Systèmes et Réseaux – ENS L3 Info.
Année 2017-2018

Cours 1 : Un micro-noyau en OCaml

8 février 2018

L'objectif est de programmer un micro-noyau simplifié d'un système d'exploitation. [1] Son rôle est d'ordonnancer les processus en fonction de leur priorité, de gérer la création et l'arrêt de processus ainsi que la communication entre processus. On suppose que l'architecture machine est capable d'exécuter un seul processus à la fois et qu'elle possède cinq registres (`r0`, `r1`, `r2`, `r3` et `r5`). Le micro-noyau réagit à deux types d'événements : l'interruption d'un compteur de temps ("timer") et des interruptions logicielles ("system trap" ou "software interrupt").

On ignorera les détails des processus de l'utilisateur, en supposant qu'ils peuvent changer le contenu des registres et générer des appels système arbitraires.

1 Interface et structures de données pour les appels système

Quand un appel système est déclenché, le micro-noyau lit le contenu des registres pour déterminer l'appel effectué et les arguments de cet appel. Il réagit en effectuant l'appel (par exemple, la mise à jour de l'état du système) et en plaçant les valeurs de retour dans les registres. Le code des appels systèmes est défini ci-dessous.

Registre `r0`	appel système correspondant
0	`new_channel`
1	`send`
2	`receive`
3	`fork`
4	`exit`
5	`wait`

Si un processus p effectue un appel système invalide (par exemple, en donnant la valeur 10 à `r0`), le noyau n'exécute aucun code correspondant. Il place la valeur -1 dans `r0`. Le noyau est caractérisé par les constantes et types suivants, définis en OCaml.

```
let max_time_slices = 5   (* 0 <= t < max_time_slices *)
let max_priority    = 15  (* 0 <= p <= max_priority *)
let num_processes  = 32
let num_channels   = 128
let num_registers  = 5

type pid = int (* process id *)
type chanid = int (* channel id *)
type value = int (* values transmitted on channels *)
```

1. Le sujet reprend le principe du micro-noyau seL4, en le simplifiant et en remplaçant le modèle monadique Haskell par un modèle impératif en OCaml.

```
type interrupt = int (* software interrupt *)
type priority = int (* priority of a process *)

type registers = {
  r0    : int;
  r1    : int;
  r2    : int;
  r3    : int;
  r4    : int; }

type process_state =
  | Free (* non allocated process *)
  | BlockedWriting of chanid
  | BlockedReading of chanid list
  | Waiting
  | Runnable
  | Zombie

type process = {
  mutable parent_id   : pid;
  mutable state       : process_state;
  mutable slices_left : int;
  saved_context       : int array;
}

type channel_state =
  | Unused (* non allocated channel *)
  | Sender of pid * priority * value
  | Receivers of (pid * priority) list

type state = {                    (* kernel state *)
  mutable curr_pid   : pid;       (* process id of the running process *)
  mutable curr_prio  : priority; (* its priority *)
  registers  : int array;         (* its registers *)
  processes  : process array;        (* the set of processes *)
  channels   : channel_state array; (* the set of channels *)
  runqueues  : pid list array;    (* the set of processes ordered by priority *)
}

let get_registers { registers = registers } = {
  r0 = registers.(0); r1 = registers.(1);
  r2 = registers.(2); r3 = registers.(3);
  r4 = registers.(4); }

let set_registers {registers = registers } { r0; r1; r2; r3; r4 } =
  registers.(0) <- r0; registers.(1) <- r1;
  registers.(2) <- r2; registers.(3) <- r3;
  registers.(4) <- r4

let get_current { curr_pid = c } = c
```

```
type event = | Timer | SysCall

type syscall =
  | Send of chanid * value
  | Recv of chanid list
  | Fork of priority * value * value * value
  | Wait
  | Exit
  | NewChannel
  | Invalid
```

Question 1. Écrire une fonction OCaml `decode: state -> syscall` qui décode la valeur des registres (champ `registers`) et détermine l'appel système.

