

**Informatique industrielle A7-19571**  
**Systemes temps-réel**  
*J.F.Peyre*

**Partie 6 : Communication et  
synchronisation en Ada et en C/Posix**

# Plan du cours

## ■ Synchronisation en Ada

- Présentation des objets protégés d'Ada
- Sémantique de base
- Utilisation des entrées

## ■ Synchronisation en C/Posix

- Les sémaphores Posix
- Les variables conditionnelles Posix
- Exemple d'utilisation

# Synchronisation en Ada

# Synchronisation en Ada

- **Ada propose plutôt un modèle centralisé (bien que l'annexe distribuée permette la construction d'applications réparties)**
- **Il n'existe pas de sémaphores**
- **Deux mécanismes sont proposés :**
  - Les « **objets protégés** » : proche des moniteurs; permet l'encapsulation de données privées qui sont accédées par des fonctions (lecture des données), par des procédures (lectures/écritures des données du moniteurs) et par des entrées (comme une procédure mais avec mise en attente possible de l'appelant)
  - L'utilisation de « **rendez-vous** » : proche des RPC entre une tâche serveur qui accepte le rendez-vous et une tâche client qui appelle le serveur pour ce rendez-vous (*non abordé dans ce cours*)

# Objet protégé

- Un objet protégé définit des **données privées** ne pouvant être accédées que par les sous-programmes (fonctions, procédures ou entrées) associés à l'objet protégé
- C'est un élément « **passif** » : ce sont les tâches qui exécutent le code des sous-programmes ou entrées des objets protégés
- Syntaxe à respecter :

```
protected Nom_De_L_Objet_Protege is
    déclaration des fonctions, procédures et entrées de l'objet protégé
private
    déclaration des données privées de l'objet protégé
end Nom_De_L_Objet_Protege;
```

## Objet protégé : sémantique de base

- La partie **privée** (« private ») définit les données ou contenu de l'objet protégé
- Les **fonctions** définissent des actions de « **lecture** » du contenu de l'objet protégé (interdiction de modifier la valeur de celles-ci)
- Les **procédures** et les **entrées** définissent des actions de « **lecture** » et « **d'écriture** » des données de l'objet protégé (contrairement à une procédure ou à une fonction, l'appel à une entrée peut être bloquant)
- Plusieurs lectures peuvent avoir lieu simultanément
- Une action d'écriture exclut toute autre action (lecture ou écriture)

# Objet protégé : exemple

```
type Valeur_Registre is array(1..Taille_Registre) of Integer;
protected Le_Registre is
  function Lire return Valeur_Registre;
  procedure Incremente;
private
  La_Valeur : Valeur_Registre := (others => 0);
end Le_Registre ;
```

*Partie spécification*

```
protected body Le_Registre is
  function Lire return Valeur_Registre is
  begin
    return La_Valeur ;
  end Lire;
  procedure Incremente is
  begin
    for I in La_Valeur 'range loop
      La_Valeur(I) := La_Valeur(I) + 1;
    end loop;
  end Incremente;
end Le_Registre ;
```

*Partie implémentation (body)*

## Objet protégé : exemple (suite)

```
task type T_F;  
task body T_F is  
begin  
  for I in 1..1_000_000 loop  
    Le_Registre.Incremente;  
  end loop;  
end T_F;
```

```
R_Local : Valeur_Registre;
```

```
begin -- début du main  
  declare  
    T1, T2: T_F; -- déclaration de deux tâches de type T_F  
  begin  
    null; -- execution de t1 et de t2 en concurrence; on attend la fin  
  end;  
  Put_line("Valeur finale du registre ");  
  R_Local := Le_Registre.Lire;  
  for I in R_Local'Range loop  
    Put_line(Integer'Image( R_Local(I) ));  
  end loop;  
end Prog_Exc;
```

## Objet protégé : exemple (suite)

**Voici un exemple d'exécution**

Valeur finale du registre

```
2000000  
2000000  
2000000  
2000000  
2000000  
2000000  
2000000  
2000000  
2000000  
2000000  
2000000
```

**On a bien 10 lignes avec la valeur 2000000 (deux millions)**

*-> les séquences lecture / incrémentation / écriture ne sont plus entrelacées : les incrémentations peuvent être considérées comme atomiques au niveau de l'objet protégé; ceci est dû à la sémantique d'accès exclusif des objets protégés*

