Master d'informatique | **module PC2R - MI019** | **Université Pierre et Marie Curie**
Documents autorisés | *Durée 2 heures* | **Année 2011-2012**

Examen de rattrapage du 31 janvier 2012

Exercice 1 : station d'eau en Esterel

Une station d'eau est composée d'un bassin, de 2 pompes de remplissage et d'une pompe de vidange. Tout le système fonctionne en continu.

Le bassin
- est vide au départ.
- a un contenu limité par sa constante `CAPACITE`.
- peut être démarré ou arrêté autant de fois que l'on veut (voir les signaux `DEMARRER` et `ARRET` du module `Bassin`).
- est géré par une variable `x` indiquant à chaque instant son contenu.
- doit réactualiser sa variable `x` à la réception des signaux `REMPLIR_VIDER` valués émis par des pompes.
- émet à chaque instant le signal `QUANTITE` valué contenant la valeur de `x`.
- émet le signal `STOP_SECURITE` lorsque son contenu dépasse sa `CAPACITE` de 10%. Ce signal permettra au module principal d'arrêter complètement le système.

Chaque pompe
- est au départ à l'arrêt.
- est indépendante (voir les signaux `DEMARRER` et `ARRET` du module Pompe qui permettent de démarrer ou d'arrêter chaque pompe indépendamment).
- peut être démarrée ou arrêtée autant de fois que l'on veut.
- caractérisée par sa constante `VALEUR` positive (resp. négative) pour une pompe de remplissage (resp. vidange). C'est la quantité de liquide que la pompe peut remplir ou vidanger à chaque tick.
- émet à chaque tick un signal `REMPLIR_VIDER` valué contenant la valeur `VALEUR`.

Le système principal
- contient un bassin de `CAPACITE` 10, 2 pompes de remplissage de `VALEUR` 1 et une pompe de vidange de `VALEUR` -1 qui fonctionnent en parallèle.
- attend le signal `DEMARRER` donné par l'utilisateur pour démarrer le bassin et les pompes.
- arrête le remplissage (resp. le vidange) si le bassin est plein (resp. vide).
- démarre dès que possible le remplissage ou le vidange.
- arrête tout le système à la réception du signal `STOP_SECURITE` ou `ARRET`.

Vous pouvez (devez) ajouter le ou les variables ou signaux que vous jugez nécessaires. On évitera toutes attentes actives.

1. Compléter le module `Bassin` permettant de répondre au comportement énoncé ci-dessus.
2. Compléter le module `Pompe` permettant de répondre au comportement énoncé ci-dessus.
3. Compléter le module principal `Examen_Rattrapage_Esterel` permettant de répondre au comportement énoncé ci-dessus.

Modules	Exécution

`module Bassin :`
 	
`constant CAPACITE : integer;`
`input ARRET, DEMARRER;`
`output QUANTITE : integer;`
`output STOP_SECURITE;`
 	
`% ????????????????????????????????? %`
 	
`end module`
 	
`module Pompe :`
 	
`constant VALEUR : integer;`
`input ARRET, DEMARRER;`
 	
`% ????????????????????????????????? %`
 	
`end module`
 	
`module Examen_Rattrapage_Esterel :`
 	
`input ARRET, DEMARRER;`
`output QUANTITE : integer;`
`output STOP_SECURITE;`
 	
`% ????????????????????????????????? %`
 	
`end module`
	`%{  Et un exemple d'execution`
	
	`$ Examen_Rattrapage_Esterel`
	`Examen_Rattrapage_Esterel> ;;`
	`--- Output:`
	`--- Output:`
	`Examen_Rattrapage_Esterel> DEMARRER;;;;;;;;;;`
	`--- Output: QUANTITE(1)`
	`--- Output: QUANTITE(2)`
	`--- Output: QUANTITE(3)`
	`--- Output: QUANTITE(4)`
	`--- Output: QUANTITE(5)`
	`--- Output: QUANTITE(6)`
	`--- Output: QUANTITE(7)`
	`--- Output: QUANTITE(8)`
	`--- Output: QUANTITE(9)`
	`--- Output: QUANTITE(10)`
	`--- Output: QUANTITE(9)`
	`Examen_Rattrapage_Esterel> ;;;;;;;;;;`
	`--- Output: QUANTITE(10)`
	`--- Output: QUANTITE(9)`
	`--- Output: QUANTITE(10)`
	`--- Output: QUANTITE(9)`
	`--- Output: QUANTITE(10)`
	`--- Output: QUANTITE(9)`
	
