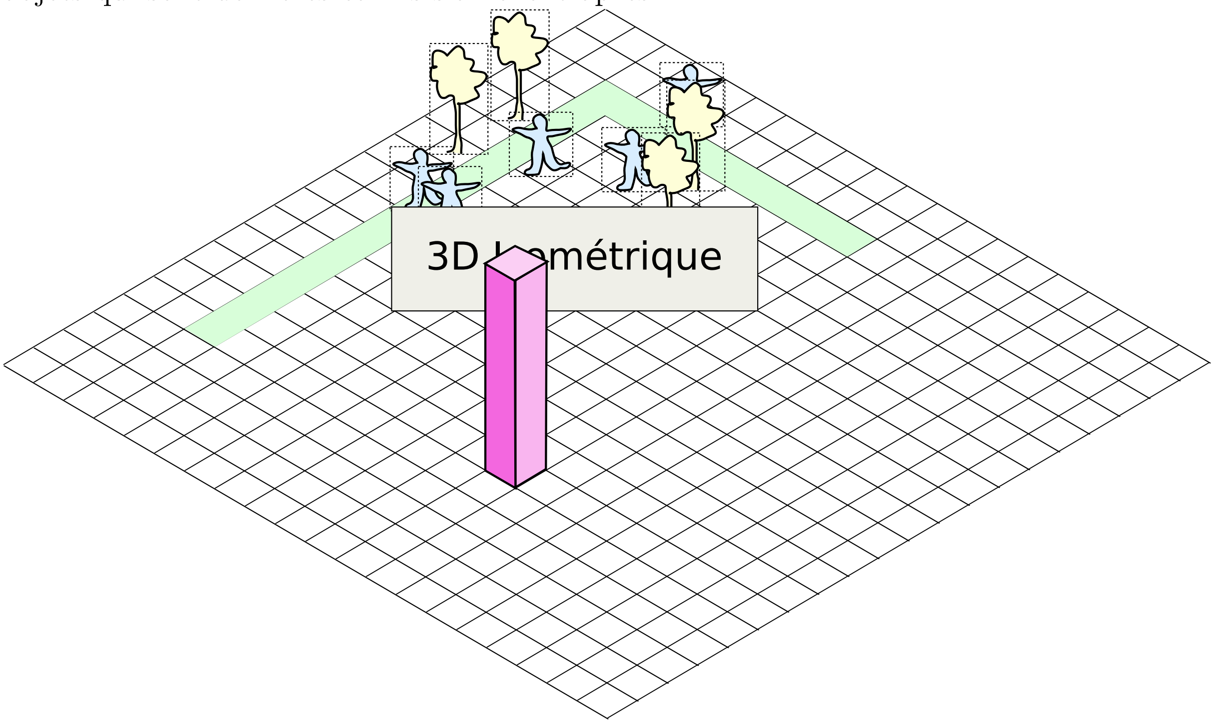
Examen 2007/2008 Temps Réel

Le 11/6/2008, Tout document autorisé
Durée : 1 heure et demie

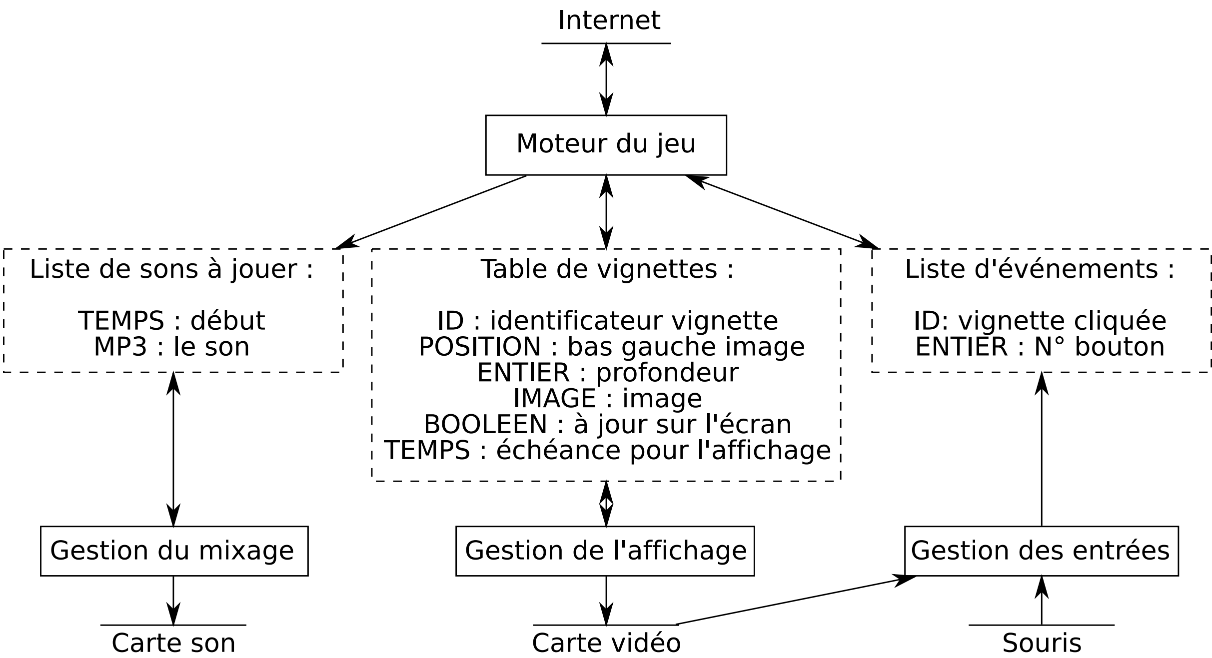
Considérons un jeu 2D isométrique multi-joueur sur Internet.

L'affichage en 2D isométrique est une simple superposition d'images les unes par dessus les autres (comme dans Simcity, Dofus...). On affiche sur l'écran d'abord ce qui est loin puis ce qui est près. L'impression de 3D est notamment du au fait que les objets qui sont devant cachent les objets qui sont derrières car ils s'affichent après.



L'affichage complet de l'image sur l'écran est impossible à faire 60 fois par seconde car la machine n'est pas assez rapide. Il faut donc **afficher seulement la zone de l'écran qui a changé**, c'est beaucoup plus rapide, il y a seulement quelques petits personnages qui se déplacent sur l'écran et peut être des éléments graphiques qui changent dans l'interface utilisateur (temps, points de vie...).

La figure suivante présente l'architecture du jeu lui-même. Il y a 4 processus (rectangles avec le trait plein) et 3 zones de mémoire partagée (rectangles pointillés).



1. On pourrait faire cette application en utilisant un seul processus. *Donnez les avantages et inconvénients de cette approche.*

2. Les deux listes sont particulières, un seul processus ajoute des éléments dedans et un seul en enlève. *Proposez une implémentation de ces listes qui tienne compte de ce fait. Prouvez que votre implémentation fonctionne.*

3. La table des vignettes contient le modèle de ce que l'on veut afficher sur l'écran. Le booléen indique si l'image sur l'écran est à jour pour la vignette considérée (on a eu le temps de l'afficher ou bien l'écran ne reflète pas encore son changement).

   Le gestionnaire d'affichage n'enlève pas d'élément de cette table un fois qu'ils sont affichés, il se contente mettre le booléen à `vrai`. *Pourquoi ? (la réponse est triviale)*

4. Quand le moteur de jeu modifie une vignette existante :

   - Si elle était à jour sur l'écran : il calcule une échéance pour son affichage.
   - Sinon, l'écran ne reflétait pas son état, dans ce cas, il laisse l'échéance déjà définie.

   Il met le booléen à `faux` pour indiquer que la vignette n'est pas à jour sur l'écran.

   Expliquez pourquoi il change l'échéance de cette manière.

