istic Informatique Électronique

Année scolaire 2017-2018

Isabelle Puaut

Systèmes d'exploitation, fondements et programmation système sous Linux (SEL)

Travaux dirigés

Petits exercices sur les sémaphores

Objectifs

L'objectif de ce TD est d'acquérir une maîtrise de l'utilisation des sémaphores à compteurs via la programmation de problèmes de synchronisation très simples, mais néanmoins très courants.

Questions

1. Deux programmes P1 et P2 se partagent deux variables a et b. Ecrire le code d'entrée et sortie de section critique permettant de protéger ces deux variables. Quel est l'effet d'un P bloquant à l'intérieur d'une des sections critiques ?

P1:	P2:
if (a >0) a = a-1;	b = a+4;
b = b*2;	

2. Soient deux processus P1 et P2 dont le code est le suivant :

P1	:	…….	A1	…….
P2	:	…….	A2	…….

   Ecrire le code de P1 et de P2 pour que P2 attende, au point A2, que P1 soit au point A1, et uniquement quand c'est nécessaire.
3. En reprenant l'exemple précédent, écrire le code des processus P1 et P2 pour établir un *rendez-vous* entre P1 et P2 juste après passage aux points A1 et A2 (attente de l'autre processus quand c'est nécessaire).
4. Ecrire le code pour maintenant réaliser un rendez-vous entre trois processus P1, P2 et P3. Généraliser la solution pour un nombre *n* de processus en utilisant un tableau de *n* sémaphores.
5. Donner une solution à la question précédente qui minimise le nombre de sémaphores utilisés, par exemple en utilisant un compteur de processus arrivés au rendez-vous.
6. Soit un parking à N places, initialement vide. Chaque voiture désirant entrer dans le parking est représentée par un processus. Ecrire le code de synchronisation bloquant les voitures quand le parking est plein :
   - en n'utilisant pas d'attente active
   - en utilisant une variable partagée comptant le nombre de places libres dans le parking

Processus voiture :

<entrer dans le parking, avec attente d'une place libre>

<rester dans le parking (durée quelconque inconnue a priori)>

<sortir du parking>

Est-il possible de réaliser ce problème de synchronisation sans variable partagée, en utilisant exclusivement des sémaphores ? Si oui, donner le code correspondant.

Application des sémaphores à deux problèmes de type général

Objectifs

L'objectif de ce TD est d'acquérir la maîtrise de deux problèmes de synchronisation très courants : producteur/consommateur et lecteur/rédacteur.

Producteur-consommateur

Deux processus (le producteur et le consommateur) se partagent un tableau de N éléments. Le producteur produit de nouveaux éléments dans le tableau, que le consommateur est chargé de consommer. Le producteur produit les éléments un par un. Le consommateur consomme les éléments également un par un. L'échange d'informations entre producteur et consommateur suit le schéma suivant, dans lequel les ??? représentent du code de synchronisation :

```
Producteur :
while (1) {
    < bâtir une information >
    ???
    < la communiquer au consommateur >
    ???
}
```

```
Consommateur :
while (1) {
        ???
        < recevoir une information >
        ???
        < exploiter l'information >
}
```

Le producteur produit les informations dans un tableau, géré de manière circulaire. Le producteur utilise un indice de production nommé *iplein* (indice de la dernière case remplie), tandis que le consommateur utilise un indice de consommation nommé *ivide* (indice de la dernière case vidée).

1. Compléter le code des processus producteur et consommateur (gestion des indices et synchronisations à l'aide de sémaphores).
2. Si l'on voulait avoir plusieurs producteurs et plusieurs consommateurs, quelles modifications faudrait-il apporter au schéma précédent ? On supposera que les producteurs produisent pour n'importe quel consommateur.
3. On revient à un seul producteur et un seul consommateur. Si le producteur est actif, il continue tant qu'il y a une case vide ; si le producteur est bloqué, il n'est réveillé que s'il y a X cases vides (X > 1). Ecrire les nouvelles synchronisations pour le producteur et le consommateur.

