Systèmes concurrents

2SN

17 septembre 2020

UE Systèmes concurrents et communicants

3 matières

- Systèmes concurrents : modèles, méthodes, outils pour le parallélisme « local »
- Intergiciels : mise en œuvre du parallélisme dans un environnement réparti (machines distantes)
- Projet données réparties : réalisation d'un service de support à la programmation concurrente, parallèle ou répartie.

Evaluation de l'UE

- Examen Systèmes concurrents : écrit, sur la conception de systèmes concurrents
- *(Examen Intergiciels : écrit)*
- Projet commun : réalisation d'un service de support à la programmation concurrente, parallèle ou répartie.
  - présentation mi-octobre, rendu final mi janvier
  - travail en groupe de 4, suivi + points d'étape réguliers

Matière : systèmes concurrents – organisation

Composition

- Cours (50%) : définitions, principes, modèles
- TD (25%) : conception et méthodologie
- TP (25%) : implémentation des schémas et principes

Fonctionnement

- TDs : classique (si présentiel)
- Cours, TP : à distance, style classe inversée
  travail en amont de la séance (avec retour), séance en semi-autonomie

Evaluation

- Si examen standard : ecrit + bonus (rendus TPs, Quiz)
- Si examen à distance contrôle continu (rendus TPs, quiz) + petit examen en ligne

Pages de l'enseignement : `http://moodle-n7.inp-toulouse.fr`
Contact : `mauran@enseeiht.fr`, `queinnec@enseeiht.fr`

Objectifs

Objectif

Être capable de comprendre et développer des applications parallèles (*concurrentes*)

- → modélisation pour la conception de programmes parallèles
- → connaissance des schémas (patrons) essentiels
- → raisonnement sur les programmes parallèles : exécution, propriétés
- → pratique de la programmation parallèle avec un environnement proposant les objets/outils de base

Plan du cours

1. Introduction : problématique
2. Exclusion mutuelle
3. Synchronisation à base de sémaphores
4. Interblocage
5. Synchronisation à base de moniteur
6. API Java, Posix Threads
7. Processus communicants – Go, Ada
8. Transactions – mémoire transactionnelle
9. Synchronisation non bloquante

Première partie

Introduction

Contenu de cette partie

- nature et particularités des programmes concurrents
  $\Rightarrow$ conception et raisonnement systématiques et rigoureux
- modélisation des systèmes concurrents
- points clés pour faciliter la conception des applications concurrentes
- intérêt et limites de la programmation parallèle
- mise en œuvre de la programmation concurrente sur les architectures existantes

Plan

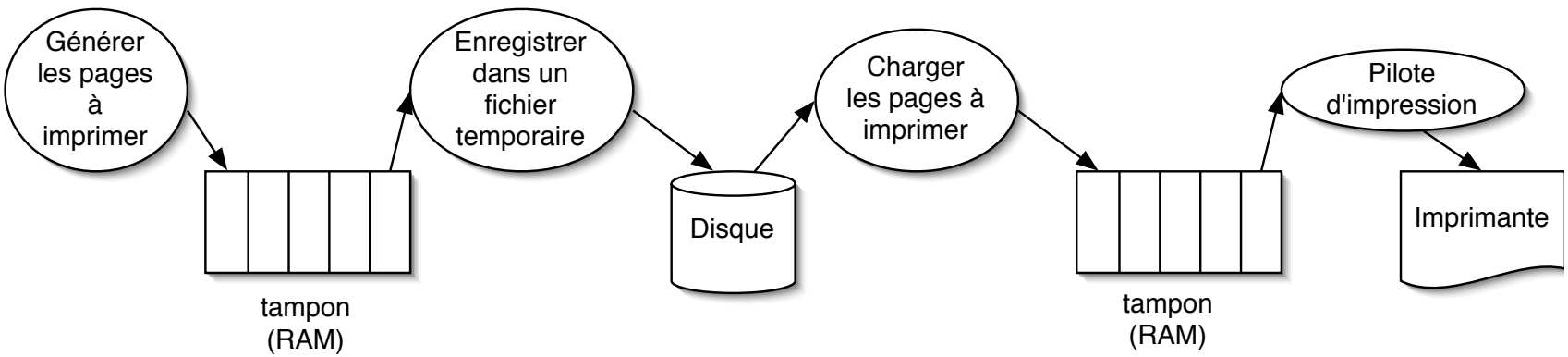
Le problème

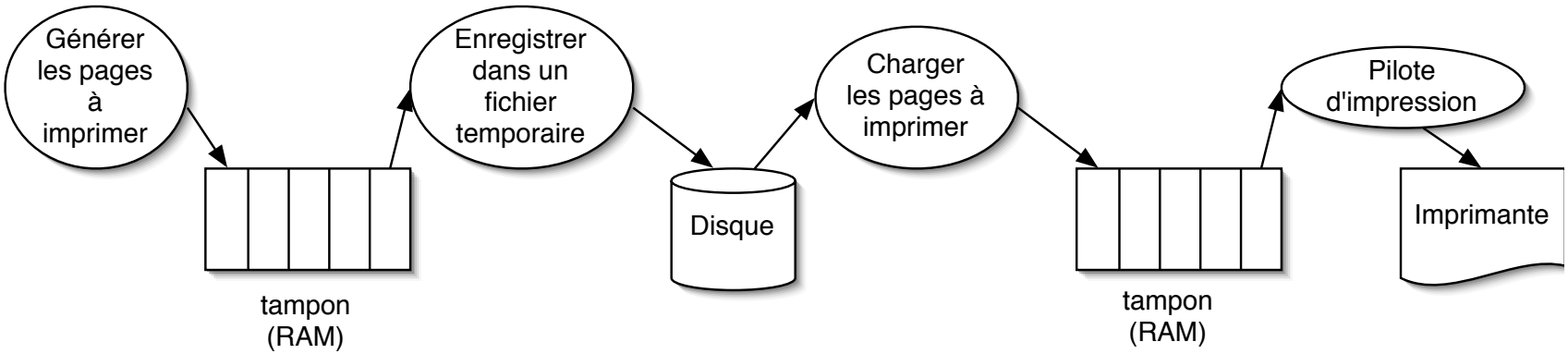
Système concurrent

Ensemble de processus s'exécutant simultanément

- en compétition pour l'utilisation de ressources partagées
- et/ou contribuant à l'obtention d'un résultat commun (global)

Exemple : service d'impression différée

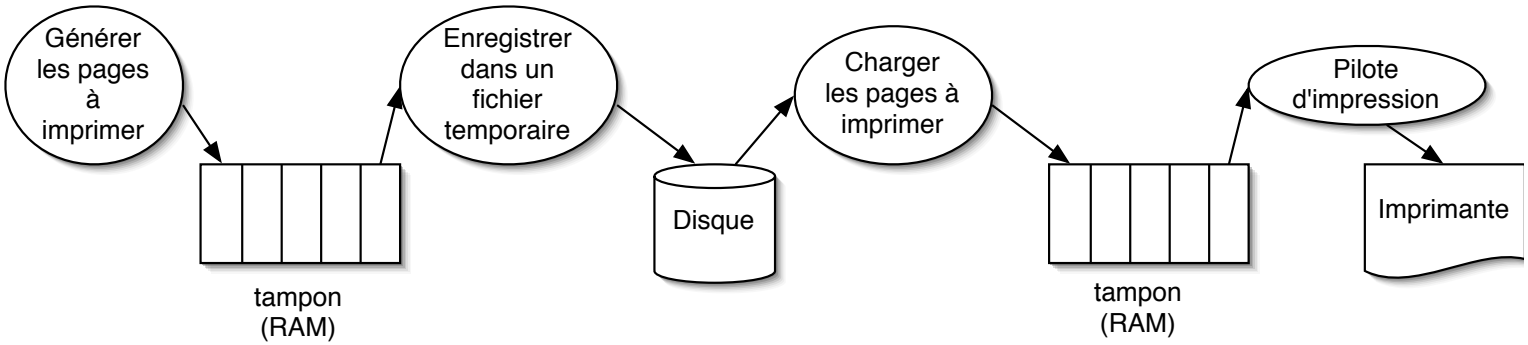




Conception : parallélisation d'un traitement

- décomposition en traitements séquentiels (*processus*)
- exécution simultanée (*concurrente*)
- les processus concurrents ne sont pas indépendants :
  ils partagent des objets (ressources, données)
  $\Rightarrow$ spécifier et contrôler les interactions entre processus

Relations entre activités composées



Chaque activité progresse à son rythme, avec une vitesse arbitraire

$\Rightarrow$ nécessité de réaliser un couplage des activités interdépendantes

- fort : arrêt/reprise des activités «en avance» (*synchronisation*)
- faible : stockage des données échangées et non encore utilisées (*schéma producteur/consommateur*)

Expression du contrôle des interactions : 2 niveaux d'abstraction

- *coopération* (dépôt/retrait sur le tampon) :
  les activités « se connaissent » (interactions explicites)
- *compétition* (accès au disque) :
  les activités « s'ignorent » (interactions transparentes)

Intérêt de la programmation concurrente

Intérêt de la programmation concurrente

- Facilité de conception
  le parallélisme est naturel sur beaucoup de systèmes
  - temps réel : systèmes embarqués, applications multimédia
  - mode de fonctionnement : modélisation et simulation de systèmes physiques, d'organisations, systèmes d'exploitation
- Pour accroître la puissance de calcul
  algorithmique parallèle et répartie
- Pour faire des économies
  mutualisation de ressources coûteuses via un réseau
- Parce que la technologie est mûre
  banalisation des systèmes multi-processeurs, des stations de travail/ordinateurs en réseau, services répartis

Nécessité de la programmation concurrente

- La puissance de calcul monoprocesseur atteint un plafond
  - l'augmentation des performances d'un processeur dépend directement de sa fréquence d'horloge f
    - l'énergie consommée et dissipée augmente comme f^3
      $\rightarrow$ une limite physique est atteinte depuis quelques années
  - les gains de parallélisme au niveau du processeur sont limités
    - processeurs vectoriels, architectures pipeline conviennent mal à des calculs irréguliers/généraux
  - coût excessif de l'augmentation de la taille des caches qui permettrait de compenser l'écart croissant de performances entre processeurs et mémoire
- La loi de Moore reste valide :
  la densité des transistors double tous les 18 à 24 mois

$\rightarrow$ les architectures multiprocesseurs sont pour l'instant le principal moyen d'accroître la puissance de calcul

Qu'est-ce qui fait que la programmation concurrente est différente de la programmation séquentielle ?

```
                variables globales : s, i
        P1                                  ||
s := 0                                      ||
pour i:= 1 à 10 pas 1                       ||
    s := s+i                                ||
fin_pour                                    ||
afficher(s,i)                               ||
```

- P1 seul → 12 états 

Qu'est-ce qui fait que la programmation concurrente est différente de la programmation séquentielle ?

- plusieurs activités simultanées ⇒ explosion de l'espace d'états

```
          variables globales : s, i
     P1                          ||      P2
s := 0                           ||   s := 0
pour i:= 1 à 10 pas 1            ||   pour i:= 1 à 10 pas 1
   s := s+i                      ||      s := s+i
fin_pour                         ||   fin_pour
afficher(s,i)                    ||   afficher(s,i)
```

  - P1 seul → 12 états 
  - P1 ‖ P2 → 12 x 12 = 144 états 

Qu'est-ce qui fait que la programmation concurrente est différente de la programmation séquentielle ?

- plusieurs activités simultanées ⇒ explosion de l'espace d'états

```
          variables globales : s, i
     P1                          ||      P2
s := 0                           ||   s := 0
pour i:= 1 à 10 pas 1            ||   pour i:= 1 à 10 pas 1
   s := s+i                      ||      s := s+i
fin_pour                         ||   fin_pour
afficher(s,i)                    ||   afficher(s,i)
```

  - P1 seul → 12 états 
  - P1 ‖ P2 → 12 x 12 = 144 états 

- interdépendance des activités
  - logique : production/utilisation de résultats intermédiaires
  - chronologique : disponibilité des résultats

  ⇒ non déterminisme (⇒ difficulté du raisonnement par scénarios)

Qu'est-ce qui fait que la programmation concurrente est différente de la programmation séquentielle ?

- plusieurs activités simultanées ⇒ explosion de l'espace d'états

```
        variables globales : s, i
     P1                          ||     P2
s := 0                           ||  s := 0
pour i:= 1 à 10 pas 1            ||  pour i:= 1 à 10 pas 1
   s := s+i                      ||     s := s+i
fin_pour                         ||  fin_pour
afficher(s,i)                    ||  afficher(s,i)
```

  - P1 seul → 12 états
  - P1 ‖ P2 → 12 x 12 = 144 états

- interdépendance des activités
  - logique : production/utilisation de résultats intermédiaires
  - chronologique : disponibilité des résultats

  ⇒ non déterminisme (⇒ difficulté du raisonnement par scénarios)

⇒ nécessité d'outils (conceptuels et logiciels) pour assurer le raisonnement et le développement

Plan

Modèle d'exécution

Activité (ou : processus, processus léger, thread, tâche...)

- Représente l'activité d'exécution d'un programme séquentiel par un processeur
- Vision simple (simplifiée) : à chaque cycle, le processeur
  - extrait (lit et décode) une instruction machine à partir d'un flot séquentiel (le code exécutable),
  - exécute cette instruction,
  - puis écrit le résultat éventuel (registres, mémoire RAM).

$\rightarrow$ exécution d'un processus P

$=$ suite d'instructions effectuées $p_1; p_2; \ldots p_n$ (*histoire* de P)

Exécution concurrente

L'exécution concurrente (simultanée) d'un ensemble de processus $(P_i)_{i \in I}$ est représentée comme une exécution consistant en un *entrelacement arbitraire* des histoires de chacun des processus P_i

Exemple : 2 processus P=$p_1; p_2; p_3$ et Q=$q_1; q_2$
L'exécution concurrente de P et de Q sera vue comme (équivalente à) l'une des exécutions suivantes :

$p_1; p_2; p_3; q_1; q_2$ ou $p_1; p_2; q_1; p_3; q_2$ ou $p_1; p_2; q_1; q_2; p_3$ ou
$p_1; q_1; p_2; p_3; q_2$ ou $p_1; q_1; p_2; q_2; p_3$ ou $p_1; q_1; q_2; p_2; p_3$ ou
$q_1; p_1; p_2; p_3; q_2$ ou $q_1; p_1; p_2; q_2; p_3$ ou $q_1; p_1; q_2; p_2; p_3$ ou
$q_1; q_2; p_1; p_2; p_3$

Le modèle d'exécution par entrelacement est il réaliste ?

Le modèle d'exécution par entrelacement est il réaliste ?

Abstraction réalisée

Deux instructions a et b de deux processus différents ayant une période d'exécution commune donnent un résultat identique à celui de $a; b$ ou de $b; a$

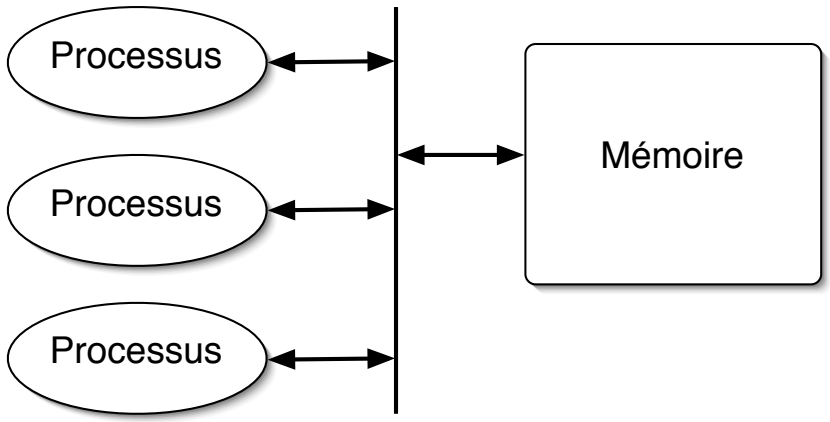
Motivation

- abstrait (ignore) les possibilités de chevauchement dans l'exécution des opérations
  $\Rightarrow$ on se ramène à un ensemble *discret* de possibilités (espace d'états/produit d'histoires)
- entrelacement *arbitraire* : pas d'hypothèse sur la vitesse relative de progression des activités
  $\Rightarrow$ modélise l'hétérogénéité et la charge des processeurs
- abstraction $\ll$ raisonnable $\gg$ au regard des architectures réelles (voir dernière section)

Modèles d'interaction : interaction par mémoire partagée

Système centralisé multi-tâches

- communication implicite, résultant de l'accès par chaque processus à des variables partagées
- processus anonymes (interaction sans identification)
- coordination (synchronisation) nécessaire (pour déterminer l'instant où une interaction est possible)



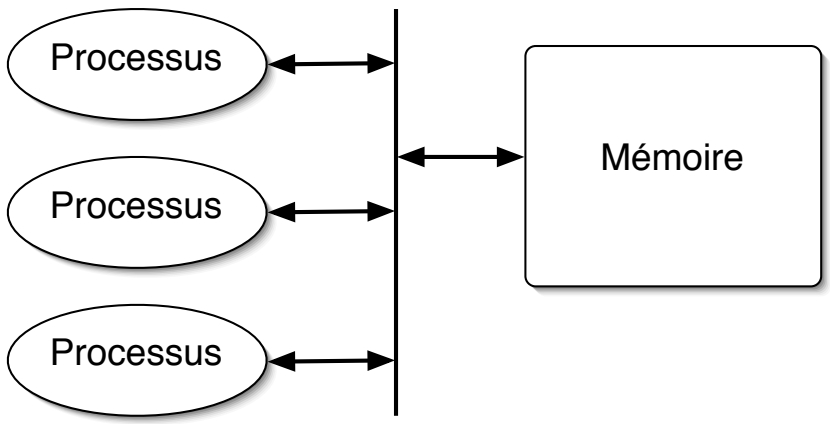
Exemples

- multiprocesseurs à mémoire partagée,
- processus légers,
- Unix : couplage mémoire (mmap), fichiers

