Processus

Abdelkader Gouaïch

IUT de Montpellier

2014-2015

Définition d'un processus

- Il est important de distinguer : processus et programme
- Un processus est une instance d'un programme qui s'exécute
- Un programme est un fichier qui contient toutes les informations nécessaires pour la création d'un processus

Structure du fichier d'un programme

Le fichier d'un programme contient :

- méta informations sur le format binaire utilisé
  - Permet au noyau de savoir comment interpréter les instructions
  - Format a.out
  - Format COFF (Common Object File Format)
  - ELF (Executable and Linking Format)

Structure du fichier d'un programme

Le fichier d'un programme contient :

- Instructions en langage machine
  - résultat final de la compilation du fichier source
  - encodage des algorithmes et des fonctions

Structure du fichier d'un programme

Le fichier d'un programme contient :

- Adresse du point d'entrée du programme
  - Localisation de la première instruction à exécuter (main())

Structure du fichier d'un programme

Le fichier d'un programme contient :

- Données
  - Variables avec des valeurs définies par le programmeur
  - Évidemment il faut conserver ces valeurs dans le fichier du programme
  - Ce n'est pas le cas des variables non initialisées : il suffit de connaitre leur taille

Structure du fichier d'un programme

Le fichier d'un programme contient :

- Table des symboles
  - Table qui associe les noms des fonctions et variables avec des adresses (pages jaunes)

Structure du fichier d'un programme

Le fichier d'un programme contient :

- Informations sur les librairies externes
  - Une liste des librairies externes que le processus va utiliser durant son exécution
  - Les instructions de ces fonctions ne sont pas contenues dans le fichier du programme
  - Elles sont copiées à la création du processus (liens dynamiques)

Structure du fichier d'un programme

Le fichier d'un programme contient :

- Autres...

Le processus

Comment le noyau (royaume du noyau) considère un processus ?
Un processus :

- un espace d'adressage utilisateur (royaume de l'utilisateur)
- cet espace contient :
  - code du programme (instructions)
  - variables du programme (données fournies par le programmeur)
  - données de gestion utiles pour le noyau

Données de gestion

Le noyau maintient un ensemble de données de gestion pour un processus :

- Identifiants associés au processus
- table des descripteurs de fichiers ouverts
- table des signaux + les gestionnaires des signaux
- les ressources utilisées par le processus avec les limites autorisées
- répertoire courant (CWD)
- autres...

Process ID

- Chaque processus possède un unique identifiant (PID)
- pid_t getpid(**void**);

- Chaque processus possède un père
- Le père c'est le processus qui crée le processus fils (rappel : fork/vfork)
- Arbre des processus visibles avec la commande **pstree**
- pid_t getppid(**void**);

Le processus et sa mémoire

- Chaque processus va avoir *sa propre* mémoire !
- Une mémoire : des associations (adresse $\mapsto$ contenu)

@	Contenu
FFFF FFFF	00
FFFF FFFE	00
...	..
0000 0001	CA
0000 0000	FE

Sections (Segments)

- La mémoire c'est simplement un tableau avec des cases !
- Chaque case a une adresse et un contenu
- l'adresse s'exprime sur 32bits (ou 64bits)
- le contenu sur un byte (octet)
- nous allons structurer/segmenter cette table en zones contigües
- ces zones sont appelées des sections (segments)

Sections (Segments)

- Pourquoi segmenter ?
- On peut gérer plus facilement des droits particuliers pour certaines sections
- droit de lecture simple dans la zone qui contient les instructions
- doit de lecture et d'écriture dans les zones qui contiennent les variables

Les sections/segments d'un processus

- Segment *.text* : contient les instructions du programme à exécuter (lecture seule et partageable)

Les sections/segments d'un processus

- Segment *.text* : contient les instructions du programme à exécuter (lecture seule et partageable)
- Segment données non initialisées (*bss*) contient les variables non initialisées du programme

Les sections/segments d'un processus

- Segment *.text* : contient les instructions du programme à exécuter (lecture seule et partageable)
- Segment données non initialisées (*bss*) contient les variables non initialisées du programme
- segment données initialisées : contient les variables initialisées dans le programme.

