Révision ...

1. Quelle différences y a-t-il entre processus et coprocessus ?
2. Comment se met-on en attente sur une condition ?
3. Comment signale-t-on sur une condition ?
4. Donner des exemples typiques d'utilisation de coprocessus ?

Structure des processus et leur ordonnancement

Généralités

- Mode utilisateur et mode privilégié, Interruptions
- Mémoire virtuelle

Structure des processus

- Organisation mémoire
- Appel système, (Interface OCaml C)
- Changement de contexte

Ordonancement

- Algorithme de round Robin
- Machines multi-processeurs

Le système vu de l'intérieur

Quelles sont les contraintes sur le matériel ?

- Sécurité : *e.g.* instructions privilégiées (mode superviseur)
- Efficacité : *e.g.* MMU (unité mémoire)
- Simplicité : *e.g.* instruction test-and-set

Quelles sont les contraintes sur le logiciel ?

- Efficacité : algorithmes rapides
- Simplicité (éviter les erreurs) : algorithmes simples, souvent allocation statique des tables, recherches linéaires.

Quel compromis faire entre les deux contraintes ?

(Certains choix du passé parfois remis en question)

Où est la frontière entre l'intérieur et l'extérieur ?

- Elle peut être déplacée, mais
- Un minimum d'opérations doivent rester à l'intérieur.

Les très gros-noyaux (*e.g.* Multics)

On met le maximum dans le noyau :

- plus simple (à écrire)
- plus efficace (communication légère entre les composants)
- plus fragile (la sûreté dépend du maillon le plus faible)
- noyau *trop* gros : *e.g.* faut-il charger tous les drivers ?

La frontière intérieur/extérieur

Les très gros-noyaux (*e.g.* Multics)

Les micro-noyaux (*e.g.* système Mach.)

On met le miminum nécessaire pour assurer :

- la sûreté (bon fonctionnement)
- la sécurité (pas d'intrusion) [préliminaire à la sûreté].

Propriétés :

- Gros avantage : plus robustes
  - la base de confiance est plus petite.
  - si un service hors du système casse, le système reste intègre.
- Gros inconvénient : très lent. Beaucoup trop de communication entre les services et le système

Échec de Mach ?

Les très gros-noyaux (*e.g.* Multics)

Les micro-noyaux (*e.g.* système Mach.)

Propriétés :

- Gros avantage : plus robustes
  - la base de confiance est plus petite.
  - si un service hors du système casse, le système reste intègre.
- Gros inconvénient : très lent. Beaucoup trop de communication entre les services et le système

  Échec de Mach ?

  Non, car les machines sont de plus en plus rapides, les périphériques de plus en plus autonomes (gros buffeurs).
  Le temps gagné est plus passé à calculer qu'en appels système.

  Ressuscité par Mac OS !

  = Architecture Mach + BSD au dessus

La frontière intérieur/extérieur

Les très gros-noyaux (*e.g.* Multics)

Les micro-noyaux (*e.g.* système Mach.)

Les noyaux modulaires (*e.g.* linux, tendance actuelle)

Un peu la combinaison des deux, mais pas tout à fait :

- Système monolithique dans sa conception.
- Mais des services du noyau sont fournis comme des modules, chargeables (et déchargeables) à la demande.
- On ne charge que les modules nécessaires (tendance indispensable avec l'explosion du matériel et des drivers : 90% du code d'un système concerne les drivers)
- Fragilité des gros-noyaux pas bien résolue : tentative de circonscription des erreurs à un module sans briser l'intégrité du système.

Nécessité de deux modes de fonctionnement

- Le système doit assurer une certaine sécurité, et pour cela interdire certaines opérations aux programmes utilisateurs.
  - L'accès direct aux périphériques (qui pourrait bloquer la machine)
  - L'accès aux autres programmes en train de tourner
  - L'accès à certains fichiers.
  - De monopoliser la machine (par exemple en bloquant les interruptions).
  - *etc.*

Nécessité de deux modes de fonctionnement

- Le système doit assurer une certaine sécurité, et pour cela interdire certaines opérations aux programmes utilisateurs.
- Le système doit pouvoir effectuer toutes les tâches interdites à l'utilisateur, dites privilégiées.

Il faut donc distinguer :

- un mode utilisateur, et
- un mode privilégié pour effectuer les tâches privilégiées.

Ils doivent être gérés par le processeur

- Le programme utilisateur est une suite d'instructions (octets) L'utilisateur connait le jeu d'instructions (ou il peut l'inventer). Il faut donc l'empêcher d'exécuter les instructions privilégiées.
- Le processeur a deux modes d'exécution, le mode utilisateur et le mode superviseur.

Changement de mode

Contraintes

- Le passage doit pouvoir être fait à l'initiative de l'utlisateur lors d'un appel système (tôt ou tard).
- L'exécution ne peut pas continuer dans le code utilisateur : il doit y avoir une rupture de séquence.

Solution

- Le passage se fait par une interruption logicielle (trap)
- L'exécution continue en mode système à une adresse définie au démarrage de la machine où *l'utilisateur ne doit pas avoir accès en écriture*.
- Il faut donc également une zône mémoire privilégiée, où l'écriture ne peut se faire qu'en mode privilégié.

Changement de mode

Zône Utilisateur	
961	before
962	$r0 \leftarrow 2$
963	trap
964	after

Zône Système

	0	1	2	...	
trap	13	21	30	...	82

30	instr
	...
	rti

Changement de mode

Zône Utilisateur	
961	before
962	$r0 \leftarrow 2$
963	trap ◀
964	after

Zône Système

	0	1	2	...	
trap	13	21	30	...	82

30 instr

...

rti

Changement de mode

Zône Utilisateur	Zône Système

961 before

962 $r0 \leftarrow 2$

963 trap

964 after

	0	1	2	...	
trap	13	21	30	...	82

30 instr

...

rti

Changement de mode

Zône Utilisateur		Zône Système	
961	before		
962	$r0 \leftarrow 2$		
963	trap		
964	after		

	0	1	2	...	
trap	13	21	30	...	82

30

instr
...
rti

Changement de mode

Zône Utilisateur		Zône Système

961	before
962	$r0 \leftarrow 2$
963	trap
964	after

	0	1	2	...	
trap	13	21	30	...	82

30	instr
	...
	rti

Changement de mode

Zône Utilisateur		Zône Système

961	before
962	$r0 \leftarrow 2$
963	trap
964	after

	0	1	2	. . .	
trap	13	21	30	. . .	82

30	instr
	. . .
	rti

Changement de mode

Zône Utilisateur	
961	before
962	$r0 \leftarrow 2$
963	trap
964	after

Zône Système

	0	1	2	...	
trap	13	21	30	...	82

30	instr
	...
	rti

Différence avec jump :

- L'état des registres est sauvé automatiquement.
- L'utilisateur ne choisit pas à quelle adresse se fait le saut (seulement le numéro du saut).
- L'adresse est dans un vecteur d'interruptions (qui ne peut être modifié qu'en mode privilégié)
- Le processeur passe en mode privilégié.

