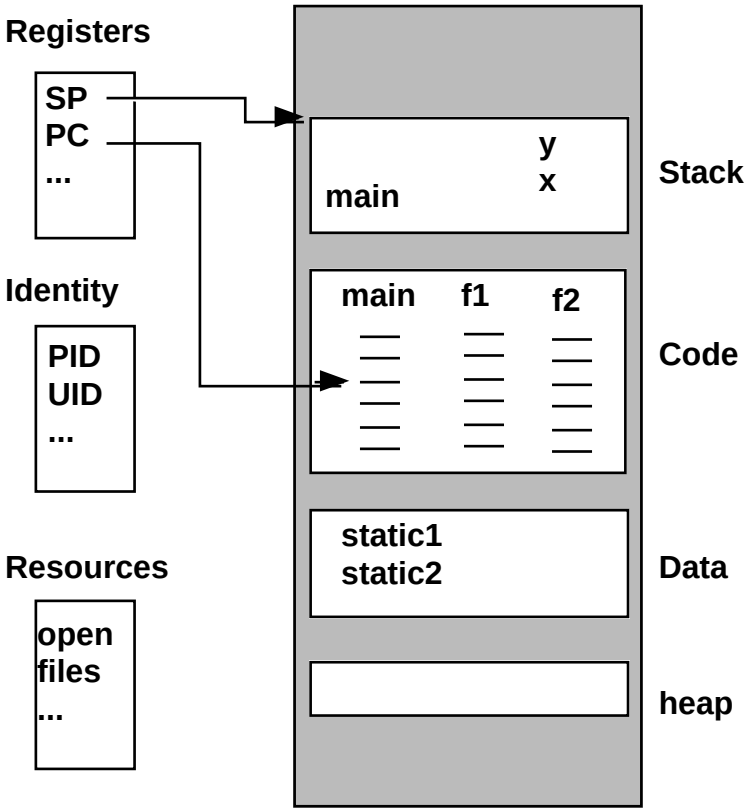
LES THREADS

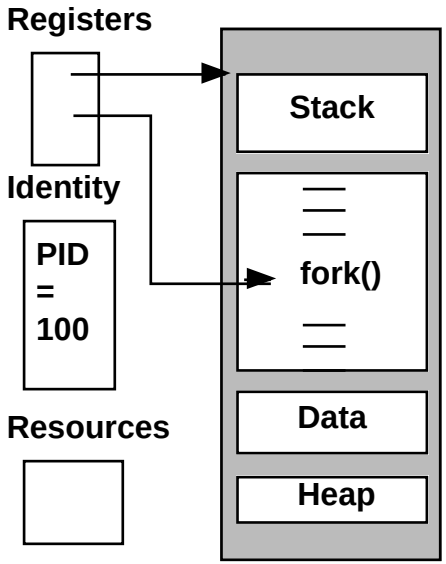
- **Programmation concurrente et parallèle à base de threads**
- **Le multithreading en Java**
- **Un exemple de programmation par threads**

