

Corrigé

Les téléphones portables doivent être éteints et rangés dans vos sacs.
Le barème (sur 40) est donné à titre indicatif.

Avant propos

Toutes vos réponses doivent être dûment justifiées.

Pensez à présenter clairement vos réponses, en particulier les programmes.

Lisez bien attentivement le sujet avant de répondre aux questions. Toute réponse hors sujet sera considérée comme fausse.

*Vous devez impérativement indiquer **.a11** chaque fois que vous aurez affaire à un pointeur. Nous considérerons que l'absence des **.a11** lorsqu'on les attend correspond à une réponse fausse.*

Exercice 1 – Distribuons des iPhones (18 points)

Considérons le programme suivant.

```
4  task Ma_Pomme is
5      entry Achete_Iphone (
6          Nb : in      Positive );
7      entry Recoit_Iphone (
8          Nb : in      Positive );
9  end Ma_Pomme;

10 task type Client (Id : Positive);
11 type A_Client is access Client;

12 task body Client is
13 begin
14     Ma_Pomme.Achete_Iphone (Id);
15     Put_Line ("Youpie, moi" & Positive'Image (Id) & ", j'ai obtenu mes iPhones...");
16 end Client;

17 task body Ma_Pomme is
18     Nb_Iphone : Natural := 0;
19 begin
20     loop
21         select
22             when Nb_Iphone > 0 =>
23                 accept Achete_Iphone (
24                     Nb : in      Positive ) do Nb_Iphone := Nb_Iphone - Nb;
25                     Put_Line ("Apple: j'ai vendu" & Positive'Image (Nb) & " iPhones.");
26                 end Achete_Iphone;
27             or
28                 accept Recoit_Iphone (
29                     Nb : in      Positive ) do Nb_Iphone := Nb_Iphone + Nb;
30                     Put_Line ("Apple: j'ai recu" & Positive'Image (Nb) & " iPhones.");
31                 end Recoit_Iphone;
32             end select;
33         end loop;
34     end Ma_Pomme;
```

Question 1 (2 points) : La tâche Ma_Pomme sait-elle se terminer seule ? si non, expliquez pourquoi et comment y remédier ? le nombre de clients joue-t-il un rôle dans la terminaison ? Justifiez vos réponses.

Solution de la question 1 de l'exercice 1:

- (1) Non. Elle ne sait pas se terminer car elle est constituée d'une boucle infinie.
- (2) Pour remédier au problème, il faut ajouter `or terminate;` juste avant la fin du `select`.
- (3) Le nombre de clients ne joue aucun rôle dans le problème de la terminaison.

Fin de la solution de la question 1 de l'exercice 1.

On ajoute à ce programme la partie suivante :

```
36  declare
37      Les_Clients : array (1 .. 5) of A_Client;
38  begin
39      for I in Les_Clients'range loop
40          Les_Clients (I) := new Client(I);
41      end loop;
42      Ma_Pomme.Recoit_Iphone (11);
43  end;
```

Question 2 (3 points) : Combien de tâches sont créées en tout ? Pour chacune d'elle, indiquez à quel niveau du programme elles sont créées, et à quel niveau du programme leur exécution est activée.

Solution de la question 2 de l'exercice 1:

7 tâches sont créées :

- La tâche `Ma_Pomme`, au niveau de sa déclaration (ligne 4 ou 17, les deux sont acceptables à ce stade),
- Le programme principal ou "tâche d'environnement", créé au niveau du `begin` (on n'a pas la ligne ici),
- 5 tâches de type `Client` créées explicitement via l'instruction `new` par la tâche d'environnement (ligne 40).

Sinon, les tâches sont toutes activées au moment où l'on passe le `begin` de la procédure principale ou du paquetage principal sauf pour les clients qui sont activés à leur création, c'est-à-dire au moment du `new`.

Fin de la solution de la question 2 de l'exercice 1.

Question 3 (4 points) : Quelles sont les tâches qui se terminent normalement et quelles sont celles qui ne se terminent pas normalement ? Justifiez votre réponse.

Solution de la question 3 de l'exercice 1:

L'exécution du programme se déroule comme suit.

