Systèmes d'exploitation

Marc Pouzet et **Timothy Bourke**

Cours 1: micro-noyau

Un micro-noyau en OCaml

Ce que nous allons faire aujourd'hui.

- Programmer un micro-noyau très simplifié d'un système d'exploitation en OCaml. [1]
- Un micro-noyau (par opposition à monolithique) : retenir l'essentiel d'un noyau de système, avec le moins possible de fonctions exécutées en mode "super-utilisateur" qui ont un accès non protégé aux ressources.
- Gérer les processus, la communication entre eux et la mémoire virtuelle.
- Dans un micro-noyau, le système de fichier ou un driver de disque, par exemple, est une application (exécution en mode "utilisateur").

1. Nous reprenons ici les principes du micro-noyau seL4, en le simplifiant et en remplaçant le modèle monadique Haskell par un modèle impératif en OCaml. Voir : `https://sel4.systems/Info/Docs/seL4-manual-latest.pdf`

Fonctionnalités

- 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.
- Elle possède cinq registres (`r0`, `r1`, `r2`, `r3` et `r4`).
- 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 ignore les détails des processus de l'utilisateur ; ils peuvent changer le contenu des registres et générer des appels système arbitraires.

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 ainsi :

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 (e.g., 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 *)
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;
}
```

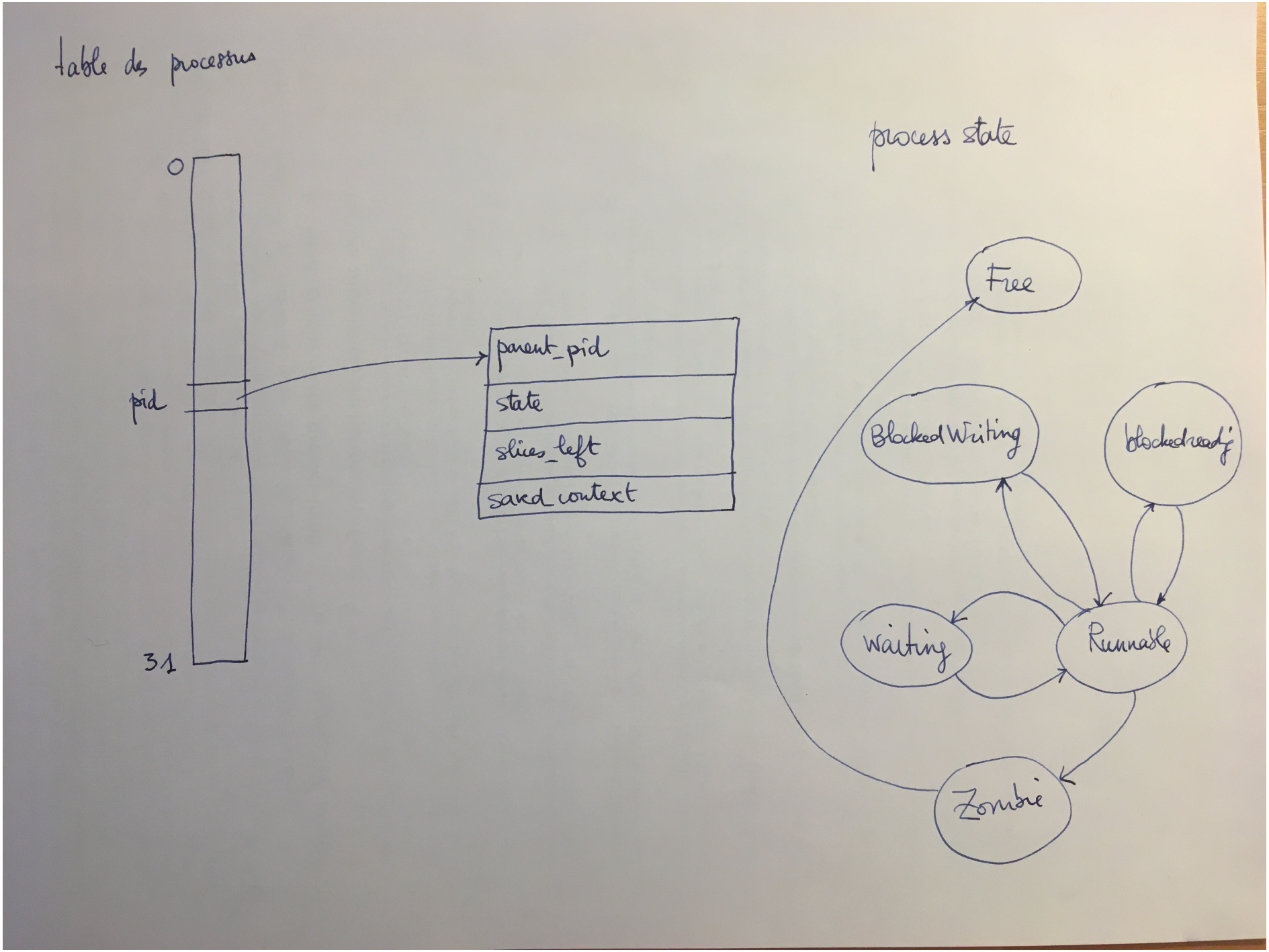


table des processus
0
pid
31
parent_pid
state
slices_left
saved_context
process state
Free
BlockedWriting
blockedreadf
waiting
Runnable
Zombie

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

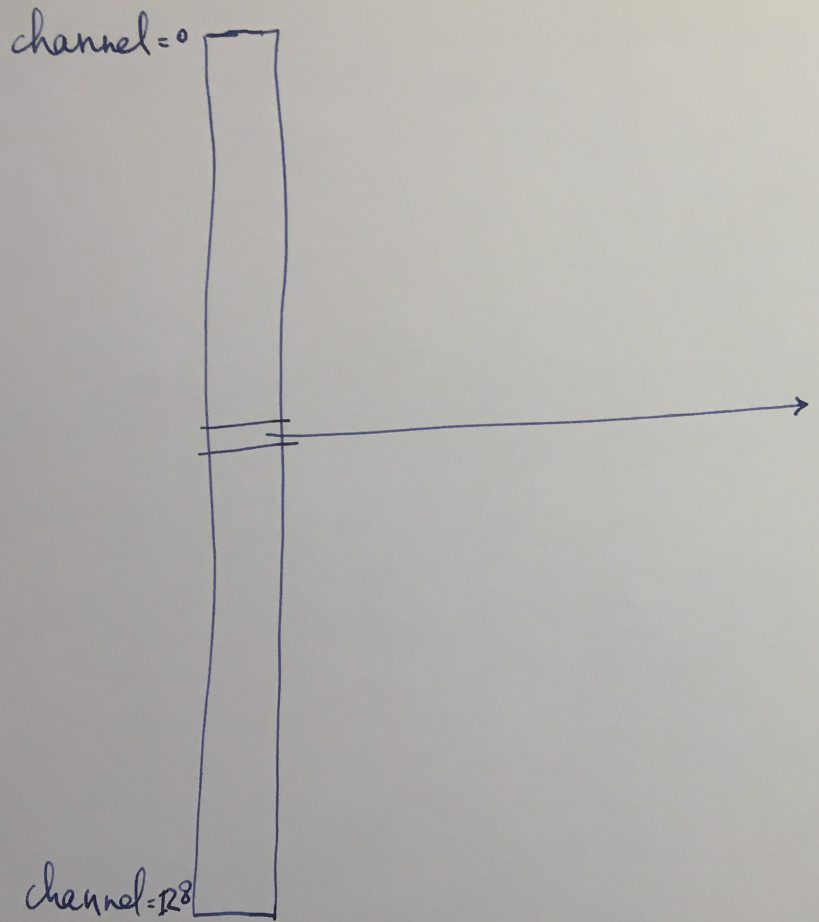
```
let set_registers {registers } { r0; r1; r2; r3; r4 } =
  registers.(0) <- r0; registers.(1) <- r1;
  registers.(2) <- r2; registers.(3) <- r3;
  registers.(4) <- r4

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
```

table des canaux :



Unused

+

Sender (pid , priority , value)

Receivers [pid_0 , $prio_0$; ... ; pid_n , $prio_n$]

Question

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

Réponse

