Ce que ça veut dire en pratique

Une **instruction atomique** est une suite d'actions telle que

① « *Les variables lues ou modifiées par cette instruction ne peuvent pas être modifiées par le reste du programme au cours de son exécution.* »

Donc *l'effet de l'instruction atomique sur l'état global du programme* est le même que les autres processus poursuivent leur exécution ou non.

② « *Les modifications effectuées par cette instruction ne sont visibles par le reste du programme qu'à la fin de son exécution.* »

Donc *l'effet des processus concurrents à l'instruction atomique sur l'état global du programme* est le même que s'ils étaient suspendus du début à la fin de l'exécution de l'instruction atomique.

Ainsi, tout se passe « comme si » les autres processus sont suspendus ! c'est-à-dire que l'instruction semble s'exécuter de façon exclusive.

La notion de moniteur (P. Brinch Hansen, 1973 & C. A. R. Hoare 1974)

Caractéristique principale d'un programme avec moniteurs : il est composé de deux sortes d'objets :

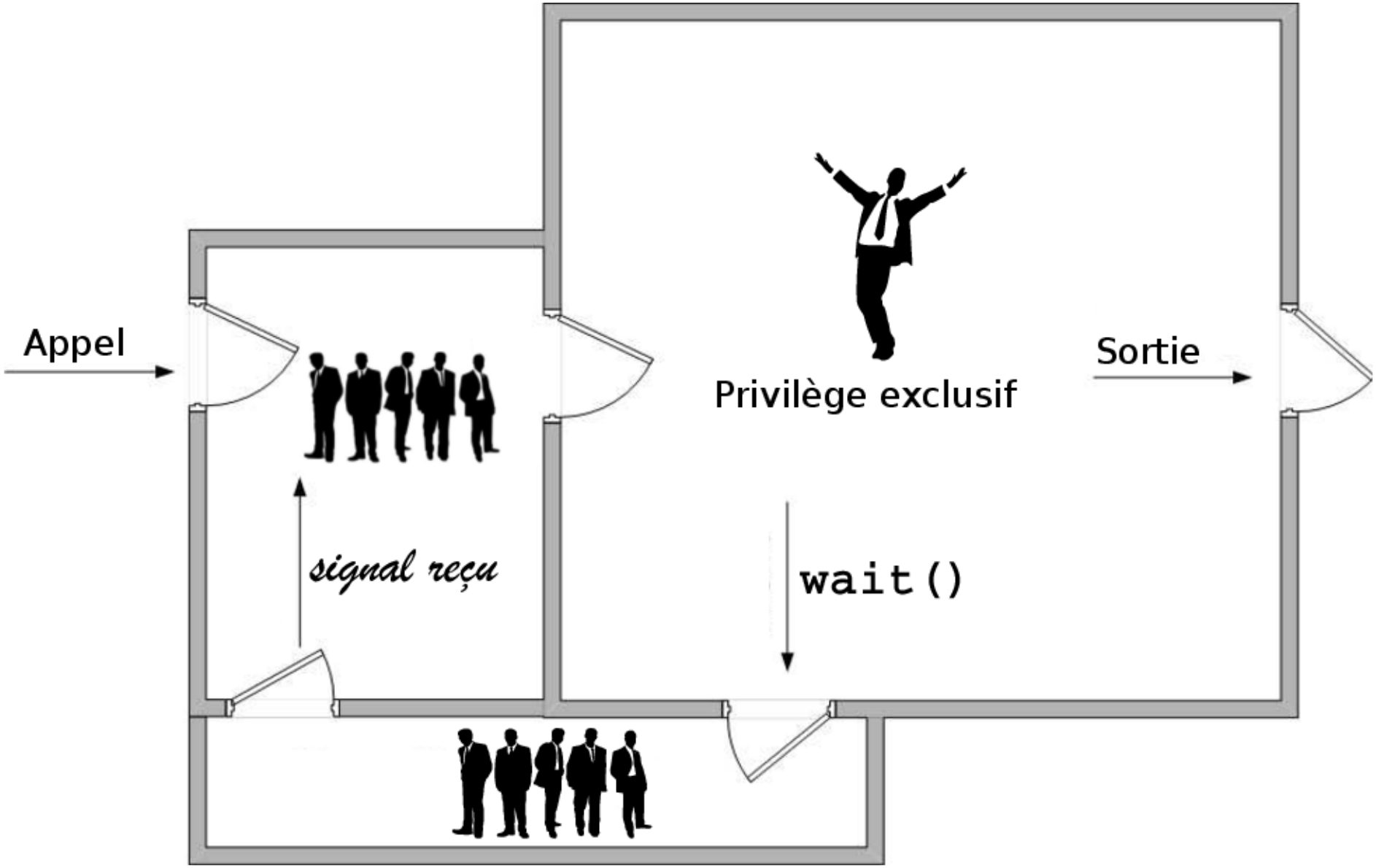
- les processus qui agissent ; Par exemple, les 7 nains.
- les moniteurs qui subissent. Par exemple, Blanche-Neige.

Les interactions et synchronisations entre processus (actifs) se font alors par l'interface des moniteurs (passifs).

