Université Bordeaux I
Session de janvier 2000

U.F.R. Mathématiques et Informatique
Alain Griffault

Maîtrise d'Informatique
Systèmes d'exploitation
(Durée : 3 heures)

Documents interdits

SUJET + CORRIGE

---

1 Cours : 2 points

Question 1.1 *Expliquez le problème de la liaison d'adresses (adresse logique ↔ adresse physique). Que signifient les termes* code absolu, code translatable, code relogable *?*

Réponse à la question 1.1 (2 points) : Un utilisateur écrit un programme qui contient des *adresses symboliques*. Le compilateur peut générer :

1. *des adresses physiques*, le code est alors *absolu.*
2. *des adresses logiques*, le code est alors *translatable.*
   Dans cette situation, le chargeur peut générer :
   (a) *des adresses physiques*, le code est alors *non relogable.*
   (b) *des adresses logiques*, le code est alors *relogable.*
   Dans cette situation, il faut pouvoir au moment de l'exécution faire la liaison d'adresses, cela n'est possible qu'avec du matériel spécifique (registre de translation). C'est le rôle de la MMU. On peut distinguer différentes situations :
      i. *partition unique* : il faut un registre de translation
      ii. *pagination* : l'adresse logique est généralement appelée adresse virtuelle (page, déplacement) : il faut une table des pages et une valeur de translation par cadres de page.
      iii. *segmentation* : l'adresse logique est simplement un déplacement dans le segment associé : il faut une valeur de translation par segment.

2 Algorithme du banquier : 5 points

Dans l'algorithme du banquier, chacun des processus exécute entre autres choses le code suivant :

```
Processus Pi
debut
  envoyer(banquier, annonceDesBesoinsMaximaux); // toujours autorisé
  Pour chaque vecteur V de ressources désirées
     envoyer(banquier,requete(V));              // soumis à un accord
     recevoir(banquier,accord);
  Pour chaque vecteur V' de ressources libérées
     envoyer(banquier,liberer(V'));             // toujours autorisé
fin
```

et le code du processus `banquier` ressemble à :

```
Processus banquier
debut
  tant que vrai faire debut
```

```
    recevoir(Pi, annonceDesBesoinsMaximaux);
      ou
    recevoir(Pi, liberer(V'));
      ou
    recevoir(Pi, requete(V));
    Suivant le résultat de l'algorithme du banquier:
      envoyer(Pi, accord);
        ou
      stocker la requete(V) de Pi
  fin
fin
```

Soit un système dont les ressources sont gérés par un processus banquier. Supposons maintenant que les événements suivants soient possibles :

Evt 1. un processus peut être créé dynamiquement par un processus P_i.

Evt 2. un processus peut être détruit dynamiquement par un processus P_i.

Evt 3. un processus P_i peut modifier son annonce maximale en cours d'exécution.

Question 2.1 *Dites pour chacun de ces événements, s'ils doivent être soumis à l'autorisation du banquier. Justifiez brièvement.*

Réponse à la question 2.1 (2 points) :

Evt 1. Deux réponses possibles selon que parmi les ressources protégées par l'algorithme du banquier se trouve ou non la liste (bornée) des processus :
- si oui, il faut une autorisation du banquier pour créer dynamiquement un processus,
- si non, l'autorisation n'est pas nécessaire, il y aura éventuellement un refus par le S.E.

Evt 2. Détruire dynamiquement un processus ne peut pas transformer une situation saine en une situation malsaine. Il n'y a donc pas lieu de demander l'autorisation au banquier. Il faut simplement le prévenir pour qu'il sache que les ressources détenues par ce processus sont maintenant disponibles.

Evt 3. Si la modification des besoins maximaux est une diminution, il n'est pas nécessaire d'avoir l'autorisation. Si c'est une augmentation, il se peut que la situation devienne malsaine, et l'autorisation du banquier est alors nécessaire.

