SYSTÉMES INFORMATIQUES ET APPLICATIONS CONCURRENTES

Révision

Exercice 1. Gestion des ressources

Un système comprend 12 processus qui se partagent un stock de 36 ressources banalisées qu'ils demandent dynamiquement. Chaque processus est un interpréteur de commandes et selon la commande reçue fait une des deux suites de requêtes suivantes :

A : commande de lecture :: demande 3 ressources ; utilisation ; demande 2 ressources supplémentaires ; utilisation ; demande 1 ressources supplémentaires ; utilisation ; restitution des 6 ressources.

B : commande de mise à jour :: demande 3 ressources ; utilisation ; demande 3 ressources supplémentaires ; utilisation ; demande 2 ressources supplémentaires ; utilisation ; demande 1 ressources supplémentaires ; utilisation ; restitution des 9 ressources.

Question 1. Au démarrage du système, celui-ci reçoit les premières requêtes (demande de 3 ressources) de chacun des 12 processus. On ne sait pas quelles commandes (lecture ou mise à jour) vont être demandées. On suppose le pire pour l'allocateur. Celui-ci applique l'algorithme de prévention d'interblocage dit algorithme du banquier. Combien de processus peut-il servir sans crainte d'interblocage ? Expliquer votre réponse.

Question 2. On veut augmenter le stock pour que l'on ait assez de ressources pour ne jamais avoir d'interblocage et donc pour ne pas avoir à utiliser l'algorithme du banquier. On ne sait toujours pas quelles commandes (lecture ou mise à jour) vont être demandées. On suppose encore le pire. Quel est le nombre minimal de ressources qui permette de garantir l'absence d'interblocage. Expliquer votre réponse.

Question 3. Au démarrage du système, celui-ci reçoit les premières requêtes (demande de 3 ressources) de chacun des 12 processus. On configure le système pour qu'il y ait toujours 6 commandes de lecture et 6 commandes d'écriture. L'allocateur applique l'algorithme du banquier. Combien de processus peut-il servir sans crainte d'interblocage ? Expliquer votre réponse.

Question 4. On veut augmenter le stock pour que l'on ait assez de ressources pour ne jamais avoir d'interblocage et donc pour ne pas avoir à utiliser l'algorithme du banquier. On configure le système pour qu'il y ait toujours 6 commandes de lecture et 6 commandes d'écriture. Quel est le nombre minimal de ressources qui permette de garantir l'absence d'interblocage. Expliquer votre réponse.

Exercice 2. Ordonnancement de processus

Un système monoprocesseur comprend 5 processus A, B, C, D, E, qui vont être activés à leur date de déclenchement (à sa date de déclenchement, un processus est mis alors dans la file des processus prêts) pour demander l'unité centrale pour une durée d'exécution.

Processus	Durée d'exécution	Date de déclenchement	Date de fin
A	12	1	
B	3	0	
C	2	2	
D	1	3	
E	1	4	

Question . Pour chaque processus on vous demande de calculer la date de fin d'exécution selon la nature de l'ordonnancement du processeur.

1 Selon la politique de l'ancienneté (FIFO)

2 Selon la politique du tourniquet avec un quantum de 2. Quand un processus a une date de déclenchement qui coïncide avec un quantum (cas de C et E), on suppose qu'il est déclenché juste avant le quantum et qu'il est donc déjà dans la file des processus prêts au moment du quantum.