5. Si l'on programme sans réfléchir en suivant ce qui est dit dans les deux questions précédentes, il y aura un problème de section critique. *Dans quel cas se produit-il ? Proposez la "meilleure" méthode pour le résoudre.*

6. *Quels sont les processus consommant le plus de CPU ?*

7. *Quels sont les processus qui ne doivent pas manquer de temps CPU ?*

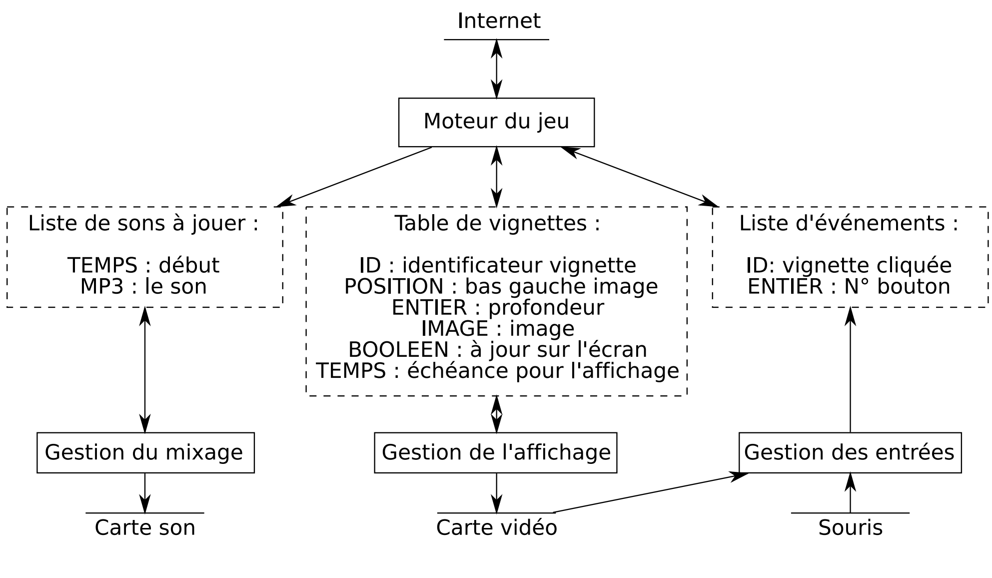
8. *Quels sont les processus que l'on peut brider ?*

9. *Comment ordonnanceriez-vous les 4* processus *?*

10. Le processus qui gère l'affichage décide lui-même quelles sont les vignettes à mettre à jour. On aurait pu faire autant de *threads* que de vignettes à afficher. *Indiquez les avantages et inconvénients des deux méthodes.*

11. *Proposez un méthode* **simple** *permettant d'avoir n* threads *pour l'affichage et qui ne nécessite pas de sections critiques.*

12. Dans le cas où il y a un processus unique pour faire l'affichage. *Proposez une méthode pour ordonnancer l'affichage des vignettes. Justifiez votre choix.*



Une correction.

1. On pourrait faire cette application en utilisant un seul processus. *Donnez les avantages et inconvénients de cette approche.*

   Avantages :

   - Pas besoin de gérer les sections critiques.
   - Cela utilise moins de mémoire pour la pile.
   - Synchronisation des actions plus simple à faire.
   - Le programme doit faire un ordonnanceur en interne, cela lui permet de faire intelligemment l'ordonnancement.
   - Moins de changement de contexte pour l'ordonnanceur système, mais c'est un gain négligeable.

   Inconvénients :

   - On n'utilise qu'un seul processeur.
   - Il faut gérer des entrées/sorties asynchrones.
   - Le programme doit faire un ordonnanceur en interne, cela complique le code.
   - S'il y a un plantage, le jeu complet plante.

2. Les deux listes sont particulières, un seul processus ajoute des éléments dedans et un seul en enlève. *Proposez une implémentation de ces listes qui tienne compte de ce fait. Prouvez que votre implémentation fonctionne.*

   Petite remarque : la liste pour le son n'est pas une FIFO car des éléments au milieu de la liste peuvent être supprimés.

   Si vous proposez une section critique c'est que vous faites une implémentation générale des listes. C'est malheureusement hors sujet.

   Je vous propose la structure de donnée suivante : une liste chaînée classique avec un champ supplémentaire booléen nommé `a_détruire`.

   Le processus qui veut détruire un élément de la liste passe ce booléen à `vrai`.

   Le processus qui ajoute un élément à la liste recycle les éléments détruits sans les enlever physiquement de la liste. Si un élément est (re)ajouté, il est d'abord initialisé.

