Module NOY
Mise en œuvre de noyaux de systèmes

Isabelle Puaut

istic Informatique Électronique

Plan du module

- Appels système et synchronisation
- Gestion de la mémoire
- Entrées-sorties
- Sécurité
- Virtualisation

(Ordre à confirmer, cours en chantier)

Mise en œuvre d'un noyau de système d'exploitation

- Mécanismes de contrôle de l'exécution
- Représentation des processus
- Mise en œuvre des ordonnanceurs
- Mise en œuvre de la synchronisation
  - Exclusion mutuelle
  - Sémaphores
  - Particularités des architectures multi-cœurs

Mécanismes de contrôle de l'exécution

- Mécanismes de protection : isolation
  - Système vis à vis de l'usager
  - Usagers vs usagers
- Commutation du contexte d'exécution

Mécanismes de protection

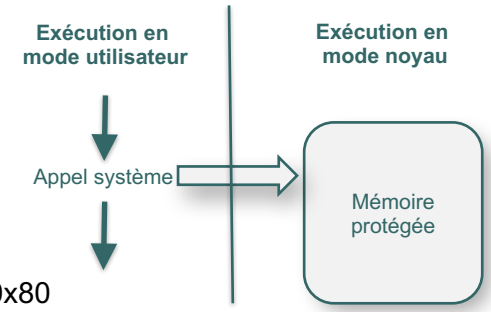
- Contrôle de l'accès à la mémoire
- Contrôle de l'utilisation des instructions
  - Exemple : *cli, sti in, out, hlt*
- Modes d'exécution pour le processeur
  - Mode noyau (superviseur) exécution de toutes les instructions est possible
  - Mode utilisateur (userland) exécution de certaines instructions et accès à certaines zones mémoire) interdit
  - Mode courant : mot d'état du processeur (registre, SR)
- Conjointement à l'adressage virtuel (usagers vs usagers)

Changement de mode d'exécution Appels système

- Réalisation de certaines opérations (entrées/sorties) nécessite de passer en mode noyau
- Passage en mode noyau doit être contrôlé ⇒ Instruction spécialisée de passage en mode noyau, appel système
  - *int* en x86, *syscall* en MIPS, *trap* en 68k

Changement de mode d'exécution
Appels système

- Passage en mode noyau + branchement à une adresse fixée



  - x86/Linux : int 0x80
    - Numéro appel système dans rax (ex 1 = exit)
    - Paramètres dans des registres (rbx, rcx, rdx, rsi et rdi)
    - (section 2 du man pour la liste des appels système)

Changement de mode d'exécution
Appels système

- Remarques
  - Mécanisme de protection au niveau mémoire indispensable
    - Ne sert a rien de forcer l'utilisateur à passer par un appel système s'il peut modifier le noyau, …
  - Le traitement d'une interruption impose également le passage en mode noyau

Changement de mode d'exécution
Changements de contexte

- Contexte d'exécution du processeur
  - Informations accessibles par le processeur à un instant donné, stockées dans ses registres
    - Mot d'état (SR - FLAGS)
    - Compteur ordinal (PC)
    - Sommet de pile (SP), pointeur de frame (FP),
    - Registres généraux (entiers, flottants, etc) … en x86 rax, rbx, rcx, rdx, rsi, rdi et bien d'autres

Changement de mode d'exécution
Changements de contexte

- Pourquoi la commutation de contexte ?
  - Plusieurs processus pour un seul processeur, (et un seul jeu de registres)
  - Quand on suspend un processus, on veut le reprendre à son point de suspension
  - ⇒ Opération de commutation de contexte
    - Sauvegarde du contexte d'exécution courant
    - Mise en place d'un nouveau contexte d'exécution (restauration du contexte)
  - Remarques
    - La mémoire n'est pas sauvegardée
    - Chaque processus a l'illusion d'un processeur dédié

Changement de mode d'exécution Changements de contexte

- Où sauvegarder le contexte d'exécution ?
  - En mémoire
  - A un emplacement ou on peut le retrouver au redémarrage du processus
    - Pile d'exécution du processus (+ lien pour retrouver le sommet de pile)
    - Descripteur du processus (voir après)