Changement de mode

Zône Utilisateur		Zône Système					
			0	1	2	...	
961	before	trap	13	21	30	...	82
962	$r0 \leftarrow 2$						
963	trap	30	instr				
964	after		...				
			rti				

Instructions privilégiés

- Un drapeau du processeur indique le mode d'exécution.
- Chaque instruction privilégiée teste ce drapeau et provoque une erreur si le processeur est en mode utilisateur.

Protection mémoire

Au minimum

En mode utilisateur, certaines adresses doivent être innacessibles en écriture. En particulier les adresses de traitement des appels système.

Insuffisant

- Permet de protéger le système de l'utilisateur
- Mais pas les utilisateurs entre eux.

Besoin de matériel spécialisé

- pour une vraie protection mémoire entre utilisateurs.
- pour une gestion efficace de la mémoire (*c.f.* cours suivant).

(c.f. prochain cours)

- Les adresses sont organisées par pages :

  adresse = (numéro de page, décalage dans la page)
- Les pages/adresses sont virtuelles,

  traduites au vol par le **matériel**

  en pages/adresses réelles.
- Le système gère la correspondance dans des tables de pages.

Types d'interruptions

- Interruptions matérielles (périphériques : terminal, disk, *etc.*).
- Interruptions logicielles (passage en mode privilégié).

Les interruptions sont asynchrones (c.f. signaux)

Elles arrivent n'importe quand (entre deux instructions)

- En mode utilisateur : OK.
- En mode système : IDEM, mais elles peuvent être masquées.
  - Les interruptions sont hiérarchisées.
  - Le niveau de privilège du processeur également.
  - seules les interruptions de plus haute priorité que le mode de fonctionnement sont prises en compte.
  - Le traitement d'une interruption bloque les interruptions de niveau inférieur ou identique.
    Essentiel pour préserver les structures de données système.

Processus et mode système en Unix

Caractéristiques

- Il n'y a qu'une seule copie partagée du code système.
- Mais il n'y a pas de processus `système`.
- Le système «emprunte» le processus en cours d'exécution.
- Chaque processus a une pile réservée pour les calculs système.

Activation «automatique» lors d'un appel système

- Sauvegarde de l'état du processus, passage en mode système.
- Le calcul est effectué pour le processus en cours d'éxécution.
- Si l'appel système est bloqué (absence d'une ressource), le processus peut s'endormir et donner la main à un autre (*c.f.* changement de contexte).
- L'appel système reprendra dans le même processus lorsque la ressource sera libérée.
- Puis retour en mode utilisateur.

Processus et mode système en Unix

Caractéristiques

- Il n'y a qu'une seule copie partagée du code système.
- Mais il n'y a pas de processus `système`.
- Le système «emprunte» le processus en cours d'exécution.
- Chaque processus a une pile réservée pour les calculs système.

Activation «automatique» lors d'un appel système

Utilisé pour traiter les interruptions

- L'interruption ne sert pas un processus particulier, mais effectue la mise à jour de structures systèmes partagées.
- Le processus en cours d'exécution «prête» sa pile système pour le traitement de l'interruption
  (après avoir sauvé son contexte d'exécution).
- L'appel traite complétement le code de l'interruption, sans bloquer, et retourne dans le contexte précédent.

Mémoire globale

- Les structures du système sont partagées entre tous les processus : caches de blocs, caches d'inodes, tables des fichiers, table des processus, *etc.*

Mémoire globale

Pile locale à chaque processus

- La pile d'exécution système n'est visible que dans le processus où elle réside : on n'a effectivement besoin de la voir que quand on revient au processus où elle réside.
- Elle contient un empilement de contextes d'exécution
  (le contexte d'exécution est sauvé à chaque interruption)
  bornée par le nombre de niveaux d'interruptions.

Mémoire système

Mémoire globale

Pile locale à chaque processus

u area

- Chaque processus a une zone système privée appelée `u_area` qui contient des informations visibles par le système mais seulement lorsqu'il s'exécute dans le processus courrant.
- Cette zône permet d'identifier le processus en cours d'exécution et d'accéder directement à ses paramètres.
- À chaque appel système, les paramètres sont recopiés de la pile utilisateur dans la `u_area`, et inversement au retour.
- Les droits sont vérifiés lorsque le système lit/écrit dans la mémoire utilisateur. La `u_area` sert donc aussi de tampon entre la zône utilisateur et la zône système.

Organisation mémoire d'un processus

La mémoire utilisateur

- Le code du programme en train d'être exécuté.
- Un espace pour les données
- La pile utilisateur (au fond `argc` et `argv` passés à `main`).

La mémoire système

- Partie statique
  - La `u-area` (zône système dépendant du processus)
  - L'entrée dans la table des processus (état du processus)
  - La mémoire du processus (table des pages)
- Partie dynamique
  - La pile système utilisée pendant les appels systèmes et les interruptions.
  - Elle est vide lorsque le processus est en mode utilisateur.

Contexte d'éxécution

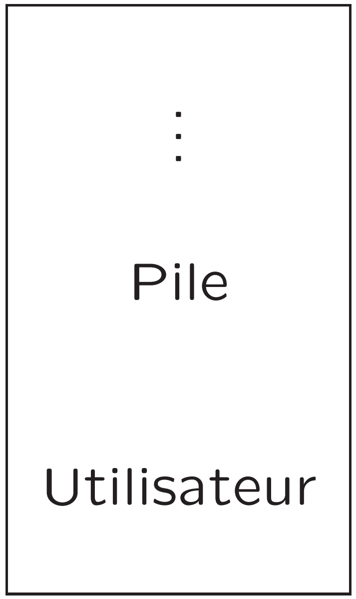
Sauvegarde

- Contient une sauvegarde des registres spéciaux, des registres système, du pointeur de pile.
- Permet un retour anticipé d'une interruption ou d'un appel système, en abandonnant l'exécution en cours.
- Analogue à `try/with` en OCaml, ou à `setjmp/longjmp` en C.
- Les contextes d'évaluations sont chaînés.

