Se souvenir

Le système de fichiers (exemple du disque dur)

Les caches (inodes et blocks)

Accès à un fichier : double indirection

Compteurs de références :

Les processus

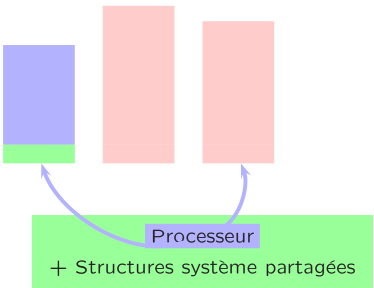
- Les processus (Clonage, Attente, Exécution)
- Shell
- Signaux
- Le temps

Qu'est-ce qu'un processus ?

Un processus
- c'est un programme en train de tourner...
- ... avec une structure allouée par le système pour son contrôle.

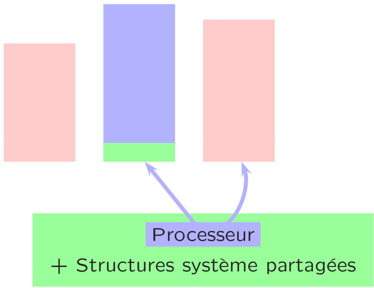
Qu'est-ce qu'un processus ?

Un processus
- c'est un programme en train de tourner...
- ... avec une structure allouée par le système pour son contrôle.



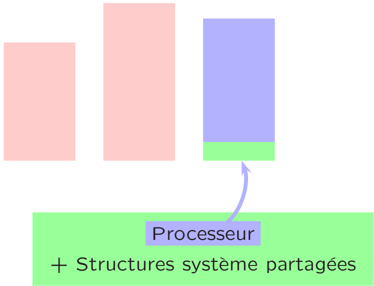
Qu'est-ce qu'un processus ?

Un processus
- c'est un programme en train de tourner...
- ... avec une structure allouée par le système pour son contrôle.



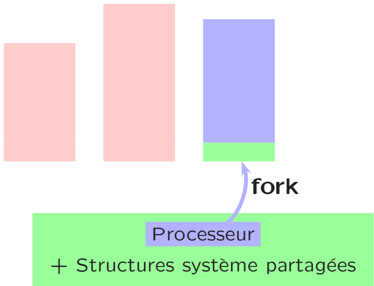
Qu'est-ce qu'un processus ?

Un processus
- c'est un programme en train de tourner...
- ... avec une structure allouée par le système pour son contrôle.



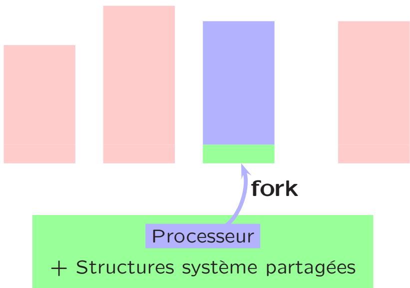
Qu'est-ce qu'un processus ?

Un processus
- c'est un programme en train de tourner...
- ... avec une structure allouée par le système pour son contrôle.



Qu'est-ce qu'un processus ?

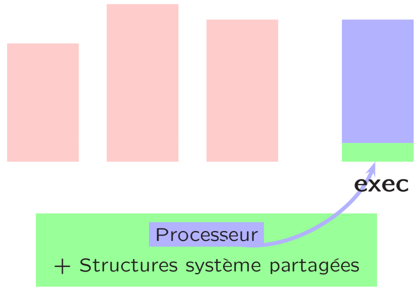
Un processus
- c'est un programme en train de tourner...
- ... avec une structure allouée par le système pour son contrôle.



Qu'est-ce qu'un processus ?

Un processus

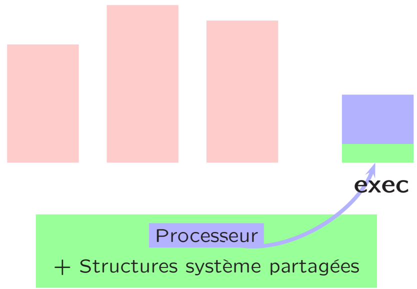
- c'est un programme en train de tourner...
- ... avec une structure allouée par le système pour son contrôle.



