La coopération entre processus

Motivation

Des processus actifs simultanément sont dits exécutés en *parallèle*. Ils peuvent opérer de façon tout à fait indépendante, mais une *interaction* entre processus parallèles est souvent nécessaire. L'origine de cette nécessité d'interaction peut être :

- La coopération à un but commun, par exemple
  - le calcul parallèle sur une machine à processeurs multiples,
  - la coopération entre processus se trouvant sur des machines différentes dans la contexte d'applications distribuées,
  - l'organisation d'un programme en tâches parallèles, notamment dans le cas d'un système de contrôle ;
- Le partage de ressources, par exemple un système de fichiers sur disque.

Gérer l'interaction entre processus parallèles : La communication entre processus

Nous considérons pour le moment des processus exécutés sur une seule machine, comportant éventuellement plusieurs processeurs.

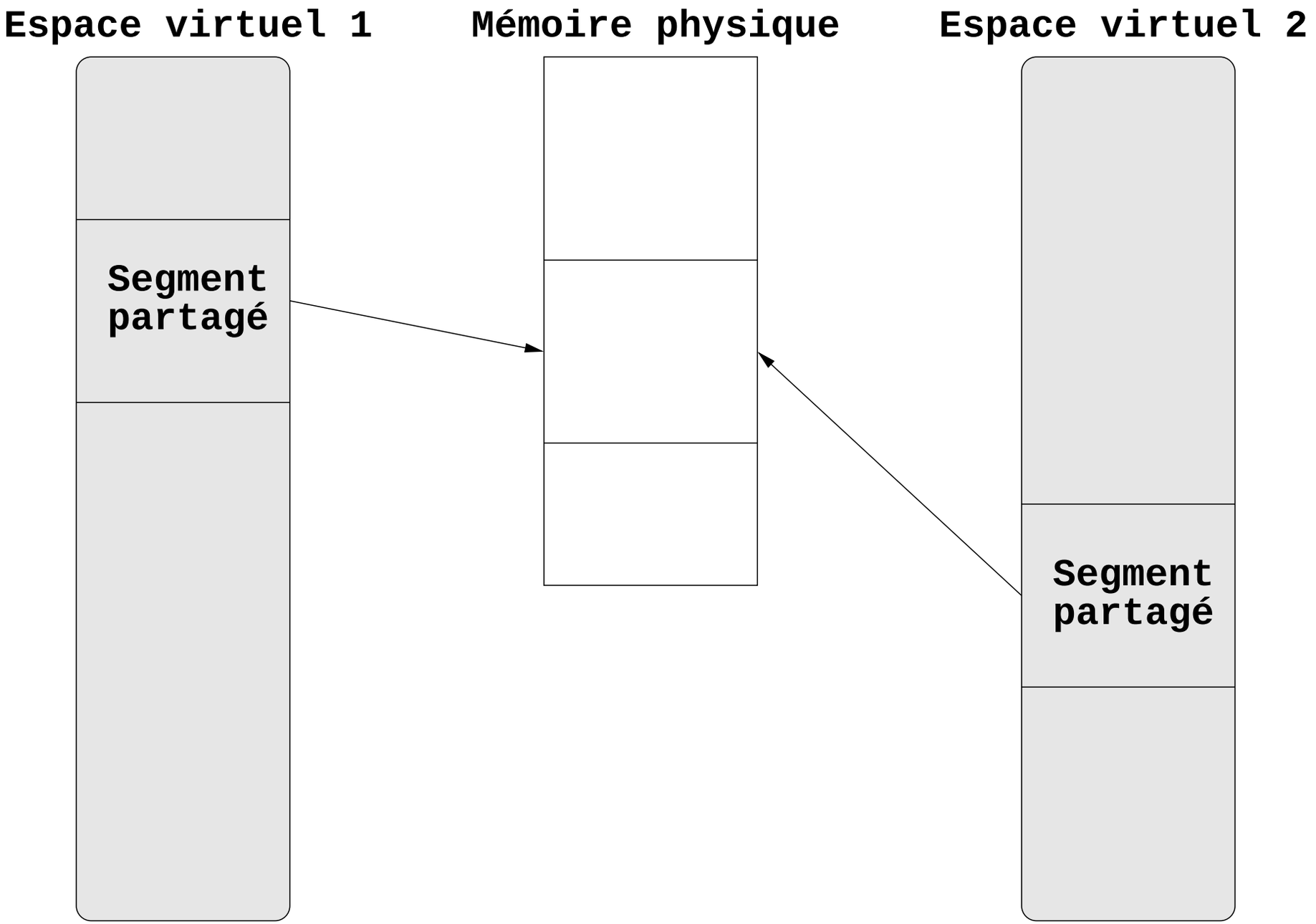
- L'interaction entre processus se fera à l'aide d'appels au système prévus dans ce but.

