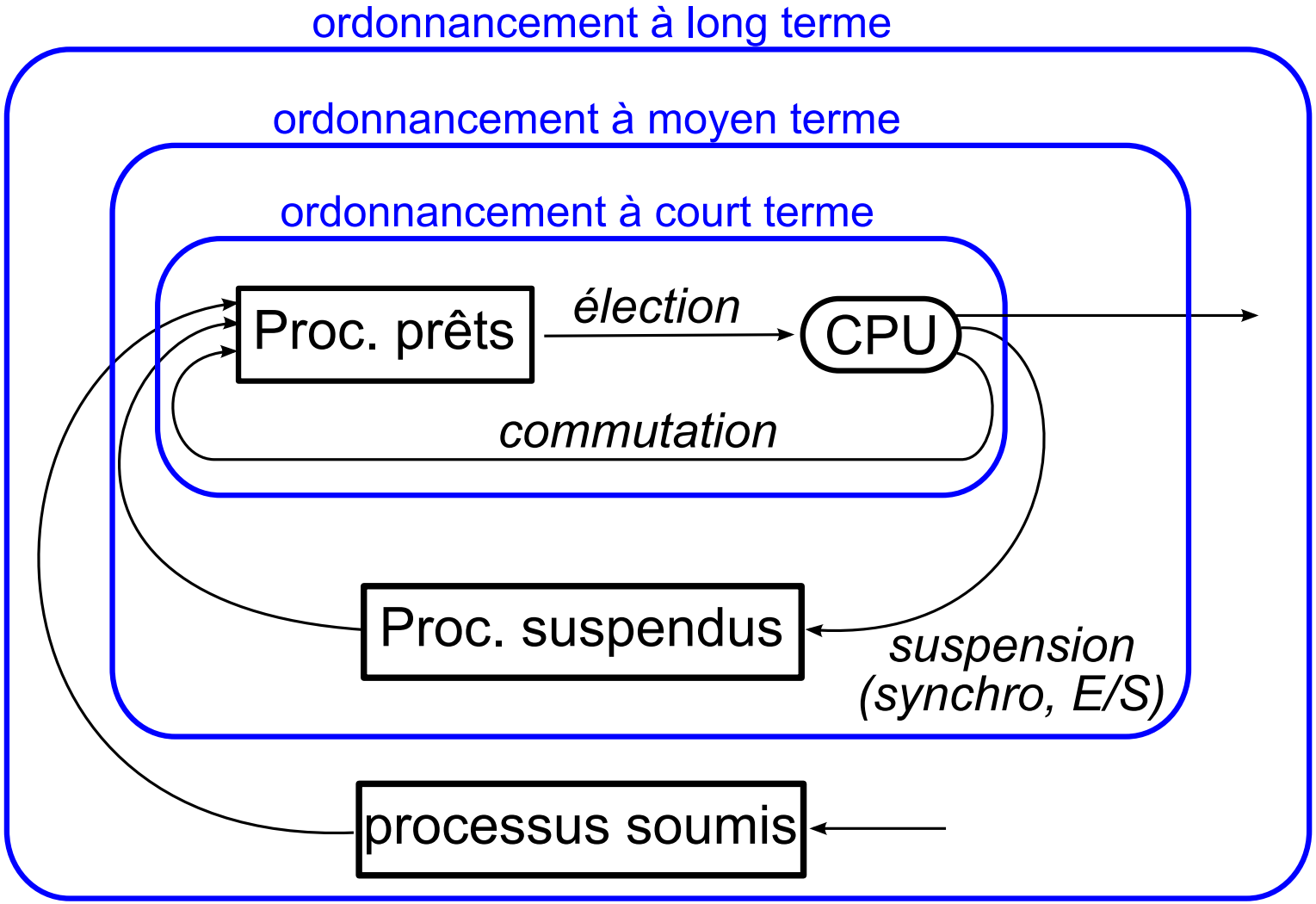
Couche Processus

Philippe Quéinnec

18 mai 2017

Plan

Les trois niveaux d'ordonnancement



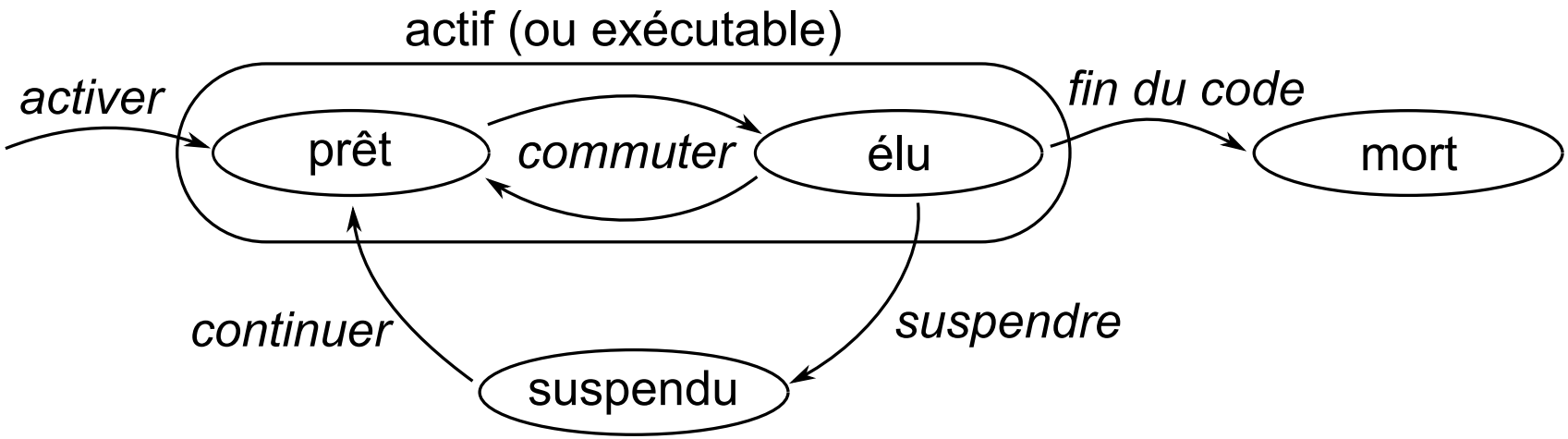
Processus exécutable = prêt ou élu

Couche processus élémentaires

Trois modules :

- processus élémentaire
- ordonnanceur préemptif
- TSD (données spécifiques à chaque processus)
- ...

Cycle de vie


actif (ou exécutable)
activer
prêt
commuter
élu
fin du code
mort
continuer
suspendre
suspendu

Interface

- Type opaque `processus_t`
- création d'un nouveau processus

```
processus_t proc_activer(char *nom,
                         void (*code)(void *a), void *arg)
```

- changement de processus élu

```
void proc_commuter()
```

- suspension du processus appelant

```
void proc_suspendre()
```

- remise à prêt du processus spécifié

```
void proc_continuer(processus_t p)
```

- soi-même

```
processus_t proc_self()
```

Implantation (1)

```
typedef enum {
  proc_PRET, proc_ELU, proc_SUSPENDU, proc_MORT
} proc_status;

typedef struct processus {
    char *nom;
    proc_status état;
    void (*code) (void *); /* Le code du processus */
    void *arg;             /* et son argument. */
    coroutine_t cor;       /* la coroutine sous-jacente. */
} *processus_t;

// variables globales
processus_t élu;  /* le processus s'exécutant (l'ÉLU) */
file<processus_t> les_prêts; /* les processus PRÊTS. */
```

Implantation (2)

```
void proc_commuter()
{
    processus_t ancien_élu = élu;
    ancien_élu->état = PRÊT;
    insérer_dans_file_des_prêts(ancien_élu);
    élu = extraire_un_prêt();
    élu->état = ÉLU;
    cor_transférer(ancien_élu->cor, élu->cor);
}
```

