Résumé TE1

Loïc Herman

Architecture de base

Séparation des instructions privilégies / sensibles du reste au moyen d'un kernel.
Ainsi, il y a 2 modes d'exécution pour le CPU:

- User mode
- Kernel mode (privileged mode)

Transition user → kernel uniquement via des interruptions logicielles ou matérielles.

Interruptions

Une interruption matérielle vient de l'IRQ (Interrupt Request) du CPU, c'est une interruption asynchrone.

Une interruption logicielle provient de l'exécution d'une instruction, c'est une interruption synchrone.
Quelques types d'interruptions logicielles:

- syscall
- undefined instruction
- division by zero
- data abort

Gestion des interruptions

1. Arrêt de l'exécution du programme en cours
2. Passage en mode kernel
3. Activation de l'ISR (Interrupt Service Routine)
4. Passage en mode user
5. Reprise de l'exécution du programme en cours

Architecture d'un OS

Il existe deux types: **monolith** et **microkernel**.

Architecture monolithique

Système performant car pas de changement de mode d'exécution, tous les sous-système du kernel sont placé dans un seul kernel space. C'est cependant moins sécurisé car un bug dans un sous-système peut planter tout le système.

Architecture microkernel

Système plus robuste car il y a un nombre minime de sous-système est placé dans le kernel space, le reste est placé dans le user space et la communication est faite via d'autres moyens.

Syscalls et images binaires

Permet d'appeler un service du kernel depuis le user space.

Gestion des appels

La fonction d'un appel syscall en C est en fait un *stub*, qui va en fait gérer d'inscrire les arguments nécessaire et faire appel à l'instruction système.

La procédure d'exécution est la suivante:

- Le stub va inscrire dans un registre (SO3: `R7`) le code correspondant au syscall, puis les arguments nécessaires dans les registres suivants (SO3: `R0` à `R3`).
- L'instruction système va ensuite être appelée, et va passer en mode kernel.
- Le kernel va ensuite récupérer le code du syscall et les arguments, puis va appeler la fonction correspondante.
- La fonction va ensuite retourner le résultat dans le registre `R0` et le kernel va retourner en mode user.

Gestion d'une image binaire

Un compilateur va génerer un *fichier objet*, un résultat intérmédiaire contenant les instructions machines ainsi que les données statiques et les espaces statiques réservés. Ce fichier objet n'est pas exécutable en tant que tel, car il manque des informations d'exécution système (librairies à lier dynamiquement, par exemple).

Un linker va ensuite lier les résultats intérmédiaires et les assembler dans un fichier exécutables qui contient les différents fichiers objets ainsi que les informations d'exécution système (librairies à lier dynamiquement, par exemple).

Librairies

Quand un programme fait appel à une librairie externe, le compilateur insère une adresse symbolique que le linker va remplacer par l'adresse non-symbolique de la fonction dans la librairie une fois celle-ci liée.

Processus

Un processus a deux contextes: le contexte d'exécution et le contexte mémoire.

Le contexte d'exécution contient les instructions et registres du CPU. Le contexte mémoire contient les données du processus, les variables, etc.

Format des processus

Les processus sont organisés de manière hierarchique. Au démarrage, après la phase d'initialisation, un premier processus (par exemple, init) est démarré. Les processus suivants seront donc des processus enfants du P_0.

Quand un processus se termine, le processus parent doit être informé afin de pouvoir libérer les ressources allouées au processus enfant.

Un processus parent qui se termine alors qu'il a encore des enfants causera les processus enfants à devenir orphelins, et seront adoptés par un autre processus parent choisi par le système (souvent P_0).