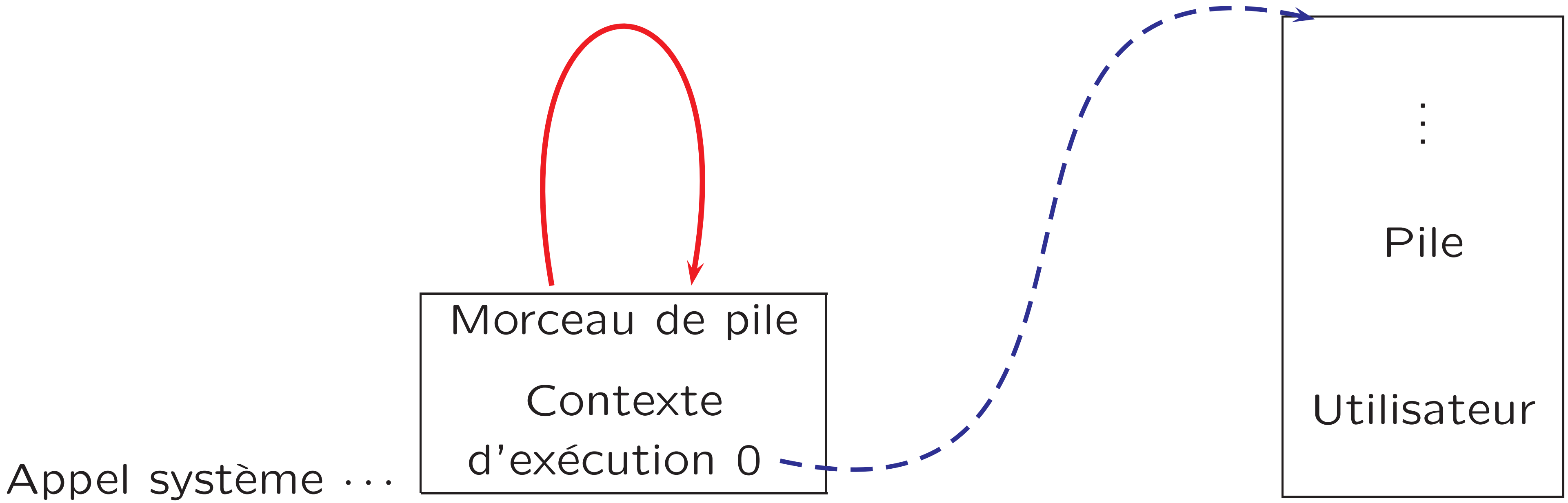
Quand

- À chaque appel système.
- À chaque interruption.

