

Applications Concurrentes :

Conception,

Outils de Validation

ACCOV_B

Chapitre 1

INTRODUCTION ET EXEMPLES

Applications concurrentes

Représentation des processus

Gestion des processus

CONCEPTS DE LA PROGRAMMATION

Notion de sous-programme

Notion de module, d'objet, de paquetage

Programmation des processus concurrents

en Posix, Java et Ada

Exemples.

1.1. APPLICATIONS COMPORTANT DES PROCESSUS CONCURRENENTS

MOTIVATIONS POUR LES PROCESSUS CONCURRENENTS

- **améliorer l'utilisation de la machine (cas monoprocesseur)**
 - **multiprogrammation sur "main frame"**
 - **parallélisme calcul et E.S. sur micro personnel**
(lecture disquette, impression, accès réseau)
- **accélérer les calculs en utilisant une architecture parallèle et lancer concurremment des sous-calculs**
- **utiliser des architectures multiprocesseur à mémoire commune, des "clusters" ou des architectures réparties**
- **créer des systèmes réactifs pour répondre à des requêtes ou à des événements issus d'une application intrinsèquement concurrente ou répartie**
- **utiliser les ressources de calcul et les composants disponibles à distance sur internet et le web**
- **gérer du travail coopératif et des services répartis**
- **gérer des utilisateurs mobiles**
- **construire des applications parallèles, concurrentes ou réparties**

EXEMPLES D'APPLICATION CONCURRENTES

- **Centre de calcul ou centre de ressources accessibles aux filiales ou aux agences, par télétraitement ou via un réseau**
- **Station de travail avec processeurs parallèles (CAO de circuits, traitement d'images, Systèmes d'Information Géographique, serveur de Bases de Données pour transactions)**
- **Systèmes de réservation de places (hôtels, avions, train, concerts)**
- **SGBD avec accès multiple par des guichets ou des GAB (banque, PTT, assurances, mutuelles)**
- **Systèmes embarqués de conduite de mobiles (avion, voiture, train)**
- **Systèmes de conduite de réseaux d'infrastructure (contrôle de la production et du transport EDF de courant à 400 000 volts, gestion de réseaux d'oléoducs, de chauffage urbain, de distribution d'eau, de distribution d'électricité régionale - 225 000 volts -, de transport ferroviaire)**
- **Centraux téléphoniques interconnectés, avec valeur ajoutée**
- **Plates-formes ("middleware") : CORBA, Peer to Peer (JXTA de Sun)**
- **Structures d'accueil pour des objets répartis sur le Web (SOAP)**
- **Work Flow Management (WFMC)**
- **Diffusion Multimedia ("multicast"), synchronisation voix et image**
- **"mobile distributed systems and mobility management"**
- **Services coopératifs : annuaires, miroirs, caches Web**

NIVEAU D'INTERVENTION DE LA PROGRAMMATION CONCURRENTTE

Une application se construit en dégageant des niveaux

1) CAHIER DES CHARGES + ANALYSE FONCTIONNELLE

=> organisation fonctionnelle

(QUE FAIRE)

2) ANALYSE OPERATIONNELLE +

CHOIX DES UNITES LOGIQUES

=> architecture logique

(COMMENT FAIRE)

3) ANALYSE ORGANIQUE + CHOIX TECHNOLOGIQUES

=> architecture physique

(AVEC QUELS MOYENS)

architecture matérielle + carte de répartition du logiciel (placement)

PROCESSUS CONCURRENTS

Les processus concurrents sont les unités qui servent à représenter des activités concurrentes de l'architecture logique (au niveau de l'expression opérationnelle de l'application)

**LE COMPORTEMENT OPERATIONNEL
DE L'APPLICATION CONCURRENTTE
EST EXPRIME
PAR LE COMPORTEMENT DES PROCESSUS**

UNE ETUDE DE CAS : LE SNC D'EDF

ystème national de conduite informatique (SNC) pour le dispatching du centre national d'exploitation du système (CNES) de production et de transport d'énergie électrique à 400 000 volts

L'analyse opérationnelle a donné un découpage en acteurs SCADA (Système de conduite et d'acquisition de données actives). Chaque acteur, dévolu à une fonction applicative importante, peut être installé sur une machine séparée. Les machines sont reliées par un réseau local. Les acteurs ne communiquent entre eux que par messages.

Les éléments d'un acteur se partagent une mémoire commune qui contient une base de données temps réel et une messagerie. Un acteur est structuré en une plate-forme et en processus clients appelés transactions applicatives. Les transactions utilisent les services de la plate-forme par des appels à des API ("application programming interface", "apport de primitives d'interface")

La plate-forme est une structure d'accueil qui comprend :

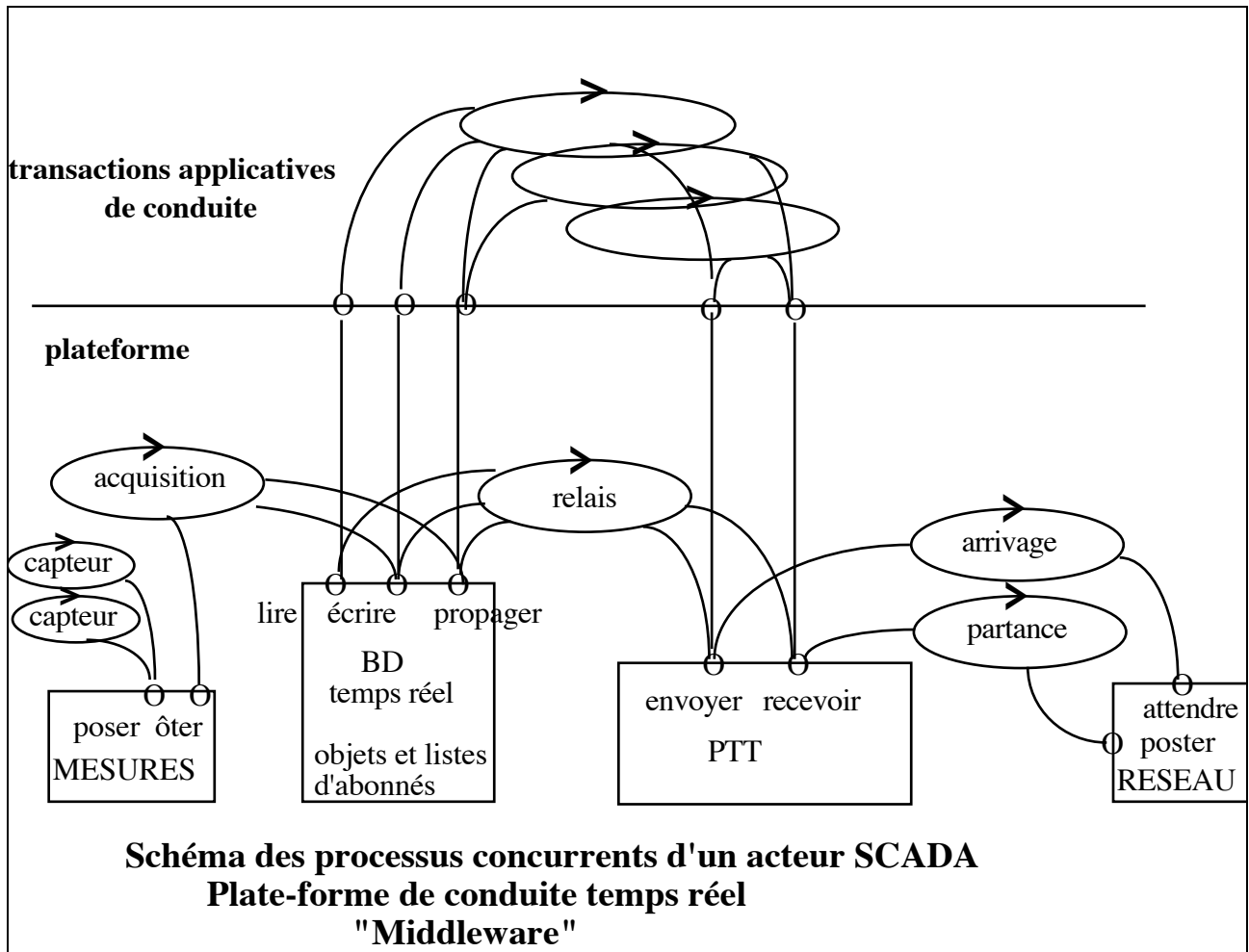
- une base de données temps réel qui contient les objets de la fonction (données mesurées par les capteurs, résultats des traitements réalisés par les transactions), et avec chaque objet la liste des processus abonnés à cet objet. L'accès à cette base de données appelée BD se fait par les API : BD.lire, BD.écrire, BD.propager.
- un service de messagerie, appelé PTT, qui fournit les API suivantes : PTT.envoyer, PTT.recevoir.
- un module RESEAU qui assure l'accès au réseau local entre sites.
- un module MESURES qui stocke les mesures en cours d'acquisition.
- des processus de service, comme :
 - les processus capteurs qui font la lecture des capteurs externes,
 - le processus acquisition qui les transfère dans la BD temps réel,
 - les processus partance et arrivage qui assurent la communication entre la messagerie PTT et les autres agents de l'architecture,
 - le processus relais qui est serveur d'accès à la BD pour les processus distants situés sur d'autres acteurs. Ces requêtes arrivent par PTT.recevoir et les réponses sont fournies par PTT.envoyer.