Lecteur-rédacteur

Une donnée (par exemple un fichier) est partagée par deux types de processus :

- Les processus qui la consultent (processus *lecteurs*)
- Les processus qui la modifient (processus *rédacteurs*). Les lectures et les écritures n'utilisent pas de temps processeur, donc en l'absence de synchronisation s'exécutent en parallèle.

On veut assurer un maximum de parallélisme entre les différents processus tout en assurant que les informations lues aient un sens et que les modifications se fassent correctement. Pour que les informations lues aient un sens, on ne veut pas avoir un lecteur et un écrivain en même temps, ou deux écrivains simultanément sur la même donnée.

4. Donner le code des processus lecteur et rédacteur basé sur le principe suivant :

- Un processus rédacteur se bloque s'il existe un processus lecteur ou écrivain en cours d'exécution
- Un processus lecteur se bloque uniquement s'il existe un processus écrivain en cours d'exécution

5. Que se passe t'il si on a les arrivées suivantes ? Dessiner le chronogramme correspondant au code écrit dans la question précédente, en supposant que les lectures peuvent s'effectuer en parallèle car elles n'utilisent pas le processeur (ex : lecture sur un périphérique).

Date d'arrivée	Nature	Durée
1	Lecteur 1	3
2	Rédacteur 2	2
3	Lecteur 2	3
5	Lecteur 3	3

6. Reprendre la première question en mettant une « *priorité* » égale entre les processus lecteurs et rédacteurs. Dès qu'un processus rédacteur arrive, tous les processus attendent, quelle que soit leur catégorie.

Décomposition en processus

Objectif

L'objectif de cet exercice est de mettre en application votre connaissance de la synchronisation par sémaphores (producteur-consommateur) sur un exemple complet, tout en restant simple. L'exercice permettra aussi de se convaincre des gains en performances que l'on peut obtenir en décomposant un système en processus. Le système utilisé est volontairement générique (débarrassé de toute considération de bas niveau).

Le système considéré

Dans le système on peut distinguer trois parties : entrée, calcul, sortie. Ces trois parties échangent de l'information à travers deux types de tampons TE et TS. Les tampons de même nature (TE, TS) sont chainés circulairement.

- *Entrée(TE)* symbolise le remplissage du tampon TE. Cette partie dure 20 ms et se décompose en une phase d'initialisation d'1 ms, une phase d'attente sur le périphérique d'entrée et une phase terminale qui utilise également 1 ms de temps processeur. L'attente sur le périphérique d'entrée est non active (i.e. le processus en attente est bloqué sur un sémaphore pendant la durée de la lecture).
- *Calcul(TE,TS)* utilise les informations contenues dans TE pour remplir le tampon TS. Cette opération dure 21 ms et ne consomme que du temps processeur.
- *Sortie(TS)* symbolise le transfert sur le périphérique de sortie du contenu du tampon TS. Cette partie dure 25 ms et se décompose en une phase d'initialisation de 2 ms, une phase d'attente (non active) de 22 ms et une phase terminale d'1 ms. Comme pour l'entrée, le processus qui attend la fin de Sortie est bloqué sur un sémaphore pendant la durée de l'écriture.

On regroupe ces 3 parties dans une boucle :

```
while (1) {
        entrée(TE);
        calcul(TE,TS);
        sortie(TS);
}
```

Questions

1. Evaluer la durée d'un cycle, ainsi que le taux d'occupation du processeur.

Afin d'améliorer les performances, on envisage d'autres organisations. On suppose que les entrées et les sorties se font sur des périphériques indépendants et que l'ordonnanceur ne réalise la commutation de processus **que** sur l'appel à une primitive P bloquante (pas de changement de processus lors d'un V).

Les schémas temporels mettront bien en évidence à chaque instant le processus qui s'exécute sur le processeur.