Le nouveau code d'un processus peut donc ressembler à :

```
Processus Pi
debut
  envoyer(banquier, annonceDesBesoinsMaximaux);           // toujours autorisé
  Pour chaque vecteur V de ressources désirées
     envoyer(banquier,requete(V));                        // soumis à un accord
     recevoir(banquier,accord);
  Pour chaque vecteur V' de ressources libérées
     envoyer(banquier,liberer(V'));                       // toujours autorisé
  envoyer(banquier, creationProcessus)                    // toujours autorisé
  envoyer(banquier, destructionProcessus)                 // toujours autorisé
  envoyer(banquier, annonceDiminutionBesoinsMaximaux);    // toujours autorisé
  envoyer(banquier, annonceAugmentationBesoinsMaximaux); // soumis à un accord
  recevoir(banquier,accord);
fin
```

Question 2.2 *Pensez vous qu'il soit nécessaire de modifier l'algorithme du banquier pour traiter ces nouveaux événements. Justifiez et/ou donnez les caractéristiques des modifications.*

Réponse à la question 2.2 (3 points) : Lorsqu'un processus augmente son annonce maximale, il peut transformer (sans qu'aucune nouvelle ressource soit allouée) une situation saine en une situation malsaine ; car il n'existe peut-être plus d'ordonnancement des processus dans le cas ou chacun d'entre eux effectuerait une requête maximale.

L'algorithme de détection d'un état sain n'est donc pas suffisant, Deux solutions : soit refuser une augmentation qui transformerait l'état sain en un état malsain, soit mettre en attente, en espérant que dans le futur, il existera une situation saine permettant l'augmentation (Attention : il n'y a aucune garantie à ce sujet). Voici donc un code possible pour le banquier :

```
Processus banquier
debut
  tant que vrai faire debut
    recevoir(Pi, creationProcessus);
      ou
    recevoir(Pi, destructionProcessus);
                   // il faut libérer les ressources du processus détruit,
                   // il faut supprimer ses requêtes en attente.
      ou
    recevoir(Pi, annonceDesBesoinsMaximaux);
      ou
    recevoir(Pi, annonceDiminutionBesoinsMaximaux);
      ou
    recevoir(Pi, liberer(V'));
      ou
    recevoir(Pi, requete(V));
    Suivant le résultat de l'algorithme du banquier:
      envoyer(Pi, accord);
        ou
      stocker la requete(V) de Pi
      ou
    recevoir(Pi, annonceAugmentationBesoinsMaximaux);
    Suivant le résultat de l'algorithme du banquier:
      envoyer(Pi, accord);
        ou
      stocker annonceAugmentationBesoinsMaximaux de Pi
                   // il faut continuer à traiter les nouvelles
                   // requetes avec les anciennes valeurs maximales
  fin
fin
```

3 Sémaphores : 5 points

Une ligne de chemin de fer reliant deux villes A et B comporte une section à voie unique

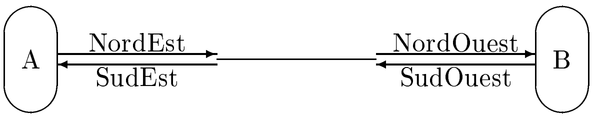

Figure 1: Une ligne de chemin de fer

On représente les trains par des processus :

```
Processus trainsA-B
debut
  NordEst
  ``Protocole d'entree''
  VoieUnique
```

```
Processus trainsB-A
debut
  SudOuest
  ``Protocole d'entree''
  VoieUnique
```

```
  ''Protocole de sortie''              ''Protocole de sortie''
  NordOuest                            SudEst
fin                                  fin
```

Deux contraintes doivent toujours être satisfaites :

C 1. Pas de collision (c-à-d un `trainsA-B` et un `trainsB-A` en même temps sur la voie unique).

C 2. La voie unique ne reste pas indéfiniment inutilisée.

et si possible la contrainte suivante serait un plus :

C 3. Aucun train n'attend indéfiniment longtemps avant d'entrer sur la voie unique.

