Synchronisation fine

— primitives noyau pour la concurrence —

Vocabulaire

Noyau (*kernel*) = logiciel commun à tout processeur, il est le coeur du module logiciel. Parmi d'autres tâches, il doit simuler un processeur virtuel à chaque processus.

Pour implémenter **parbegin** dans un noyau, il faut fournir :

- un mécanisme pour créer et lancer des processus,
- un mécanisme pour arrêter un processus,
- un mécanisme pour assurer la terminaison synchrone des processus **parend**.

D'où les primitives :

- **fork**(P(e1,...,en)) : création et lancément du processus
- **quit**() : arrêt du processus appelant
- synchronisation sur la terminaison par attente active :

```
bool done[N] = ([N] false); /* var globale */
...
for(i=0; i < N; i++) fork(P(i));
for(i=0; i < N; i++) while (!done[i]) ;
...
/* P(i) */ ... ; done[i] = true; quit()
```

Noyau mono-processeur

Composantes

- données pour représenter les processus
- interruptions et leurs traitement (*handler*)
- primitives noyau
- ordonnaceur (*dispatcher*)

Implémentation monolithique = chaque primitive noyau s'exécute comme une action atomique.

Comportement du noyau

1. arrivée d'interruption
2. sauvegarde de l'état du processus exécuté
3. entrée dans le *handler* de l'interruption
   (a) inhibe autres interruptions
   (b) traitement atomique
   (c) appel *dispatcher*
4. *dispatcher* selectionne un processus et charge son état
5. *dispatcher* active interruptions

Appel d'une primitive du noyau = levée de l'exception SVC (*SuperVisor Call*) avec les arguments passés en registres

Équité du *dispatcher*

- utilisation d'une queue FIFO pour les processus à exécuter
- utilisation d'un dispositif *timer*(d) qui déclanche l'exception *Timer* après d unités de temps.

Variables globales

```
proc_type proc_descriptors[N];
          /* processus s'exécutant */
int exec_proc;
          /* processus en attente d'exécution */
list_of_int ready_proc;
          /* descripteurs non utilisés */
list_of_int free_proc;
```

Traitement SVC

```
SVC_Handler: /* à son entrée, interr. inhibées */
   sauvegarde état d'exec_proc
   détermine la primitive à appeler
   appel primitive
```

Traitement Timer

```
Timer_Handler: /* à son entrée, interr. inhibées *
   ready_proc.push_end(exec_proc)
   exec_proc = 0
   dispatcher()
```

Primitive *dispatcher*

```
dispatcher () {
   if (exec_proc == 0) {
      exec_proc = ready_proc.top_and_pop();
   }
   lancement du timer
   chargement état exec_proc
       /* interr. activées automatiquement*/
}
```


Primitive fork

```
fork(état initial processus) {
  dscp = free_proc.top_and_pop()
  init dscp avec état processus
  insertion dscp dans ready_proc
  dispatcher()
}
```

Primitive quit

```
quit() {
  free_proc.push(descripteur d'exec_proc)
  exec_proc = 0;
  dispatcher()
}
```

État processeur libre = exécution d'un processus vide

Noyau multi-processeur

Extension du cas mono-processuer :

- primitives du noyau dans la mémoire partagée
- accès aux données noyau en exclusion mutuelle
- *dispatcher* doit gérer plusieurs exec_proc

avec les hypothèses suivantes :

- une interruption est traitée par le processeur où elle a été levée
- chaque processeur a un *timer*
- chaque processeur peut exécuter toute primitive noyau et tout processus

Exclusion mutuelle pour les données noyau critiques : ready_proc et free_proc.

Principes :

- sections critiques de courte étendue
- sections critiques séparées pour chaque donnée critique

Variables globales

```
proc_type proc_descriptors[N];
int exec_proc[P]; /* 1 / processeur */
list_of_int ready_proc;
/* variables pour sec. crit. ready_proc */
list_of_int free_proc;
/* variables pour sec. crit. free_proc */
```

Traitement SVC sur processeur i

```
SVC_Handler: /* à son entrée, interr. inhibées */
   sauvegarde état d'exec_proc[i]
   détermine la primitive à appeler
   appel primitive
```

Traitement Timer sur processeur i

```
Timer_Handler: /* à son entrée, interr. inhibées *
   sec. crit. ready_proc
      ready_proc.push_end(exec_proc[i])
   fin sec. crit. ready_proc
   exec_proc = 0
   dispatcher()
```

Primitive fork sur processeur i

```
fork(état initial processus) {
  sec. crit. free_proc
     dscp = free_proc.top_and_pop()
  fin sec. crit. free_proc
  init dscp avec état processus
  sec. crit. ready_proc
     insertion dscp dans ready_proc
  fin sec. crit. ready_proc
  dispatcher()
}
```

Primitive quit sur processeur i

```
quit() {
  sec. crit. free_proc
     free_proc.push(descripteur d'exec_proc[i])
  fin sec. crit. free_proc
  exec_proc[i] = 0;
  dispatcher()
}
```


État processeur i libre = exécution d'un processus spécial Idle

```
Idle:
  while (exec_proc[i] = Idle) {
    while (ready_proc.is_empty()) sleep(...);
    sec. crit. ready_proc
    if (! ready_proc.is_empty()) {
      exec_proc[i] = ready_proc.top_and_pop();
    }
    fin sec. crit. ready_proc
  }

  lancement timer i

  chargement etat exec_proc[i]
      /* interr. activées automatiquement*/
```

Primitive *dispatcher* sur processeur i

```
dispatcher () {
   if (exec_proc[i] == 0) {
     sec. crit. ready_proc
       if (! ready_proc.is_empty())
         exec_proc[i] = ready_proc.top_and_pop();
       else
         exec_proc[i] = Idle
     fin sec. crit. ready_proc
   }

   if (exec_proc[i] != Idle)
        lancement du timer i

   chargement état exec_proc[i]
       /* interr. activées automatiquement*/
}
```