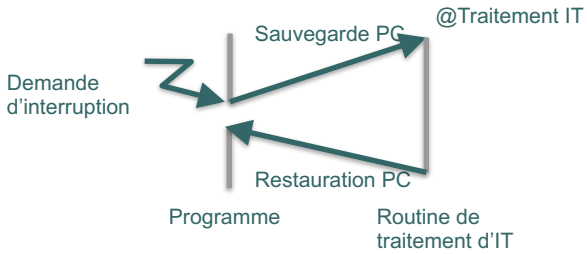
Mécanismes provoquant une commutation de contexte

- Exception (définition)
  - On nomme exception tout mécanisme provoquant une commutation de contexte, i. e.
    - Occasionnant l'exécution d'une séquence de code extérieure au programme
    - Avec retour possible ultérieur

Types d'exceptions : interruptions

- Cause externe au programme, non demandée explicitement
- Un niveau d'interruption (IT) peut être :
  - Armé/désarmé (activé/désactivé) : suppression de la source de l'interruption (signal d'interruption)
    - Obtenu par configuration du contrôleur de périphérique
  - Masqué/démasqué : configuration du processeur pour qu'il n'effectue pas de déroutement en cas de signal d'interruption
    - Stocké dans le mot d'état du processeur
    - Instructions spécifiques (instructions privilégiées cli/sti)

Types d'exceptions : interruptions



- Contexte minimal sauvegardé en cas d'interruption (PC,SR) par le matériel
- Une routine d'interruption n'a pas de contexte
  - Pas de blocage possible
  - Pas d'appel système bloquant autorisé

Types d'exceptions : exceptions matérielles

- Cause interne au programme, non demandée explicitement
- Détection d'une situation d'exécution anormale
- Exemples
  - Division par zéro, débordement
  - Accès à une zone mémoire invalide
  - Exécution d'une instruction interdite
- Routine de traitement associée (signal)
  - Remédie à l'anomalie si possible

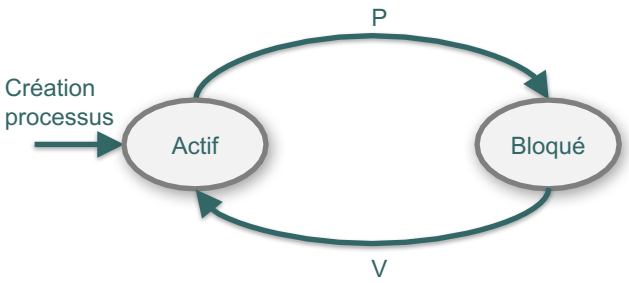
Types d'exceptions : appels système

- Cause interne au programme, demande explicite
  - Passage contrôlé en mode système, schéma d'exécution associé
    1. Sauvegarde du contexte
    2. Exécution de l'appel système
    3. Restauration du contexte
- Remarques
  - Quel contexte est restauré en 3 ? Dépend de l'appel système réalisé (bloquant, non bloquant)
  - Contexte restauré au retour d'une interruption ?
    - Processus interrompu ou autre processus ?
    - Dépendant de la mise en œuvre du noyau

Mise en œuvre du noyau de synchronisation
Représentation des processus

- Etats d'un processus
  - Etats logiques d'un processus (en faisant abstraction du partage du processeur entre processus)
    - Actif : peut logiquement s'exécuter et s'exécute car il dispose d'un processeur dédié pour s'exécuter
    - Bloqué : ne peut pas logiquement s'exécuter (attente liée à une synchronisation)

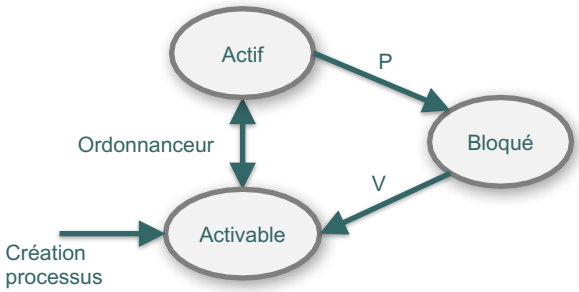
Mise en œuvre du noyau de synchronisation
Représentation des processus