L'accès à la BD d'un autre acteur se fait via la messagerie, par une requête au processus relais de l'acteur distant et par lecture de sa réponse.

Les processus transactions applicatives utilisent la BD locale et le service de messagerie avec toutes les API disponibles.

La figure 1 donne un schéma d'un acteur avec des processus (ovales fléchés), des modules-paquetages (rectangles décorés avec les API d'accès) et des appels des API (arcs entre processus et modules).

FIGURE POUR L'ETUDE DE CAS



ARCHITECTURES LOGIQUES POUR LA CONCURRENCE

ENTITES DE BASE

- **objets actifs : processus**
- **objets passifs : ressources**
- **communication : messages ou partage de ressources**
- **regroupement de processus et de ressources**
(explicite ou par définition récursive des processus)

EXEMPLE CHORUS / JALUNA

- **activité (ou "thread") : processus (processeur logique)**
- **régions de mémoire virtuelle : ressource mémoire**
- **acteur : regroupement logique d'activités, de portes et d'ensembles de régions (mémoire virtuelle)**
- **porte : entrée ou sortie de message entre acteurs**
- **message : communication synchrone ou asynchrone entre acteurs**
- **communication entre processus :**
 - dans le même acteur, par mémoire commune
 - de deux acteurs différents : par messages

EXEMPLE ADA 95

- **unités de programmation = entités logiques**
 - **sous-programme : procédure ou fonction**
 - **paquetage : collection d'entités logiques**
 - **tâches : processus**
 - **objet protégé : objet partageable en exclusion mutuelle**
- **orthogonalité : une entité peut contenir n'importe quelle autre**
(définition récursive des entités)
- **communication par rendez-vous ou par objet protégé**

EXEMPLE JAVA

- **unités de base : classes, objets et méthodes**
- **objets actifs "threads" héritant de la classe "thread"**
- **objets distants héritant de la classe "applet"**

CRITERES POUR DECOUPER UNE APPLICATION EN PROCESSUS CONCURRENTS

a) parallélisme physique présent dans l'application ou dans l'architecture support

monoprocasseur

- **un processus pour chaque organe d'entrée ou sortie : lecteur de cassette, imprimante, clavier, écran**
- **un ou plusieurs processus pour le traitement de base**

multiprocasseur ou système réparti

- **un processus au moins par processeur**
- **sites clients et sites serveurs ou gestionnaires d'objets**

ensemble de capteurs d'une fonction d'un système réactif

- **un processus par réaction**

b) parallélisme logique de l'application

- **activités avec différentes échéances ou importances.**
- **activités avec différents comportement :**
 - périodiques, sporadiques, réactives, cycliques.**
- **activités de natures différentes : calcul, acquisition, présentation.**
 - clients, serveurs ou gestionnaires d'objets**
- **regroupement des actions fonctionnellement proches pour limiter les interactions interprocessus.**
- **regroupement d'actions en relation causale (séquencement causal).**
- **actions périodiques avec des périodes différentes.**
- **actions logiquement indépendantes avec peu d'interactions.**
- **activités distantes, réparties ou mobiles**

PROGRAMMATION ET ORDONNANCEMENT DES ACTIONS

Dans la spécification fonctionnelle d'une application, il apparaît :

- des actions ordonnées par causalité ou précédence**
- des actions non ordonnées car indépendantes**

Toutes les actions ne sont pas classées (ordonnées). La spécification impose seulement un ordre partiel.

A) Solution par programmation séquentielle impérative

On détermine dès la programmation l'ordre total de l'exécution

B) Solution séquentielle impérative avec non-déterminisme

On laisse des choix pour l'exécution

C) Programmation séquentielle déclarative

On déclare les contraintes de causalité ou de précédence

D) Programmation concurrente

On déclare des processus séquentiels (programmes séquentiels impératifs) et des relations de synchronisation entre eux (ordre partiel)

1.2. REPRESENTATION DES PROCESSUS

◆ **Processus :**

- séquence d'actions distinguée pour le besoin de l'analyse logique
- déroulement de cette séquence (processus = évolution) par exécution sur un processeur logique doté de ressources logiques
- représentation de ce processus (par une structure de données) dans le moteur d'exécution (le noyau du système)

◆ **Décomposition d'une application en processus**

- statique : le nombre des processus est fixe et connu dès le départ
- dynamique : création et destruction pendant le déroulement

◆ **Niveau et hiérarchie de processus**

- un seul niveau : tous égaux (ex. VM, VMS, concurrent Pascal)
- hiérarchie de processus avec emboîtement (ex. Unix, Ada)

relation père-fils et arborescence de la descendance

terminaison : le père doit attendre que tous ses fils aient terminé, ceci à cause de l'emboîtement et du partage de ressources (bas de la pile d'exécution, mémoire, canaux, fichiers, périphériques) : le père ne doit pas libérer des ressources encore utilisées par un fils.

Problème de détection et de synchronisation de terminaison.