- Un premier mécanisme de communication entre processus est la création de *segments de mémoire partagée*, c'est-à-dire de segments de mémoire physique apparaissant dans la mémoire virtuelle de plusieurs processus.

- Une fois un segment de mémoire partagée crée, les processus peuvent interagir par l'intermédiaire de cette mémoire.

- Le problème est alors de mettre au point des techniques de programmation pour exploiter cette forme d'interaction.

Un segment de mémoire partagée



Comprendre la mémoire partagée

- Les opération de base sur la mémoire partagée sont les accès en lecture et en écriture (`LD` et `ST`).
- Les instructions `LD` et `ST` sont exécutées de façon atomique (sans interaction). En effet,
  - sur un processeur unique il n'est possible de passer d'un processus à un autre qu'entre deux instructions ;
  - sur un multiprocesseur à mémoire partagée, il doit y avoir un mécanisme d'accès exclusif au bus connectant les processeurs à la mémoire.
- Le comportement de processus parallèles dépendra donc de l'ordre dans lequel les instructions `LD` et `ST` se trouvant dans des processus différents seront exécutées.
- Dans la plupart des cas, cet ordre est inconnu et peut varier d'une exécution à l'autre.

Un exemple d'interaction entre processus

Dans le programme ci-dessous,nous supposons que pour chaque processus `r2` contient l'adresse d'un mot donné en mémoire partagée, initialisé à 0.

Processus 1	**Processus 2**
`LD(r2,0,r0)`	`LD(r2,0,r0)`
`ADDC(r0,1,r0)`	`ADDC(r0,1,r0)`
`ST(r0,0,r2)`	`ST(r0,0,r2)`

Suivant que l'ordre d'exécution sera

```
LD(r2,0,r0)                          LD(r2,0,r0)
ADDC(r0,1,r0)                                        LD(r2,0,r0)
ST(r0,0,r2)                          ADDC(r0,1,r0)
                                                     ADDC(r0,1,r0)
                  LD(r2,0,r0)        ST(r0,0,r2)
                  ADDC(r0,1,r0)                      ST(r0,0,r2)
                  ST(r0,0,r2)
```

ou

le mot de mémoire partagée aura la valeur 2 ou 1 après l'exécution des processus. Il faut noter que, chaque processus opérant dans son propre contexte, les registres ne sont pas partagés.

Les ordres d'exécution possibles : Le modèle de l'entrelacement

- Si l'interaction entre processus se limite à des accès atomiques `LD` et `ST` à la mémoire partagée, le résultat d'une exécution parallèle sera le même que celui d'une séquentialisation (un *entrelacement*) des instructions des processus.

- Cela est évident sur une machine à processeur unique ; sur un multiprocesseur, cela résulte du fait que les opérations `LD` et `ST` sont atomiques et que ces opérations seront donc exécutées de façon séquentielle. Vu qu'il n'y a pas d'interaction au niveau des autres instructions, les exécuter en parallèle ou séquentiellement donne le même résultat.

- La séquentialisation qui sera effectivement observée, va dépendre du gestionnaire de processus et de l'état de la machine au moment de l'exécution du programme.

- Ces éléments n'étant pas parfaitement connus et pouvant varier d'une exécution à l'autre au fera l'hypothèse que la séquentialisation observée pourra être *absolument quelconque*. C'est le *modèle de l'entrelacement* :

  L'effet de l'exécution de processus parallèles peut être celui de tout entrelacement des instructions élémentaires exécutées par ces processus.