Exercice 3. Coopération de processus concurrents

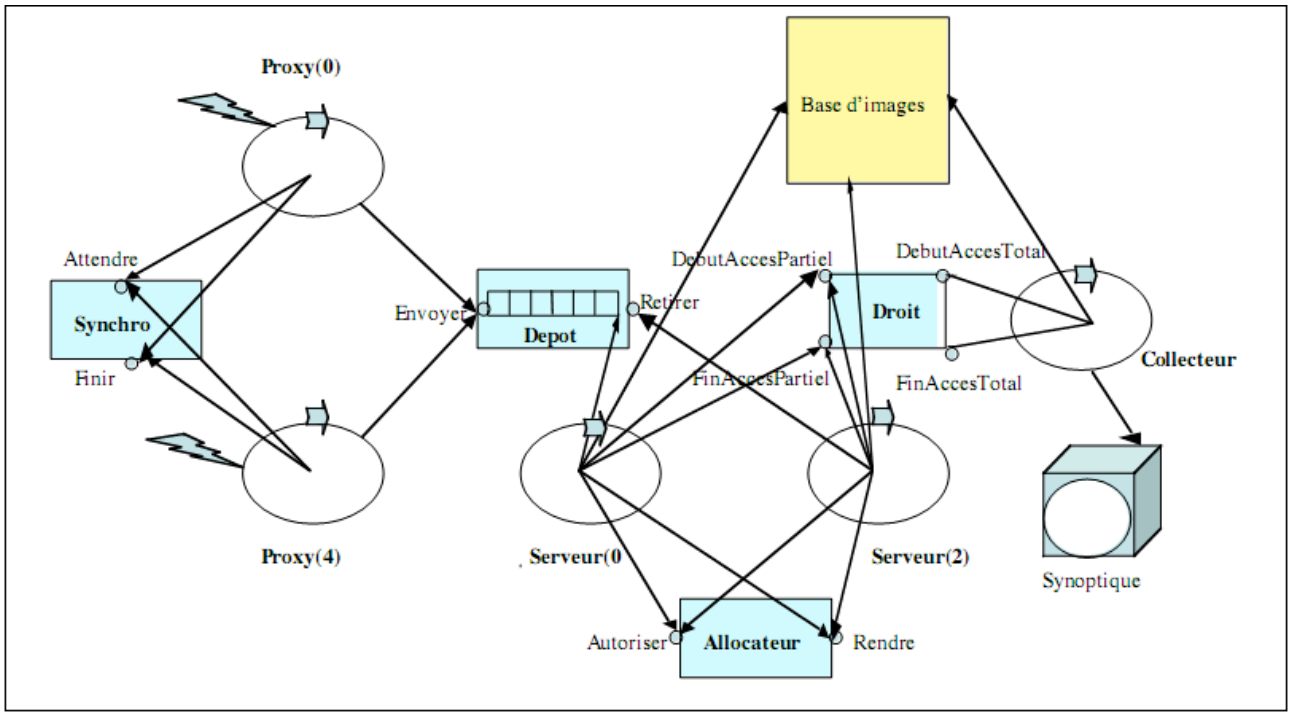
Une société qui possède 5 usines veut installer à son siége un écran synoptique pour suivre l'activité de ses usines. Après avoir étudié le système d'information et produit une architecture logique, cette société fait appel à vous pour gérer la coopération entre les processus concurrents de cette architecture.

// les processus coopérants
5 tâches **Proxy**
1 tâche **Serveur**
1 tâche **Collecteur**
 	
// procédures de coopération
void **Attendre**(Id_Usine X);
void **Finir**(Id_Usine X);
 	
// procédures de dépôt
void **Envoyer**(Rapport X);
Rapport **Retirer**() ;
	//procédures d'allocation
	void **Autoriser**(Id_S I, int X);
	void **Rendre**(Id_S I, int X);
	
	// procedure de gestion de droit
	void Debut_Acces_Partiel(Id_Usine X);
	void Fin_Acces_Partiel(Id_Usine X) ;
	void Debut_Acces_Total();
	void Fin_Acces_Total();

```
typedef  Id_Usine  enum {0,1,2,3,4} ;
typedef  Id_S  enum {0,1,2} ;

typedef struct {
        Id_Usine Usine ;
        int Date ;
        char Info ;
 } Rapport ;
```

```
typedef struct {
      Id_Usine : Usine ;
      int Date ;
      Dessin : Figure
} Image ;
```



Un processus Proxy(X) est attaché à chaque usine X pour en recevoir ses rapports, puis pour les déposer dans le tampon de dépôt. Les 5 processus Proxy sont cycliques et synchronisés entre eux pour qu'ils déposent les rapports les uns après les autres dans un ordre fixe.

```
TASK_CODE Un_Proxy ( Id_Usine X) {
Rapport Y;
while (1) {
    transfert_Du_Rapport_De_X(Y);// réception du rapport par un accès réseau
    Attendre(X);  /* synchronisation entre les proxys pour fixer l'ordre des dépôts */
    Envoyer(Y); // maintenant l'envoi du rapport Y est possible
    Finir(X);  // permet de passer la main au successeur de X
   }
} // end Un_Proxy;
```