Qu'est-ce qu'un processus ?

Un processus

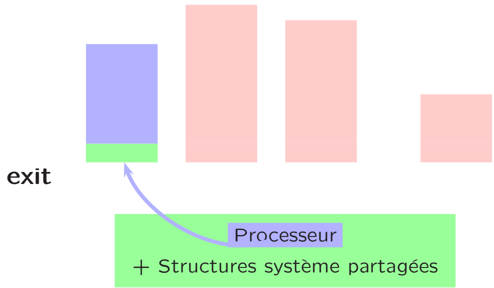
- c'est un programme en train de tourner...
- ... avec une structure allouée par le système pour son contrôle.



Qu'est-ce qu'un processus ?

Un processus

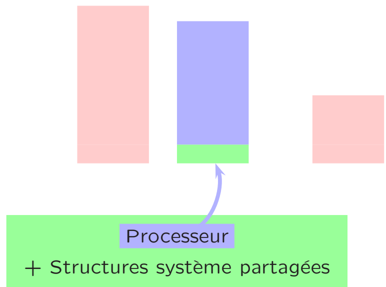
- c'est un programme en train de tourner...
- ... avec une structure allouée par le système pour son contrôle.



Qu'est-ce qu'un processus ?

Un processus

- c'est un programme en train de tourner...
- ... avec une structure allouée par le système pour son contrôle.



Lancement d'un processus

Le lancement d'un programme se fait en deux temps :

- Allocation d'un nouveau processus, par clonage (`fork`)
  - le processsus fils hérite de l'environnement de son père (environement d'exécution, variables-système)
  - en fait il en reçoit une copie : après le clonage, la modification de ses variables chez le processus père ou le processus fils n'affecte pas le processus fils et vice versa.
  - à l'exception de son identité `pid` et de celle de son père `ppid`.
- Remplacement du programme par un nouveau programme : chargement du fichier de code et lancement du programme.

Gestion des processus

Parallèlisme

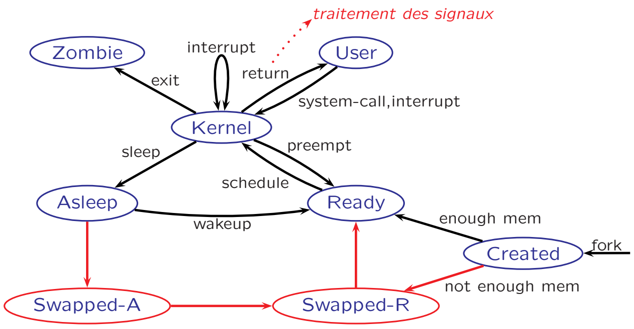
- Plusieurs processus tournent en parallèle sur la machine.
- Sur une machine mono-processeur, le système se charge d'entrelacer les processus en donnant un peu de temps à chacun (ordonnancement) : donne l'illusion du parallèlisme.
- L'entrelacement est imprévisible et non reproductible : l'exécution d'un programme peut être interompue de façon autoritaire par le système pour donner la main à un autre.

« Race condition » Deux entrelacements possibles de deux processus changent le sens du programme (de façon non voulue) : accès aux ressources, lecture/écriture d'une même donnée, *etc.*

Changer la priorité (ajoute une valeur de gentillesse, *i.e.* diminue la prorité. Valeurs négatives possibles avec privilège).

```
nice : int -> int
```

Un processus dans tous ses états...



Hiérarchie des processus

Un processus est identifié de façon univoque par son `pid` (entier)

```
getpid : unit -> int
```

Les processus sont organisés sous forme d'arbre avec le processus 1 pour racine. Un processus peut retrouver son père :

```
getppid : unit -> int
```

Lorsqu'un processus devient orphelin (parce que son père meurt), il est adopté par le processus initial 1 (la racine de l'arbre) et non par son grand-père.