- Donc, lorsque l'on écrit un programme qui sera exécuté sous la forme de processus parallèles (un *programme parallèle*), il faudra s'assurer que son comportement est correct, pour tout entrelacement des instructions élémentaires qui le composent.

Ecrire des programmes parallèles

- Le modèle de l'entrelacement est une abstraction simple et sûre du comportement de programmes parallèles, mais il n'est pas aisé d'écrire des programmes qui sont corrects pour ce modèle.

- Considérons le cas d'une structure de données partagée. Si plusieurs processus appliquent simultanément à cette structure une modification composée de plusieurs opérations, il est difficile de garantir que ces modifications simultanées se passeront correctement.

- La solution est de rendre *atomiques* les opérations de modification de la structure partagée, autrement dit de s'assurer qu'un seul processus à la fois peut modifier la structure.

- Pour ce faire, il faut un algorithme permettant de garantir *l'exclusion mutuelle*.

L'exclusion mutuelle

- Le problème de *l'exclusion mutuelle* consiste à garantir que des sections de code, dites *sections critiques*, de processus parallèles ne sont jamais exécutées simultanément.

- Précisément, il faut que, lorsqu'un processus entame l'exécution d'une section critique, aucun autre processus ne puisse faire de même avant que ce processus n'ait achevé l'exécution de la section critique.

- L'exclusion mutuelle est un outil de base qui permet le développement de programmes parallèles complexes.

- Utilisant uniquement la mémoire partagée, l'exclusion mutuelle est plus difficile à réaliser qu'il n'y paraît.

- Nous allons traiter le cas de 2 processus.

L'exclusion mutuelle : schéma général

Soit deux processus exécutant un programme dont la structure est la suivante.

```
#define True  = 1
#define False = 0
```

Processus 1 :

```
while (True)
{   nc1: /* section
            non critique */

         /* protocole d'accès */
  crit1: /* section critique */
         /* protocole de fin */
}
```

Processus 2 :

```
while (True)
{   nc2: /* section
            non critique */

         /* protocole d'accès */
  crit2: /* section critique */
         /* protocole de fin */
}
```

Les deux processus ne peuvent être simultanément aux localisations `crit1` et `crit2`.

Exclusion mutuelle : premier essai

```
int Turn = 1;
```

Processus 1 :

```
while (True)
{   nc1: /* section
            non critique */ ;
         while (Turn == 2) {};
  crit1: /* section critique */ ;
         Turn = 2;
}
```

Processus 2 :

```
while (True)
{   nc2: /* section
            non critique */
         while (Turn == 1) {};
  crit2: /* section critique */
         Turn = 1;
}
```

Cette solution garantit l'exclusion mutuelle, mais impose une alternance stricte entre les deux processus, ce qui est trop restrictif.

Exclusion mutuelle : deuxième essai

```
int c1,c2  = 1;
```

Processus 1 :

```
while (True)
{   nc1: /* section
            non critique */ ;
         while (c2 == 0) {};
         c1 = 0;
  crit1: /* section critique */ ;
         c1 = 1;
}
```

Processus 2 :

```
while (True)
{   nc2: /* section
            non critique */
         while (c1 == 0) {};
         c2 = 0;
  crit2: /* section critique */
         c2 = 1;
}
```

L'exclusion mutuelle n'est pas garantie :

c1	c2	
1	1	P1 vérifie c2
1	1	P2 vérifie c1
1	1	P1 : c1 = 0
0	1	P2 : c2 = 0
0	0	*Aïe* !

Exclusion mutuelle : troisième essai

```
int c1,c2  = 1;
```

Processus 1 :

```
while (True)
{   nc1: /* section
            non critique */ ;
         c1 = 0;
         while (c2 == 0) {};
  crit1: /* section critique */ ;
         c1 = 1;
}
```

Processus 2 :

```
while (True)
{   nc2: /* section
            non critique */
         c2 = 0;
         while (c1 == 0) {};
  crit2: /* section critique */
         c2 = 1;
}
```

Il y a un blocage possible :

c1	c2	
1	1	P1 : c1 = 0
0	1	P2 : c2 = 0
0	0	P1 vérifie c2
0	0	P2 vérifie c1

On appelle ce type de blocage un *deadlock* (*étreinte fatale*).