On utilisera la notation x = suivant(x) pour effectuer un changement de tampon dans un ensemble x de tampons gérés circulairement.

2. On met la partie entrée dans un processus et on regroupe calcul et sortie dans un autre.
   - Combien de tampons TE et TS faudra-t-il déclarer pour avoir le meilleur débit ?
   - Donner le code des deux processus avec leur synchronisation par sémaphores.
   - Montrer sur un schéma temporel le fonctionnement de l'ensemble et indiquer la durée du cycle lorsque le régime permanent est atteint.

3. On décompose le travail en trois processus : *entrée*, *calcul*, *sortie*.
   - Quelle valeur de la durée du cycle espère-t-on ainsi atteindre ?
   - Donner le code des trois processus.
   - Montrer sur le début d'un schéma temporel que l'ordonnanceur va avoir à choisir entre plusieurs processus. Quelle règle de choix proposez-vous ?
   - Donner un début de schéma temporel.

Programmation des entrées/sorties caractère

Objectif

L'objectif de cet exercice est de programmer un pilote de périphérique caractère par attente active puis demande d'interruptions. Le contrôleur de périphérique est volontairement très simple pour se libérer des contraintes d'accès aux registres du contrôleur.

Spécification du matériel

Une ligne série est utilisée pour communiquer entre deux ordinateurs. Le matériel fonctionne en "full-duplex", ce qui signifie que l'envoi et la réception d'un caractère sont deux opérations indépendantes pouvant se dérouler en parallèle.

Le contrôleur d'entrées/sorties comprend quatre registres internes :

- Un registre d'entrée pour la réception d'un caractère
- Un registre de sortie pour l'émission d'un caractère
- Un registre d'état
- Un registre de contrôle (commande)

Chaque voie du contrôleur (*émission* ou *réception*) peut être gérée en mode "test d'état" (attente active), ou avec demande d'interruption.

On suppose disposer des primitives suivantes pour la programmation du contrôleur :

- *void lire(char *c)* : range dans *c* le contenu du registre d'entrée
- *void écrire(char c)* : range *c* dans le registre de sortie
- *char état_sortie* : rend vrai quand le registre de sortie est vide
- *char état_entrée* : rend vrai quand le registre d'entrée est plein
- *void autoriser_it_s()* : une interruption sur le niveau it_s est postée par le contrôleur dès que le registre de sortie *devient* vide. L'acquittement de cette interruption s'effectue en écrivant un caractère dans le registre de sortie
- *void autoriser_it_e()* : une interruption sur le niveau it_e est postée par le contrôleur dès que le registre d'entrée *devient* plein. L'acquittement de cette interruption s'effectue en lisant un caractère.

- *void interdire_it_e()* : le contrôleur ne poste pas d'interruption sur le niveau *it_e*
- *void interdire_it_s()* : le contrôleur ne poste pas d'interruption sur le niveau *it_s*
- *void initialiser()* : Initialise le contrôleur. Le contrôleur ne poste pas d'interruption sur les niveaux *it_e* et *it_s*.

Les niveaux d'interruption *it_e* et *it_s* sont distincts ; le niveau *it_e* est plus prioritaire que le niveau it_s.

FIXME : interface et code pas top pour débordement de buffer

Questions

Dans un premier temps, on suppose qu'un seul processus utilise la ligne série. On souhaite réaliser les trois primitives suivantes :

- void tty_init(void)
- *void tty_em_ligne (char *l);* : émet la ligne située dans le tableau l. Une ligne se termine par un zéro.
- *void tty_rec_ligne (int *lg ; char *l)* : reçoit une ligne de caractères terminée par 0 et retourne sa longueur dans lg.