Modèles d'interaction : interaction par mémoire partagée

Système centralisé multi-tâches

- communication implicite, résultant de l'accès par chaque processus à des variables partagées
- processus anonymes (interaction sans identification)
- coordination (synchronisation) nécessaire (pour déterminer l'instant où une interaction est possible)



Exemples

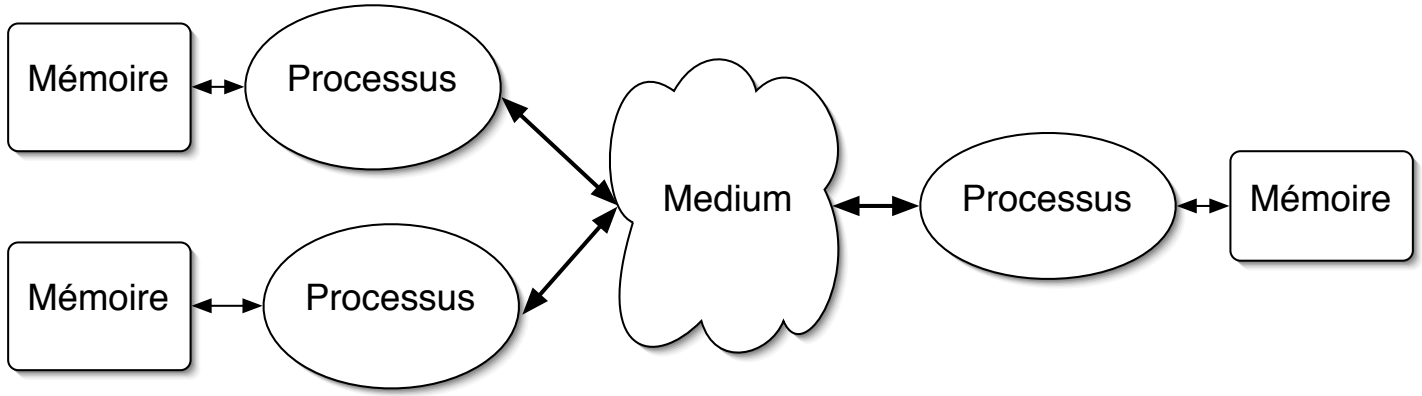
- multiprocesseurs à mémoire partagée,
- processus légers,
- Unix : couplage mémoire (mmap), fichiers

Modèles d'interaction : échange de messages

Processus communiquant par messages

Système réparti

- communication explicite par transfert de données (messages)
- désignation nécessaire du destinataire
- coordination implicite, découlant de la communication



Exemples

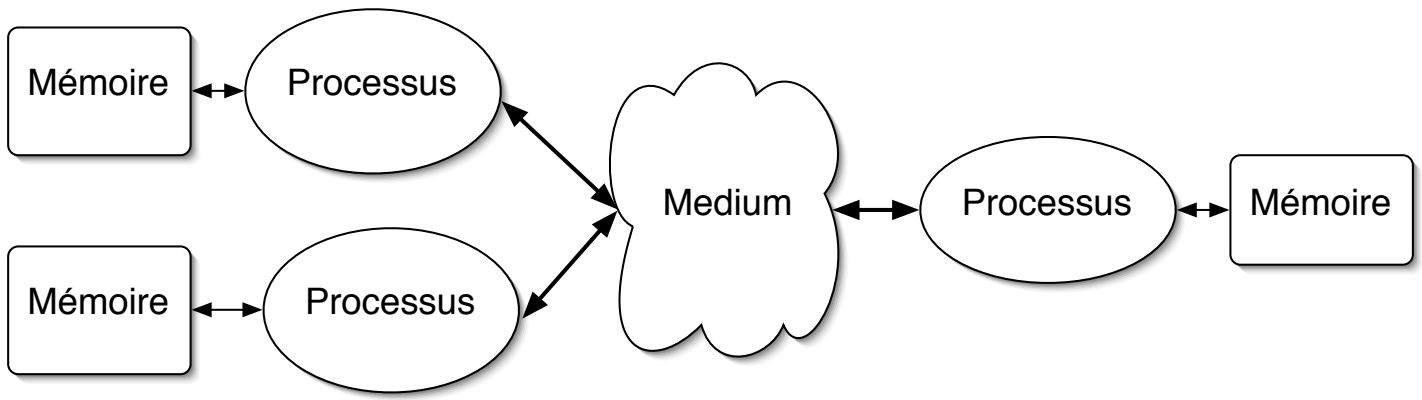
- processeurs en réseau,
- architectures logicielles réparties (client/serveur. . .),
- Unix : tubes, signaux

Modèles d'interaction : échange de messages

Processus communiquant par messages

Système réparti

- communication explicite par transfert de données (messages)
- désignation nécessaire du destinataire
- coordination implicite, découlant de la communication



Exemples

- processeurs en réseau,
- architectures logicielles réparties (client/serveur. . .),
- Unix : tubes, signaux

Spécifier un programme

Pourquoi ?

Difficulté à raisonner sur les systèmes concurrents
(explosion combinatoire de l'espace d'états/des histoires possibles)

Comment ?

Approche classique : donner les propriétés souhaitées du système, puis vérifier que ces propriétés sont valides lors des exécutions

Spécifier un programme

Pourquoi ?

Difficulté à raisonner sur les systèmes concurrents
(explosion combinatoire de l'espace d'états/des histoires possibles)

Comment ?

Approche classique : donner les propriétés souhaitées du système, puis vérifier que ces propriétés sont valides lors des exécutions

Particularité : calculs **interdépendants** et/ou **réactifs**

→ propriétés fonctionnelles (S=f(E)) insuffisantes/inappropriées
→ propriétés sur l'évolution des traitements, au fil du temps

Spécifier un programme

Pourquoi ?

Difficulté à raisonner sur les systèmes concurrents
(explosion combinatoire de l'espace d'états/des histoires possibles)

Comment ?

Approche classique : donner les propriétés souhaitées du système, puis vérifier que ces propriétés sont valides lors des exécutions

Particularité : calculs **interdépendants** et/ou **réactifs**

→ propriétés fonctionnelles (S=f(E)) insuffisantes/inappropriées
→ propriétés sur l'évolution des traitements, au fil du temps

- Un programme est caractérisé par l'ensemble de ses exécutions possibles
- exécution = histoire, suite d'instructions/d'états
  (état = valeur des variables)

→ propriétés d'un programme = propriétés de ses histoires possibles

Propriété d'une histoire (suite d'états)

Validité d'un prédicat d'état

- à chaque étape de l'exécution :
  propriété de sûreté (il n'arrive jamais rien de mal)
- après un nombre de pas fini :
  propriété de vivacité (une bonne chose finit par arriver)

Exemple

- Sûreté : *Deux serveurs ne prennent jamais le même travail.*
- Vivacité : *Un travail déposé finit par être pris par un serveur*

Remarque : les propriétés exprimées peuvent porter sur

- toutes les exécutions du programme (logique temporelle linéaire)
- ou seulement certaines exécutions du programme (LT arborescente)

Les propriétés que nous aurons à considérer se limiteront généralement au cadre (plus simple) de la LT linéaire.

Vérifier les propriétés : analyse des exécutions

Définition de l'effet d'une opération : triplets de Hoare

{précondition} Opération {postcondtion}

- précondition (hypothèse) :
  propriété devant être vérifiée avant l'exécution de l'opération
- postcondition (conclusion) :
  propriété garantie par l'exécution de l'opération

Exemple

$\{t = \text{nb requêtes en attente} \wedge t > 0 \wedge r = \text{nb résultats}\}$
le serveur traite une requête
$\{\text{nb requêtes en attente} = t - 1 \wedge \text{nb résultats} = r + 1\}$

Analyse d'une exécution

- partir d'une propriété (hypothèse) caractérisant l'état initial
- appliquer en séquence les opérations de l'histoire :
  propriété établie par l'exécution d'une op. = précondition de l'op. suivante

Vérifier les propriétés : analyse des exécutions

Définition de l'effet d'une opération : triplets de Hoare

{précondition} Opération {postcondtion}

- précondition (hypothèse) :
  propriété devant être vérifiée avant l'exécution de l'opération
- postcondition (conclusion) :
  propriété garantie par l'exécution de l'opération

Exemple

$\{t = \text{nb requêtes en attente} \wedge t > 0 \wedge r = \text{nb résultats}\}$
le serveur traite une requête
$\{\text{nb requêtes en attente} = t - 1 \wedge \text{nb résultats} = r + 1\}$

Analyse d'une exécution

- partir d'une propriété (hypothèse) caractérisant l'état initial
- appliquer en séquence les opérations de l'histoire :
  propriété établie par l'exécution d'une op. = précondition de l'op. suivante

Analyse des exécutions : propriétés d'actions concurrentes

Propriétés établies par la combinaison des actions (exemples)

Sérialisation (sémantique de l'entrelacement) :

$$\frac{\{p\}A_1; A_2\{q_{12}\},\ \{p\}A_2; A_1\{q_{21}\}}{\{p\}A_1 \parallel A_2\{q_{12} \vee q_{21}\}}$$

Indépendance (des effets de calculs séparés) :

$$\frac{A_1 \text{et } A_2 \text{ sans interférence},\ \{p\}A_1\{q_1\},\ \{p\}A_2\{q_2\}}{\{p\}A_1 \parallel A_2\{q_1 \wedge q_2\}}$$

Plan

Conception des systèmes concurrents

Point clé :
contrôler les effets des interactions/interférences entre processus

- isoler (raisonner indépendamment) $\rightarrow$ modularité
- contrôler/spécifier l'interaction
  - définir les instants où l'interaction est possible
  - relier ces instants au flot d'exécution de chacun des processus

Modularité : pouvoir raisonner sur chaque activité séparément

Atomicité

mécanisme/protocole garantissant qu'une (série d')opération(s) est exécutée complètement et sans interférence (isolément)

- grain fin (instruction)
  - (modèle) utile pour le raisonnement : entrelacement
  - (matériel) utile pour déterminer un résultat en cas de conflit
- gros grain (bloc d'instructions) : utile pour la conception.

Réalisation directe :

Modularité : pouvoir raisonner sur chaque activité séparément

Atomicité

mécanisme/protocole garantissant qu'une (série d')opération(s) est exécutée complètement et sans interférence (isolément)

- grain fin (instruction)
  - (modèle) utile pour le raisonnement : entrelacement
  - (matériel) utile pour déterminer un résultat en cas de conflit
- gros grain (bloc d'instructions) : utile pour la conception.

Réalisation directe :

exclusion mutuelle (bloquer tous les processus sauf 1)

- verrous
- masquage des interruptions (sur un monoprocesseur)
- ...

Contrôle des interactions : synchronisation

Mise en œuvre : attente

Un processus prêt pour une interaction est mis en attente (bloqué), jusqu'à ce que **tous** les processus participants soient prêts.

Expression

- en termes de
  - flot de contrôle : placer un *point de synchronisation commun* dans le code de chacun des processus d'un groupe de processus. Ce point de synchronisation définira un instant d'exécution commun à ces processus.
  - flot de données : définir les *échanges* de données entre processus (émission/réception de messages, ou d'événements). L'ordonnancement des processus suit la circulation de l'information.
- globale (barrière, événements, invariants) ou individuelle (rendez-vous, canaux)

Comment pouvoir raisonner sur chaque interaction séparément ? (1/3)

Principe

Définir les interactions permises, indépendamment des calculs

Première idée

Spécifier les suites d'interactions possibles (légales) pour les activités

→ grammaire définissant les suites d'opérations (interactions) permises (expressions de chemins)

→ moyen de vérifier de manière simple et indépendante du code des processus si 1 exécution (trace) globale est correcte (légale)

Exemple : interaction client/serveur

A tout moment, *nb d'appels à déposer_tâche* $\geq$ *nb d'appels à traiter_tâche*

Difficulté

Composition (ajout/retrait d'opérations $\Rightarrow$ redéfinir les suites)

Comment pouvoir raisonner sur chaque interaction séparément ? (2/3)

Deuxième étape

Définir les interactions permises, indépendamment des opérations

Idée

Les processus doivent se synchroniser parce qu'il **partagent** un objet

- à construire (coopération)
- à utiliser (concurrence)

→ spécifier un objet partagé, caractérisé par un ensemble d'états possibles (légaux) : invariant portant sur l'état de l'objet partagé

Exemple : *la file des travaux à traiter peut contenir de* 0 *à Max travaux*

→ indépendance par rapport aux opérations des processus
(Les interactions correctes sont celles qui maintiennent l'invariant)

Difficulté

Nécessite de connaître l'invariant (OK pour un système fermé)

Comment pouvoir raisonner sur chaque interaction séparément ? (3/3)

Systèmes ouverts

Situation : tous les processus ne sont pas connus à l'avance (au moment de la conception)

→ définition de critères de cohérence :

- proposer 1 interface d'accès aux objets partagés, permettant de
- contrôler (automatiquement) les accès pour garantir une propriété globale sur le résultat de l'exécution, indépendamment de l'ordre d'exécution réel

Exemples

- Equivalence à une exécution en exclusion mutuelle
  → maintien de tout invariant : mémoire transactionnelle
- Equivalence à une exécution entrelacée : cohérence mémoire

Plan

Bilan

+ modèle de programmation naturel
– surcoût d'exécution (synchronisation, implantation du pseudo-parallélisme).
– surcoût de développement : nécessité d'expliciter la synchronisation, vérifier la réentrance des bibliothèques, danger des variables partagées.
– surcoût de mise-au-point : débogage souvent délicat (pas de flot séquentiel à suivre, non déterminisme) ; effet d'interférence entre des activités, interblocage...
+ **parallélisme (répartition ou multiprocesseurs) = moyen actuel privilégié pour augmenter la puissance de calcul**

Parallélisme et performance

Idée naïve sur le parallélisme

« Si je remplace ma machine mono-processeur par une machine à N processeurs, mon programme ira N fois plus vite »

Parallélisme et performance

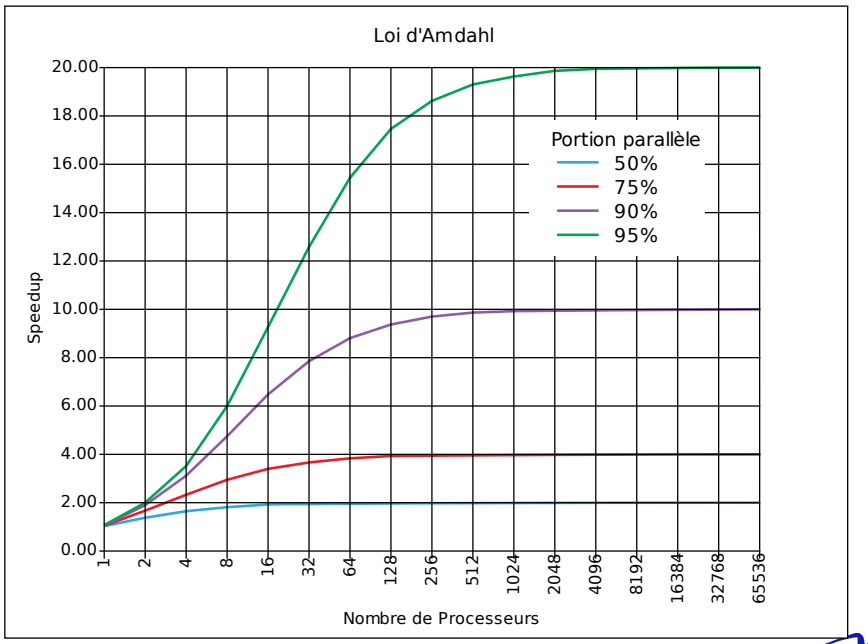
Idée naïve sur le parallélisme

« *Si je remplace ma machine mono-processeur par une machine à N processeurs, mon programme ira N fois plus vite* »

Soit un système composé par une partie p parallélisable + une partie $1-p$ séquentielle.

CPU	durée	$p = 40\%$	$p = 80\%$
1	$p + (1-p)$	100	100
4	$\frac{p}{4} + (1-p)$	70	40
8	$\frac{p}{8} + (1-p)$	65	30
16	$\frac{p}{16} + (1-p)$	62,5	25
∞	$0 + (1-p)$	60	20

Loi d'Amdahl :
facteur d'accélération maximal $= \frac{1}{1-p}$



(source : wikicommons)

Parallélisme et performance

Idée naïve sur la performance

« Si je remplace ma machine par une machine N fois plus rapide, mon programme traitera des problèmes N fois plus grands dans le même temps »

Parallélisme et performance

Idée naïve sur la performance

« Si je remplace ma machine par une machine N fois plus rapide, mon programme traitera des problèmes N fois plus grands dans le même temps »

Pour un problème de taille n soluble en temps T, taille de problème soluble dans le même temps sur une machine N fois plus rapide :

complexité	$N = 4$	$N = 16$	$N = 1024$
$O(n)$	$4n$	$16n$	$1024n$
$O(n^2)$	$\sqrt{4}n = 2n$	$\sqrt{16}n = 4n$	$\sqrt{1024}n = 32n$
$O(n^3)$	$\sqrt[3]{4}n \approx 1.6n$	$\sqrt[3]{16}n \approx 2.5n$	$\sqrt[3]{1024}n \approx 10n$
$O(e^n)$	$\ln(4)n \approx 1.4n$	$\ln(16)n \approx 2.8n$	$\ln(1024)n \approx 6.9n$

En supposant en outre que tout est 100% est parallélisable et qu'il n'y a aucune interférence !