Implantation (3)

```
void proc_suspendre()
{
    processus_t ancien_élu = élu;
    élu->état = SUSPENDU;
    if (file_des_prêts_est_vide()) { /* arrêt machine */ }
    élu = extraire_un_prêt();
    élu->état = ÉLU;
    cor_transférer(ancien_élu->cor, élu->cor);
}

void proc_continuer(processus_t p)
{
    assert(p->état == SUSPENDU);
    p->état = PRÊT;
    insérer_dans_file_des_prêts(p);
}
```

Implantation (4)

```
processus_t proc_activer(char *n, void (*code)(void *a), void* a
{   processus_t nouv = malloc(sizeof(struct processus));
    nouv->nom = strdup(n);
    nouv->code = code; nouv->arg = arg;
    nouv->état = PRÊT;
    nouv->cor = cor_créer(nom, enveloppe, nouv);
    insérer_dans_file_des_prêts(nouv);
    return nouv;
}
static void enveloppe (void *pt)
{   processus_t p = (processus_t)pt;
    (p->code) (p->arg);
    p->état = MORT;
    if (file_des_prêts_est_vide) { /* arrêt machine */ }
    élu = extraire_un_prêt();
    élu->état = ÉLU;
    cor_transférer(p->cor, élu->cor);
}
```

Destruction

- Pas de possibilité de suspendre ou de tuer un processus (mais ça ne serait pas bien dur, cf préemption plus loin).
- Quand a lieu le ménage (en particulier libération des ressources mémoires allouées) ?
  - Quand le code du processus se termine
    $\Rightarrow$ à la fin de l'enveloppe ;
  - Mais il n'est pas possible de détruire la coroutine et en particulier la pile d'exécution tant qu'elle est en service
    $\Rightarrow$ ramasse-miettes élémentaire (ranger la coroutine à détruire dans une liste de nettoyage à faire, nettoyage effectué « plus tard », par exemple lors de la création d'un processus).

Plan

Interruptibilité

Objectifs

- Atomicité d'une séquence d'instructions (pas de déroutement, pas d'interruption, pas de préemption)
- masquage des signaux

Mise en œuvre

Un masque de signaux pour chaque processus élémentaire + opérations de masquage

- Ajouter dans le descripteur de processus un champ : `sigset_t mask;`
- Ce champ n'est valide que
  - si le processus n'est pas élu,
  - ou si le processus est élu et non interruptible.
- Sinon (processus élu interruptible), on utilise le masque de la machine support (masque processeur ou masque Unix).

Implantation

- Manipulation des signaux : `sigprocmask(mode, nouveau, ancien)`, où
  - mode = SIG_SETMASK : fixe le masque courant à nouveau ;
  - mode = SIG_BLOCK : ajoute nouveau au masque courant ;
  - mode = SIG_UNBLOCK : enlève nouveau du masque courant ;
  - le masque précédent est sauvé dans ancien.
- Processus initial : valeur par défaut = valeur courante du masque unix. Dans `proc_init` :
  ```
  sigprocmask(SIG_SETMASK, NULL, &(élu->mask));
  ```
- Lors de l'activation d'un nouveau processus : héritage du masque du processus créateur. Dans `proc_activer` :
  ```
  nouveau-> mask = élu->mask;
  ```

Non-interruptibilité

rendre_noninterruptible

= Tout masquer

```
rendre_noninterruptible() {
    sigset_t protect;
    sigfillset(&protect);
    sigprocmask(SIG_SETMASK, &protect, &(élu->mask));
}
```

rendre_interruptible

= Restaurer l'ancien masque

```
rendre_interruptible() {
    sigprocmask(SIG_SETMASK, &(élu->mask), NULL);
}
```

Règles de bonne utilisation

1. Quand un processus cesse d'être élu (commutation, suspension, mort), le système doit être en mode non interruptible.
2. Quand un processus devient élu, le système est en mode non interruptible, et il doit le rendre interruptible (ou cesser d'être élu).