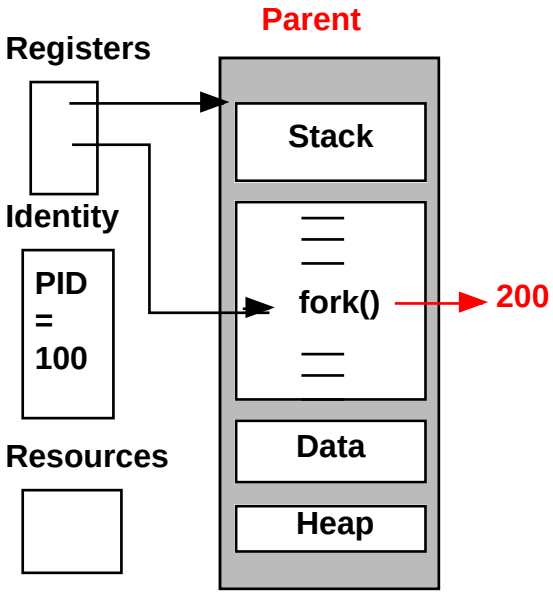
Processus

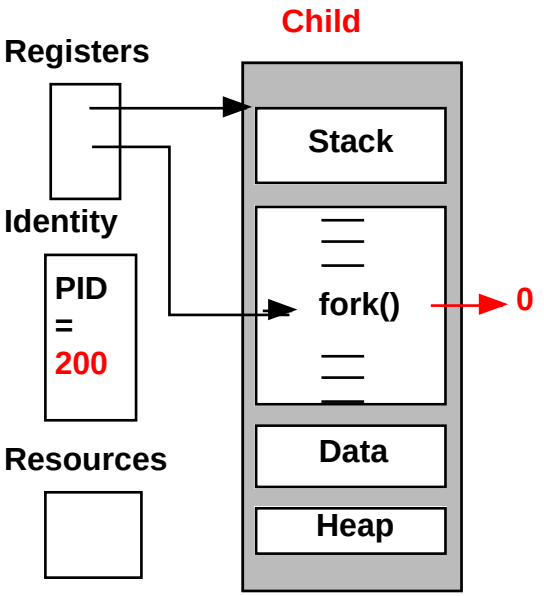


processus = unité de concurrence en Unix

Fork

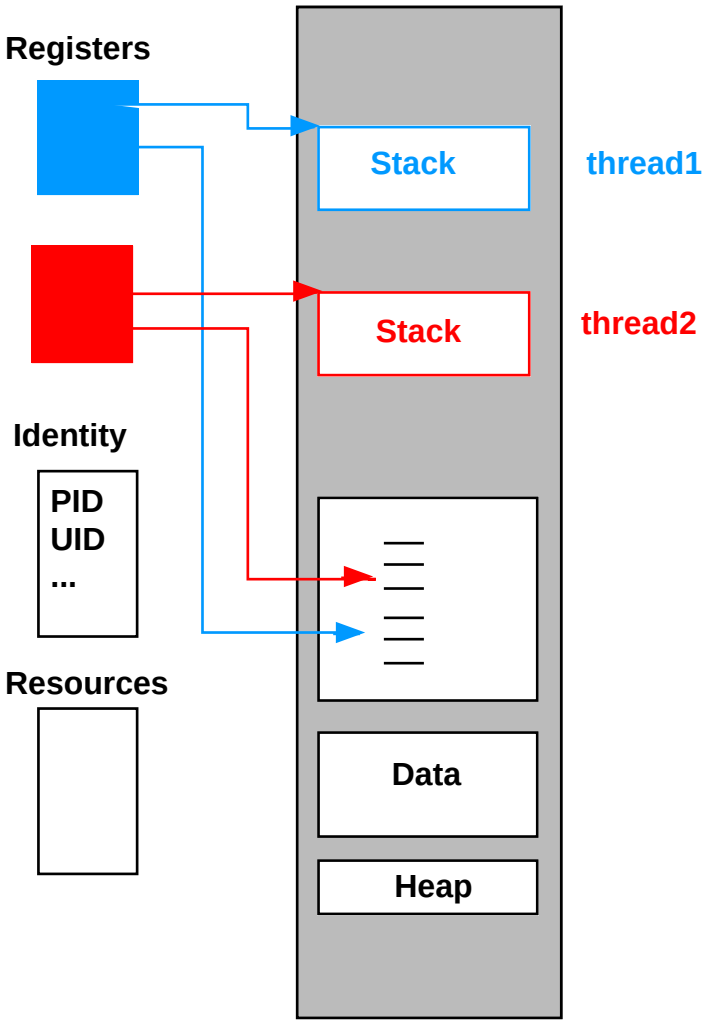







```
if (fork() == 0){
    Child
}else{
    Parent
}
```

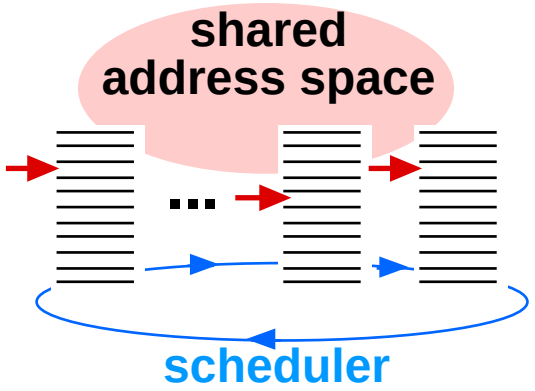
Thread



- Les zones code et données sont partagées
- Les piles et les registres sont propres à chaque threads

thread = “processus léger”

Scheduler



Sauvegarde et restauration des registres : *context switching*

Stratégie de scheduling : comment le scheduler distribue-t-il le processeur aux threads ?