Definition 3.1 *Un sémaphore est une structure de données*

```
type semaphore = (n : entier; l : liste de processus);
```

manipulée uniquement par les actions atomiques `wait` *et* `signal`.

```
procedure wait(var S : semaphore)
debut
S.n := S.n - 1;
si (S.n < 0) alors debut
        ajouter(moi,S.l);
        block;
        fin
fin
```

```
procedure signal(var S : semaphore)
debut
S.n := S.n + 1;
si (S.n <= 0) alors debut
        enlever(P,S.l);
        wakeup(P);
        fin
fin
```

En s'inspirant du problème des lecteurs-rédacteur, donnez pour chacune des questions suivantes, les sémaphores nécessaires (ainsi que leurs valeurs initiales), les variables partagées utilisées (ainsi que leurs valeurs initiales), les codes des processus `trainsA-B` et `trainsB-A`, et les contraintes satisfaites par votre solution.

Question 3.1 *On suppose que le nombre de trains présents au même instant sur la voie unique est limité à un.*

Réponse à la question 3.1 (1 point) : La voie unique est dans ce cas une ressource non partageable, le problème devient le problème de la section critique, d'où une solution :

```
Semaphore voieUnique : (1,{});
```

```
Processus trainsA-B
debut
  NordEst
  wait(voieUnique) ;
  VoieUnique
  signal(voieUnique) ;
  NordOuest
fin
```

```
Processus trainsB-A
debut
  SudOuest
  wait(voieUnique) ;
  VoieUnique
  signal(voieUnique) ;
  SudEst
fin
```

Les trois contraintes C 1, C 2 et C 3 sont vérifiées.

On peut remarquer que cette solution est également une solution (mais peu optimale en terme d'utilisation de la voie unique) aux deux questions suivantes car les contraintes y sont moins fortes.

Question 3.2 *On suppose que le nombre de trains présents au même instant sur la voie unique n'est pas limité.*

Réponse à la question 3.2 (2 points) : La voie unique est une ressource partageable, mais en deux modes exclusifs (les deux sens de circulation). Il suffit donc de transformer une solution du problème des lecteurs-rédacteur vers une solution à un problème des lecteursType1-lecteursType2, d'où une solution :

```
Entier    AB, BA : 0; // servent à compter les trains.
Semaphore sens, SemAB, SemBA : (1,{});
          // SemAB et SemBA protègent les manipulations des variables AB et BA
          // sens gère l'exclusion entre les deux sens de circulation
```

```
Processus trainsA-B
debut
  NordEst
  wait(SemAB) ;
    AB++ ;
    si (AB==1) alors
      wait(sens) ;
  signal(SemAB) ;
  VoieUnique
  wait(SemAB) ;
    AB-- ;
    si (AB==0) alors
      signal(sens) ;
  signal(SemAB) ;
  NordOuest
fin
```

```
Processus trainsB-A
debut
  SudOuest
  wait(SemBA) ;
    BA++ ;
    si (BA==1) alors
      wait(sens) ;
  signal(SemBA) ;
  VoieUnique
  wait(SemBA) ;
    BA-- ;
    si (BA==0) alors
        signal(sens) ;
  signal(SemBA) ;
  SudEst
fin
```

Les contraintes C 1 et C 2 sont vérifiées, alors que C 3 ne l'est pas. Les solutions distribués qui vérifient C 3 sont beaucoup plus complexes. Pour de telle propriété, il est souvent plus facile de mettre en œuvre une solution centralisée de type ordonnanceur.

Question 3.3 *On suppose que le nombre de trains présents au même instant sur la voie unique est limité à une constante N.*

Réponse à la question 3.3 (2 points) : Adaptée la solution précédente n'est pas très difficile, il suffit d'un sémaphore qui contrôle l'entrée dans la voie unique.

```
Entier    AB, BA : 0; // servent à compter les trains.
Semaphore sens, SemAB, SemBA : (1,{});
          // SemAB et SemBA protègent les manipulations des variables AB et BA
          // sens gère l'exclusion entre les deux sens de circulation
          voieUnique : (N,{});
          // initialisation à N pour limiter à N les trains sur la voieUnique
```

```
Processus trainsA-B
debut
  NordEst
  wait(SemAB) ;
    AB++ ;
    si (AB==1) alors
      wait(sens) ;
  signal(SemAB) ;
  wait(voieUnique) ;
  VoieUnique
  signal(voieUnique) ;
  wait(SemAB) ;
    AB-- ;
    si (AB==0) alors
      signal(sens) ;
  signal(SemAB) ;
  NordOuest
```

```
Processus trainsB-A
debut
  SudOuest
  wait(SemBA) ;
    BA++ ;
    si (BA==1) alors
      wait(sens) ;
  signal(SemBA) ;
  wait(voieUnique) ;
  VoieUnique
  signal(voieUnique) ;
  wait(SemBA) ;
    BA-- ;
    si (BA==0) alors
        signal(sens) ;
  signal(SemBA) ;
  SudEst
```

```
fin                                         fin
```
Les contraintes C 1 et C 2 sont vérifiées, alors que C 3 ne l'est toujours pas. La remarque précédente concernant C 3 est encore vraie.