◆ **Création de processus**

- par déclaration d'un modèle (type, classe) et instantiation d'un objet selon ce modèle (ex. Ada, Concurrent Pascal, Pascal FC, Modula, Java)
- par création d'une copie (clone) du père dans son état au moment de la création du fils (ex. Unix : fork)

REPRESENTATION DES PROCESSUS

(suite)

◆ Initialisation de processus

- passage de paramètres à la création, comme un appel de procédure
- pas de paramètre, donc acquisition explicite par le processus un fois créé (ex. Unix : exec pour code et donnée; Ada 83 pour données)

◆ Démarrage ou activation des processus

- à la création (les processus sont créés actifs, en Ada)
- par un ordre spécial (start process, cobegin...coend)

◆ Mode de comportement des processus

- séquentiel, cyclique
- client, serveur, agents intermédiaires
- périodique, apériodique, sporadique

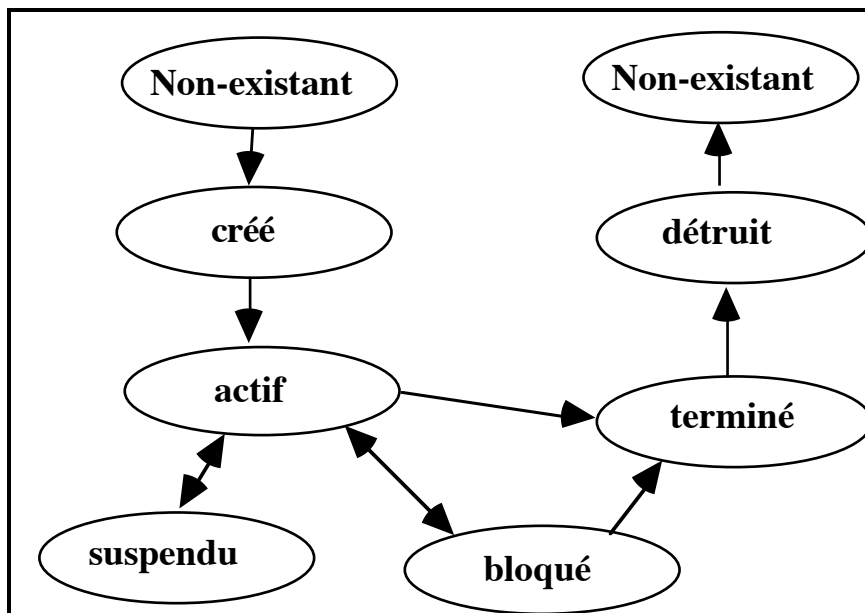
◆ Terminaison de processus

- fin correcte du code
- terminaison imposée par le noyau (erreur, dépassement de ressource)
- auto terminaison (suicide) sur exception
- terminaison forcée par un autre processus ("abort", "change mode")
- serveur cyclique devenu inutile (Ada : clause "terminate")
- jamais : processus cyclique permanent ("démon")
- jamais : boucle infinie par erreur

1.3. GESTION SYSTEME DES PROCESSUS

◆ Etats d'un processus

- ◆ **actif** : en évolution dans son processeur logique
- ◆ **bloqué** : évolution arrêtée dans l'attente d'un événement de synchronisation
- ◆ **prêt** : sous-état de actif indiquant que le processeur logique du processus n'est pas mappé sur l'un des processeurs physiques (aucun processeur n'est alloué à ce processus actif). Le processus, bien qu'actif et exécutable, n'a pas de processeur pour exécuter des instructions de son code (on dit qu'il avance à vitesse nulle)
- ◆ **élu** : sous-état de actif indiquant qu'un processeur physique est alloué au processus et que celui-ci exécute des instructions de son code
- ◆ **suspendu** : activité arrêtée pour un délai; le processus redevient actif au bout de ce délai



états logiques d'un processus

GESTION SYSTEME DES PROCESSUS

(suite)

◆ Descripteur de processus

◆ bloc de contrôle comportant :

- **identificateur repérant univoquement chaque processus**
- **état du processus**
- **environnement volatile (copie des registres du processeur, du compteur ordinal et du pointeur de pile, à la fin de la dernière élection)**
- **liens permanents de chaînage vers les descripteurs de ressources**

◆ utilisé par le moteur d'exécution

- **noyau exécutif ou micronoyau**
- **"run time kernel" (RTK)**
- **"run time support system" (RTSS)**

◆ **permet de gérer la "visite" d'un processeur par un processus, et de préparer la "visite" suivante ...**

◆ **permet de gérer le partage des ressources entre les processus et mettre en place les moyens d'exécution et de communication**

ORDONNANCEMENT DES PROCESSUS ACTIFS

◆ allocation du processeur

- sans réquisition possible (préemption) jusqu'à fin de l'état actif
- avec réquisition si :
 - activation d'un processus plus "prioritaire"
 - baisse dynamique de "priorité"
 - fin de quantum d'allocation

◆ quelques règles de gestion des processus prêts:

- ancienneté (FIFO)
- quantum de taille fixe ou variable
- priorités fixes ou dynamiques
- échéances
- partage aléatoire équitable (méthode de Ben Ari)
 - tirage au sort d'un des processus prêts
 - élection sans préemption pour une durée aléatoire
- activation, sans préemption, selon l'ordre textuel du programme

◆ cas des processus périodiques pour le temps réel

processus A période 20 durée d'exécution 4
 processus B période 40 durée d'exécution 10
 processus C période 80 durée d'exécution 40
 processus D aperiodique durée d'exécution 4

utilisation périodique du processeur $4/20 + 10/40 + 40/80 = 76/80$

règle "rate monotonic" :

$\text{priorité}(\text{processus}) = K/\text{période}(\text{processus})$
 $\text{priorité}(D) > \text{priorité}(A) > \text{priorité}(B) > \text{priorité}(C)$
 correct quelle que soit la date d'arrivée de D (une seule fois)
 (il y a à tout moment une laxité de 4)

◆ exemples d'ordonnancement avec "rate monotonic":

date	0	4	14	20	24	40	44	54	60	64	76	80
prêts	ABC	BC	C	AC	C	ABC	BC	C	AC	C	{}	ABC
élu	A	B	C1	A	C2	A	B	C3	A	C4		A
durée	4	10	6	4	16	4	10	6	4	12	4	4

date	0	4	14	20	24	28	40	44	54	60	64	80
prêts	ABC	BC	C	AC	CD	C	ABC	BC	C	AC	C	ABC
élu	A	B	C1	A	D	C2	A	B	C3	A	C4	A
durée	4	10	6	4	4	12	4	10	6	4	16	4

GESTION DE COLLECTIONS DE PROCESSUS

◆ groupes de processus

- appartenance à un groupe ou à plusieurs groupes
- création de groupe par un créateur ou un propriétaire de groupe
- entrée et sortie d'un groupe

◆ hiérarchies de processus

- père-fils (Unix, Ada) : les fils partagent les ressources du père, avec le père et entre eux
exemples de ressource partagée :
mémoire logique (ou virtuelle),
fichiers, canaux, périphériques,
objets protégés, ressources logicielles
- problème de terminaison :
 - le père doit attendre que tous ses fils aient terminé
 - simple si tous les processus sont séquentiels
 - cas plus complexe :
des fils sont séquentiels et clients
d'autres fils sont serveurs cycliques
terminaison complète (et destruction) quand :
 - tous les clients ont terminé leur exécution séquentielle,
 - tous les serveurs ont traité toutes les requêtes pendantes,
 - il n'y a plus de message en chemin vers aucun serveur.

clause "terminate" en Ada : signale chaque endroit du programme d'une tâche serveur où le noyau exécutif doit vérifier la condition de terminaison

GESTION DE COLLECTIONS DE PROCESSUS

◆ partage de mémoire pour le partage de variable

- "threads" d'un "process" Unix dans le système Mach
- activités d'un acteur dans le système Chorus•
 - communication par
 - mémoire entre "thread" d'un même "process" (acteur)
 - messages seulement entre "process"(acteurs)
- objets protégés de Ada
 - communication entre tâches Ada par rendez-vous (et messages)
 - avec invocation à distance

◆ partage de processeurs

- ferme de processus répartie sur plusieurs sites (stations de travail)
 - pouvant exécuter une même opération en concurrence
- hiérarchie à un niveau : 1 initiateur, des agents, pas de sous-agents
 - soit un nombre fixe d'agents dans la ferme
 - soit création dynamique d'agents selon le besoin
 - avec tolérance aux pannes et aux surcharges locales
- extension avec allocation dynamique du type d'opération à effectuer