Exclusion mutuelle : quatrième essai

```
int c1,c2  = 1;
```

Processus 1 :

```
while (True)
{   nc1: /* section
            non critique */ ;
         c1 = 0;
         while (c2 == 0)
         {   c1 = 1;
             /* attendre */;
             c1 = 0;
         };
  crit1: /* section critique */ ;
         c1 = 1;
}
```

Processus 2 :

```
while (True)
{   nc2: /* section
            non critique */
         c2 = 0;
         while (c1 == 0)
         {   c2 = 1;
             /* attendre */;
             c2 = 0;
         };
   crit2: /* section critique */
         c2 = 1;
}
```

Dans cette solution, si un processus constate que l'exclusion mutuelle n'est pas disponible, il renonce un moment avant de réessayer. Un blocage est encore possible, mais il ne se maintiendra que s'il y a symétrie parfaite entre les exécutions des deux processus.

c1	c2	
1	1	P1 : c1 = 0
0	1	P2 : c2 = 0
0	0	P1 vérifie c2
0	0	P2 vérifie c1
0	0	P1 : c1 = 1
1	0	P2 : c2 = 1
1	1	P1 : c1 = 0
0	1	P2 : c2 = 0
0	0	P1 vérifie c2
0	0	P2 vérifie c1

Exclusion mutuelle : l'algorithme de Dekker

```
int c1,c2, Turn  = 1;
```

Processus 1 :

```
while (True)
{   nc1: /* section
            non critique */ ;
         c1 = 0;
         while (c2 == 0)
         {  c1 = 1;
            while (Turn == 2) {};
            c1 = 0;
         };
  crit1: /* section critique */ ;
         Turn = 2;
         c1 = 1;
}
```

Processus 2 :

```
while (True)
{   nc2: /* section
            non critique */
         c2 = 0;
         while (c1 == 0)
         {  c2 = 1;
            while (Turn == 1) {};
            c2 = 0;
         };
  crit2: /* section critique */
         Turn = 1;
         c2 = 1;
}
```

La symétrie de la solution précédent est rompue grâce à la variable `Turn`.

L'algorithme de Dekker est-il correct ?

- En apparence oui, mais comment s'en convaincre de façon rigoureuse ?

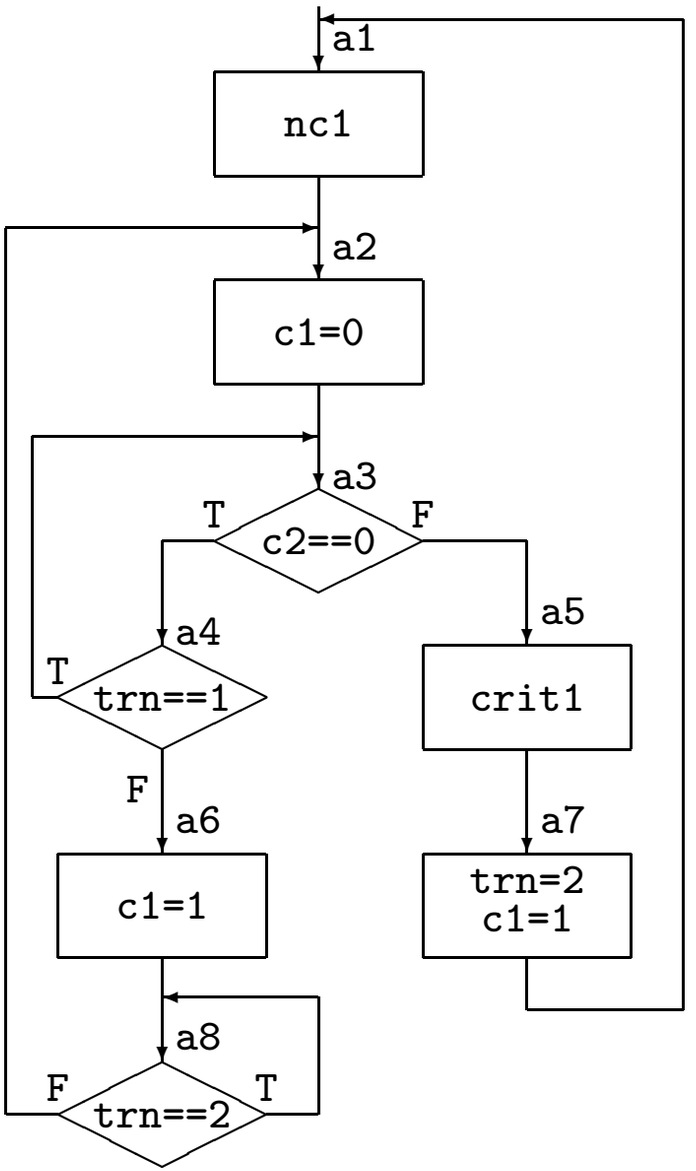
- Une approche est d'examiner, de façon systématique, tous les entrelacements possibles des actions des différents processus.

