Projet : développement d'un mini système d'exploitation pour PC x86

La gestion des processus

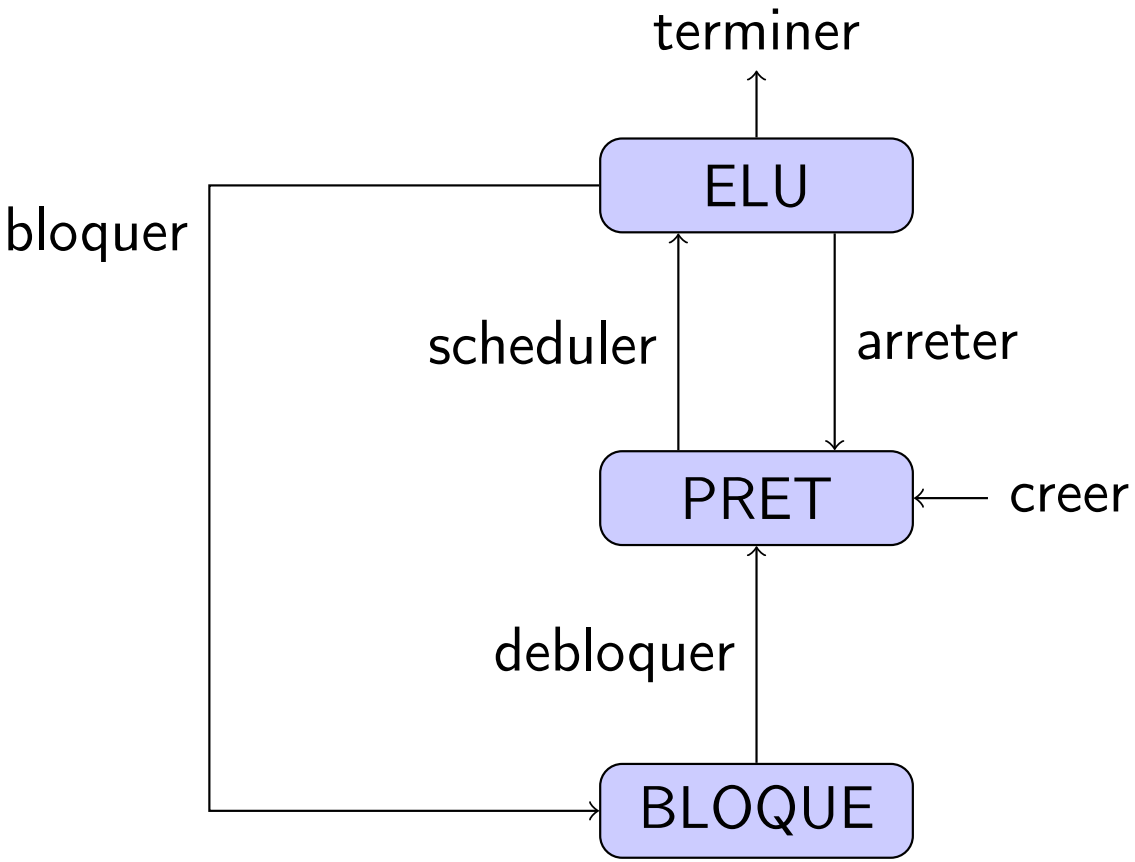
Jérôme Ermont et Emmanuel Chaput

IRIT - Toulouse INP/ENSEEIHT

Qu'est-ce qu'un processus ?

- Abstraction de l'exécution d'un programme
- Défini par un descripteur de processus :
  - Nom du programme
  - Identifiant du processus
  - Etat du processus
  - Priorité
  - Pile d'exécution
  - Contexte d'exécution
  - ... (liste non exhaustive)
- Nombre de processus limité (par ex. 255)
- Enregistrés dans une table de processus
  - Insertion dans la table
  - Suppression de la table

Etats du processus



La pile d'exécution

- Elle sert à stocker l'état courant de l'éxecution du processus :
  - Valeur des registres courants
  - Position courante dans le code
  - La taille des données est donc de 32 bits
- Tableau de taille `STACK_SIZE` (à définir, une valeur possible : 1024 = taille d'une page)
- Initialement, le sommet de pile se situe à la position `STACK_SIZE - 1`
- A chaque insertion, le sommet de pile est décrémenté
- (Tout ça n'est qu'un rappel à priori)

Créer un processus

- (Voir TD1)
- La création du processus consiste à fournir le nécessaire pour l'exécution du processus :
  - Informations connues : nom du programme et fonction (ou code) à exécuter
  - Attribuer un numéro d'identifiant
  - Ajouter le processus à la table des processus
  - Allouer un espace de stockage pour la pile
  - Stocker la fonction en sommet de pile
  - Sauvegarder le sommet de pile
  - Le processus est prêt à s'exécuter

