la variable globale `errno` : si une seconde erreur apparaît avant de pouvoir prendre connaissance de la valeur d'`errno`, on perd la première erreur...
- On souhaite donc interrompre l'exécution d'un programme précisément à l'endroit où une erreur système surgit, afin de transmettre un maximum de renseignements au code appelant.
- Idée : une classe `CExc` dérivée de la classe standard `exception`, et dotée d'une fonction d'affichage adéquat.
- Les exceptions C++ interrompent le flot d'exécution du programme, remontant au premier `catch` qui peut les traiter.

Gestion des erreurs avec try-catch

Les exceptions C++ interrompent le flot d'exécution du programme, remontant au premier *catch* qui peut les traiter :

```
int main(int argc, char* argv[]) {  
    //...  
    try{  
        // corps de l'exercice appelant des fonctions  
        // susceptibles de lever des exceptions  
    }  
    catch(const CExc & Exc) {  
        // affichage des informations relatives  
        // a l'erreur  
    }  
    //...  
}
```

Classe CExc (déclaration)

```
class CExc: public std::exception {
protected:
    std::string m_info, m_nomf;
    int m_descrpic; bool m_qdescrpic;
protected:
    std::ostream & _Edit (std::ostream & os) const;
public:
    CExc (const std::string & NomFonction,
          const std::string & Info);          // throw ()
    CExc (const std::string & NomFonction,
          int Descrpic)                      // throw ()
    virtual ~ CExc (void);                  // throw ()
    friend std::ostream & operator << (std::ostream & os,
                                       const CExc & Item);
};

nsSysteme::CExc::CExc (const std::string & NomFonction,
                       const std::string & Info) // throw ()
: m_info(Info), m_nomf(NomFonction),
  m_descrpic(-1), m_qdescrpic(false) {}
```

_Edit et surcharge de << (définition)

```
// constructeurs, destructeurs ...
```

```
inline
```

```
std::ostream & CEX::_Edit (std::ostream & os) const
{
    os<<"\nNom de la fonction : "<<m_nomf
        <<"\nNo. erreur systeme:"<<m_errnoval
        <<"\ni.e.:"<<std::string(" ")+m_strerrorval
        <<"\nParametres au moment de l'erreur" <<"\n";
    if (m_qdescrpic) os<<"descripteur fichier: "<<m_descrpic;
    else os<<m_info;
    os<<"\n";
    return os;
} // _Edit()
```

```
inline
```

```
std::ostream & nsSysteme::operator << (std::ostream & os,
                                         const CExc & Item)
{return Item._Edit (os);} // operator <<
```

Wrappers des fonctions système

- Chaque fonction utilisée est placée dans un *wrapper* de même nom mais commençant par une majuscule, par exemple `Stat()` pour la fonction système `stat()`
- Ce wrapper sert de surcouche permettant de gérer les erreurs via la classe `CExc`. Ainsi pour la définition de `Stat()` :

```
Stat(const char * ChemFic, struct stat * buf)
// throw (CExc)
{
    if (::stat(chemFic, buf))
        throw CExc("stat()", chemFic);
} // Stat()
```


Wrappers des fonctions système

- Chaque fonction utilisée est placée dans un *wrapper* de même nom mais commençant par une majuscule, par exemple `Stat()` pour la fonction système `stat()`
- Ce wrapper sert de surcouche permettant de gérer les erreurs via la classe `CExc`. Ainsi pour la définition de `Stat()` :

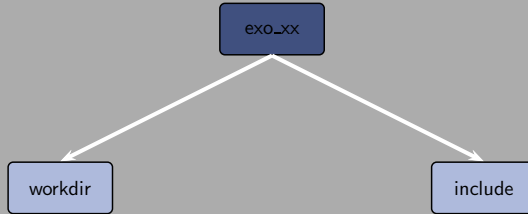
```
Stat(const char * ChemFic, struct stat * buf)
// throw (CExc)
{
    if (::stat(chemFic, buf))
        throw CExc("stat()", chemFic);
} // Stat()
```

→ *Détails en TD/TP...*

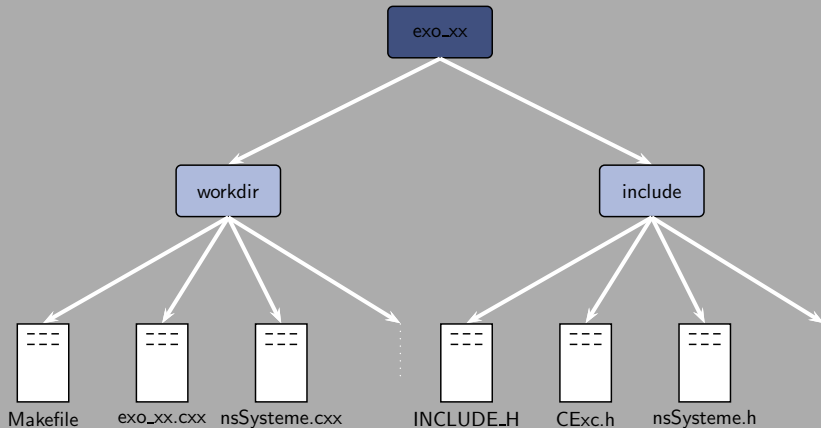
Organisation des répertoires pour chaque TP

exo_xx

Organisation des répertoires pour chaque TP



Organisation des répertoires pour chaque TP



Organisation des répertoires pour chaque TP

- *workdir* (Makefiles, fichiers .cxx, exécutables...)
- *include* (Fichiers d'en-tête, macros...)

Organisation des répertoires pour chaque TP

- *workdir* (Makefiles, fichiers .cxx, exécutables...)
 - *Makefile*
 - *exo_xx.cxx*
 - *nsSysteme.cxx* : définition des wrappers non inline des fonctions système
 - ...
- *include* (Fichiers d'en-tête, macros...)

Organisation des répertoires pour chaque TP

- *workdir* (Makefiles, fichiers .cxx, exécutables...)
 - *Makefile*
 - *exo_xx.cxx*
 - *nsSysteme.cxx* : définition des wrappers non inline des fonctions système
 - ...
- *include* (Fichiers d'en-tête, macros...)
 - *INCLUDE_H* : fichier de macros pour le Makefile
 - *CExc.h* : classe des exceptions système
 - *nsSysteme.h* : prototypes des wrappers des fonctions système dans l'espace de noms correspondant
 - ...