Plan

Evaluation : architecture monoprocesseur

Modèle d'exécution abstrait : entrelacement

L'exécution concurrente (simultanée) d'un ensemble de processus $(P_i)_{i \in I}$ est représentée comme une exécution consistant en un *entrelacement arbitraire* des histoires de chacun des processus P_i

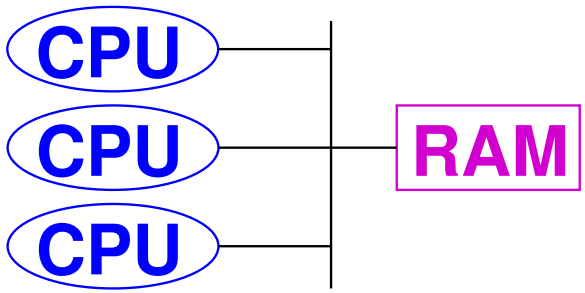
Réalisation sur un monoprocesseur

Pseudo parallélisme (ou parallélisme virtuel)

- le processeur est alloué à tour de rôle à chacun des processus par l'ordonnanceur du système d'exploitation
- le modèle reflète la réalité
- le parallélisme garde tout son intérêt comme
  - outil de conception et d'organisation des traitements,
  - et pour assurer une indépendance par rapport au matériel.

Evaluation : multiprocesseurs SMP (vrai parallélisme)

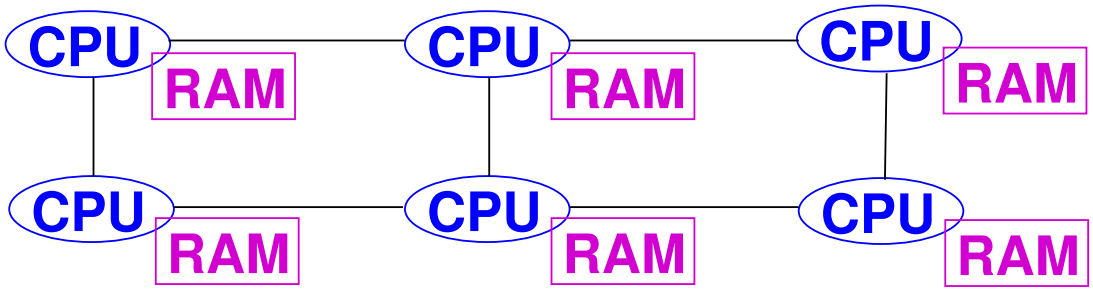
[SMP] Symmetric MultiProcessor :
une mémoire + un ensemble de processeurs



- tant que les processus travaillent sur des zones mémoires distinctes $a; b$ ou $b; a$ ou encore une exécution réellement simultanée de a et b donnent le même résultat
- si a et b opèrent simultanément sur une même zone mémoire, le résultat serait imprévisible, *mais* les requêtes d'accès à la mémoire sont (en général) traitées en séquence par le matériel, pour une taille de bloc donnée.
  Le résultat sera donc le même que celui de $a; b$ ou de $b; a$
- le modèle reflète donc la réalité

Evaluation : multiprocesseurs NUMA (vrai parallélisme)

[NUMA] : Non-Uniform Memory Access
graphe d'interconnexion de {CPU+mémoire}



- chaque nœud/site opère sur sa mémoire locale, et traite en séquence les requêtes d'accès à sa mémoire locale provenant d'autres sites/nœuds
- le modèle reflète donc la réalité

Modèle et réalité : un bémol

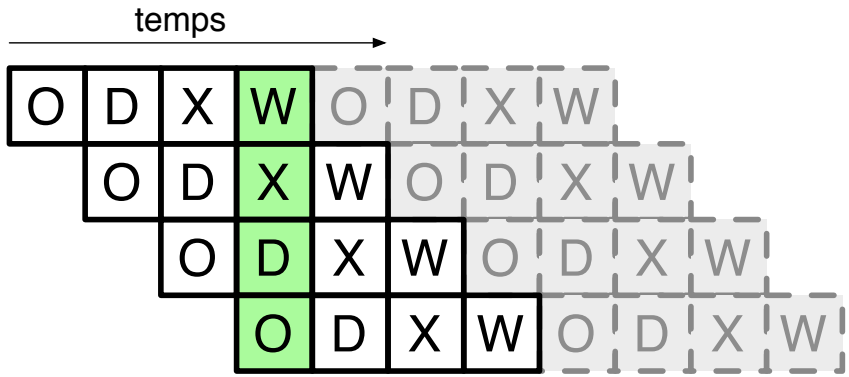
Les architectures récentes éloignent le modèle de la réalité :

- au niveau du processeur : fragmentation et concurrence à grain fin
  - pipeline : plusieurs instructions en cours dans un même cycle : obtention, décodage, exécution, écriture du résultat
  - superscalaire : plusieurs unités d'exécution (et pipeline)
  - instructions vectorielles
  - réordonnancement (out-of-order)
- au niveau de la mémoire : utilisation de caches

Concurrence à grain fin : pipeline

Principe

- chaque instruction comporte une série d'étapes :
  obtention (O)/décodage (D)/exécution (X)/écriture du résultat (W)
- chaque étape est traitée par un circuit à part
- le pipeline permet de charger plusieurs instructions et ainsi d'utiliser simultanément les circuits dédiés, chacun opérant sur une instruction



Difficulté

dépendances entre données utilisées par des instructions proches

```
ADD R1, R1,1     # R1++
SUB R2, R1, 10   # R2 := R1 - 10
```

Remèdes

- insertion de NOP (bulles) pour limiter le traitement parallèle
- réordonnancement (éloignement) des instructions dépendantes

Caches

La mémoire et le processeur sont éloignés : un accès mémoire est considérablement plus lent que l'exécution d'une instruction (peut atteindre un facteur 100 dans un ordinateur, 10000 en réparti).
Principe de localité :

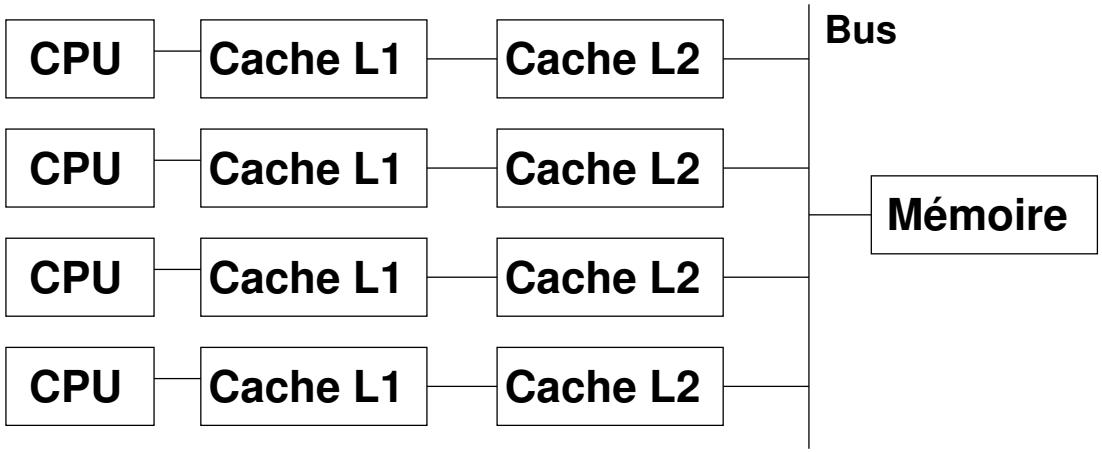
- **temporelle** si on utilise une adresse, on l'utilisera probablement de nouveau dans peu de temps
- **spatiale** si on utilise une adresse, on utilisera probablement une adresse proche dans peu de temps

⇒ conserver près du CPU les dernières cases mémoire accédées
⇒ Cache : mémoire rapide proche du processeur

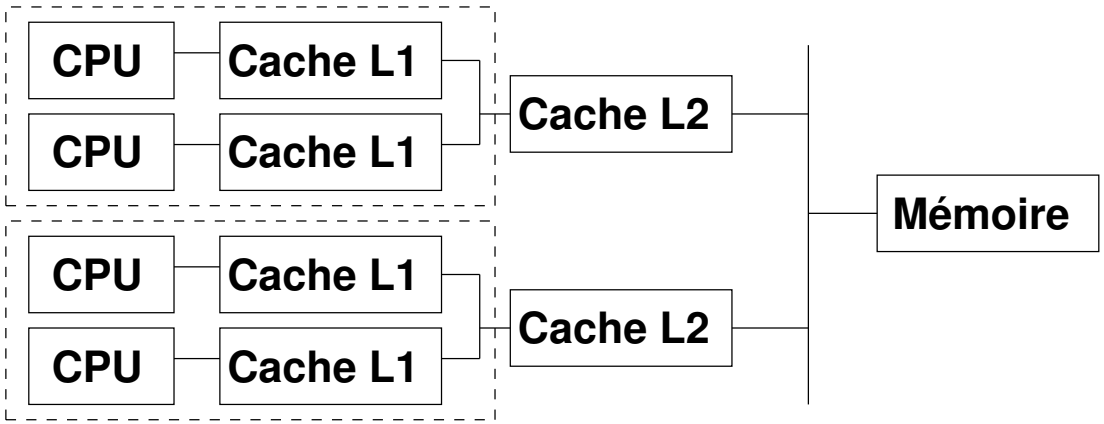
Plusieurs niveaux de caches : de plus en plus gros, de moins en moins rapides (couramment 3 niveaux).

Caches sur les architectures à multi-processeurs

Multi-processeurs « à l'ancienne » :



Multi-processeurs multi-cœurs :



Problème :

cohérence/arbitrage si plusieurs copies en cache d'un même mot mémoire

Comment fonctionne l'écriture d'une case mémoire avec les caches ?

Write-Through diffusion sur le bus à chaque valeur écrite

- \+ visible par les autres processeurs ⇒ invalidation des valeurs passées
- \+ la mémoire et le cache sont cohérents
- − trafic inutile : écritures répétées, écritures de variables privées au thread

Write-Back diffusion uniquement à l'éviction de la ligne

- \+ trafic minimal
- − cohérence cache - mémoire - autres caches

Cohérence mémoire

Si un processeur écrit la case d'adresse a_1, quand les autres processeurs verront-ils cette valeur ? Si plusieurs écritures consécutives en a_1, $a_2 \ldots$, sont-elles vues dans cet ordre ?

Règles de cohérence mémoire

Cohérence séquentielle le résultat d'une exécution parallèle est le même que celui d'une exécution séquentielle qui respecte l'ordre partiel de chacun des processeurs.

Cohérence PRAM (pipelined RAM ou fifo) les écritures d'un même processeur sont vues dans l'ordre où elles ont été effectuées ; des écritures de processeurs différents peuvent être vues dans des ordres différents.

Cohérence « lente » (slow consistency) : une lecture retourne *une* valeur précédemment écrite, sans remonter dans le temps.

Cohérence Mémoire – exemple

Init : $\texttt{x} = 0 \wedge \texttt{y} = 0$

Processeur P1	Processeur P2
(1) $\texttt{x} \leftarrow 1$	(a) $\texttt{y} \leftarrow 1$
(2) $\texttt{t1} \leftarrow \texttt{y}$	(b) $\texttt{t2} \leftarrow \texttt{x}$

Un résultat $\texttt{t1} = 0 \wedge \texttt{t2} = 0$ est possible en cohérence PRAM et slow, impossible en cohérence séquentielle.

Le mot de la fin

Les mécanismes disponibles sur les architectures actuelles permettent d'accélérer l'exécution de traitements indépendants, mais n'offrent pas de garanties sur la cohérence du résultat de l'exécution d'activités coordonnées/interdépendantes

- contrôler/débrayer ces mécanismes
  - vidage des caches
  - inhibition des caches ($\approx$ variables `volatile` en Java)
  - remplissage des pipeline
  - choix de protocoles de cohérence mémoire
- préciser les hypothèses faites sur le matériel par les différents protocoles de synchronisation
  Exemple : accès séquentiels sur les variables partagées

Systèmes concurrents

Philippe Quéinnec

ENSEEIHT
Département Sciences du Numérique

22 septembre 2020

Deuxième partie

L'exclusion mutuelle

Contenu de cette partie

- Difficultés résultant d'accès concurrents à un objet partagé
- Mise en œuvre de protocoles d'isolation
  - solutions synchrones (i.e. bloquantes) : attente active
    - → difficulté du raisonnement en algorithmique concurrente
    - → aides fournies au niveau matériel
  - solutions asynchrones : gestion des processus

Plan

Trop de pain ?

♪♪♪

Vous

1. Arrivez à la maison
2. Constatez qu'il n'y a plus de pain
3. Allez à une boulangerie
4. Achetez du pain
5. Revenez à la maison
6. Rangez le pain

Votre colocataire

1. Arrive à la maison
2. Constate qu'il n'y a plus de pain
3. Va à une boulangerie
4. Achète du pain
5. Revient à la maison
6. Range le pain

Spécification

Propriétés de correction

- Sûreté : un seul pain est acheté
- Vivacité : s'il n'y a pas de pain, quelqu'un en achète

Que se passe-t-il si

- votre colocataire était arrivé après que vous soyez revenu de la boulangerie ?
- Vous étiez arrivé après que votre colocataire soit revenu de la boulangerie ?
- Votre colocataire attend que vous soyez là pour vérifier s'il y a du pain ?

$\Rightarrow$ *race condition* quand la correction dépend de l'ordonnancement des actions

Solution 1 ?

Vous (processus A)

```
A1. si (pas de pain
        && pas de note) alors
A2.   laisser une note
A3.   aller acheter du pain
A4.   enlever la note
   finsi
```

Colocataire (processus B)

```
B1. si (pas de pain)
        && pas de note) alors
B2.   laisser une note
B3.   aller acheter du pain
B4.   enlever la note
    finsi
```

$\Rightarrow$ deux pains possibles si entrelacement A1.B1.A2.B2...

Solution 2 ?

♪♪♪

Vous (processus A)

```
laisser une note A
si (pas de note B) alors
   si pas de pain alors
      aller acheter du pain
   finsi
finsi
enlever la note A
```

Colocataire (processus B)

```
laisser une note B
si (pas de note A) alors
   si pas de pain alors
      aller acheter du pain
   finsi
finsi
enlever la note B
```

$\Rightarrow$ zéro pain possible

Solution 3 ?

♪♪♪

Vous (processus A)

```
laisser une note A
tant que note B faire
   rien
fintq
si pas de pain alors
     aller acheter du pain
finsi
enlever la note A
```

Colocataire (processus B)

```
laisser une note B
si (pas de note A) alors
   si pas de pain alors
      aller acheter du pain
   finsi
finsi
enlever la note B
```

Pas satisfaisant

Hypothèse de progression / Solution peu évidente / Asymétrique / Attente active

Interférence et isolation

```
(1) x := lire_compte(2);          ||  (a) v := lire_compte(1);
(2) y := lire_compte(1);          ||  (b) v := v - 100;
(3) y := y + x;                   ||  (c) ecrire_compte(1, v);
(4) ecrire_compte(1, y);          ||
```

- Le compte 1 est partagé par les deux traitements ;
- les variables `x`, `y` et `v` sont locales à chacun des traitements ;
- les traitements s'exécutent en parallèle, et leurs actions peuvent être entrelacées.

(1) (2) (3) (4) (a) (b) (c) est une exécution possible, cohérente.
(1) (a) (b) (c) (2) (3) (4) " " " " "
(1) (2) (a) (3) (b) (4) (c) est une exécution possible, incohérente.

Section critique

Définition

Les séquences $S_1 = (1);(2);(3);(4)$ et $S_2 = (a);(b);(c)$ sont des sections critiques, qui doivent chacune être exécutées de manière atomique (indivisible) :

- le résultat de l'exécution concurrente de S_1 et S_2 doit être le même que celui de l'une des exécutions séquentielles $S_1; S_2$ ou $S_2; S_1$.
- cette équivalence peut être atteinte en contrôlant directement l'ordre d'exécution de S_1 et S_2 (exclusion mutuelle), ou en contrôlant les effets de S_1 et S_2 (contrôle de concurrence).

« Y a-t-il du pain ? Si non alors acheter du pain ; ranger le pain. »

Accès concurrents

Exécution concurrente ♪♪♪

```
init x = 0;  // partagé
⟨ a := x; x := a + 1 ⟩ ‖ ⟨ b := x; x := b - 1 ⟩
```
⇒ x = -1, 0 ou 1

Modification concurrente ♪♪♪

⟨ x := 0x|00|01| ⟩ ‖ ⟨ x := 0x|02|00| ⟩

⇒ x = 0x0001 ou 0x0200 ou 0x0201 ou 0x0000 ou 1234 !

Cohérence mémoire ♪♪♪

```
init x = 0 ∧ y = 0
⟨ x := 1; y := 2 ⟩ ‖ ⟨ printf("%d %d",y,x); ⟩
```
⇒ affiche 0 0 ou 2 1 ou 0 1 ou 2 0 !