Stratégies de scheduling

Deux stratégies possibles :

Coopérative

Le thread qui a le processeur doit explicitement le relâcher pour permettre aux autres threads de s'exécuter

Préemptive

Le thread qui a le processeur peut être forcé par le scheduler de le relâcher (suivant divers critères : priorité, temps écoulé, interruption, ...)

Stratégie coopérative

- **Simplicité de programmation : coroutines, déterminisme (ou presque...)**
- **Efficacité : maîtrise complète des context switchs**

mais :

- **Problème des threads non coopératives ; risque de blocage de tout le processus !**
- **Réutilisabilité difficile ...**

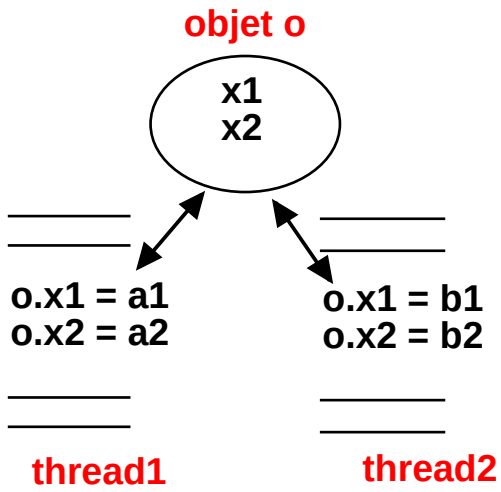
Stratégies préemptives

- **Facilite la réutilisabilité (bien que des problèmes subsistent : réentrance, par exemple)**
- **Adaptation aux multiprocesseurs**

mais :

- **Complexité de programmation : nondéterminisme**
- **Problème d'efficacité : des context switchs inutiles**
- **Besoin de protéger les données partagées**

Protection des données

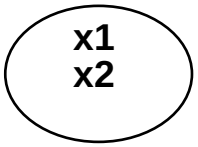


o.x1 = a1
o.x1 = b1
o.x2 = b2
o.x2 = a2

à la fin :
o.x1 == b1
o.x2 == a2

---

boolean lock = false




```
while (lock == true){}
lock = true
o.x1 = a1
o.x2 = a2
lock = false
```

```
while (lock == true){}
lock = true
o.x1 = b1
o.x2 = b2
lock = false
```

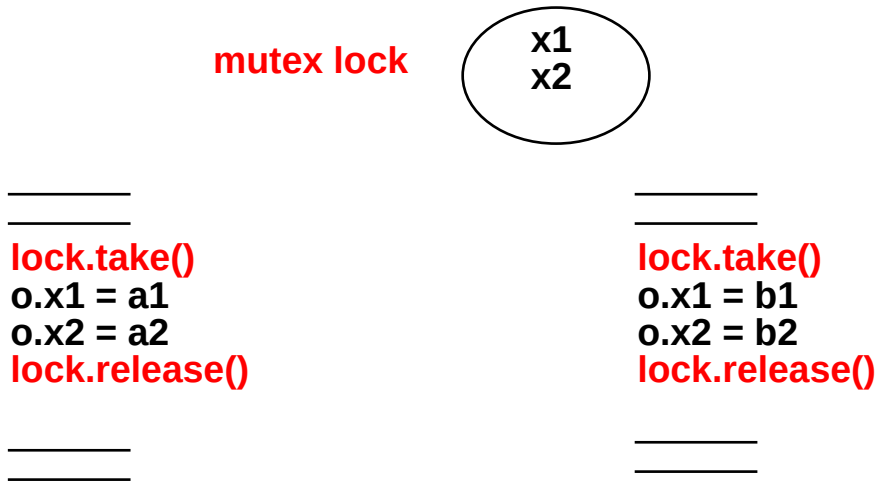
Ne marche pas !
(pourquoi ?)

