Comment régler les problèmes de synchronisation ?

P-A. Champin

Département Informatique
IUT A – Université Claude Bernard Lyon 1
2009

Problématiques

- Ressource critique
- Problème des producteurs/consommateurs
- Problème des lecteurs/rédacteurs
- Problème des philosophes
- Exclusion mutuelle au sein du SE

Problématiques

- Ressource critique
- Problème des producteurs/consommateurs
- Problème des lecteurs/rédacteurs
- Problème des philosophes
- Exclusion mutuelle au sein du SE

Ressource critique Exemple sans problème

tâche 1 (compte += 1000)	tâche 2 (compte -= 500)	ressources *compte*	*r1*	*r2*
charger (r1, compte)		2000	**2000**	0
charger (r2, 1000)		2000	2000	**1000**
ajouter (r1, r2)		2000	**3000**	1000
écrire (r1, compte)		**3000**	3000	1000
sauvegarde contexte			3000	1000
	restauration contexte		**0**	**0**
	charger (r1, compte)	3000	**3000**	0
	charger (r2, 500)	3000	3000	**500**
	soustraire (r1, r2)	3000	**2500**	500
	écrire (r1, compte)	**2500**	2500	500

Ressource critique
Exemple avec un problème

tâche 1 (compte += 1000)	tâche 2 (compte -= 500)	compte	r1	r2
charger (r1, compte)		2000	2000	0
charger (r2, 1000)		2000	2000	1000
ajouter (r1, r2)		2000	3000	1000
commutation de contexte →			0	0
	charger (r1, compte)	2000	2000	0
	charger (r2, 500)	2000	2000	500
	soustraire (r1, r2)	2000	1500	500
	écrire (r1, compte)	1500	1500	500
← commutation de contexte			3000	1000
écrire (r1, compte)		3000	3000	1000

(ressources : compte, r1, r2)

Définitions autour de la notion de ressource critique

- Une ressource est dite **ressource critique** lorsque des accès concurrents à cette ressources peuvent résulter dans un état incohérent
- On parle aussi de **situation de compétition** (*race condition*) pour décrire une situation dont l'issue dépend de l'ordre dans lequel les opérations sont effectuées
- Une **section critique** est une section de programme manipulant une ressource critique

Définition : Exclusion mutuelle

- Une section de programme est dite **atomique** lorsqu'elle ne peut pas être interrompue par un autre processus *manipulant les mêmes ressources critiques*
  - c'est donc une atomicité *relative* à la ressource
- Un mécanisme d'**exclusion mutuelle** sert à assurer l'atomicité des sections critiques relatives à une ressource critique
  - en anglais : *mutual exclusion*, ou **mutex**

Exclusion mutuelle Mise en oeuvre

entier compte

proc entrer_SC_compte ()

proc sortir_SC_compte ()

afficher (« Ajout de 1000€ »)

entrer_SC_compte ()

compte += 1000

sortir_SC_compte ()

afficher («Ajout effectué »)

afficher (« Retrait de 500€ »)

entrer_SC_compte ()

compte -= 500

sortir_SC_compte ()

afficher («Retrait effectué »)

Exclusion mutuelle
Critères d'évaluation

- **Exclusion** : deux tâches ne doivent pas se trouver en même temps en SC
- **Progression** : une tâche doit pouvoir entrer en SC si aucune autre ne s'y trouve
- **Équité** : une tâche ne devrait pas attendre indéfiniment pour entrer SC
- L'exclusion mutuelle doit fonctionner dans un contexte **multi-processeurs**

Exclusion mutuelle
Principe d'une solution

- On associe à la ressource un *jeton*, que les processus peuvent prendre et reposer
- Seul le processus possédant le jeton **devrait** manipuler la ressource
- Donc, si un processus souhaite manipuler la ressource et que le jeton est pris, il doit d'abord attendre que le jeton redevienne disponible
- Utilisation d'un sémaphore initialisé à 1

Exclusion mutuelle Exemple

```
int compte = 2000;
sem_t mutex_compte = sem_open("/compte", flags, mode, 1);
```

```
printf ("Ajout de 1000€\n");
sem_wait (mutex_compte);
compte += 1000;
sem_post (mutex_compte);
printf ("Ajout effectué\n");
```

```
printf ("Retrait de 500€\n");
sem_wait (mutex_compte);
compte -= 500;
sem_post (mutex_compte);
printf ("Retrait effectué\n");
```