On décrit maintenant chacun des six appels système. Un appel système modifie l'état du système (défini par le type `state`).

2 Création, terminaison et attente de processus

2.1 L'appel système `fork` :

Cet appel crée un nouveau processus fils. Chaque processus est associé à une priorité comprise entre 0 (la plus basse) et 15 (la plus haute). Le registre `r1` spécifie la priorité du processus créé. L'appel système se termine sans créer de processus et en plaçant 0 dans `r0` si la priorité donnée est strictement plus grande que la priorité du processus qui crée le processus fils.

Si la priorité est valide et qu'un nouveau processus peut être créé, `r0` reçoit la valeur 1 et `r1` reçoit le numéro du processus créé. Si un nouveau processus ne peut pas être créé, `r0` reçoit la valeur 0.

Dans le processus fils créé, `r0` est initialisé à 2, `r1` est initialisé au numéro de processus du père (qui a fait l'appel à `fork`), et les autres registres (`r2`, `r3` et `r4`) sont copié du processus parent.

`num_processes` est le nombre maximum de processus pouvant être créés.

L'appel système `exit` :

Cet appel termine l'exécution du processus l'exécutant. Son argument est plaçé dans le registre `r1`. C'est la valeur de retour de l'appel à `exit`.

Un processus qui exécute un appel à `exit` entre dans l'état `Zombie` jusqu'à l'exécution de l'appel système `wait` qui récupèrera la valeur de retour.

Un fils d'un processus terminé devient orphelin ; l'identifiant de son père devient alors le processus 1 (processus `init`).

2.2 L'appel système `wait` :

Le processus est en attente (mode `Waiting`) jusqu'à ce qu'un de ses fils meure. S'il ne reste plus aucun fils, l'appel système rend la main immédiatement en plaçant 0 dans `r0`.

S'il reste un processus fils dans le mode `Zombie` ou lorsque un fils termine, l'appel à `wait` termine en plaçant 1 dans `r0`, l'identifiant du fils dans `r1` et la valeur de retour de ce fils dans `r2`. S'il y a plusieurs fils dans le mode `Zombie`, l'un d'eux est choisi arbitrairement.

Question 2. Donner une implémentation de l'appel système `fork state nprio d0 d1 d2` où `state` est l'état du système, `nprio` est la priorité à donner au processus fils, `d0`, `d1` et `d2` sont les valeurs à passer au fils pour initialiser ses trois derniers registres. `fork` est de type `state -> int -> int -> int -> int -> unit`.

Question 3. Donner une implémentation de l'appel système `exit: state -> unit`.

Question 4. Donner une implémentation de l'appel système `wait: state -> bool`. Le résultat de `wait state` est vrai s'il est nécessaire de réordonnancer le processus courant (c'est-à-dire le replacer dans l'état du système et choisir un nouveau processus à ordonnancer).

3 Communication entre processus

La communication entre processus s'effectue par envoi et écriture dans un canal, suivant un protocole de *rendez-vous* ("handshake").

3.1 L'appel système `new_channel` :

Les processus au sein du système communiquent par rendez-vous sur des canaux numérotés. Cet appel système crée un nouveau canal. La valeur de retour `r0` de cet appel système est soit le numéro du canal créé ou une valeur négative si un nouveau canal n'a pas pu être créé. `new_channel` est la seule opération pour créer un nouveau canal ; tous les autres sont invalides. `num_channels` est le nombre maximal de canaux pouvant être créés.

L'appel système `send` :

Cet appel prend deux arguments : `r1` est le canal sur lequel une valeur est envoyée ; `r2` contient la valeur à envoyer. Le numéro du canal doit être valide (c'est-à-dire avoir été créé par un appel à `new_channel`).

Si un autre processus est déjà en train d'envoyer une valeur sur le canal (c'est-à-dire qu'il est bloqué en attente d'un récepteur) ou si le canal est invalide, la valeur de retour de l'appel système `send` placée dans `r0` est 0.