Verrous

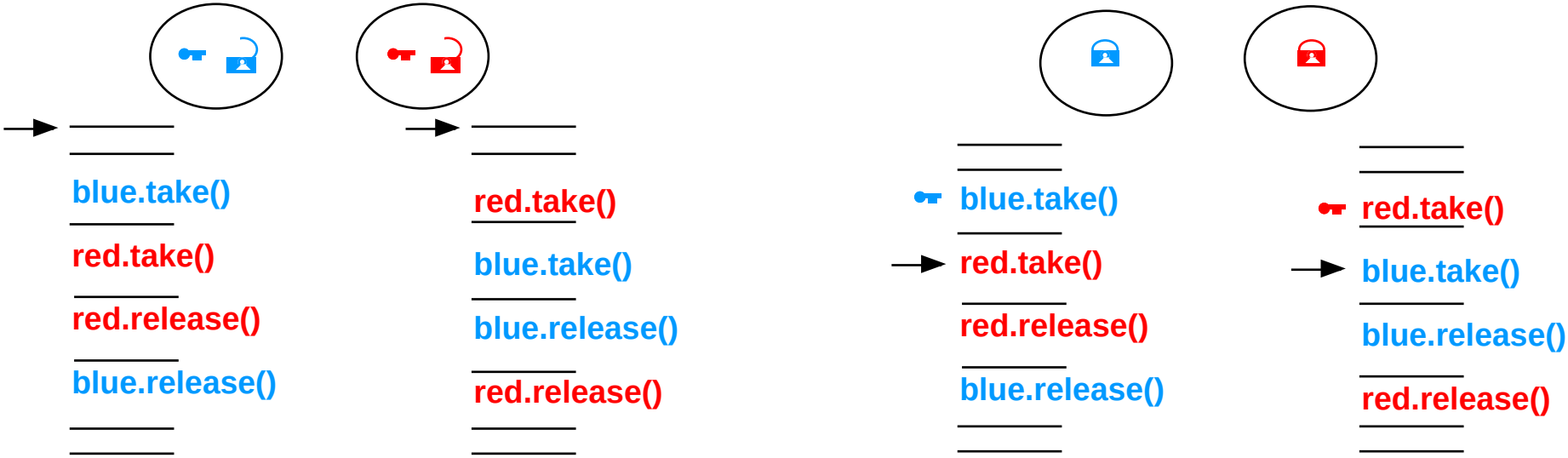
Solution : verrou pour exclusion mutuelle

Action atomique “test and set” (hard)

mutex, lock, semaphore, ...



Deadlock

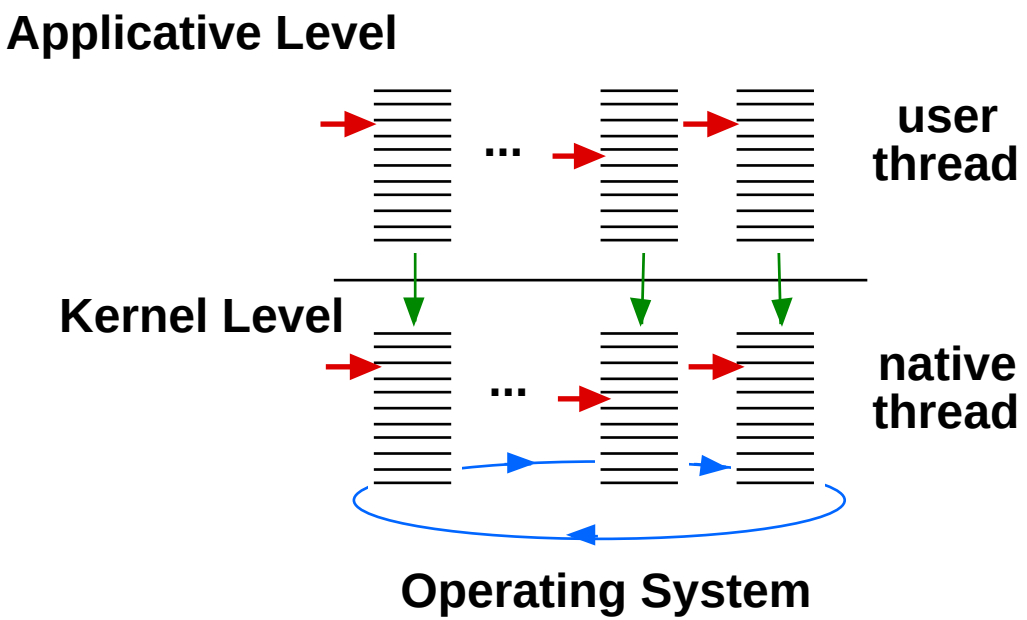


Pas de possibilité de déterminer si un programme tombe dans un deadlock ni d'éliminer les deadlocks, dans le cas général (pourquoi ?).