Tout d'abord, tous les clients qui sont créés sont bloqués sur l'accept de `Achete_Iphone` car, comme il y a 0 iPhones en stock, la garde associée à ce point d'entrée ne peut se déclencher.

Ensuite, tout se passe bien tant que les quatre premiers clients s'exécutent. En effet, ces derniers obtiennent ce qu'ils veulent. Mais lorsque le 5ème souhaite récupérer 5 iPhone, il n'en reste qu'un seul. La garde associée au point d'entrée `Achete_Iphone` est toujours vérifiée car il reste bien un iPhone. Mais l'opération de soustraction provoque alors une erreur puisque `Nb_Iphone` est de type `Natural`.

`Ma_Pomme` se termine en erreur avant la fin du point d'entrée (sur levée de `Constraint_Error`). Au passage, notons que le client 5 se termine aussi brutalement en erreur.

Donc Toutes tâches se terminent normalement sauf `Les_Clients(5)` et `Ma_Pomme`.

L'exécution du programme provoque donc l'affichage suivant.

```
Apple: j'ai reçu 11 iPhones.
Apple: j'ai vendu 1 iPhones.
Apple: j'ai vendu 2 iPhones.
Apple: j'ai vendu 3 iPhones.
Apple: j'ai vendu 4 iPhones.
Youpie, moi 2, j'ai obtenu mes iPhones...
Youpie, moi 3, j'ai obtenu mes iPhones...
Youpie, moi 1, j'ai obtenu mes iPhones...
Youpie, moi 4, j'ai obtenu mes iPhones...
```

Fin de la solution de la question 3 de l'exercice 1.

Question 4 (2 points) : Pour quelle raison le problème ne peut-il être résolu en changeant la clause `when` dans l'accept de `Achete_Iphone` ?

Solution de la question 4 de l'exercice 1:

Parce que la condition d'acceptation de ce point d'entrée est clairement liée au paramètre transmis par l'appelant, ce que l'on ne peut faire.

Fin de la solution de la question 4 de l'exercice 1.

Question 5 (3 points) : Proposez une solution pour remédier à ce problème. Sans la programmer, vous définirez précisément le mécanisme mis en œuvre et son usage.

Solution de la question 5 de l'exercice 1:

Il faut pouvoir refuser l'accept de Achete_Iphone si le nombre d'iPhones demandé. Pour cela, il faut utiliser la directive `requeue` qui permet de remettre une requête qu'on a lue sur la file d'attente...

La solution complète est donnée ci après pour information.

```

task body Ma_Pomme is
    Nb_Iphone : Natural := 0;
    Bloquer   : Boolean := True;

begin
    loop
        select
            when Nb_Iphone > 0 and not Bloquer=>
                accept Achete_Iphone (
                    Nb : in Positive ) do null;
                    if Nb > Nb_Iphone then
                        Put_Line (
                            "Apple: je n'ai plus assez d'iPhones.");
                        Bloquer := True;
                        requeue Achete_Iphone;
                    else
                        Nb_Iphone := Nb_Iphone -Nb;
                        Put_Line ("Apple: j'ai vendu" & Positive'Image (Nb) & " iPhones.");
                    end if;
                end Achete_Iphone;
            or
                accept Recoit_Iphone (
                    Nb : in Positive ) do Nb_Iphone := Nb_Iphone + Nb;
                    Bloquer := False;
                    Put_Line ("Apple: j'ai reçu" & Positive'Image (Nb) & " iPhones.");
                end Recoit_Iphone;
            or
                terminate;
            end select;
        end loop;
    exception
        when E : others =>
            Put_Line ("Ma_Pomme : " & Exception_Name (E));
    end Ma_Pomme;

```

Fin de la solution de la question 5 de l'exercice 1.

Question 6 (4 points) : Transformez la tâche Ma_Pomme du listing avant la question 1 en un type protégé au comportement équivalent (spécification et body).