Zones mémoires d'un processus

Un processus a 4 zones mémoires:

1. Stack $T_0, T_1, ..., T_n$ (variables locales, arguments, adresse de retour. Local à un thread, chaque thread a sa propre stack qui s'étend vers le bas)
2. ...
3. Heap (zone d'allocation dynamique globale au contextes d'exécution, s'étend vers le haut)
4. BSS (variables **statiques non**-initialisées)
5. Data (variables **statiques** initialisées et constantes)
6. Code (instructions CPU)

BSS

Variables statiques non-initialisées.

```
static int i;
static char a[12];
```

Data

Variables statiques initialisées.

```
int i = 3;
char a[] = "Hello World";
static int b = 2023;

void foo(void) {
  static int c = 2023;
}
```

PCB

Le PCB (Process Control Block) est une structure de données qui contient les informations sur un processus. Il est stocké dans le kernel space. Il est surtout utilisé lors d'un changement de contexte. Chaque processus a son propre PCB, qui contient les informations suivantes:

- État du processus
- Espace d'adressage
- Pointeur vers le heap
- Références vers les processus enfants
- Descripteurs des fichiers ouverts
- Contextes d'exécution
  - État du contexte
  - Pointeur sur l'instruction en cours
  - Registres
  - Stack

Forking processes

Le syscall `fork` permet de créer un nouveau processus **enfant**. Toutes les données du processus parent sera copié dans le processus enfant. Le PCB sera donc copié et la MMU devra être reconfigurée pour les adresses du processus enfant. Le processus enfant va ensuite être ajouté à la liste des processus enfants du processus parent.

Attente

Via le syscall `waitpid()`, il est possible d'attendre qu'un processus enfant se termine en référençant son PID. La fonction permettra d'assigner le code de retour du processus enfant à une variable.

Il est obligatoire d'attendre l'exécution d'un processus enfant depuis le parent, afin de s'assurer qu'on ne laisse pas de processus orphelins (qui devront être gérés par le kernel).

Terminaison

Un appel au syscall `exit(int status)` permettra d'arrêter le processus et transmettra le code de retour au processus parent (via une variable du PCB).

Appel à un autre programme

Depuis un processus enfant, on peut faire appel à un syscall de la famille `exec` (en général `execv` ou `execve`). Ce syscall va permettre de charger un nouveau programme dans le processus enfant, et va remplacer le code du processus enfant par celui du nouveau programme.

Changement de contexte

Plusieurs événements peuvent causer un changement de contexte:

- Un processus se termine
- Un processus est mis en attente
- L'ordonnanceur le décide

Une opération de changement de contexte est couteuse, car elle nécessite de sauvegarder le contexte du processus courant dans le PCB et de charger le contexte du processus suivant. Les registres, le cache, et la MMU doivent aussi être reconfigurés, puis le PCB doit aussi être mis à jour.

Les opérations induites par un changement de contexte sont les suivantes:

1. Sauvegarde du contexte du processus courant
2. Chargement du contexte du processus suivant
3. Reconfiguration de la MMU
4. Mise à jour du PCB

États
Un processus peut avoir les états suivants:
- New
- Ready
- Running (peut revenir sur ready dans le cas où le processus est préempté)
- Waiting (uniquement depuis running, et peut revenir seulement sur ready)
- Zombie (état final transitoire avant que le processus parent s'occupe de récupérer le code de sortie et nettoyer définitivement le PCB)

Threading
La création de processus est une opération couteuse, car elle nécessite de copier le PCB et de reconfigurer la MMU. Les threads permettent de créer des processus légers, qui partagent le même espace d'adressage et le même PCB. Les threads sont donc plus légers à créer et à gérer.

Création
Un thread est créé via le syscall `pthread_create()`. Il prend en paramètre un pointeur vers une fonction qui sera exécutée par le thread, ainsi que des arguments pour cette fonction.
Un stack sera donc alloué par le kernel pour ce nouveau contexte d'éxecution.

Le thread aura donc accès au même contexte mémoire que le processus parent, et pourra donc accéder aux mêmes variables globales et au même heap.