Problème central de la programmation concurrente.

Mapping sur les ressources

threads utilisateurs -> threads dans le noyau

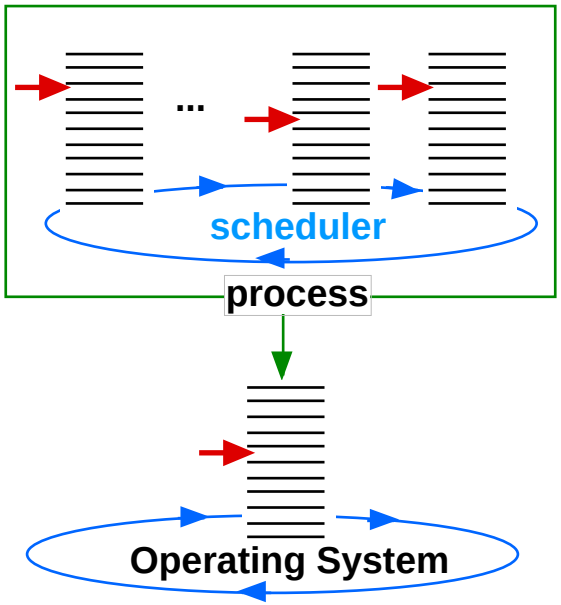


Mapping one-to-one

NT

- **Context switchs dans le noyau plus coûteux**
- **Scheduling forcément préemptif**

Mapping sur les ressources - 2

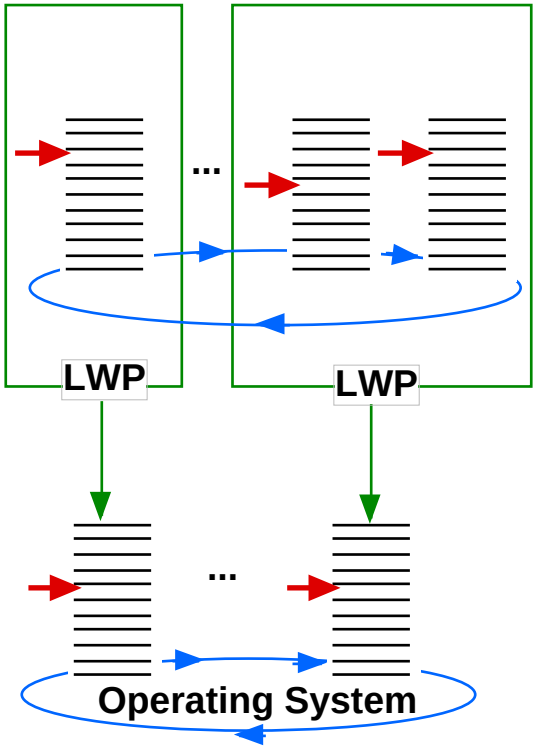


Mapping many-to-one

Anciennes versions de Solaris (green threads)

- **Pas de préemption entre threads utilisateurs : scheduling coopératif**
- **Efficacité : context switchs au niveau applicatif**
- **Pas adapté au parallélisme**