```
let decode { registers } =
match registers.(0) with
| 0 -> NewChannel
| 1 -> Send (registers.(1), registers.(2))
| 2 -> Recv [registers.(1);
              registers.(2);
              registers.(3);
              registers.(4)]
| 3 -> Fork (registers.(1), registers.(2),
              registers.(3), registers.(4))
| 4 -> Exit
| 5 -> Wait
| _ -> Invalid
```

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 copiés 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`).

L'appel système `wait`

- Le processus est en attente (mode `Waiting`) jusqu'à ce qu'un de ses fils meurt.
- 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

Donner une implémentation de l'appel système `fork`, de type :

```
state -> int -> int -> int -> int -> unit
```

où :

`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.

Appel système Fork

```
let fork { curr_pid; curr_prio; registers; processes; runqueues }
         nprio d0 d1 d2 =
  let rec new_proc i =
    if i >= num_processes then None
    else if processes.(i).state = Free then
      let np = processes.(i) in
      np.parent_id <- curr_pid;
      np.state <- Runnable;
      np.slices_left <- max_time_slices;
      np.saved_context.(0) <- 2;
      np.saved_context.(1) <- curr_pid;
      np.saved_context.(2) <- d0;
      np.saved_context.(3) <- d1;
      np.saved_context.(4) <- d2;
      Some i
    else new_proc (i + 1)
  in
  match new_proc 0 with
  | None -> registers.(0) <- 0
  | Some npid -> begin
      registers.(0) <- 1;
      registers.(1) <- npid;
      runqueues.(nprio) <- runqueues.(nprio) @ [npid]
    end
```

Question

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

Appel système Exit

```
let exit { curr_pid; curr_prio; registers; processes; runqueues } =
  let { parent_id } as p = processes.(curr_pid) in

  (* tous mes fils ont maintenant comme pere le proc. no. 1 *)
  let f p = if p.parent_id = curr_pid then p.parent_id <- 1 in
  Array.iter f processes;

  runqueues.(curr_prio) <-
    List.filter (fun pid -> pid <> curr_pid) runqueues.(curr_prio);

  if processes.(parent_id).state = Waiting
  then begin
    processes.(parent_id).state <- Runnable;
    processes.(curr_pid).state <- Free;
    let saved_registers = processes.(parent_id).saved_context in
    saved_registers.(0) <- 1;
    saved_registers.(1) <- curr_pid;
    saved_registers.(2) <- registers.(0)
  end
  else processes.(curr_pid).state <- Zombie
```

Question

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).

Appel système `Wait`

```
let wait {curr_pid; registers; processes} =
let rec already_dead has_child i =
  if i = num_processes then has_child, None
  else begin
    let { state; parent_id; saved_context} = processes.(i) in
    if state = Zombie && parent_id = curr_pid
    then true, Some (i, saved_context.(0))
    else already_dead (has_child || parent_id = curr_pid) (i + 1)
  end
in
match already_dead false 0 with
| has_child, None ->
    if has_child
    then (processes.(curr_pid).state <- Waiting; true)
    else (registers.(0) <- 0; false)
| _, Some (pid, v) ->
    processes.(pid).state <- Free;
    registers.(0) <- 1;
    registers.(1) <- pid;
    registers.(2) <- v;
    false
```

Communication entre processus

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

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 émetteur 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 émission ait lieu sur un des canaux spécifiés.

Question

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

L'appel système `new_channel`

```
let new_channel {registers; channels} =
let rec new_channel i =
  if i >= num_channels then -1
  else if channels.(i) = Unused
       then (channels.(i) <- Receivers []; i)
       else new_channel (i + 1)
in
registers.(0) <- new_channel 0
```

Question

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

```
(* il y a deja quelqu'un qui attend sur le canal *)
let release_receiver {channels; processes} rid =
  let clear ch =
    channels.(ch) <-
      match channels.(ch) with
      | Receivers rs ->
          Receivers (List.filter (fun (pid, _) -> pid <> rid) rs)
      | v -> v
  in
  (match processes.(rid).state with
   | BlockedReading rchs -> List.iter clear rchs
   | _ -> assert false);
  processes.(rid).state <- Runnable
```