## Objet protégé : sémantique des entrées

- Une entrée permet de définir des traitements sous conditions
  - Chaque **entrée** possède une **file d'attente** à laquelle est associée une expression booléenne que l'on appelle « **garde** » ou « **barrière** »
  - L'entrée n'est « **ouverte** » que si cette expression (la garde) est vraie
  - Lorsque l'entrée est « **fermée** » (la garde est fausse), les appels sur cette entrée sont mis en attente (ils ne seront traités que lorsque la garde redeviendra vraie)
- Une tâche, exécutant le code d'une entrée, peut être placée dans la file d'attente d'une autre entrée par l'instruction « **requeue** »
- Une tâche en attente sur une entrée interne est plus prioritaire qu'une tâche faisant un "nouvel" appel à une entrée (ou procédure) de l'objet protégé.

## Exemple d'utilisation des entrées

- Supposons que le registre étudié précédemment
  - corresponde à des paramètres de vol d'un avion et que la première case de ce registre indique si l'avion monte (valeur positive) ou s'il descend (valeur négative).
  - qu'il puisse être modifié par incrémentation ou décrémentation.
  - qu'il soit utilisé par une tâche qui ne gère que la descente de l'avion
- Une première possibilité est de faire faire une boucle d'attente à la tâche
  - lecture du registre;
  - vérification si le sens est « montée » ou « descente »;
  - si le sens est « montée »
    - attendre en certain temps et reboucler

***Cette technique est inefficace (consommation cpu) et complique inutilement le code de la tâche***

## Exemple d'utilisation des entrées (suite)

- Une seconde possibilité est d'utiliser une **entrée** d'un objet protégé afin de mettre sous conditions la lecture : dans notre exemple, la lecture ne sera possible **que si la valeur de la première case du registre est négative**
- Ceci s'implémente simplement de la façon suivante (*la partie qui ne change pas par rapport à la solution précédente est grisée*)

```
protected Le_Registre is
  function Lire return Valeur_Registre;
  procedure Incremente;
  procedure Decremente;
  entry Lire_Sous_Conditions(V: out Valeur_Registre);
private
  La_Valeur : Valeur_Registre := (others => 0);
end Le_Registre ;
```

*déclaration de l'entrée*



## Exemple d'utilisation des entrées (suite)

- Le corps de l'entrée utilise une *garde* qui correspond à la condition de progression

```
protected body Le_Registre is
  function Lire return Valeur_Registre is
  begin
    return La_Valeur ;
  end Lire;

  entry Lire_Sous_Conditions(V: out Valeur_Registre) when ( La_Valeur(1) < 0 ) is
  begin
    V := La_Valeur ;
  end Lire_Sous_Conditions;

  procedure Decremente is
  begin
    for I in La_Valeur 'range loop
      La_Valeur(I) := La_Valeur(I) - 1;
    end loop;
  end Decremente;
end Le_Registre ;
```

*Barrière associée à l'entrée*

*La barrière peut « s'ouvrir »*

# **Synchronisation en C / Posix**

# Mécanismes étudiés

- **Mutex Posix** (sémaphore binaire)
- **Sémaphores Posix**
- **Moniteurs Posix** : **variables conditionnelles** associées aux **Mutex**
- **Nous n'abordons pas**
  - l'utilisation de tube ("pipe")
  - l'utilisation des « sockets » (API TCP)
  - l'utilisation des sémaphores à la Unix (tableau de sémaphores)

# Gestion des Mutex

- Un « **Mutex** » est un sémaphore **binaire** pouvant prendre un des deux états "lock" (verrouillé) ou "unlock" (déverrouillé)
- Un « **Mutex** » ne peut être partagé que par des thread d'un même process
- Un « **Mutex** » ne peut être verrouillé que par un seul thread à la fois.
- Un thread qui tente de verrouiller un « **Mutex** » déjà verrouillé est **suspendu** jusqu'à ce que le « **Mutex** » soit déverrouillé.