Piles système utilisateur

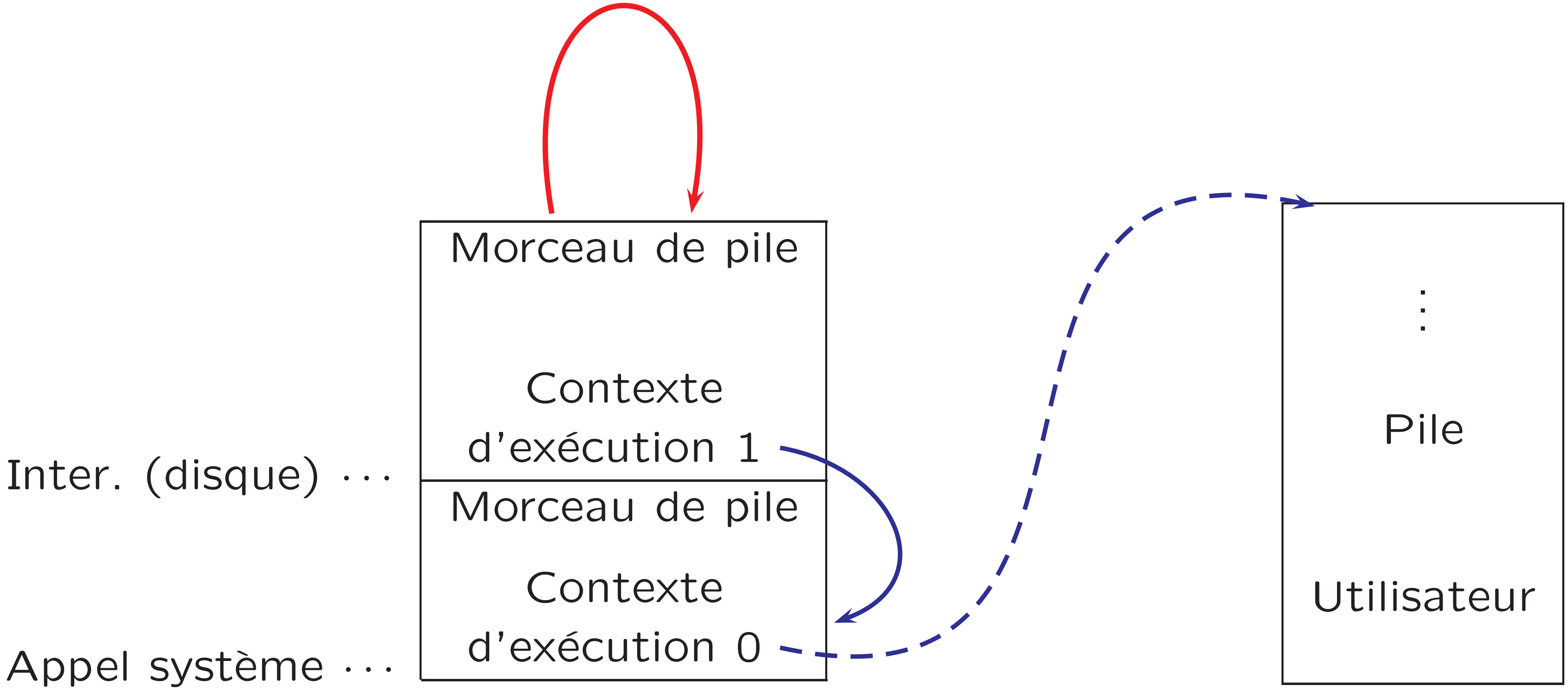


Piles système utilisateur

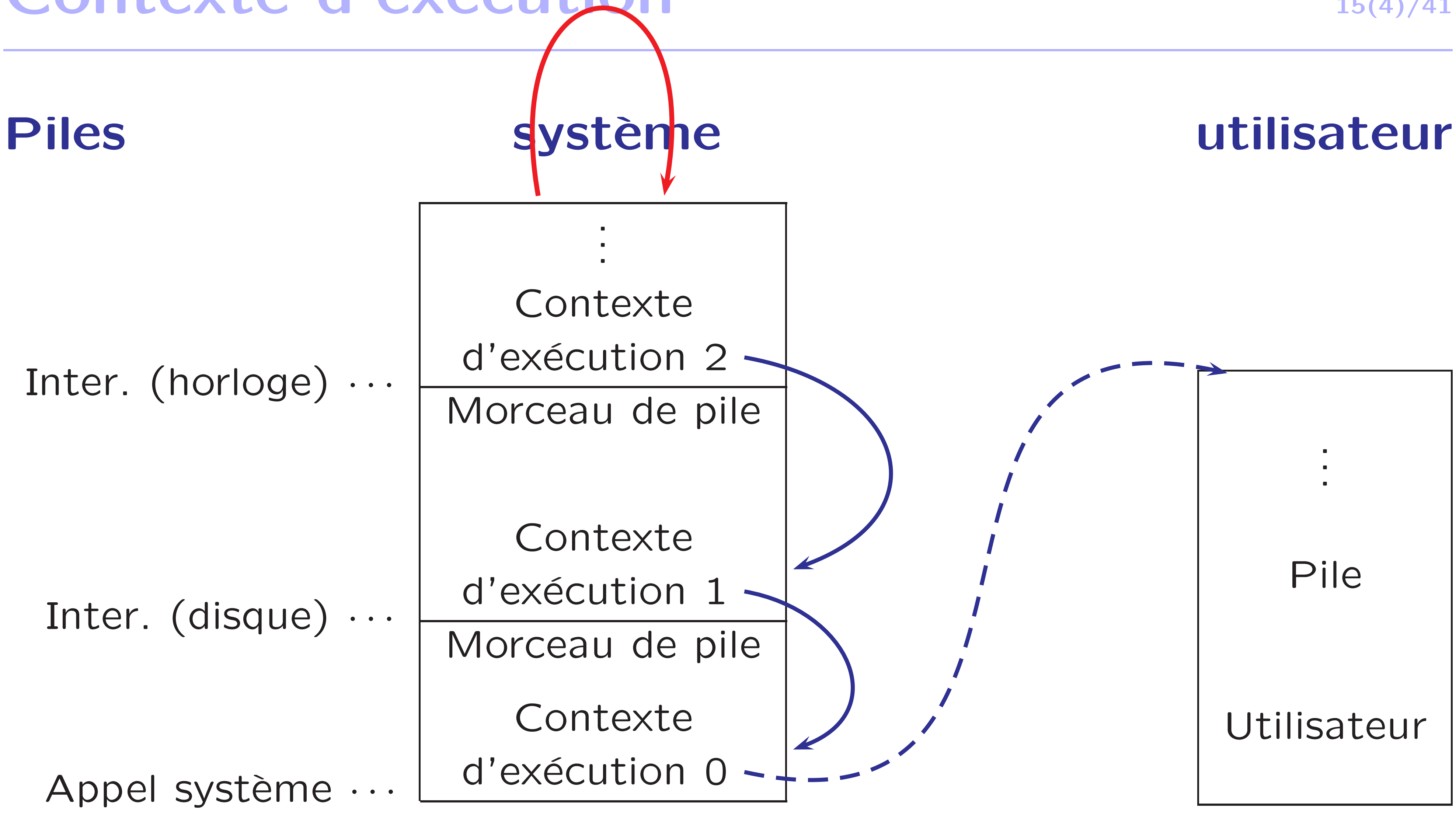


Contexte d'éxécution

Piles **système** **utilisateur**



Contexte d'éxécution



Contexte d'un processus

Qu'est-ce ?

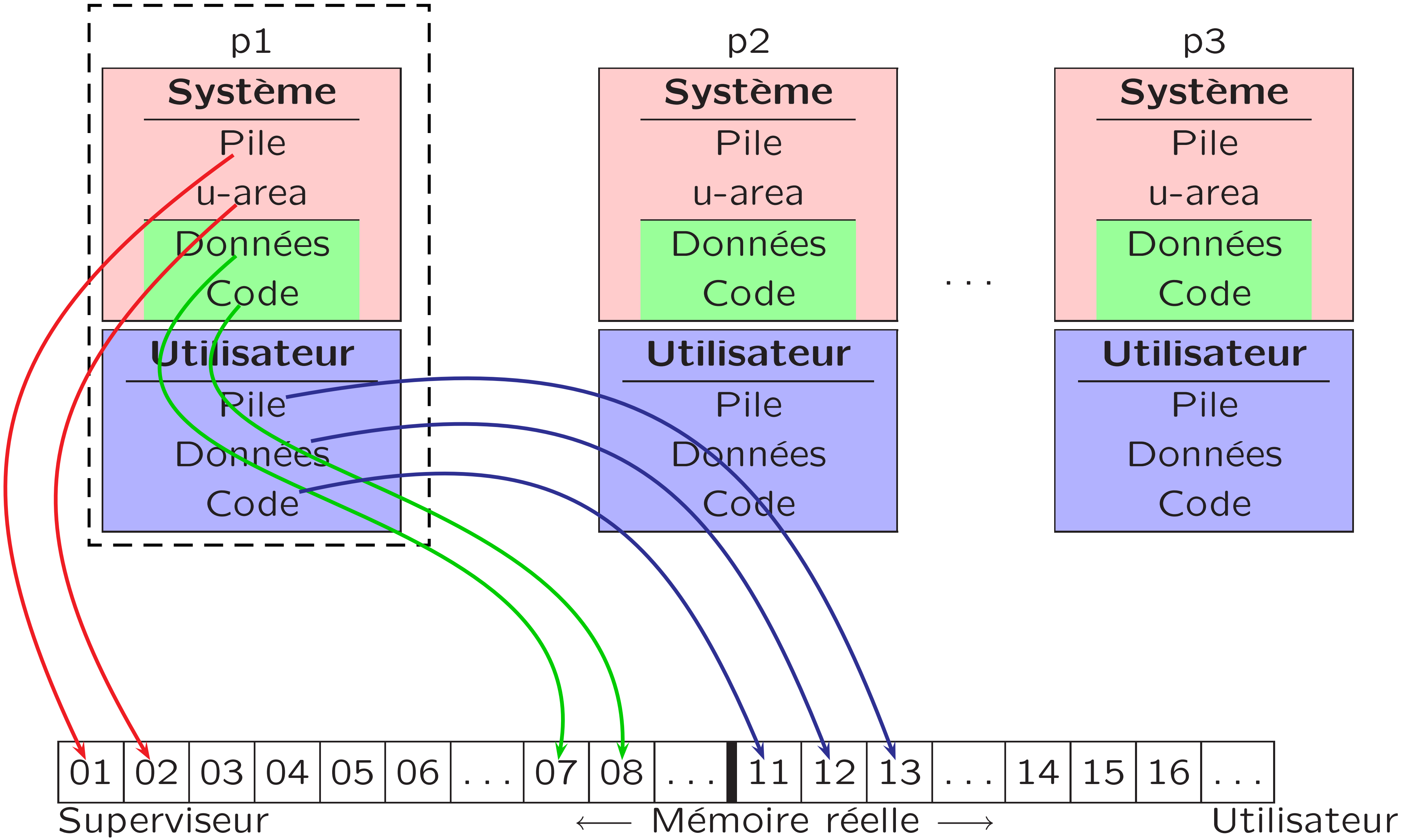
L'ensemble des informations qui déterminent le processus en cours d'exécution. En particulier :

- Les adresses des zônes mémoires :
  - tas, piles utilisateur et système, u-area.
  - Pour chaque zône, la table des pages (correspondance entre mémoire virtuelle et mémoire réelle)
- Le contexte d'exécution (pointeur de pile, registres)