Les sections/segments d'un processus

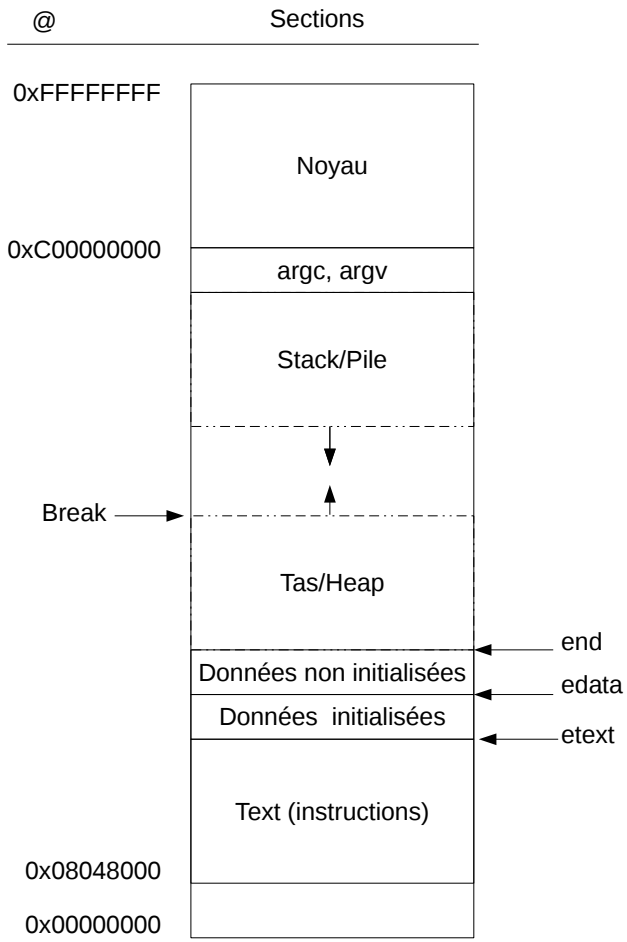
- Segment *.text* : contient les instructions du programme à exécuter (lecture seule et partageable)
- Segment données non initialisées (*bss*) contient les variables non initialisées du programme
- segment données initialisées : contient les variables initialisées dans le programme.
- Segment de la pile (stack) : ce segment à taille variable sert :
  - à stocker les variables automatiques des fonctions
  - des informations sur le contexte de l'appel utilisable lors du *return*

Les sections/segments d'un processus

- Segment *.text* : contient les instructions du programme à exécuter (lecture seule et partageable)
- Segment données non initialisées (*bss*) contient les variables non initialisées du programme
- segment données initialisées : contient les variables initialisées dans le programme.
- Segment de la pile (stack) : ce segment à taille variable sert :
  - à stocker les variables automatiques des fonctions
  - des informations sur le contexte de l'appel utilisable lors du *return*
- semgment du tas (heap) : ce segment à taille variable sert à ranger les variables allouées dynamiquement (malloc/brk). La fin de ce segment est appelée *programme break*.

En C, vous pouvez avoir des informations sur les sections bss, data et text :

extern char etext; *//&etext adresse de la section instructions*
extern char edata ; *//&edata adresse section data*
extern char end; *//&end debut section heap*


@
Sections
0xFFFFFFFF
Noyau
0xC00000000
argc, argv
Stack/Pile
Break
Tas/Heap
end
Données non initialisées
edata
Données initialisées
etext
Text (instructions)
0x08048000
0x00000000