   Pourquoi cela fonctionne-t-il sans section critiques :

   - Un seul processus modifie les pointeurs (écrivain).
   - Un seul processus modifie la valeur `a_détruire` pour la passer à 1 : lecteur, ou à 0 : écrivain.
   - Le processus lecteur qui parcours la liste en calculant et en enlevant des éléments obtient toujours un pointeur sur une zone mémoire valide, elle peut être notée comme `a_détruire` dans ce cas on en tient pas compte.
   - Cela n'a aucune importance si la valeur de `a_détruire` change pendant que le processus écrivain la teste, la cellule sera réutilisée au prochain passage.
   - La valeur de `a_détruire` ne peut passer de 0 à 1 pendant que le processus lecteur parcours la liste. En effet, il est le seul à pouvoir la modifier dans ce sens. Si elle passe de 1 à 0, on en tiendra compte à la prochaine itération.

```
/* Code non compilé ni testé */
Liste * ajoute(Liste **l)        // Seulement appelée par l'écrivain.
{
        Liste *item ;

        while( *l )
           {
              if ( (*l)->a_detruire )
                 return *l ;
              *l = (*l)->suivant ;
           }
        ALLOUER(item) ;
        item->suivant = *l ;
        item->a_detruire = 1 ;
        *l = item ;
        return item ; // La fonction appelante mettra a_detruire à 0
}
void enleve(Liste *l)        // Seulement appelée par le lecteur.
{
        l->a_detruire = 1 ;
}
```

On peut améliorer le code précédent en enlevant les éléments détruits de la liste **sans libérer leur mémoire ni modifier le pointeur** `suivant`. En effet, le processus lecteur peut être en train de parcourir la liste.

On peut améliorer en faisant tenir à jour par le processus lecteur une liste chaînée contenant les éléments à détruire. L'écrivain n'aurais plus alors à parcourir la liste pour trouver les cellules réutilisables. L'insertion sera alors en O(1).

Cette implémentation permet de s'affranchir des verrous.

Il est possible de libérer la mémoire si le processus lecteur indique quand il a fini de parcourir la liste. Dans ce cas, l'écrivain sait que le lecteur ne peut plus avoir de pointeur vers des éléments qu'il s'apprête à détruire.

Attention, beaucoup on proposé un FIFO sans s'apercevoir que le même pointeur pouvait être écrit simultanément par les deux processus quand la liste passe de 1 à 0 éléments.

3. La table des vignettes contient le modèle de ce que l'on veut afficher sur l'écran. Le booléen indique si l'image sur l'écran est à jour pour la vignette considérée (on a eu le temps de l'afficher ou bien l'écran ne reflète pas encore son changement).

   Le gestionnaire d'affichage n'enlève pas d'élément de cette table un fois qu'ils sont affichés, il se contente mettre le booléen à `vrai`. *Pourquoi ? (la réponse est triviale)*

   Parce que si une vignette en premier plan bouge, il faut afficher ce qui est derrière, le gestionnaire d'affichage doit donc forcément connaître les vignettes qui sont derrière. *Ce n'est pas au moteur de jeu, de décider de ce qui doit être réaffiché.*

   J'ai néanmoins tenu compte de la réponse disant que cela évite de continuellement ajouter et enlever des éléments de la table. Bien que cette réponse ne soit pas la bonne.

4. Quand le moteur de jeu modifie une vignette existante :

   - Si elle était à jour sur l'écran : il calcule une échéance pour son affichage.
   - Sinon, l'écran ne reflétait pas son état, dans ce cas, il laisse l'échéance déjà définie.

Il met le booléen à `faux` pour indiquer que la vignette n'est pas à jour sur l'écran.

Expliquez pourquoi il change l'échéance de cette manière.

Si la vignette est à jour sur l'écran, son ancienne échéance n'a plus aucune signification puisque l'affichage a été fait.

Si la vignette n'est pas à jour sur l'écran cela veut dire que le système a manqué de temps pour l'afficher. On laisse donc la même échéance car cela nous assure que la vignette sera affichée régulièrement. Dans le cas contraire, on reporte continuellement dans le futur son échéance d'affichage, il se pourrait que la vignette ne soit jamais affichée car elle ne serait jamais suffisamment prioritaire.