Solution de la question 6 de l'exercice 1:

On donne pour information le code complet mais seul celui du type protégé (spec et body) compte

```

with Ada.Text_IO, Ada.Exceptions;
use Ada.Text_IO, Ada.Exceptions;

procedure Ma_Pomme3 is
    protected Ma_Pomme is
        entry Achete_Iphone (
            Nb : in Positive );
        procedure Recoit_Iphone (
            Nb : in Positive );
    private

```

```

    Nb_Iphone : Natural := 0;
end Ma_Pomme;

task type Client (Id : Positive);
type A_Client is access Client;

task body Client is
begin
    Ma_Pomme.Achete_Iphone (Id);
    Put_Line ("Youpie, moi" & Positive'Image (Id) & ", j'ai obtenu mes iPhones...");
exception
    when E : others =>
        Put_Line ("client" & Positive'Image (Id) & " : " & Exception_Name (E));
end Client;

protected body Ma_Pomme is
    entry Achete_Iphone (
        Nb : in Positive ) when Nb_Iphone > 0 is
    begin
        Nb_Iphone := Nb_Iphone - Nb;
        Put_Line ("Apple: j'ai vendu" & Positive'Image (Nb) & " iPhones.");
    end Achete_Iphone;

    procedure Recoit_Iphone (
        Nb : in Positive ) is
    begin
        Nb_Iphone := Nb_Iphone + Nb;
        Put_Line ("Apple: j'ai recu" & Positive'Image (Nb) & " iPhones.");
    end Recoit_Iphone;
end Ma_Pomme;

Les_Clients : array (1 .. 5) of A_Client;

begin
    for I in Les_Clients'range loop
        Les_Clients (I) := new Client(I);
    end loop;
    Ma_Pomme.Recoit_Iphone (11);
end Ma_Pomme3;

```

Fin de la solution de la question 6 de l'exercice 1.

Exercice 2 – Train de tâches (22 points)

On souhaite constituer un programme pour représenter un train constitué d'une ou plusieurs locomotives et de wagons. Chacune de ses entités seront représentées par des tâches et l'ensemble formera un convoi. Pour cela, nous définissons dans un premier temps les types et opérations suivants :

```

-- Les constantes
|| All_Entities designe tous les elements d'un convoi. Les elements d'un convoi ne sont jamais numerotes avec cette valeur. Envoyer un
|| message a ce destinataire particulier signifie donc qu'il sera traite par tous les elements du convoi.
All_Entities : constant Natural := 0;
|| Master_Locomotive designe la locomotive de tete qui est maitre du convoi et a qui tout message d'erreur doit etre renvoye.
Master_Locomotive : constant Natural := 1;
|| Max_Entities designe le nombre maximum d'elements d'un convoi
Max_Entities : constant Natural := 100;

|| Le type decrivant l'adresse d'une entite dans le convoi
subtype Address is Integer range All_Entities .. Max_Entities;

|| Les actions associees au message
type Msg_Action is
    (Add_Locomotive, -- creer une locomotive
    Add_Wagon,      -- creer un wagon
    Error,          -- propager une erreur (vers la locomotive de tete)
    Shutdown_System); -- arreter le systeme

```

```

|| Le type des elements dans un convoi
type Convoy_Ent_Type is
    (Locomotive,
     Wagon);

|| Le type decrivant un message
type Message is
    record
        To      : Address;
        Action  : Msg_Action;
    end record;

|| Description d'une entite...
type Convoy_Entity
    (My_Id : Address;
     Etype : Convoy_Ent_Type);
type A_Convoy_Entity is access Convoy_Entity;

|| Creation d'un convoi reduit a une locomotive ayant 1 comme identite. Le convoi est represente par le pointeur sur la premiere locomotive
(il peut y en avoir plusieurs qui seront rajoutees par la suite en envoyant des messages de type Add_Locomotive)
procedure Create_Convoy (
    Convoy : out A_Convoy_Entity );

|| Envoi d'un message a un element d'un convoi. Ce message est propage vers le destinataire.
procedure Send_Message (
    Convoy : in    A_Convoy_Entity;
    M      : in    Message          );

|| Creation d'un message par assemblage de ses composantes
function Create_Message (
    To      : in    Address;
    Action  : in    Msg_Action );
return Message;

```

Le type Convoy_Entity est un type tâche possédant trois points d'entrée :

- Init qui permet de transmettre à la tâche un pointeur sur elle-même et sur son prédécesseur,
- Receive qui permet de transmettre un message à la tâche (la création d'une entité, un message d'erreur, un message d'arrêt, etc.),
- Display qui déclenche l'affichage du convoi à partir de l'entité contactée. L'affichage peut se faire dans les deux sens (vers l'arrière ou vers l'avant). Un paramètre booléen (précisant "vers l'arrière" comme valeur par défaut) permet de préciser le sens de l'affichage.