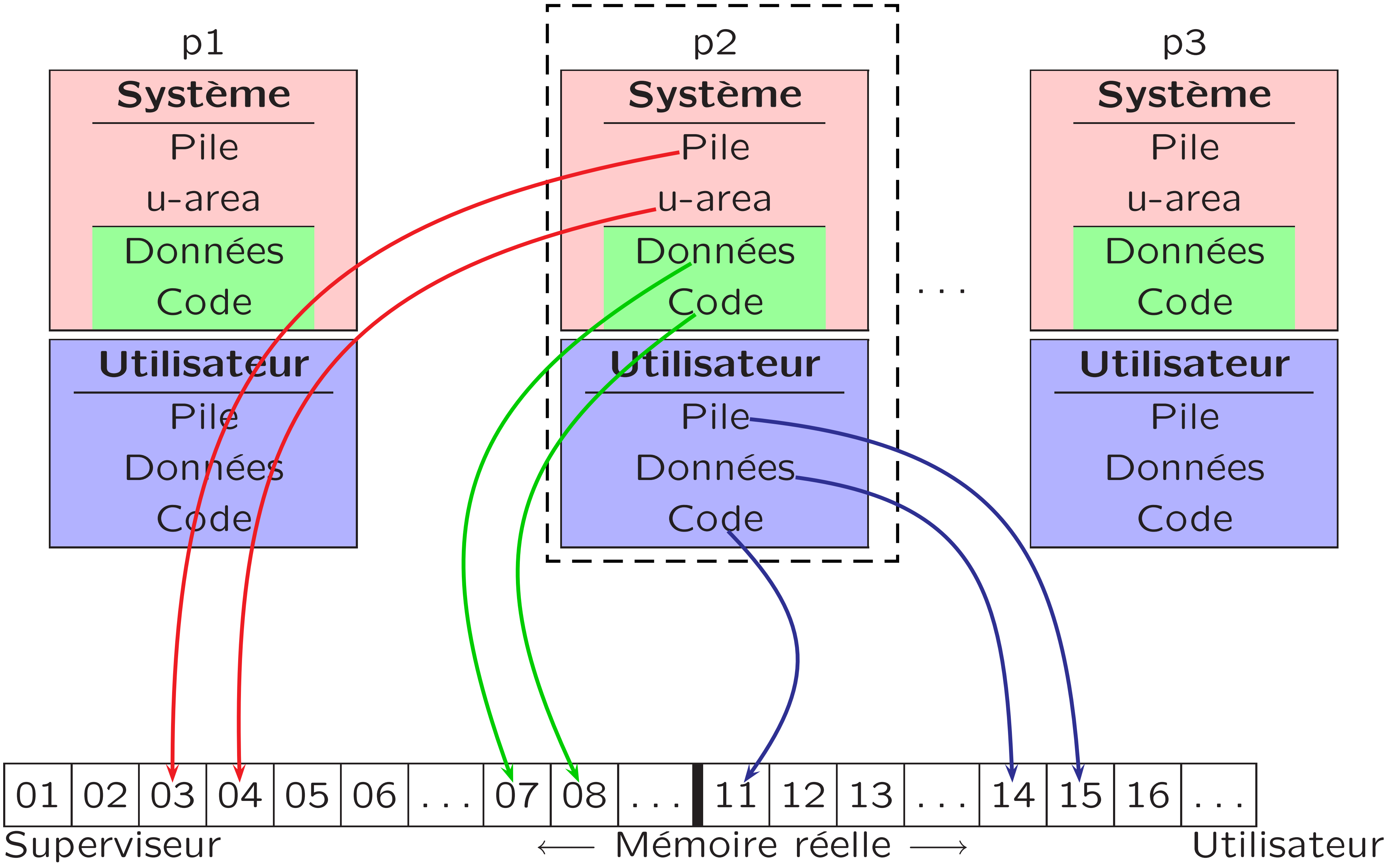
Changement de contexte

- C'est le passage d'un processus à un autre.
- Il faut changer les informations sur le contexte en cours d'exécution.
- Ce qui implique des coûts cachés (e.g. invalider les caches).