Arbre des processus (example)

Information sur les processus

Commande unix `ps` (voir `man ps`) ou `pstree`

Création de processus

Le commande `fork` clone le programme en cours d'exécution en deux copies conformes, sauf leur identité (`pid`)

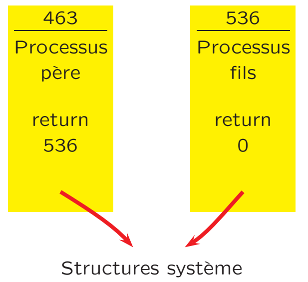
```
fork : unit -> int
```

- retourne 0 dans le processus fils et
- le numéro du processus du fils dans le processus père.

Le programme source contient le code du père et du fils, séparés par le retour de l'appel `fork` :

```
(* code exécuté avant le fork ---par le père *)
  begin match fork() with
  | 0 -> (* code du fils *)
  | k -> (* code du père dont le nouveau fils est k *)
  end;
(* code exécuté par les deux *)
```

Clonage



Indépendance après clonage

Le fils reçoit une copie de la mémoire du père

Cela peut avoir des effets pervers :

- Si besoin, il faut vider les tampons d'écriture (sinon, le fils et le père peuvent partager une partie non prévisible du tampon).
- Il peut y avoir un problème avec la lecture tamporisée dans `Pervasives.stdin` si le tampon n'est pas vide.

Père et fils partagent des structures systèmes

- le père est averti de la terminaison du fils (par signal ou exit).
- Les descripteurs de fichiers sont dupliqués, mais pointent vers une même structure système : `lseek` fait par l'un est vu par l'autre. (voir prochain cours)
- Cas particulier des tuyaux. (voir prochain cours)

Exemple illustratif


```
let pid = getpid() in let ppid = getppid() in
begin match fork() with
| 0 ->
    let son_pid = getpid() in let son_ppid = getppid() in
    printf "Je m'appelle %d, " son_pid;
    if son_ppid = pid then
      printf "fils de %d né %d.\n" pid ppid
    else if son_ppid = 1 then printf "je suis orphelin.\n"
    else printf "je suis perdu.\n"
| k ->
    printf "Je m'appelle %d né %d, " pid ppid;
    printf "je suis le père de %d.\n" k;
end;
printf "Au revoir! [from %d]\n" (getpid());;            (* run *)
```

Mise en garde

Un processsus, un ulilisateur, mais aussi le système ont un nombre limité de processus...

Si un programme alloue des processus en boucle, il risque de consommer toutes les resources (en processus) et de ne même plus pouvoir allouer de processus pour tuer le programme fautif.

Pour se prémunir contre ce genre d'erreur en TD, instrumenter la commande `fork` pour qu'elle intéragisse avec l'utilisateur :

```
let interactive_fork() =
  Printf.printf "Fork processus %d ?\n%!" (getpid());
  input_char();
  fork();;
```

Taper `^C` au lieu d'un caractère pour interrompre le programme.

Terminaison des processus

À sa mort un processus transmet des informations à son père.

Il devient un processus zombi en attendant que son père lise ces informations et lui permette ainsi de disparaître à jamais.

Un processus zombi encombre la mémoire, car ses resources ne peuvent pas être désallouées.

Il faut donc libérer les processus zombis dès que possible.

Le processus initial qui récupère les processus orphelins contient une boucle qui libère ses fils adoptifs dès qu'ils meurent.

Si l'on n'a pas besoin d'intéragir avec son fils, on peut le rendre orphelin immédiatement (par la technique du double fork), afin de ne avoir à attendre sa mort pour le libérer.

Attente de la terminaison d'un fils

La mort d'un fils quelconque

```
wait : unit -> int * process_status
```

Plus d'options

`WNOHANG`	Retourner 0 au lieu de bloquer.
`WUNTRACED`	Indiquer aussi les fils suspendus.

```
waitpid : wait_flag list -> int -> int * process_status
```

`pid`	numéro du fils attendu.
`-1`	attendre n'importe quel fils.