Remarques sur le problème de ressource critique

- Certaines opérations sur la ressource critiques ne nécessitent pas forcément d'exclusion mutuelle
  - exemple : lecture de la valeur du compte (si on accepte que la valeur lue devienne rapidement invalide...)
- Le mécanisme d'exclusion mutuelle n'est **pas** une protection, mais une **convention** entre les processus souhaitant utiliser la ressource sans la corrompre
  - il est toujours possible de manipuler la ressource sans « prendre le jeton »

Remarque sur les sections critiques (1)

- Les sections critiques sont un **mal nécessaire** :
- un mal, parce qu'elles empêches le parallélisme qu'on a eu tant de mal à mettre en place... elles réduisent donc les performances
- nécessaire, parce qu'elles garantissent l'intégrité des ressources critiques
- Conséquence : lorsqu'on peut les éviter, il faut le faire (structures de données toujours cohérentes)

Remarque sur les sections critiques (2)

- Étant donné qu'un processus en section critique est susceptible d'en bloquer d'autres, il est souhaitable qu'il reste le moins longtemps possible en section critique, et surtout qu'il ne s'y *bloque pas*.

```
// n'écrivez pas :
sem_wait (mutex_compte);
printf ("Ajout de 1000€\n");
compte += 1000;
sem_post (mutex_compte);
```

```
// mais écrivez :
printf ("Ajout de 1000€\n");
sem_wait (mutex_compte);
compte += 100;
sem_post (mutex_compte);
```

Définition : Inter-blocage

- Un interblocage (*deadlock*) esr une situation ou deux processus (ou plus) sont bloqués en attente d'un événement qui doit être produit par l'autre

```
sem_wait (mutex_A);
utiliser_ressource(A);
sem_wait (mutex_B);
utiliser_ressources(A,B);
// ...
```

```
sem_wait (mutex_B);
utiliser_ressource(B);
sem_wait (mutex_A);
utiliser_ressources(A,B);
// ...
```

Inter-blocage
Solution possible

- Une manière d'éviter les inter-blocages consiste à toujours réclamer les ressources dans le même ordre (quitte à libérer une ressource avant de la réclamer à nouveau)

```
sem_wait (mutex_A);

utiliser_ressource(A);

sem_wait (mutex_B);

utiliser_ressources(A,B);

// ...
```

```
sem_wait (mutex_B);

utiliser_ressource(B);

sem_post (mutex_B);

sem_wait (mutex_A);

sem_wait (mutex_B);

utiliser_ressources(A,B); // ...
```

Problématiques

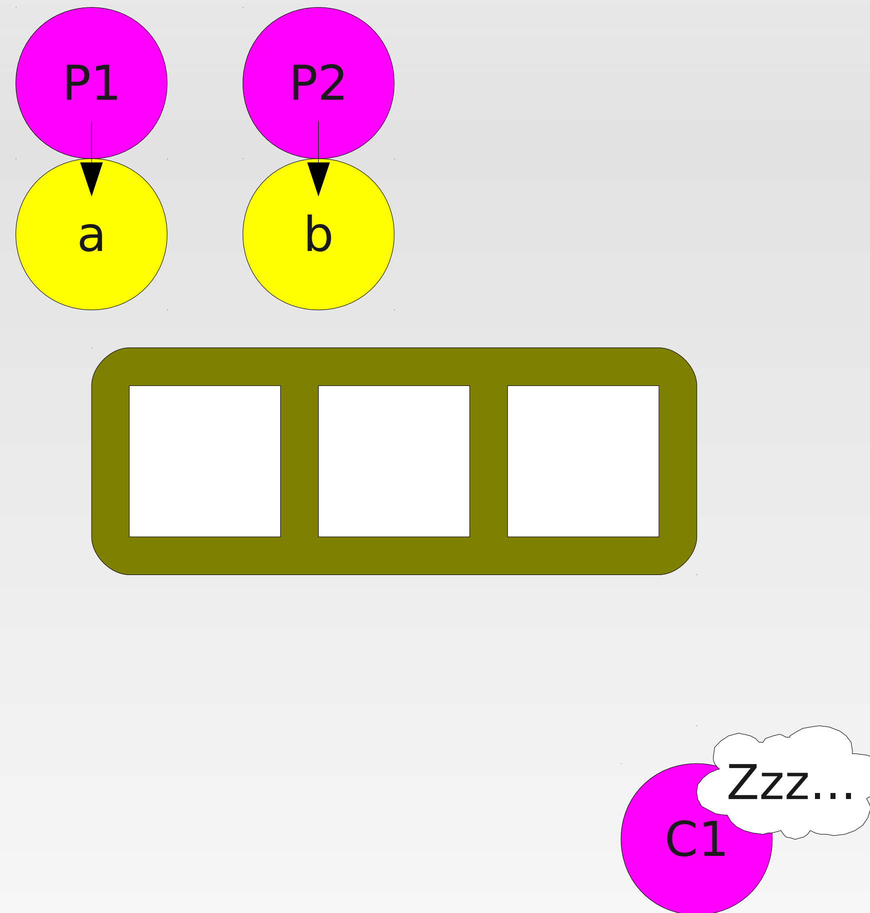
- Ressource critique
- Problème des producteurs/consommateurs
- Problème des lecteurs/rédacteurs
- Problème des philosophes

