Couche Processus

Philippe Quéinnec

14 mai 2020

Plan

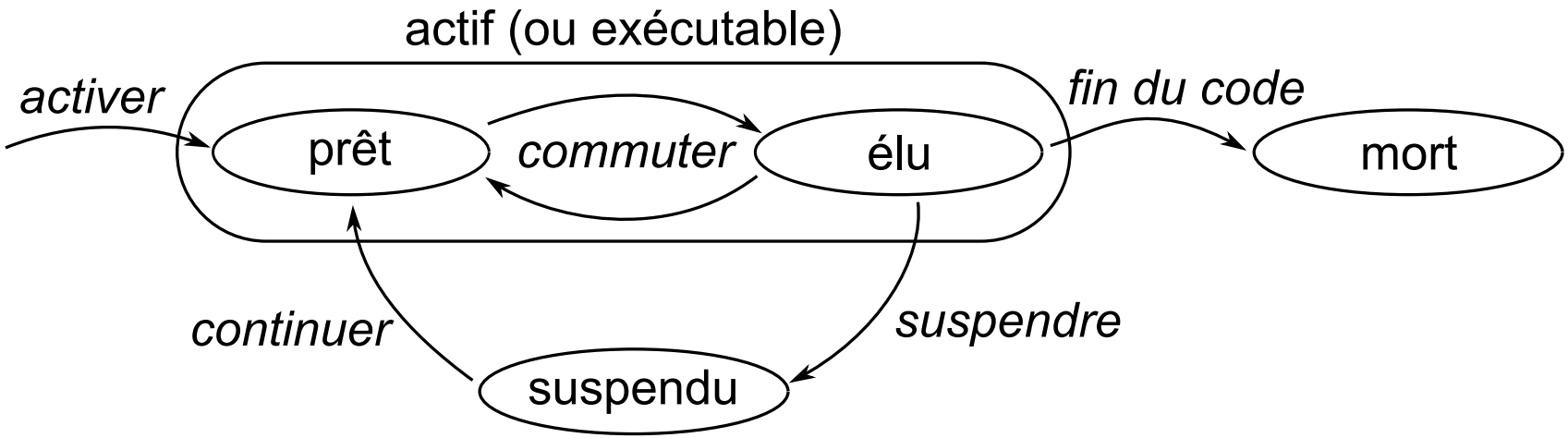
Couche processus élémentaires

Points abordés :

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

Cycle de vie d'un processus

Un processus est un calcul en cours d'exécution. Plusieurs processus peuvent exister même sur un mono-processeur.