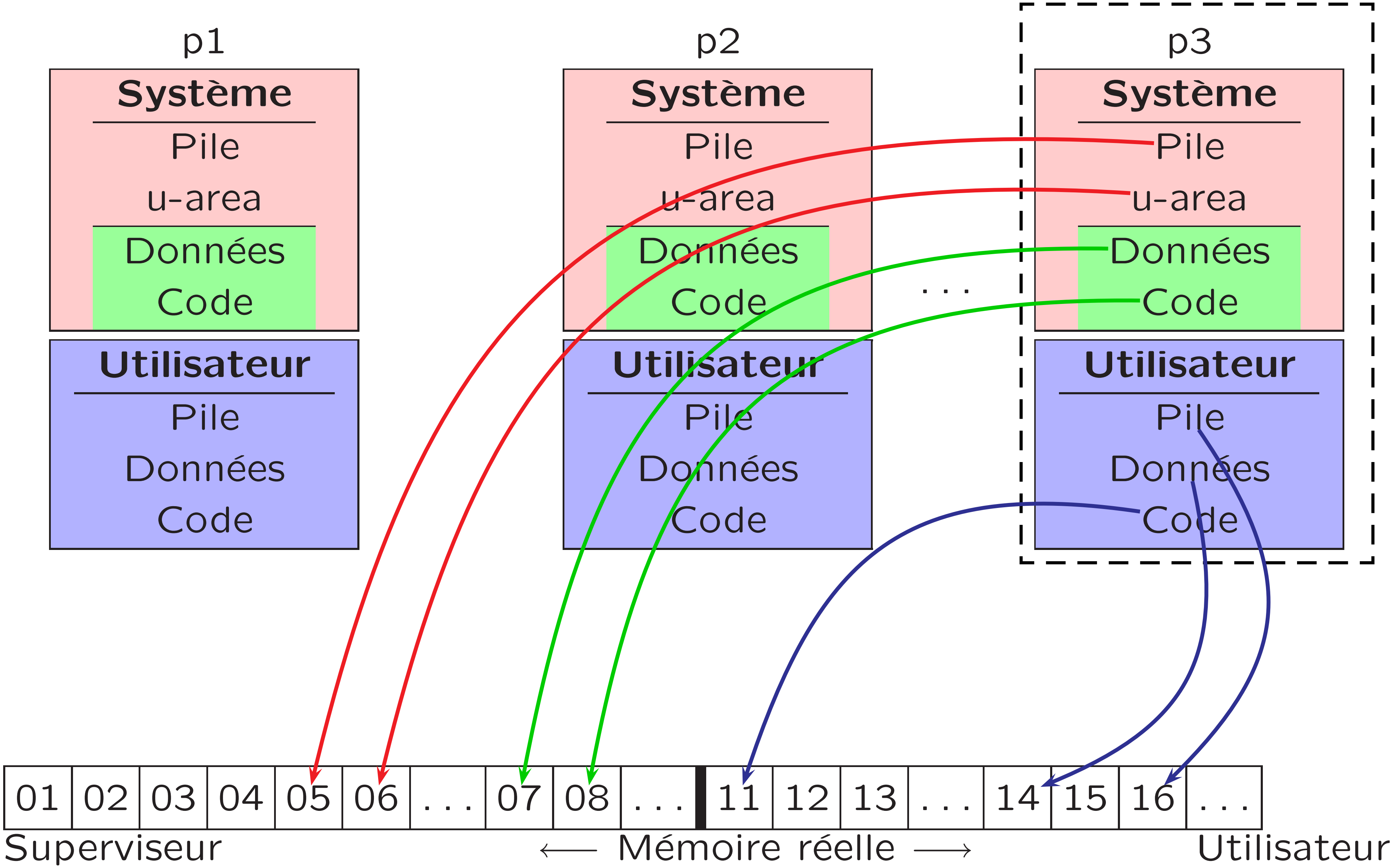
Changement de contexte



Changement de contexte



Changement de contexte



Appel système

Algorithme

- Préparer et placer les paramètres sur la pile.
- Placer le numéro de l'appel dans un registre ou sur la pile.
- Exécuter l'instruction trap avec le numéro «appel système»
  - Passage en mode privilégié, utilisation de la pile système.
  - Décodage du numéro de l'appel, du nombre d'arguments.
  - Copie des arguments dans la u-area (vérifications).
  - sauvegarde du contexte (pour un retour anticipé, *e.g.* signal)
  - Exécution de l'appel (primitive correspondante du noyau)
  - Recopie du résultat de la u-area vers la pile/registres ou place le code d'erreur dans le registre correspondant.
- Reprise de la pile utilisateur et passage en mode utilisateur.
- Traiter l'erreur ou retourner le résultat.

Coût minimal environ 500 instructions cycles.

Phases

- mode utilisateur : code d'emballage par la libc (noir)
- mode utilisateur : code d'appel proprement dit (vert)
- mode superviseur : code de traitement (rouge)

Code du noyau. À chaque appel système `f` correspond

- Une fonction `sys_f` du noyau.
- Un numéro d'entrée dans une table : $n_f \mapsto$ `sys_f`.

Code de la libc. Elle contient

- Une table de définitions `sys_f` $\mapsto n_f$.
- Des macros `syscall0`, `syscall1`, `syscall2`, *etc.* pour mettre les arguments sur la pile selon un emplacement conventionnel.
- Pour chaque appel système une fonction `f` qui effectue essentiellement, si `f` est binaire, `syscall2(`n_f`,arg1,arg2)` .

Phases

- mode utilisateur : code d'emballage par la libc (noir)
- mode utilisateur : code d'appel proprement dit (vert)
- mode superviseur : code de traitement (rouge)

Code du noyau. À chaque appel système `f` correspond

- Une fonction `sys_f` du noyau.
- Un numéro d'entrée dans une table : $n_f \mapsto$ `sys_f`.

Code de la libc. Elle contient

- Une table de définitions `sys_f` $\mapsto n_f$.
- Des macros `syscall0`, `syscall1`, `syscall2`, *etc.* pour mettre les arguments sur la pile selon un emplacement conventionnel.

Vérifications

- Le noyau ne peut pas faire confiance à la `libc`.
- Il doit vérifier le nombre et la forme des arguments.

Schéma

- Il y a une couche en plus : ocaml appelle la libc qui effectue l'appel système.
- Un appel système est (presque) transparent pour OCaml : il s'agit simplement d'un appel à du code C qui se trouve effectuer un appel système.
- OCaml ajoute sa couche d'interprétation/transformation des arguments. mais celle-ci est (relativement) négligeable par rapport au coût de l'appel lui-même. Par ailleurs, elle vérifie la forme des arguments et élimine des erreurs telles que `EINVAL`.

Schéma

Les appels à C en OCaml

- Il suffit de déclarer dans le source (*e.g.* fichier `foo.ml`)

```
external incr : int -> int = "caml_incr"
```

- Implémenter la fonction C (*e.g.* fichier `bar.c`)

```
#include <caml/memory.h>
CAMLprim value caml_incr(value x)
{
  CAMLparam1(x);
  CAMLreturn (Val_int(Int_val(x)+1));
}
```

- Compilation en bytecode :

```
ocamlc -o foo -custom bar.c foo.ml ...
```

En natif bytecode :

```
ocamlopt -o foo bar.c foo.ml ...
```

Schéma

Les appels à C en OCaml

Macros d'interfacage

Fournies dans `.../lib/ocaml/`

- Conversion des arguments et des résultats `<caml/mlvalues.h>`
  Bool_val, Int_val, String_val, ...Val_int, ...
- Informer OCaml des racines du GC `<caml/memory.h>`
- Allouer des valeurs OCaml `<caml/alloc.h>`
- Levée et capture d'exceptions `<caml/fail.h>`
- Configuration de OCaml `<caml/config.h>`

Voir le manuel de référence.

Schéma

Les appels à C en OCaml

Macros d'interfacage

Interfaçage inverse

Il est possible

- d'appeler `OCaml` depuis du code `C` appelé depuis `OCaml`.
- de rendre le code `C` principal et `OCaml` uniquement appelé par du code C.

Voir le manuel de référence.

Change l’identificateur de groupe d’un processus

File setpgid.c

```
#include <unistd.h>
#include <caml/memory.h>
#include "unixsupport.h"
CAMLprim value caml_setpgid(value pid, value pgid)
{
  CAMLparam2 (pid, pgid)
  if (setpgid(Int_val(pid),Int_val(pgid)) == -1)
     uerror("setpgid", Nothing);
  CAMLreturn (Val_unit);
}
```

Repport d’erreur : Entête `unixsupport.h` à prendre dans les sources.

unixext.ml

```
external setpgid : int -> int -> unit = "caml_setpgid"
```

Exemple single_write

Attention aux appels systèmes bloquants !

```
CAMLprim value caml_single_write
        (value fd, value buf, value ofs, value vlen) {
  CAMLparam4(fd, buf, ofs, vlen);
  long numbytes;
  int ret = 0;
  char iobuf[UNIX_BUFFER_SIZE];
  numbytes = Long_val(len);
  if (numbytes > UNIX_BUFFER_SIZE)
    numbytes = UNIX_BUFFER_SIZE;
  memmove (iobuf, &Byte(buf, Long_val(ofs)), numbytes);
    enter_blocking_section();
    ret = write(Int_val(fd), iobuf, (int) numbytes);
    leave_blocking_section();
  if (ret == -1) uerror("write", Nothing);
  CAMLreturn (Val_int(ret));
}
```

Table des interruptions

C'est une structure fixe du système.

Table d'interruption

Numéro	**Code**
0	`clockintr`
1	`diskintr`
2	`ttyintr`
...	...
5	`softintr`

Table des interruptions

Traitement des interruptions

1. sauver le contexte système.
2. determiner la source de l'interruption.
3. trouver le code de traitement de l'interruption.
4. exécuter le traitement.
5. restaurer le contexte système.

Changement de contexte

C'est le passage de l'exécution d'un processus à un autre.

À des instants bien choisis par le système

($\neq$ interruptions) : invariants plus facile à assurer.

- un processus (en mode système) attend une ressource.
- un processus est prêt à reprendre en mode utilisateur.

Changement de contexte

C'est le passage de l'exécution d'un processus à un autre.

À des instants bien choisis par le système

Algorithme

1. Vérifier que le changement de contexte est souhaité/permis
2. Sauver tout le contexte du processus
   (contexte d'exécution, organisation mémoire du processus)
3. l'ordonanceur choisit alors un autre processus à exécuter.
4. restaurer le contexte du nouveau processus.