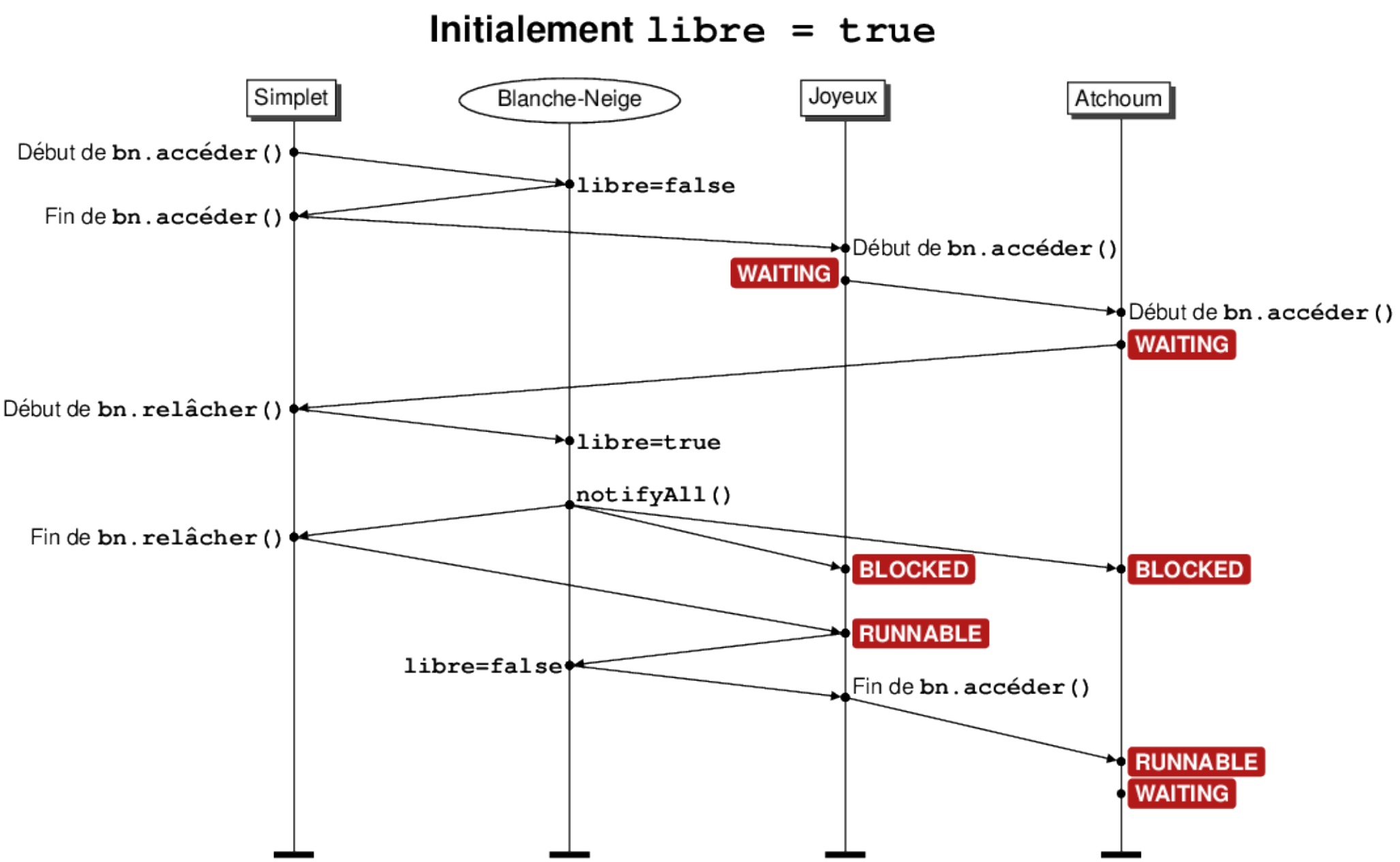
Un moniteur est un module qui regroupe et encapsule des données partagées ainsi que les procédures qui permettent de synchroniser les opérations sur ces données.

En Java, ce sera simplement un objet d'une classe particulière.

Fonctionnement d'un moniteur



Fonctionnement du moniteur Blanche-Neige



Ce qu'il faut retenir

Les problèmes d'*atomicité* sont inhérents à la programmation parallèle (notamment la programmation distribuée) et une source d'erreurs fréquente dans les applications.

Les *verrous* sont bien pratiques pour assurer l'atomicité ; cependant, mal utilisés, ils provoquent facilement des *interblocages*. Les *sémaphores* sont aussi une technique très puissante, mais risquée, peu claire et donc non recommandée dans ce cours. En revanche, le patron de conception par *moniteur* permet d'aborder méthodiquement les problèmes de programmation concurrente. Il sera illustré également par les *collections synchronisées*.

Un thread peut quitter l'instruction **`wait()`** sans qu'il y ait eu d'instruction **`notify()`** exécutée. Ce phénomène étrange et rare se nomme ≪ *spurious wake-up* ≫ que je traduis en ≪ signal intempestif ≫ car cela n'a rien à voir avec **`sleep()`**. Il faut par conséquent protéger chaque **`wait()`** par une boucle **`while()`** comme l'indique la documentation officielle.

Il en résulte qu'un programme correct restera forcément correct en remplaçant chaque appel à **`notify()`** par un **`notifyAll()`**.

Pourquoi continuer à hésiter ?