Mise en œuvre du noyau de synchronisation
Représentation des processus

- Etats d'un processus
  - En supposant un seul processeur disponible
    - ⇒ Un seul processus actif à un instant donné
    - ⇒ Découpage de l'état actif en deux sous-états
      - Actif : peut logiquement s'exécuter et s'exécute car il dispose à cet instant du processeur réel
      - Activable (éligible) : peut logiquement s'exécuter mais ne s'exécute pas car ils ne dispose pas à cet instant du processeur réel
      - Bloqué : ne peut pas logiquement s'exécuter
    - NB : plusieurs actifs en multi-processeur

Mise en œuvre du noyau de synchronisation
Représentation des processus



- Transitions Actif – Activable : ordonnanceur

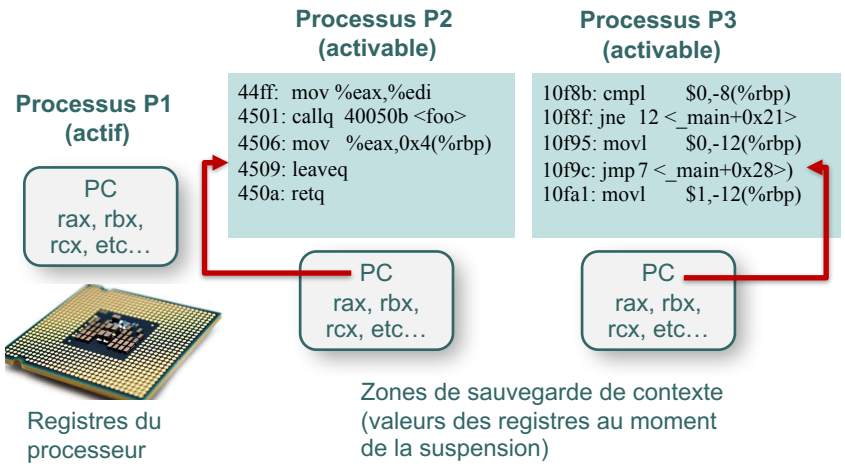
Mise en œuvre du noyau de synchronisation Représentation des processus

- Etats d'un processus affichés par la commande *ps* sous Mac OS X
  - I (Idle) : endormi depuis plus de 20 secondes
  - R (Ready) : actif
  - Re (Runnable) : activable
  - S (Sleeping) : endormi depuis moins de 20 secondes
  - U : dans une attente non interruptible
  - Z (Zombie) : terminé alors que son père ne l'est pas (ne consomme pas de ressource, juste un descripteur)

Contexte d'un processus

- Processus actif
  - Contexte d'exécution dans les registres du processeur, qui le fait évoluer
- Autre processus (activable, bloqué)
  - Contexte d'exécution sauvegardé en mémoire

Contexte d'un processus

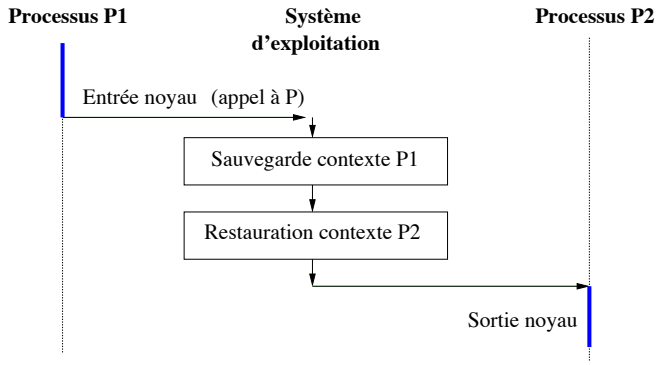


Commutation de contexte

- Association allocation/dé-allocation du processeur
  - Perte du processeur par un processus
    - Sauvegarde de son contexte d'exécution
    - Registres du processeur → zone de sauvegarde du contexte
  - Allocation du processeur au nouveau processus
    - Restauration du contexte sauvegardé au préalable
    - Zone de sauvegarde du contexte → registres du processeur

Commutation de contexte

- Exemple : commutation de contexte lors d'un P bloquant



Commutation de contexte

- Remarques
  - C'est l'ordonnanceur du système d'exploitation qui sélectionne le processus à activer (ici, P2) parmi les processus activables
  - Contrairement à un appel de fonction, on ne ressort pas nécessairement par la fonction appelée
  - On peut ne sauvegarder le contexte que lorsque l'on sait que l'appel système est bloquant