Process status

`WEXITED` (*ret*)	Retour normal par exit avec la valeur *ret*
`WSIGNALED` (*sig*)	Tué par le signal *sig*
`WSTOPPED` (*sig*)	Arrêté par le signal *sig*

Exemple


```
let pid = getpid() in let ppid = getppid() in
begin match fork() with
| 0 -> ...
| k ->
    printf "Je m'appelle %d né %d, " pid ppid;
    printf "je suis le père de %d.\n" k;
    match wait() with
    | p, WEXITED 0 when p = k ->
      printf "Mon fils %d repose en paix [%d]\n" p pid
    | p, (WSIGNALED _ | WSTOPPED _ when) p = k ->
      printf "Mon fils %d died awfully [%d]\n" p pid
    | p, _ ->   (* Ici, p <> k *)
      printf "%d est-il mon fils? [%d]\n" p pid
end;
printf "Au revoir! [from %d]\n" (getpid());;            (* Run *)
```

Recherche parallèle en table

On cherche un élément du tableau t qui satisfait le prédicat p

```
let search p t = ...
```

Version parallèle

```
let fork_search cond v =
  let n = Array.length v in
  match fork() with
    0 ->
      let found = search cond (Array.sub v (n/2) (n-n/2)) in
      exit (if found then 0 else 1)
  | _ ->
      let found = search cond (Array.sub v 0 (n/2)) in
      match wait() with
        (pid, WEXITED retcode) -> found or (retcode = 0)
      | (pid, _)               -> failwith "fork_search";;
```

Le « double fork »

Problème

- Lorsque le père n'est pas intéressé par la terminaison de son fils, il serait plus facile de «l'abandonner» immédiatement.
- Comment faire pour que son fils ne reste pas un zombi ?
- Formellement, ce n'est pas possible : la redirection vers le processus init ne se fait qu'à la mort du père.

Solution

- Faire faire le travail par un petit fils. Ainsi, le fils ne vit que le temps très court de créer un petit fils, puis meurt.
- Le petits fils n'a plus de père et est rataché au processus init.
- Le père peut libérer immédiatement son fils (attente courte).

Le « double fork »

Mise en œuvre

```
match fork() with
| 0 -> if fork() <> 0 then exit 0;
       (* code du fils, en fait executé par le petit-fils *)
| _ -> wait();
       (* code du père, plus à s'occuper du petit fils *)
```

Lancement d'un programme

- Un programme peut seulement céder sa place à un autre.
- Si besoin, il se sera cloné auparavant et le fils cédera sa place.

La commande de base :

```
execv : string -> string array -> unit
```

commande arguments

Le lancement d'un programme remplace le programme en cours par le nouveau programme qui hérite de ses variables système :

- même numéro de processus, même père, *etc.*
- mêmes variables d'environnement.
- mêmes descripteurs ouverts
  (sauf pour les descripteurs avec le drapeau `close_on_exec`)
- variantes : `execv`, `execvp`, `execvpe`.

Note

La commande `exec` ne retourne pas !

La seconde ligne ne sera jamais exécutée :

```
execvp cmd argv;
print_endline "after";
```

La commande `exec` peut échouer

Pour repporter les erreurs, on pourra écrire :

```
handle_unix_error (execvp com) argv
```

En cas d'échec, seul l'ancien processus continue à tourner (si l'on a bien rattrapé et traité l'erreur) : le nouveau n'est pas lancé.

Les fichiers interprétés

Particularité Ce sont des fichiers textes commançant par `#!` :

- le premier mot est le chemin absolu de l'interprête
- les autres mots sont les arguments à lui passer

Exemple Si le fichier `/home/remy/cours/system/3/script.ml` commence par :

```
#!/usr/bin/ocamlrun /usr/bin/ocaml
```

alors

```
Unix.execv "/home/remy/system/script.ml" [| "foo"; "bar" |]
```

exécute le programme `/usr/bin/ocamlrun` avec les arguments

```
 [| "/usr/bin/ocamlrun"; "/usr/bin/ocaml";
    "/home/remy/cours/script.ml"; "foo"; "bar" |]
```

(Ne fonctionne pas de façon récursive...)