- Exclusion mutuelle au sein du SE

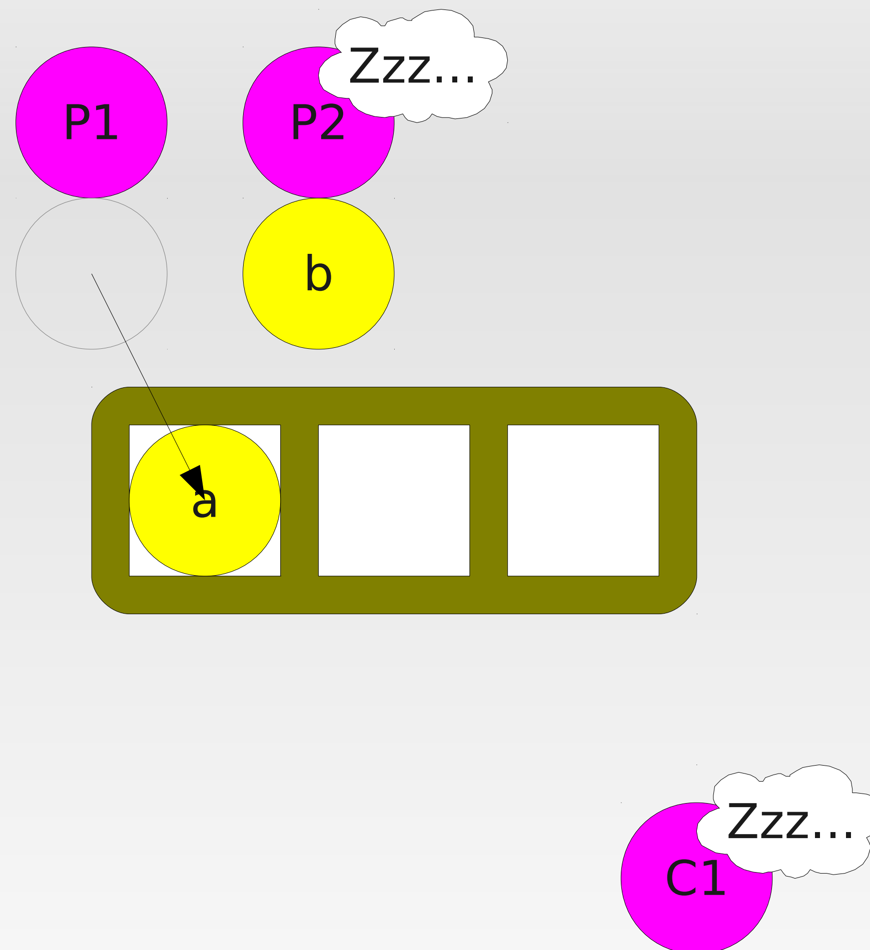
Producteurs - consommateurs Scénario

- Un ensemble de processus, divisé en deux catégories, partage une zone mémoire
- Les premiers *(producteurs)* remplissent la mémoire partagée, avec des éléments
  - la mémoire ne peut contenir qu'un nombre d'éléments limité et connu à l'avance
- Les seconds *(consommateurs)* utilisent ces éléments et les retirent de la mémoire
- Exemple : file d'impression