```
let send
  ({curr_pid; curr_prio; registers; processes; channels} as s) ch v
  match channels.(ch) with
  | Sender _ | Unused -> (registers.(0) <- 0; false)

  | Receivers [] ->
     (* personne n'attend sur le canal *)
     channels.(ch) <- Sender (curr_pid, curr_prio, v);
     processes.(curr_pid).state <- BlockedWriting ch;
     true

  | Receivers ((rid, rprio)::rs) ->
      release_receiver s rid;
      let saved_registers = processes.(rid).saved_context in
      saved_registers.(0) <- 1;
      saved_registers.(1) <- ch;
      saved_registers.(2) <- v;
      registers.(0) <- 1;
      rprio >= curr_prio
```

Question

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

```
let recv {curr_pid; curr_prio; registers; processes; channels} chs =
  let rec sender_ready chans =
    match chans with
    | [] -> None
    | ch::chs ->
        match channels.(ch) with
        | Sender (sid, sprio, sv) -> Some (ch, sid, sprio, sv)
        | _ -> sender_ready chs  in
  let rec add_to_receivers blocked chans =
     match chans with
     | [] -> blocked
     | ch::chs ->
        match channels.(ch) with
          Unused | Sender _ -> add_to_receivers blocked chs
        | Receivers rs ->
            channels.(ch) <-
              Receivers (insert_receiver (curr_pid, curr_prio) rs);
            add_to_receivers (ch::blocked) chs
  in ...
```

```
  match sender_ready chs with
| Some (ch, sid, sprio, sv) ->
    channels.(ch) <- Receivers [];
    processes.(sid).state <- Runnable;
    registers.(0) <- 1;
    registers.(1) <- ch;
    registers.(2) <- sv;
    sprio >= curr_prio
| None ->
    match add_to_receivers [] chs with
    | [] -> (registers.(0) <- 0; false)
    | bchs -> (processes.(curr_pid).state <- BlockedReading bchs;
               true)
```

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.
- On peut 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 suppose 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 processu s et en lui allouant un quantum de temps maximum ;
- et 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

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.

```
let save_context { registers; processes } pid =
  Array.blit
    registers 0 processes.(pid).saved_context 0 num_registers

let restore_context { registers; processes } pid =
  Array.blit
    processes.(pid).saved_context 0 registers 0 num_registers

let choose_process { runqueues; processes } =
  let rec find_within rq =
    match rq with
    | [] -> None
    | rid::rq' ->
        if processes.(rid).state = Runnable then Some rid
        else find_within rq'
  in
  let rec find prio =
    match find_within runqueues.(prio) with
    | None -> find (prio - 1)
    | Some pid -> prio, pid
  in
  find max_priority
```

```
let schedule ({curr_pid=prev_pid} as s) =
  let next_prio, next_pid = choose_process s in
  save_context s prev_pid;
  restore_context s next_pid;
  s.curr_pid <- next_pid;
  s.curr_prio <- next_prio

let timer_tick { curr_pid; curr_prio; processes; runqueues } =
  let ({ slices_left = remaining } as p) = processes.(curr_pid) in
  p.slices_left <- remaining - 1;
  if remaining = 0
  then
   (p.slices_left <- max_time_slices;
    runqueues.(curr_prio) <-
  List.filter
     (fun pid -> pid <> curr_pid) runqueues.(curr_prio)
     @ [curr_pid]; true)
  else false
```

```
let transition ev ({curr_pid; registers; curr_prio} as s) =
let reschedule =
  match ev with
  | Timer -> timer_tick s
  | SysCall ->
      match decode s with
      | Send (ch, v) -> send s ch v
      | Recv chs -> recv s chs
      | Fork (prio, d0, d1, d2) ->
          ((if prio <= curr_prio
            then fork s prio d0 d1 d2
            else registers.(0) <- 0); false)
      | Wait -> wait s
      | Exit -> (exit s; true)
      | NewChannel -> (new_channel s; false)
      | Invalid -> (registers.(0) <- -1; false)
in
if reschedule then schedule s
```

```
let init =
  let new_procs i =
    if i = 0 then {
      parent_id = 0;
      state = Runnable;
      saved_context = Array.make num_registers 0;
      slices_left = max_time_slices }
    else if i = 1 then {
      parent_id = 1;
      state = Runnable;
      saved_context = Array.make num_registers 0;
      slices_left = max_time_slices }
    else {
      parent_id = 0;
      state = Free;
      saved_context = Array.make num_registers 0;
      slices_left = 0 }
  in
  let new_queues i =
    if i = max_priority then [1]
    else if i = 0 then [0]
    else []
```