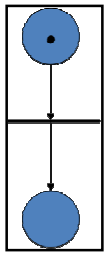
Université Toulouse III – Paul Sabatier
118 route de Narbonne
31062 Toulouse cedex 9

Travaux dirigés – n°1 – Réseaux de Petri

Quelques rappels

Un réseau de Petri (RdP) est un moyen de :

- modéliser le comportement des systèmes dynamiques à événements discrets ;
- décrire des relations existantes entre des conditions et des événements.

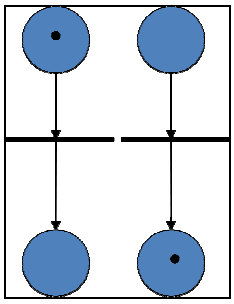


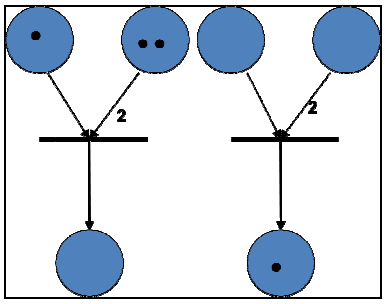
Un RdP est un graphe orienté composé de places et de transitions :

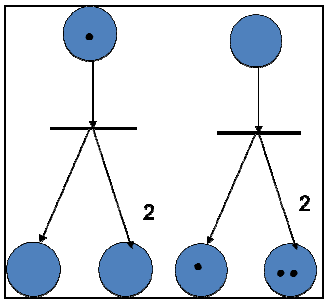
- Une place est représentée par un cercle ;
- Une transition est représentée par un trait ;
- Un arc relie soit une place à une transition, soit une transition à une place.

Chaque place contient un nombre entier positif ou nul de jetons.

Une transition est armée (ou franchissable) lorsque toutes les places qui sont en entrée de cette transition contiennent au moins le nombre de jetons spécifié par le poids de l'arc d'entrée correspondant. Le franchissement consiste alors à retirer les jetons de chacune des places d'entrée et à rajouter un ou plusieurs jetons à chacune des places de sortie de la même transition, selon le poids des arcs de sortie.







Remarques :

- Ces deux actions se font simultanément ; le franchissement d'une transition n'est pas divisible.
- On peut remarquer qu'il n'y a pas conservation du nombre de jetons.
- Lorsqu'une transition est validée, cela n'implique pas qu'elle soit franchie immédiatement. Cette validation n'est qu'une possibilité de progression à cet instant là.
- Lorsque deux arcs sortent d'une place et que les transitions correspondantes sont validées, le choix de la transition qui sera franchie est arbitraire (indéterminisme).

Modélisation d'une exécution séquentielle d'actions

Produire la portion de RdP correspondant aux schémas d'exécution suivants :

☞ Exécution de trois actions A1, A2 et A3 en séquence.

☞ N répétitions d'une séquence de deux actions A1 et A2.

Modélisation du parallélisme

Produire le RdP correspondant au schéma d'exécution suivant :

☞ Lancement de deux séquences d'actions (A1, A2, A3) et (B1 et B2) en parallèle puis attente de leurs terminaisons.

Modélisation d'une activité

Produire le RdP correspondant à une activité réalisant la réception d'un message à partir du réseau puis son traitement par deux tâches en parallèle.

- La première tâche est constituée de la séquence : A1 et A2 ;
- La seconde tâche est constituée de la séquence : B1 et B2.

Les contraintes suivantes doivent être respectées :

- L'action B2 ne peut pas commencer avant la fin de l'action A1.
- Un nouveau message ne peut pas être admis tant qu'il y a un traitement en cours (session en cours).

Modélisation d'un affichage concurrent

On considère une première activité réalisant un cycle infini constitué de la séquence : T1, A1, A2, T2 et une seconde activité qui réalise un cycle infini constitué de la séquence : T3, A3, A4, T4 où l'action Ti consiste à effectuer un certain traitement et l'action Ai à afficher une portion de message à l'écran.

☞ Produire le RdP permettant à ces deux activités d'afficher respectivement le message produit par (A1, A2) et le message produit par (A3, A4).

☞ Produire le RdP permettant à ces deux activités d'afficher les messages produits par (A1, A2) et par (A3, A4) de manière alternée à l'écran.

