

Ecriture d'un noyau

Abstractions proposées par Modula-2

Objectifs

- identifier les mécanismes de bas niveau permettant l'écriture d'un noyau;
- illustrer ces mécanismes.

Ces mécanismes ont été définis par N. Wirth dans le langage Modula-2.

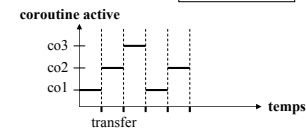
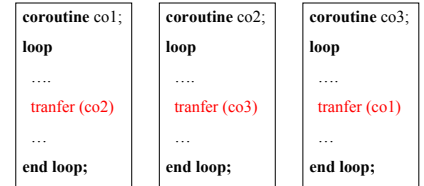
Transfert

- schéma d'exécution quasi-parallèle (coroutines);
- primitive pour passer le contrôle: **transfer (...)**.

Sémantique de **transfer(co2)** :

- la coroutine active (disons **co1**) qui exécute l'instruction **transfer(co2)** cesse de s'exécuter;
- la coroutine désignée par **co2** reprend son exécution;
- lors d'une exécution ultérieure de **transfer(co1)**, la coroutine **co1** reprend son exécution à l'instruction qui suit immédiatement l'appel à **transfer(co2)**.

Illustration:



- **transfer** est un mécanisme de bas niveau;
- **transfer** devrait être caché à l'intérieur d'un noyau (et n'être utilisé que dans le noyau):

Noyau
- threadsPrêts (liste de threads)
- proc créerThread // utilise transfer
- proc P // utilise transfer
- proc V // utilise transfer

Application (pour les exercices):

Classe Java **Coroutine**:

- constructeur: **Coroutine** (java.lang.String **className**)
 - crée une coroutine (= thread Java);
 - le code de la coroutine est la méthode **run()** de la classe **className** ;
- méthode **transfer** (Coroutine **co**)
 - stoppe la coroutine (= thread Java) appelante et reprend l'exécution de la coroutine passée en paramètre;
- méthode **endSystem** ()
 - arrête toutes les coroutines (= threads Java) et termine la JVM.

Utilisation de **transfer** dans la primitive **P**

```

proc P ();
....
  if n < 0 then
    insérer l'ident du thread courant en queue de la liste
    d'attente du sémaphore ;
    threadSuivant ← ident du thread en tête de threadsPrêts ;
    transfer (threadSuivant) ;
  end if ;
end P ;

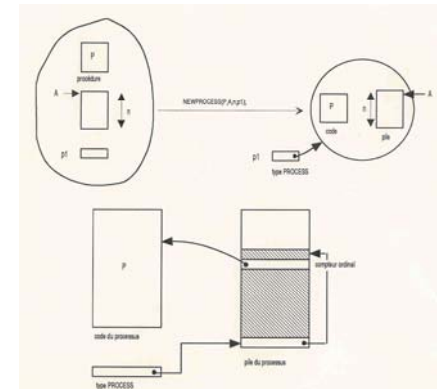
```

Classes Java à réaliser (en utilisant la classe **Coroutine**)

- class **NoyauSemaphore**, qui contient
 - **threadsPrêts**: liste de threads prêts ;
 - **createThread**: méthode permettant la création de threads en utilisant le constructeur de la classe **Coroutine** (les threads créés sont insérés dans la liste **threadsPrêts**) ;
 - **demarrerSysteme**: méthode appelée après la création de tous les threads. Passe le contrôle au thread en tête de la liste **threadsPrêts** ;
- class **Semaphore**, qui contient
 - les attributs **n**, **enAttente** ;
 - la méthode **P** ayant accès à la liste **threadsPrêts** ;
 - la méthode **V** ayant accès à la liste **threadsPrêts** .

Remarques concernant Modula-2

- Modula-2 est basé sur un schéma d'exécution **quasi-parallèle** (identique au schéma mis en œuvre par la classe **Coroutine** ci-dessus);
- dans Modula-2 un processus (= coroutine) est créé grâce à la procédure **NEWPROCESS** (à comparer au constructeur de la classe Coroutine);



Procédure NEWPROCESS de Modula-2

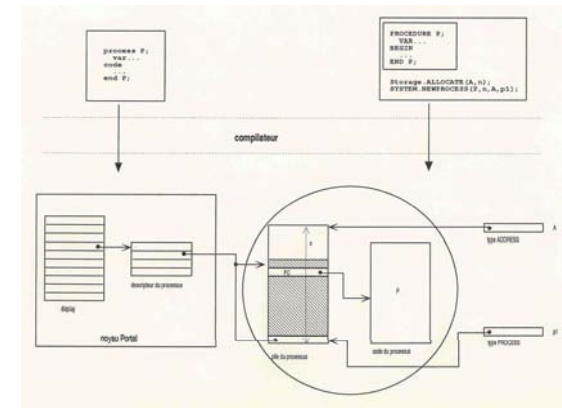
```

DEFINITION MODULE SYSTEM;
EXPORT QUALIFIED ADDRESS, PROCESS,
  NEWPROCESS, TRANSFER, ...;
...
PROCEDURE NEWPROCESS (P: PROC;
  A: ADDRESS;
  n: CARDINAL;
  VAR p1: PROCESS);
...
END SYSTEM.