L'exclusion mutuelle

Exécution en exclusion mutuelle d'un ensemble de sections critiques

- ensemble d'activités concurrentes A_i
- variables partagées par toutes les activités
  variables privées (locales) à chaque activité
- structure des activités

```
cycle
      entrée   section critique   sortie
      ⋮
fincycle
```

- hypothèses :
  - vitesse d'exécution non nulle
  - section critique de durée finie

Propriétés du protocole d'accès ♪♪♪

- (sûreté) à tout moment, au plus une activité est en cours d'exécution d'une section critique

$$\text{invariant } \forall i, j \in 0..N-1 : A_i.excl \wedge A_j.excl \Rightarrow i = j$$

- (progression ou vivacité globale) lorsqu'il y a (au moins) une demande, une activité qui demande à entrer sera admise

$$(\exists i \in 0..N-1 : A_i.dem) \rightsquigarrow (\exists j \in 0..N-1 : A_j.excl)$$
$$\forall i \in 0..N-1 : A_i.dem \rightsquigarrow (\exists j \in 0..N-1 : A_j.excl)$$

- (vivacité individuelle) si une activité demande à entrer, elle finira par obtenir l'accès (son attente est finie)

$$\forall i \in 0..N-1 : A_i.dem \rightsquigarrow A_i.excl$$

($p \rightsquigarrow q$: à tout moment, si p est vrai, alors q sera vrai ultérieurement)

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
```

Conception des systèmes concurrents

2SN

ENSEEIHT
Département Sciences du Numérique

27 septembre 2020

Troisième partie

Sémaphores

Contenu de cette partie

- présentation d'un objet de synchronisation « minimal » (sémaphore)
- patrons de conception élémentaires utilisant les sémaphores
- exemple récapitulatif (schéma producteurs/consommateurs)
- schémas d'utilisation pour le contrôle fin de l'accès aux ressources partagées
- mise en œuvre des sémaphores

Plan

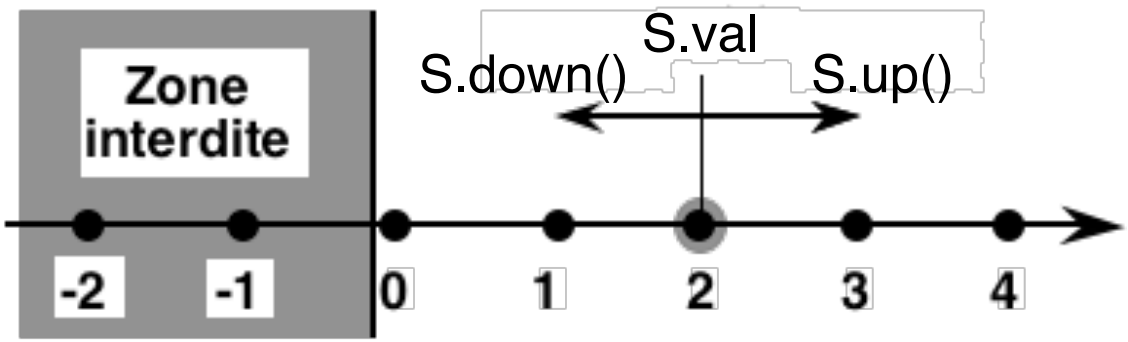
But

- Fournir un moyen *simple*, élémentaire, de contrôler les effets des interactions entre processus
  - isoler (modularité) : atomicité
  - spécifier des interactions précises : synchronisation
- Exprimer ce contrôle par des interactions sur un *objet partagé* (indépendant des processus en concurrence) plutôt que par des interactions entre processus (dont le code et le comportement seraient alors interdépendants)

Définition – Dijkstra 1968

Un sémaphore S est un objet dont

- l'état val est un attribut entier *privé* (l'état est encapsulé)
- l'ensemble des états permis est contraint par un invariant (*contrainte de synchronisation*) :
  invariant $S.val \geq 0$ (l'état doit toujours rester positif ou nul)
- l'interface fournit deux opérations principales :
  - *down* : bloque si l'état est nul, décrémente l'état s'il est > 0
  - *up* : incrémente l'état
    $\rightarrow$ permet de débloquer un éventuel processus bloqué sur `down`
  - les opérations `down` et `up` sont *atomiques*



- *Autre opération* : constructeur (et/ou initialisation)
  $S = new\ Semaphore(v_0)$ (ou $S.init(v_0)$)
  (crée et) initialise l'état de S à v_0
- *Autres noms des opérations*

`P`	Probeer (essayer [de passer])	*down*	wait/attendre	acquire/prendre
`V`	Verhoog (augmenter)	*up*	signal(er)	release/libérer

Modèle intuitif

Un sémaphore peut être vu comme un tas de jetons avec 2 actions

- Prendre un jeton, en attendant si nécessaire qu'il y en ait ;
- Déposer un jeton.

Attention

- les jetons sont anonymes et illimités : un processus peut déposer un jeton sans en avoir pris ;
- il n'y a pas de lien entre le jeton déposé et le processus déposeur ;
- lorsqu'un processus dépose un jeton et que des processus sont en attente, *un seul* d'entre eux peut prendre ce jeton.

Définition formelle : Hoare

Définition

Un sémaphore S encapsule un entier *val* tel que

$$\begin{array}{ccc} \text{init} & \Rightarrow & S.val \geq 0 \\ \{S.val = k \wedge k > 0\} & S.down() & \{S.val = k - 1\} \\ \{S.val = k\} & S.up() & \{S.val = k + 1\} \end{array}$$

Remarques

- Si la précondition de $S.down()$ est fausse, le processus attend.
- Si l'exécution de l'opération *up*, rend vraie la précondition de $S.down()$ et qu'il y a au moins une activité bloquée sur *down*, une telle activité est débloquée (et décrémente le compteur).
- l'invariant du sémaphore peut aussi s'exprimer à partir des nombres $\#down$ et $\#up$ d'opérations *down* et *up* effectuées : **invariant** $S.val = S.val_{\text{init}} + \#up - \#down$

Remarques

1. Lors de l'exécution d'une opération *up*, s'il existe plusieurs processus en attente, la politique de choix du processus à débloquer peut être :
   - par ordre chronologique d'arrivée (FIFO) : équitable
   - associée à une priorité affectée aux processus en attente
   - indéfinie.
     C'est le cas le plus courant : avec une primitive rapide mais non équitable, on peut implanter (laborieusement) une solution équitable, mais avec une primitive lente et équitable, on ne peut pas implanter une solution rapide.
2. Variante : *down* non bloquant (*tryDown*)

$$\left\{ S.val = k \right\} r \leftarrow S.tryDown() \left\{ \begin{array}{c} (k > 0 \wedge S.val = k - 1 \wedge r) \\ \vee (k = 0 \wedge S.val = k \wedge \neg r) \end{array} \right\}$$

Attention aux mauvais usages : incite à l'attente active.

Sémaphore binaire (booléen) – Verrou

Définition

Sémaphore S encapsulant un entier b tel que

$$\begin{array}{ccc} \{S.b = 1\} & S.down() & \{S.b = 0\} \\ \{true\} & S.up() & \{S.b = 1\} \end{array}$$

- Un sémaphore binaire est différent d'un sémaphore entier initialisé à 1.
- Souvent nommé verrou/lock
- Opérations down/up = lock/unlock ou acquire/release

Plan

Schémas d'utilisation essentiels (0/4)

Réalisation de l'isolation : exclusion mutuelle

Algorithme

```
global mutex = new Semaphore(--); //objet partagé

// Protocole d'exclusion mutuelle
// (suivi par chacun des processus)

    section critique
```

Schémas d'utilisation essentiels (0/4)

Réalisation de l'isolation : exclusion mutuelle

Algorithme

```
global mutex = new Semaphore(1); //objet partagé

// Protocole d'exclusion mutuelle
// (suivi par chacun des processus)

mutex.down()
     section critique
mutex.up()
```

Schémas d'utilisation essentiels (1/4)

Généralisation : contrôle du degré de parallélisme

Algorithme

Pour limiter à *Max* le nombre d'accès simultanés à la ressource *R* :

- Objet partagé :

```
global accèsR = new Semaphore(Max)
```

- Protocole d'accès à la ressource *R* (pour *chaque* processus) :

```
accèsR.down()
    accès à la ressource R
accèsR.up()
```

Règle de conception

- Identifier les portions de code où le parallélisme doit être limité
- Définir un sémaphore pour contrôler le degré de parallélisme
- Encadrer ces portions de code par down/up sur ce sémaphore

Schémas d'utilisation essentiels (2/4)

Synchronisation élémentaire : attendre/signaler un événement E

- Objet partagé :

```
occurrenceE = new Semaphore(0) // initialisé à 0
```

- attendre une occurrence de E : `occurrenceE.down()`
- signaler l'occurence de l'événement E : `occurrenceE.up()`

Règle de conception

- Identifier les événements qui doivent être attendus avant chaque action
- Définir un sémaphore *semE* par événement E à attendre
  - appel à *semE.down()* avant l'action oùl'attente est nécessaire
  - appel à *semE.up()* après l'action provoquant l'occurrence de l'événement

Schémas d'utilisation essentiels (3/4)

Synchronisation élémentaire : rendez-vous entre 2 processus A et B

Problème : garantir l'exécution ≪ virtuellement ≫ simultanée d'un point donné du flot de contrôle de A et d'un point donné du flot de contrôle de B

- Objets partagés :

```
aArrivé = new Semaphore(0);
bArrivé = new Semaphore(0)  // initialisés à 0
```

- Protocole de rendez-vous :

```
Processus A          Processus B
   ...                  ...
aArrivé.up()         bArrivé.up()
bArrivé.down()       aArrivé.down()
   ...                  ...
```

Schémas d'utilisation essentiels (4/4)

Généralisation : rendez-vous à N processus (≪ barrière ≫)

Fonctionnement : pour passer la barrière, un processus doit attendre que les $N - 1$ autres processus l'aient atteint.

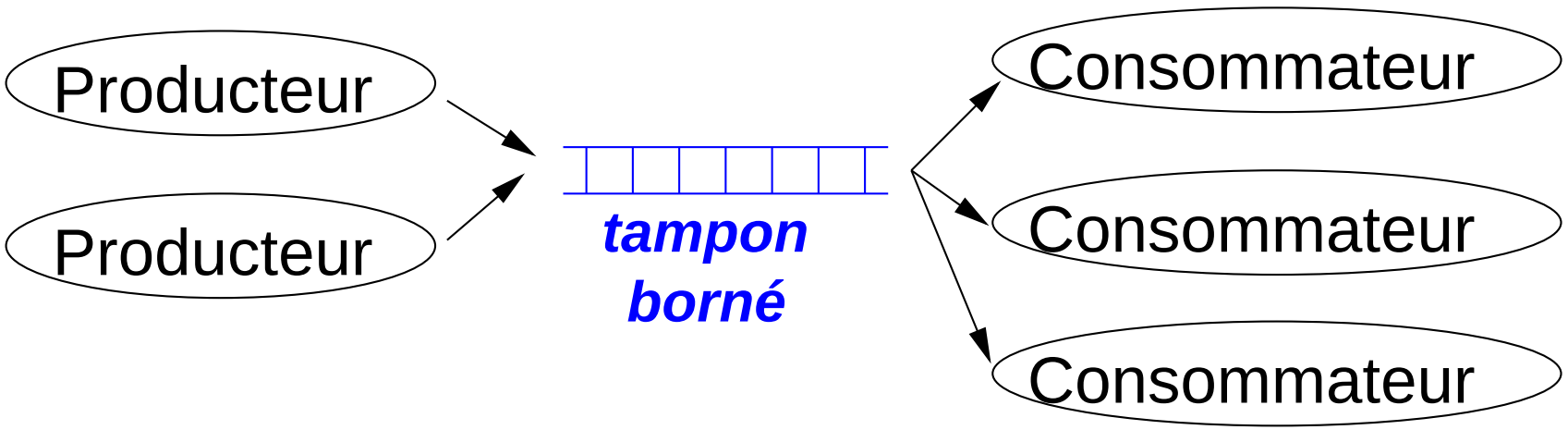
- Objet partagé :

```
barrière = tableau [0..N-1] de Semaphore;
pour i  := 0 à N-1 faire barrière[i].init(0) finpour;
```

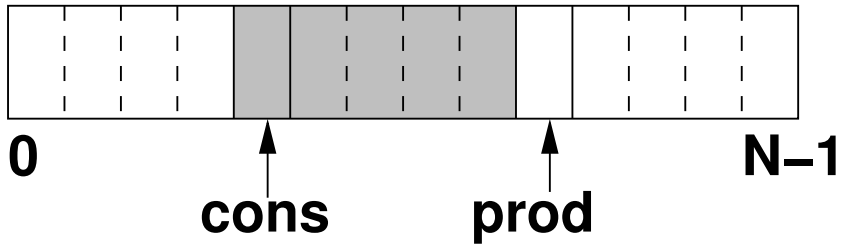
- Protocole de passage de la barrière (pour le processus i) :

```
pour k := 0 à N-1 faire
    barrière[i].up()
finpour;
pour k := 0 à N-1 faire
    barrière[k].down()
finpour;
```

Schéma producteurs/consommateurs : tampon borné



- tampon de taille borné et fixé
- nombre indéterminé et dynamique de producteurs
- " " " " de consommateurs

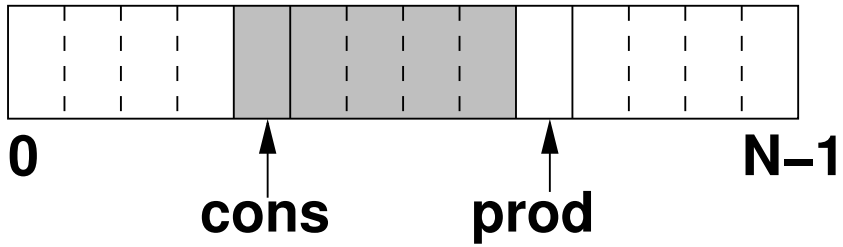


producteur

```
produire(i) {i : Item}

    { dépôt dans le tampon }
    tampon[prod] := i
    prod := prod + 1 mod N
```

consommateur

```
    { retrait du tampon }
    i := tampon[cons]
    cons := cons + 1 mod N

consommer(i) {i : Item}
```



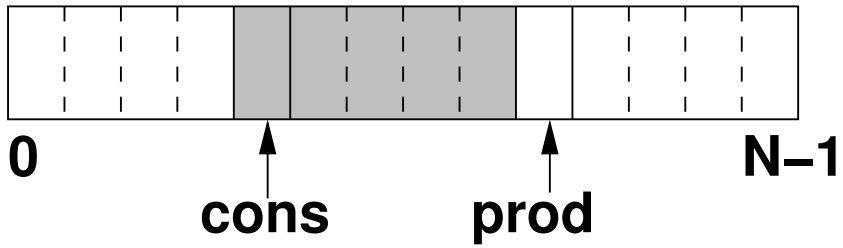
producteur

```
produire(i) {i : Item}

mutex.down()
      { dépôt dans le tampon }
      tampon[prod] := i
      prod := prod + 1 mod N
mutex.up()
```

consommateur

```
mutex.down()
      { retrait du tampon }
      i := tampon[cons]
      cons := cons + 1 mod N
mutex.up()

consommer(i) {i : Item}
```

Sémaphores : mutex := 1



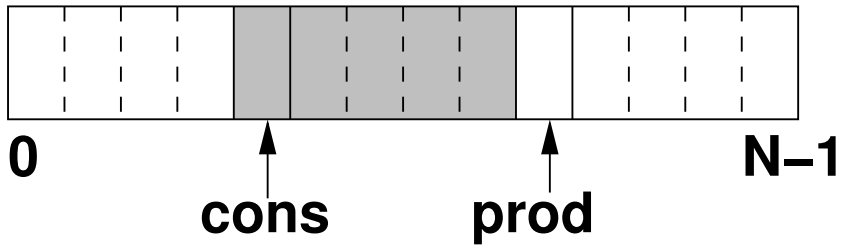
producteur

```
produire(i) {i : Item}

mutex.down()
      { dépôt dans le tampon }
      tampon[prod] := i
      prod := prod + 1 mod N
mutex.up()
```

consommateur

```
occupé.down()
{ ∃ places occupées }
mutex.down()
      { retrait du tampon }
      i := tampon[cons]
      cons := cons + 1 mod N
mutex.up()

consommer(i) {i : Item}
```

Sémaphores : mutex := 1, occupé := 0



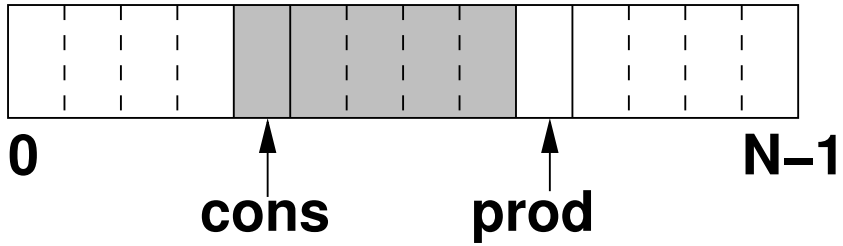
producteur

```
produire(i) {i : Item}
libre.down()
{ ∃ places libres }
mutex.down()
     { dépôt dans le tampon }
     tampon[prod] := i
     prod := prod + 1 mod N
mutex.up()
```

consommateur

```
occupé.down()
{ ∃ places occupées }
mutex.down()
     { retrait du tampon }
     i := tampon[cons]
     cons := cons + 1 mod N
mutex.up()

consommer(i) {i : Item}
```

Sémaphores : mutex := 1, occupé := 0, libre := 0



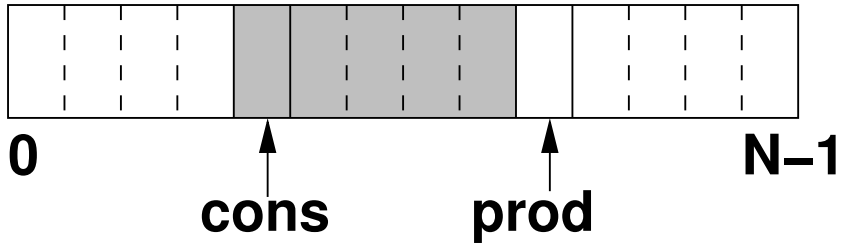
producteur

```
produire(i) {i : Item}
libre.down()
{ ∃ places libres }
mutex.down()
      { dépôt dans le tampon }
      tampon[prod] := i
      prod := prod + 1 mod N
mutex.up()
```

consommateur