Coût élevé

- changement de la configuration mémoire : direct (chargement des tables) et indirect (induit par la rupture de flux, caches invalidés)
- coût de l'ordonnancement (choisir un processus).

Ordonnancement

Rôle

C'est la partie du système qui détermine quand changer de contexte et à quel processus donner alors la main.

Contraintes

Équité : empêcher un processus de mobiliser le CPU au détriment des autres.

Réactivité : donner l'impression que tous les processus ont à chaque instant un pourcentage du CPU (pas d'à-coups).

Efficacité :

- ▷ le coût (non nul) de l'ordonancement doit rester faible par rapport au temps réellement alloué aux processus.
- ▷ maximisé l'utilisation du CPU et des périphériques.

Ordonnancement

Rôle

C'est la partie du système qui détermine quand changer de contexte et à quel processus donner alors la main.

Contraintes

Enjeux

- Ce n'est pas difficile à comprendre, mais...
- L'ajustement des paramètres est souvent crucial et délicat.
- Un mauvais algorithme d'ordonancement peut faire qu'un système ne « tienne pas la charge »,
  i.e. à partir d'une certaine charge, l'une des propriétés se dégrade fortement, entrainant souvent une perte d'efficacité qui retarde d'autant le retour à un fonctionnement normal.

Différents types de processus

Les processus

- Ceux qui calculent :
  Demandent beaucoup de CPU, peu d'autres resources.
- Ceux qui communiquent :
  Demandent peu de CPU, mais beaucoup d'autres ressources.

Deux raisons de changer de contexte

- Nécessaires : le processus se met en attente d'une ressource.
- Préemptés : le processus calcule depuis longetmps.

Efficacité bien entrelacer les deux types de processus.

Donner la main par priorité à ceux qui consomment peu de CPU :

- Il vont la rendre tout de suite
- Si ont leur donne la main plus tard, il n'y aura peut-être plus de calcul à faire en parallèle pendant l'attente des ressources.

Stratégies possibles

Utiliser différentes files d'attentes

Par exemple selon le type du processus.

Donner à chaque processus un quantum de temps

- Après écoulement du temps, le processus est préempté.
- Le reste du dernier quantum non-utilisé sert de priorité.
- Les quantums de temps des processsus sont diminués de façon exponentielle en fonction du temps.
- Le quantum de temps d'un processus choisi est remis à zéro.

Ordonnancement par tirage au sort

On distribue des tickets (en fonction de la priorité que l'on veut donner à chacun). On tire un ticket au hasard. Permet un contrôle statistiquement très fidèle des priorités.

Algorithme de Round Robin

Principe

- Le noyau choisit un processus de plus haute priorité dans la file d'attente des processus prêts à s'exécuter et lui donne un quantum de temps.
  - Si plusieurs processus ont la même priorité, celui le plus longtemps en attente est choisi.
  - Si aucun processus n'est prêt, le système se met dans l'état de repos («idle»).
- Le noyau préempte un processus lorsqu'il a épuisé son quantum de temps, et le replace dans une file d'attente.

Quand l'ordonnanceur est-il appelé ?

- au retour d'un changement de contexte.
- les priorités sont réajustées à intervalles réguliers (sur interrup -tion d'horloge), ce qui peut pré-empter le processus en cours.

Raffinement

Le système utilise des niveaux de priorités multiples, *e.g.* :

Noyau	Niveau de priorité	File de Processus
↑	Attente d'être ravivé	
	Attente d'I/O sur disque	$\longrightarrow p_1 \longrightarrow p_2 \longrightarrow \ldots p_5$
	Attente d'un bloc/inode	$\longrightarrow p_6 \longrightarrow p_7$
	Attente d'I/O sur TTY	$\longrightarrow p_8$
	Attente d'un fils	$\longrightarrow p_9 \longrightarrow p_{10}$
Barrière	Niveau utilisateur 0	
↑	Niveau utilisateur 1	$\longrightarrow p_{11} \longrightarrow p_{12} \longrightarrow p_{13}$
Utilisateur	Niveau utilisateur 2	$\longrightarrow p_{14} \longrightarrow p_{15}$

Les processus sont interruptibles par un signal.

Les processus ne le sont pas (attente courte).

Les processus bloqués en mode système

Ils sont prioritaires, car ils peuvent bloquer des ressources système que leur réveil va peut-être libérer.

- Le niveau de proirité correspond au type de ressource sur laquelle le processus à été mis en attente.
- Un processus de plus haute priorité risque de libérer des ressources attendues par un processus de plus basse priorité.
  - un processus en attende de disque I/O a verrouillé un bloc
  - un processus qui attend un bloc est donc moins prioritaire.
- Certaines appels système peuvent être interrompus par un signal (vert). d'autres non (rouge).

Explications

Les processus bloqués en mode système

Ils sont prioritaires

Les processus prêts à tourner en mode utilisateur

Ces processus ne bloquent pas de ressources.

- Ils ne bloquent donc pas d'autres processus,
  (sauf peut-être ceux avec qui ils collaborent, *e.g.* par pipe).
- Ils sont organisés par niveau de priorité, qui dépend
  - du « CPU récemment utilisé » (décroissance logarithmique).
    - prioritaires s'ils ont utilisés peu de CPU, même récemment.
    - prioritaires s'ils n'ont pas utilisé de CPU récemment.
  - d'un facteur fixe (priorité du processus, cf. `nice`)
- Au cours du temps, ils changent de niveau de priorité.

Sur interruption d'un périphérique

(typiquement un disque a fini d'effectuer une I/O)

- Le processeur reçoit une interruption.
- Elle est traitée par le processus en cours d'exécution qui réveille le processus bloqué (le met à l'état prêt).
- Au prochain ordonnancement, le processus pourra être choisi.

Ajustement des priorités

Sur interruption d'un périphérique

(typiquement un disque a fini d'effectuer une I/O)

Lors de la libération d'une ressource système

- Un processus libère un bloc.
- Il reveille tous les processus qui attendaient ce bloc ou un bloc.
- Au prochain ordonnancement, le processus pourra être choisi.

Ajustement des priorités

Sur interruption d'un périphérique

(typiquement un disque a fini d'effectuer une I/O)

Lors de la libération d'une ressource système

Sur interruption d'horloge

Le système programme une interruption d'horloge qui provoque l'exécution d'une fonction `clock` à intervalles réguliers qui :

- ajuste le temps CPU du processus en cours d'exécution.
- si cela n'a pas été fait depuis un certains temps, ajuste le « CPU récemment utilisé » de chaque processus en attente.
- si le temps CPU du processus en cours dépasse un certain seuil, effectue un changement de contexte.

Partage équitable du temps

Problème

- L'unité d'équité est le processus
- Sur une machine multi-utilisateurs, si un utilisateur lance 100 fois plus de processus que les autres, il va avoir à peu près 100 fois plus de CPU.