1. Ecrire les trois opérations *tty_init, tty_em_ligne* et *tty_rec_ligne*, le contrôleur étant programmé en mode test d'état (attente active). L'émission et la réception sont des opérations synchrones.
2. Ecrire de nouveau les trois opérations en programmant maintenant le contrôleur avec demande d'interruption en entrée et en sortie. La voie de réception est gérée avec anticipation : les caractères reçus sont mémorisés dans un tampon de réception jusqu'à leur consommation (appel à *tty_rec_ligne*). Un tampon est également affecté à la voie émission. Dès que la ligne à émettre est recopiée dans le tampon émission, le processus suppose que l'entrée/sortie est terminée. Vous écrirez également les traitants d'interruption associés aux niveaux *it_e* et *it_s*.

Hiérarchie de mémoire

Objectifs

L'objectif de ce TD est de comprendre le fonctionnement de différentes architectures de caches (correspondance directe, associatifs par ensembles, totalement associatifs).

Description de l'architecture

On se place dans le cas d'une architecture dans laquelle les instructions sont codées sur 32 bits et utilisant des adresses sur 32 bits. L'architecture dispose de caches d'instructions et de données séparés. La taille des caches est de 4Ko et la taille des lignes de caches de de 32 octets. La politique de gestion du cache de données est *write-back/write-allocate.*

L'associativité du cache et la politique de remplacement de cache associée seront précisées dans la suite.

Programme de test

```
#define N 1024
long t1[N]={0,1,2,3,4, … , 1023} ;
long t2[N]={1,0,1,0,1, … , 0} ;
long ts[N]={0,} ;          // Initialisé à 0
long i ;                   // Sera supposé stocké dans un registre
// Sommation des deux tableaux t1 et t2, résultat dans ts
for (i=0;i<N;i++) ts[i] = t1[i] + t2[i];
```

Un entier long est représenté sur 4 octets. L'éditeur de liens attribue les adresses virtuelles suivantes aux sections :

- Code : 0x100000
- Données :0x200000 hexadécimal

Questions

On s'intéresse dans un premier temps au cache de données uniquement et on considère que le cache est à correspondance directe.

1. Quelle type de localité (spatiale/temporelle) observe t'on sur ce programme : pour les données et pour le code
2. Donner en hexadécimal les adresses des trois tableaux
3. Donner pour ce cache les tailles en bits des zones d'index, étiquette et offset. Donner les valeurs correspondantes pour le premier élément des trois tableaux.
4. Dessiner l'évolution du contenu du cache et en déduire le taux de succès associé

On suppose maintenant et jusqu'à la fin de l'exercice que le cache de données est associatif par ensemble, avec un degré d'associativité de 4 et une politique de remplacement LRU. Une architecture identique est utilisée pour le cache d'instructions.

5. Reprendre les deux questions précédentes et expliquer la différence de taux de succès s'il y en a une
6. D'après-vous, est-ce que le taux de succès dans le cache d'instructions est plus faible ou plus élevé que dans le cache de données ? A quelle propriété des programmes est-ce dû ?
7. Soit l'extrait de programme suivant, s'appliquant sur des tableaux à deux dimensions (matrices) d'entiers longs, avec N=1024

```
for (i=0 :i<N ;i++)
    for (j=0 ;j<N ;j++)
        M[i][j] = M[i][j] * 2 ;
```

Quelle représentation des tableaux (a) par ligne – chaque ligne du tableau est stockée en mémoire juste après la précédente (b) par colonne, donne les meilleures performances ? Expliquer pourquoi.

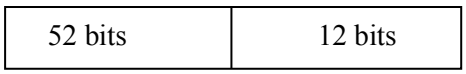
Principes de la traduction d'adresse

Objectifs

L'objectif de cet exercice est d'étudier un système de traduction d'adresses très simple pour mieux appréhender les notions d'adresses virtuelles et réelles et comprendre l'intérêt du mécanisme de traduction d'adresses.