Préemption

Préemption = commutation forcée (par le noyau) après une certaine utilisation de ressources (quantum de temps) ⇒ appel à commuter sur réception d'une interruption d'alarme émise par une horloge.

```
void armer_timer() {
    struct itimerval delai;
    delai.it_value = { 0 /* sec */, 100 /* msec */ };
    delai.it_interval = { 0, 0 };
    setitimer (ITIMER_REAL, &delai, NULL);
}
void commutateur(void) {
    rendre_noninterruptible();
    armer_timer();
    proc_commuter();
    rendre_interruptible();
}
```

Activation de la préemption

```
void activer_preemption ()
{
 struct sigaction act;
 act.sa_handler = commutateur;
 sigemptyset (&(act.sa_mask));
 act.sa_flags = 0;
 act.sa_flags &= ~SA_SIGINFO; /* use sa_handler, not sa_sigaction */
 act.sa_flags &= ~SA_RESETHAND;/* don't reset the handler to SIG_DFL
 act.sa_flags |= SA_NODEFER; /* don't block the signal in handler */
 act.sa_flags |= SA_RESTART; /* restart interrupted system call */
 sigaction (SIGALRM, &act, NULL);
 armer_timer ();
}
```

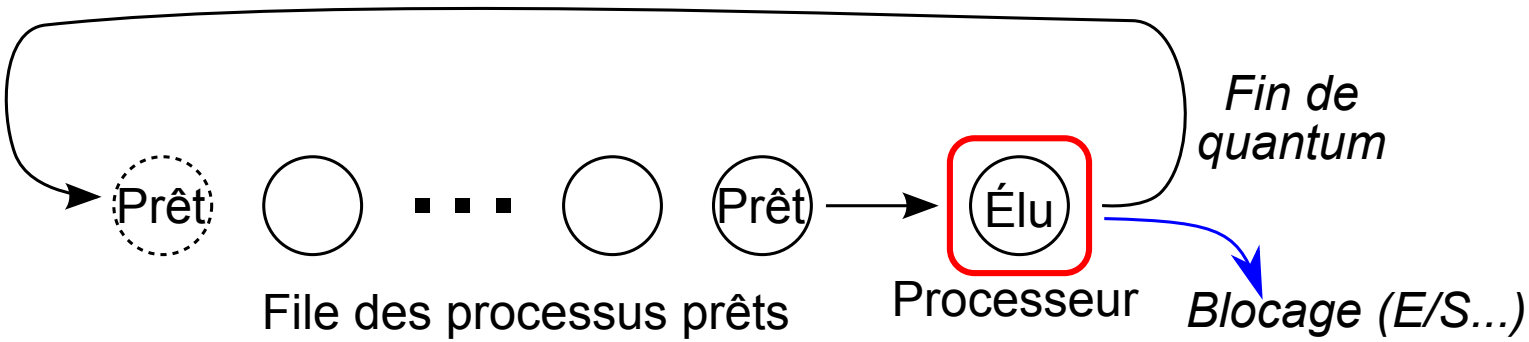
Stratégie d'ordonnancement

Choix du processus élu :

- Quantum de temps et tourniquet (round-robin)
- Priorités
- Stratégie temps réel

Tourniquet (roud-robin)

- Quantum = durée maximale d'activité continue.
- Préemption du processeur au profit d'un autre processus prêt
  - soit en fin de quantum ;
  - soit sur blocage du processus actif.
- Adaptation possible de la durée du quantum au profil d'exécution (calcul ou entrées/sorties).
- En expiration du quantum, mise en fin de la file des prêts.



Priorités

Priorités relatives fixes

Le quantum est fonction de la priorité

Priorités absolues

La place dans la file est fonction de la priorité.
(ou plusieurs files, une par priorité, round-robin au sein d'une file)

Priorités adaptatives / Multi-niveaux

- Plusieurs files ;
- Quand un processus atteint son quantum, il est préempté et déplacé dans un niveau moins prioritaire ;
- Quand un processus libère le CPU avant la fin de son quantum, il reste au même niveau ;
- Quand un processus se bloque sur une E/S, il sera placé (quand il sera débloqué) à un niveau plus prioritaire.