|| La tache représentant une entite du convoi (locomotive ou wagon)

```

task type Convoy_Entity (My_Id : Address ; Etype : Convoy_Ent_Type) is
    entry Init (
        Me,
        My_Prev : in    A_Convoy_Entity );
    entry Receive (
        M : in    Message );
    entry Display (
        Forward : in    Boolean := True );
end Convoy_Entity;

```

Question 1 (1 point) : Écrivez le source de la procédure Send_Message qui envoie un message au convoi à l'entité passée en paramètre.

Solution de la question 1 de l'exercice 2:

```

procedure Send_Message (
    Convoy : in    A_Convoy_Entity;
    M      : in    Message          ) is
    begin
        Convoy.all.Receive (M);
    end Send_Message;

```

Fin de la solution de la question 1 de l'exercice 2.

Pour assurer une transmission des messages dans les deux sens, chaque `Convoy_Entity` est doublement chaînée (lien vers le successeur et vers le prédécesseur). Les pointeurs sont représentés par des variables locales à la tâche incarnant l'entité du convoi (variables notées `Prev` et `Next`).

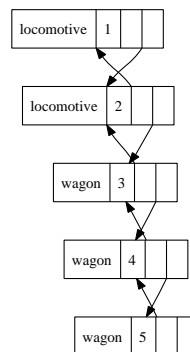
Les entités du convoi sont créées à la réception d'un Message de type `Add_Locomotive` ou `Add_Wagon`. Le wagon ou la locomotive ainsi créée (si les conditions s'y prêtent) possède un identifiant obtenu par incrémentation de l'identifiant de l'entité créatrice.

Considérons les instructions suivantes (la variable `The_Convoy` est de type `A_Convoy_Entity`) :

```
begin
  Create_Convoy (The_Convoy);
  The_Convoy.all.Receive (Create_Message(1,Add_Locomotive));
  for I in 2.. 4 loop
    The_Convoy.all.Receive (Create_Message(I,Add_Wagon));
  end loop;
end;
```

Question 2 (3 points) : Dessinez la structure de tâches construite par ces instructions. Vous indiquerez le valeur des discriminants et les liens entre entités d'un convoi.

Solution de la question 2 de l'exercice 2:



Fin de la solution de la question 2 de l'exercice 2.

Pour procéder à la construction d'un convoi, on dispose de la procédure suivante :

Creation d'une entité d'un convoi. La procedure rend un pointeur vers l'entité créée. On lui transmet le type de l'entité (locomotive ou wagon), son adresse, et un pointeur vers l'entite creatrice qui sera le predecesseur de l'entite creee.

```
procedure Create_Convoy_Entity (
  Ce      : out A_Convoy_Entity;
  Id      : in  Address;
  Ce_Type : in  Convoy_Ent_Type;
  Myself  : in  A_Convoy_Entity );
```

Cette procédure crée une nouvelle entité rendue dans `Ce` qui possède l'adresse `Id` et est de type `Ce_Type`. `Myself` est un pointeur sur une entité qui doit être affecté comme prédécesseur à la nouvelle entité créée.

Question 3 (3 points) : Écrivez la source de la procédure `Create_Convoy_Entity`.

Solution de la question 3 de l'exercice 2:

```
procedure Create_Convoy_Entity (
  Ce      : out A_Convoy_Entity;
  Id      : in  Address;
  Ce_Type : in  Convoy_Ent_Type;
  Myself  : in  A_Convoy_Entity ) is
begin
  Ce := new Convoy_Entity (Id, Ce_Type);
  Ce.all.Init (Ce, Myself);
end Create_Convoy_Entity;
```

Fin de la solution de la question 3 de l'exercice 2.

Question 4 (1 point) : Écrivez la source de la procédure `Create_Convoy`.

Solution de la question 4 de l'exercice 2:

```

procedure Create_Convoy (
    Convoy : out A_Convoy_Entity ) is
begin
    Create_Convoy_Entity (Convoy, Master_Locomotive, Locomotive, null);
end Create_Convoy;
    
```

Fin de la solution de la question 4 de l'exercice 2.

