Gestion des niveaux de protection

Y.Denneulin, J. Mossière,
G.Mounié, S.Nieuviarts,
C. Rippert, F.Rousseau,
S. Viardot

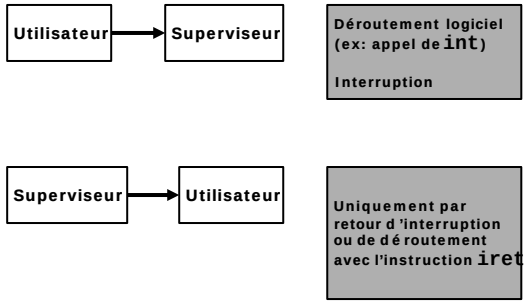
Plan

- Niveaux de protection
- Interface utilisateur du noyau
- Changement de contexte
- Gestion de la mémoire

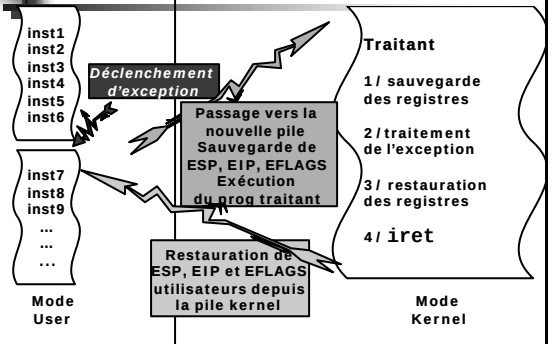
Niveaux de protection

- Mode Utilisateur (*User*) : niveau 3
  - Jeu d'instructions réduit : pas d'accès à certains registres ni à la mémoire du noyau ni aux entrées-sorties
  - Utilisé exécuter les programmes d'application
- Mode Superviseur (*Kernel*) : niveau 0
  - Mode le plus privilégié : accès à toutes les instructions du processeur et toute la mémoire
  - Choisi pour exécuter le code du noyau (interruptions ou déroutements)
- Autres niveaux de privilèges de l'Intel : ignorés
- Chaque niveau de privilège utilise sa propre mémoire
  - Un processus a donc deux piles : une pour chaque niveau

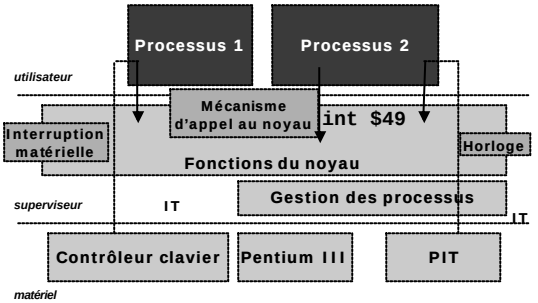
Changement de mode



Changement de mode



Causes de chang. de mode



Organisation des primitives du noyau
niveaux noyau et utilisateur

- Chaque primitive accessible en mode utilisateur
  - **`int kill(int pid);`**
  - à définir dans un fichier **`syscalls.h`**
  - **`.globl kill`**
  - à implanter dans un fichier **`syscalls.S`** et dont le code ne fait que l'appel à la primitive noyau via **`int $49`**
- Primitive correspondante accessible en mode noyau
  - **`int kill(int pid);`**
  - définie dans **`kernel/processus.h`**
  - implémentée dans **`kernel/processus.c`**
  - appelée dans le gestionnaire d'interruption **`49`**

Appels noyau

- Réalisation des bibliothèques de primitives noyau par **`int`**

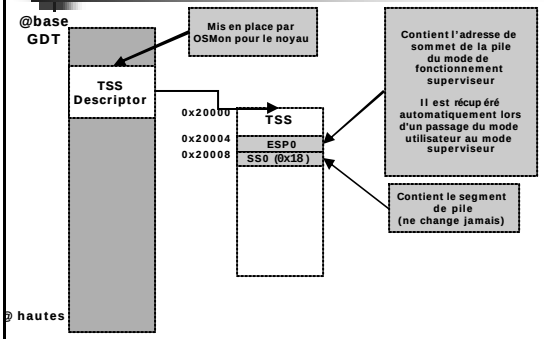
dans `syscalls.S`

```
kill:
   movl 4(%esp), %ecx
   movb $<numéro primitive>, %eax
   int $49
   ret
```