4 Transactions et verrouillage deux phases : 8 points

Une transaction est une suite d'actions élémentaires (*lire* et *ecrire*). On s'intéresse aux exécutions possibles d'un ensemble de transactions $\{T_i\}$ qui accèdent à un ensemble d'objets communs.

un exemple Soient $T_1 = \{ecrire(a,1); ecrire(b,1)\}$ et $T_2 = \{ecrire(a,2); ecrire(b,2)\}$ deux transactions. Nous avons un ensemble de six exécutions possibles :

E1 : $\{ecrire(a,1); ecrire(b,1); ecrire(a,2); ecrire(b,2)\}$
E2 : $\{ecrire(a,1); ecrire(a,2); ecrire(b,1); ecrire(b,2)\}$
E3 : $\{ecrire(a,1); ecrire(a,2); ecrire(b,2); ecrire(b,1)\}$
E4 : $\{ecrire(a,2); ecrire(a,1); ecrire(b,1); ecrire(b,2)\}$
E5 : $\{ecrire(a,2); ecrire(a,1); ecrire(b,2); ecrire(b,1)\}$
E6 : $\{ecrire(a,2); ecrire(b,2); ecrire(a,1); ecrire(b,1)\}$

Les exécutions E1 et E6 sont dites séquentielles, les autres parallèles (ou entrelacées).

Definition 4.1 *Soient A_i et A_j deux actions d'une même exécution. A_i dépend de A_j si et seulement si :*
- *A_i est après A_j,*
- *A_i et A_j accèdent au même objet,*
- *A_i ou A_j* **au moins** *est une écriture.*

Cette relation définit entre les actions d'une exécution un graphe de dépendance étiqueté et orienté. Chaque arc du graphe sera noté $A_i^m \xrightarrow{o} A_j^n$ ou o est l'objet sur lequel porte le conflit et A_k^p représente la $p$ème actions de la transaction T_k

Question 4.1 *Soit une exécution d'un ensemble de transaction $\{T_i\}$. Montrez que son graphe de dépendance est sans cycle.*

Réponse à la question 4.1 (1 point) : La relation est anti-symétrique (première condition), elle définit donc un ordre. Comme tout ordre caractérise un graphe sans cycle, le graphe de dépendance de toute exécution est sans cycle.

Question 4.2 *Montrez que si deux exécutions de l'ensemble $\{T_i\}$ produisent le même graphe de dépendance, alors elles ont le même effet (c-à-d les valeurs des objets à la fin des deux exécutions sont identiques).*

Réponse à la question 4.2 (1 point) : Supposons que les deux exécutions produisent sur un objet o deux valeurs différentes. Deux cas sont à considérer :

1. Les deux valeurs proviennent (la dernière écriture) de deux processus P_i et P_j différents.
$$\left\{\begin{array}{l} A_i^m = ecrire(o) \\ A_j^n = ecrire(o) \\ i \neq j \end{array}\right. \Rightarrow \left\{\begin{array}{l} A_i^m \xrightarrow{o} A_j^n \text{ si } A_i^m \text{ est après } A_j^n \\ \text{ou exclusif} \\ A_j^n \xrightarrow{o} A_i^m \text{ si } A_j^n \text{ est après } A_i^m \end{array}\right.$$
$$\Rightarrow \text{Les graphes de dépendances diffèrent.}$$
2. Les deux valeurs proviennent du même processus (la valeur de o est dépendante pour ce processus d'au moins un autre objet o' lu) et cette valeur lue diffère dans les deux exécutions.