Exemple

Ajouter un argument à une commande

```
let grep() =
  let args = Array.concat [
    [| Sys.argv.(0); |]; [|"-i"|];
    (Array.sub Sys.argv 1 (Array.length Sys.argv -1));
  ] in Unix.execvp args.(0) args
```

Héritage

Au lancement d'un programme, le nouveau programme s'exécute dans le même processus que l'ancien. Il conserve son `pid` son `ppid` (et donc se comporte comme l'ancien programme vis à vis de `wait` pour son parent.)

Plus généralement il hérite par défaut des variables-système du programme précédent `cwd`, `root` déjà vus, masque des signaux (voir ci-après).

Il garde également les mêmes descripteurs de fichiers, sauf si la fermeture automatique de ceux-ci a été demandée explicitement (par la fonction `set_close_on_exec`).

Shell

Langage de commandes

C'est un langage de commande pour intéragir avec le système.

```
mkdir foo
cd foo
echo 'print_string "hello";;' > foo.ml
ocamlc -c foo.byte foo.ml && exec ./foo
```

Langage de script

```
#!/bin/bash
echo BEGIN;
cat "$@";
echo END
```

OCaml comme langage de script

```
#!/usr/bin/ocamlrun /usr/bin/ocaml
#load "unix.cma"
open Unix
let echo s = print_string "BEGIN"; print_newline();;
let copy_chan c = ...;;
let copy_file = ...;;
let cat args =
  match args with
    [] -> copy_chan std_in
  | _ -> List.iter copy_file args;;

echo "BEGIN";
cat (List.tl (Array.to_list Sys.argv));
echo "END";;
```

Scripts compilés

```
open Unix
let echo s = print_string "BEGIN"; print_newline();;
let copy_chan c = ...;;
let copy_file = ...;;
let cat args =
  match args with
    [] -> copy_chan std_in
  | _ -> List.iter copy_file args;;

echo "BEGIN";
cat (List.tl (Array.to_list Sys.argv));
echo "END";;
```

Avantage rapidité + détection statique des erreurs de type.

Mini-shell écrit en OCaml

C'est un programme OCaml qui lit des commandes, les analyse et les exécute.

- Chaque ligne est une commande suivie de ses arguments.
- Les espaces séparent les arguments (sauf entre «"»)
- Certaines commandes, reconnues (pwd, cd, *etc.*) sont exécutées directement par le shell.
- Les autres sont exécutées par un fork+exec. Le shell attend le retour de la commande.
- Mise en tache de fond : `&` en fin de ligne.
- Reconnaître les opérateurss «`;`», «`&&`» et «`||`» pour l'évaluation séquentielle avec code de retour de la dernière commande ou en les combinant par «et» ou «ou» logique.

Mini-cash

Cash est une librairie de fonctions pour l'écriture de scripts.

Ce n'est pas un shell : le script est écrit en OCaml à l'aide de cette librairie.

Intérêt

- On profite de OCaml comme langage de programmation.
- Système ouvert : on peut ajouter d'autres librairies.
- − Pas de support syntaxique.

Exercice Écrire une petite bibliothèque

- permettant la composition des commandes.
- l'exécution de commande en tâche de fond, leur gestion, *etc.*
- se protéger contre les signaux pendant l'exécution des commandes.

Les signaux

Les signaux

Signaux

Événements externes qui changent le déroulement d'un programme, de manière asynchrone.

Comportement à la réception

Les signaux sont asynchrones : ils sont reçus par le programme à n'importe quel moment4,2) de l'exécution.

En fait, en OCaml,

1) leur traitement immédiat (à un instant quelquonque) consiste à les mettre de coté,
2) leur code n'est effectivement exécuté qu'au moment d'une allocation, d'un appel récursif ou d'une boucle while.

La réception d'un signal provoque, selon le signal et les réglages :

- La terminaison de l'exécution (avec ou sans «core»).
- La suspension de l'exécution (le processus père est prévenu.)
- Rien : le signal est simplement ignoré.
- L'exécution d'une fonction définie par l'utilisateur.

Livraison

Garantie

Si un processus p délivre un signal s à un procesus q, et que le signal s n'est pas bloqué par q alors on est sûr que q recevra au moins une occurrence du signal s.