Descripteur de processus

- Descripteur par processus (PCB, Process Control Block)
  - Caractéristiques propres du processus (algorithme d'ordonnancement, priorité...)
  - Caractéristiques décrivant son état d'exécution (état, temps CPU utilisé...)
  - Zone de sauvegarde de son contexte (ou son adresse)
- Descripteurs chaînés entre eux pour regrouper les processus de caractéristiques identiques
  - File des processus activables,
  - File des processus bloqués sur un sémaphore

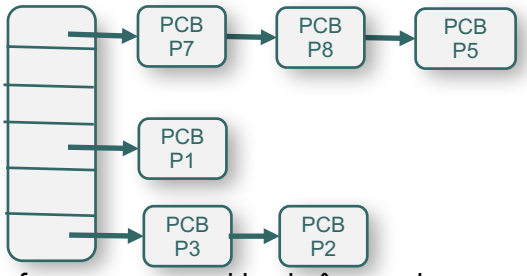
Descripteur de processus

- Remarques
  - La structure de données dépendra de l'ordonnancement mis en œuvre
  - Dans les systèmes à processus ``lourds'', le descripteur du processus contient en plus le contexte mémoire du processus (pointeur sur la table des pages)

Descripteur de processus

- Round-robin sans priorités

- Priorités fixes



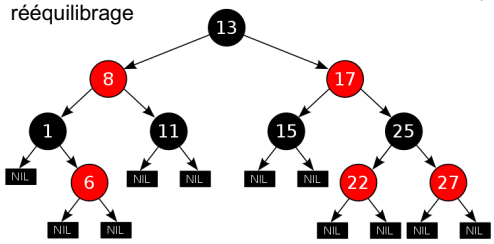
- Performances : seul le chaînage change

Descripteur de processus

- Etude de cas : Completely Fair Scheduler Linux
  - Objectif : équité du temps alloué aux processus
  - Moyen : tri des processus selon une valeur représentative du *manque* de ces processus en temps d'allocation du processeur, par rapport au temps qu'aurait alloué un processeur dit *multitâche idéal* (équité stricte)
  - Structure de données utilisée : arbre rouge-noir

Descripteur de processus

- Arbres rouge-noir
  - Nœuds rouge ou noirs racine noire
  - Quand nœud rouge, fils noirs. Feuilles (nœuds NIL) noires
  - Arbre binaire : max 2 fils, valeur fils gauche ≤ valeur nœud ≤ valeur fils droit
  - Pseudo équilibré : pire profondeur ≤ 2 * plus petite profondeur
  - Efficace dans le pire cas en insertion, recherche et destruction
    - Arbres binaires « de base » très mauvais dans le pire cas – rééquilibrage



Mise en œuvre de l'exclusion mutuelle

- Variables partagées entre plusieurs processus ⇒ manipulation en exclusion mutuelle
  - En mode utilisateur mais aussi noyau (files de descripteurs de processus, etc.)
- Besoin d'un mécanisme de bas niveau pour en complément des sémaphores
  - Problème de récursivité : on ne peut pas rendre indivisible les opérations sur les sémaphores en utilisant les sémaphores eux-mêmes
  - Sémaphore trop lourd pour être utilisé dans un noyau, ou les sections critiques sont souvent nombreuses et de courte durée

Mise en œuvre de l'exclusion mutuelle
a. Masquage des interruptions

- Permet au processeur d'ignorer (temporairement) les interruptions pendant une période
- Pas d'interruption ⇒ pas de déroutement de l'exécution sur le processeur ⇒ le processus garde le processeur pour lui seul
- Code

```
...
masquer_IT;          (1)
section critique;    (2)
démasquer_IT;        (3)
...                  (4)
```


Mise en œuvre de l'exclusion mutuelle
a. Masquage des interruptions

- Solution très rapide ⇒ intéressante à un niveau très bas dans le système
- Non généralisable
  - Aux sections critiques longues : risque de perte d'interruptions
  - Aux processus utilisateur : problème de sécurité (oubli de démasquer ou démasquage tardif ⇒ risque de compromettre des E/S en cours pour tout le système) - impossible : cli/sti sont priviliégiées
  - Aux systèmes multi-processeur : le masquage ne concerne que le processeur sur lequel l'instruction de masquage est exécutée