# Déclaration et initialisation d'un Mutex

- Un mutex est une variable de type "`pthread_mutex_t`"
- Il existe une constante `PTHREAD_MUTEX_INITIALIZER` de ce type permettant une déclaration avec initialisation statique du mutex (avec les valeurs de comportement par défaut)

```
pthread_mutex_t monMutex = PTHREAD_MUTEX_INITIALIZER;
```

- Un mutex peut également être initialisé par un appel de la primitive

```
int pthread_mutex_init(pthread_mutex_t *mutex,  
                        const pthread_mutexattr_t *mutexattr);
```

avec une initialisation par défaut lorsque `mutexattr` vaut `NULL`

```
ex : pthread_mutex_init( &monMutex, NULL);
```

# Prise (verrouillage) d'un mutex

- Un mutex peut être verrouillé par la primitive

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- Si le mutex est déverrouillé il devient verrouillé
- Si le mutex est déjà verrouillé par un autre thread la tentative de verrouillage **suspend** l'appelant jusqu'à ce que le mutex soit déverrouillé.
- Si le mutex est déjà verrouillé *par le même thread* l'appel peut être
  - soit bloquant (comportement par défaut) : **attention risque d'interblocage**
  - soit non bloquant avec incrémentation d'un compteur définissant le nombre d'exemplaires du mutex possédé par le thread (comportement à la Java)

# Relâchement (déverrouillage) d'un mutex

- Un mutex peut être déverrouiller par la primitive

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Si le mutex est déjà déverrouillé, cet appel n'a aucun effet (comportement par défaut)
- Si le mutex est verrouillé, un des threads en attente obtient le mutex (qui reprend alors l'état verrouillé) et ce thread redevient actif (il n'est plus bloqué)
- L'opération est toujours non bloquante pour l'appelant

# Exemple d'utilisation de mutex

.../..

```
pthread_mutex_t monMutex = PTHREAD_MUTEX_INITIALIZER;
```

```
void LitRegistre(ValeurRegistre R){  
    int i; for (i=0; i<TAILLE_REGISTRE; i++) R[i] = LeRegsitre[i];  
}
```

```
void EcritRegistre(ValeurRegistre R){  
    int i; for (i=0; i<TAILLE_REGISTRE; i++) LeRegsitre[i] = R[i];  
}
```

```
void f(void){  
    int i, j; ValeurRegistre RLocal;
```

```
    for(j=0; j < 1000000; j++){  
        pthread_mutex_lock( &monMutex );  
        LitRegistre( RLocal );  
        for(i=0; i<TAILLE_REGISTRE; i++){ RLocal[i] ++; }  
        EcritRegistre( RLocal );  
        pthread_mutex_unlock( &monMutex );
```

*On verrouille le Mutex : accès exclusif*

*On déverrouille le Mutex*

```
    }  
}
```

## Exemple d'utilisation de mutex (*autre version*)

Autre possibilité : introduire une fonction d'incrémentation qui inclue la synchronisation

```
void IncrmenteRegistre(ValeurRegistre R){  
    int i;  
    ValeurRegistre RLocal;  
  
    pthread_mutex_lock(&leMutex);  
    LitRegistre( RLocal);  
    for(i=0; i<TAILLE_REGISTRE; i++){ RLocal[i] ++; }  
    EcritRegistre( RLocal);  
    pthread_mutex_unlock(&leMutex);  
}
```

Synchronisation

```
void f(void){  
    int i, j; ValeurRegistre RLocal;  
  
    for(j=0; j < 1000000; j++){  
        IncrmenteRegistre( RLocal );  
    }  
}
```

# Gestion des sémaphores Posix

- Un sémaphore Posix est un sémaphore à compte pouvant être partagé par plusieurs thread de plusieurs processus (selon les implémentations)
- Le **compteur** associé à un sémaphore peut donc prendre des valeurs plus grande que 1 (contrairement à un mutex)
- La prise d'un sémaphore dont le compteur est négatif ou nul **bloque** l'appelant
- Il ne faut pas confondre les sémaphores Posix avec les sémaphores Unix qui sont en fait des tableaux de sémaphores (non étudiés ici)

# Création / Initialisation d'un sémaphore Posix

- Les prototypes des fonctions et les types sont définis dans "`semaphore.h`"
- Un sémaphore est une variable de type "`sem_t`"
- Il est initialisé par un appel de la primitive

```
int sem_init(sem_t *sem, int pshared, unsigned int valeur);
```