Chacun des trois processus Serveur est cyclique et commence chaque cycle en retirant le plus ancien rapport encore présent dans le dépôt. Soit X l'usine qui a envoyé ce rapport. Le serveur acquiert la mémoire supplémentaire nécessaire pour traiter les statistiques et préparer l'image Z de X. On suppose qu'il faut X pages supplémentaires pour l'usine de numéro X. Quand il a cette mémoire supplémentaire, le Serveur exécute les calculs et construit l'image Z de X, puis il dépose cette image dans la base d'images à un emplacement réservé pour X et fixe.

```
TASK_CODE Un_serveur (Id_S I) {
 int X; Rapport Y; Image Z;
// I :  nom du Serveur
while (1) {
    Retirer(Y); //  retrait du plus ancien rapport présent dans le dépôt
    X=Y.Usine; // récupère le numéro d'usine
// si X > 0, demande le droit de prendre X pages de mémoire
    if (X > 0) { Autoriser(I, X) ; }
    Calculs_Statistiques_Et_Preparation_Image(Z);  // utilise un progiciel spécifique
    Debut_Acces_Partiel(X); /* demande le droit d'accéder à l'image Z de X dans la base
*/
    Mise_A_Jour_De_Image_De_X_Dans_La_Base; // accès à Z, réservé pour X
    Fin_Acces_Partiel(X); // sort de cet accès
    if (X > 0) {Rendre(I, X); }// n'a plus besoin de la mémoire supplémentaire
  } // fin du while
} // end Un_Serveur
```

Le processus Collecteur est un processus cyclique activé périodiquement pour aller lire toutes les images de la base d'images, préparer une image de synthèse, la stocker dans la base. Ensuite il visualise les images sur le synoptique.

```
TASK_CODE Collecteur () {
while (1) {
    Attendre_Le_Reveil_Periodique; //déclenchement  externe
    Debut_Acces_Total(); // demande le droit d'accéder à la base en exclusion mutuelle
    Travail_Dans_La_Base; // avec un progiciel spécialisé
    Visualiser_Les_Images_De_La_Base; // avec un outil de visualisation
    Fin_Acces_Total(); // libère l'accès à la base
    } // end while
} // end Collecteur
```

On vous demande de programmer les différentes procédures présentées ici en utilisant des sémaphores pour le contrôle de concurrence :

Question 1. Synchro. Les procédures *Attendre*() et *Finir*() servent à coordonner les Proxys de telle façon que leurs phases de dépôt de rapport soient exécutées les unes après les autres selon l'ordre des numéros d'usine: 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, …

Avec *Attendre*(X) le processus Proxy(X) attend que le processus Proxy(X - 1) ait fini et quand

Proxy(X) a fini, il le signale par *Finir*(X), ce qui autorise le Proxy(X + 1) à s'exécuter. Les additions et soustractions sont modulo 5 car c'est le type de X.

Programmer ces procédures en gérant la concurrence avec des sémaphores. Ne pas oublier d'initialiser les sémaphores.

Question 2. Depot. Les procédures *Envoyer*() et *Retirer*() servent à contenir les rapports déposés par les Proxys et retirés par les Serveurs. Le tampon est géré à l'ancienneté et peut contenir jusqu'à 8 rapports.

On demande de programmer pour avoir le maximum de concurrence d'accès entre les processus utilisant ce paquetage. A un instant donné, quel est le nombre maximum de Proxys qui peuvent appeler ces procédures? En déduire la programmation qui utilise le moins de sémaphores.

Question 3. Allocateur. Les procédures *Autoriser*() et *Rendre*() sont utilisées par les Serveurs pour contrôler leurs demandes de pages. On peut programmer la procédure *Autoriser*(I,X) pour traiter la demande de X pages en demandant l'un après l'autre le droit de les utiliser et cela jusqu'à ce que l'utilisation de X pages soit autorisé. Cela permet d'éviter la famine. Le Serveur est alors autorisé à aller prendre ces X pages et à les utiliser puisqu'elles ont toutes été réservées.