Travaux dirigés – n°2 – Sémaphores

Quelques rappels

Un sémaphore S encapsule une variable entière non négative et une file d'attente. Un sémaphore S est muni de 3 opérations que peuvent utiliser des processus :

init(S, n)	Initialise la valeur de la variable de S au nombre n donné
P(S)	**Si** la valeur de la variable de S est nulle **alors** bloque le processus dans la file d'attente associée à S ; Décrémente la valeur de la variable de S de 1
V(S)	Incrémente la valeur de la variable de S de 1 ; **Si** des processus sont en attente **alors** réveille le premier processus de la file d'attente associée à S

On peut considérer de manière plus intuitive un sémaphore comme une corbeille dans laquelle seraient placés des jetons.

Pour être autorisé à poursuivre, un processus utilise l'opération P (« - Puis-je ? ») qui tenterait de retirer un jeton dans la corbeille à condition qu'il y ait des jetons présents. Un processus peut déposer un jeton dans la corbeille ou bien le transmettre à un processus qui attendrait un tel jeton par l'opération V (« - Vas-y »).

Exercice 1 – Alternance d'affichage

Un processus Afficheur possède le comportement suivant :

```
Processus Afficheur {
  while (1) {
    Effectuer un traitement ;
    Afficher un message à l'écran ;
    Effectuer un traitement ;
  }
}
```

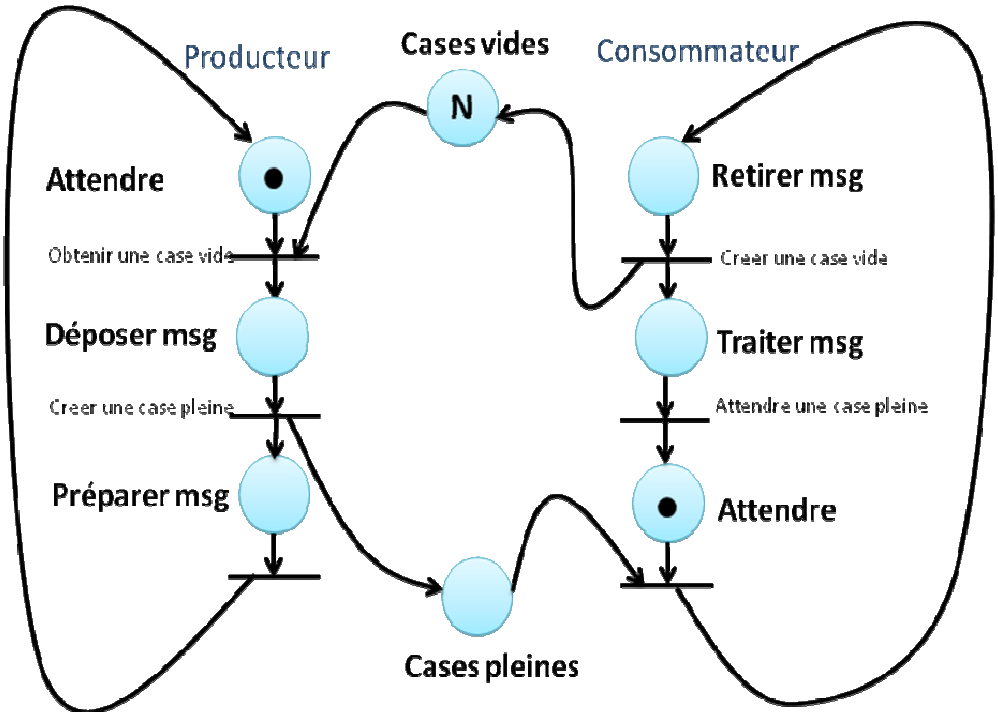
☞ Proposer une solution utilisant des sémaphores pour synchroniser l'affichage ***alterné*** à l'écran de deux processus Afficheur. Ce problème a été modélisé par un réseau de Petri lors du TD précédent.

☞ Comment faut-il modifier la solution précédente pour ***généraliser*** le problème à N processus Afficheur ?

Exercice 2 – Gestion de ressources : Modèle des producteurs/consommateurs

On se propose d'étudier plusieurs variantes du modèle des producteurs/consommateurs présenté en cours. Dans ces variantes, on suppose que plusieurs processus producteurs et plusieurs processus consommateurs coexistent et veulent accéder à un tampon, partagé, de N cases, gérées de manière circulaire, pour y déposer ou retirer un message.

