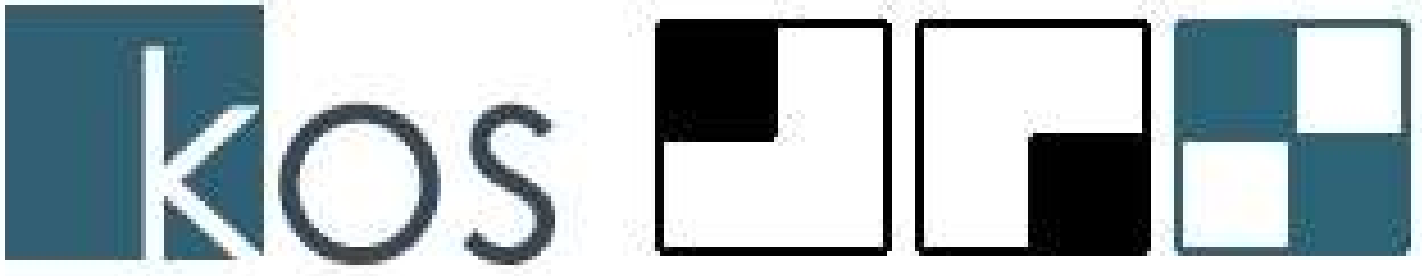


TX

« Implémentation de nouvelles fonctionnalités pour le système d'exploitation KOS »

Automne 2003

Plan

- Présentation du projet
- État au début de la TX
- Travail réalisé durant la TX
  - Générateur d'interfaces XML
  - Vers l'application utilisateur ...
  - Pilote de périphérique PCI
- Perspectives

Le projet

- Création d'un *petit* système d'exploitation pour PC
- Vocation éducative
  - Pour les développeurs : apprentissage du fonctionnement d'un OS
  - Pour les autres : documentations
- Créé en 1998
- Développé « from scratch »
- Jusqu'à 10 développeurs, 3 actifs à l'heure actuelle

État du projet

Au début de la TX

- Système modulaire, préemptif
- Gestion mémoire (allocateur noyau, mapping, reverse mapping)
- Scheduling : threads noyaux
- Pilotes de périphériques : disque, partition, console, clavier, série
- Interruptions
- Synchronisation
- Débuggage