Temps réel

Processus périodiques

- Processus périodiques : chaque processus a une période + une pire durée d'exécution d'un pas
- Échéance = date à laquelle un processus doit avoir terminé son pas de calcul (= début période suivante − pire durée d'exécution)

Exemples de stratégies

- Stratégie RMS (Rate Monotonic Scheduling) : choix du processus le plus prioritaire : chaque processus (tâche) a une priorité fixe attribuée selon l'ordre croissant de leur fréquence d'exécution ;
- Stratégie EDF (Earliest Deadline First) : choix du processus prêt ayant l'échéance la plus proche.

Thread Specific Data

- Analogue à une variable globale ayant une valeur distincte dans chaque processus.
- Exemple de TSD : nom, priorité, date de création, processus créateur, utilisateur...
- Seul le processus peut lire ou modifier ses données spécifiques.

TSD interface

- Créer une nouvelle clef, commune à tous les processus.
  La fonction `destructeur` sera appelée à la mort d'un processus pour nettoyer sa valeur associée à cette clef.
  ```
  int TSD_créer_clef(void (*destructeur) (void *));
  ```
- Obtenir la valeur associée à la clef pour le processus courant
  ```
  void *TSD_get(int clef);
  ```
- Changer la valeur associée à la clef pour le processus courant, et renvoie la valeur précédente
  ```
  void *TSD_set(int clef, void *val);
  ```

TSD Implantation

Ajouter dans le descripteur de processus un tableau de valeurs :

```
struct processus {
    ...
    void *tsd[TSD_MAX_CLEFS];
}

/* Renvoie un numéro de clef non encore utilisée */
int TSD_créer_clef (void) {
    static int clef_suivante = 0;
    if (clef_suivante == MAX_NB_CLEFS) return -1;
    else return clef_suivante++;
}

void *TSD_get (int clef) {
    return élu->tsd[clef];
}
```

Sleep

Principe

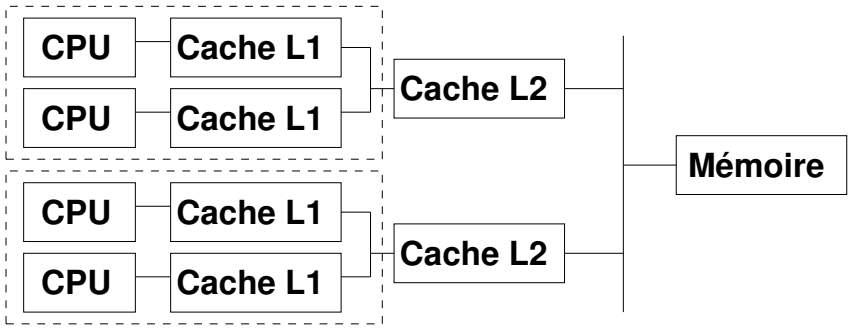
Sleep = se suspendre (en espérant que ça ne dure qu'un certain temps).
Qui débloque (= continuer) ?

- Le noyau, chaque fois qu'il a la main (ici, les procédures `proc_*`) ;
- Un autre processus, dédié à cela.

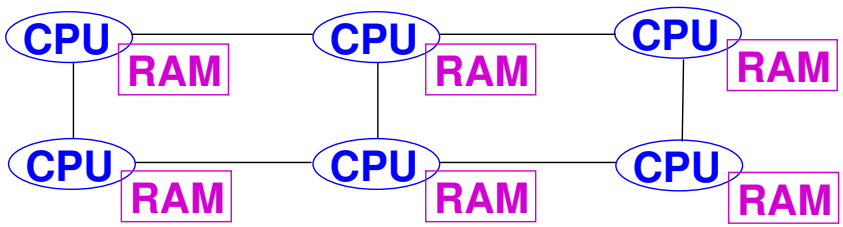
Le processus dédié doit avoir la main « assez souvent » et doit se suspendre quand il n'y a aucun sleep en cours.