Mise en œuvre de l'exclusion mutuelle
a. Solutions algorithmiques

- Utilisation de variables partagées entre processus
  - Basées sur la propriété d'exclusivité d'accès à un emplacement mémoire
  - Séquence d'entrée en section critique : attente active (attente des conditions lui permettant d'entrer en section critique, tout en conservant le processeur)
  - Aussi nommé spinning (spin-locks)

Mise en œuvre de l'exclusion mutuelle
a. Solutions algorithmiques

- Solution algorithmique triviale

```
char occup=0; // Indique si un processus est en SC
while (occup) ; // Attente active
occup=1;
<section critique>;
occup=0;
```

- Ne fonctionne pas
  - Si deux processus essaient d'entrer simultanément en section critique, les deux voient *occup* à faux et entrent en section critique
  - Le problème vient de l'absence d'atomicité de la séquence test-modification de *occup*

Mise en œuvre de l'exclusion mutuelle
a. Solutions algorithmiques

- Algorithme de Peterson
  - Solution simple fonctionnant dans le cas de deux processus
  - Correction basée sur la propriété d'exclusivité d'accès à un emplacement mémoire

Mise en œuvre de l'exclusion mutuelle
a. Solutions algorithmiques

- Algorithme de Peterson

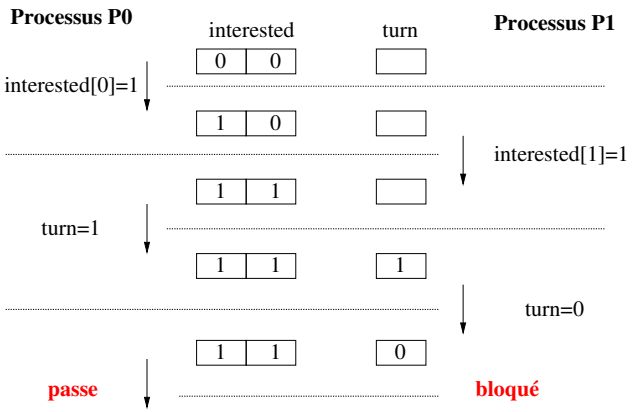
```
int turn;
char interested[2] := {0, 0};
void entrer_sc (int p); // p = nº processus demandeur {
      interested[p]=1;          (1)
      turn=1-p;                 (2)
      while ((turn==1-p) && (interested[1-p])); (3)
}
void sortie_sc  (int p); //p = nº processus demandeur{
      interested[p]=0;
}
```

Mise en œuvre de l'exclusion mutuelle
a. Solutions algorithmiques

- Algorithme de Peterson
  - La variable *interested*
    - Sert à enregistrer toutes les demandes d'accès (interested[i] uniquement manipulée par le processus i) Indique que le processus i est en section critique
    - Un élément interested[i] par processus ⇒ pas de risque d'écrasement de la demande par l'autre processus
    - La variable turn sert à gérer l'entrée simultanée en section critique des deux processus (départager les ex-aequo)

Mise en œuvre de l'exclusion mutuelle
a. Solutions algorithmiques

- Algorithme de Peterson : exemple



Mise en œuvre de l'exclusion mutuelle a. Solutions algorithmiques

- Il existe d'autres variantes basées sur le même principe
- Généralisables à N processus, si N connu

- Limites
  - Difficiles à utiliser à un niveau bas du système (N inconnu, si on borne, problème de consommation mémoire)
  - Attente active acceptable que si elle est de durée limitée (monopolisation du processeur)

Mise en œuvre de l'exclusion mutuelle a. Solutions basées sur un support matériel

- Solution algorithmique triviale

```
char occup=0; // Indique si un processus est en SC
while (occup) ; // Attente active
occup=1;
<section critique>;
occup=0;
```

- Ne fonctionne pas
  - Parce que la séquence test-modification de *occup* n'est pas indivisible
  - ⇒ Instructions dédiées résolvant ce problème (read-modify-write - RMW)

Mise en œuvre de l'exclusion mutuelle a. Solutions basées sur un support matériel

- Instructions dédiées à la synchronisation : assurent de manière indivisible
  - Une consultation (test)
  - Une modification (set) d'un emplacement mémoire
  - Un accès exclusif au bus
- Exemples :
  - TAS (68k) (Test and Set),
  - XCHG (x86)
  - Fetch-and-add

Mise en œuvre de l'exclusion mutuelle a. Solutions basées sur un support matériel