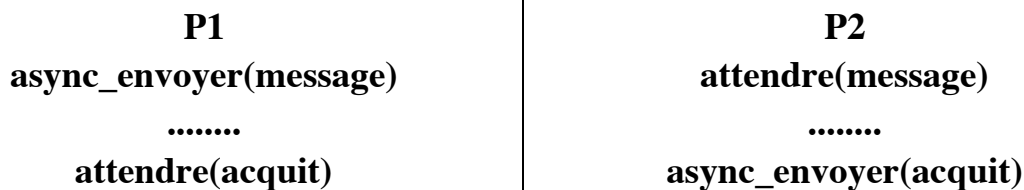
Voir les langages tels que PVM ou MPI ("message passing interface")

COMMUNICATION PAR MESSAGE

SYNCHRONISATION ELEMENTAIRE AU NIVEAU DES MESSAGES

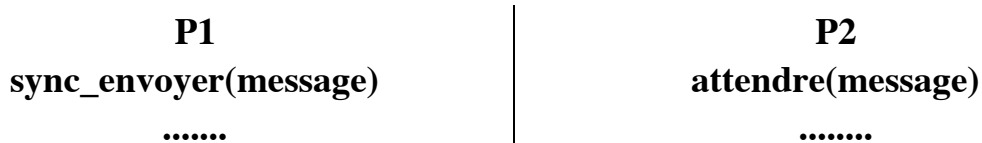
◆ Message asynchrone (sans attente)

- l'émetteur n'est pas bloqué par l'attente de la réception
- le récepteur a un rythme autonome de réception, avec deux modes
 - API de réception bloquante s'il n'y a pas de message
 - API de réception non bloquante, avec un témoin de réception
- schéma producteur-consommateur,
 - *avantage* : indépendance temporelle des correspondants
 - *inconvenients* : pas d'acquiescement implicite
 - pas de relation entre les états de l'émetteur et du récepteur
 - difficultés en cas d'erreur
 - *exemples* : Plits (Feldman, 1979), Conic (Kramer, 1983), RC 4000 (Brinch Hansen, 1970), Accent (Rashid, 1981), etc... très employé



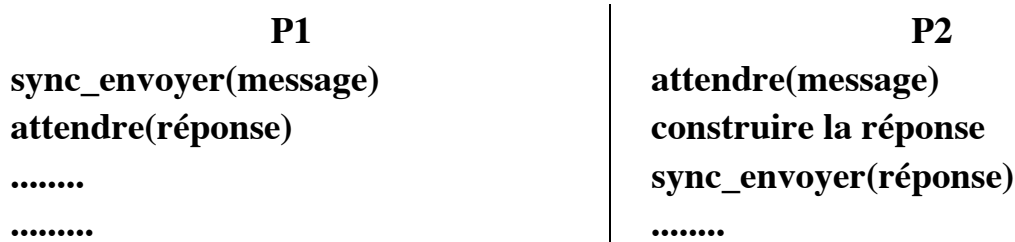
◆ Message synchrone (rendez-vous simple)

- l'émetteur attend que le récepteur ait lu le message
 - le récepteur qui attend un message est bloqué jusqu'à son arrivée
 - *avantages* :
 - émetteur et récepteur sont dans un état connu de leur évolution
 - on peut implanter des styles de calcul concurrents
 - par flot de données ou par calcul systolique
 - exemples* : CSP (Hoare, 1978), Occam
 - *inconvenients* : fort couplage entre les correspondants
- communication 1 - 1**



◆ **Invocation à distance (rendez-vous étendu)**

- demande l'exécution d'une procédure à un autre processus
- est accompagnée d'un passage de messages entre processus
- l'émetteur attend que le récepteur ait lu le message, qu'il ait traité la requête et qu'il ait donné une réponse (la réponse est facultative)
exemple : Ada 83, Ada 95, SR (Andrews, 1981), Conic (Kramer, 1983), V-kernel (Cheriton 1984)
- *avantage* : sémantique claire; correspondants dans des états connus
 communication N - 1 : paradigme N clients – 1 serveur
 à comparer à l'appel de procédure distante (APD)
 ou "Remote Procedure Call" (RPC)



◆ **Analogies**

- poster une lettre <<<<◇>>>> message asynchrone
- envoyer un fax <<<<◇>>>> message synchrone
- téléphoner <<<<◇>>>> invocation à distance avec réponse

◆ **appel de procédure à distance : APD ("remote procedure call" RPC)**

- la procédure (l'objet appelé) est localisée sur un autre site
- pour permettre l'exécution, on met en place une communication entre site du processus appelant et site de la procédure appelée; pour cela, on installe les ressources nécessaires à cette exécution sur le site de l'appelant et sur le site de l'appelée et on assure la gestion de la tolérance aux pannes de communication ou d'exécution.
- on est au niveau physique, pas au niveau logique (voir cours réseau : souche client, souche serveur, passage des arguments et des résultats, gestion de l'exécution sur le site serveur par un processus subrogé -tuteur, proxy, courtier-. Voir normes DCE, CORBA.

COMPARAISON

Invocation à distance <--> appel de procédure à distance

INVOCATION A DISTANCE

construction langage
niveau logique
appel d'un processus

appel exécuté par un autre processus, connu, le processus appelé, avec son environnement (dont sa pile)

Appel exécuté quand le processus appelé le permet; il peut y avoir des conditions de synchronisation; l'appel peut n'être jamais exécuté

les deux processus sont en *rendez-vous* dans le même programme

APPEL DE PROCEDURE DISTANTE

construction réseaux
niveau physique réparti
appel d'un site

appel exécuté par un processus subrogé ou courtier ("proxy") créé sur un autre site, le site appelé

Appel exécuté dès que possible (délai de mise en place des ressources) comme une procédure; en général, il n'y a pas de condition d'exécution

appel de procédure entre des espaces d'adressage différents, des programmes différents

si le processus appelé est sur un autre site, l'APD peut servir à installer l'invocation à distance entre deux sites.

Voir la solution retenue pour la répartition avec ADA.

DESIGNATION DES PROCESSUS COMMUNICANTS

◆ Désignation directe ou indirecte

- désignation directe : nom du processus correspondant

envoyer(message, nom_de_processus)

- désignation d'un relais intermédiaire :

nom d'un canal, d'une boîte aux lettres, d'une porte, d'un port

envoyer(message, nom_de_canal)

exemple : Occam 2 (INMOS 1984)

ch!e (* envoie la valeur de l'expression e sur le canal ch *)

ch?v (* reçoit par le canal ch une valeur qui est assignée à v *)

le premier processus qui arrive sur une action à un bout du canal attend jusqu'à l'arrivée de l'autre sur l'autre bout du canal.

Quand tous les deux sont présents, le rendez-vous a lieu et la valeur de l'expression e est passée à v.