Le réseau de Petri ci-après modélise la variante de base étudiée en cours :



Dans un but de réutilisation, commencer par écrire les opérations permettant l'insertion et l'extraction proprement dites d'un message dans le tampon.

Variante 1

Dans la première variante étudiée, les messages peuvent être de deux types différents (exemples : blanc/noir ou recto/verso).

Les producteurs sont différenciés par le type du message qu'ils déposent dans le tampon. Les demandes des producteurs sont traitées dans l'ordre de leur arrivée mais on désire que les dépôts dans le tampon se fassent de manière alternée selon le type des messages déposés.

Les consommateurs retirent les messages dans l'ordre des messages déposés.

☞ Proposer une solution utilisant des sémaphores pour que ces processus déposent et retirent leurs messages de manière cohérente.

Variante 2

Dans cette variante, deux types de messages existent toujours.

Les producteurs sont différenciés par le type des messages qu'ils déposent dans le tampon. Contrairement à la variante précédente, les dépôts dans le tampon se font dans l'ordre des arrivées, quel que soit le type du message déposé.

En revanche, les consommateurs précisent quel type de message ils désirent retirer. Les messages sont toujours retirés dans l'ordre des dépôts et les demandes des consommateurs sont traitées dans l'ordre de leur arrivée.

Travaux dirigés – n°3 – Sémaphores

Gestion d'une voie unique

On considère des véhicules circulant sur des voies parallèles, mais avec parfois des tronçons à voie unique. Bien entendu, il ne peut y avoir deux véhicules circulant en sens inverse sur une telle voie unique.

Le comportement d'un véhicule, circulant sur ces voies et caractérisé par son sens de déplacement (vers l'est ou l'ouest), est le suivant :

```
Processus Vehicule (monSens) {
    . . . . .
    Circuler sur la voie à double sens ;
    /* Demander à s'engager sur la voie unique (dans un certain sens) */
    DemanderAccesVU(monSens) ;
    Rouler sur la voie unique ;
    /* Quitter la voie unique */
    LibererAccesVU() ;
    Continuer à circuler sur le voie à double sens ;
}
```

Plusieurs variantes sont envisageables pour synchroniser l'accès à la voie unique.

Variante 1

Il ne peut y avoir qu'un seul véhicule sur la voie unique.

Variante 2

Le nombre de véhicules à un instant donné sur la voie unique est illimité.

Variante 3

Le nombre de véhicules à un instant donné sur la voie unique est limité à *MAX*.

☞ Pour chacune de ces variantes, proposer une solution utilisant des sémaphores pour synchroniser l'accès à cette voie unique (cela revient à écrire `DemanderAccesVU(monSens)` et `LibererAccesVU()`).

Travaux dirigés – n°4 – Moniteurs de Hoare

Le modèle des producteurs-consommateurs

On considère à nouveau le modèle des producteurs-consommateurs mais on souhaite réaliser la synchronisation nécessaire grâce à un moniteur de Hoare.

Dans ce modèle de synchronisation, deux familles de processus accèdent à un buffer partagé pour y déposer (processus Producteurs) ou retirer des messages (processus Consommateurs). Le buffer est géré de manière circulaire. Les retraits s'effectuent dans l'ordre des dépôts.

Le comportement de ces processus est donc le suivant :

Un producteur :

```
Début
   ...
   Déposer un message dans le buffer
                  (à la suite de ceux qui s'y trouvent déjà);
   ...
Fin;
```

Un consommateur :

```
Début
   ...
   Retirer le message du buffer (le plus ancien);
   ...
Fin;
```

On se propose d'écrire un moniteur de Hoare gérant l'accès à la ressource commune selon la politique définie précédemment et selon les différentes variantes suivantes.

La spécification du moniteur se présente de la manière suivante:

```
Moniteur Prod_Conso {
  void Deposer(. . . );
  void Retirer(. . . );
end Prod_Conso ;
```

Variante 1 – Buffer de N cases, messages d'un type unique

C'est la variante de base, dans laquelle la capacité du buffer est de N messages et la politique appliquée est celle décrite plus haut.

Variante 2 – Message de deux types, dépôts alternés

Dans cette variante, les messages considérés peuvent être de deux types (par exemple, noir/blanc ou recto/verso…). Un producteur dépose des messages d'un certain type. Les dépôts des messages se font de manière alternée. Les retraits se font selon la même politique que précédemment.