$$\left\{\begin{array}{l} A_i^m = ecrire(o) \\ A_i^p = lecture(o') \\ p < m \end{array}\right. \quad \Rightarrow \quad \exists j / \left\{\begin{array}{l} A_i^m = ecrire(o) \\ A_i^p = lecture(o') \\ A_j^n = ecrire(o') \\ i \neq j \\ p < m \end{array}\right.$$

$$\Rightarrow \quad \exists j / \left\{\begin{array}{l} A_i^p \xrightarrow{o'} A_j^n \text{ si } A_i^p \text{ est après } A_j^n \\ \text{ou exclusif} \\ A_j^n \xrightarrow{o'} A_i^p \text{ si } A_j^n \text{ est après } A_i^p \end{array}\right.$$

$$\Rightarrow \quad \text{Les graphes de dépendances diffèrent.}$$

Nous avons ainsi montré que deux exécutions qui ont le même graphe de dépendance produisent les mêmes valeurs pour tous les objets.
Remarque : il est également possible de faire une démonstration directe par une récurrence sur la longueur de la transaction.

Definition 4.2 *Une exécution d'un ensemble de transactions $\{T_i\}$ est cohérente si et seulement si il existe une exécution séquentielle (c-à-d un ordonnancement des transactions) qui produit le même effet sur les objets.*

Le verrouillage à deux phases est une méthode pour interdire certaines exécutions entrelacées afin que les seules exécutions possibles d'un ensemble de transactions $\{T_i\}$ soient cohérentes.
Une transaction **avec verrouillage** T_i est maintenant une suite A_i^m d'actions élémentaires (*lire*, *ecrire*, *verrouiller*, *deverrouiller*). Intuitivement, on n'accède qu'aux objets verrouillés dans un mode compatible. Le mode *ecriture* étant exclusif et le mode *lecture* partagé. Plus formellement :

- $A_i^m = verrouiller(o, lecture)$ est acceptée si et seulement si o n'est pas verrouillé en *ecriture* par une autre transaction. La transaction est mise en attente sinon.
- $A_i^m = verrouiller(o, ecriture)$ est acceptée si et seulement si o n'est pas verrouillé ni en *lecture* ni en *ecriture* par une autre transaction. La transaction est mise en attente sinon.
- $A_i^m = deverrouiller(o)$ est toujours acceptée.
- $A_i^m = lire(o) \Rightarrow \left\{\begin{array}{l} \exists n < m \ / \ A_i^n = verrouiller(o, mode), mode \in \{lecture, ecriture\} \\ \forall p \ / \ n < p < m, A_i^p \neq deverrouiller(o) \end{array}\right.$
- $A_i^m = ecrire(o) \Rightarrow \left\{\begin{array}{l} \exists n < m \ / \ A_i^n = verrouiller(o, ecriture) \\ \forall p \ / \ n < p < m, A_i^p \neq deverrouiller(o) \end{array}\right.$

Une transaction avec verrouillage est dite **à deux phases** si de plus elle respecte les règles suivantes :

- $A_i^m = verrouiller(o, mode) \Rightarrow \exists p \ / \ p > m, A_i^p = deverrouiller(o)$
- $A_i^m = deverrouiller(o) \Rightarrow \forall o', \forall p \ / \ p > m, A_i^p \neq verrouiller(o', mode)$

Intuitivement, tous les objets doivent être déverrouillés, et dès lors que l'on a commencé à déverrouiller un objet, on ne peut plus en verrouiller aucun. **La seconde phase d'une transaction débute donc avec le premier déverrouillage.**
Dans toute la suite de l'exercice, les transactions T_i sont supposées avec verrouillage à deux phases.

Question 4.3 *Soit $A_i^{d_i}$ l'action qui débute la seconde phase de T_i.*
Montrez que le graphe de dépendance d'une exécution d'un ensemble de transactions $\{T_i\}$ est identique à celui correspondant à l'exécution séquentielle des $\{T_i\}$ suivant l'ordre chronologique des actions $\{A_i^{d_i}\}$.