5. Si l'on programme sans réfléchir en suivant ce qui est dit dans les deux questions précédentes, il y aura un problème de section critique. *Dans quel cas se produit-il ? Proposez la "meilleure" méthode pour le résoudre.*

   En fait dans plein de cas :

   - Quand les 2 processus modifient le booléen indiquant que c'est à jour sur l'écran.
   - Quand le moteur de jeu modifie la position ou l'image de la vignette quand le gestionnaire d'affichage lit ces valeurs.

   Solutions possibles :

   - Un mutex par vignette : c'est très lent et pose des problèmes de verrous. C'est clairement pas la bonne solution pour cette application.
   - Une solution astucieuse est extrêmement rapide :

     Quand le gestionnaire d'affichage veut afficher une vignette :

     - Il met `affichage_en_cours` à vrai.
     - Si `modification_en_cours` est vrai : alors il n'affiche pas la vignette.
       Sinon, il fait l'affichage
     - Il met `affichage_en_cours` à faux.

     Quand le moteur de jeu veut modifier une vignette,

     - Il met `modification_en_cours` à vrai.
     - Si `affichage_en_cours` est vrai : alors il réessayera de modifier la vignette dans le futur.
       Sinon il modifie la vignette.
     - Il met `modification_en_cours` à faux.

     Il est possible (improbable) que les 2 processus laissent tomber la vignette simultanément mais il est **impossible** qu'ils continuent tous les 2 le traitement sur la même vignette.
   - Une solution plus classique mais aussi plus lourde : RCU.

     Le moteur du jeu rempli la structure de donnée vignette puis fait la substitution de pointeur.

     Le gestionnaire d'affichage travaille sur l'ancienne vignette. Le moteur de jeux ne doit donc pas la recycler tout de suite mais seulement quand le gestionnaire d'affichage aura terminé son cycle d'affichage, en effet à ce moment là, il n'aura plus aucun pointeur en mémoire.

6. *Quels sont les processus consommant le plus de CPU ?*

   Dans l'ordre :

   (a) Le gestionnaire d'affichage doit traiter des millions de pixels par secondes.
   (b) Le gestionnaire de sons doit traiter des dizaines de milliers d'échantillons de sons par secondes.
   (c) Le moteur de jeux doit traiter des centaines de modifications d'état par secondes. La plupart du temps, il ne fait que modifier des pointeurs et des entiers, il ne manipule pas de grosses données.
   (d) Le gestionnaire des entrées traite des dizaines d'événements souris par seconde.

   Sous unix on ne scrute pas les périphériques pour savoir s'il y a des données à lire ou non (fonctions `select` ou `poll`).

7. *Quels sont les processus qui ne doivent pas manquer de temps CPU ?*

   Si le gestionnaire d'entrée manque de temps, c'est pas grave, la souris se déplacera avec un à coup la prochaine fois.

   Si gestionnaire d'affichage manque de temps, c'est pas grave non plus, les mouvements seront un peu plus saccadé car on ne pourra pas afficher tous les changements.

   Si le moteur de jeux manque de temps, il se rattrapera la prochaine fois.

   Si le gestionnaire de son manque de temps, c'est critique pour deux raisons :

   - Le son s'arrête, ce qui est très pénible.
   - Quand le son reprend, il y a toutes les chances pour qu'il soit décalé par rapport à l'image.

8. *Quels sont les processus que l'on peut brider ?*

   Il n'y a aucun intérêt à brider le gestionnaire d'entrée car il ne consomme pas de ressources.

   Le moteur de jeux n'a pas beaucoup d'intérêt à être bridé car il ne consomme pas beaucoup de ressources. De plus il faut le brider sans que cela change le fonctionnement du jeux. C'est complexe à réaliser et dépend du jeu.

   Le gestionnaire d'affichage est bridé par définition. En effet, il n'affiche pas les images au moment ou on lui demande, il affiche les images en fonction de leur échéance. On peut lui donner aussi peu de temps CPU que l'on veut, le contenu de l'écran ne sera pas à jour mais il n'affichera pas n'importe quoi. Si ce n'est pas suffisant, l'utilisateur final peut réduire la résolution d'affichage. *Attention, la notion de nombre d'images par seconde (FPS) n'a pas de sens avec un tel afficheur puisque que l'écran n'est pas mis à jour dans sa totalité.*

   Finalement, c'est le gestionnaire de son qui peut facilement être bridé. Il est difficile de le brider en change la résolution du son (fréquence à 8000Hz au lieu de 40000Hz) car cela nécessiterai d'échantillonner à nouveau le son. Par contre, il est facile de le brider en fixant une borne sur le nombre de son que l'on peut mixer simultanément. Le moteur de jeu devra définir les sons importants et ceux qui le sont moins.