On réalise, sans tampon intermédiaire, l'affectation distribuée

v := e

où v est dans un processus et e dans l'autre

◆ Appels symétriques ou dissymétriques

- symétrique :

l'émetteur et le récepteur se nomment directement ou indirectement

envoyer(message, nom_du_destinataire)

recevoir(message, nom_de_l'émetteur)

envoyer(message, nom_du_canal)

recevoir(message, nom_du_canal)

- asymétrique :

le récepteur ne nomme pas de source, mais accepte des messages de tout processus (ou canal, boîte aux lettres, porte, port)

attendre(message)

adapté au paradigme client -serveur

- Exemple : Ada. Le client doit nommer le serveur, mais le serveur n'a pas besoin de connaître le client, sauf s'il doit donner une réponse (en invocation à distance, la réponse est envoyée à la fin de la phase de rendez-vous et la désignation du client est implicite)

◆ Contrôle de la réception

réception contrôlée : attente si canal vide

réception sélective : le message peut venir d'un canal parmi N

1.4. PROGRAMMATION CONCURRENTE

◆ **L'exécution d'un programme concurrent est non déterministe, à cause de cet ordre partiel entre les processus.**

◆ **Le programme est défini entièrement avec un ordre partiel. Or l'exécution sur un monoprocesseur (ou l'observation) est séquentielle et induit un ordre total. Cet ordre peut-être différent d'une exécution à l'autre. Il doit seulement ne pas être en contradiction avec l'ordre partiel initial.**

◆ **D'où la difficulté à :**

- **comprendre le comportement du programme**
- **modéliser et évaluer ce comportement**
- **faire la mise au point**
- **obtenir une réexécution exacte du programme**

◆ **Difficulté supérieure avec multiprocesseur, avec application répartie car l'exécution avec plusieurs processeurs n'apporte pas un ordre total observable : il faut en déterminer un qui soit compatible avec la synchronisation ou les messages.**

(voir cours SAR_B, UE NFP111 ou SRR_C, UE NFP214)

CONCEPTS DE LA PROGRAMMATION AVANCEE

RAPPELS

APPLICATION A LA CONCURRENCE

◆ Notion de sous-programme (procédure et fonction)

- abstraction d'une instruction plus puissante
du calcul d'une expression

◆ notion de déclaration

avec nom du modèle et paramètres formels (signature),
variables locales internes au modèle

- éventuellement accès à des variables globales dans un bloc externe à la déclaration (liaison statique, à la déclaration)
- selon les langages, emboîtement à un ou plusieurs niveaux

◆ notion d'appel de sous-programme (instantiation)

utilisation du modèle avec :

- des paramètres effectifs

règles de passage paramètres effectifs \rightarrow paramètres formels

moment du passage (appel du sous-programme; par nécessité)

nature : copie de valeur, de référence, de résultat; par référence

- un jeu de variables de travail

◆ notion de retour de sous-programme

- retour de résultat

- purge de l'utilisation du modèle

casser la liaison paramètres effectifs \rightarrow formels

détruire les variables de travail

◆ gestion de mémoire associée à l'utilisation d'un sous-programme

- gestion en pile pour stocker pour chaque appel

les paramètres effectifs

les variables de travail

les liens de retour vers le sous-programme appelant

(adresse de retour dans le code et environnement de retour)

au retour du s.p., on dépile les éléments empilés à l'appel

◆ Notion de sous-programme (suite)

◆ sous-programme séquentiel

appels successifs ou emboîtés de s.p : une seule pile suffit

◆ sous-programme séquentiel avec récursivité

appels emboîtés du même s.p. : une seule pile suffit

récursivité croisée : une seule pile suffit

◆ sous-programme réentrant

accès concurrents possibles par plusieurs processus appelant si :

- code invariant : contrainte sur le jeu d'instruction du processeur, en particulier les appels de s.p. (BAL :branch and link) ne doivent pas ranger l'adresse de retour dans le code du s.p., mais dans une zone de travail de chaque processus (pile ou registres sauvegardés et commutés à chaque changement de processus élu)

- une zone de travail par processus : une pile par processus

◆ sous-programme partagé par plusieurs processus

plusieurs processus peuvent utiliser le modèle et se trouver à un instant t à exécuter des instructions différentes de ce modèle (éventuellement la même) sur des processeurs différents. Il y a plusieurs déroulements du modèle ("thread" ou fils de contrôle) et plusieurs lieux dans le modèle ("locus of control") où s'exécutent des instructions. Il faut un compteur ordinal pour chaque processus

◆◆ accès parallèle, concurrence sans contrôle

- pas de variable globale, ou les variables globales sont des constantes lues seulement, alors les exécutions sont indépendantes.
- écritures dans les variables globales. Si l'ensemble des écritures n'est pas indivisibles, on ne garantit aucune cohérence.

◆◆ accès parallèle, écritures gérées par exclusion mutuelle

sous-programmes protégés (appels concurrents séquentialisés)

nota : la récursivité de l'appel peut poser un problème si l'accès récursif est accompagné de pose de barrière d'exclusion mutuelle

◆ Notion de module, d'objet, de paquetage

• définition :

1. regroupement de données et des sous-programmes qui les manipulent
2. règles d'accès à ces regroupements : accès autorisé aux entités exportées explicitement par une déclaration d'export
(en bonne règle, pas d'accès aux données, mais seulement aux sous-programmes exportables)
(en programmation par objets ces s.p. sont appelés des méthodes)

• utilisation :

objet protégé : accès aux méthodes en exclusion mutuelle

objet concurrent : intérêt si les données sont en lecture seulement ou si le problème traité entraîne une cohérence des accès en écriture

• règles de visibilité : un module, c'est une frontière étanche

- les données du module sont des données globales pour tous les s.p. du module (on dit aussi données rémanentes). Leur durée de vie est plus grande que l'appel d'un sous-programme du module, c'est la durée de validité de la déclaration du module.
Ce sont des variables d'état conservables entre plusieurs appels de sous-programmes du module, ce sont des données persistantes.
- les données du module ne sont pas accessibles de l'extérieur du module (sauf si elles sont déclarées explicitement exportables)

DÉFINITIONS

PROCESSUS (Task en Ada, Thread en Java) : unité de concurrence

Un programme concurrent commence séquentiellement avec une tâche principale ("main") qui crée (déclare) les autres processus concurrents. Chacune des tâches déclarées se déroule en parallèle comme si il y avait autant de processeurs que de tâches.

L'APPEL DE PROCÉDURE

(sous-programme ADA : procédure ou fonction) (méthode Java)

- une procédure donnée peut-être appelée de plusieurs endroits du programme (au départ un programme est une procédure principale "main")
- un appel de procédure entraîne un passage d'information de l'appelant vers l'environnement de la procédure appelée (in) ou réciproquement (out) ou dans les deux sens (in out)
- une procédure contient une série d'instructions (le corps de procédure) qui sont exécutées pour le compte de l'appelant, jusqu'au retour de procédure accompagné du passage de résultat : on dit que la procédure appelante "attend" le retour de la procédure appelée
(même si appelante et appelée sont exécutées pour le même processus)

APPEL DE PROCÉDURE DANS UN PROGRAMME CONCURRENT

- une procédure peut être appelée par un ou plusieurs processus
mais ce n'est pas une structure de communication ou synchronisation
 - chaque appel crée un environnement indépendant (variables de travail et paramètres effectifs) qui ne communique pas avec les environnements des autres appels; les paramètres effectifs et les variables de travail sont rangés dans la pile du processus appelant et sont donc des données propres au processus (non partageables)
 - Le code d'une procédure partagée doit rester invariant
(on dit que la procédure doit être réentrante)

L'INVOCATION À DISTANCE EN ADA

- demande l'exécution d'une procédure à un autre processus
- est accompagnée d'un passage de messages entre processus
- comme pour l'appel de procédure, l'appelant "attend" la réponse de l'appelé, mais ici appelant et appelé ne sont pas le même processus
- son écriture ressemble à un appel de procédure
 - les procédures invocables à distance sont des entrées du processus
 - l'appel désigne le processus concerné et l'entrée appelée

L'APPEL D'UN OBJET PROTÉGÉ EN ADA

Utilisation d'un objet de synchronisation partagé entre processus

PROCESSUS CONCURRENTS EN C ET API POSIX

CLASSE PTHREAD__T

Un processus est une instance du type (opaque) `pthread_t`. Un programme crée et démarre un «thread» en appelant «`pthread_create`» avec l'adresse de son nom, une structure «attribut», la fonction qu'il exécutera, et les arguments de cette fonction.

