8 - MEMOIRE VIRTUELLE : EFFICACITE

1. CACHE ET MEMOIRE VIRTUELLE

Avec une mémoire paginée, l'accès à une donnée nécessite deux accès à la mémoire : un pour lire la table des pages, l'autre pour lire la donnée. Pour ne pas diviser par deux les performances du système, on utilise un cache spécial, appelé TLB (Translation Look-aside Buffers), qui permet de stocker le couple <n° de page, case mémoire>. Si le couple correspondant à la page accédée est présent dans la TLB, alors l'accès à la table des pages devient inutile. Typiquement, la TLB contient entre 8 et 2048 entrées.

1.1.

On suppose qu'un accès à la TLB demande 20 ns et qu'un accès mémoire demande 100 ns. Pour une TLB avec un taux de hit de 90%, quel est le temps moyen d'accès à une donnée ?

Temps d'accès hit : 20 ns (accès TLB) + 100 ns (accès mémoire) = 120 ns

Temps d'accès miss : 20 ns + 100 ns (accès table des pages) + 100 ns = 220 ns

Temps moyen d'accès : 0,9 x 120 + 0,1 x 220 = 130 ns.

En plus de la TLB qui permet une translation rapide de l'adresse logique à l'adresse physique, le système dispose d'un cache pour améliorer la vitesse d'accès aux données. Le cache est une mémoire à accès rapide, structurée en lignes. Chaque ligne stocke un bloc de données d'une part, tout ou partie de l'adresse du bloc permettant d'identifier le bloc de façon unique, ainsi que diverses informations relatives à la validité, l'ancienneté, ... dudit bloc.

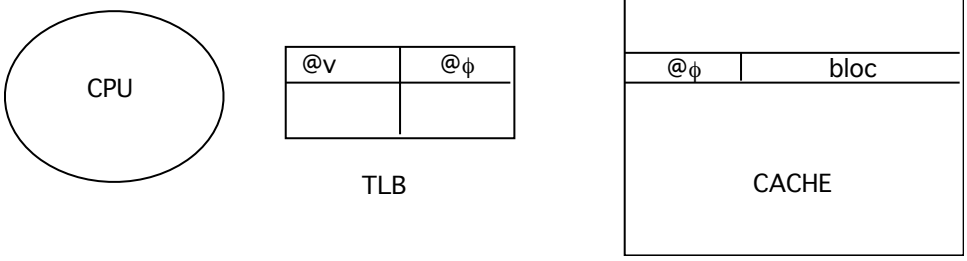
La combinaison de la mémoire cache et de la mémoire virtuelle est délicate : ou bien le cache contient les adresses physiques des blocs, et dans ces conditions la traduction adresse virtuelle - adresse physique doit être faite entre le processeur et le cache; ou bien le cache contient les adresses virtuelles des blocs, et la traduction adresse virtuelle-adresse physique a lieu entre le cache et la mémoire.

1.2.

Représentez schématiquement l'organisation de la hiérarchie mémoire dans les deux cas. Quels sont les avantages et inconvénients de chaque organisation ?

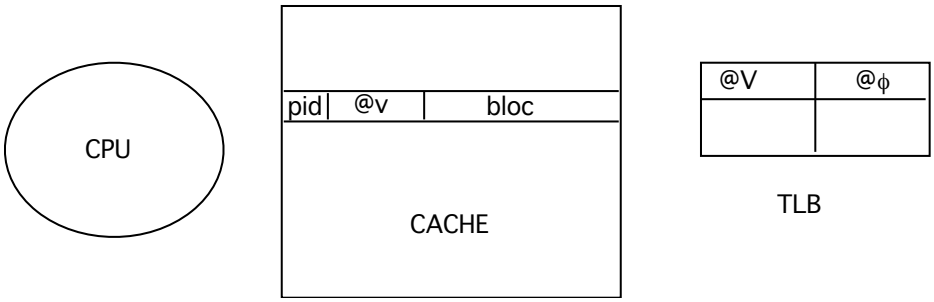
NB : De la même manière que la table des pages est changée à chaque commutation, la TLB doit être effacée à chaque commutation.