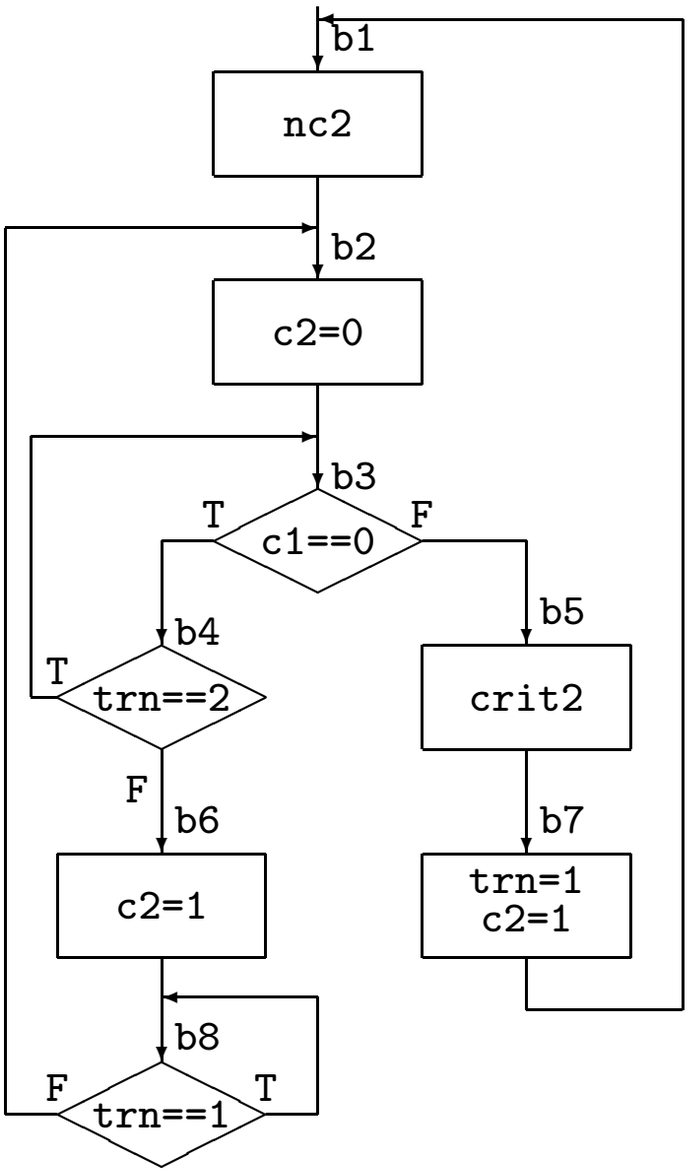
- Pour ce faire, il est utile de présenter le programme sous une forme où la structure de contrôle et les actions des processus sont explicites.

- On explore alors les exécutions possibles du programme, tout en mémorisant les états atteints en vue d'arrêter l'exploration dès que l'on rencontre un état déjà atteint précédemment.