```
occupé.down()
{ ∃ places occupées }
mutex.down()
      { retrait du tampon }
      i := tampon[cons]
      cons := cons + 1 mod N
mutex.up()
{ ∃ places libres }
libre.up()
consommer(i) {i : Item}
```

Sémaphores : mutex := 1, occupé := 0, libre := 0



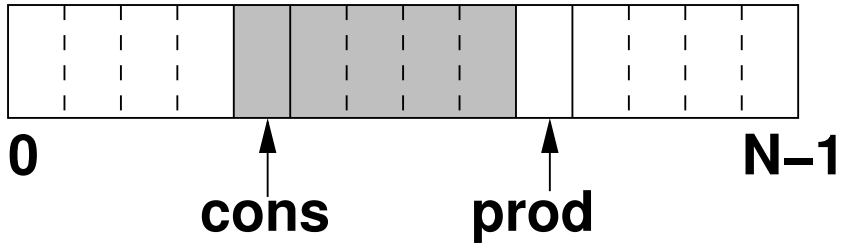
producteur

```
produire(i) {i : Item}
libre.down()
{ ∃ places libres }
mutex.down()
      { dépôt dans le tampon }
      tampon[prod] := i
      prod := prod + 1 mod N
mutex.up()
{ ∃ places occupées }
occupé.up()
```

consommateur

```
occupé.down()
{ ∃ places occupées }
mutex.down()
      { retrait du tampon }
      i := tampon[cons]
      cons := cons + 1 mod N
mutex.up()
{ ∃ places libres }
libre.up()
consommer(i) {i : Item}
```

Sémaphores : mutex := 1, occupé := 0, libre := 0



producteur

```
produire(i) {i : Item}
libre.down()
{ ∃ places libres }
mutex.down()
      { dépôt dans le tampon }
      tampon[prod] := i
      prod := prod + 1 mod N
mutex.up()
{ ∃ places occupées }
occupé.up()
```

consommateur

```
occupé.down()
{ ∃ places occupées }
mutex.down()
      { retrait du tampon }
      i := tampon[cons]
      cons := cons + 1 mod N
mutex.up()
{ ∃ places libres }
libre.up()
consommer(i) {i : Item}
```

Sémaphores : mutex := 1, occupé := 0, libre := N

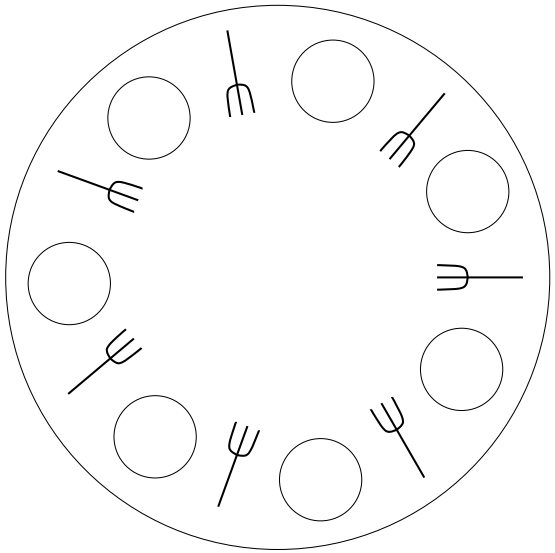
Contrôle fin du partage (1/3) : pool de ressources

- N ressources critiques, équivalentes, réutilisables
- usage exclusif des ressources
- opération `allouer` $k \leq N$ ressources
- opération `libérer` des ressources précédemment obtenues
- bon comportement :
  - pas deux demandes d'allocation consécutives sans libération intermédiaire
  - un processus ne libère pas plus que ce qu'il détient

Mise en œuvre de politiques d'allocation : FIFO, priorités...

Contrôle fin du partage (2/3) : philosophes et spaghettis

N philosophes sont autour d'une table.
Il y a une assiette par philosophe, et
une fourchette entre chaque assiette.
Pour manger, un philosophe doit
utiliser les deux fourchettes adjacentes
à son assiette (et celles-là seulement).



Un philosophe peut être :

- penseur : il n'utilise pas de fourchettes ;
- mangeur : il utilise les deux fourchettes adjacentes ; aucun de ses voisins ne peut manger ;
- demandeur : il souhaite manger mais ne dispose pas des deux fourchettes.

Allocation multiple de ressources différenciées, interblocage. . .

Contrôle fin du partage (3/3) : lecteurs/rédacteurs

Une ressource peut être utilisée :

- concurremment par plusieurs lecteurs (plusieurs lecteurs simultanément) ;
- exclusivement par un rédacteur (pas d'autre rédacteur, pas d'autre lecteur).

Souvent rencontré sous la forme de verrou lecture/écriture (read-write lock).
Permet l'isolation des modifications avec un meilleur parallélisme que l'exclusion mutuelle.

Stratégies d'allocation pour des classes distinctes de clients . . .

Plan

Implantation d'un sémaphore

Repose sur un service de gestion des processus fournissant :

- l'exclusion mutuelle (cf partie II)
- le blocage (suspension) et déblocage (reprise) des processus

Implantation

Sémaphore = ⟨ int nbjetons ;
File<Processus> bloqués ⟩

Algorithme

```
S.down() =   entrer en excl. mutuelle
               si S.nbjetons = 0 alors
                  insérer self dans S.bloqués
                  suspendre le processus courant
               sinon
                  S.nbjetons ← S.nbjetons - 1
               finsi
             sortir d'excl. mutuelle
```

```
S.up() =   entrer en excl. mutuelle
             si S.bloqués ≠ vide alors
                procRéveillé ← extraire de S.bloqués
                débloquer procRéveillé
             sinon
                S.nbjetons ← S.nbjetons + 1
             finsi
           sortir d'excl. mutuelle
```

Compléments (1/3) :

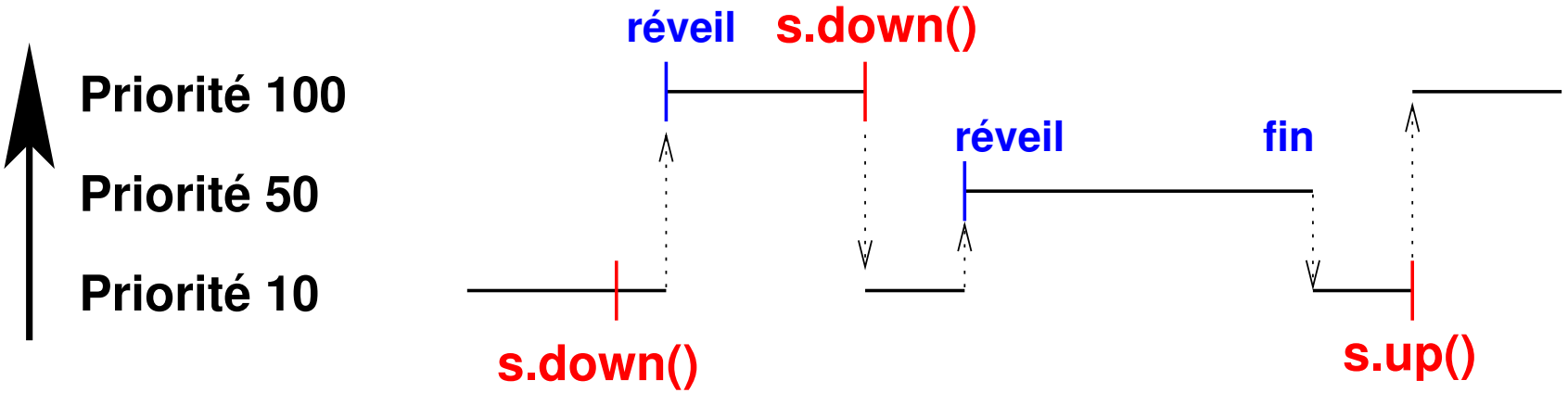
réalisation d'un sémaphore général à partir de sémaphores binaires