Réponse à la question 4.3 (1,5 point) : Notons $<$ l'ordre chronologique. Considérons un arc du graphe G de dépendance d'une exécution. Soit $Seq(G)$ le graphe séquentiel associé. Deux cas à considérer :

1. Les deux sommets appartiennent au même processus P_i.
   $A_i^m \xrightarrow{o} A_i^n \in G \Leftrightarrow A_i^m \xrightarrow{o} A_j^n \in Seq(G)$
2. Les deux sommets appartiennent à deux processus différents P_i et P_j.

$$
\begin{aligned}
A_i^m \xrightarrow{o} A_j^n \in G \quad & \Leftrightarrow \quad \begin{cases} A_i^m \xrightarrow{o} A_j^n \in G \\ verrouiller_i(o, mode_i) < A_i^m < deverrouiller_i(o) \\ verrouiller_j(o, mode_j) < A_j^n < deverrouiller_j(o) \\ mode_i \text{ ou } mode_j \text{ est une } ecriture \end{cases} \\
& \Leftrightarrow \quad \begin{cases} A_i^m \xrightarrow{o} A_j^n \in G \\ verrouiller_j(o, mode_j) < A_j^n < deverrouiller_j(o) < \\ \qquad\qquad verrouiller_i(o, mode_i) < A_i^m < deverrouiller_i(o) \end{cases} \\
& \Leftrightarrow \quad \begin{cases} A_i^m \xrightarrow{o} A_j^n \in G \\ A_j^{d_j} < A_i^{d_i} \end{cases} \\
& \Leftrightarrow \quad A_i^m \xrightarrow{o} A_j^n \in Seq(G)
\end{aligned}
$$

Le graphe de dépendance d'une exécution d'un ensemble de transactions $\{T_i\}$ est donc bien identique à celui correspondant à l'exécution séquentielle des $\{T_i\}$ suivant l'ordre chronologique des actions $\{A_i^{d_i}\}$.

Question 4.4 *Toutes les exécutions d'un ensemble de transactions $\{T_i\}$ sont-elles cohérentes ? Justifiez.*

Réponse à la question 4.4 (1 point) : Soit une exécution E.
D'après la réponse à la question 4.3, il existe une exécution séquentielle dont le graphe de dépendance est identique.
D'après la réponse à la question 4.2, il existe une exécution séquentielle qui produit les mêmes effets.
D'après la définition 4.2, on en déduit que E est cohérente.

Question 4.5 *Le verrouillage à deux phases possède un gros défaut. Lequel ?*

Réponse à la question 4.5 (1 point) : Soit l'exécution suivante :

1. $verrouiller_i(o_1, ecriture)$
2. $verrouiller_j(o_2, ecriture)$
3. $verrouiller_i(o_2, ecriture)$ // mise en attente
4. $verrouiller_j(o_1, ecriture)$ // mise en attente

Cette situation est inextricable, le verrouillage à deux phases peut donc provoquer des interblocages. On peut noter (du fait qu'il n'y a pas de verrouillage dans la seconde phase) que l'interblocage est obligatoirement pendant la première phase d'au moins une des transactions bloquées.

Definition 4.3 *Le schéma* `Wait-Die` *pour éviter l'interblocage s'appuie sur un technique sans réquisition. Lorsqu'un processus P_i désire une ressource détenue par P_j, soit il attend s'il est plus vieux, soit il meurt s'il est plus jeune.*

Definition 4.4 *Le schéma* `Wound-Wait` *pour éviter l'interblocage s'appuie sur un technique avec réquisition. Lorsqu'un processus P_i désire une ressource détenue par P_j, soit il attend s'il est plus jeune, soit il blesse P_j en lui prenant ses ressources s'il est plus vieux.*

Question 4.6 *Dans le cas de transactions vues comme des processus, quel est le problème posé par la mort et/ou la blessure d'une transaction. Proposez une solution.*

Réponse à la question 4.6 (1,5 point) : Lorsqu'un processus meurt au "milieu" d'une transaction, il laisse en général le système (les valeurs des objets) dans un état incohérent. Le problème est donc la remise en état du système dans un état cohérent correspondant en général à l'état du système au début de la transaction.
Dans le cas d'un ensemble de transactions, cela peut générer la mort et/ou la blessure d'autres processus par un effet de cascade : chacun ayant lu un objet modifié par une transaction annulée.