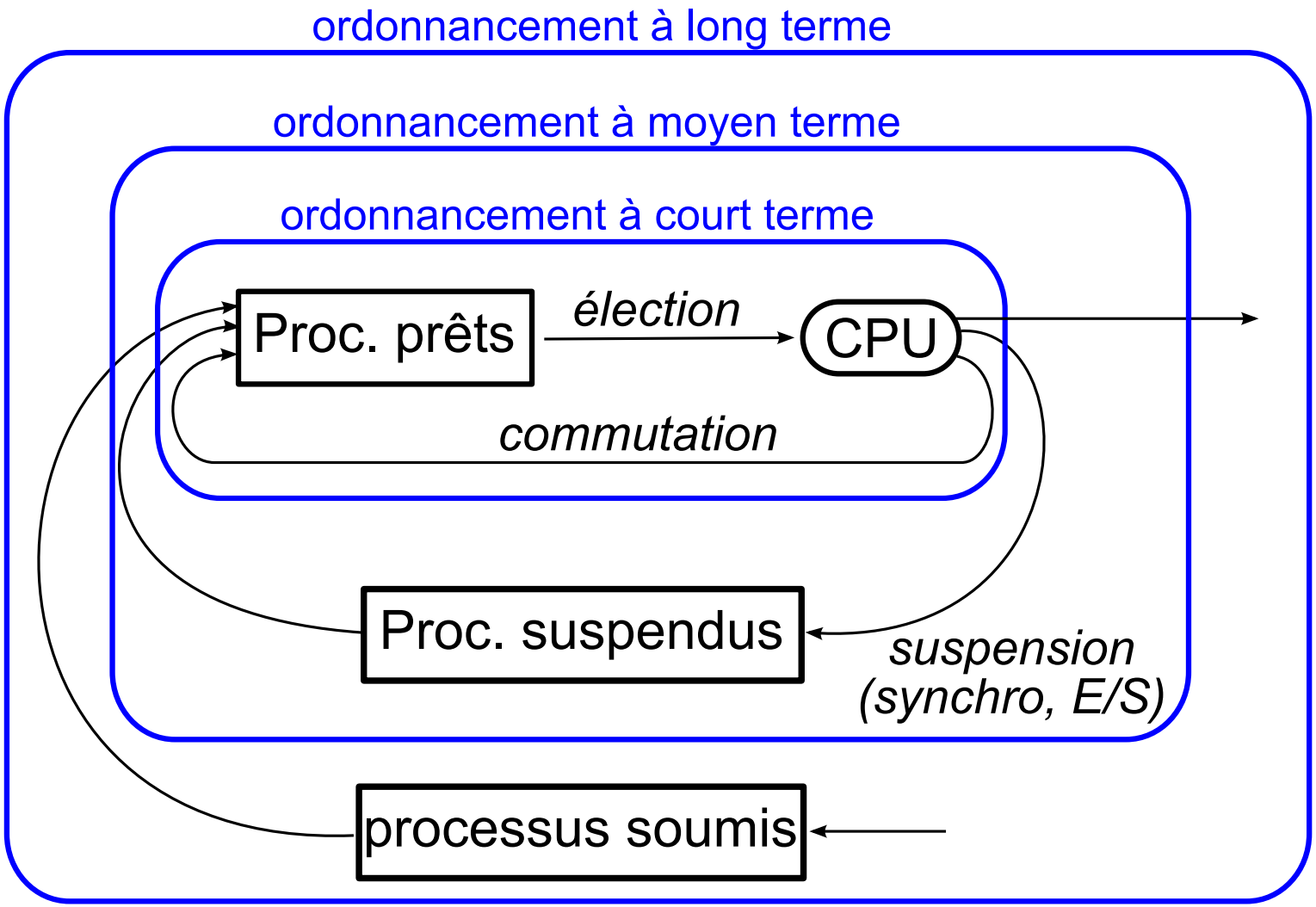


élu effectivement exécuté par le CPU

prêt candidat au CPU

suspendu bloqué, pas candidat au CPU

Les trois niveaux d'ordonnancement



Processus exécutable = prêt ou élu

Interface

- Type opaque `processus_t`
- Initialisation de la couche processus
  `void proc_init();`
- 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()`

Exemple 1

```
/* Terminaison du système avec p1 et p2 terminés,
* et le processus initial (pour main) suspendu. */

void code(void *arg) {
    char *s = (char *)arg;
    for (int i = 0; i < 10; i++) {
        printf("%s %d\n", s, i);
        proc_commuter();
    }
}

/* Un processus est implicitement créé pour exécuter ce code.
 * La terminaison de main entraîne la terminaison du système. */
int main() {
    proc_init();
    processus_t p1 = proc_activer("P1", code, "hello");
    processus_t p2 = proc_activer("P2", code, "coucou");
    preemption_activate(1);
    proc_suspendre();
    return 0;
}
```

Exemple 2

```
processus_t pmain, p1, p2;
int compteur;

void code1(void *unused) {
    for (int i = 0; i < 1000; i++) {
        compteur++; proc_continuer(p2); proc_suspendre();
    }
}
void code2(void *unused) {
    proc_suspendre();
    while (compteur < 10) {
        proc_continuer(p1); proc_suspendre();
    }
    proc_continuer(pmain); /* redonne la main à main */
}
int main() {
    pmain = proc_self();
    p2 = proc_activer("P2", code2, NULL);
    p1 = proc_activer("P1", code1, NULL);
    proc_suspendre();
    return 0; /* terminaison du système */
}
```

Implantation

- Chaque processus contient une coroutine : le point où il en est, qu'il faut installer pour qu'il s'exécute.
- La couche maintient la liste des processus prêts (en attente de CPU) + le processus élu.
- Commuter ou suspendre : changement de processus élu
  $\Rightarrow$ choix d'un nouvel élu et transfert.
- Commuter : insertion de l'ancien processus courant dans les prêts ; pas pour suspendre.
- Continuer : insertion du processus spécifié dans les prêts, pas de changement d'élu.
- Terminaison du code d'un processus $\Rightarrow$ changer d'élu (s'il reste des prêts).
  Détection de cette terminaison ? Une enveloppe !