```
Sg = ⟨val := ?,
      mutex = new SemaphoreBinaire(1),
      accès = new SemaphoreBinaire(val>0;1;0) // verrous
      ⟩
Sg.down() =  Sg.accès.down()
             Sg.mutex.down()
             S.val ← S.val - 1
             si S.val ≥ 1 alors Sg.accès.up()
             Sg.mutex.up()
Sg.up() =  Sg.mutex.down()
           S.val ← S.val + 1
           si S.val = 1 alors Sg.accès.up()
           Sg.mutex.up()
```

→ les sémaphores binaires ont (au moins) la même puissance d'expression que les sémaphores généraux

Compléments (2/3) : sémaphores et priorités

Temps-réel ⇒ priorité ⇒ sémaphore non-FIFO.
Inversion de priorités : un processus moins prioritaire bloque/retarde indirectement un processus plus prioritaire.



Compléments (3/3) : solution à l'inversion de priorité

- Plafonnement de priorité (priority ceiling) : monter systématiquement la priorité d'un processus verrouilleur à la priorité maximale des processus potentiellement utilisateurs de cette ressource.
  - – Nécessite de connaître a priori les demandeurs
  - – Augmente la priorité même en l'absence de conflit
  - \+ Simple et facile à implanter
  - \+ Prédictible : la priorité est associée à la ressource
- Héritage de priorité : monter dynamiquement la priorité d'un processus verrouilleur à celle du demandeur.
  - \+ Limite les cas d'augmentation de priorité aux cas de conflit
  - – Nécessite de connaître les possesseurs d'un sémaphore
  - – Dynamique $\Rightarrow$ comportement moins prédictible

Conclusion

Les sémaphores

+ ont une sémantique, un fonctionnement simples à comprendre
+ peuvent être mis en œuvre de manière efficace
+ sont suffisants pour réaliser les schémas de synchronisation nécessaires à la coordination des applications concurrentes
– mais sont un outil de synchronisation élémentaire, aboutissant à des solutions difficiles à concevoir et à vérifier
  → schémas génériques

Systèmes concurrents

2SN

ENSEEIHT
Département Sciences du Numérique

7 octobre 2020

Quatrième partie

Interblocage

Contenu de cette partie

- définition et caractérisation des situations d'interblocage
- protocoles de traitement de l'interblocage
  - préventifs
  - curatifs
- apport déterminant d'une bonne modélisation/formalisation pour la recherche et la validation de solutions

Plan

Allocation de ressources multiples

But : gérer la compétition entre activités

- N processus, 1 ressource → protocole d'exclusion mutuelle
- N processus, M ressources → ? ? ? ?

Modèle/protocole ≪ général ≫

- Ressources banalisées, réutilisables, identifiées
- Ressources allouées par un gérant de ressources
- Interface du gérant :
  - **demander** (NbRessources) : {IdRessource}
  - **libérer** ({IdRessource})
- Le gérant :
  - rend les ressources libérées utilisables par d'autres processus
  - libère les ressources détenues, à la terminaison d'un processus.

Garanties sur les réponses aux demandes d'allocation par le gérant

- Vivacité faible (*progression*) :
  si *des* processus déposent des requêtes continûment,
  l'*une* d'entre elles finira par être satisfaite ;
- Vivacité forte (*équité faible*) :
  si un processus dépose sa requête de manière continue,
  elle finira par être satisfaite ;

Négation de la vivacité forte : famine (privation)

Un processus est en famine lorsqu'il attend infiniment longtemps la satisfaction de sa requête (elle n'est jamais satisfaite).

Garanties sur les réponses aux demandes d'allocation par le gérant

- Vivacité faible (*progression*) :
  si *des* processus déposent des requêtes continûment,
  l'*une* d'entre elles finira par être satisfaite ;
- Vivacité forte (*équité faible*) :
  si un processus dépose sa requête de manière continue,
  elle finira par être satisfaite ;

Négation de la vivacité forte : famine (privation)

Un processus est en famine lorsqu'il attend infiniment longtemps la satisfaction de sa requête (elle n'est jamais satisfaite).

Plan

Le problème

Contexte : allocation de ressources réutilisables

- non réquisitionnables
- non partageables
- en quantités entières et finies
- dont l'usage est indépendant de l'ordre d'allocation

Problème

P_1 demande A puis B,
P_2 demande B puis A
$\rightarrow$ risque d'interblocage :

1. P_1 demande et obtient A
2. P_2 demande et obtient B
3. P_2 demande $A \rightarrow$ se bloque
4. P_1 demande $B \rightarrow$ se bloque

Interblocage : définition

Un ensemble de processus est en interblocage si et seulement si tout processus de l'ensemble est en attente d'une ressource qui ne peut être libérée que par un autre processus de cet ensemble.

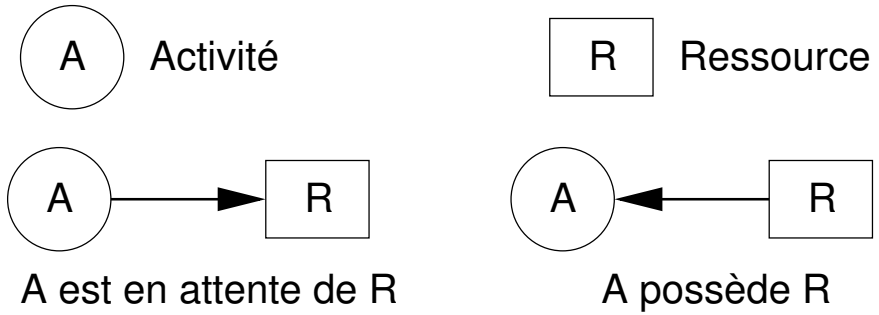
Pour l'ensemble de processus considéré :

Interblocage $\equiv$ négation de la vivacité faible (progression)

$\rightarrow$ absence de famine (viv. forte) $\Rightarrow$ absence d'interblocage (viv. faible)

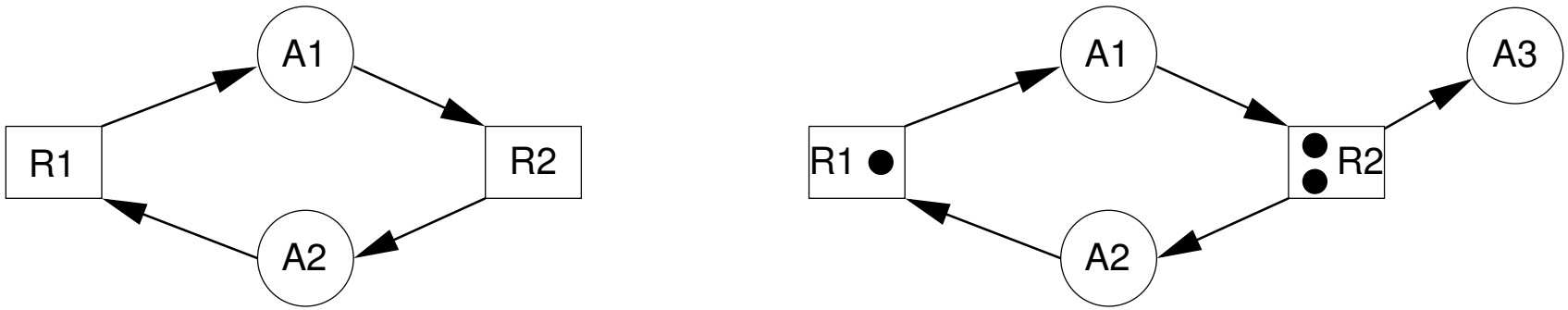
Plan

Notation : graphe d'allocation



Condition nécessaire à l'interblocage

Attente circulaire (cycle dans le graphe d'allocation)



Solutions

Prévention : empêcher la formation de cycles dans le graphe

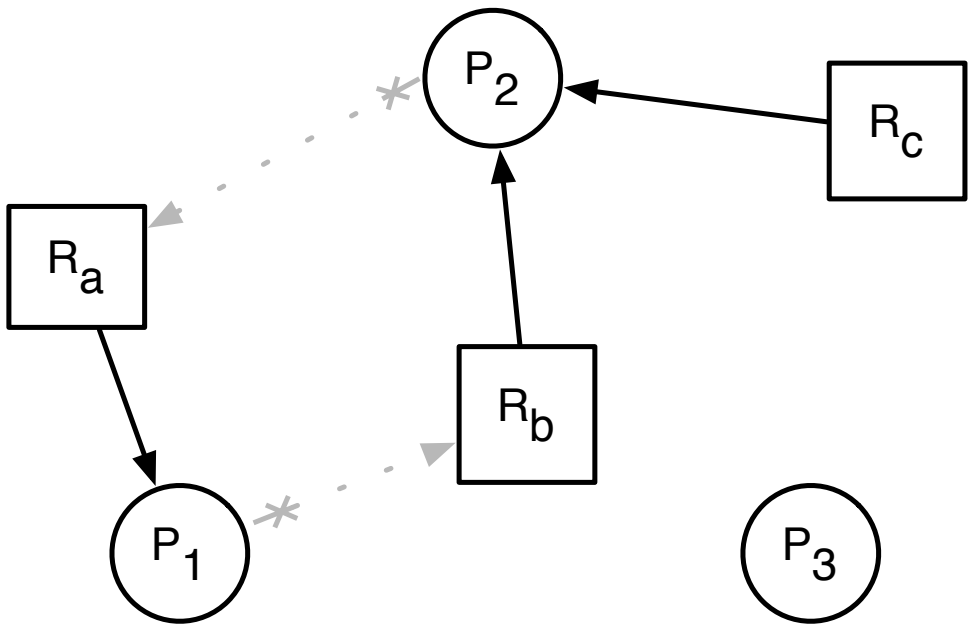
Détection + guérison : détecter l'interblocage, et l'éliminer

Plan

Comment éviter **par construction** la formation de cycles ? (1/4)

Éviter le blocage des processus

→ pas d'attente → pas d'arcs sortant d'un processus



- *Ressources virtuelles* : imprimantes, fichiers
- *Acquisition non bloquante* : le demandeur peut ajuster sa demande si elle ne peut être immédiatement satisfaite

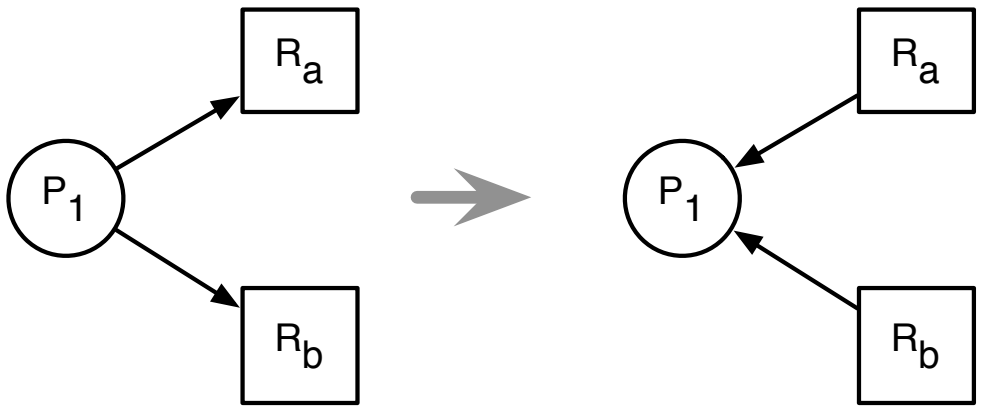
Comment éviter par construction la formation de cycles ? (2/4)

Éviter les demandes fractionnées

Allocation globale : chaque processus demande et obtient en bloc, en une seule fois, toutes les ressources nécessaires

→ une seule demande pour chaque processus

- demande satisfaite → arcs entrants uniquement
- demande non satisfaite → arcs sortants (attente) uniquement



- suppose la connaissance a priori des ressources nécessaires
- sur-allocation et risque de famine

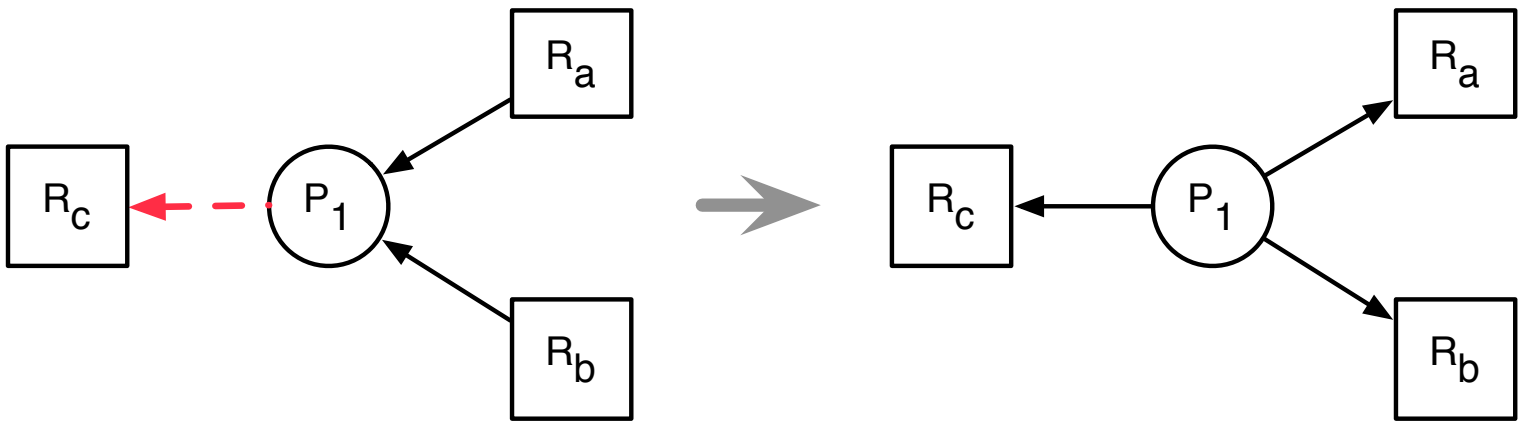
Comment éviter par construction la formation de cycles ? (3/4)

Permettre la réquisition des ressources allouées

→ éliminer/inverser les arcs entrants d'un processus en cas de création d'arcs sortants

Un processus bloqué doit

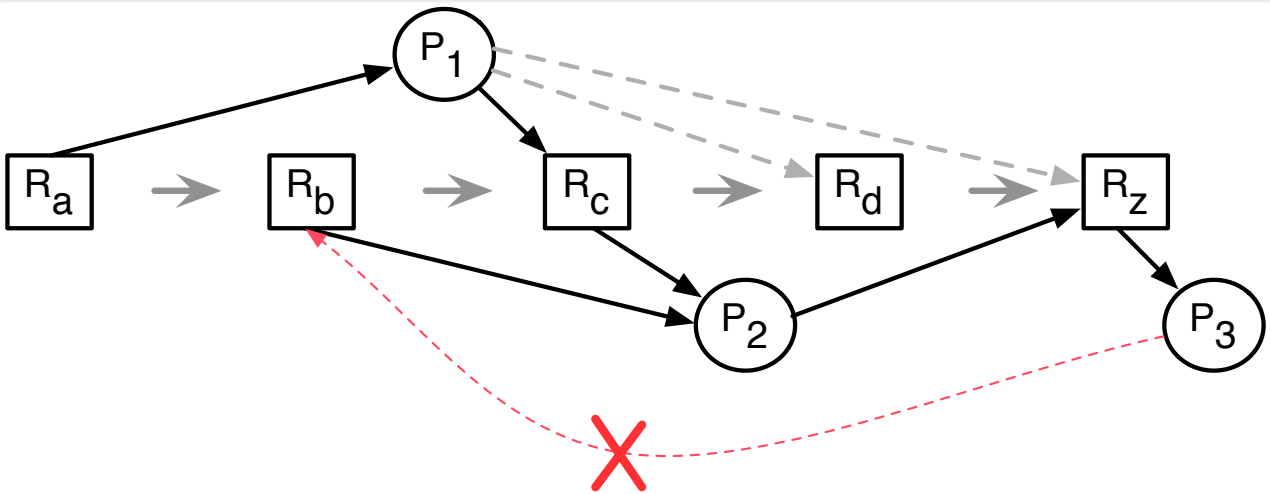
- libérer les ressources qu'il a obtenues
- réobtenir les ressources libérées, avant de pouvoir poursuivre
  → risque de famine



Comment éviter par construction la formation de cycles ? (4/4)

Fixer un ordre global sur les demandes : classes ordonnées

- un ordre est défini sur les ressources
- tout processus doit demander les ressources en suivant cet ordre



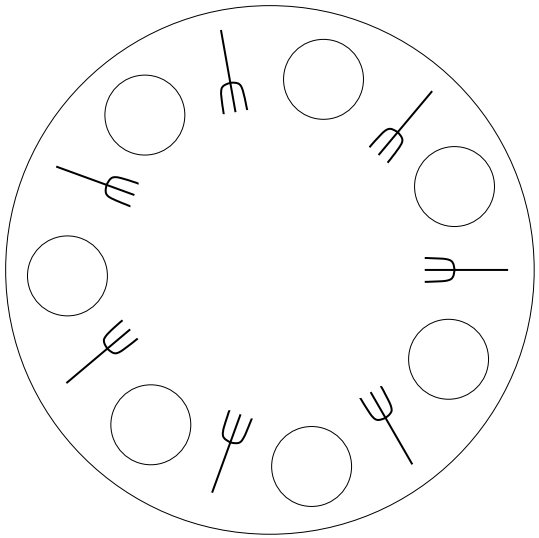
→ pour chaque processus, les chemins du graphe d'allocation vont des ressources inférieures (déjà obtenues) aux supérieures (demandées)

⇒ tout chemin du graphe d'allocation suit l'ordre des ressources

⇒ le graphe d'allocation est sans cycle
(car un cycle est un chemin sur lequel l'ordre des ressources n'est pas respecté)

Exemple : philosophes et interblocage (1/2)

N philosophes sont autour d'une table. Il y a une assiette par philosophe, et une fourchette entre chaque assiette. Pour manger, un philosophe doit utiliser les deux fourchettes adjacentes à son assiette (et celles-là seulement).



Un philosophe peut être :

- penseur : il n'utilise pas de fourchettes ;
- mangeur : il utilise les deux fourchettes adjacentes ; aucun de ses voisins ne peut manger ;
- demandeur : il souhaite manger mais ne dispose pas des deux fourchettes.

Exemple : philosophes et interblocage (2/2)

Risque d'interblocage

Chaque philosophe demande sa fourchette gauche et l'obtient. Puis quand tous ont leur fourchette gauche, chaque philosophe demande sa fourchette droite et se bloque. $\Rightarrow$ interblocage

Solutions

Allocation globale : chaque philosophe demande simultanément les deux fourchettes.

Non conservation : quand un philosophe essaye de prendre sa seconde fourchette et qu'elle est déjà prise, il relâche la première et se met en attente sur la seconde.

Classes ordonnées : imposer un ordre sur les fourchettes $\equiv$ tous les philosophes prennent d'abord la gauche puis la droite, sauf un qui prend d'abord droite puis gauche.

Plan

Esquive

Avant toute allocation, évaluation dynamique du risque (ultérieur) d'interblocage, compte tenu des ressources déjà allouées.

L'algorithme du banquier

- chaque processus annonce le nombre maximum de ressources qu'il est susceptible de demander ;
- l'algorithme maintient le système dans un état fiable, c'est-à-dire tel qu'il existe toujours une possibilité d'éviter l'interblocage dans le pire des scénarios (= celui où chaque processus demande la totalité des ressources annoncées) ;
- lorsque la requête mêne à un état non fiable, elle n'est pas traitée, mais est mise en attente (comme si les ressources n'étaient pas disponibles).

Algorithme du banquier : exemple

12 ressources,
3 processus $P_0/P_1/P_2$ annonçant 10/4/9 comme maximum

	max	poss.	dem	
P_0	10	5		
P_1	4	2	+1	oui
P_2	9	2		

$$(5 + \mathit{4} + 2 \leq 12) \wedge (\mathit{10} + (0) + 2 \leq 12) \wedge ((0) + (0) + \mathit{9} \leq 12)$$

Algorithme du banquier : exemple

12 ressources,
3 processus $P_0/P_1/P_2$ annonçant 10/4/9 comme maximum

	max	poss.	dem	
P_0	10	5		
P_1	4	2		
P_2	9	2	+1	non

$$
\begin{aligned}
&(\mathit{10} + 2 + 3 > 12)\\
\wedge&(5 + 2 + \mathit{9} > 12)\\
\wedge&(5 + \mathit{4} + 3 \leq 12)\\
&\quad\wedge(\mathit{10} + (0) + 3 > 12)\\
&\qquad\wedge(5 + (0) + \mathit{9} > 12))
\end{aligned}
$$

Algorithme du banquier (1/2)

Allocation de Demande ressources au processus IdProc

```
 var Demande, Disponibles : entier = 0,N;
     Annoncées, Allouées : tableau [1..NbProc] de entier;
     fini : booléen = faux;

si Allouées[IdProc]+Demande > Annoncées[IdProc] alors erreur
sinon
   tant que non fini faire
    si Demande > Disponible alors <bloquer le processus>
    sinon
      si étatFiable({1..NbProc}, Disponibles - Demande) alors
        Allouées[IdProc] := Allouées[IdProc] + Demande ;
        Disponibles := Disponibles - Demande;
        fini := vrai;
      sinon <bloquer le processus>;
      finsi
    finsi
  fintq
finsi
```

Algorithme du banquier (2/2)

fonction étatFiable(demandeurs:ensemble de 1..NbProc, dispo : entier): booléen

```
 var d : 1..NbProc;
     vus, S : ensemble de 1..NbProc := ∅, ∅;
     solution : booléen := (demandeurs = ∅);

début
  répéter
   S := {p∈demandeurs-vus / Annoncées[p]-Allouées[p] <= dispo}
   si S ≠ ∅ alors
      choisir d ∈ S;
      vus := vus ∪ {d};
      solution := étatFiable(demandeurs-{d},
                             dispo+Annoncées[d]-Allouées[d]);
   finsi;
  jusqu'à (S = ∅) ou (solution);
  renvoyer solution;
fin étatFiable;
```

Plan

Détection

- construire le graphe d'allocation
- détecter l'existence d'un cycle

Coûteux → exécution périodique (et non à chaque allocation)

Guérison : Réquisition des ressources allouées à un/des processus interbloqués

- fixer des critères de choix du processus victime (priorités...)
- annulation du travail effectué par le(s) processus victime(s)
  - coûteux (détection + choix + travail perdu + restauration),
  - pas toujours acceptable (systèmes interactifs ou embarqués).

- plus de parallélisme dans l'accès aux ressources qu'avec la prévention.
- la guérison peut être un service en soi (tolérance aux pannes...)
  → Mécanismes de reprise : service de sauvegarde périodique d'états intermédiaires *(points de reprise)*

Plan

- Usuellement : interblocage = inconvénient occasionnel
  - → laissé à la charge de l'utilisateur/du programmeur
    - traitement :
      - utilisation de méthodes de prévention simples (classes ordonnées, par exemple)
      - ou détection empirique (délai de garde) et guérison par choix « manuel » des victimes
- Cas particulier : systèmes ouverts, (plus ou moins) contraints par le temps
  - systèmes interactifs, multiprocesseurs, systèmes embarqués
  - recherche de méthodes efficaces, prédictibles, ou automatiques
  - compromis/choix à réaliser entre
    - la prévention qui est plus statique, coûteuse et restreint le parallélisme
    - la guérison, qui est moins prédictible, et coûteuse quand les conflits sont fréquents.
  - émergence d'approches sans blocage (→ prévention), sur les architectures multiprocesseurs (mémoire transactionnelle)

Systèmes concurrents

2SN

ENSEEIHT
Département Sciences du Numérique

8 octobre 2020

Cinquième partie

Moniteurs

Contenu de cette partie

- motivation et présentation
  d'un objet de synchronisation ≪ structuré ≫ (moniteur)
- démarche de conception basée sur l'utilisation de moniteurs
- exemple récapitulatif (schéma producteurs/consommateurs)
- annexe : variantes et mise en œuvre des moniteurs

Plan

Limites des sémaphores

- imbrication aspects de synchronisation/aspects fonctionnels
  $\rightarrow$ manque de modularité, code des processus interdépendant
- pas de contrainte sur le protocole d'utilisation des sémaphores
  $\rightarrow$ démarche de conception artisanale, à partir de schémas élémentaires (attendre/signaler un événement, contrôler l'accès à une ressource...)
- approche ($\rightarrow$ raisonnement) *opératoire*
  $\rightarrow$ vérification difficile

Exemples

- sections critiques entrelacées $\rightarrow$ interblocage
- attente infinie en entrée d'une section critique

Plan

Notion de moniteur Hoare, Brinch-Hansen 1973

Idée de base

La synchronisation résulte du besoin de partager «convenablement» un objet entre plusieurs processus concurrents

→ un moniteur est une construction qui permet de définir et de contrôler le bon usage d'un objet partagé par un ensemble de processus

Définition

Un moniteur = un module exportant des procédures (*opérations*)

- Contrainte :
  exécution des procédures du moniteur en exclusion mutuelle
- La synchronisation des opérations du moniteur est réalisée par des opérateurs internes au moniteur.

Un moniteur est passif : ce sont les processus utilisant le moniteur qui l'activent, en invoquant ses procédures.

Notion de moniteur Hoare, Brinch-Hansen 1973

Idée de base

La synchronisation résulte du besoin de partager «convenablement» un objet entre plusieurs processus concurrents

→ un moniteur est une construction qui permet de définir et de contrôler le bon usage d'un objet partagé par un ensemble de processus

Définition

Un moniteur = un module exportant des procédures (*opérations*)

- Contrainte :
  exécution des procédures du moniteur en exclusion mutuelle
- La synchronisation des opérations du moniteur est réalisée par des opérateurs internes au moniteur.

Un moniteur est passif : ce sont les processus utilisant le moniteur qui l'activent, en invoquant ses procédures.

Expression de la synchronisation : type *condition*

La synchronisation est définie *au sein du moniteur*, en utilisant des variables de type *condition*, internes au moniteur

- Une file d'attente est associée à *chaque* variable condition
- Opérations possibles sur une variable de type condition C :
  - C.attendre() *[C.wait()]* : bloque et range dans la file associée à C le processus appelant, puis libère l'accès exclusif au moniteur.
  - C.signaler() *[C.signal()]* : si des processus sont bloqués sur C, en réveille un ; sinon, nop (opération nulle).

- condition ≈ événement
  - → condition ≠ sémaphore (pas de mémorisation des « signaux »)
  - → condition ≠ prédicat logique
- autres opérations sur les conditions :
  - C.vide() : renvoie vrai si aucun processus n'est bloqué sur C
  - C.attendre(priorité) : réveil des processus bloqués sur C selon une priorité

Expression de la synchronisation : type *condition*

La synchronisation est définie *au sein du moniteur*, en utilisant des variables de type *condition*, internes au moniteur

- Une file d'attente est associée à *chaque* variable condition
- Opérations possibles sur une variable de type condition C :
  - C.attendre() *[C.wait()]* : bloque et range dans la file associée à C le processus appelant, puis libère l'accès exclusif au moniteur.
  - C.signaler() *[C.signal()]* : si des processus sont bloqués sur C, en réveille un ; sinon, nop (opération nulle).

- condition ≈ événement
  - → condition ≠ sémaphore (pas de mémorisation des « signaux »)
  - → condition ≠ prédicat logique

- autres opérations sur les conditions :
  - C.vide() : renvoie vrai si aucun processus n'est bloqué sur C
  - C.attendre(priorité) : réveil des processus bloqués sur C selon une priorité

Exemple : travail délégué (schéma client/serveur asynchrone) : 1 client + 1 serveur

Les activités (processus utilisant le moniteur)

Client