- La mémoire du processus présentée est une mémoire logique
- Nous faisons croire au processus qu'il existe une table de mémoire avec des adresses sur 32 ou 64 bits
- En réalité la mémoire physique (RAM) est souvent inférieure à 2^{32} (2^{64}) !
- Nous parlons alors de mémoire virtuelle !
- C'est la technique qui permet de faire croire au processus qu'il dispose d'un espace contigu de 2^{32} (2^{64}) bytes

Principes de la mémoire virtuelle

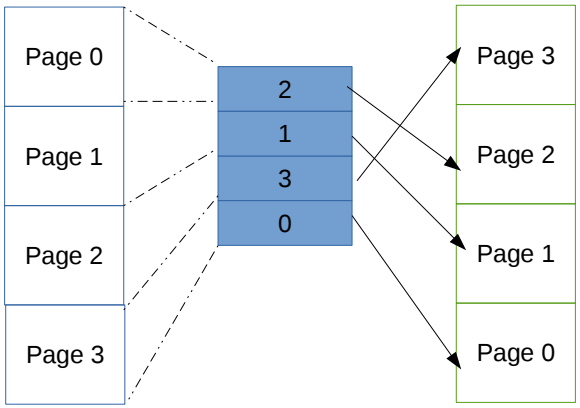
- L'espace d'adressage est divisé en *pages* à *taille fixe*
- La RAM (mémoire physique) est également divisée en *page frames* de taille *égale* aux pages
- L'idée est de maintenir dans la RAM que certaines pages dans les page frames
- Une fois la limite de la RAM atteinte, on stocke les pages sur le disque (partition swap)

Défaut de pages

- Si une page n'est plus dans la RAM mais dans le swap que se passe-t-il si un processus veut y accéder ?
- Il se produit un défaut de page !
- Dans ce cas, le noyau décharge une page de la RAM vers le disque et déplace la page demandée vers la RAM

Réalisation de la mémoire virtuelle

- Le noyau maintient un mapping entre les pages (MV) et les page frames (RAM) :
  - indique le numéro de la page frame dans la RAM
  - indique que la page est sur le disque



Restriction sur les @

- En pratique, un processus ne peut pas accéder à tout l'espace d'adressage de la MV (2^{32})
- Toutes les pages ne peuvent pas être dans le mapping pages/page frames
- Si un processus veut accéder à une page non autorisée, il reçoit le signal SIGSEGV

Augmentation / Diminution de l'espace accessible

- Durant l'exécution d'un processus l'espace accessible évolue avec les cas suivants :
  - La pile (stack) atteint un niveau jamais atteint auparavant
  - Allocation dynamique de la moire sur le tas (malloc, brk)
  - fonction d'attachement/partage de la mémoire (shmat
  - mapping de mémoire/fichier (mmap)

La pile

- Rappel : La pile va contenir les cadres (frames) nécessaires l'exécution *d'un* appel de fonction
- La pile va augmenter/diminuer avec les appels/retours des fonctions
- Linux place la pile vers la partie haute de la mémoire
- Elle augmente donc vers le bas
- Un registre spécial (Stack Pointer) indique la tête de la pile (où mettre le nouveau cadre/frame)

Appel de fonction

- Chaque appel de fonction nécessite la création d'une frame
- Frame contient
  - arguments de l'appel
  - les variables automatiques (locales à la fonction)
  - sauvegarde des registres (permet de rétablir le contexte lors du return)

Appel de fonction

- Avec 'return' l'appel de la fonction est terminé
- Sa frame est dépilée de la stack
- Pour cela, il suffit simplement de remettre SP à son ancienne valeur
- Avec cette opération toutes les variables automatiques de la fonction n'existent plus !
- Ceci n'est pas le cas des variables statiques qui ne sont pas dans la pile (durée de vie du processus)

Exemple

```
void f2(int i)
{
int j = 1;
printf("%d",j+i);
return;
}
```

Exemple

```
void f1(int i)
{
if(i == 0)
{
f2(−1);
return;
}
else
{
f1(i−1);
return;
}
}
```

Exemple

```
int main(int argc, char** argv)
{
f1(2);
}
```