  - si "`pshared`" vaut 0 le sémaphore ne peut pas être partagé entre tâches de différents processus (partage uniquement au sein d'un même processus)
  - `valeur` définit la valeur initiale de ce sémaphore (positif ou nul)
- Cette primitive correspond à la primitive E0 défini dans le chapitre précédent

# Prise / Relâchement d'un sémaphore Posix

- Les deux opérations P et V sont implémentées par

```
P : sem_wait(sem_t * sem);  
V : sem_post(sem_t * sem);
```

avec les mêmes comportements que les primitives génériques P et V

- Il existe également une version non bloquante de la primitive P :

```
int sem_trywait(sem_t * sem);
```

qui retourne 0 si quand la prise est possible (et non bloquante) et qui retourne **EAGAIN** sinon (dans le cas où l'appel normal serait bloquant)

# Sémaphores vs Mutex

- **Les sémaphores et les mutex ont à peu près le même rôle.**
  - Avantages Mutex
    - Les primitives de manipulation de mutex sont plus efficaces que les primitives de manipulation de sémaphore
  - Avantages sémaphores Posix
    - Les sémaphores Posix permettent une synchronisation entre threads de même processus (ce qui n'est pas possible avec les mutex)
    - Ils permettent de réaliser des compteurs (valeur plus grande que 1)
- **Bilan**

On préférera des mutex pour des sémaphores binaires au sein d'un même processus et des sémaphores Posix dans les autres cas

# Moniteurs Posix

## ■ Un moniteur Posix est l'association

- d'un mutex ( type `pthread_mutex_t` ) qui sert à protéger la partie de code où l'on teste les conditions de progression
- d'une variable conditionnelle ( type `pthread_cond_t` ) qui sert de point de signalisation :

- on se met en attente sur cette variable par la primitive

```
pthread_cond_wait(&laVariableConditionnelle, &leMutex);
```

- on signale sur cette variable avec la primitive

```
pthread_cond_signal(&laVariableConditionnelle);
```

# Schéma d'utilisation

Soit la condition de progression C

Le schéma d'utilisation des moniteurs Posix est le suivant :

```
pthread_mutex_lock(&leMutex);  
évaluer C;  
while ( ! C ){  
    pthread_cond_wait(&laVariableConditionnelle, &leMutex);  
    ré-évaluer C si nécessaire  
}  
Faire le travail;  
pthread_mutex_unlock(&leMutex);
```

# Exemple

```
void LitRegistreSousCondition(ValeurRegistre R){  
  
    pthread_mutex_lock(&leMutex);  
    while ( LeRegsitre[0] >= 0 ){  
        // la première valeur du registre est positive => on attend  
        // d'être signalé sur la variable conditionnelle (le mutex  
        // est automatiquement relâché lors de la mise en attente  
        // et repris lors du réveil)  
        printf("on va se bloquer\n");  
        pthread_cond_wait(&laVariableConditionnelle, &leMutex);  
    }  
    LitRegistre(R);  
    pthread_mutex_unlock(&leMutex);  
  
}
```

## Exemple (suite)

```
void DecrementeRegistre(ValeurRegistre R){
    int i;
    ValeurRegistre RLocal;

    pthread_mutex_lock(&leMutex);
    LitRegistre( RLocal);
    for(i=0; i<TAILLE_REGISTRE; i++){    RLocal[i] --; }
    EcritRegistre( RLocal);

    // si le registre est devenu négatif (première case) on signale
    // cet événement à un thread qui peut l'attendre
    if (RLocal[0] <0)
        pthread_cond_signal(&laVariableConditionnelle);
    pthread_mutex_unlock(&leMutex);
}
```

# Conclusion

- **Ada n'offre pas de sémaphore mais propose les objets protégés qui ont la sémantique des moniteurs et qui se manipulent très simplement**
  - Données internes (partie « privée »)
  - Lectures des données : fonctions
  - Modification des données : procédures
  - Lecture / Modification sous condition : entrée + barrière
- **Posix propose plusieurs mécanismes**
  - Mutex : pour les sections critiques
  - Sémaphores : pour les sections critiques et les coopérations élaborées
  - Variables conditionnelles : pour l'utilisation de moniteur