	`...`

Exercice 2 : serveur d'amphithéâtre virtuel (langage libre)

Pour des examens en ligne, on cherche à implanter un amphithéâtre virtuel pour le passage d'épreuves. Cet amphithéâtre est géré par un serveur et accessible par des clients correspondant aux participants de l'épreuve.

Une épreuve est composée d'une phase d'installation, d'une phase de distribution du sujet, d'une phase de composition et d'une phase de remise des copies.

Pour cela le serveur doit gérer les tâches suivantes :
- ouvrir les portes de l'amphithéâtre ;
- autoriser l'installation des participants en leur fournissant un identifiant unique ;
- fermer les portes ;
- diffuser le sujet ;
- indiquer à certains intervalles le temps restant pour l'épreuve ;

– accepter la remise d'une copie ;
– accepter à partir d'un certain temps la sortie d'un participant ;
– clôre l'épreuve.

Un tel serveur peut être caractérisé par le nombre de participants possibles. Chaque participant possède un identifiant unique. Une épreuve a une certaine durée, le temps minimal avant de partir est la moitié de la durée de l'épreuve. Pour simplifier la diffusion du sujet et le rendu des copies, on effectuera ces opérations par une seule communication. On supposera un temps discret, et connues les différentes fonctions le manipulant. De même on supposera les connexions fiables et les participants non malveillants.

Il vous est demandé de construire un tel serveur

1. Définir un protocole (celui-ci pourra s'étendre en cas de besoin au fur et à mesure des questions) indépendant du langage permettant d'indiquer l'ouverture des portes, l'installation d'un participant, la fermeture des portes, la diffusion du sujet, la remise des copies, la sortie d'un participant et la clôture de l'épreuve. Répondez à cette question avant de passer aux suivantes.
2. Implanter dans le langage de votre choix (O'Caml, Java ou C) un serveur gérant un tel amphithéâtre virtuel en respectant le protocole décrit. Vous pouvez, en les citant, utiliser des morceaux de code issus du polycopié. Chaque participant (client) communiquera avec le serveur dans en mode connecté et fiable (TCP/IP) et sera une fois installé géré par un thread particulier. Lors de l'installation un identifiant unique sera fourni au participant. Préciser comment vous le construisez. A cette question on ne s'intéressera pas aux questions de temps.
   (a) Au début de cette question indiquer l'architecture logicielle que vous allez employer, et indiquer le type des méthodes/fonctions principales de votre développement.
   (b) Écrire la partie du serveur correspondant à l'ouverture de l'amphithéâtre, l'accueil des participants, et la fermeture des portes (la fin de l'acceptation de nouveaux participants).
   (c) Écrire ensuite la partie épreuve correspondant à la diffusion du sujet à l'ensemble des participants, la remise d'une copie et la sortie d'un participant. Il est autorisé de remettre sa copie plusieurs fois mais c'est la dernière soumission qui compte. Pour cette question on suppose que l'épreuve est finie quand tous les participants sont partis ce qui clôt l'épreuve. A la fin de l'épreuve le serveur conservera dans uns structure de données adéquate l'ensemble des couples (identifiant,copie) des participants. Si un participant part sans soumettre sa copie, celle-ci sera considérée comme vide.
3. On cherche maintenant à gérer le temps de l'épreuve.
   (a) Après la moitié du temps de l'épreuve un participant est autorisé à partir. Implanter cette fonctionalité.
   (b) L'épreuve, une fois le sujet diffusé, dure un certain temps. Le serveur rappelle à certains intervalles de temps (à chaque 1/10 du temps total) le temps qu'il reste pour l'épreuve en diffusant à l'ensemble des participants connectés cette information. Ainsi l'épreuve est finie soit quand tous les participants ont remis leur copies, soit quand le temps imparti est écoulé. Implanter cette fonctionalité.