Implantation – structures globales

```
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 initial du proc. */
    void *arg;             /* et son argument. */
    coroutine_t cor;       /* la coroutine sous-jacente. */
}
typedef struct processus *processus_t;

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

Implantation – commuter

1. Insérer le processus courant (l'ancien élu) dans les prêts
2. Choisir un nouvel élu parmi les prêts
   (politique discutée plus loin)
3. Transfert : sauvegarde de l'état (la coroutine) de l'ancien élu
   et installation de la coroutine du nouvel élu

```
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 – suspendre

1. Choisir un nouvel élu parmi les prêts
   (politique discutée plus loin)
2. Si plus de processus prêt $\rightarrow$ arrêt définitif
3. Transfert : sauvegarde de l'état (la coroutine) de l'ancien élu
   et installation de la coroutine du nouvel élu

```
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);
}
```

Implantation – continuer

1. Mettre le processus spécifié parmi les prêts

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

Implantation – activer

1. Allouer un nouveau descripteur de processus
2. Créer une nouvelle coroutine pour ce processus
   - La coroutine exécutera une enveloppe, laquelle appellera le code utilisateur
   - Au retour du code utilisateur, le processus est terminé
     $\rightarrow$ faire comme suspendre (choix d'un nouvel élu et transfert)
   - Pour que l'enveloppe connaisse le code utilisateur, il est rangé dans le descripteur du processus
3. Insérer le processus parmi les prêts

Implantation – activer

```
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);
}
processus_t proc_activer(char *n, void (*code)(void *a), void* arg)
{   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;
}
```

Commuter vs suspendre/continuer

Les algorithmes de `commuter` et `suspendre` sont proches mais

- `commuter` est pour l'ordonnancement à court terme. L'appelant ne choisit pas qui obtient le CPU, c'est le rôle de l'ordonnanceur.
- Avec la préemption (à venir), le noyau assure que tous les processus prêts obtiennent à leur tour un peu de temps CPU. Le code utilisateur n'utilise plus `commuter`.
- Continuer/suspendre est pour l'ordonnancement à moyen terme. Via `continuer`, le code utilisateur choisit qui est débloqué. Un processus peut rester définitivement suspendu.

Destruction

- Pas de possibilité de suspendre ou de tuer un processus hors soi-même (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

Préemption

- Préemption = commutation forcée par le noyau après une certaine utilisation de ressources (quantum de temps).
- Rappel : quand le CPU exécute du code utilisateur, il n'exécute pas le code du noyau. Et si le code utilisateur ne fait aucun appel au noyau ?

→ appel à `commuter` sur réception d'une interruption d'alarme émise par une horloge.

Préemption

Préemption = appel à un commutateur 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) { /* à chaque IT ALARM */
    armer_timer();
    proc_commuter();
}
```

Activation de la préemption

En gros : connecter commutateur au signal SIGALRAM :
`signal(SIGALRM, commutateur);`
En détail : ne pas masquer le signal dans le handler

```
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 (`extraire_un_prêt`) :

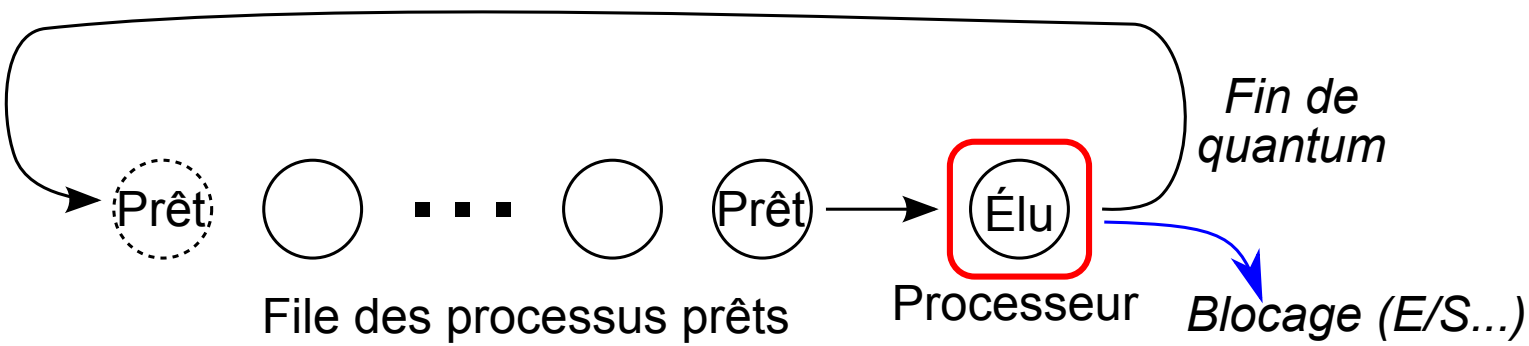
- Quantum de temps et tourniquet (round-robin)
- Aléatoire parmi les prêts
- Priorités fixes ou dynamiques
- Temps réel