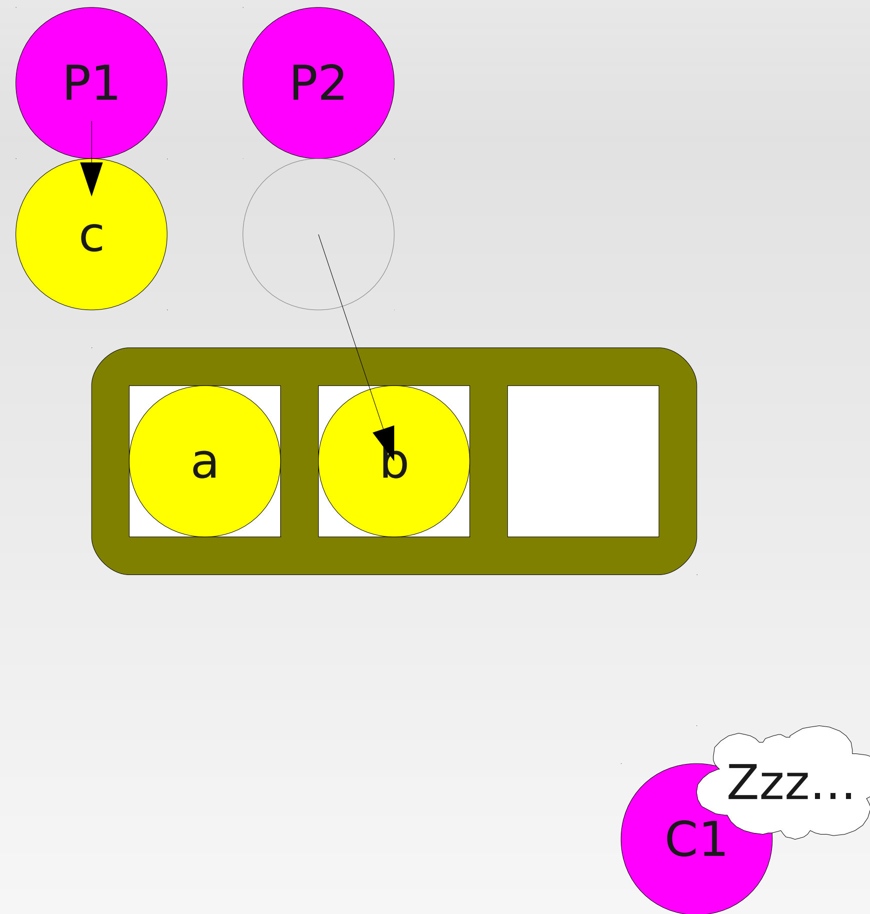
Producteurs - consommateurs Illustration



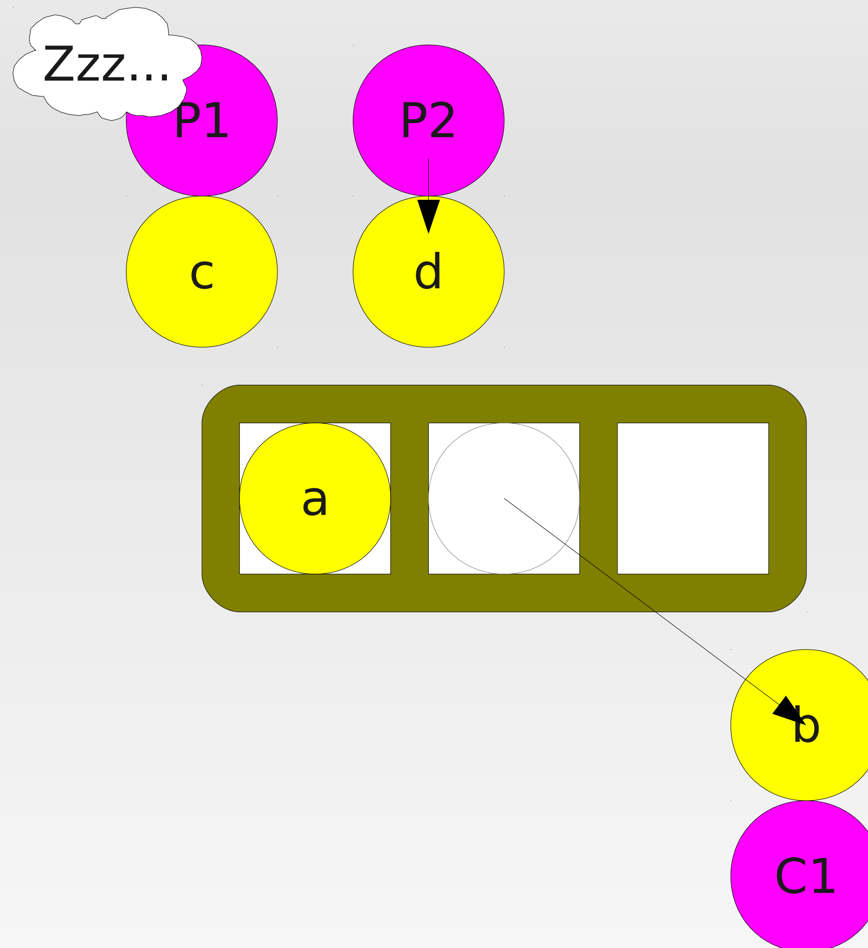
Producteurs - consommateurs Illustration