```
boucle
    ⋮
    déposer_travail(t)
    ⋮
    r ←lire_résultat()
    ⋮
fin_boucle
```

Serveur

```
boucle
    ⋮
    x ← prendre_travail()
    // (y ← f(x))
    rendre_résultat(y)
    ⋮
fin_boucle
```

Exemple – le moniteur

Le moniteur

```
variables d'état : req, rés --Requête/Résultat en attente (null si aucun(e))
variables condition : Dépôt, Dispo
```

entrée déposer_travail(*in* t) {(pas d'attente)} req ← t Dépôt.signaler()	*entrée* prendre_travail(*out* t) *si* req = null *alors* Dépôt.attendre() *finsi* t ← req req ← null {RAS}
entrée lire_résultat(*out* r) *si* rés = null *alors* Dispo.attendre() *finsi* r ← rés rés ← null {RAS}	*entrée* rendre_résultat(*in* y) {(pas d'attente)} rés ← y Dispo.signaler()

Transfert du contrôle exclusif

Les opérations du moniteur s'exécutent en exclusion mutuelle.
→ Lors d'un réveil par *signaler()*, qui obtient l'accès exclusif ?

Priorité au signalé

Lors du réveil par *signaler()*,

- l'accès exclusif est transféré au processus réveillé (signalé) ;
- le processus signaleur est mis en attente dans une file globale spécifique, prioritaire sur les processus entrants

Priorité au signaleur

Lors du réveil par *signaler()*,

- l'accès exclusif est conservé par le processus réveilleur ;
- le processus réveillé (signalé) est mis en attente
  - soit dans une file globale spécifique, prioritaire sur les processus entrants,
  - soit avec les processus entrants.

Transfert du contrôle exclusif

Les opérations du moniteur s'exécutent en exclusion mutuelle.
→ Lors d'un réveil par *signaler()*, qui obtient l'accès exclusif ?

Priorité au signalé

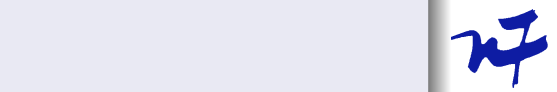
Lors du réveil par *signaler()*,

- l'accès exclusif est transféré au processus réveillé (signalé) ;
- le processus signaleur est mis en attente dans une file globale spécifique, prioritaire sur les processus entrants

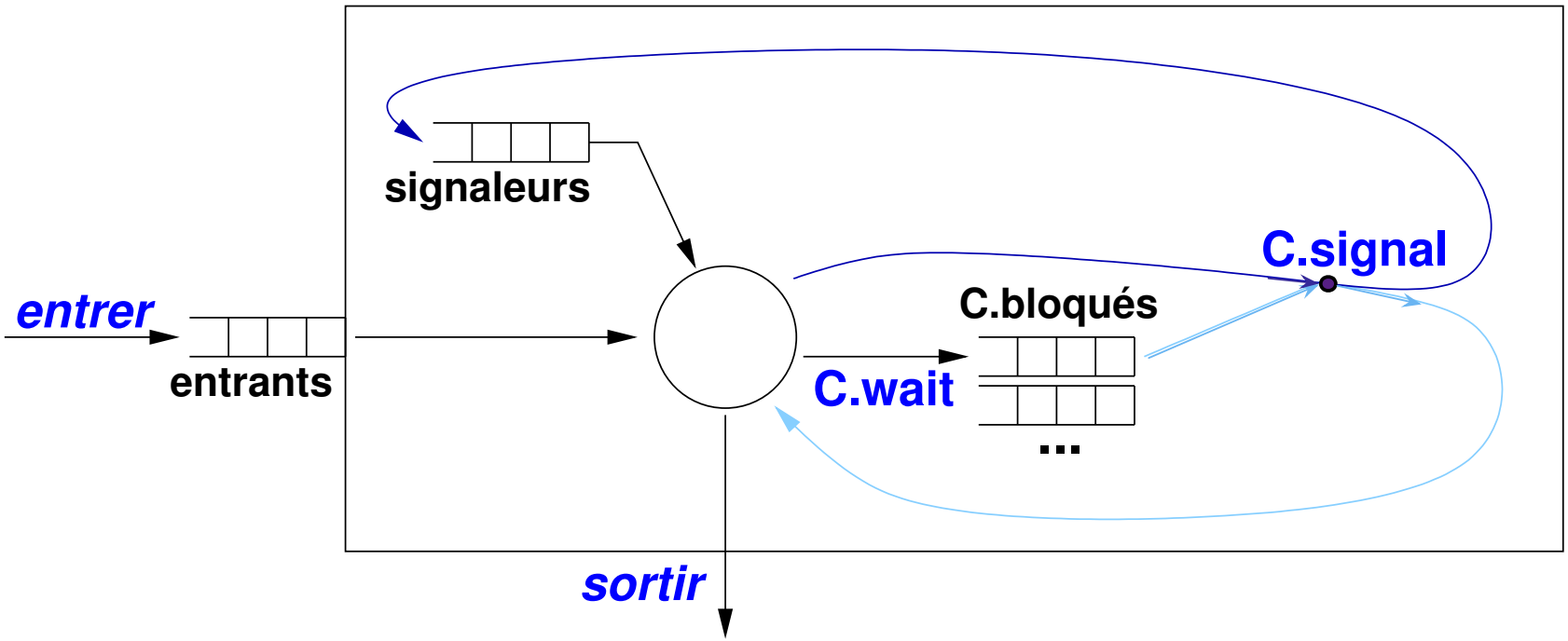
Priorité au signaleur

Lors du réveil par *signaler()*,

- l'accès exclusif est conservé par le processus réveilleur ;
- le processus réveillé (signalé) est mis en attente
  - soit dans une file globale spécifique, prioritaire sur les processus entrants,
  - soit avec les processus entrants.



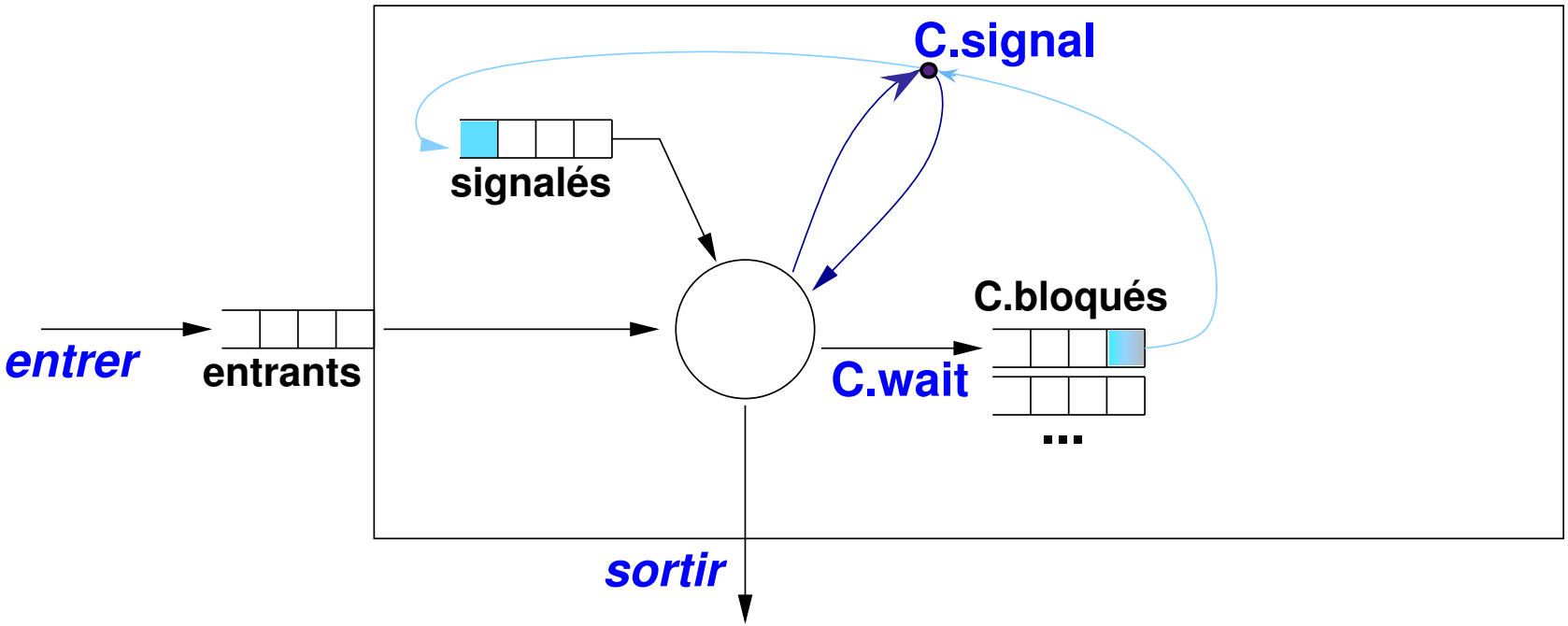
Priorité au signalé



C.signal()

- = opération nulle si pas de bloqués sur C
- sinon,
  - suspend et ajoute le signaleur à la file des signaleurs
  - extrait le processus en tête des bloqués sur C et lui passe le contrôle
- signaleurs prioritaires sur les entrants (progression garantie)

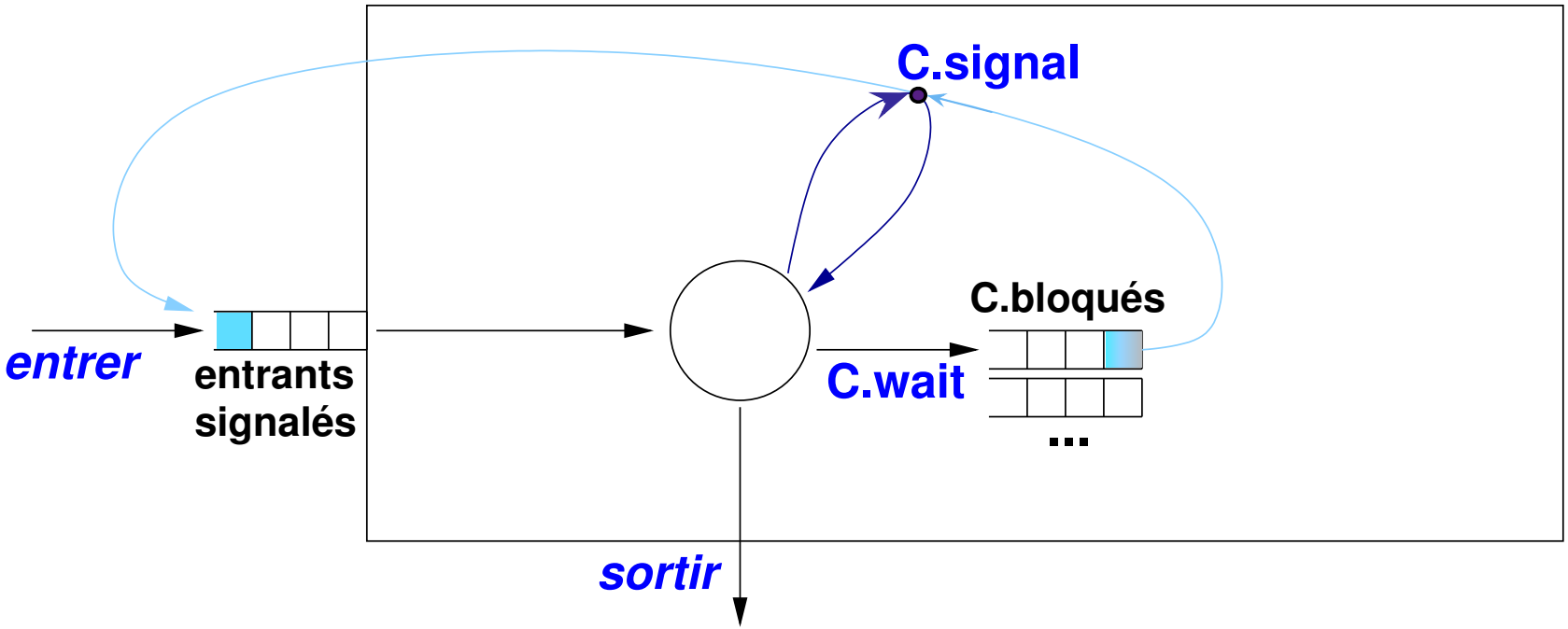
Priorité au signaleur avec file spécifique des signalés



C.signal()

- si la file des bloqués sur C est non vide, en extrait le processus de tête et le range dans la file des signalés
- le signaleur conserve le contrôle
- signalés prioritaires sur les entrants

Priorité au signaleur sans file spécifique des signalés



C.signal()

- si la file des bloqués sur C est non vide, en extrait le processus de tête et le range dans la file des entrants
- le signaleur conserve le contrôle
- signalés non prioritaires vis-à-vis des entrants

Exemple signaleur vs signalé : travail délégué avec 1 client, 2 ouvriers

Priorité au signalé

OK : quand un client dépose une requête et débloque un ouvrier, celui-ci obtient immédiatement l'accès exclusif et prend la requête.

Priorité au signaleur

- KO : situation : ouvrier n°1 bloqué sur `Dépôt.attendre()`.
- Le client appelle `déposer_travail` et en parallèle, l'ouvrier n°2 appelle `prendre_travail`. L'ouvrier n°2 attend l'accès exclusif.
- Lors de `Dépôt.signaler()`, l'ouvrier n°1 est débloqué de la var. condition et se met en attente de l'accès exclusif.
- Quand le client libère l'accès exclusif, qui l'obtient ? Si ouvrier n°2, il « vole » la requête, puis ouvrier n°1 obtient l'accès exclusif et récupère `null`.

Comparaison des stratégies de transfert du contrôle

- **Priorité au signalé** : garantit que le processus réveillé obtient l'accès au moniteur dans l'état où il était lors du signal.
  - Raisonnement simplifié (le signaleur produit un état, directement utilisé par le signalé)
  - Absence de famine facilitée
- **Priorité au signaleur** : le réveillé obtient le moniteur ultérieurement, éventuellement après d'autres processus
  - Implantation du mécanisme plus simple et plus performante
  - Au réveil, le signalé doit retester la condition de déblocage
    - → Possibilité de famine, écriture et raisonnements plus lourds

Peut-on simplifier encore l'expression de la synchronisation ?

Peut-on simplifier encore l'expression de la synchronisation ?

Idée (d'origine)

Attente sur des prédicats,

plutôt que sur des événements (= variables de type condition)

→ opération unique : *attendre(B)*, B expression booléenne

Exemple : moniteur pour le tampon borné, avec *attendre*(prédicat)

```
variables d'état : req, rés --Requête/Résultat en attente (null si aucun(e))
```

entrée déposer_travail(*in* t)
  req ← t
	entrée prendre_travail(*out* t)
	  attendre(req ≠ null)
	  t ← req
	  req ← null

entrée lire_résultat(*out* r)
  attendre(rés ≠ null)
  r ← rés
  rés ← null
	entrée rendre_résultat(*in* y)
	  rés ← y

Pourquoi *attendre*(prédicat) n'est-elle pas disponible en pratique ?

Efficacité problématique :

$\Rightarrow$ à chaque nouvel état (= à chaque affectation),
évaluer chacun des prédicats attendus.

$\rightarrow$ **gestion de l'évaluation laissée au programmeur**

- à chaque prédicat attendu (P)
  est associée une variable de type condition (P_valide)
- *attendre*(P) est implantée par
  si $\neg$ P **alors** $P\_valide.attendre()$ **fsi** $\{P\}$
- le programmeur a la possibilité de signaler ($P\_valide.signaler()$)
  les instants/états (pertinents) où P est valide

Principe

- concevoir en termes de prédicats attendus, puis
- simuler cette attente de prédicats au moyen de variables de type condition

Exemple – le moniteur (reprise planche 10)

Le moniteur