Dekker avec contrôle et actions explicites (version légèrement modifiée)

Initialisations : `c1 = 1; c2 = 1; trn = 1`





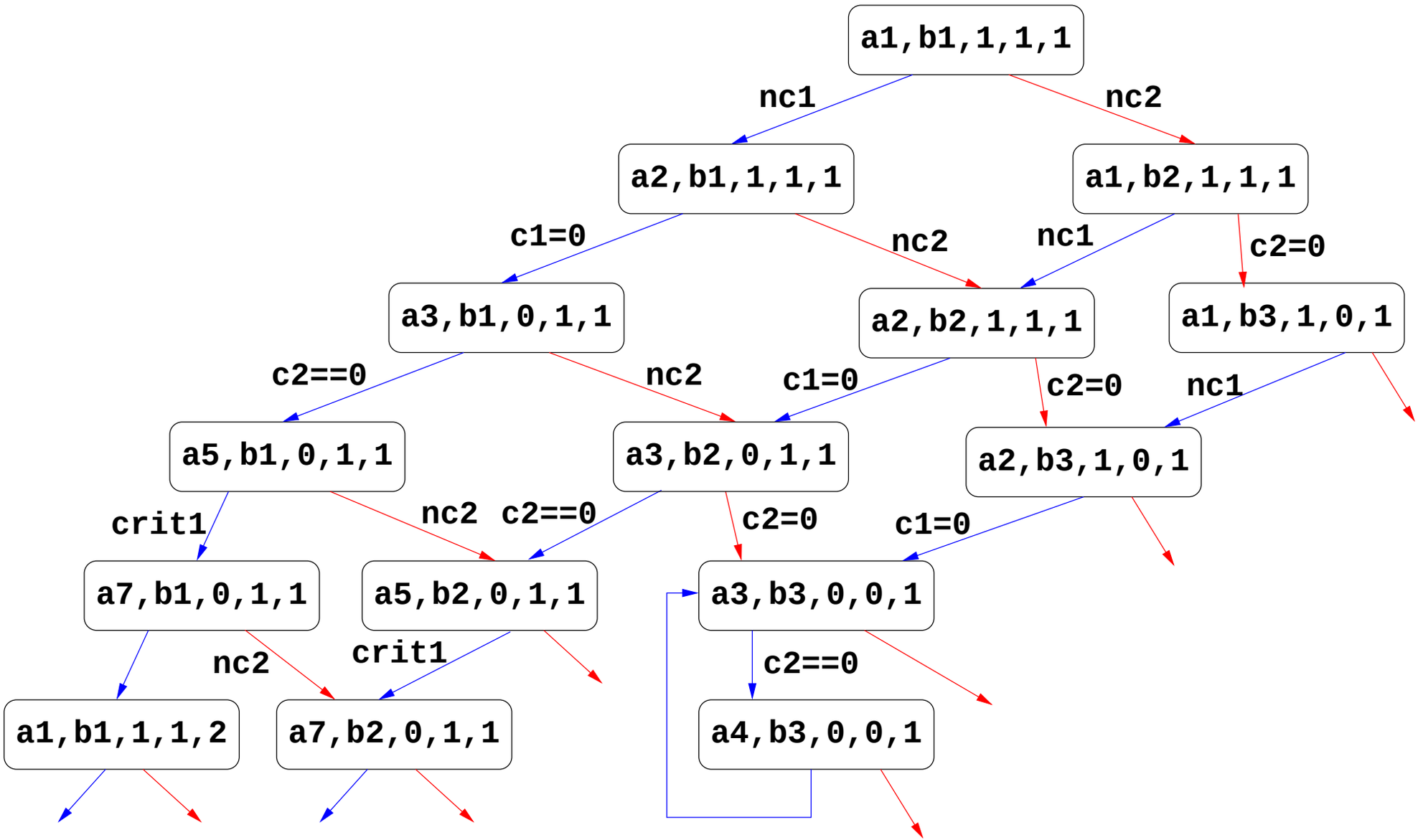
Explorer les exécutions de l'algorithme de Dekker

- Lors de l'exécution de cet algorithme, un état du système est caractérisé par
  - une localisation de contrôle pour chaque processus,
  - une valeur pour chacune des trois variables c_1, c_2 et trn.
- Un état est donc un quintuplet

  (`loc1,loc2,c1,c2,trn`)

- L'état initial est (`a1,b1,1,1,1`)
- Il suffit alors, partant de l'état initial, de générer de nouveaux états en examinant toutes les actions possibles.
- L'exploration des exécutions (des états) s'arrête lorsqu'aucun nouvel état est rencontré, ce qui arrivera immanquablement vu qu'il y a au plus $8 \times 8 \times 2 \times 2 \times 2 = 512$ états.