cache adressé physiquement (ex : Pentium) : Tout accès à une donnée passe nécessairement par la TLB. Il faut donc que la TLB soit très rapide pour ne pas trop augmenter le temps de hit du cache. Par contre, le cache est simple.



cache adressé virtuellement : la TLB étant placée entre le cache et la mémoire, elle est moins soumise à des impératifs de rapidité. Par contre, le cache doit être capable de distinguer deux blocs de même adresse virtuelle appartenant à des processus différents, ce qui implique l'adjonction au tag d'un identificateur de processus propriétaire d'un bloc. De même, il se peut que le système et un processus référencent le même bloc avec des adresses virtuelles différentes (c'est le phénomène d'alias multiples), ce qui peut conduire à la présence d'un même bloc dans différentes lignes de cache, avec tous les problèmes de cohérence qui s'ensuivent. Les caches adressés virtuellement imposent au préalable une résolution d'alias (au niveau système), imposant un alias unique pour chaque bloc, afin d'assurer qu'à un instant donné, un bloc est présent dans une seule ligne du cache.

ex : MIPS R4000 et HP PA_RISC



2. PARTAGE D'INFORMATION EN MEMOIRE

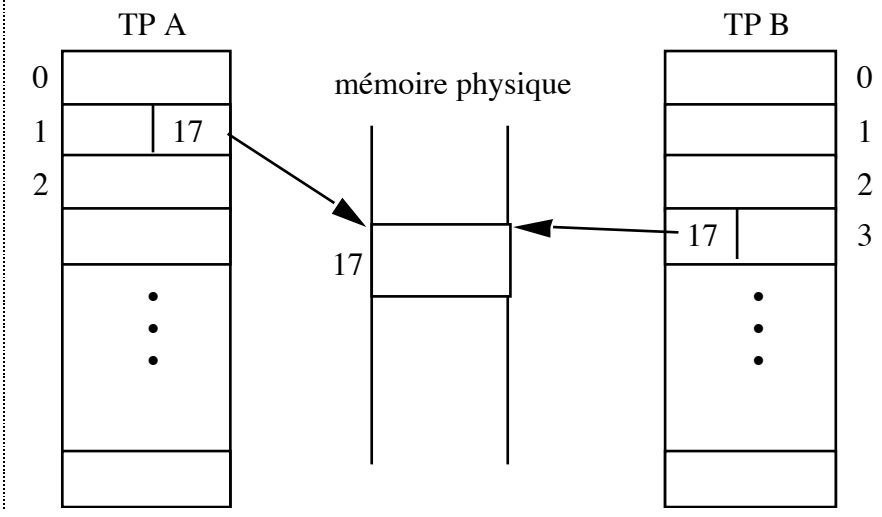
Le partage d'information entre plusieurs processus soulève deux principaux problèmes :

- le partage physique : comment faire en sorte que les données partagées existent en un exemplaire unique ?
- la protection : comment garantir la cohérence des accès aux informations partagées ?

2.1.

Représentez la topologie des tables de pages de deux processus, A et B, partageant une donnée de la case 17 située aux adresses virtuelles <1,155> pour A et <3,155> pour B.

Chaque processus possède une table des pages. La topologie est alors la suivante :



Soient 2 processus A et B exécutant le même programme. Le segment 0 correspond au code, le segment 1 à la pile et le segment 2 aux données. Seules les pages suivantes ont été chargées (on représente le doublet <n° segment, n° page>) :

pour le processus A : les pages <0,0> dans la case 10, <1,3> dans la case 20 et <2,1> dans la case 30.

pour le processus B : les pages <0,1> dans la case 40 et <1,4> dans la case 50.