Mécanisme de traduction d'adresses

On dispose d'une machine disposant d'adresses virtuelles (octets) sur 64 bits, décomposées de la manière suivante :



N° de page virtuelle déplacement

Les instructions du processeur sont de la forme INSTR destination, source. Elles sont systématiquement représentées sur 64 bits. La machine dispose des modes d'adressage suivants (la syntaxe est de notre invention) :

- Immédiat
  ex: MOV EAX,#10 place dans le registre 32-bits EAX la valeur 10
- Absolu
  ex: MOV EAX,10 place dans le registre EAX la valeur contenue à l'adresse (virtuelle) 10
- Indirect
  ex: MOV EAX,[10] place dans EAX le contenu de ce qui est à l'adresse contenue en 10.

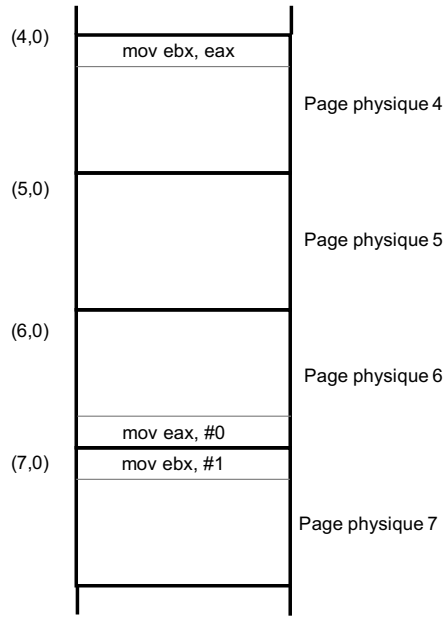
On notera PC le compteur ordinal de la machine, et on préfixera par 0x les valeurs exprimées en hexadécimal.

La traduction d'adresse utilisera une table des pages linéaire notée TPV, indexée par le numéro de page virtuelle de l'adresse virtuelle. Chaque entrée de la table, notée DPV, disposera des champs classiques V (validité, sur 1 bit) et pr (numéro de page physique).

Questions

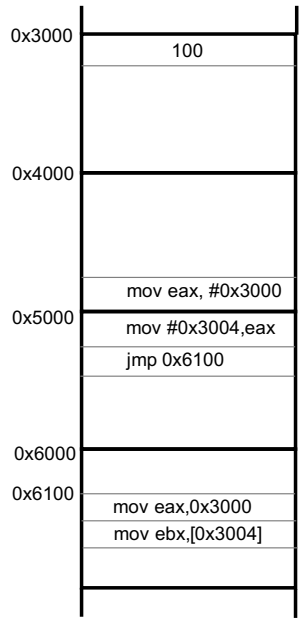
1. Quelle est la taille d'une page ?

On se place en cours d'exécution d'un programme, juste avant l'exécution de l'instruction de PC = 0x3FF8. On suppose que seuls TPV[3].V et TPV[4].V sont à 1, et que TPV[3].pr=6 et TPV[4].pr=4. La mémoire physique contient les informations suivantes :



2. Donner le contenu du registre ebx après l'exécution de deux instructions

On considère l'image mémoire suivante d'un programme, dessinée maintenant par adresse *virtuelle* croissante. TODO : changer dans schéma le mov #0x3004,eax (pas de # ici). Mélange 32 et 63 et endianness. Grrr ! Deuxième schéma : mettre page virtuelle en légende.



On lance l'exécution du programme avec PC=0x4FF8 et TPV[3,4,5,6] = { (1,0), (1,1), (1,2), (1,3) }.

3. Donner l'état de la mémoire physique après l'exécution de deux instructions.

On bloque le processus et on sauvegarde son contexte, puis on le recharge ultérieurement avec le contexte suivant : PC=valeur sauvegardée au préalable et TPV[3,4,5,6] = { (1,3), (1,2), (1,1), (1,0) }.

4. Donner l'état de la mémoire physique après exécution de trois instructions.

Performances de la pagination à un niveau

Objectifs

L'objectif de cet exercice est d'examiner dans un cas simple l'impact de la pagination à la demande sur les performances d'un programme.