`trap_handler:`
sauvegarde des registres comme pour un gestionnaire d'interruption classique
on copie les paramètres depuis les registres sur la pile superviseur
leur nombre dépend du numéro de primitive contenu dans **`eax`**
si **`eax`** contient le numéro de **`kill`** alors appel de la fonction noyau **`kill`**
restauration des registres
`iret`

dans `processus.c`

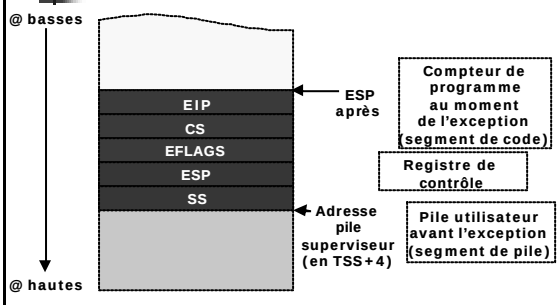
Les *Task State Segments*



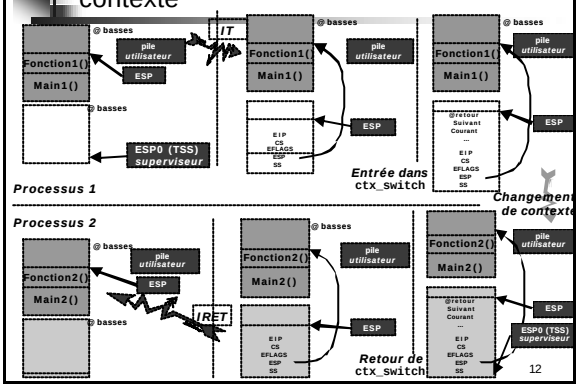
Changement de contexte

- Le changement de contexte n'est effectué que par le noyau, alors que le processeur est déjà en mode superviseur
  - peu de changement par rapport aux phase précédentes du projet
  - chaque processus doit toujours avoir sa propre pile superviseur
- Action à réaliser
  - Modifier ESP0 dans le TSS (*Task State Segment*) en fonction du nouveau processus élu
    - Au prochain passage en kernel, il faudra utiliser sa pile

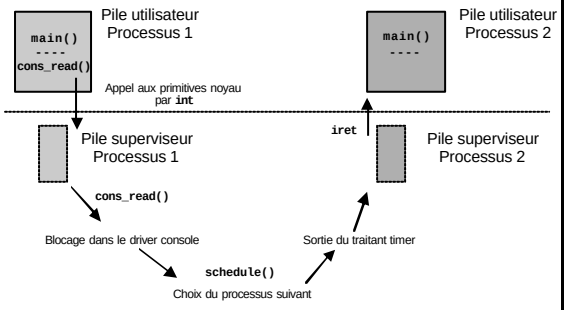
Sauvegarde sur la pile du noyau lors d'un changement de niveau de privilège



Jeux de piles lors du changement de contexte



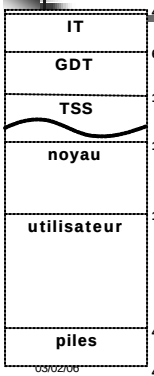
Lien avec la gestion des processus



Pistes pour la création de processus

- Étudier l'algorithme de création d'un processus
- Il faudra en particulier
  - Allouer une pile utilisateur et superviseur par processus
    - Stocker les adresses de début de pile
  - Remplir la pile superviseur de données permettant le passage au niveau utilisateur par **`iret`** après le premier changement de contexte
  - Remplir la pile utilisateur pour permettre le démarrage de la première fonction
    - Paramètres
    - Adresse de retour vers du code qui fait la terminaison

Carte mémoire



- **`4 Ko`** : la table d'interruptions
- **`64 Ko`** : la table globale des descripteurs GDT
- **`128 Ko`** : le segment d'état de tâche TSS, utilisé par OSMon pour y stocker l'état de votre système quand il s'exécute
- **`1 Mo`** : adresse où est chargé le noyau
- **`16 Mo`** : début de l'espace utilisateur, le code utilisateur est chargé à cette adresse par le noyau
- **`40 Mo`** : début de la zone où sont allouées les piles utilisateurs des processus (**`user_stack_alloc`**)
- **`48 Mo`** : fin de l'espace utilisateur

Chargement du code en mémoire

- Édition de liens
  - Suivant les directives du fichier
    - **`kernel/kernel.lds`**
  - Création d'un seul binaire
- Chargement du noyau
  - Chargement à 1 Mo
    - **`. = 0x100000`**
  - Exécution du code de **`crt0`**
  - Copie de la partie binaire utilisateur
    - **`mem_heap_end = 0x1000000`**
    - **`user_start = mem_heap_end;`**
    - **`user_stack_heap = 0x2800000`**
    - **`user_end = 0x3000000`**

Allocation mémoire

- Allocation de mémoire dans le noyau
  - Utilisation du module **`mem`**
    - **`kernel/mem.h`** et **`kernel/mem.c`**
- Allocation des piles utilisateur
  - Allocation faite par le noyau
  - Mais dans la partie utilisateur au dessus de 40 Mo
  - Utilisation du module **`user_stack_mem`**
    - **`kernel/user_stack_mem.[hc]`**
- Allocation de mémoire utilisateur possible
  - Utilisation du même module **`mem`** à copier dans **`user`**, les codes noyau et utilisateur étant séparés
  - Synchronisation nécessaire autour des appels au module (à cause de l'IT timer)