- Instruction Test-And-Set
  - Syntaxe : TAS adresse
  - Fonctionnement :
    - Positionne le registre de *code condition* selon l'ancienne valeur de MEM[adresse]
    - MEM[adresse] = 1;
  - Code d'entrée en section critique :

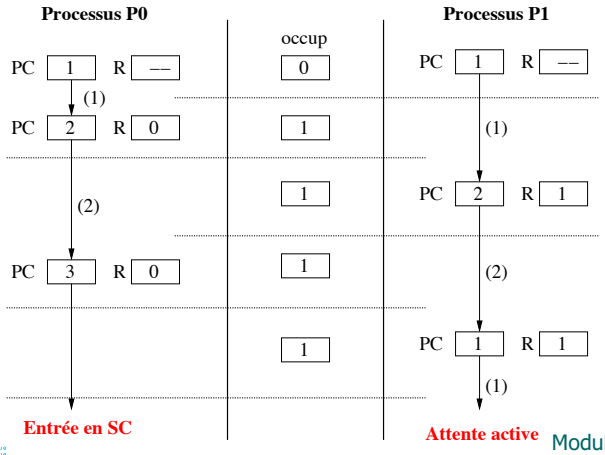
```
test:  tas occup;        (1)
if cc==1 goto test;      (2)
```

  - Code de sortie de section critique

```
occup= 0;
```

Mise en œuvre de l'exclusion mutuelle a. Solutions basées sur un support matériel

- Instruction Test-And-Set : exemple



Mise en œuvre de l'exclusion mutuelle

- Bilan des solutions de synchronisation utilisées
  - Solutions par attente active : consommation de temps processeur lors de la phase d'attente ⇒ mauvaise utilisation du processeur si l'attente est trop longue
    ⇒ Outils de synchronisation tels que les sémaphores, pour lesquels on relâche le processeur pendant les phases d'attente : attente passive
  - Solutions de bas niveau (IT, solutions par attente active) utilisées pour assurer la propriété d'atomicité des primitives P et V sur les sémaphores ou en multiprocesseurs

Sémaphore - rappels

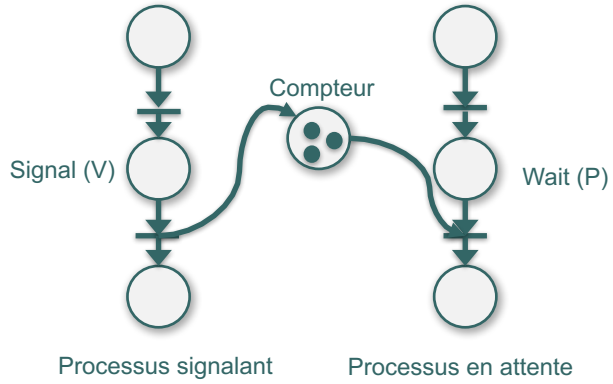
- Introduit par Edsger Dijkstra en 1965
- Sémaphore à compteur
- Un sémaphore *s* est un composé :
  - D'un compteur *s.e*
    - Valeur initiale du compteur ≥ 0
  - Primitives d'accès atomiques
    - P (ou Wait)
    - V (ou Signal)
    - P et V viennent de termes Hollandais (nationalité de Dijkstra, P =Proberen = test, V=Verhogen = increment)
    - P = Puis-je ?, V = Vas-y !

Sémaphore - rappels

Description des primitives d'accès P et V

```
typedef struct {int e} sémaphore;
void P (sémaphore *s) {
    when (s → e>0)
        s → e = s → e-1; //Tant que pas vrai, on attend
}
void V (sémaphore *s) {
    s → e = s → e+1;
}
```

Sémaphore : représentation graphique par des réseaux de Petri



Mise en œuvre des sémaphores

- Objectif
  - Ne pas monopoliser le processeur par un processus ne pouvant plus progresser (bloqué)
- Principe
  - Chaînage du descripteur de processus dans une file d'attente associée au sémaphore (blocage = mise en file)
- Représentation d'un sémaphore
  - Entier associé au sémaphore (compteur)
  - File d'attente : liste des descripteurs de processus bloqués sur le sémaphore