Un processus se termine par un `return` ou en appelant «`pthread_exit`». Cela délivre un résultat à un processus qui attend par «`join`» et rend les ressources.

Un processus a un pointeur vers la mémoire de son créateur ; attention aux pointeurs perdus «`dangling reference`».

EXCLUSION MUTUELLE EN POSIX

Sémaphores de type mutex avec des fonctions de verrouillage/déverrouillage.

```
mutex pthread_mutex_t
pthread_mutex_lock(&mutex)           /*verrouillage
pthread_mutex_trylock(&mutex)        /* sans verrouillage
pthread_mutex_unlock(&mutex)         /* libère un processus bloqué
pthread_mutex_destroy(&mutex)        /* détruit le sémaphore
```

SYNCHRONISATION EN POSIX

Variables de type *condition* avec des événements instantanés et diffusés (moniteur). Utilisées avec un mutex.

```
pthread_cond_wait(&cond_vbl, &mutex) /* opération d'attente
pthread_cond_timedwait(&cond_vbl, &mutex, &timeout)
pthread_cond_signal(&cond_vbl)      /* envoi d'un signal pulsé
pthread_cond_broadcast(&cond_vbl)   /* diffusion d'un signal pulsé
pthread_cond_init(&cond_vbl)
pthread_cond_destroy(&cond_vbl)
```

Sémaphores de type à compteur

Passage de données à la création par «`pthread_create`» et fourniture de résultat à la fin d'exécution par `return` ou «`pthread_exit`» vers le processus attendant par «`join`».

Synchronisation compliquée avec les *signaux d'exception*.

PROCESSUS CONCURRENENTS EN JAVA

CLASSE THREAD

**Instance de classe dérivée de la classe Thread, lancée par la méthode start()
L'ordonnanceur lance la méthode run() qui déroule le processus**

```

class Exemple {
    public static void main(String args[]){           // programme principal
        new UnProcessus("Claude").start();           // lance une instance
        UnProcessus a =new UnProcessus ("Jean François");
        a.start(); // lance une instance
    }
}
class UnProcessus extends Thread {
    public UnProcessus(String st){                   // constructeur d'objet
        super(st);
    }
    public void run(){                               // programme du processus
        for (int i=0; i<10; i++){
            System.out.println(i + " " + getName() );
            try{sleep((int)(Math.random()*10);}
            catch(InterruptedException e){}
        }
        System.out.println(getName()+" a fini");
    }
}

```

On peut aussi utiliser l'interface Runnable. On doit le faire si l'objet hérite d'une classe car il n'y a pas d'héritage multiple en Java

```

class ObjetRunnable extends Applet implements Runnable{
    ...
}
ObjetRunnable k = new ObjetRunnable();           // création d'un objet
Thread t = new Thread(k);                       // création d'un processus
t.start();                                       // lancement du processus

```

PROCESSUS CONCURRENTS EN JAVA

MÉTHODES PUBLIQUES DE LA CLASSE Thread

```
final void suspend();  
final void resume();  
static native void yield();  
final native boolean isAlive();  
final void setName(String Nom);  
final String getName();  
public void run();
```

SYNCHRONISATION EN JAVA

La synchronisation entre thread utilise des méthodes de la classe Object

A l'intérieur d'un objet, un thread utilise :

```
wait(); // met le thread en attente et relache l'accès à l'objet  
notify(); // réveille un processus des qui attendent l'objet par wait()  
// le choix est arbitraire et n'est pas obligatoirement FIFO  
notifyAll(); // réveille tous les processus attendant l'objet par wait()  
// ils seront exécuté dans un ordre quelconque
```

Ces méthodes doivent être lancées dans des méthodes synchronized

EXCLUSION MUTUELLE EN JAVA

On utilise le mot clé réservé

synchronized

comme modifieur de méthode de classe ou de méthode d'instance ou de bloc;

Il garantit que cette méthode ou ce bloc est une section critique

(donc exécuté en exclusion mutuelle entre processus)

A chaque objet est associé un verrou. Un thread qui invoque une méthode synchronized doit obtenir le verrou avant d'exécuter la méthode

PROCESSUS CONCURRENTS EN ADA

SCHÉMA GÉNÉRAL D'UN PROGRAMME ADA CONCURRENT

procédure **COURS_ACCOV** is

 -- déclaration du programme principal

task type UN_TYPE_DE_PROCESSUS; -- déclare un type de processus

GLOBAL : ..; -- variables communes partagées entre les processus

task body UN_TYPE_DE_PROCESSUS is

 -- déclaration du programme du processus

LOCAL : ; --variables privées à chaque processus créé

begin

 -- code des processus du type UN_TYPE_DE_PROCESSUS

end UN_TYPE_DE_PROCESSUS;

procedure UNE_PROCEDURE(X : Integer ; Y : Boolean) is

LOCALES : ; -- variables locales à la procédure

begin

 -- code de la procédure qui a deux paramètres formels, X et Y

end UNE_PROCEDURE

protected Fin is

procedure Signaler ;

entry Attendre(I : Integer) ;

end Fin ;

protected body Fin is separate ;

-- le « begin » qui suit marque la fin de l'élaboration du programme

-- = fin de l'exécution déduite des déclarations

-- = fin de la mise en place des objets déclarés du programme

begin -- liste des instructions du programme principal

GENESE :

déclare

P, Q : UN_TYPE_DE_PROCESSUS;

 -- création-activation de processus du type

 -- la création met en place deux objets processus

begin -- on a maintenant 3 processus concurrents : P, Q et COURS_ACCOV

null;

end GENESE;

end COURS_ACCOV ;

EXEMPLE COMPARATIF

le processus principal père crée deux processus fils ; le père imprime puis, chaque fils lance une impression, enfin le père imprime une seconde fois.

----- SYNCHRONISATION PAR SÉMAPHORES -----

```
with SI_B; use SI_B; -- importation d'objet en bibliothèque et visibilité
with Ada.Text_Io; use Ada.Text_Io;
procedure Main is
```

```
package Fin is -- déclaration d'un objet de synchronisation, objet partagé
  procedure Signaler;
  procedure Attendre(I : in Integer);
end Fin;
```

```
package body Fin is -- implantation de l'objet paquetage
  procedure Signaler is
  begin V(S); end Signaler; -- V ou Signal ou pthread_mutex_unlock(&S)
  procedure Attendre(I : in Integer) is
  begin for J in 1..I loop P(S); end loop; end Attendre;
        -- P ou Wait ou pthread_mutex_lock(&S)
  begin E0(S,0); end Fin; -- initialisation du sémaphore : S pthread_mutex_t
```

```
-- déclaration du type des fils
task type Un_Fils (K : Integer) is
begin
  Put_Line("BONJOUR DU FILS " & Integer'Image(K));
  Fin.signaler; -- appel de l'objet de synchronisation
end Un_Fils;
```

```
Fils1 : Un_Fils(1); Fils2 : Un_Fils (2); -- déclaration et création des fils
```

```
begin -- programme du processus père
  Put_Line("BONJOUR DU PERE")
  Fin.Attendre(2); -- appel de l'objet de synchronisation pour être bloqué
  Put_Line("FIN DU PROGRAMME");
end Main;
```