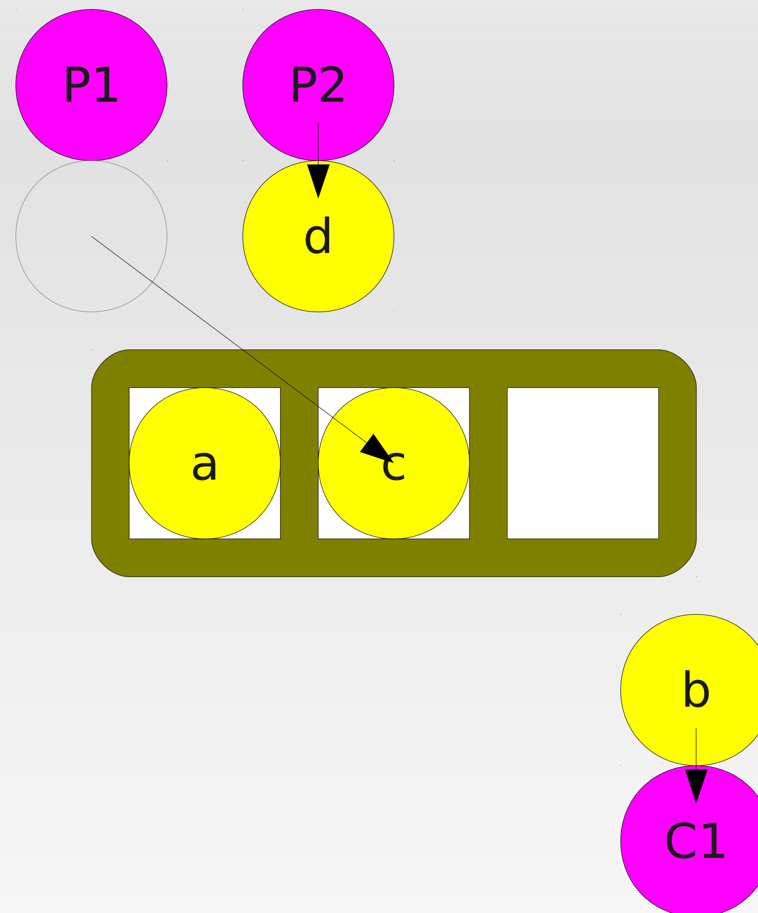
Producteurs - consommateurs Illustration



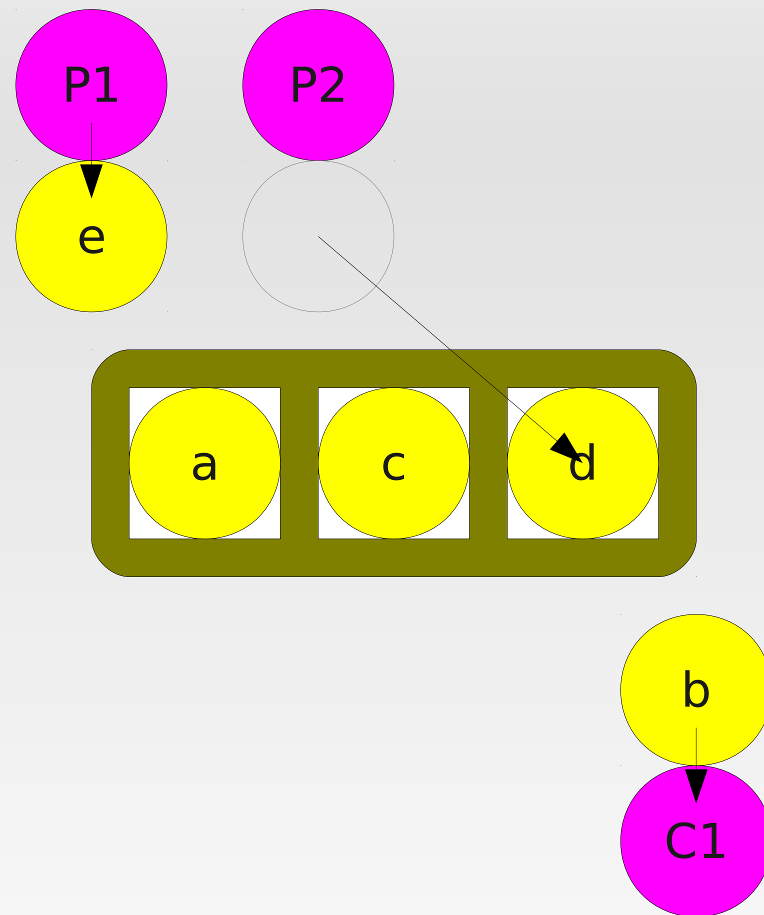
Producteurs - consommateurs
Illustration