Terminer un processus

- La terminaison du processus est le traitement inverse de la création
- Libérer l'espace mémoire attribué au processus

Bloquer un processus

- Le blocage d'un processus s'effectuera en fonction du temps
- La durée sera spécifiée en secondes
- Lors du réveil, le processus se place dans l'état prêt
- Il faudra donc connaître la date de réveil

Travail à faire

Définition des processus

Définir le type processus ainsi que les fonctions associées

- (Prendre exemple sur le TD 1)
- Le nombre de processus est statique (un `#define NB_PROC nbprocessus`, je vous laisse choisir le nb de processus de votre système)
- Définir le type `pid_t` (par exemple : `typedef uint32_t pid_t`)
- Définir ce qu'est la table de processus
- On peut définir aussi les appels systèmes associés :
  - `typedef (void*) (*fnptr)()`
  - `pid_t fork(const char *name, fnptr function);`
  - `int exit();`
  - `pid_t getpid();`
  - `int sleep(int seconds);`

Ordonnancement

- 2 solutions :

Ordre dans la table des processus

- Un compteur de processus courant : 0 à NB_PROC
- Si le processus existe et n'est pas terminé, il est exécuté

Liste des processus prêts

- On maintient une liste de processus prêts
- Nécessite des fonctions pour gérer la file : insertion, extraction, liste vide, ...
- La liste peut être gérée sous la forme d'un tableau pour éviter la gestion mémoire compliquée (allocation mémoire ...)
- Un peu d'algorithmique ...

Ordonnancement

- Quand choisir le processus à exécuter ?

Explicitement

- Appel à la fonction qui gère l'ordonnancement dans le code du processus en cours :

```
void code_process() {
        ...
        schedule();
        while (1);
}
```

Régulièrement

- Utilisation du Timer pour appeler la fonction d'ordonnancement

La fonction d'ordonnancement

- `void schedule()`
- Son rôle : basculer d'un processus à l'autre
- Son fonctionnement :
  1. Choisir le processus à exécuter : Comment ?
  2. Changer le contexte d'exécution en cours pour le nouveau contexte = la fonction assembleur `ctx_sw`

Changement de contexte

- Consiste à sauvegarder l'état de l'exécution du processus en cours et de restaurer l'état du processus qui doit maintenant s'exécuter
- Principe : sauvegarde des registres essentiels dans la pile puis lecture de la pile du deuxième processus
- Réalisé par la fonction assembleur `ctx_sw`
  - Code fourni
  - Interface en C : `void ctx_sw(void *ctx_old, void *ctx_new)`
  - `ret`, en fin du code assembleur, met le registre CO (ou PC) à la bonne ligne du code qui doit s'exécuter $\rightarrow$ Magique, non ?

A quoi ressemble la fonction ctx_sw ?

```
ctx_sw:
# sauvegarde du contexte de l'ancien processus
        movl 4(%esp), %eax    # %eax <- 1er argument de ctx_sw
        movl %ebx, (%eax)     # mem[%eax] <- %ebx
        movl %esp, 4(%eax)    # mem[%eax+4] <- %esp
        movl %ebp, 8(%eax)    # mem[%eax+8] <- %ebp
        movl %esi, 12(%eax)   # ... ainsi de suite
        movl %edi, 16(%eax)
# restauration du contexte du nouveau processus
        movl 8(%esp), %eax   # %eax <- 2ème argument de ctx_sw
        movl (%eax), %ebx    # %ebx <- mem[%eax]
        movl 4(%eax), %esp   # %esp <- mem[%eax+4]
        movl 8(%eax), %ebp   # %ebp <- mem[%eax+8]
        movl 12(%eax), %esi # et ainsi de suite
        movl 16(%eax), %edi
# on passe la main au nouveau processus
        ret
```

- arguments de la fonction = tableaux qui contiennent la valeur des registres ebx, esp, ebp, esi et edi respectivement

Comment se sert-on de la fonction ctx_sw ?

- 5 registres à sauvegarder =

  ```
  uint32_t regs[5]
  ```

- regs[0] correspond à %ebx, regs[1] correspond à %esp (le sommet de pile), ...
- le tableau est à stocker dans la structure de chaque processus
- initialiser le tableau : tout à 0 sauf regs[1] = adresse du sommet de pile du processus
- appel de la fonction :

  ```
  ctx_sw(processus_actif->regs, processus_elu->regs);
  ```

Travail à faire

1. Construire la fonction `schedule()` qui réalise l'ordonnancement des processus par appel explicite
2. Créer 2 processus :
   - un processus `idle()` :

```
void idle() {
        ...
        // code de idle
        ...
        schedule();
        while (1) {
                hlt();
        }
```

   - un processus `processus1()` dont le code est dans `bin/processus1.c`
3. Tester le basculement d'un processus à l'autre
4. Modifier le code pour utiliser le Timer