Si un autre processus est déjà en attente sur le canal, l'appel `send` réussit immédiatement. Le processus en attente sur le canal passe alors du mode `Blocked` au mode `Runnable`. Sinon le processus emetteur se bloque jusqu'à l'arrivée d'un recepteur. Lorsque plusieurs processus récepteurs sont en attente, le processus de plus forte priorité est choisi arbitrairement et les autres restent bloqués. La valeur de retour de l'appel système est 1 (registre `r0`).

L'appel système `receive` :

L'appel système `receive` permet de se synchroniser avec au plus 4 canaux, spécifiés dans les registres `r1` à `r4`. Cet appel permet donc d'écouter sur plusieurs canaux à la fois. Elle réussit lorsqu'un rendez-vous a lieu avec un des émetteurs. Les canaux invalides sont ignorés. Si aucun canal valide n'est spécifié, l'appel système rend la main immédiatement en plaçant 0 dans `r0`.

Si un ou plusieurs émetteurs sont en attente sur un des canaux valides, l'un est choisi arbitrairement et l'appel `receive` rend la main immédiatement en plaçant 1 dans `r0` et en donnant à `r1` la valeur du canal choisi pour la réception et en plaçant dans `r2` la valeur envoyée sur le canal. Sinon, le récepteur bloque jusqu'à ce qu'une emission ait lieu sur un des canaux spécifiés.

Question 5. Donner une implémentation de l'appel système `new_channel: state -> unit`.

Question 6. Donner une implémentation de l'appel système `send`. Il a pour signature `send: state -> chanid -> value -> bool`.

Question 7. Donner une implémentation de l'appel système `receive`. Sa signature est `receive: state -> chanid list -> unit`.

État initial du système et ordonnancement des processus

Le système démarre en créant deux processus. Le processus `idle` de numéro 0, de priorité 0 et de père égal à lui-même ; le processus `init` de numéro 1, de priorité 15 et de père égal à lui-même. Tous les registres sont initialisés à 0 et aucun canal n'est créé. Vous pouvez supposer que le processus `idle` est toujours exécutable et que ni le processus `idle` ni le processus `init` ne terminent jamais.

On suppose que l'état observable du système est l'idenfiant du processus en cours d'exécution et le contenu des cinq registres. On supposera que l'état interne du système d'exploitation ne peut être observé ni modifié de l'exterieur.

Le rôle de noyau est d'élir un processus à exécuter parmi la liste des processus et en lui allouant un quantum de temps maximum et de traiter les appels système considérés précédemment. Un processus est exécutable ou prêt (`Runnable`) s'il n'est pas bloqué sur un canal ni n'attend l'un de ses fils, et n'est pas un zombie.

Le noyau choisit les processus prêts de priorité la plus forte avec un protocole "round robin" : lorsqu'un processus en cours d'exécution est interrompu, il retourne en fin de queue parmi les processus de même priorité.

Le système reçoit des interruptions périodiques venant d'une horloge externe (timer). Une interruption indique la fin d'une tranche de temps (time slice). Un processus ne peut pas s'exécuter pendant un durée égale à au plus de cinq tranches de temps (`max_time_slices`). Ce temps n'est décompté que pour le processus en cours d'exécution. Le noyau doit donc mettre à jour le compteur de temps du processus en cours d'exécution.

Un changement de contexte (changement du processus en cours d'exécution) se produit dans deux cas : 1/ lorsque le processus se bloque (par exemple lorqu'il exécute un `send` et qu'aucun processus n'écoute sur le canal correspondant) ; 2/ il est préempté parce qu'il a atteint sa durée maximale d'exécution. Les valeurs des registres doivent alors être sauvegardées et restaurées au travers du changement de contexte.

Question 8. Écrire une fonction `transition: event -> state -> unit` qui, en fonction de l'événement reçu, exécute le code de l'appel système, fait avancer le pas de temps du processus en cours d'exécution ou élit un processus à exécuter.