Variante 3 – Messages de deux types, choix du type de message retiré

Dans cette variante, on a toujours des producteurs de deux types mais les dépôts ne sont plus forcément alternés. On a aussi des consommateurs de deux types et un consommateur spécifie le type du message qu'il désire retirer. Les retraits se font toujours dans l'ordre des dépôts.

Questions

Pour chacune des variantes :

- ☞ Donner la spécification du moniteur.
- ☞ Préciser les conditions de blocage et de réveil d'un processus producteur et d'un processus consommateur.
- ☞ En déduire les variables d'état et les variables « condition » gérées dans le moniteur.
- ☞ Donner le code du moniteur.

Travaux dirigés – n°5 – Moniteurs de Hoare

Le modèle des lecteurs-rédacteurs

Le modèle des lecteurs-rédacteurs schématise une situation rencontrée dans la gestion de fichiers partageables ou dans l'accès à des bases de données.

Deux familles de processus accèdent à ces informations. Les lecteurs désirent seulement consulter l'information. Les rédacteurs désirent modifier cette information. Les lecteurs peuvent donc accéder à l'information en parallèle alors que les rédacteurs doivent y accéder en exclusion mutuelle.

Les comportements de ces processus sont donc les suivants :

Un lecteur:

```
Début
    ...
    Demander à lire;
    Faire la lecture;
    Signaler la fin de lecture;
    ...
Fin;
```

Un rédacteur:

```
Début
    ...
    Demander à écrire;
    Faire la modification;
    Signaler la fin d'écriture;
    ...
Fin;
```

On se propose d'écrire un moniteur de Hoare gérant l'accès à la ressource commune selon la politique définie précédemment et selon les différentes variantes suivantes.

La spécification du moniteur se présente de la manière suivante:

```
Moniteur Lecteur_Redacteur {
  void Debut_Lire();
  void Fin_Lire();
  void Debut_Ecrire();
  void Fin_Ecrire();
end Lecteur_Redacteur ;
```

Variante 1

Quand aucun lecteur ne lit, les lecteurs et les rédacteurs ont la même priorité. En revanche, dès qu'un lecteur est en train de lire, tous les autres lecteurs qui le demandent peuvent également lire puisque les lectures peuvent être faites en parallèle, quel que soit le nombre de rédacteurs en attente.

Lorsqu'un rédacteur écrit, aucun autre client ne peut accéder à la ressource (ni lecteur, ni rédacteur).

Lorsque le rédacteur a terminé d'écrire, il essaye d'activer un rédacteur en priorité sur les lecteurs.

Variante 2 - Priorité des rédacteurs sur les lecteurs

On souhaite donner la priorité aux rédacteurs afin que l'information disponible ne soit pas obsolète pour le lecteur. Lorsqu'un rédacteur demande à accéder à la ressource, il doit donc l'obtenir le plus tôt possible. Bien sûr, il ne lui est pas possible d'interrompre des lectures ou une écriture en cours. De même, il n'a pas de passe-droit vis-à-vis d'autres rédacteurs en attente (arrivés avant lui et non encore acceptés). En revanche, il est prioritaire par rapport aux lecteurs en attente.

Variante 3 - Gestion plus équitable des accès

À quelles situations erronées peuvent conduire les variantes 1 et 2 ?

Quelle solution proposer pour corriger de telles situations ?

Dans cette variante, un rédacteur qui termine d'écrire doit laisser l'accès en priorité à tous les lecteurs en attente à cet instant, et non au rédacteur suivant comme il est spécifié dans la variante 1. En revanche, les lecteurs qui arriveront ultérieurement (après cette demande d'écriture) devront respecter la règle de priorité des rédacteurs sur les lecteurs telle qu'elle est exprimée dans la variante 2.

Variante 4 - Gestion FIFO des accès

On suppose maintenant que l'accès aux données est effectué suivant une politique FIFO ; les requêtes sont traitées dans l'ordre de leurs arrivées.

Pour ordonner globalement les lecteurs et les rédacteurs en attente, tous les processus seront bloqués sur une même condition. En effet, il n'est pas possible de savoir si le 3^e rédacteur est arrivé avant ou après le 4^e lecteur lorsque lecteurs et rédacteurs sont rangés dans des files distinctes. On notera que, lors du réveil, il n'est pas possible (pour le « signaleur ») de distinguer un lecteur d'un rédacteur. Aussi, on peut être amené à réveiller un processus (lecteur ou rédacteur), puis le rebloquer par la suite si le déblocage s'avère impossible dans la situation actuelle.