9. *Comment ordonnanceriez-vous les 4* processus *?*

   Seule la tâche son est périodique, les autres sont déclenchées par des événements.

   Les capacités des tâches sont dynamiques. On ne peut rien prévoir à l'avance car le système est complexe.

   Il faut donc utiliser un ordonnanceur HPF. De la plus à la moins prioritaire :

- La gestion des entrées : cette tâche ne consomme pas de temps CPU. En la mettant prioritaire on garantit la meilleure interactivité sans causer de problèmes aux tâches moins prioritaires.
- Le son (c'est une tâche critique).
- Le moteur du jeu car il consomme peu et cela garanti une bonne interactivité.
- Le gestionnaire d'affichage. Il prend le temps restant. Si une vignette est mise à jour 30 fois par seconde au lieu de 300 c'est pas grave.

Petite remarque : sous unix, si vous lancez ces 4 tâches avec la priorité par défaut, le système va corriger de lui même les priorités pour les mettre à peu près dans le même ordre que celui que je viens de donner. Il augmente automatiquement la priorité des tâches bloquées par des entrées sorties et baisse celle qui ne font qu'utiliser le CPU.

10. Le processus qui gère l'affichage décide lui-même quelles sont les vignettes à mettre à jour. On aurait pu faire autant de *threads* que de vignettes à afficher. *Indiquez les avantages et inconvénients des deux méthodes.*

    Avantages 1 processus : simple à programmer.

    Inconvénients 1 processus : n'utilise pas plusieurs processeurs.

    Avantages n *threads* : utilise plusieurs processeurs.

    Inconvénients n *threads* :

    - Cela prend beaucoup de mémoire.
    - On perd du temps CPU en changement de contexte.
    - Il faut que l'ordonnancement des *threads* soit fait avec EDF sinon on ne respecte pas le cahier des charges.
    - Il ne faut faire attention à ce que l'ensemble des *threads* laisse tourner les autres processus.
    - **Les pixels de l'écran sont une donnée partagée entre les *threads*.** Ce simple fait rend cette approche inutilisable de par la lourdeur de gestion des sections critiques.

    Remarques :

    - On crée les *threads* au démarrage du programme, on ne s'amuse pas à les lancer et les arrêter à chaque fois qu'il y a du travail à faire.

11. *Proposez un méthode* **simple** *permettant d'avoir n* threads *pour l'affichage et qui ne nécessite pas de sections critiques.*

    Pour que cela marche bien, on prend n égal au nombre de processeurs de la machine.

    Puisque dans la méthode précédente la section critique est l'écran, il faut donc que chaque *thread* s'occupe seulement d'un zone écran. Il n'y aura donc plus de zone critique.

    Chaque *thread* s'occupe donc de toutes les vignettes mais seulement pour une zone réduite de l'écran.

    Problèmes à résoudre :

    - Une vignette peut être sur plusieurs zones. Il faut donc plusieurs booléens pour indiquer qu'elle est jour.
    - Il faut essayer d'afficher simultanément les vignettes à cheval sur plusieurs zones pour éviter qu'elles soient coupées en morceaux.

Pour optimiser la gestion du cache de donnée, il est préférable de découper l'écran en bandes horizontales.

12. Dans le cas où il y a un processus unique pour faire l'affichage. *Proposez une méthode pour ordonnancer l'affichage des vignettes. Justifiez votre choix.*

    Nous connaissons les échéances, il faut donc utiliser EDF.

    Comme les tailles des vignettes sont connues, on pourrait calculer la capacité nécessaire pour l'affichage, dans ce cas LLF peut être utilisé.

    Dans tous les cas, quand on met à jours une vignette il faut afficher toutes les vignettes qui sont derrières (qui ne sont plus cachées) et toutes les vignettes qui sont devant. Il ne faut pas oublier de toutes les noter comme *à jour sur l'écran*.