Le noyau fourni permet de spécifier un ordonnanceur au démarrage :

```
sched_set_scheduler(&sched_fifo);           // tourniquet
sched_set_scheduler(&sched_aleatoire);      // aléatoire
sched_set_scheduler(&sched_prio_tempsreel); // priorité absolue
sched_set_scheduler(&sched_prio_dynamique); // prio dynamique - TP3
```

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 la 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 exemple

```
int clef_msg; /* globale, partagée par tous les processus */

void foo(void) {
    printf("%s\n", (char *)TSD_get(clef_msg));
}
void code(void *argument) {
    TSD_set(clef_msg, argument);
    for (int i = 0; i < 10; i++) {
        foo();
        proc_commuter();
    }
}
int main() {
    clef_msg = TSD_creer_clef(NULL);
    TSD_set(clef_msg, "je suis main");
    foo();
    proc_activer("P1", code, "je suis p1");
    proc_activer("P2", code, "je suis p2");
    proc_suspendre();
    return 0;
}
```

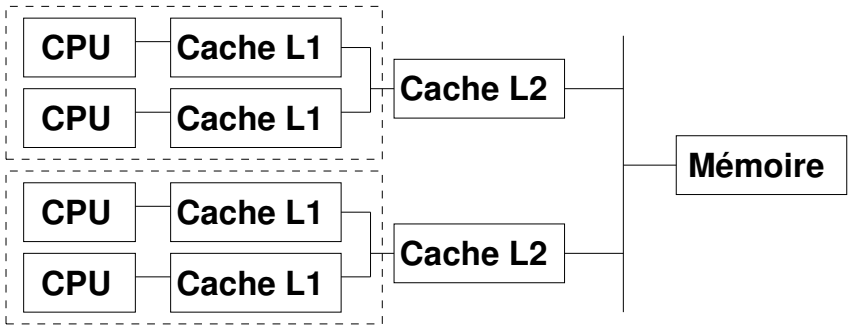
TSD Implantation

Ajouter dans le descripteur de processus un tableau de valeurs :

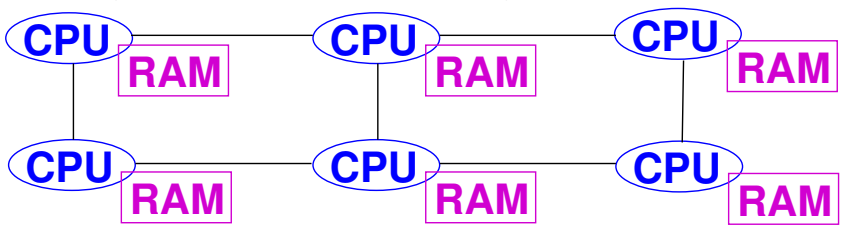
```
struct processus {
    ...
    void *tsd[TSD_MAX_CLEFS];
}

int TSD_créer_clef (...) {  /* renvoie une nouvelle clef */
    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]; }

void *TSD_set (int clef, void *val) {
    void *old = élu->tsd[clef];
    élu->tsd[clef] = val;
    return old;
}
```