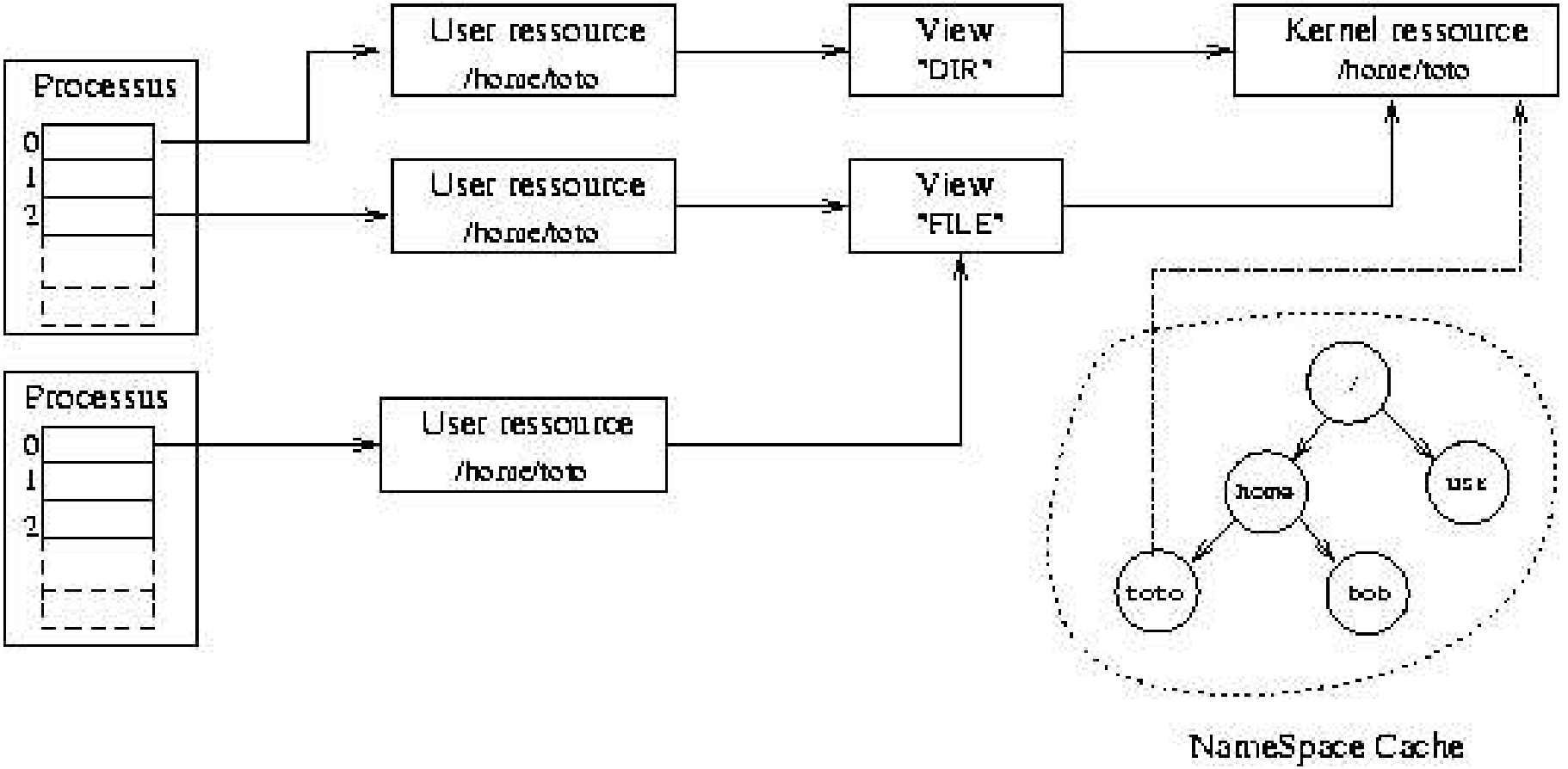
Terminologie

- **Thread** : contexte d'éxécution
- **Team** : ensemble des threads fonctionnant dans le même espace d'adressage

Au sens *Unix,* un processus est une **team** + un **thread**

Karm

- Fonctionnalité originale pour la *gestion des ressources*
- Remplacement du ioctl() Unix
  - `int ioctl(int d, int request, ...);`
- Ouverture d'une ressource selon une *interface*
  - `int open(char *path, unsigned interface);`
- Structure interne dans le noyau



Appel système

- Sur une ressource
- Avec un numéro de méthode
- Avec des paramètres

Interfaces : problématique

Définition des interfaces en C

```
struct block {
  result_t (*read)(struct ures *ur, char *buffer,
                   count_t block_start,
                   count_t *inout_block_count);
  result_t (*write)(struct ures *ur, const char *buffer,
                    count_t block_start,
                    count_t *inout_block_count);
};
```

Plusieurs problèmes

- Pas de vérification du nombre de paramètres
- Pas d'identifiants d'interfaces et de méthodes pour l'espace utilisateur
- Distinction entre méthodes noyau / méthodes utilisateur

- **Solution peu flexible**

Solution : XML

Solution adoptée : description des *interfaces* en XML

- Langage à balises simple et flexible
- Analyseurs existants
- Possibilité de générer divers sorties à partir d'une description XML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<interface name="block">

 <method name="read">
  <arg type="struct ures*" name="ur"/>
  <arg type="char*" name="buffer"/>
  <arg type="count_t" name="block_start"/>
  <arg type="count_t*" name="inout_block_count"/>
  </method>
 <method name="write">
  <arg type="struct ures*" name="ur"/>
  <arg type="const char*" name="buffer"/>
  <arg type="count_t" name="block_start"/>
  <arg type="count_t*" name="inout_block_count"/>
 </method>

</interface>
```

Définition du langage

DTD : Document Type Definition

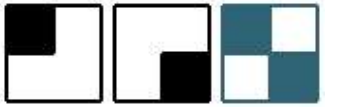
```
<!ELEMENT interface (code*,method+)>
<!ATTLIST interface
 name CDATA #REQUIRED
>

<!ELEMENT code (#PCDATA)>
<!ATTLIST code
 domain CDATA #IMPLIED
>

<!ELEMENT method (arg+)>
<!ATTLIST method
 name CDATA #REQUIRED
 domain CDATA #IMPLIED
>