Chaque élément de convoi reçoit une adresse automatiquement attribuée lors de sa création. Les adresses sont uniques et permettent de désigner de manière non ambiguë le destinataire d'un message. On rappelle que 0 (constante `All_Entities`) est une adresse impossible – elle représente le « broadcast », c'est-à-dire le fait que tous les éléments du convoi sont destinataires de ce message.

Les adresses du convoi sont strictement croissantes au fur et à mesure que l'on se déplace vers la fin du convoi. Cela permet de déterminer le sens que doit prendre un message pour être routé correctement vers l'entité qui en est destinataire).

On s'attaque maintenant au corps d'une tâche `Convoy_Entity`. Considérons les déclarations suivantes comme locales au corps d'une tâche de ce type :

```

Local_m : Message;
Myself,
Prev,
Next    : A_Convoy_Entity := null;
    
```

|| Procédures d'envoi d'un message. Si le paramètre `Problem_Is_Fatal` vaut `True`, alors si l'envoi est impossible (le pointeur correspondant de la tâche est `null`) l'exception `Catastrophe` est levée.

```

procedure Send_Next_If_Possible (
    M           : in    Message;
    Problem_Is_Fatal : Boolean := True );
procedure Send_Prev_If_Possible (
    M           : in    Message;
    Problem_Is_Fatal : Boolean := True );
    
```

Question 5 (2 points) : Écrivez la source des procédures `Send_Next_If_Possible` et `Send_Prev_If_Possible`.

Solution de la question 5 de l'exercice 2:

Les affichages ne sont là qu'à des fins de trace. On considère dans le corrigé que la seule présence des appels de point d'entrée et des tests est primordiale.

remarque, on peut bien sûr remplacer le `Prev/Next.all.Receive (M)` ; par un appel à la procédure `Send_Message` à condition de passer les bons paramètres.

```

procedure Send_Next_If_Possible (
    M           : in    Message;
    Problem_Is_Fatal : Boolean := True ) is
begin
    if Next = null then
        if Problem_Is_Fatal then
            raise Catastrophe;
        end if;
    else
        Put_Line (Address'Image(My_Id) (2..Address'Image(My_Id)'Last) & " transmet le message
(to = " & Address'Image(M.To) &
            ", Action = " & Msg_Action'Image(M.Action)&"));
        Next.all.Receive (M);
    end if;
end Send_Next_If_Possible;

procedure Send_Prev_If_Possible (
    M           : in    Message;
    
```

```

        Problem_Is_Fatal :      Boolean := True ) is
begin
    if Prev = null then
        if Problem_Is_Fatal then
            raise Catastrophe;
        end if;
    else
        Put_Line (Address'Image(My_Id) (2..Address'Image(My_Id)'Last) & " transmet le message
(to =" & Address'Image(M.To) &
            ", Action = " & Msg_Action'Image(M.Action)&"));
        Prev.all.Receive (M);
    end if;
end Send_Prev_If_Possible;

```

Fin de la solution de la question 5 de l'exercice 2.

On considère que la structure de la tâche `Convoy_Entity` respecte le schéma suivant :

- Attente de l'appel de `Init`,
- boucle avec un `select` pour les appels de `Receive` et `Display`.

Question 6 (2 points) : Écrivez la source de l'accept du point d'entrée `Init` qui permet à une tâche de type `Convoy_Entity` de connaître son pointeur et celui du `Convoy_Entity` qui la précède.

Solution de la question 6 de l'exercice 2:

```

accept Init (
    Me,
    My_Prev : in    A_Convoy_Entity ) do Prev := My_Prev;
    Myself := Me;
end Init;

```

Fin de la solution de la question 6 de l'exercice 2.

On considère les contraintes suivantes sur l'adressage des éléments d'un convoi :

- Le premier élément du convoi (forcément une locomotive) possède la plus petite adresse du convoi,
- Le convoi peut comporter plusieurs locomotives mais on considèrera qu'elles se situent toutes en tête de convoi. Dans ce cas, la locomotive de tête est considérée comme le pilote de l'ensemble du convoi et elle-seule peut prendre une décision de terminaison du système
- Important : Toute violation des règles de constitution d'un convoi aboutissent à l'envoi d'un message de type `Error` vers la locomotive de tête qui procède alors à la destruction du système en faisant une diffusion ("broadcast") en un seul message à destination de toutes les entités du convoi.