Mapping sur les ressources - 3



Mapping many-to-many

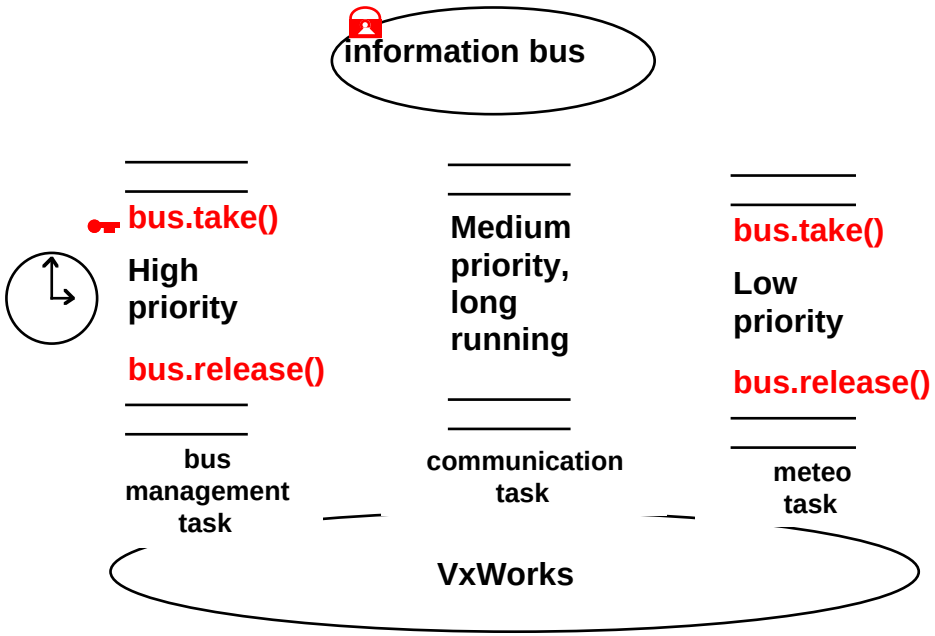
Solaris

- **Souplesse du mélange préemptif/coopératif**
- **Difficulté de programmation...**

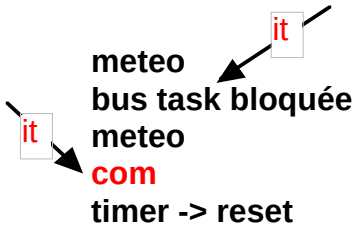
Priorités

Priorité associée à un thread ; le scheduler choisit le thread de priorité maximale

- **Niveaux : NT = 7, Solaris = 2^{31}, Java = 10 ...**
- ***Priority boosting* de NT...**
- **Inversion de priorité**



Mars Pathfinder 1997



Conclusion

Programmation complexe, pour spécialistes

Nombreuses notions, avec “sémantique faible” (dépendant de la plate-forme)

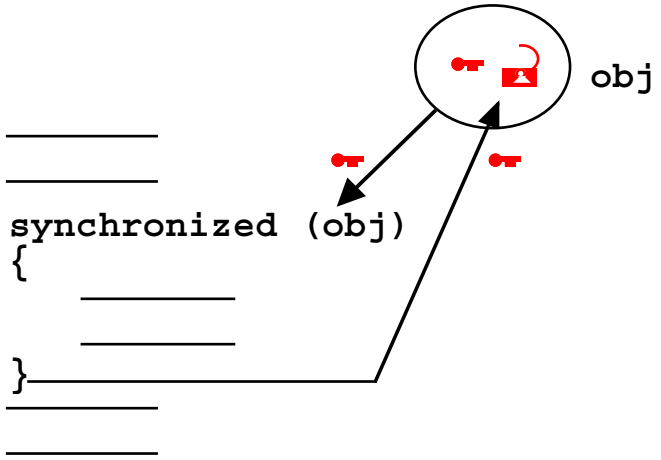
Problèmes de debugging et de portage

Commence à être utilisée dans les noyaux d’OS (BeOS, Linux)

Introduction dans les langages :
Java, CML, OCaml, ...

Et en Java ?

- **API de threads : la classe `Thread`**
- **notion de code synchronisé : `synchronized`**