L'exploration des états de l'algorithme de Dekker



Interpréter l'exploration des états

- La propriété d'exclusion mutuelle est satisfaite s'il est impossible d'arriver à un état où les localisations sont `a5-a7` et `b5-b7`.

- On peut aussi se servir de l'exploration des états pour vérifier qu'un processus qui le souhaite (arrive en `a3` ou `b3`) finira par atteindre sa section critique.

- Pour ce faire, il faut, par exemple, vérifier que tout chemin partant d'un état ou le processus 1 est en `a3` finira par arriver en `a5`.

- On constate toutefois que ce n'est pas le cas. Pourquoi ?

L'équité

- Le fonctionnement correct de l'algorithme de Dekker dépend du fait que chaque processus finit toujours par progresser.

- Cette propriété est dépendante du gestionnaire de processus qui n'est pas toujours connu en détail.

- On ne modélise pas le gestionnaire de processus, mais on corrige le modèle de l'entrelacement par une contrainte qui représente une propriété minimale qui doit être satisfaite par tout gestionnaire de processus : une *hypothèse d'équité*.

- Une hypothèse d'équité simple, suffisante pour raisonner au sujet de l'algorithme de Dekker, est la suivante.

Tout processus qui a la possibilité d'exécuter une instruction finira toujours par le faire.

Propriétés de sûreté - Propriétés de vivacité.

Parmi les propriétés que l'on spécifie pour un programme parallèle, on distingue les deux catégories suivantes.

- **Les propriétés de sûreté**. Ce sont les propriétés qui expriment que certains états indésirables ne sont jamais atteints. Elles ne dépendent pas de l'utilisation d'une hypothèse d'équité.

- **Les propriétés de vivacité**. Celles-ci expriment que des états désirables seront immanquablement atteints. Habituellement, ces propriétés ne sont vraies qu'en présence d'une hypothèse d'équité.

Les limites de la mémoire partagée

- Même si cela est possible, synchroniser des processus en ne se servant que de mémoire partagée est délicat à réaliser.

- En effet, un algorithme comme celui de Dekker n'est pas évident et ne se généralise pas immédiatement à plus de deux processus.

- Un autre inconvénient d'un algorithme comme celui de Dekker est qu'un processus en attente continue à utiliser du temps de calcul.

- Il serait très utile de disposer d'un autre mécanisme de synchronisation qui permettrait notamment une meilleure implémentation de l'exclusion mutuelle.

- Nous étudierons les *sémaphores* qui se présentent sous la forme de variables partagées uniquement utilisables à l'aide d'appels au système spécifiques.

Le concept de sémaphore

Un sémaphore est une variable entière partagée. Sa valeur est positive ou nulle et elle est uniquement manipulable à l'aide de deux opérations `wait(s)` et `signal(s)`, où `s` est un identificateur désignant le sémaphore.

- `wait(s)` décrémente `s` si `s > 0` ; si non, le processus exécutant l'opération `wait(s)` est mis en attente.

- `signal(s)` incrémente `s`. L'exécution de `signal(s)` peut avoir pour effet (pas nécessairement immédiat) qu'un processus en attente sur le sémaphore `s` reprend son exécution.

- L'exécution d'une opération `wait(s)` ou `signal(s)` se fait sans interaction possible (de façon atomique).

L'exclusion mutuelle à l'aide de sémaphores

Les sémaphores permettent une implémentation très simple de l'exclusion mutuelle.

```
semaphore s = 1;
```

Processus 1 :

```
while (True)
{   nc1: /* section
            non critique */ ;
         wait(s);
  crit1: /* section critique */ ;
         signal(s);
}
```

Processus 2 :

```
while (True)
{   nc2: /* section
            non critique */
         wait(s);
  crit2: /* section critique */
         signal(s)
}
```

Exclusion mutuelle et sémaphores : notes

- La solution donnée se généralise immédiatement à un nombre quelconque de processus.

- Si le sémaphore est initialisé à une valeur k autre que 1, on obtient une solution qui permet à k processus d'être simultanément dans leur section critique.

- Un fonctionnement correct de l'exclusion mutuelle implémentée à l'aide des sémaphores nécessite que l'implémentation de l'attente dans l'opération `wait` soit équitable.