```

paramètres de la procédure **NEWPROCESS**

- P**: procédure sans paramètre (de type **PROC**) qui constitue le code du processus
 - le *programme se termine* dès que l'exécution arrive à **end P**
- A** et **n** définissent la zone qui sert de pile au processus
 - taille de la pile déterminée par le programmeur
 - zone allouée au préalable, p.ex. en employant la procédure **ALLOCATE** du module **Storage**
- p1**: désigne le processus après sa création
 - processus ne démarre pas immédiatement (schéma d'exécution quasi-parallèle)
 - paramètre indispensable pour faire démarrer ultérieurement la processus



Procédure TRANSFER de Modula-2

- Modula-2 définit une procédure **TRANSFER(p1, p2)** avec **deux** paramètres:
 - le paramètre **p1** désigne le processus suspendu (pour une reprise ultérieure de son exécution à l'instruction suivant l'appel à la procédure TRANSFER) ;
 - les deux paramètres sont nécessaires parce que dans Modula-2 la désignation d'un processus n'est pas invariante durant toute l'exécution du processus;
 - cette solution permet une implémentation plus efficace, mais constitue une erreur de conception (rend l'utilisation de TRANSFER plus complexe) .

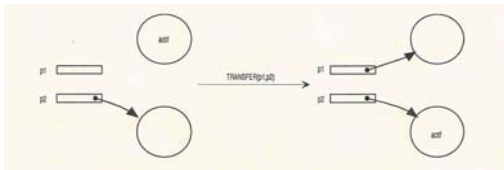
ioTransfer

- un schéma d'exécution purement quasi-parallèle est insuffisant car il ne permet pas la **prise en compte des interruptions**;
- il faut donc ajouter une dose de pseudo-parallélisme au schéma d'exécution quasi-parallèle défini par **transfer** ;
- ceci est réalisé à l'aide de **ioTransfer** .

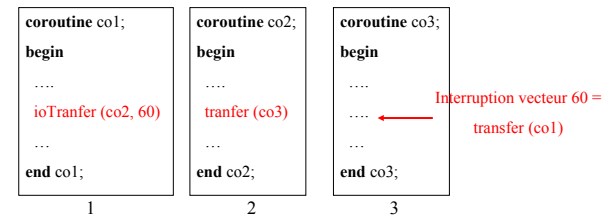
Sémantique de **ioTransfer(co2, vectInt)** :

- la coroutine active (disons **co1**) qui exécute **ioTransfer()** cesse de s'exécuter ;
- la coroutine désignée par **co2** reprend son exécution;
- lors d'une interruption à travers le vecteur **vectInt**, exécution de **transfer(co1)**.

Illustration de TRANSFER



Illustration



Remarque: du point de vue de vue de co3, le schéma d'exécution est **pseudo-parallèle** !

- **ioTransfer** est un mécanisme de bas niveau;
- **ioTransfer** doit être caché à l'intérieur d'un noyau (et n'être utilisé que dans le noyau).

Noyau

```
- threadsPrêts (liste de threads)

- proc créerThread    // utilise transfer
- proc P              // utilise transfer
- proc V              // utilise transfer
- proc attendreInterr (vectInterr) // utilise ioTransfer (., vectInterr)
```

- **ioTransfer** peut aussi être utilisé pour faire du timeslicing (changer de processus actif à chaque interruption de l'horloge);
- dans ce cas le paramètre **vectInterr** doit être le vecteur d'interruption de l'horloge.

Utilisation de **attendreInterr** par un processus
(exemple: lecture au clavier en mode interruption):

```
process clavier ;
loop
....
if interface pas prête (c-à-d aucun car disponible) then
    autoriser les interruptions du clavier ;
    attendreInterr (vecteurClavier);
    masquer les interruptions du clavier
end if ;
lire car dans le registre d'interface du clavier ;
end clavier ;
```

Remarques

- la JVM ne permet pas l'accès aux interruptions;
- cela rend impossible d'ajouter une méthode **ioTransfer** à la classe **Coroutine** (slide #7) tout en conservant la sémantique décrite ci-dessus;
- solution de substitution pour la sémantique de **ioTransfer(co2, ...)** :
 - la coroutine active (disons **co1**) qui exécute **ioTransfer()** cesse de s'exécuter ;
 - la coroutine désignée par **co2** reprend son exécution ;
 - lorsque <RETURN> est entré au clavier, exécution de **transfer(co1)****NB** en terme d'implémentation, **ioTransfer** contient un appel à **readLine** (lecture clavier).

Extensions de la classe Java **Coroutine** pour les exercices (cf slide #7):

- méthode **ioTransfer** (Coroutine **co**, int **interrupt**)
 - le 2^{ème} paramètre doit être **Coroutine.keybIO**
 - stoppe la coroutine (= thread Java) appelante jusqu'à ce que <RETURN> soit entré au clavier; reprend l'exécution de la coroutine passée en paramètre
- méthode **masquerInterruption** ()
 - empêche les commutations liées à **ioTransfer**
- méthode **retablirInterruption** ()
 - permet à nouveau les commutations liées à **ioTransfer**

Exercices (en utilisant **Coroutine**):

- adapter les classes **NoyauSemaphore** et **Semaphore** pour y ajouter du timeslicing (l'interface de ces classes reste donc inchangée).

Remarque concernant Modula-2:

- Modula-2 définit une procédure **IOTRANSFER(p1,p2,vecteur)** avec **trois** paramètres (pour la même raison que TRANSFER a été défini avec deux paramètres).