Question 7 (6 points) : Écrivez la source des actions exécutées par une tâche lorsqu'elle est appelée sur le point d'entrée `Receive` qui applique les commandes associées aux différents messages du système (type `Msg_Action`). C'est ici que seront traitées toutes les contraintes qu'un convoi doit respecter. C'est également dans ce point d'entrée que vous devez procéder au routage des messages.

Solution de la question 7 de l'exercice 2:

```

accept Receive (
    M : in    Message ) do Local_m := M;
end Receive;
delay 0.1 * Duration (Calcule_Attente);-- pour tester le sujet uniquement
if My_Id = Local_m.To or Local_m.To = 0 then
    case Local_m.Action is
        when Add_Locomotive =>
            Put_Line (Address'Image(My_Id) (2..Address'Image(My_Id)'Last) & " procede
a la creation d'une locomotive");
        if Etype /= Locomotive then
            || une locomotive suit forcement une locomotive
            raise Catastrophe;
        else
            Create_Convoy_Entity (Next,My_Id+1, Locomotive, Myself);
        end if;
    end if;
end if;

```

```

        when Add_Wagon =>
            Put_Line (Address'Image(My_Id) (2..Address'Image(My_Id)'Last) & " procede
a la creation d'un wagon");
            Create_Convoy_Entity (Next,My_Id+1, Wagon, Myself);
        when Error =>
            if My_Id = Master_Locomotive then
                || seule la locomotive de tete est concernee par le traitement des erreurs
                Put_Line (Address'Image(My_Id) (2..Address'Image(My_Id)'Last) & " genere
un message de shutdown");
                Next.all.Receive (Create_Message(All_Entities,Shutdown_System));
                exit;
            else
                -- il faut propager le message d'erreur vers la locomotive de tete
                Send_Prev_If_Possible (Local_m);
            end if;
        when Shutdown_System =>
            || on propage le message au suivant puis on se termine;
            Put_Line (Address'Image(My_Id) (2..Address'Image(My_Id)'Last) & " se termine
sur demande");
            Send_Next_If_Possible (Local_m, False);
            exit;
        end case;
    elsif Local_m.To > My_Id then
        || envoyer le message au suivant
        Send_Next_If_Possible (Local_m);
    else
        || envoyer le message au precedent
        Send_Prev_If_Possible (Local_m);
    end if;
end if;

```

Attention, dans cette programmation, l'exception Catastrophe est rattrapée et génère l'envoi d'un message d'erreur vers la locomotive de tête. Il n'est pas fait mention de cette exception aux étudiants pour ne pas les perturber...

Fin de la solution de la question 7 de l'exercice 2.

L'affichage réalisé via l'entrée Display se présente de la manière suivante :

```

LOCOMOTIVE (1) >-< LOCOMOTIVE (2) >-< WAGON (3) >-< WAGON (4) >-< WAGON (5) >-<
>-< WAGON (5) >-< WAGON (4) >-< WAGON (3) >-< LOCOMOTIVE (2) >-< LOCOMOTIVE (1)

```

La première ligne est un affichage "vers l'arrière" et la seconde correspond à un affichage "vers l'avant".

Question 8 (4 points) : Écrivez le source de l'accept du point d'entrée Display. *Important : on considère que cet appel se fait toujours à partir de la locomotive de tête.*

Solution de la question 8 de l'exercice 2:

```

    accept Display (
        Forward : in Boolean := True ) do if Forward then Put (
(My_Id)'Last) & " ) >-< " );
        if Next /= null then
            Next.all.Display (Forward);
        else
            || c'est la fin du convoi
            New_Line;
        end if;
        else
            if Next /= null then
                Next.all.Display (Forward);
            end if;
            Put (">-< " & Convoy_Ent_Type'Image (Etype) & " (" & Address'Image(My_Id) (2..Address'Image
& " ) ");
        end if;
        if Prev= null then
            || c'est le debut du convoi
            New_Line;
        end if;
    end Display;

```

Fin de la solution de la question 8 de l'exercice 2.