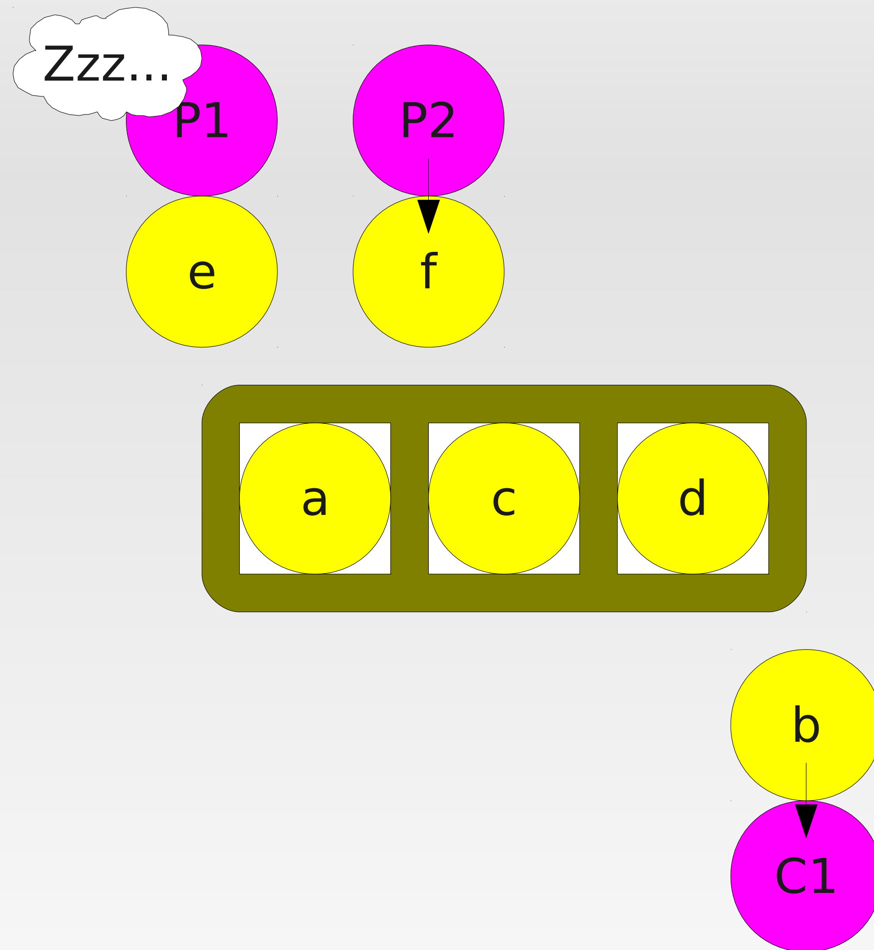
Producteurs - consommateurs Illustration



Producteurs - consommateurs Illustration



Producteurs - consommateurs Illustration



Producteurs - consommateurs Problème

- Un producteur doit se bloquer lorsque la mémoire partagée est pleine
- Un consommateur doit se bloquer lorsque la mémoire partagée est vide

Producteurs - consommateurs Solution – principe (1)

- Utilisation de deux sémaphores, implémentant les deux critères de blocage
  - l'un représente le nombre de cases libres
  - l'autre représente le nombre de cases occupées
- Analogie : deux piles de jetons, avec une quantité de jetons constante
  - les jetons passent d'une pile dans une autre lorsqu'une case change d'état

Producteurs - consommateurs Solution – principe (2)

- Si les opérations sur la zone mémoire partagée ne sont pas atomiques, il faut les protéger par une section critique
  $\rightarrow$ troisième sémaphore

Producteurs - consommateurs Solution – Algorithme

```
sémaphore pleines(0), vides(n), mutex (1)

élément tab[n]
```

```
proc Producteur :

tant que VRAI:
    élément e ← produire()
    sem_wait(vides)
      sem_wait(mutex)
        ajouter(e, tab)
      sem_post(mutex)
    sem_post(pleines)
```

```
proc Consommateur :

tant que VRAI:
    sem_wait(pleines)
      sem_wait(mutex)
        élément e ← retirer(tab)
      sem_post(mutex)
    sem_post(vides)
    consommer(e)
```

Producteurs - consommateurs Remarques

- Dans l'algorithme précédent, l'ordre dans lequel les sémaphores sont utilisés est primordial, sans quoi on risque un inter-blocage
- Les opérations de production et de consommation à proprement parler doivent être en dehors des sections critiques
  - sans quoi seul un processus pourrait s'exécuter à un moment donné
- De même, elles doivent être en dehors du « passage de jeton » entre *pleines* et *vides*