Mise en œuvre des sémaphores

```
PCB *active ; // processus actif
file PCB *ready_list ; // processus prêts (supposé FIFO ici)

void P (semaphore s) {
        if (s.e == 0) {
                enqueue(active, s.f); // mise en attente
        }
        else s.e = s.e-1;
}
```


Mise en œuvre des sémaphores

```
void V (semaphore s) {
        PCB *pr ;
        if   (s.f non empty) {
                pr=dequeue(s.f);
                enqueue(pr,ready_list);
        }
        else s.e = s.e+1;
}
```

Mise en œuvre des sémaphores

- Remarques
  - Sans plus d'informations sur les fonctions *enqueue* et *dequeue*, on ne sait pas quel processus est réveillé lors d'un V (dépend du mode de gestion de file)
  - Le code ne précise pas comment les routines P et V sont rendues indivisibles
  - Le code ne montre pas les commutations de contexte
  - Le code est conforme à la spécification donnée au premier semestre (on n'a jamais s.e < 0)

Mise en œuvre alternative des sémaphores

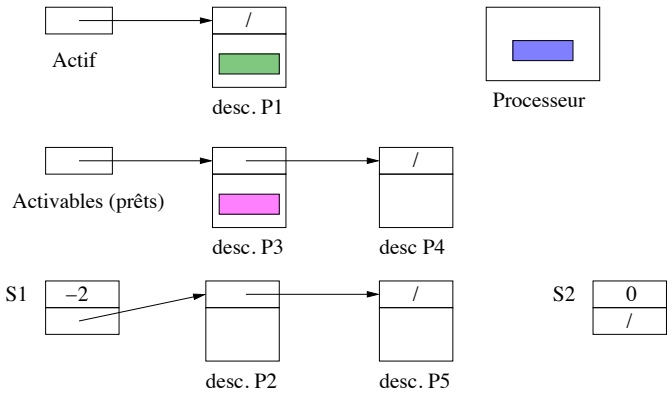
```
void P (semaphore s) {
        s.e = s.e-1;
        if (s.e < 0) {
                enqueue(active, s.f); // mise en attente
        }
}
```

Mise en œuvre alternative des sémaphores

- Exemple
  - Cinq processus P1, P2, P3, P4, P5
  - P1 est actif
  - P3 et P4 sont activables
  - P2 et P5 sont bloqués sur le sémaphore S1
  - Le sémaphore S2 est initialisé à 0

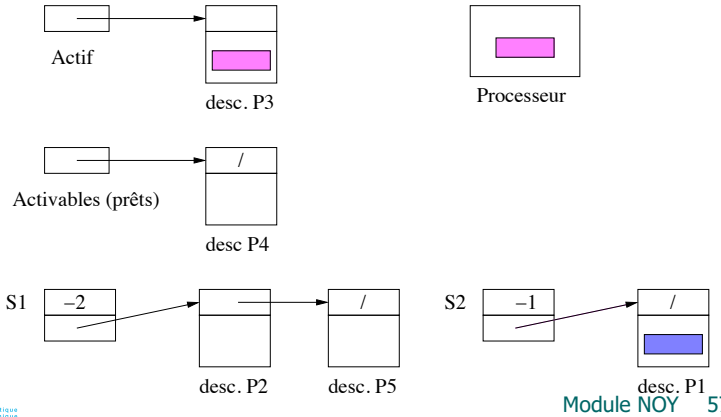
Mise en œuvre alternative des sémaphores

- Exemple : (processus actif en dehors de readylist)



Mise en œuvre alternative des sémaphores

- Exemple : après exécution par P1 de P(s2) et attribution du processeur à P3



Spinning or blocking ?

- Attente passive (sémaphores)
  - Pas de monopolisation du processeur pendant la phase d'attente
  - Mais, changement de contexte
- Attente active (spin-locks)
  - Monopolisation du processeur
  - Mais, pas changement de contexte
- Spin-locks peuvent être plus efficaces si on sait que la durée d'attente est très courte, et en multi-cœurs

Particularités des architectures multi-cœurs

- Atomicité des sémaphores
  - Ne peut pas être totalement assurée par un masquage d'interruptions (local à un processeur)
  - Utilisation d'attente active
- Principe de réalisation
  - Attente active (spin-lock) pour assurer l'atomicité des P/V
  - Tolérable car durée "d'attente" courte (durée d'exécution du P/V hors blocage)