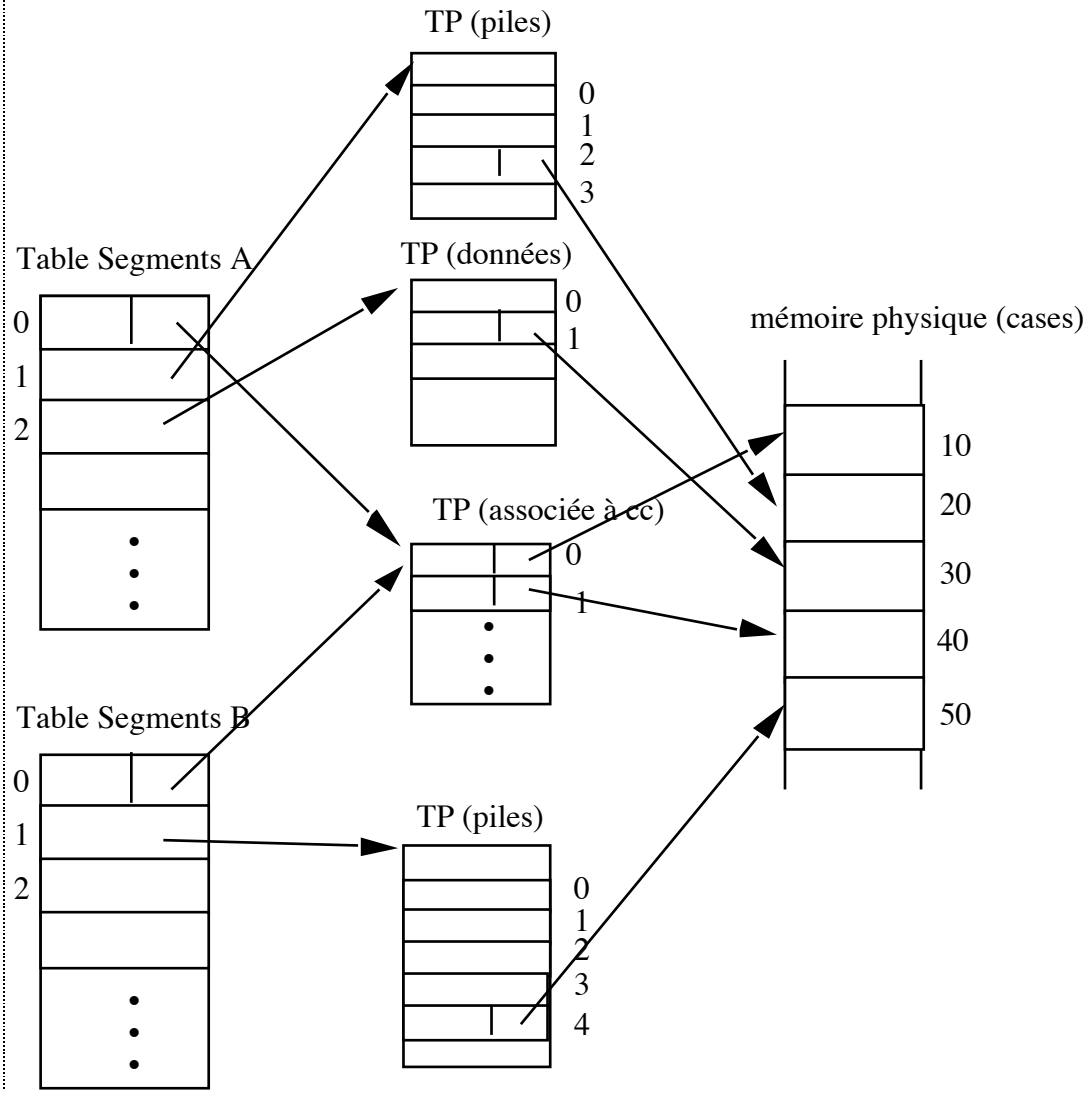
2.2.

Quels segments peuvent être partagés entre ces 2 processus ?