Perte de signaux

Un signal émis plusieurs fois peut n'être délivré qu'une seule fois, mais il sera toujours délivré au moins une fois.

Les principaux signaux

Nom	Signification	Comportement
`sighup`	Hang-up (fin de connexion)	T(erminaison)
`sigint`	Interruption (`ctrl-C`)	T
`sigquit`	Interruption forte (`ctrl-\`)	T + Core
`sigfpe`	Erreur arithmétique	T + Core
`sigkill`	Interruption immédiate et absolue	T Toujours
`sigsegv`	Violation des protections mémoire	T + Core
`sigpipe`	Écriture sur un tuyau sans lecteurs	T
`sigalrm`	Interruption d'horloge	Ignoré
`sigtstp`	Arrêt temporaire (`ctrl-Z`)	Suspension
`sigcont`	Redémarrage d'un fils arrêté	Ignoré
`sigchld`	Un des fils est mort ou arrêté	Ignoré

Les autres

Faire `man 7 signal` pour avoir la liste complète sur votre machine.

Modularité

- Sauf utilisation très particulière, se limiter aux signaux POSIX (ceux décrits dans le module `Sys` à l'exception de `sigprof`).
- Les signaux ont des noms conventionnels.
- Les numéros changent d'une implémentation à une autre.
- Il faut absoluement utiliser les noms pour être portable. (définis dans le module Sys).

Les signaux `sigusr1` et `sigusr2`

Ils n'ont pas d'usage conventionnel et sont à la disposition de l'utilisateur.

Changer le comportement d'un signal

Le type `signal_behavior`

`Signal_default`		redonne le comportement par défaut du signal
`Signal_ignore`		ignore the signal
`Signal_handle`	`int -> unit`	applique la fonction en argument au numéro du signal.

Modification :

La valeur précédente est retournée (Voir aussi `set_signal`)

```
signal : int -> signal_behavior -> signal_behavior
```

Pour lire la valeur d'un signal s (approximation) :

```
let get_signal_behavior s =
  let b = signal s Signal_default in ignore (signal s b); b
```

Code de traitement d'un signal

Asynchronisme

Lorsque l'utilisateur change le comportement d'un signal, l'utilisateur introduit du code de traitement qui va être exécuté à un moment inconnu, en principe entre deux instructions machines quelconques. C'est ce qui se passe en C.

Compétition

A priori, le code de traitement lit et écrit dans la mémoire (sinon, son comportement est non observable).

Si le code principal lit et écrit également dans la même partie de la mémoire, alors il y a risque d'incohérence par entrelacement :

Code principal	$x \rightarrow k$			$x \leftarrow k+1$
Code du signal		$x \rightarrow k$	$x \leftarrow k+1$	

Points de contrôle en OCaml

À la réception du signal

Le code passé en argument à `Signal_handle` n'est pas exécuté à l'arrivée d'un signal, mais mis dans une table et remplacé par du code qui enregistre simplement la réception.

Traitement aux points de contrôles

Le runtime vérifie, lorsqu'il effectue une allocation ou une boucle, si des événements sont arrivés. C'est à ce moment là seulement qu'il exécute le code de traitement correspondant des signaux arrivés.

Garantie

Une expression qui n'alloue pas et ne boucle pas (par exemple écrit seulement un entier ou un booléen) n'est jamais entrelacée avec le code de traitement d'un signal.

Cas général

Protection

Il faut se protéger pendant la section critique dans le programme principal, par exemple en bloquant les signaux pendant la modification de la mémoire

Exemple

```
let sig_handler p = v := p :: !v;;
let main () = ...
  let mask = sigprocmask SIG_BLOCK [ s ]
  let _ = match !v with h :: t -> v := t; ... in
  sigprocmask SIG_SETMASK mask;;
```

Masquer les signaux

On peut bloquer un signal, ce qui n'est pas la même chose que l'ignorer : le signal reste en attente pour être envoyé plus tard.

```
sigprocmask : sigprocmask_command -> int list -> int list
```

Le premier argument décrit le sens à donner à la liste des signaux passés en second argument :

`SIG_SETMASK`	bloquer exactement ces signaux.
`SIG_BLOCK`	ajouter ces signaux aux signaux bloqués.
`SIG_UNBLOCK`	retirer ces signaux des signaux bloqués.