Exercice 3 : jeu des 100 familles en RMI

Le but de cet exercice est d'implanter un jeu de cartes en utilisant entre autres les technologies RMI.

Le jeu choisi sera le jeu de 100 familles.

C'est un jeu tour par tour où chaque joueur essaye de compléter le plus possible de familles de cartes. Pour ce faire, quand c'est au tour du joueur, celui-ci demande à un autre joueur une carte. Si ce dernier possède la carte demandée il la donne et le joueur demandeur peut réitérer une nouvelle demande *possiblement à un autre joueur*. Sinon, si le joueur à qui on demande une carte ne la possède pas, il répond *pioche* et le joueur demandeur pioche une carte dans une pile de cartes et finit son tour. Si la pioche est vide, le joueur finit son tour. Lorsqu'un joueur a toute une famille il le fait savoir et retire cette famille du jeu. Une carte d'une famille sera identifiée par deux `int` son numéro de fammille, de 0 à 99 et son numéro de carte de 0 à 7.

On utilisera un objet réparti pour gérer la partie, cet objet sera de classe `partie` et disposera des fonctions suivantes :

- `int connect()` qui renvoie au joueur son numéro, les numéros commencent à partir de 1
- `int[] play(int)` qui renvoie au joueur dont on donne le numéro le tableau de ses cartes (voir `pioche` pour le codage de celles ci).
- `bool ask(int joueur, int famille , int carte)` utilisée pour demander une carte à un joueur, et qui renvoie `vrai` ssi le joueur désigné a la carte demandée.
- `int getrequest()` permettant de récuperer le numéro de la carte demandée.
- `void answer(bool reponse)` permettant de répondre à une requête.
- `int wakeup()` qui est une fonction *bloquante* permettant d'attendre une notification. La valeur de retour correspond à un numéro de joueur et indique à celui-ci ce qu'il doit faire : si la valeur est un chiffre positif le joueur concerné doit poser une question, si la valeur est négative, le joueur dont le numéro est l'opposé doit répondre. Si la valeur de retour est 0 la partie est finie et le joueur doit appeler la fonction `int gagnant()` pour savoir qui est vainqueur. Dans tous les autres cas le joueur doit appeler `wakeup`.
- `void famille(int numero)` permettant d'incrémenter le compteur de famille du joueur.
- `int retour pioche()` retourne -1 si la pioche est vide, et sinon `10 * numéro famille + numéro carte`.
- la fonction `int gagnant()` renvoie le numéro du joueur ayant le plus de familles.

Pour faire une partie, les joueurs se connectent, quand 5 joueurs sont connectés les cartes sont mélangées et chaque joueur prend ses cartes, les cartes restantes sont mises dans une file (la pioche). Ensuite l'objet partie décide que ce sera le joueur 1 qui débute. La partie se termine lorsque qu'un joueur n'a plus de cartes.

1. Ecrire l'interface `Iremotepartie` de la classe `partie`.
2. Donner une définition des champs de la classe `partie` dont vous aurez besoin pour gérer la partie (quels tableaux quels entiers...) et donner une implantation de la fonction `bool ask(int joueur, int famille , int carte)` et des constructeurs de cette classe.
3. Expliciter comment implanter le comportement de `wakeup` à l'aide de `synchronized`, `wait` et `notify(all)` (dire aussi pourquoi on est fondé à les utiliser !) et donner une définition de la fonction `wakeup`.
4. Donner une implantation de la fonction `connect`, que l'on souhaite bloquante tant que moins de 5 joueurs se sont connectés. On prendra garde aux probleme d'accès concurrents !
5. En supposant que l'objet `toto` est disponible sur le registry `192.168.1.234` donner une implantation d'un joueur qui demandera à l'utilisateur les valeurs des cartes à demander mais qui répondra tout seul à une requête.
6. Donner une implantation de la fonction `gagnant`. Y a t'il un problème de concurence ? L'expliciter si besoin.
7. A chaque tour tous les joueurs sont réveillés deux fois même s'il n'ont rien à faire ! Proposer un paliatif à ce problème.