Le programme exexécuté par A et B est identique (c'est le même code) : le segment de code peut être partagé. Les 2 autres (pile et données) sont spécifiques à chaque exécution.

2.3.

Faites un schéma des structures de données (tables des segments et tables des pages) pour ces deux processus.



2.4.

Quels nouveaux problèmes sont introduits par le partage d'une case ?

Le partage introduit de nombreux problèmes :

la gestion des droits d'accès est plus complexe car une page partagée peut recevoir des droits d'accès différents pour chaque processus,

le chargement/déchargement d'une page exige de mettre à jour toutes les tables de processus partageant la case.

enfin, un dernier problème que nous détaillerons pas dans ce TD, est la désignation : comment adresser de manière uniforme les informations partagées. Dans le cas d'une mémoire segmentée, les informations sont désignées par le nom du segment.

2.5.

Quel est l'influence du partage sur les politiques de remplacement de pages ?

La politique de remplacement de pages est plus complexe, car intervient le comportement mémoire de tous les processus partageant le page. Par exemple, pour une politique LRU il faudrait choisir la page la plus anciennement utilisée par tous les processus.

3. GESTION OPTIMISE D'UN FORK

Dans UNIX, l'appel système "fork" permet de créer un processus (le fils) à l'image de l'appelant (le père). C'est-à-dire que les segments de code, de pile et de données du processus père sont **copiés** dans l'espace d'adressage du processus fils. Chaque processus (père et fils) évolue ensuite de façon indépendante.

3.1.

Comment peut-on utiliser les mécanismes de partage pour réduire le coût du fork (le coût de la copie) ?

Dans un système à pagination (Système 5), le noyau duplique ***uniquement*** chez le processus fils les tables des pages des différents segments. Une nouvelle page est allouée ***uniquement*** lorsque le fils la modifie. De cette manière on ne recopie que ce qui est nécessaire.

3.2.

Quelles sont les vérifications nécessaires avant l'accès à une page ? Proposez une solution simple pour gérer les conflits d'accès lorsqu'une page est partagée en lecture/écriture par plusieurs processus.

Il faut bien sûr que la page soit préalablement chargée en mémoire. Ensuite il faut vérifier si le processus possède les droits nécessaires pour réaliser l'opération. Viennent ensuite les verifications propres au partage. Si le processus demande une opération d'écriture sur une page partagée la page risque d'être modifiée pour les autres processus.

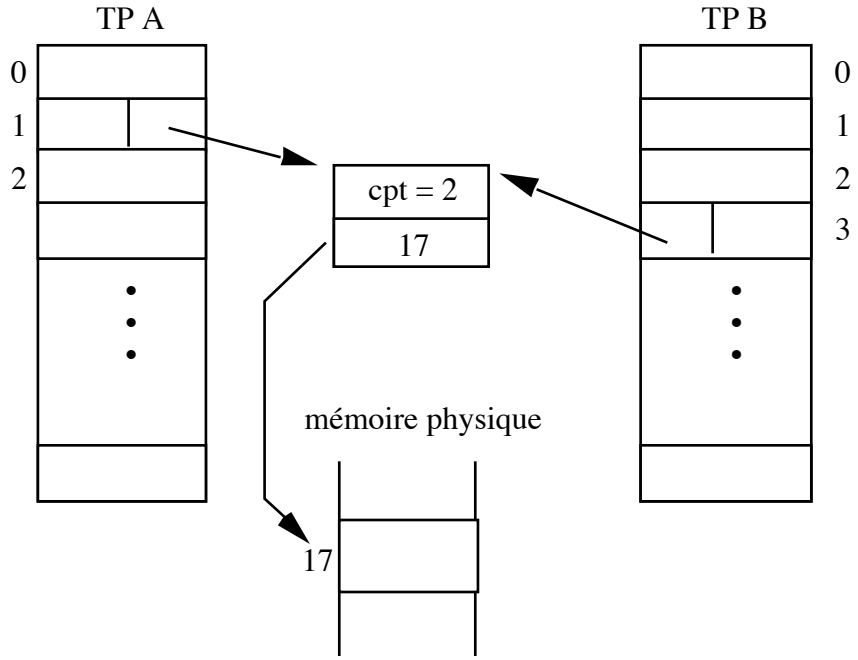
Les protocoles classiques de synchronisation sont trop complexes pour résoudre ce type de conflits. De manière générale les systèmes (par exemple UNIX système 5) utilisent une technique de copie-sur-écriture (copy-on-write). En cas d'accès en écriture, la page est recopiée dans une nouvelle page. De cette manière, les autres processus ignorent la modification de la page. Ce mécanisme est utilisé dans la réalisation d'un fork dans UNIX.