La valeur retournée est la liste des signaux à bloquer (avant l'appel)

```
sigpending : unit -> int list
```

Retourne les signaux bloqués en attente (sans les débloquer)

Protection

Le changement temporaire du masque des signaux doit être protégé : son rétablissement au retour doit être effectué pour le retour normal ou anormal, par exemple en utilisant `try_finalize`.

(Voir par exemple la fonction system).

Mise en attente

Arrête le processus jusqu'à l'arrivée d'un signal (parmi la liste).

```
sigsuspend : int list -> unit
```

Cas particulier où les signaux sont tous les signaux non bloqués et non ignorés.

```
pause : unit -> unit
```

Exercices

Définir la fonction `Sys.catch_break` de type `bool -> unit`.

```
exception Break;;
let catch_break on =
  if on then
    set_signal sigint (Signal_handle(fun _ -> raise Break))
  else
    set_signal sigint Signal_default;;
```

Définir la fonction `pause`.

```
  let sigs = sigprocmask SIG_BLOCK [] in sigsuspend sigs
```

Envoyer un signal à un programme


```
kill : int -> int -> unit
```

`kill` *sig pid* envoit le signal *sig* au processus *pid*.

Synchronisation

Si le signal *sig* n'est pas bloqué par *pid*, `kill` retourne seulement après que le process *pid* ait été notifié du signal *sig*, *i.e.* lorsque le processus *pid* reprendra la main, la première chose qu'il fera sera de traiter un signal (*sig* ou un autre signal non bloqué).

Sur une machine multiprocesseurs, si *pid* peut avoir la main sur un autre processeur, il ne verra pas le signal tant que le système ne prendra pas la main sur ce processeur (i.e. il peut s'écouler quelques miliseconds).

Signaux et appels systèmes

Les signaux peuvent interrompre les appels systèmes, dit *lents*, qui peuvent ne pas retourner immédiatement : `read`, `write`, `wait`, `waitpid`, `system`, ...

Dans ce cas, l'appel système retourne l'erreur `EINTR` «`Interrupted system call`»

Il est important de savoir si un appel système est *lent*/interruptible our non.

Il est parfois possible (selon le système) de demander au système de relancer automatiquement les appels systèmes lorsqu'ils ont été interrompus par des signaux.

Protection

- Masquer les exceptions pendant un appel.
- Relancer un appel interrompu.

L'appel `Unix.system`

Cette commande permet de faire exécuter séquentiellement une commande complexe par le shell `/bin/sh`.

```
system : string -> process_status
```

La valeure retournée est celle retournée par le shell.

Ce n'est pas un appel système primitif : la librairie `Unix` la définit de la façon suivante (Attention, voir ci-après) :

```
let system cmd =
  match fork() with
     0 -> begin try
            execv "/bin/sh" [| "/bin/sh"; "-c"; cmd |]
          with _ -> exit 127 end
  | id -> snd(waitpid [] id)
```

La version de la librairie C

La version de la librairie Unix n'est pas conforme à la norme POSIX.1 :

1. L'appel système `waitpid` peut être interrompu par un signal.
2. Pendant l'appel système, le programme principal doit
   - ignorer les signaux `sigint` et `sigquit`.
   - bloquer le signal `sigchld`.

Donner une implémentation conforme à la norme POSIX.1

Solution

Remarque : dans certains cas, on peut vouloir ne pas implémenter "2." pour pouvoir interrompre un programme appelé qui boucle. Par contre, il n'y a aucune raison pour ne pas corriger "1.".