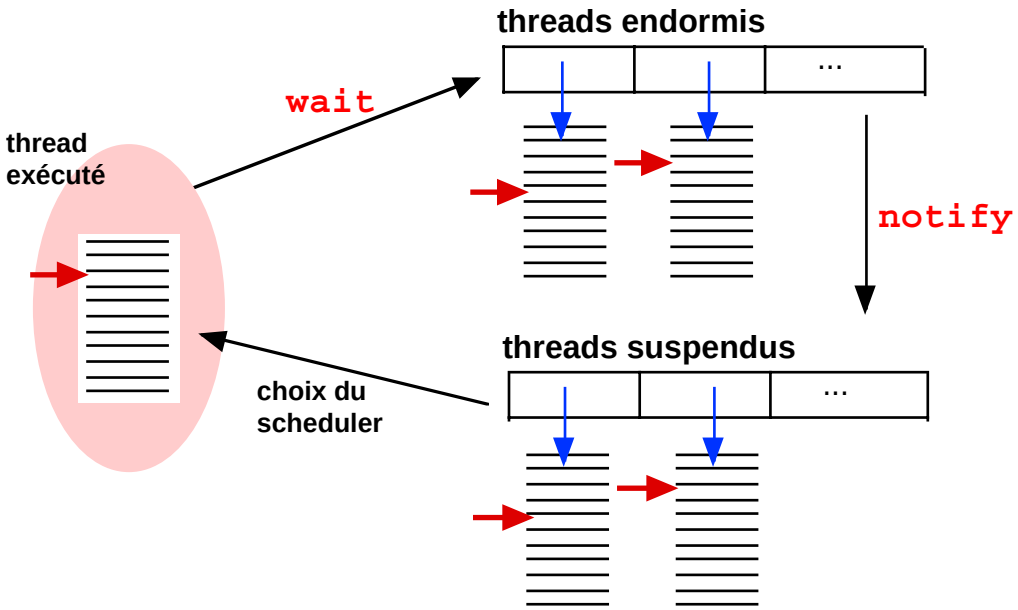


Aucune hypothèse sur la façon dont la JVM schedule les threads : portabilité...