Questions

Pour chacune des variantes :

- ☞ Donner la spécification du moniteur.
- ☞ Préciser les conditions de blocage et de réveil d'un processus lecteur et d'un processus rédacteur.
- ☞ En déduire les variables d'état et les variables « condition » gérées dans le moniteur.
- ☞ Donner le code du moniteur.

Travaux dirigés – n°6 – Moniteurs de Hoare

Traitement de commandes

A l'occasion des fêtes de Noël, votre magasin préféré a décidé de mettre à la disposition de ses fidèles clients, un certain nombre (NB_GUICHETS) de guichets spéciaux pour prendre en charge leurs commandes. Le principe de fonctionnement de ces guichets est le suivant :

- Un client attend un guichet libre, y dépose sa commande, puis attend que celle-ci soit prête.
- Le traitement d'une commande comporte un certain nombre (NB_ETAPES) d'étapes successives et des employés spécialisés sont affectés à chacune de ces étapes. L'étape i du traitement d'une commande ne peut débuter que lorsque l'employé chargé de l'étape précédente (i-1) a terminé sa tâche.
- Lorsque sa commande a été traitée, le client quitte le guichet, pleinement satisfait de ce nouveau service.

Par exemple, en pratique, le magasin met trois guichets à disposition de ses clients et le traitement d'une commande consiste en quatre étapes : (1) établir la liste des articles commandés, (2) aller chercher les articles commandés, (3) les emballer avec du papier cadeau et (4) faire payer le client.

Problème

On considère deux types de processus : Client et Employe.

- Un processus Client dépose une commande à un guichet libre et attend que cette commande soit traitée avant de quitter ce guichet. Au plus NB_GUICHETS commandes peuvent être traitées en parallèle.
- Un processus Employé, spécialisé dans l'étape i, prend en charge la première commande en attente de cette étape, accomplit sa part de travail avant de rapporter la commande traitée à son guichet d'origine. Il peut alors prendre en charge une nouvelle commande. Lorsque la dernière étape a été exécutée sur une commande, le processus Client ayant formulé cette commande peut reprendre son exécution.

On suppose définis les types :

- NumeroEtape : qui désigne un entier compris entre 0 et NB_ETAPES
- NumeroGuichet : qui désigne un entier compris entre 1 et NB_GUICHETS
- Commande : qui désigne la commande établie par un client

On suppose aussi donné, le sous-programme :

- void appliquerEtape (NumeroEtape numEtape, Commande *uneCommande) ;

qui peut être utilisé par un processus Employe afin d'accomplir sa part de travail (i.e. l'étape numEtape) sur une commande donnée et ainsi lui apporter une modification.

On se propose de synchroniser, en utilisant un moniteur de Hoare nommé GestionRequetes, des processus Client et des processus Employé pour que les commandes soient traitées le plus efficacement possible.

La spécification de ce moniteur est la suivante :

```
Moniteur GestionRequetes {
        void commander (void) ;                 // Utilisé par un Client
                                                // Utilisés par un Employe
        void commencerEtape (NumeroEtape etape, Commande *cmde, NumeroGuichet *guichet) ;
        void terminerEtape (Commande cmde, NumeroGuichet guichet) ;
}
```

Le comportement des processus Client et Employe est donc le suivant :

```
Processus Client {
        // Se rendre au magasin
        GestionRequetes.commander() ;
        // Rentrer chez soi, satisfait
}
```

```
Processus Employe (NumeroEtape etapeAppliquee) {
        while (1) {
        GestionRequetes.commencerEtape(etapeAppliquee,
                        &cmdeATraiter,  &guichetOrigine) ;
        appliquerEtape(etapeAppliquee, cmdeATraiter) ;
        GestionRequetes.terminerEtape(cmdeATraiter,
                        guichetOrigine) ;
        }
}
```

Questions

- ☞ Préciser les conditions de blocage et de réveil d'un processus Client et d'un processus Employe.
- ☞ En déduire les variables d'état et les variables « condition » gérées dans le moniteur.
- ☞ Donner le code du moniteur.