Les temps anciens

Compté en secondes

- La date
  ```
  time : unit -> float
  ```
  Retourne le nombre de secondes écoulées depuis `00:00:00 GMT, Jan. 1, 1970`.
- Pour arrêter l'éxécution pendant un certain temps :
  ```
  sleep : int -> unit
  ```
- Pour recevoir le signal `sigalrm` après un certain délai.
  ```
  alarm : int -> int
  ```
  retourne le délai qui reste avant la dernière alarme programmée et 0 si aucune alarme n'était déjà programmée.

Sleep (exemple)


```
let sleep secs =
  let old_alarm =
    signal sigalrm (Signal_handle (fun s -> ())) in
  let old_mask = sigprocmask SIG_BLOCK [ sigalrm ] in
  let _ = alarm secs in
  let suspmask =
    List.filter (fun x -> x <> sigalrm) old_mask in
  sigsuspend suspmask;
  (* on est réveillé, peut-être pas par l'alarme,
     que l'on remet à zéro *)
  let unslept = alarm 0 in
  ignore (signal sigalrm old_alarm);
  ignore (sigprocmask SIG_SETMASK old_mask);;
```

Les temps modernes

Compté en micro-secondes

- La date :

  ```
  gettimeofday : unit -> float
  ```

  Comme `time` mais avec une meilleure résolution (en micro-secondes).
- Conversions :
  La structure `tm` représente le temps selon le calendier (année, mois, etc.). Voir les fonctions de conversion : `gmtime`, `localtime`, `mktime`, etc.
- Temps de calcul, donné par la fonction `times`.

Les sabliers

Chaque processus à trois sortes de sabliers décrits :

ITIMER_REAL	temps réel	sigalrm
ITIMER_VIRTUAL	temps utilisateur	sigvtalrm
ITIMER_PROF	utilisateur et système (debug)	sigprof

L'état d'un sablier est décrit par le type `interval_timer_status` :

it_interval : float	Période
it_value : float	Valeure courante

Sémantique :

- Un sablier est inactif lorsque ses deux champs sont nuls.
- Sinon, le champ `it_value` décroît selon le type du sablier.
- Lorsqu'il devient nul le signal correspondant est émis et le sablier est remis à la valeur du champ `it_inverval`.

Les sabliers

Consultation

```
getitimer : interval_timer -> interval_timer_status
```

Modification

```
setitimer : interval_timer -> interval_timer_status ->
                                        interval_timer_status
```

La valeur retournée est l'ancienne valeur du sablier au moment de la modification.

Retenir

Les processus

- Exécution en parellèle par entrelacement (géré par le système).

Vie : Fork + Exec

- Clonage d'un processus existant, avec partage total.
- Remplacement d'un processus par un autre.

Mort : waitpid

- Récupérer leur état (zombies)

Les signaux

- Leur exécution est asynchrone, d'où difficulté.
- Ils interrompent les appels systèmes longs (les relancer).
- Doivent être bloqués ou ignorés pendant les phases critiques.

Éviter les signaux si possible

Source de difficultés

- Ils portent peu d'information.
- Difficulté inhérente dues à l'asynchronisme.
- Superposition possible des signaux proches dans le temps.

Problème de portabilité

- Le traitement des signaux dépend de la variante d'Unix.
- Les signaux eux-mêmes peuvent en dépendre.

Une des parties les moins bien conçues d'Unix

Alternatives

- select parfois : (ex. usleep)
- coprocessus : vraie concurrence (voir plus loin)

```
 ...
let system_call () =
  match fork() with
  | 0 ->
      reset();
      begin try
        Unix.execv "/bin/sh" [| "/bin/sh"; "-c"; arg |]
      with _ -> exit 127
      end
  | k ->
      let rec wait() =
        try snd (waitpid [] k)
        with Unix_error (EINTR, _, _) -> wait() in
      wait() in
 ...
```

```
let system arg =
  let old_mask = sigprocmask SIG_BLOCK [ sigchld ] in
  let old_int = signal sigint Signal_ignore in
  let old_quit = signal sigquit Signal_ignore in
  let reset() =
    ignore (signal sigint old_int);
    ignore (signal sigquit old_quit);
    ignore (sigprocmask SIG_SETMASK old_mask) in
  let system_call () = ... in
  try_finalize system_call() reset()
```