Algorithme de Peterson (1/5)

```
class Peterson extends Thread {
  static int tour = 0;
  static boolean[ ] actif  = {false, false};
  int i, j;
  Peterson (int x) { i = x; j = 1 - x; }

  public void run() {
    while (true) {
      actif[i] = true;
      tour = j;
      while (actif[j] && tour  == j)
        ;
      // section critique
      actif[i] = false;
  } }

  public static void main (String[ ] args) {
    Thread t0 = new Peterson(0), t1 = new Peterson(1);
    t0.start(); t1.start();
} }
```


Algorithme de Peterson (2/5)

Preuve de sureté. (*safety*)

Si t_0 et t_1 sont tous les deux dans leur section critique. Alors $actif[0] = actif[1] = true$.

Impossible car les deux tests auraient été franchis en même temps alors que $tour$ favorise l'un deux. Donc un seul est entré. Disons t_0.

Cela veut dire que t_1 n'a pu trouver le $tour$ à 1 et n'est pas entré en section critique.

Preuve de vivacité. (*lifeness*)

Supposons t_0 bloqué dans le while.

Cas 1 : t_1 non intéressé à rentrer dans la section critique. Alors $actif[1] = false$. Et donc t_0 ne peut être bloqué par le while.

Cas 2 : t_1 est aussi bloqué dans le while. Impossible car selon la valeur de $tour$, l'un de t_0 ou t_1 ne peut rester dans le while.

Algorithme de Peterson (3/5)

- avec des assertions où on fait intervenir la ligne des programmes c_0 et c_1 (compteur ordinal) exécutée par t_0 et t_1

```
.  public void run() {
.    while (true) {
       {¬ actif[i] ∧ c_i ≠ 2}
1      actif[i] = true;
       {actif[i] ∧ c_i = 2}
2      tour = j;
       {actif[i] ∧ c_i ≠ 2}
3      while (actif[j] && tour == j)
.        ;
       {actif[i] ∧ c_i ≠ 2 ∧ (¬ actif[j] ∨ tour = i ∨ c_j = 2)}
.      // section critique
.      ...
.      // fin de section critique
5      actif[i] = false;
       {¬ actif[i] ∧ c_i ≠ 2}
6      Thread.yield();
.  } }
```


Algorithme de Peterson (4/5)

Preuve de sureté :

- Si t_0 et t_1 sur la ligne 5, on a :

$$\begin{aligned} & actif[0] \ \wedge\ c_0 \neq 2 \ \wedge\ (\neg\, actif[1] \vee tour = 0 \vee c_1 = 2) \\ \wedge \quad & actif[1] \ \wedge\ c_1 \neq 2 \ \wedge\ (\neg\, actif[0] \vee tour = 1 \vee c_0 = 2) \end{aligned}$$

  équivaut à

$$\begin{aligned} & actif[0] \wedge c_0 \neq 2 \wedge tour = 1 \\ \wedge \quad & actif[1] \wedge c_1 \neq 2 \wedge tour = 0 \end{aligned}$$

  équivaut à

$$tour = 0 \wedge tour = 1 \wedge P$$

  implique

$$tour = 0 \wedge tour \neq 0 \wedge P \ \equiv\ \text{false}$$

  impossible.

- Preuve par énumération des cas (*model checking*)

Algorithme de Peterson (5/5)

Preuve de vivacité.

- Si t_0 et t_1 dans la boucle while :

$$\begin{array}{ll} & \mathit{actif}[1] \wedge \mathit{tour} = 1 \wedge \mathit{actif}[0] \wedge \mathit{tour} = 0 \\ \equiv & \mathit{tour} = 0 \wedge \mathit{tour} = 1 \wedge P \\ \equiv & \text{false} \end{array}$$

- Si t_0 en dehors de la boucle while et t_1 dedans, la preuve se complique car faisant intervenir l'évolution dans le temps (logique temporelle ou modale).
  Par exemple :

$$\neg\, \mathit{actif}[0] \wedge \mathit{tour} = 0 \wedge \mathit{actif}[0] \wedge c_0 = 2$$

  équivaut à

$$\neg\, \mathit{actif}[0] \wedge \mathit{tour} = 0 \wedge c_0 = 2$$

  alors le programme évolue vers $\mathit{tour} = 1$ et le tout devient faux. On quitte donc la boucle while.

Exercices

Exercice 1 Dans le langage Ada, la communication se fait par rendez-vous. Deux processus P et Q font un rendez-vous s'ils s'attendent mutuellement chacun à un point de son programme. On peut en profiter pour passer une valeur. Comment organiser cela en Java ?

Exercice 2 Une barrière de synchronisation entre plusieurs processus est un endroit commun que tous les processus actifs doivent franchir simultanément. C'est donc une généralisation à n processus d'un rendez-vous. Comment la programmer en Java ?

Exercice 3 Toute variable peut être partagée entre plusieurs processus, quel est le gros danger de la communication par variables partagées ?

Exercice 4 Un ordonnanceur est juste s'il garantit à tout processus prêt de passer dans l'état exécution. Comment le construire ?

Exercice 5 (difficile) Généraliser l'algorithme de Peterson au cas de n processus ($n \geq 2$).