On peut donc quantifier les solutions lors de la mort d'un processus :

très mauvaise : n'évite pas l'effet de cascade,

mauvaise : évite l'effet de cascade, mais nécessite un retour arrière pour revenir dans un état cohérent,

bonne : évite l'effet de cascade, et ne nécessite pas de retour arrière car l'état du système est cohérent.

D'après la réponse à la question 4.3, il suffit lors d'un interblocage de tuer une transaction qui est dans sa première phase pour avoir une **bonne** solution.

Question 4.7 *Listez les besoins et les problèmes rencontrés pour la mise en œuvre de ces deux schémas. Lequel des deux schémas précédents vous semble le mieux adapté dans le cas du verrouillage à deux phases ? Justifiez.*

Réponse à la question 4.7 (1 point) : Pour les deux schémas, il faut mettre en place un système pour dater des événements. On peut utiliser par exemple les couples (IP,numéro de processus). Cette date est utile :

- afin de pouvoir comparer l'âge des différents processus.
- afin qu'une transaction annulée ou tuée débute sa nouvelle exécution avec son ancienne date, ce qui augmente sa priorité, donc ses chances de ne pas être annulée une seconde fois.

Il faut également pouvoir déverrouiller les objets verrouillés par une transaction annulée.
Dans l'exemple précédent, si l'on suppose que P_i est plus vieux que P_j, on obtient :

Wait-Die :

1. $verrouiller_i(o_1, ecriture)$
2. $verrouiller_j(o_2, ecriture)$
3. $verrouiller_i(o_2, ecriture)$ // P_i plus vieux attend.
4. $verrouiller_j(o_1, ecriture)$ // P_j plus jeune meurt, libère o_2 et débloque ainsi P_i.

Wound-Wait :

1. $verrouiller_i(o_1, ecriture)$
2. $verrouiller_j(o_2, ecriture)$
3. $verrouiller_i(o_2, ecriture)$ // P_i plus vieux blesse P_j en lui prenant o_2.
4. $verrouiller_j(o_2, ecriture)$ // P_j devra attendre de nouveau pour o_2.
5. $verrouiller_j(o_1, ecriture)$ // avant de redemander o_1.

On peut remarquer que :

Wait-Die La transaction qui meurt est encore en première phase (elle désire verrouiller un objet), il n'y a donc pas de réaction en cascade, elle doit seulement déverrouiller tous les objets qu'elle détient avant de mourir. La solution est **bonne**.

Wound-Wait Si la transaction blessée est en première phase, il n'y a ni réaction en cascade ni retour arrière. Si elle est en seconde phase, il y a risque de cascade.

Supposons maintenant qu'elle n'ait plus aucune écriture à faire (c'est à dire qu'intuitivement, elle a fini son travail), il est dans ce cas inutile d'annuler son travail, il suffit de déverrouiller ses objets (c'est la seule tâche qui lui restait à faire), il suffit donc de la laisser finir.

Le verrouillage strict à deux phases demande que toutes les écritures soient faites dans la première phase. Avec une telle règle, le schéma Wound-Wait ne provoque ni réaction en cascade ni retour arrière : si la transaction blessée est en première phase, elle est annulée, sinon on la laisse terminer.

Le verrouillage à deux phases permet d'éviter la réaction en cascade et le retour arrière dans le cas du schéma Wait-die.
Le verrouillage strict à deux phases permet d'éviter la réaction en cascade et le retour arrière dans le cas du schéma Wound-Wait.
Quel est donc le meilleur schéma ? il n'y a pas de réponse optimale, chacun ayant ses défauts et avantages. Les deux fonctionnent bien sous l'hypothèse d'un nombre faible d'interblocages. Si celui-ci est important, il vaut mieux utiliser d'autres techniques (protocole avec estampille sur toutes les actions des transactions).