EXEMPLE COMPARATIF-----
RAPPEL UNIX-LINUX-POSIX
-----**Les processus sont créés par l'opération Fork****#include <stdio.h>****main ()****{****printf ("BONJOUR DU PERE\n");****id = fork();****if (id == 0) {****/*processus fils 1*/****printf("BONJOUR DU FILS 1 %d/n");****exit(0);****}****if (id < 0) perror("erreur au fork()");****id = fork();****if (id == 0) {****/*processus fils 2*/****printf("BONJOUR DU FILS 2 %d/n");****exit(0);****}****if (id < 0) perror("erreur au fork()");****if (id > 0) {****/*le père* attend la fin des 2 fils*/****wait((int*)0);****wait((int*)0);****printf("FIN DU PROGRAMME \n");****exit(0);****}**

EXEMPLE COMPARATIF : SOLUTION POSIX

Code source en C avec les API POSIX gérant des Threads (primitives système)

```
#include <ipc.h>                /* module de communication
#include <pthread.h>          /* module de définition des "thread"
```

```
pthread_attr_t pthread_attr_default = 1;
```

```
void threadcode(arg)           /* fonction appelée par les « threads » fils
{
  printf("BONJOUR DU FILS %d/n", arg);
  pthread_exit (- arg);
}
```

```
void Main()                   /* programme principal
{
  pthread_t FILS1, FILS2;      /* déclaration des « threads » fils
  pthread_attr_t attr;
  int result;
  printf ("BONJOUR DU PERE\n");
  pthread_attr_create(&attr);
  attr.pthread_attr_prio = 6;
  pthread_attr_setstacksize(&attr, 8192);
```

```
if(pthread_create(&FILS1, attr, threadcode, 1) < 0 {
  printf(" erreur de création du FILS 1\n");
  exit(1);
}
```

```
if(pthread_create(&FILS2, attr, threadcode, 2) < 0 {
  printf(" erreur de création du FILS 2\n");
  exit(1);
}
```

```
pthread_join(FILS1, NULL);     /* attend la fin du fils1
pthread_join(FILS2, NULL);     /* attend la fin du fils2
printf("FIN DU PROGRAMME\n");
exit(0);
}
```

EXEMPLE COMPARATIF : SOLUTION JAVA

```
class Main {  
    public static void main(String args[]){  
        // programme principal  
        System.out.println("BONJOUR DU PERE");  
  
        // lance une instance comme premier fils  
        UnProcessus a =new UnProcessus ("FILS 1");  
        a.start();  
  
        // lance une instance comme second fils  
        UnProcessus b =new UnProcessus ("FILS 2");  
        b.start();  
  
        // attendre la fin des deux fils  
        try {  
            a.join();  
            b.join();  
        } catch(InterruptedException e) {  
            System.out.println("un fils ne repond pas");  
        }  
  
        System.out.println("FIN DU PROGRAMME");  
    }  
}
```

```
    // déclaration de la classe des fils  
classe UnProcessus extends Thread {  
    public UnProcessus(String st){  
        // mémorise un objet  
        super(st);  
    }  
    public void run(){  
        // programme du processus  
        System.out.println("BONJOUR DU" + getName() );  
    }  
}
```

EXEMPLE COMPARATIF : SOLUTION TÂCHES ADA

(programmation en Ada normalisé, en C/Posix, en C/WindowsNT)

Code source en ADA 95 avec rendez-vous:

```
with Ada.Text_Io; use Ada.Text_Io; -- paquetage standard d'impression
procedure MAIN is
  -- création du père, programme principal
```

```
  -- déclaration de l'interface du type fils
  task type TT is
    entry START( ID : INTEGER);
    -- interface de rendez-vous
  end TT;
```

```
  -- création de deux fils serveurs
  FILS1, FILS2 : TT;
```

```
  -- déclaration de l'implantation des fils
  task body TT is
  begin
    accept START(ID : INTEGER) do
      PUT_LINE ("BONJOUR DU FILS " & INTEGER'IMAGE(ID));
    end START;
  end TT;
```

```
begin
  -- à cet endroit, on a trois processus concurrents le père et 2 fils
  -- début du père et activation des fils
  PUT_LINE ("BONJOUR DU PERE");
  FILS1.START(1);      -- appel et réveil du fils1 avec une requête
  FILS2.START(2);      -- appel et réveil du fils2 avec une requête
  PUT_LINE ("FIN DU PROGRAMME");
end MAIN;
```

```
-- Ada est court, clair, portable sur Unix, Linux, Posix et Windows NT
-- le langage Ada est normalisé ISO/IEC 8652:1995(E)
-- structure analogue à l'appel de procédure distante
-- on verra par la suite l'utilisation des objets protégés en Ada
```

NÉCESSITÉ BASIQUE DE L'EXCLUSION MUTUELLE

PROGRAMME ADA AVEC TROIS PROCESSUS (main, PP et QQ)

procédure **LES_MAUX_DE_L_IMPRESSION** is -- main task

task type P; task type Q;

task body P is -- déclaration de corps de processus

begin

PUT ("J'ai vu la terre distribuée ");

PUT ("en de vastes espaces ");

NEW_LINE;

PUT ("et ma pensée ");

PUT ("n'est point distraite ");

PUT ("du navigateur.");

NEW_LINE;

PUT ("Saint-John Perse");

NEW_LINE (2);

end P;

task body Q is -- déclaration de corps de processus

begin

PUT ("L'homme est capable de faire ");

PUT ("ce qu'il est ");

PUT ("incapable d'imaginer. ");

NEW_LINE;

PUT ("Sa tête sillonne ");

PUT ("la galaxie de l'absurde");

NEW_LINE;

PUT ("René Char");

end Q;

begin -- début du programme principal "main"

PUT ("IMPRESSION DE DEUX POEMES"); -- un seul processus

NEW_LINE (2);

declare

PP : P; -- création d'un processus fils

QQ : Q; -- création d'un processus fils

begin -- activation des processus fils PP et QQ : 3 processus prêts

null;

end;

end LES_MAUX_DE_L_IMPRESSION;

PAS DE MÉCANISME D'EXCLUSION MUTUELLE

PROGRAMMATION ERRONÉE

procedure ACCES_BRUTAL is

COMPTE_CLIENT : Integer := 0; -- variable commune persistante

task P is

X : Integer := 0; -- variable locale à P

begin

for I in 1 .. 16 loop

actions_hors_section_critique;

X := COMPTE_CLIENT; -- P1

X := X + 1; -- P2

COMPTE_CLIENT := X; -- P3

end loop;

end P;

task Q is

Y : Integer := 0; -- locale à Q

begin

for I in 1 .. 16 loop

actions_hors_section_critique;

Y := COMPTE_CLIENT; -- Q1

Y := Y + 1; -- Q2

COMPTE_CLIENT := Y; -- Q3

end loop;

end Q;

-- fin de l'élaboration des objets déclarés par le programme principal

begin

-- maintenant sont actifs P et Q et le programme principal ACCES_BRUTAL

null;

end ACCES_BRUTAL;

Notant []^x une boucle de x cycles, et en notant CC pour COMPTE_CLIENT, on peut tracer quelques ordonnancements significatifs et résultats associés:

{CC = 0} [P1 P2 P3]¹⁶ [Q1 Q2 Q3]¹⁶	{CC = 32}
{CC = 0} [P1 P2 P3 Q1 Q2 Q3]¹⁶	{CC = 32}
{CC = 0} [P1 Q1 P2 P3 Q2 Q3]¹⁶	{CC = 16}
{CC = 0} P1[Q1 Q2 Q3]¹⁵ P2 P3 Q1[P1 P2 P3]¹⁵ Q2 Q3	{CC = 02}