Présentation du système

On désire examiner les performances d'un petit programme de test, dont le code C est donné ci-dessous, dans un système de pagination à la demande.

```
#define N 256*1024
long tab[N];      // Un entier long est représenté sur 4 octets
long i,r;
// Initialisation du tableau
for (i=0;i<N;i++) tab[i] = i;
// Sommation des éléments du tableau
r = 0;
for (i=0;i<N;i++) r = r+tab[i];
```

On ignorera volontairement pour simplifier l'existence du code et de la pile utilisateur, ainsi que du système d'exploitation.

Les adresses virtuelles, octet, sont sur 32 bits. La taille des pages est de 16Ko, on utilise une table des pages à un seul niveau. Le tableau *tab* est implanté à l'adresse virtuelle 0.

Le système de pagination à la demande met en œuvre une politique de remplacement de page FIFO et alloue le swap dès la création d'un processus. La recopie d'une page modifiée s'effectue lors de son éviction de la mémoire. Les pages réelles seront allouées par numéro croissant. La taille mémoire disponible est de 512 Ko.

Questions

1. Donner le format d'une adresse virtuelle
2. Donner la taille du tableau *tab* et de la RAM en nombre de pages
3. Donner le contenu de la table des pages (V,pr,U,M) au début de l'exécution du programme
4. Reprendre la question 3 après un premier tour de la boucle d'initialisation du tableau
5. Donner le contenu de la table des pages après 16*1024 tours de la boucle d'initialisation

6. Donner la séquence d'événements (*défaut de page, remplacement de page*) se produisant lors de la séquence d'initialisation du tableau. Pour chaque événement, indiquer le numéro de page virtuelle et/ou réelle concerné. On se limitera aux 40 premiers événements.
7. En justifiant votre réponse, indiquer combien de défauts de page se produisent pendant l'exécution de l'ensemble du programme. Quel est le taux de défaut de pages ? Comment ce dernier varie t'il en fonction de la taille de la RAM ?

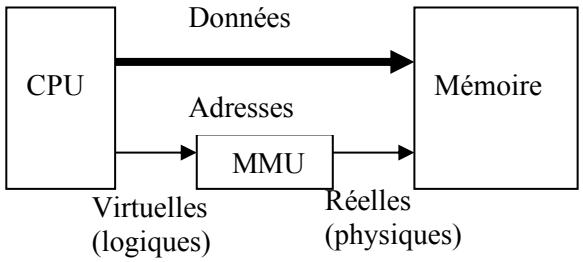
Gestion d'une mémoire physique paginée

Objectifs

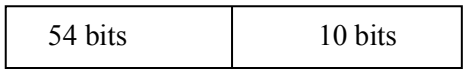
L'objectif de cet exercice est de construire le logiciel d'un système de pagination à la demande (défauts de page, remplacement de page), et d'examiner les performances de ce système, en utilisant un support matériel pour la pagination très simple.

Présentation du système

On considère un système bâti autour d'un processeur (CPU) relié à une mémoire à travers une MMU (Memory Management Unit), selon le schéma suivant :



Une adresse virtuelle est codée sur 64 bits. C'est une adresse octet qui a le format suivant :



N° de page virtuelle déplacement

Une adresse physique (réelle) a le format suivant :

N° de page réelle déplacement

A chaque processus p, on associe sa table des pages virtuelles TPV, indexée par le numéro de page virtuelle. Lorsqu'un processus est en cours d'exécution, un registre interne à la MMU pointe sur la table des pages du processus :

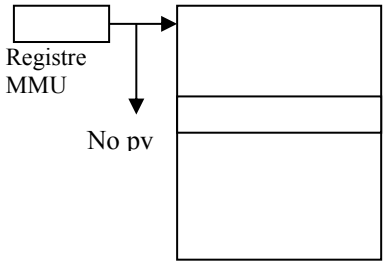
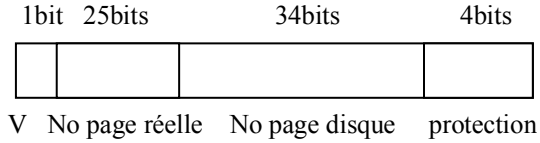