Gestion dans le PCB
Le thread recevra un TCB (Thread Control Block) qui sera stocké dans le PCB du processus parent.
Le TCB contient les informations suivantes:

- Pointeur d'instruction
- Registres
- Stack pointer
- État du thread
- Priorité

Ce TCB sera utilisé lors des changements de contexte entre les threads du processus.

Attente d'un thread
Via le syscall `pthread_exit()`, il est possible de terminer un thread et de donner une valeur de retour, qui pourra être récupérée par le processus parent. Cet appel n'est pas obligatoire.

Via le syscall `pthread_join()`, il est possible d'attendre la terminaison d'un thread. La fonction permettra d'assigner la valeur de retour du thread à une variable (si la fonction `pthread_exit()` a été appelée).

File descriptors
Un file descriptor est un entier qui permet d'identifier un fichier ouvert par un processus. Il est unique pour chaque fichier ouvert par un processus.

Un tableau de file descriptors est stocké dans le PCB du processus, et est initialisé avec les 3 premiers file descriptors: `stdin`, `stdout` et `stderr`.
Ils font référence à une table globale de file descriptors ouverts, qui est gérée par le kernel.

Redirection
Avec le syscall `dup2()`, il est possible de rediriger un file descriptor vers un autre file descriptor.
L'ensemble des redirections sont conservées lors d'un appel à `fork()`.

Pipes
Les pipes permettent de gérer la communication d'un processus à un autre. Ils sont unidirectionnels et sont gérés par le kernel. La capacité d'un pipe est limitée (en général à 4 Ko).

Deux file descriptors sont créés par pipe: `pipe_fd[0]` pour la lecture (éqiv `stdin`), et `pipe_fd[1]` pour l'écriture (éqiv `stdout`).

Création

Il existe deux types de pipes: les pipes anonymes et les pipes nommés.

Pipes anonymes

Les pipes anonymes sont créés via le syscall `pipe()`. Ils sont créés dans le kernel et sont donc partagés entre les processus. Ils sont unidirectionnels et ne peuvent être utilisés que par des processus qui ont un lien de parenté.

Pipes nommés

Les pipes nommés sont créés via le syscall `mkfifo()`. Ils sont créés au moyen d'un fichier virtuel par le kernel dans le système de fichier en userland. Ils peuvent être utilisés par n'importe quel processus.

Signaux

Les signaux sont des interruptions logicielles asynchrones de type event. Ils sont envoyés par le kernel à un processus. Ils peuvent être envoyés par le kernel ou par un autre processus.

Un processus peut ignorer un signal, le masquer, le traiter, ou le laisser par défaut. Dans tous les cas, un handler sera appelé dans le userland.

Il ne peut y avoir qu'un seul signal du même type en attente, et ils seront pris en compte dès que le processus sera dans l'état running.

Envoi et réception

Un signal peut être envoyé via le syscall `kill()`. Il prend en paramètre le PID du processus à qui envoyer le signal, ainsi que le type de signal à envoyer.

Un signal peut être reçu via le syscall `signal()`. Il prend en paramètre un pointeur vers une fonction qui permettra de traiter le signal, ainsi que le type de signal qui doit être pris.

Ignorer un signal

On peut faire un appel à la méthode `signal()` avec le paramètre `SIG_IGN` pour ignorer un signal, ou `SIG_DFL` pour le remettre par défaut.

Sockets

Les sockets sont des objets de communication entre processus. Ils sont créés via le syscall `socket()`, qui prend en paramètre le type de socket à créer (TCP ou UDP).

Serveur

Un serveur TCP est créé via les appels suivants:

1. `ofd = socket()`
2. `bind()`
3. `listen()`
4. `nfd = accept()` (fork automatique)
5. `len = recv()` / `send()`
6. `close(nfd)` (retour à accept)
7. `close(ofd)`

Client

Un client TCP est créé via les appels suivants:

1. `fd = socket()`
2. `connect()`
3. `send()` / `len = recv()`
4. `close(fd)`