Problématiques

- Ressource critique
- Problème des producteurs/consommateurs
- Problème des lecteurs/rédacteurs
- Problème des philosophes
- Exclusion mutuelle au sein du SE

Lecteurs - rédacteurs Scénario

- Un ensemble de processus, divisé en deux catégories, partage une zone mémoire
- Certains processus (les *lecteurs*) font des accès en lecture seule à cette zone
- D'autres processus (les *rédacteurs*) modifient le contenu de cette zone
- NB : on parle parfois d'*écrivains* plutôt que de rédacteurs

Lecteurs - rédacteurs Problème

- Lorsqu'un rédacteur accède à la mémoire partagée, aucune autre processus (qu'il soit lecteur ou rédacteur) ne doit y avoir accès
  $\rightarrow$ problème d'exclusion mutuelle classique
- En revanche, les lecteurs peuvent être plusieurs à utiliser la zone en même temps, cela ne pose pas de problème

Lecteurs - rédacteurs Solution - Principe

- On protège la mémoire partagée par une exclusion mutuelle, mais...
- les lecteurs n'ont besoin de cette exclusion mutuelle **que si** aucun autre lecteur n'utilise la mémoire ;
- pour le vérifier, ils utilisent un compteur (nombre de lecteurs en train d'utiliser la zone), lui aussi protégé par une exclusion mutuelle

Lecteurs - rédacteurs
Solution – Algorithme

```
sémaphore  mutex_m(1),
           mutex_c(1)

entier c // nombre de lecteurs
```

```
proc Rédacteur :
    sem_wait(mutex_m)
    écrire()
    sem_post(mutex_m)
```

```
proc Lecteur :
    sem_wait(mutex_c)
     si c = 0 alors sem_wait(mutex_m)
     c ← c + 1
    sem_post(mutex_c)

    lire()

    sem_wait(mutex_c)
     si c = 1 alors sem_post(mutex_m)
     c ← c - 1
    sem_post(mutex_c)
```

Lecteurs - rédacteurs Remarques

- Le lecteur qui prend le jeton de *mutex_m* n'est *pas forcément* celui qui le rend
- On note que les lecteurs peuvent parfois se bloquer dans la section critique du compteur *c*.
  - Cela peut-il créer des inter-blocages ?
- L'algorithme proposé fonctionne, mais n'assure pas l'équité : un rédacteur peut attendre indéfiniment avant d'avoir accès à la mémoire (famine)
  - Comment le modifier pour assurer l'équité ?

Problématiques

- Ressource critique
- Problème des producteurs/consommateurs
- Problème des lecteurs/rédacteurs
- Problème des philosophes
- Exclusion mutuelle au sein du SE

Problème des philosophes Scénario (Dijkstra)

- Cinq philosophes passent leur vie à penser et manger autour d'une table
- Pour manger, ils ont besoin de deux fourchettes, mais il n'y a que cinq fourchettes

Illustration : © Benjamin Esham
http://en.wikipedia.org/wiki/Image:Dining_philosophers.png

Problème des philosophes
Problème

- Proposer un algorithme qui évite :
  - les inter-blocages
  - la famine
- Mauvaise solution :
  - chaque philosophe, lorsqu'il a faim, prend la fourchette de gauche, puis celle de droite, mange, puis les repose

Problème des philosophes Solution - Principe

- Chaque philosophe a trois états :
  « je pense », « j'ai faim », « je mange »
  par lesquels il passe toujours dans cet ordre
- Lorsqu'il a faim, un philosophe ne peut manger que si ses deux voisins ne mangent pas, sinon attend
- Lorsqu'il termine de manger, le philosophe réveille ses voisins et se remet à penser

Problème des philosophes
Solution - Algorithme

```
proc CommenceManger(id) :
    sem_wait(mutex)
    états[id] ← FAIM
    ok ← état[id-1] ≠ MANGE et
    état[id+1] ≠ MANGE
    si ok alors
      état[id] ← MANGE
      sem_post(mutex)
    sinon
      sem_post(mutex)
      sem_wait(réveils[id])
```

```
sémaphore  mutex(1),
           réveils[n] // init 0

entier états[n] // init PENSE
```

```
proc FinitManger(id) :
    sem_wait(mutex)
    états[id] ← PENSE
    si état[id-1] = FAIM
    et état[id-2] ≠ MANGE alors
      état[id-1] ← MANGE
      sem_post(réveils[id-1])
    // idem pour i+1 et i+2
    sem_post(mutex)
```

Problème des philosophes Remarques

- L'algorithme proposé prévient les inter-blocages, mais pas la famine
- Pour l'améliorer, une solution consiste à ne réveiller un voisin que si son autre voisin (*id*±2) n'est pas dans l'état FAIM depuis trop longtemps / depuis plus longtemps que lui