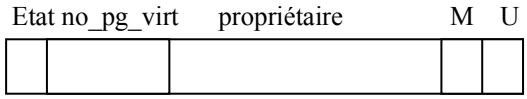
Table des pages

Une entrée de la table des pages (codée sur 64 bits) a le format suivant :



La MMU traduit l'adresse virtuelle en adresse physique de la façon suivante : le numéro de page virtuelle permet de retrouver l'entrée correspondante dans la table des pages virtuelles TPV. Si le bit V est à 0, un déroutement pour défaut de page est exécuté. Si l'accès demandé est en contradiction avec la protection de la page, un déroutement pour violation de protection est exécuté. Sinon, le numéro de page réelle est récupéré.

Une table des pages réelles (TPR) stocke l'état des pages en RAM. Une entrée dans cette table a le format suivant :



E est le bit d'écriture, U le bit d'utilisation (à noter que ces bits sont normalement localisés dans la TPV, on les a fait figurer ici dans le TPR pour simplifier le code). *Etat* peut prendre les valeurs *libre*, *occupé* ou *verrouillé*. Lorsqu'une page réelle est référencée, la MMU positionne le bit U correspondant, ainsi que le bit E si c'est une référence en écriture. On suppose que la mise à jour de ces informations de service est instantanée.

Un périphérique de pagination complète le système. Le périphérique est vu comme un ensemble de pages disques. Deux fonctions bloquantes permettent respectivement d'écrire et lire sur disque. Les requêtes de lecture ont priorité sur les requêtes d'écriture.

- écrire_pr_sur_pd(page_réelle,page_disque) ;
- lire_pd_dans_pr(page_disque,page_réelle);

On supposera que la routine de traitement des défauts de page est non interruptible, sauf pendant les accès disque.

On admet que :

- Le CPU fait en moyenne 1,4 accès mémoire par instruction
- Un cycle mémoire (transfert d'un mot de 64 bits) dure 100 ns
- Quand le CPU est connecté directement à la mémoire, une instruction dure en moyenne 2 ns, accès mémoire compris (grâce à la présence de caches de données et d'instructions).

Partie 1 : étude des performances

1. Quelle est la durée d'exécution d'un programme d'un milliard d'instructions sans MMU ?
2. Quelle est la durée d'exécution d'un même programme avec MMU mais sans défaut de pages ?
3. Pour améliorer la durée d'exécution du programme, on incorpore à la MMU une mémoire associative qui mémorise les 512 dernières correspondances (page virtuelle vers page réelle). En admettant que cette mémoire associative réalise directement 99.9% des correspondances, quelle serait la durée d'exécution de ce même programme, toujours sans défaut de page ?

Partie 2 : code du système de pagination à la demande

On va mettre en œuvre un mécanisme de pagination à la demande avec un algorithme de remplacement de page de type "LRU-global" (LRU sur l'ensemble des processus), en utilisant le bit U associé à chaque page virtuelle. On suppose que la routine de défaut de page appelle la fonction *donner_page* qui est chargée de trouver une page (page libre ou page réquisitionnée à un autre processus).

4. Ecrire la fonction *choix_page_à_vider* appelée par *donner_page*, qui fournit en résultat le numéro de la page réelle que l'on va enlever à un processus pour l'allouer au processus ayant déclenché le défaut de page. La variable *élu* désigne le processus en cours d'exécution.
5. Ecrire la fonction *trait_défaut(pv:num_page_virtuelle)* de traitement du défaut de page lors d'un accès à la page virtuelle pv par le processus actif, ainsi que la fonction *donner_page*. On suppose écrite la fonction *cherche_page_libre*, qui cherche une page physique libre.