EXCLUSION MUTUELLE MODULAIRE AVEC SÉMAPHORE

```

procedure Exclusion_Mutuelle_Modulaire_Par_Semaphore is
    -- valable quel que soit le nombre de processus

package Compte is
    procedure Section_critique;
end Compte;

package body Compte is
    -- on encapsule la donnée critique et son sémaphore de contrôle
    COMPTE_CLIENT : Integer := 0;          -- variable commune persistante
    S : Semaphore;          -- à initialiser avec une seule autorisation E.S = 1

    procedure body Section_critique is
        X : Integer := 0; -- variable locale
    begin
        P(S) ; -- ENTREE_SC1
        X := COMPTE_CLIENT;          -- P1
        X := X + 1;                  -- P2
        COMPTE_CLIENT := X;          -- P3
        V(S); -- SORTIE_SC1
    end Section_critique ;
begin
    E0(S,1) ; -- initialisation du sémaphore avant utilisation du paquetage
end Compte;          -- fin du module partagé, unité de bibliothèque

task type TP ;

task body TP is
begin
    for I in 1 .. 16 loop
        actions_hors_section_critique;
        Compte.Section_critique;
    end loop;
end TP;

P,Q, R ; TP ; -- on déclare des processus
begin les processus P, Q, R et le main sont concurrents
    null;
end Exclusion_Mutuelle_Modulaire_Par_Semaphore ;

```

EXCLUSION MUTUELLE MODULAIRE EN JAVA

```

public class Compte {
    // on encapsule la donnée critique, contrôle d'exclusion mutuelle implicite
    private int COMPTE_CLIENT = 0
        // variable locale à chaque objet instantié de la classe
        // pour une variable globale à la classe, il faudrait static
        // mais cela n'est pas permis avec synchronized – à vérifier
    public synchronized void Section_critique (){
        int X = COMPTE_CLIENT;    -- P1
        X := X + 1;                -- P2
        COMPTE_CLIENT := X;      -- P3
    } // fin de Section_critique ;
} // fin de la classe Compte

public class TP extends thread{
    private Compte compte;
    TP(Compte compte){this.compte = compte;}
    public void run(){
        for (int I=1; I <16; I++){
            actions_hors_section_critique;
            Compte.Section_critique;
        }
    } // fin de TP

public class Exclusion_Mutuelle_Modulaire_En_Java {
    // valable quel que soit le nombre de processus
    static Compte c = new Compte();
    public static void main(String[] args) {
        // programme principal
        TP p = new TP(c); TP q = new TP(c);
        // on doit passer en paramètre les objets partagés
        p.start(); q.start(); // lance les processus
        try {
            p.join();
            q.join();
        } catch (InterruptedException e) {
            System.out.println("un fils ne repond pas");
        }
        System.out.println(c); // imprime le résultat
    } // fin du programme principal main
} // fin de Exclusion_Mutuelle_Modulaire_En_Java

```

EXCLUSION MUTUELLE MODULAIRE AVEC OBJET PROTÉGÉ EN ADA

```

procedure Exclusion_Mutuelle_Modulaire_En_Ada is
    -- valable quel que soit le nombre de processus

    protected Compte is
        procedure Section_critique;
    private
        COMPTE_CLIENT : Integer := 0;    -- variable commune persistante
        -- on encapsule la donnée critique
    end Compte;

    protected body Compte is
        procedure body Section_critique is
            X : Integer := 0; -- variable locale
        begin
            X := COMPTE_CLIENT;    -- P1
            X := X + 1;            -- P2
            COMPTE_CLIENT := X;    -- P3
        end Section_critique ;
    end Compte;    -- fin de l'objet protégé

    task type TP ;

    task body TP is
    begin
        for I in 1 .. 16 loop
            actions_hors_section_critique;
            Compte.Section_critique;
        end loop;
    end TP;

    P, Q, R ; TP ; -- on déclare des processus
begin les processus P, Q, R et le main sont concurrents
    null;
end Exclusion_Mutuelle_Modulaire_En_Ada;

```

CONTRÔLE DE CONCURRENCE ET SYNCHRONISATION DES PROCESSUS

- **l'exclusion mutuelle ne suffit pas pour traiter tous les aspects de la synchronisation et le contrôle entre les processus concurrents**
- **il faut aussi pouvoir bloquer un processus et réveiller un processus**

SÉMAPHORES

- **on peut traiter tous les cas avec le seul mécanisme des sémaphores**

MONITEURS (ET JAVA)

- **le moniteur ajoute le type condition et une file d'attente associée à chaque variable de type condition**

ainsi que des primitives wait x, signal x, avec x : condition

Un processus se bloque par wait x et libère le moniteur

Un processus réveille un autre processus par signal x, mais un seul processus utilise le moniteur, l'autre doit attendre la libération du moniteur

- **En java cela s'appelle wait(), notify(), notifyAll()**

et il n'y a qu'une condition anonyme avec une seule file d'attente par moniteur

OBJETS PROTÉGÉS ADA

- **les conditions sont remplacées par des gardes sur les appels de procédure, transformées en entrées gardées par des expressions booléennes.**

• **Un processus appelant une procédure ou une entrée s'exécute en exclusion mutuelle. Les gardes sont réévaluées en fin de procédure ou d'entrée**

• **La fin d'une exécution d'entrée soit s'accompagne d'un retour du processus vers la procédure appelante, soit bloque à nouveau le processus si l'instruction requeue nom_d_entrée est programmée. L'objet protégé est libéré pour un autre processus.**

• **Avec les gardes et le requeue, on peut programmer l'utilisation d'un objet protégé comme un automate synchronisé (plus facile à valider).**

TYPE SEMAPHORE EN ADA95

```

package Semaphores is                                --specification : ads
----- ce paquetage implémente le type Semaphore
--          et les primitives classiques P, V and E0
-- chaque sémaphore déclaré est réalisé par un objet protégé
--          en Ada, un objet limité privé ne peut être
--          ni copié ni modifié en dehors du paquetage

type Semaphore is limited private;
procedure P (S : in out Semaphore);
procedure V (S : in out Semaphore);
procedure E0 (S : in out Semaphore; I : in Natural);

private

protected type Semaphore is
  entry V;
  entry P;
  entry E0(X : in Natural); -- X ≥ 0
  private -- variables persistantes de chaque Semaphore
    E : Natural; -- E ≥ 0 dans cette implantation
    INIT : Boolean := false; -- E0 fait avant P ou V
  end Semaphore ;
end Semaphores ;                                -- fin de l'ads

package body Semaphores is                        -- corps : adb

  protected body Semaphore is

    entry V when INIT is begin E := E + 1; end V;
    entry P when INIT and then E > 0 is begin E := E - 1; end P;
    entry E0(X : in natural) when not INIT is
      begin E := X; INIT := true; end E0;

    end Semaphore ;

    procedure P (S : in out Semaphore ) is begin S.P; end P;
    procedure V (S : in out Semaphore ) is begin S.V; end V;
    procedure E0 (S : in out Semaphore; I : in Natural) is
      begin S.E0(I); end E0;

end Semaphores ;                                -- fin de l'adb

```

IMPLANTATION DES SÉMAPHORES EN JAVA

```
public class semaphore {  
  
    public Semaphore() {compteur = 0; } // constructeur  
  
    public Semaphore(int v) {compteur = v; } // autre constructeur  
  
    public synchronized void P() {  
        while (compteur <= 0) {  
            try { wait() ; }  
            catch (InterruptedException e) {}  
        }  
        compteur--; // compteur = compteur - 1  
    }  
  
    public synchronized void V() {  
        ++ compteur; // compteur = compteur + 1  
        notify(); // ne réveille qu'un processus en attente  
    }  
  
    private int compteur;  
}
```