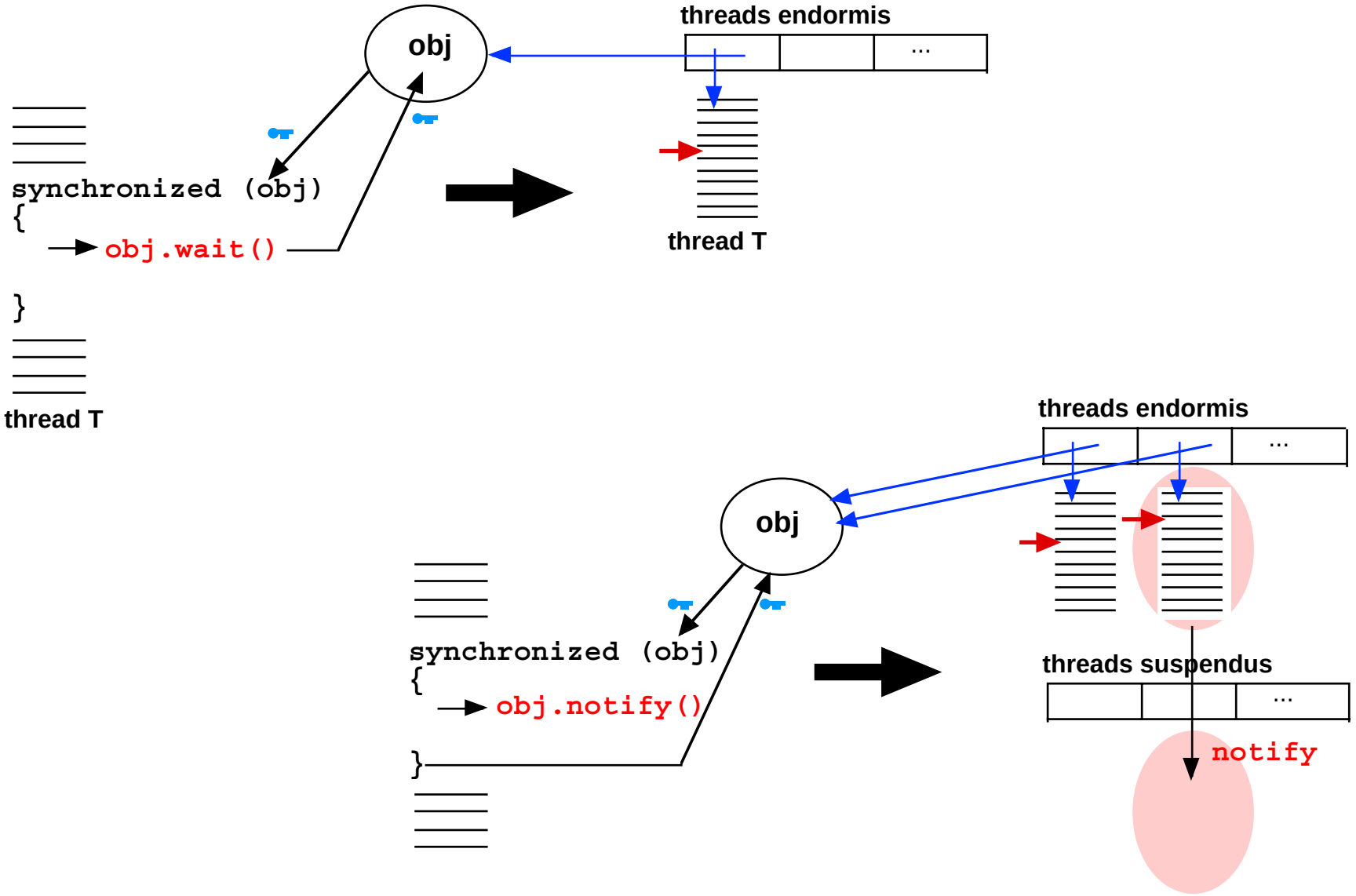
Omniprésence des threads : AWT, RMI

Wait & notify

Pour éviter l'attente active sur les locks : endormir les threads



Wait & notify - 2

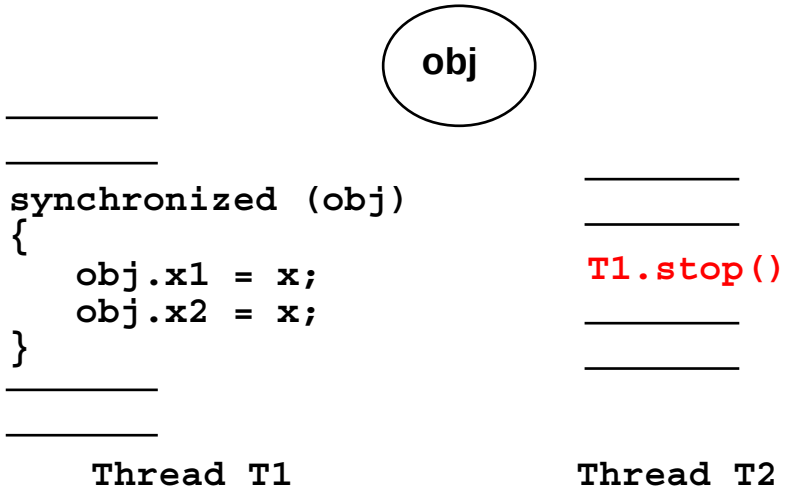


Contrôle fin des threads

Thread.stop() : terminaison du thread

Thread.suspend() : suspension du thread

Thread.resume() : reprise du thread



- choix de T1
- prise du lock sur obj
- première affectation
- préemption de T1
- choix de T2
- T1 est stoppé
- relâchement du lock sur obj

Etat incohérent : x1 et x2 n'ont pas même valeur

"Deprecated" à partir de Java 1.2 !

Autres primitives

```
class T extends Thread
{
  public void run(){
     ....
  }
}

Thread t = new T();
```

```
class T implements Runnable
{
  public void run(){
     ....
  }
}

Thread t = new Thread(new T());
```

- notifyAll
- join(Thread)
- yield()
- sleep(long)
- wait(long)
- join(Thread,long)
- ...

Conclusion

- **Les threads sont partout...**
- **Programmation dans les pires conditions : on ne peut avoir aucune assurance sur la stratégie de scheduling. Portabilité problématique...**
- **Pas de contrôle fin sur l'exécution des threads. Rend difficile la construction de schedulers particuliers.**

Références

Pthreads : standard IEEE POSIX 1003.1c
LinuxThreads (Xavier Leroy)
Gnu Portable Threads

***Pthreads Programming*,**
B. Nichols, D. Buttlar, J. Proulx Farrell, O'Reilly, 1996.

***Programming Java threads in the real world*,**
A. Hollub, JavaWorld, 1998, available at:
http://www.javaworld.com/jw-09-1998/jw-09-threads.html