<!ELEMENT arg EMPTY>
<!ATTLIST arg
 type CDATA #REQUIRED
 name CDATA #REQUIRED
>
```

- **Interface**
- **Code** : intégration de code personnalisé (définitions de types, de constantes)
- **Méthodes**
- **Arguments** : nom et type

- Notion de *domaine*

- Validation d'un fichier XML par la DTD

Sorties



Quatre sorties :

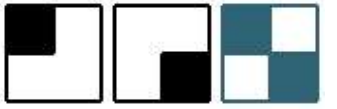
- Un *header* pour le noyau avec une structure décrivant l'interface
  - Implémentation des interfaces
- Un *source C* pour le noyau avec un tableau global des interfaces
  - Vérification des domaines et du nombre de paramètres
- Un *header* pour l'espace utilisateur avec identifiants de méthodes et d'interfaces
  - Bibliothèque standard C
- Un *source C* pour l'espace utilisateur avec des *wrappers* pour les appels systèmes
  - Bibliothèque standard C

Kos IDL

- Programme en C *(utils/kosidl.c)*
- Utilisation de libxml2 avec l'API DOM
- Différentes sorties générées automatiquement via des dépendances de Makefile
- Solution souple, flexible et bien adaptée

Démonstration

Vers l'appli utilisateur



Objectif : éxécuter des applications utilisateur

Travail sur :

- Threads utilisateurs
- Lien utilisateur / noyau
- Ressource *Process*
- Chargeur *ELF*
- Gestion de la mémoire virtuelle

Threads utilisateur

- Définition du contexte : *cpu_state_t*
- Création : *create_user_thread*
- Initialisation du contexte : *init_user_thread_context*
- Mise à jour du *TSS*
- Changement d'espace d'adressage
- Mini chargeur *ELF*

- Possibilité d'éxécuter des threads utilisateur dans un espace d'adressage séparé

Lien utilisateur / noyau

- Appel système (interruption logicielle)
  - Sur une ressource
  - D'une méthode
  - Avec des paramètres
- Point d'entrée : *syscall_entry_point*
- Lien avec KARM : *syscall*
  - Vérifications
  - Récupération des paramètres
  - Appel de la méthode demandée

- Entrée standard, sortie standard, sortie d'erreur
- Libcharfile

- Programme utilisateur qui écrit sur la sortie standard

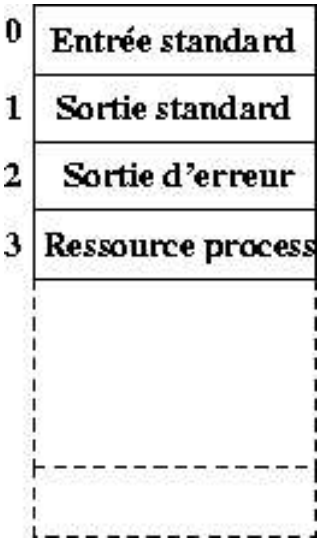
Ressource *process 1/3*

Objectif : mettre à disposition de l'utilisateur des méthodes qui ne sont pas relatives à une ressource précise

Solution : ressource *process*, toujours présente dans une team sous le descripteur 3.

Méthodes :

- open
- close
- fork
- exec
- getpid
- getppid
- brk



Ressource *process 2/3*

Open/close

- Méthodes existantes dans *Karm*
- Simples wrappers

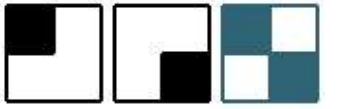
Fork : création d'un nouveau « processus »

- Emulation de la sémantique Unix : duplication de la team et du thread courant

- Création d'une nouvelle team
- Duplication du tableau des *User Ressource*
- Duplication de l'espace d'adressage
- Copie des informations relatives aux piles utilisateurs
- Copie du thread utilisateur

Fork()

Ressource *process 3/3*



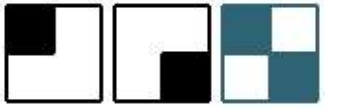
Exec : chargement d'un nouveau programme

- Ouverture du binaire
- Suppression des régions virtuelles
- Fermeture des ressources (sauf 0, 1, 2, 3)
- Chargement du binaire
- Mise à jour de l'adresse du tas
- Mapping de la pile utilisateur
- Initialisation du contexte du thread

Brk : gestion du tas

- Agrandissement et réduction dynamique d'une région virtuelle précise

Chargeur ELF



Chargeur ELF initial

- Chargement des sections de code / données
- Peu de vérifications

Chargeur ELF amélioré

- Couche d'abstraction du format binaire
- Gestion du BSS
- Calcul de l'adresse de début du tas
- Plus de vérifications
- Début de gestion d'un interpréteur

Mémoire virtuelle

- Création de régions, avec ou sans positionnement fixe, mémoire anonyme ou non
- Suppression de régions, partiellement ou non
- Copie d'un espace d'adressage, d'une région
- Elimination de toutes les régions d'un espace d'adressage
- Gestion du tas
- Gestion des défauts de page
  - Support du *Copy-On-Write*
  - Support de l'extension des piles utilisateurs
  - Support pages anonymes / *file mapping*
- Système d'allocation des piles utilisateur

Résultat

- Mini bibliothèque standard C
- Un programme utilisateur éxécuté au démarrage :
  - il ouvre un fichier
  - il se fork()
  - il éxécute un autre programme
  - il lance d'autres threads utilisateur
  - il alloue de la mémoire sur le tas

PCI

- Découverte du fonctionnement du bus PCI
- Pilote de périphérique très simple
  - Détection des périphériques PCI
  - Association à un constructeur / modèle
  - Lecture des valeurs de configuration (port d'entrées sorties, mapping mémoire)

- Un début pour, plus tard, utiliser des cartes réseaux

Conclusion / perspectives

Pour le projet KOS

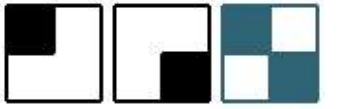
- Grandes avancées en moins de 6 mois
- Maintenant
  - Rentrer dans une phase de stabilisation
  - Travail sur la synchronisation
- Objectif à long terme
  - Portage GNU libc

Conclusion / perspectives

Apport personnel

- Mélanie
  - Comprendre le fonctionnement interne d'un OS
  - Programmation en C, découverte de libxml2
- Thomas
  - Occasion d'expliquer le fonctionnement complet de Kos à une nouvelle personne
  - Participation à un projet personnel

Démonstration



Démonstration de KOS dans l'émulateur Bochs