Solution

- On peut ajouter une notion de groupes de processus
- Compter le CPU totalisé par un groupe de processus et l'utiliser comme critère prépondérant pour calculer la priorité.

Systèmes temps réel

Un système temps-réel doit pouvoir effectuer des tâches avec un temps de réponse très petit : *e.g.* réagir à un signal extérieur, à une interruption d'horloge.

Difficultés

- Le schéma proposé ne s'applique pas au temps réel.
- Le noyau ne peut pas exécuter immédiatement un processus :
  - le processus en cours d'exécution peut être préempté
  - cela ne garantit pas que le processus désiré puisse être sélectionné, car il peut être en attente d'une ressource bloquée par un autre processus.
- Il faut donc des architectures du système d'exploitation significativement différentes pour du *vrai temps réel*.

Ordonnancement en Linux

Linux se différencie légèrement d'Unix et utilise 3 files d'attentes :

1. Real-time FIFO : n'est pas préempté, sauf par un plus récent de cette catégorie.
2. Real-time round robin : peuvent être préemptée par l'horloge.
3. Time sharing.

L'ordonnancement n'est pas du tout temps réel (sans garantie).

Chaque processus (en fait thread) a une priorité. Un calcul de préférence et de quantum assure :

- qu'un processus d'une catégorie k est toujours choisi avant un processus d'un catégorie $k' > k$.
- un processessus qui fait des I/O est préféré à celui qui calcule ;
- un quantum de temps plus grand à un processus plus prioritaire.

Les machines multi-processeurs

Problème

- Dans sa conception, le système Unix est mono-processeur : il suppose un seul CPU, une seule mémoire et des périphériques.
- Le système donne l'illusion d'une exécution en parallèle.

Machines multi-processeurs

(Plusieurs CPU, mais une seule mémoire [espace d'adressage].)

- Plusieurs processus peuvent s'exécuter réellement en parallèle.
  - À un instant, un processus s'exécute sur un seul processeur.
  - Il peut migrer et s'exécuter plus tard sur un autre processus.
- Chaque processeur peut donc aussi exécuter s'exécuter en même temps le code du noyau.
- **Le noyau n'a pas été conçu pour cela.
  Comment préserver ses invariants ?**

Solution

- Un processeur maître centralise tous les appels système.
- Les autres processeurs n'exécutent que du code utilisateur.
- Un appel système dans un esclave suspend l'exécution en indiquant que seul le maître peut la reprendre.

Algorithme `syscall` (version centralisée)

- Si le processeur n'est pas le maître,
  - Affecter le champ `processor` à 1 (celui du maître).
  - Faire un changement de contexte
- Effectuer l'appel normalement.
- Metter le champ `processor` à 0 (peu importe).
- Si d'autres processus attendent un appel système faire un changement de contexte (pour libérer le processeur maître).

Multi-processeurs centralisés

Solution

- Un processeur maître centralise tous les appels système.
- Les autres processeurs n'exécutent que du code utilisateur.
- Un appel système dans un esclave suspend l'exécution en indiquant que seul le maître peut la reprendre.

Algorithme `syscall` (version centralisée)

Problèmes

- Le temps système peut être un goulot d'étranglement.
  (Un processeur n'est pas suffisant s'il y a beauoup de processeurs auxilliaires ou si le système est très solicité.)
- Le processeur maître doit pouvoir être interrompu/préempté lorsqu'un qu'il exécute du code utilisateur, et qu'un autre processus veut faire un appel système.
- Beaucoup plus de changements de contextes.

Solution

- Un processeur maître centralise tous les appels système.
- Les autres processeurs n'exécutent que du code utilisateur.
- Un appel système dans un esclave suspend l'exécution en indiquant que seul le maître peut la reprendre.

Algorithme `syscall` (version centralisée)

Problèmes

Ordonnancement

- Le maître choisit le processeur sur lequel un processus doit tourner. Il faut prévoir un équilibre de la charge.
- Sinon, il faut un mécanisme de verrou/sémaphore pour empêcher que deux processeurs ne choisissent en même temps le même processus prêt.

Solution symmétrique (plus fréquente)

Principe

- Tous les processeurs sont équivalents.
- Ils peuvent tous exécuter du code système en parallèle.
- À l'exclusion des sections critiques protégées par des verrous/sémaphores (*c.f.* coprocessus)

Revoir la plupart des primitives (gestions des verrous)

- Avec un seul processeur, le système pouvait exécuter plusieurs instructions de façon atomiques en bloquant les interruptions.
- Avec plusieurs processeurs, il faut, en plus, en faire des sections critiques contrôlées par des verrous.

Exemple : `getblk`

Nombreux verrous/sémaphores Par exemple :

- Chaque bloc est controlé par un verrou (comme avant)
- Chaque file d'attente (baquet et liste libre) est controlée par un verrou.

« Spin lock »

Les nouveaux verrous contrôlent l'accès à des structures système (manipulation de listes libles) et ne sont pris que très brièvement.

À distinguer des verrous sur les structures système elles-mêmes (bloc, inode, *etc.*) qui existaient déjà en version monoprocesseur et qui peuvent être pris pendant un temps plus long.

Les nouveaux verrous sont en attente active (très courte), *e.g.*

```
while not (try_lock free_list) do done ;;
```

«spin lock»

Il a un coût (transit sur le bus) : nombreuses astuces pour le réduire.

Faut-il mieux faire un «spin lock» ou un changement de contexte ? Le choix entre ces deux stratégies peut se faire dynamiquement en maintenant des informations statistiques.

Scheduling

Il ne s'agit plus seulement de choisir un processus, mais de choisir un processus sur un processeur. Le placement est souvent significatif, par exemple, essayer qu'un processus reste sur le même processeur.

Hyper-threading

Un seul processeur mais deux jeux de registres : il peut entrelacer l'exécution de deux threads sans changement de contexte (la granularité est de l'ordre de l'instruction) afin d'optimiser l'utilisation des resources et réduire la latence (sauts, lectures mémoire).

Cela pose toutes les difficultés des multi-processeurs.
(Ici, le «spin-lock» gâche vraiment du CPU).

Processeurs Multicore (tendance)

Ce sont plusieurs processeurs sur la même puce (alors que les multi-processeurs classiques sont sur des puces différentes).

Multicore + Hyper-threading (très tendance)

Conclusions sur les multi-processeurs

Dans les deux modèles

- Le parallèlisme a un surcoût.
  - plus de changements de contextes (attente des resources).
  - gestion des verrous + attente (souvent active) des verrous.
- qui croît avec le nombre de processeurs.
- la mémoire peut devenir un goulôt d'étranglement.
- à un certain point l'ajout d'un nouveau processeur n'apporte plus de gain.

Problème souvent minimisé (sauf pour les jeux...)

Difficulté d'écrire des algorithmes exploitant bien le parallèlisme.

Machines distribuées

- Plusieurs machines Unix intimenent liées.
- Chaque machine à sa copie du système d'exploitation. (structures systèmes duppliquées et indépendentes)
- Communication de plus grande granularité, difficile, plus chère.