3.3.

On souhaite gérer la mémoire virtuelle pour pouvoir partager les pages tant qu'elles ne sont pas modifiées. Quelles structures de données faut-il ajouter pour savoir si une page doit être recopiée ?

Il faut introduire un **compteur de références** qui compte le nombre de processus partageant une page. En effet, il est inutile de copier une page sur écriture si la page n'est plus partagée par d'autres processus.

Les entrées dans la table des pages ne désignent plus désormais directement la case physique mais une structure intermédiaire contenant le compteur de références. Dans la terminologie UNIX, une telle structure est appelée un *cadre de page*. Le schéma suivant reprend le premier schéma de la question 1.1.



Une faute de protection signifie que le processus a accédé à une page en mémoire (valide) mais que les bits associés à la page ont permis de détecter un problème à traiter. La partie matérielle fournit à la procédure l'adresse virtuelle où la faute s'est produite.

Une faute de protection signifie que le processus a accédé à une page en mémoire (valide) mais que les bits associés à la page ont permis de détecter un problème à traiter. La partie matérielle fournit à la procédure l'adresse virtuelle où la faute s'est produite.

3.4.

Indiquez brièvement l'algorithme de la procédure de traitement des fautes de protection.

Lorsque le bit copie-sur-écriture est positionné, il y a systématiquement interruption lors d'un accès en écriture à la page. En effet, le système ne peut pas savoir a priori s'il doit ou non dupliquer la page. Deux cas en effet peuvent se produire :

- la page est effectivement partagée : le système doit la recopier

- la page était partagée, mais les autres processus ont déjà provoqué une recopie. Il n'y a donc plus qu'un processus (le processus qui a provoqué l'interruption) qui accède à la page), mais sa table des pages (le bit copie-sur-écriture) n'a pu être mise à jour lors des recopies provoquées par les autres processus : ceux-ci ne connaissent pas toujours l'identité des processus avec qui ils partagent une page et de plus, seule la table des pages du processus actif courant est accessible à un instant donné.

Dans tous les cas, après cet accès, le processus est le seul à accéder à la page. Il faut donc effacer le bit copie-sur-écriture pour ne pas provoquer d'interruption lors du prochain accès en écriture.

Lorsque le bit copie-sur-écriture n'est pas positionné, l'interruption a été causée par les bits de droit d'accès : accès en écriture d'une page en lecture, etc…

Voici l'algorithme simplifié inspiré d'Unix Système 5 [Bach 86]

```
protec_faut
entrée: adresse virtuelle de la page
{
     si (page valide)
          si (bit "copie-sur-écriture" non positionné) {
               /* erreur dans le programme */
          }
          sinon {
               si (compteur de références > 1){
               /* réaliser la copie de la page */
                    attribuer une nouvelle case physique;
                    copier le contenu de l'ancienne page dans la
nouvelle;
                    décrémenter le compteur de référence de
l'ancienne page;
                    màj la table de pages du processus courant
pour
                    désigner la nouvelle case;
}
               sinon {
                         /* recupérer directement la page
                         puisque personne ne l'utilise */
               }
               positionner le bit de modification pour la nouvelle
page;
               effacer le bit de copie sur écriture;
          }
     }
}
```