Plan

Comment ?

- Solutions logicielles utilisant de l'attente active : tester en permanence la possibilité d'entrer
- Mécanismes matériels
  - simplifiant l'attente active (instructions spécialisées)
  - évitant l'attente active (masquage des interruptions)
- Primitives du système d'exploitation/d'exécution

Forme générale

```
Variables partagées par toutes les activités
Activité A_i
    entrée
    section critique
    sortie
```

Une fausse solution

♪♪♪

Algorithme

```
occupé : shared boolean := false;

tant que occupé faire nop;
occupé ← true;
        section critique
occupé ← false;
```

(Test-and-set non atomique)

Alternance

♪♪♪

Algorithme

tour : shared 0..1;

tant que *tour* $\neq$ i faire nop;

section critique

tour $\leftarrow$ i + 1 mod 2;

- note : **i** = identifiant de l'activité demandeuse
- deux activités (généralisable à plus)
- lectures et écritures atomiques
- alternance obligatoire

Priorité à l'autre demandeur

♪♪♪

Algorithme

```
demande : shared array 0..1 of boolean;

demande[i] ← true;
tant que demande[j] faire nop;
    section critique
demande[i] ← false;
```

- **i** = identifiant de l'activité demandeuse
  j = identifiant de l'autre activité
- deux activités (non facilement généralisable)
- lectures et écritures atomiques
- risque d'attente infinie (interblocage)

Peterson 1981

♪♪♪

Algorithme

```
demande: shared array 0..1 of boolean := [false,false];
tour : shared 0..1;

demande[i] ← true;
tour ← j;
tant que (demande[j] et tour = j) faire nop;
     section critique
demande[i] ← false;
```

- deux activités (non facilement généralisable)
- lectures et écritures atomiques
- évaluation non atomique du « et »
- vivacité individuelle

Solution pour N activités (Lamport 1974) ♪♪♪

L'algorithme de la boulangerie

```
int num[N];          // numéro du ticket
boolean choix[N];    // en train de déterminer le n°

choix[i] ← true;
int tour ← 0;  // local à l'activité
pour k de 0 à N faire tour ← max(tour,num[k]);
num[i] ← tour + 1;
choix[i] ← false;

pour k de 0 à N faire
  tant que (choix[k]) faire nop;
  tant que (num[k]≠0) ∧ (num[k],k)≺(num[i],i) faire nop;
section critique
num[i] ← 0;
```

Instruction matérielle TestAndSet

Retour sur la fausse solution avec test-and-set non atomique de la variable *occupé* (page 17).

Soit TestAndSet(x), instruction indivisible qui positionne x à vrai et renvoie l'ancienne valeur :

Définition

```
function TestAndSet (x : in out boolean) : boolean
     declare oldx : boolean
begin
     oldx := x; x := true;
     return oldx;
end TestAndSet
```

Utilisation du TestAndSet

Alors : protocole d'exclusion mutuelle :

Algorithme

```
occupé : shared boolean := false;

tant que TestAndSet(occupé) faire nop;
    section critique
occupé ← false;
```

Tous les processeurs actuels possèdent une instruction analogue au TestAndSet, et adaptée aux multiprocesseurs symétriques.

Instruction FetchAndAdd

Définition

```
function FetchAndAdd (x : in out int) : int
     declare oldx : int
begin
     oldx := x; x := oldx + 1;
     return oldx;
end FetchAndAdd
```

```
ticket : shared int := 0;
tour : shared int := 0;
montour : int;   // local à l'activité
montour ← FetchAndAdd(ticket);
tant que tour ≠ montour faire nop;
     section critique
FetchAndAdd(tour);
```

Spinlock x86

♪♪♪

Spinlock Linux 2.6

```
; initialement Lock = 1
acquire: lock dec word [Lock]
         jns cs                  ; jump if not signed
   spin: cmp dword [Lock], 0
         jle spin                ; loop if ≤ 0
         jmp acquire             ; retry entry
     cs:                         ; section critique
release: mov dword [Lock], 1
```

`lock dec` = décrémentation atomique multiprocesseur avec positionnement du bit "sign"

Masquage des interruptions

Éviter la préemption du processeur par une autre activité :

Algorithme

```
masquer les interruptions
   section critique
démasquer les interruptions
```

- plus d'attente !
- mono-processeur seulement
- pas d'entrée-sortie, pas de défaut de page, pas de blocage dans la section critique

$\rightarrow$ μ-système embarqué

Le système d'exploitation

1. Contrôle de la préemption
2. Contrôle de l'exécution des activités
3. « Effet de bord » d'autres primitives

Ordonnanceur avec priorités

Ordonnanceur (scheduler) d'activités avec priorité fixe : l'activité de plus forte priorité s'exécute, sans préemption possible.

Algorithme

```
priorité ← priorité max // pas de préemption possible
   section critique
priorité ← priorité habituelle // avec préemption
```

- mono-processeur seulement
- les activités non concernées sont aussi pénalisées
- entrée-sortie ? mémoire virtuelle ?

→ système embarqué

Éviter l'attente active : contrôle des activités ♪♪♪

Algorithme

```
occupé : shared bool := false;
demandeurs : shared fifo;
```

```
bloc atomique
     si occupé alors
          self ← identifiant de l´ activité courante
          ajouter self dans demandeurs
          se suspendre
     sinon
          occupé ← true
     finsi
fin bloc
```

section critique

```
bloc atomique
     si demandeurs est non vide alors
          p ← extraire premier de demandeurs
          débloquer p
     sinon
          occupé ← false
     finsi
fin bloc
```

Le système de fichiers (!)

Pour jouer : effet de bord d'une opération du système d'exploitation qui réalise une action atomique analogue au TestAndSet, basée sur l'existence et la création d'un fichier.

Algorithme

```
tant que
    open("toto", O_RDONLY | O_EXCL | O_CREAT, 0) == -1
    // échec si le fichier existe déjà; sinon il est créé
faire nop;
    section critique
unlink("toto");
```

- ne nécessite pas de mémoire partagée
- atomicité assurée par le noyau d'exécution

La réalité

Actuellement, tout environnement d'exécution fournit un mécanisme de verrou (*lock*), avec les opérations atomiques :

- obtenir (*acquire*) : si le verrou est libre, l'attribuer à l'activité demandeuse ; sinon bloquer l'activité demandeuse
- rendre/libérer (*release*) : si au moins une activité est en attente du verrou, transférer la possession à l'un des demandeurs et le débloquer ; sinon marquer le verrou comme libre.

Algorithme

```
accès : shared lock
accès.acquire
    section critique
accès.release
```