Problématiques

- Ressource critique
- Problème des producteurs/consommateurs
- Problème des lecteurs/rédacteurs
- Problème des philosophes

- Exclusion mutuelle au sein du SE

Besoin

- Les appels systèmes manipules de nombreuses structures
  - Exemple : PCB, table des fichiers ouverts, descripteur de fichier ouvert, table de pages, etc...
- Dans un système multi-processeurs, plusieurs appels systèmes peuvent s'exécuter en même temps

  $\rightarrow$ nécessité d'exclusion mutuelle au sein des appels systèmes

Problème (1)

- Implémentation naïve d'un sémaphore
- Problème : les deux algorithmes sont des sections critiques pour le compteur *c* et la file d'attente des processus bloqués

```
proc sem_wait(sem) :
    si sem.c = 0 alors
        bloquer le processus
    finsi
    sem.c ← sem.c - 1
```

```
proc sem_post(sem) :
    sem.c ← sem.c + 1
    si un processus est bloqué en
    attente du sémaphore alors
        débloquer ce processus
    finsi
```

Problème (2)

- Protection de la section critique par une exclusion mutuelle : il faut utiliser *autre chose* qu'un sémaphore !

```
proc sem_wait(sem) :
    entrer_en_SC()
    si sem.c = 0 alors
      sortir_de_SC()
      bloquer le processus
    sinon
      sem.c ← sem.c - 1
      sortir_de_SC()
    finsi
```

```
proc sem_post(sem) :
    entrer_en_SC()
    sem.c ← sem.c + 1
    si un processus est bloqué en
    attente du sémaphore alors
      débloquer ce processus
      sem.c ← sem.c - 1
    finsi
    sortir_de_SC()
```

Attente active

- **Attente active** : le processeur est occupé à attendre
  - par opposition à un processus *bloqué*, qui n'utilise pas le processeur
- Implémentation naïve de l'exclusion mutuelle
  - n'assure pas l'exclusion...

```
proc enter_SC() :
    tant que verrou faire
      <rien>
    fintant
    verrou ← VRAI
```

```
global booléen verrou = FAUX
```

```
proc sortir_SC() :
    verrou ← FAUX
```

Solution matérielle

- L'instruction processeur **TEST_AND_SET** consulte et change une variable en une seule action (atomique)
  - si la variable est à 1, elle retourne 1
  - si la variable est à 0, elle la met à 1 et retourne 0
- Supprime la situation de compétition entre le test de la boucle et l'affectation du verrou

```
proc enter_SC() :
    tant que TEST_AND_SET(verrou)
    faire
      <rien>
    fintant
```

Solutions logicielles

- Il existe des solutions purement logicielles (i.e. qui ne nécessitent pas l'existence d'instruction atomique pour tester et modifier une variable)
  - algorithme de Peterson
  - algorithme de la Boulangerie
- Mécanismes plus complexe qu'un simple verrou, pour ne rien avoir à faire *après* l'attente active
  - tableau de verrous
  - notion de tour

Implémentation effective du sémaphore (1)

```
proc sem_wait(sem) :
    entrer_en_SC()
    si sem.c = 0 alors
      sortir_de_SC()
      bloquer le processus
    sinon
      sem.c ← sem.c - 1
      sortir_de_SC()
    finsi
```

```
proc sem_post(sem) :
    entrer_en_SC()
    sem.c ← sem.c + 1
    si un processus est bloqué
    en attente du sémaphore
    alors
      débloquer ce processus
      sem.c ← sem.c - 1
    finsi
    sortir_de_SC()
```

Implémentation effective du sémaphore (2)

```
proc sem_wait(sem) :
    entrer_en_SC()
    si sem.c = 0 alors
      masquer_interruption()
      sem.f.ajouter(processus)
      processus.état ← BLOQUÉ
      sortir_de_SC()
      dé-masquer_interruption()
      rendre_la_main()
    sinon
      sem.c ← sem.c - 1
      sortir_de_SC()
    finsi
```

```
proc sem_post(sem) :
    entrer_en_SC()
    sem.c ← sem.c + 1
    si non f.est_vide() alors
      p ← sem.f.retirer()
      p.état ← ÉLIGIBLE
      sem.c ← sem.c - 1
    finsi
    sortir_de_SC()
```

En conclusion

- Les sémaphores permettent de régler de nombreux problèmes de synchronisation entre processus
- Problèmes classiques
- Au sein du système d'exploitation, des mécanismes de synchronisation « bas niveau » sont également nécessaires : attente active avec TEST_AND_SET