Multi-processeurs

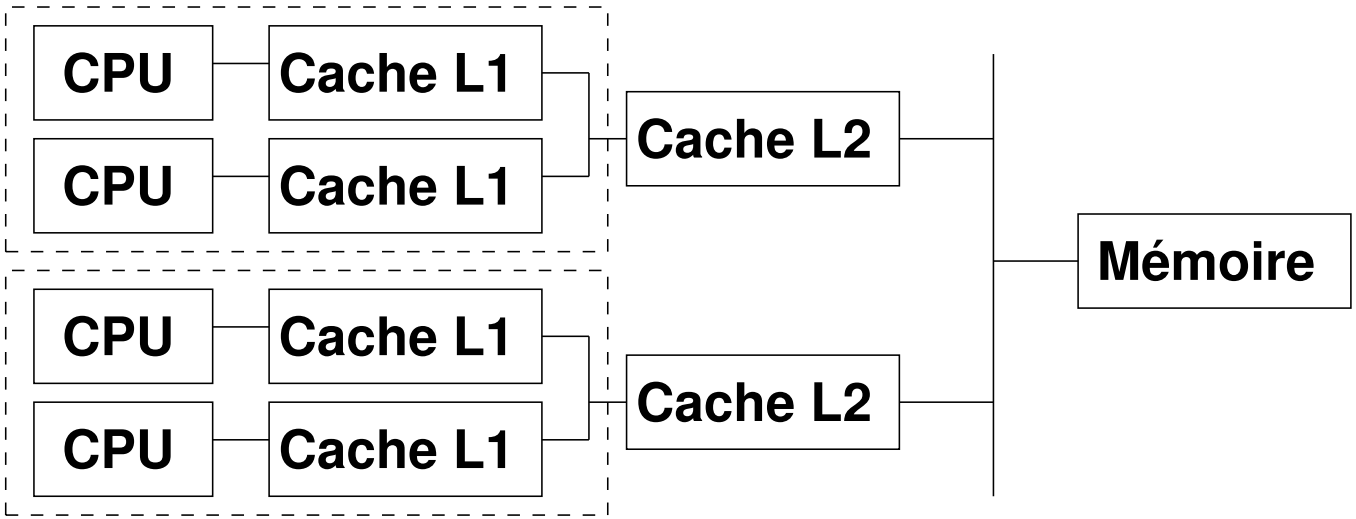
SMP : Symetric Multi-Processor : Plusieurs processeurs, tous identiques (multi-cœurs / multi-processeurs) + une seule mémoire commune + primitives de cohérence de caches.



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



SMP Symetric Multi-Processor



- Processeurs tous identiques ⇒ 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

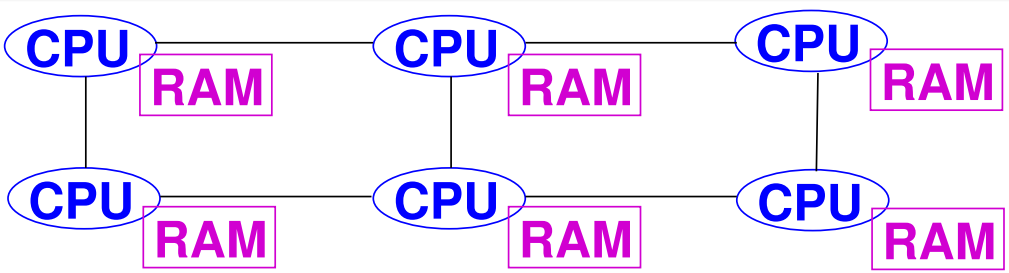
- `getCPU` = identifiant (numéro) du processeur appelant cette opération.
- `élu` ⇒ `élus[nbCPU]`
- Seule difficulté : `élus` et la file des prêts doivent être manipulés avec soin : en exclusion mutuelle ⇒ utilisation d'un verrou avec scrutation (spin lock) vu qu'un processus ne reste pas longtemps dans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



Résumé

- Processus : introduction du (pseudo-)parallélisme
- Processus élémentaires : utilisation contrôlée des coroutines, manipulation explicite des processus
- Préemption : commutation forcée
- Ordonnancement : politiques à court terme
- Multi-processeur
- Et après ? Processus synchronisés : cacher la manipulation explicite des processus au profit d'objets de synchronisation (2^{e} année, systèmes concurrents)

Processus synchronisés – exemple

Processus communiquant avec des messages

- Une file où les processus peuvent déposer ou extraire un message.
- Le retrait doit bloquer le processus si la file est vide.
- Déblocage ?

(version avancée : capacité borné, création dynamique de canaux, opérations non bloquantes, etc)

Processus synchronisés – algorithmes

Variables partagées

- msgs : une file de messages non lus
- readers : une liste de consommateurs en attente (file vide)

void put(m)

1. ajouter m à msgs
2. si readers est non vide, extraire un processus et le continuer

message take()

1. tant que msgs est vide, ajouter `proc_self` à readers et se suspendre
2. extraire un message de la file

(pourquoi “tant que” et pas “si” ? Voir en TP)