Multi-processeurs

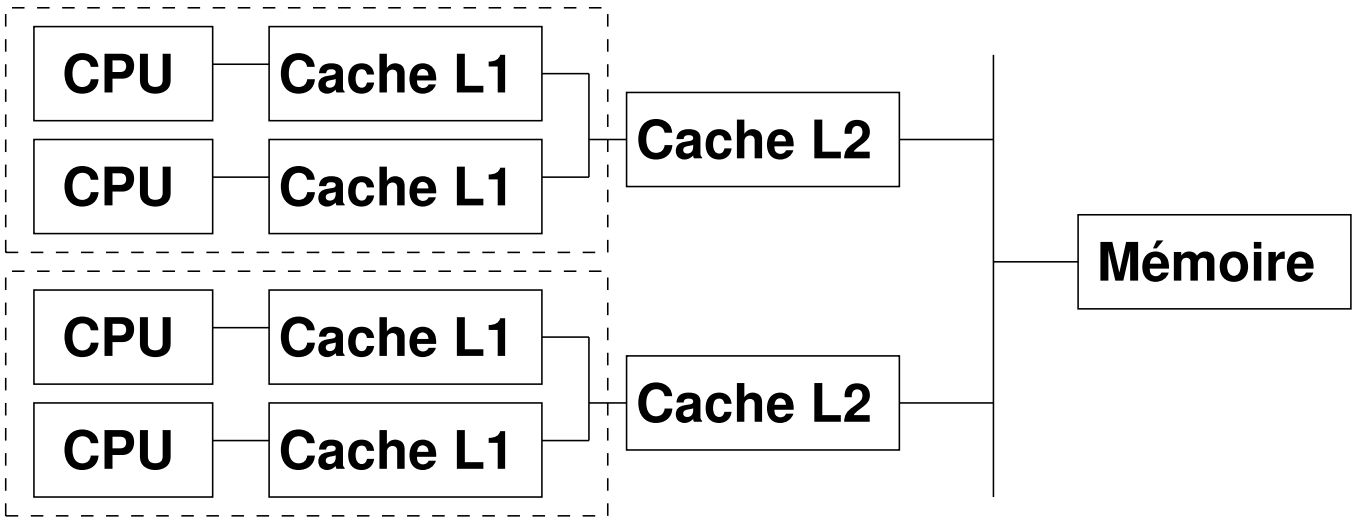
SMP : Symetric Multi-Processor : Plusieurs processeurs, tous identiques (multi-cœurs / multi-processeurs) + une seule mémoire commune + primitives de cohérence de caches.



NUMA : Non Uniform Memory Access : un graphe d'interconnexion de {CPU+mémoire}



SMP Symetric Multi-Processor



- Processeurs tous identiques $\Rightarrow$ le contexte sauvegardé sur un processeur (un cœur) est installable sur un autre processeur.
- Mémoire commune : sauf cas spécifique (calcul haute performance), ignorer la gestion des caches.

Adaptation du noyau

- `getCPU` = identifiant (numéro) du processeur appelant cette opération.
- `élu` ⇒ `élus[nbCPU]`
- Seule difficulté : `élus` et la file des prêts doivent être manipulés avec soin : en exclusion mutuelle ⇒ utilisation d'un verrou avec scrutation (spin lock) vu qu'un processus ne reste pas longtemps dans le noyau.
  Par exemple :

```
proc_suspendre() {
    while (test_and_set(verrou_noyau) == 1) /*loop*/;
    ...
    verrou_noyau = 0;
}
```

NUMA : Non Uniform Memory Access

NUMA

- Plusieurs processeurs, tous identiques ⇒ le contexte sauvegardé sur un processeur (un cœur) est installable sur un autre processeur.
- Plusieurs unités mémoires, accessibles de partout mais aux performances très différentes ⇒ le placement doit prendre en compte les accès mémoire (notion d'affinité : rester sur le même processeur).

Grilles ou grappes de PC

- Ordinateurs (SMP+mémoire) interconnectés par réseau rapide
- Processus attaché à une machine, + éventuellement migration (très coûteuse)
- Programmes applicatifs ad-hoc