```
variables d'état : req, rés --Requête/Résultat en attente (null si aucun(e))
variables condition : Dépôt, Dispo
```

entrée déposer_travail(*in* t) {(pas d'attente)} req ← t Dépôt.signaler()	*entrée* prendre_travail(*out* t) *si* req = null *alors* Dépôt.attendre() *finsi* t ← req req ← null {RAS}
entrée lire_résultat(*out* r) *si* rés = null *alors* Dispo.attendre() *finsi* r ← rés rés ← null {RAS}	*entrée* rendre_résultat(*in* y) {(pas d'attente)} rés ← y Dispo.signaler()

Plan

Méthodologie (1/3) : motivation

Moniteur = réalisation (et gestion) d'un objet partagé

→ permet de concevoir la synchronisation en termes d'interactions entre chaque processus et un objet partagé :
les seules interactions autorisées sont celles qui laissent l'objet partagé dans un état cohérent

→ Invariant du moniteur = ensemble des états possibles pour l'objet géré par le moniteur

Schéma générique : exécution d'une action A sur un objet partagé, caractérisé par un invariant I

1. *si* l'exécution de A (depuis l'état courant) invalide I
   alors attendre() finsi { prédicat d'acceptation de A}
2. effectuer A { → nouvel état courant E }
3. *réveiller()* les processus qui peuvent progresser à partir de E

Méthodologie (2/3)

Etapes

1. Déterminer l'interface du moniteur
2. Énoncer en français les prédicats d'acceptation de chaque opération
3. Déduire les variables d'état
   qui permettent d'écrire ces prédicats d'acceptation
4. Formuler l'invariant du moniteur et les prédicats d'acceptation
5. Associer à chaque prédicat d'acceptation une variable condition qui permettra d'attendre/signaler la validité du prédicat
6. Programmer les opérations, en suivant le protocole générique précédent
7. Vérifier que
   - l'invariant est vrai chaque fois que le contrôle du moniteur est transféré
   - les réveils ont lieu quand le prédicat d'acceptation est vrai

Méthodologie (3/3)

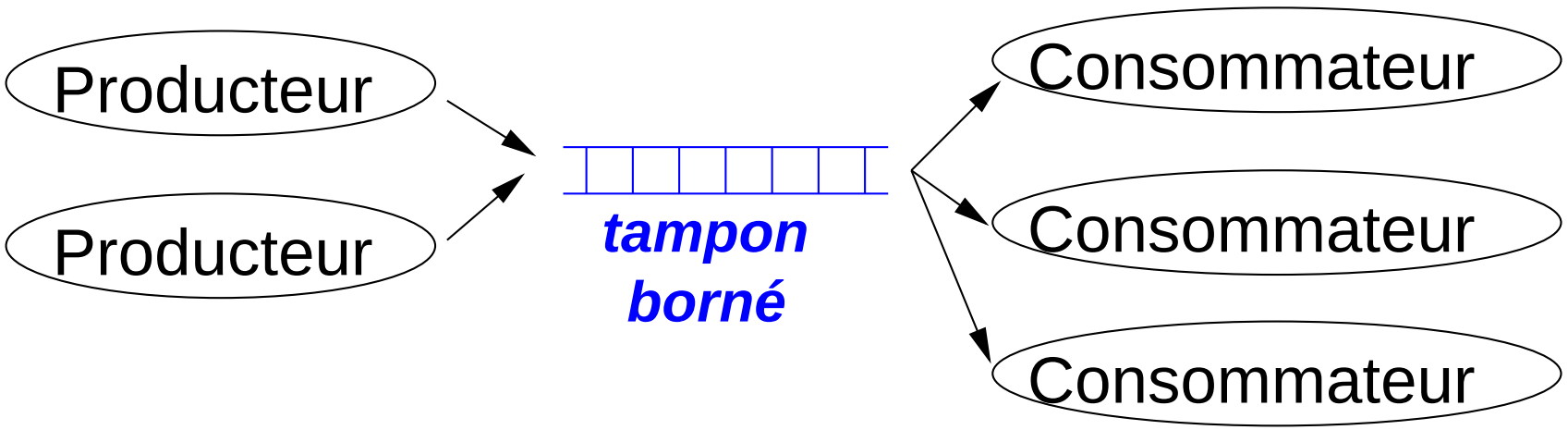
Structure standard d'une opération

si le prédicat d'acceptation est faux *alors*
    attendre() sur la variable condition associée
finsi
{ (1) État nécessaire au bon déroulement }
Mise à jour de l'état du moniteur (action)
{ (2) État garanti (résultat de l'action) }
signaler() les variables conditions dont le prédicat associé est vrai

Vérifier, pour chaque variable condition, que

chaque précondition de *signaler()* (2)
implique chaque postcondition de *attendre()* (1)

Exemple : réalisation du schéma producteurs/consommateurs



- tampon de taille borné et fixé
- nombre indéterminé et dynamique de producteurs
- " " " " de consommateurs

1. Interface :
   - `déposer(in v)`
   - `retirer(out v)`
2. Prédicats d'acceptation :
   - déposer : il y a de la place, le tampon n'est pas plein
   - retirer : il y a quelque chose, le tampon n'est pas vide
3. Variables d'état :
   - `nbOccupées` : natural
   - déposer : `nbOccupées` $< N$
   - retirer : `nbOccupées` > 0
4. Invariant : $0 \leq$ `nbOccupées` $\leq N$
5. Variables conditions : `PasPlein`, `PasVide`

1. Interface :
   - `déposer(in v)`
   - `retirer(out v)`
2. Prédicats d'acceptation :
   - déposer : il y a de la place, le tampon n'est pas plein
   - retirer : il y a quelque chose, le tampon n'est pas vide
3. Variables d'état :
   - `nbOccupées` : natural
   - déposer : `nbOccupées` $< N$
   - retirer : `nbOccupées` > 0
4. Invariant : $0 \leq$ `nbOccupées` $\leq N$
5. Variables conditions : `PasPlein`, `PasVide`

1. Interface :
   - `déposer(in v)`
   - `retirer(out v)`
2. Prédicats d'acceptation :
   - déposer : il y a de la place, le tampon n'est pas plein
   - retirer : il y a quelque chose, le tampon n'est pas vide
3. Variables d'état :
   - `nbOccupées` : natural
   - déposer : `nbOccupées` $< N$
   - retirer : `nbOccupées` > 0
4. Invariant : $0 \leq$ `nbOccupées` $\leq N$
5. Variables conditions : `PasPlein`, `PasVide`

1. Interface :
   - `déposer(in v)`
   - `retirer(out v)`
2. Prédicats d'acceptation :
   - déposer : il y a de la place, le tampon n'est pas plein
   - retirer : il y a quelque chose, le tampon n'est pas vide
3. Variables d'état :
   - `nbOccupées` : natural
   - déposer : `nbOccupées` $< N$
   - retirer : `nbOccupées` > 0
4. Invariant : $0 \leq$ `nbOccupées` $\leq N$
5. Variables conditions : `PasPlein`, `PasVide`

1. Interface :
   - `déposer(in v)`
   - `retirer(out v)`
2. Prédicats d'acceptation :
   - déposer : il y a de la place, le tampon n'est pas plein
   - retirer : il y a quelque chose, le tampon n'est pas vide
3. Variables d'état :
   - `nbOccupées` : natural
   - déposer : `nbOccupées` $< N$
   - retirer : `nbOccupées` > 0
4. Invariant : $0 \leq$ `nbOccupées` $\leq N$
5. Variables conditions : `PasPlein`, `PasVide`

1. Interface :
   - `déposer(in v)`
   - `retirer(out v)`
2. Prédicats d'acceptation :
   - déposer : il y a de la place, le tampon n'est pas plein
   - retirer : il y a quelque chose, le tampon n'est pas vide
3. Variables d'état :
   - `nbOccupées` : natural
   - déposer : `nbOccupées` $< N$
   - retirer : `nbOccupées` > 0
4. Invariant : $0 \leq$ `nbOccupées` $\leq N$
5. Variables conditions : `PasPlein`, `PasVide`

déposer(in v)

si $\neg$(nbOccupées $< N$) *alors*
    PasPlein.*attendre()*
finsi
{ (1) nbOccupées $< N$ }
// action applicative (ranger v dans le tampon)
nbOccupées++
{ (2) $N \geq$ nbOccupées > 0 }
PasVide.*signaler()*

retirer(out v)

si $\neg$(nbOccupées > 0) *alors*
    PasVide.*attendre()*
finsi
{ (3) nbOccupées > 0 }
// action applicative (prendre v dans le tampon)
nbOccupées--
{ (4) $0 \leq$ nbOccupées $< N$ }
PasPlein.*signaler()*

Vérification & Priorité

- Vérification : (2) ⇒ (3) ? (4) ⇒ (1) ?
- Si priorité au signaleur, transformer `si` en `tant que` :

déposer(in v)

```
tant que ¬(nbOccupées < N) faire
     PasPlein.wait
fintq
{ (1) nbOccupées < N }
// action applicative (ranger v dans le tampon)
nbOccupées++
{ (2) N ≥ nbOccupées > 0 }
PasVide.signal
```

Plan

Conclusion

Un moniteur implante un objet partagé, et contrôle la bonne utilisation de cet objet

Apports

- modularité et encapsulation.
- la synchronisation est localisée dans le moniteur $\rightarrow$
  - raisonnement simplifié
  - meilleure lisibilité

Limites

- dans le moniteur, la synchronisation reste mêlée aux aspects fonctionnels
- la sémantique des moniteurs est complexe
- l'exclusion mutuelle sur les opérations d'un moniteur facilite la conception, mais :
  - est une source potentielle d'interblocages (moniteurs imbriqués)
  - est une limite du point de vue de l'efficacité

Plan

Allocateur de ressources

- N ressources équivalentes, une activité en demande $p \in 1..N$ puis les libère.
- Bon comportement : pas deux demandes consécutives sans libération (cf interblocage).
- Difficulté : une libération peut débloquer 0, 1 ou plusieurs demandeurs selon le nombre de ressources rendues et attendues.

Allocateur de ressources - méthodologie

1. Interface :
   - `demander(p: 1..N)`
   - `libérer(q: 1..N)`
2. Prédicats d'acceptation :
   - demander(p) : il y a au moins p ressources libres
   - retirer(q) : rien
3. Variables d'état :
   - `nbDispo` : natural
   - demander(p) : `nbDispo` $\geq p$
   - libérer(q) : *true*
4. Invariant : $0 \leq$ `nbDispo` $\leq N$
5. Variable condition : `AssezDeRessources`

Allocateur de ressources - méthodologie

1. Interface :
   - `demander(p: 1..N)`
   - `libérer(q: 1..N)`
2. Prédicats d'acceptation :
   - demander(p) : il y a au moins p ressources libres
   - retirer(q) : rien
3. Variables d'état :
   - `nbDispo` : natural
   - demander(p) : `nbDispo` $\geq p$
   - libérer(q) : *true*
4. Invariant : $0 \leq$ `nbDispo` $\leq N$
5. Variable condition : `AssezDeRessources`

Allocateur de ressources - méthodologie

1. Interface :
   - `demander(p: 1..N)`
   - `libérer(q: 1..N)`
2. Prédicats d'acceptation :
   - demander(p) : il y a au moins p ressources libres
   - retirer(q) : rien
3. Variables d'état :
   - `nbDispo` : natural
   - demander(p) : `nbDispo` $\geq p$
   - libérer(q) : *true*
4. Invariant : $0 \leq$ `nbDispo` $\leq N$
5. Variable condition : `AssezDeRessources`

Allocateur de ressources - méthodologie

1. Interface :
   - `demander(p: 1..N)`
   - `libérer(q: 1..N)`
2. Prédicats d'acceptation :
   - demander(p) : il y a au moins p ressources libres
   - retirer(q) : rien
3. Variables d'état :
   - `nbDispo` : natural
   - demander(p) : `nbDispo` $\geq p$
   - libérer(q) : *true*
4. Invariant : $0 \leq$ `nbDispo` $\leq N$
5. Variable condition : `AssezDeRessources`

Allocateur de ressources - méthodologie

1. Interface :
   - `demander(p: 1..N)`
   - `libérer(q: 1..N)`
2. Prédicats d'acceptation :
   - demander(p) : il y a au moins p ressources libres
   - retirer(q) : rien
3. Variables d'état :
   - `nbDispo` : natural
   - demander(p) : `nbDispo` $\geq p$
   - libérer(q) : *true*
4. Invariant : $0 \leq$ `nbDispo` $\leq N$
5. Variable condition : `AssezDeRessources`

Allocateur de ressources - méthodologie

1. Interface :
   - `demander(p: 1..N)`
   - `libérer(q: 1..N)`
2. Prédicats d'acceptation :
   - demander(p) : il y a au moins p ressources libres
   - retirer(q) : rien
3. Variables d'état :
   - `nbDispo` : natural
   - demander(p) : `nbDispo` $\geq p$
   - libérer(q) : *true*
4. Invariant : $0 \leq$ `nbDispo` $\leq N$
5. Variable condition : `AssezDeRessources`

Allocateur – opérations

demander(p)

si $\neg(\texttt{nbDispo} < p)$ alors

    AssezDeRessources.wait

finsi
$\texttt{nbDispo} \leftarrow \texttt{nbDispo} - p$

libérer(q)

$\texttt{nbDispo} \leftarrow \texttt{nbDispo} + p$
si c'est bon alors -- comment le coder ?
    AssezDeRessources.signal
finsi

Allocateur – opérations

demander(p)

si $\neg(\texttt{nbDispo} < p)$ alors
    $\texttt{demande} \leftarrow p$
    `AssezDeRessources.wait`
    $\texttt{demande} \leftarrow 0$
finsi
$\texttt{nbDispo} \leftarrow \texttt{nbDispo} - p$

libérer(q)

$\texttt{nbDispo} \leftarrow \texttt{nbDispo} + p$
si $\texttt{nbDispo} \geq \texttt{demande}$ alors
    AssezDeRessources.signal
finsi

Allocateur – opérations

demander(p)

Et s'il y a plusieurs demandeurs ?

si $\neg(\texttt{nbDispo} < p)$ alors
    $\texttt{demande} \leftarrow p$
    `AssezDeRessources.wait`
    $\texttt{demande} \leftarrow 0$
finsi
$\texttt{nbDispo} \leftarrow \texttt{nbDispo} - p$

libérer(q)

$\texttt{nbDispo} \leftarrow \texttt{nbDispo} + p$
si $\texttt{nbDispo} \geq \texttt{demande}$ alors
    AssezDeRessources.signal
finsi

Allocateur – opérations

demander(p)

```
si demande ≠ 0 alors -- il y a déjà un demandeur → j'attends mon tour
    Sas.wait
finsi
si ¬(nbDispo < p) alors
    demande ← p
    AssezDeRessources.wait     -- au plus un bloqué ici
    demande ← 0
finsi
nbDispo ← nbDispo − p
Sas.signal    -- au suivant de demander
```

libérer(q)

```
nbDispo ← nbDispo + p
si nbDispo ≥ demande alors
    AssezDeRessources.signal
finsi
```

Note : priorité au signaleur ⇒ transformer le premier “si” de demander en “tant que” (ça suffit ici).

Variante : réveil multiple : signalAll/broadcast

C.signalAll (ou broadcast) : *toutes* les activités bloquées sur la variable condition *C* sont débloquées. Elles se mettent en attente de l'accès exclusif.

Rarement utilisé à bon escient. Une solution triviale à un problème de synchronisation est d'utiliser une *unique* variable condition `Accès` et d'écrire *toutes* les procédures du moniteur sous la forme :

```
tant que ¬(condition d'acceptation) faire
    Accès.wait
fintq
...
Accès.signalAll  -- battez-vous
```

Mauvaise idée ! (performance, prédictibilité)

Réveil multiple : cour de récréation unisexe

0. `type genre` $\triangleq$ `(Fille, Garçon)`
   `inv(g)` $\triangleq$ `si g = Fille alors Garçon sinon Fille`
1. Interface : entrer(genre) / sortir(genre)
2. Prédicats : entrer : personne de l'autre sexe / sortir : –
3. Variables : nb(genre)
4. Invariant : nb(Filles) $= 0 \vee$ nb(Garçons) $= 0$
5. Variables condition : accès(genre)
6.

```
entrer(genre g)
  si nb(inv(g)) ≠ 0 alors
     accès(g).wait
  finsi
  nb(g)++
```

```
sortir(genre g)
  nb(g)--
  si nb(g) = 0 alors
      accès(inv(g)).signalAll
  finsi
```

(solution naïve : risque de famine si un genre se coalise pour avoir toujours un membre présent dans la cour)

Priorité au signaleur : transformation systématique ?

Pour passer de priorité au signalé à priorité au signaleur, transformer « `si CA` » en « `tant que CA` » n'est correct que si la condition d'acceptation (à l'entrée) et la condition de déblocage (au réveil) sont identiques.
Exemple : évitement de la famine : variable `attente(genre)` pour compter les enfants en attente et ne pas accaparer la cour.

```
entrer(genre g)
  si nb(inv(g)) ≠ 0 ∨ attente(inv(g)) ≥ 4 alors
     attente(g)++
     accès(g).wait
     attente(g)--
  finsi
  nb(g)++
```

Interblocage possible avec priorité signaleur et « tant que » à la place du « si » → repenser la solution.

Variante : régions critiques

- Éliminer les variables conditions et les appels explicites à `signaler` $\Rightarrow$ déblocages calculés par le système.
- Exclusion mutuelle plus « fine », en listant les variables partagées effectivement utilisées.

```
region liste des variables utilisées
when prédicat logique
do code
```

1. Attente que le prédicat logique soit vrai
2. Le code est exécuté en exclusion mutuelle vis-à-vis des autres régions ayant (au moins) une variable commune
3. À la fin du code, évaluation automatique des prédicats logiques des régions pour débloquer éventuellement.

Exemple

```
tampon : shared array 0..N-1 of msg;
nbOcc : shared int := 0;
retrait, dépôt : shared int := 0, 0;
```

```
déposer(m)
  region
    nbOcc, tampon, dépôt
  when
    nbOcc < N
  do
    tampon[dépôt] ← m
    dépôt ← dépôt + 1 % N
    nbOcc ← nbOcc + 1
  end
```

```
retirer()
  region
    nbOcc, tampon, retrait
  when
    nbOcc > 0
  do
    Result ← tampon[retrait]
    retrait ← retrait + 1 % N
    nbOcc ← nbOcc - 1
  end
```

Implémentation des moniteurs par des sémaphores FIFO

Dans le cas où les *signaler()* sont toujours en fin d'opération

- Exclusion mutuelle sur l'exécution des opérations du moniteur
  - définir un sémaphore d'exclusion mutuelle : *mutex*
  - encadrer chaque opération par *mutex.P()* et *mutex.V()*
- Réalisation de la synchronisation par variables condition
  - définir un sémaphore *SemC* (initialisé à 0) pour chaque condition *C*
  - traduire *C.attendre()* par *SemC.P()*, et *C.signaler()* par *SemC.V()*
  - Difficulté : pas de mémoire pour les appels à *C.signaler()*
    - → éviter d'exécuter *SemC.V()* si aucun processus n'attend
    - → un compteur explicite par condition : *cptC*
      - Réalisation de *C.signaler()* :
        `si` cptC $>$ 0 `alors` SemC.V() `sinon` mutex.V() `fsi`
      - Réalisation de *C.attendre()* :
        cptC ++ ; mutex.V() ; SemC.P() ; cptC - - ;

Dans le cas général : ajout d'un compteur et d'un sémaphore pour les processus signaleurs, réveillé prioritairement par rapport à *mutex*