De quel minimum de ressources faut-il disposer pour qu'il n'y ait jamais d'interblocage, sachant qu'il y a trois Serveurs concurrents pouvant demander 4 pages chacun?

Programmer les procédures Autoriser() et Rendre() en supposant qu'on dispose du minimum de ressources correspondant. La gestion de la concurrence sera programmée avec des sémaphores. On ne programmera pas la recherche des X pages allouées, mais seulement l'autorisation de faire cette recherche.

Question 4. Droit. Les procédures de gestion du droit d'accès sont utilisées par les Serveurs et par le Collecteur. Chaque Serveur fait accès uniquement à une seule image de la base d'images, celle de l'usine X pour laquelle il travaille. La procédure *Debut_Acces_Partiel*(X) autorise les Serveurs qui travaillent pour des images différentes à utiliser la base d'images en parallèle. Comme plusieurs Serveurs peuvent servir la même usine X concurremment et demander l'accès à la même image de X, on doit, dans cette procédure, imposer au préalable qu'il n'y ait que l'un d'eux à la fois qui puisse utiliser la base d'images pour X, et donc bloquer les autres demandes pour X. Dans la procédure *Fin_Acces_Partiel*(X), il faut alors in fine penser à réveiller un autre serveur qui travaille pour X et qui aurait été bloqué dans *Debut_Acces_Partiel*(X). Et ce contrôle doit être fait pour chaque X. Le Collecteur fait un accès à plusieurs images de la base d'images. Il doit donc pouvoir utiliser la base en exclusion mutuelle avec les Serveurs.

Programmer ces procédures en gérant la concurrence avec des sémaphores.

Solution

Exercice 1 : Gestion des ressources

Question 1: Dans l'algorithme du banquier, on accepte de satisfaire une requête s'il va rester suffisamment de ressources permettant de satisfaire les futures demandes. En appliquant ce principe, on s'aperçoit que parmi les 12 requêtes de 3 ressources, 10 peuvent être servies. Il reste alors 6 ressources, ce qui permet de servir toutes les requêtes possibles d'un processus qui ferait une commande de mise à jour, puis, cette mise à jour terminée et les 9 ressources récupérées, de servir toutes les autres commandes, éventuellement l'une après l'autre (dans le pire cas, ce sont toutes des mises à jour) une fois celle-ci terminée.

Question 2: On applique le calcul de la politique de précaution avec $X > \Sigma (Ci - 1)$. $Ci = C = 9$. Le nombre minimal de ressources qui permette de garantir l'absence d'interblocage est alors $12*8 + 1 = 97$.

Question 3: 11 processus peuvent recevoir leur première requête de 3 ressources, car il reste 3 ressources pour servir une commande de lecture (il y en a au moins 5 qui peuvent être servies, sur ces 11) et celle-ci terminée, on aurait les 6 ressources pour servir une requête de mise à jour ou un autre requête de lecture, donc toutes les autres requêtes.

Question 4 : On applique le calcul de la politique de précaution avec $X > \Sigma (Ci - 1)$. On a deux annonces possibles C1= 6 et C2 = 9. Le nombre minimal de ressources qui permette de garantir l'absence d'interblocage est alors $6*5 + 6*8 + 1 = 79$.

Exercice 2

Solution 1.: Ordonnancement

FIFO	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A				A	A	A	A	A	A	A	A	A	A	A	A				
B	B	B	B																
C																C	C		
D																		D	
E																			E
File d'attente	A	AC	ACD	CDE	CDE	CDE	CDE	CDE	CDE	CDE	CDE	CDE	CDE	CDE	CDE	DE	DE	E	

tourniquet	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A			A	A						A	A	A	A	A	A	A	A	A	A
B	B	B					B												
C					C	C													
D								D											
E									E										
File d'attente	A	AC	CBD	CBDE	BDEA	BDEA	DEA	EA	A										

Exercice 3 : Coopération des processus

Question 1 : Synchro

// declaration de semaphores

```
SEM Sem_Signal[5];

void Attendre(Id_Usine X) {  P(Sem_Signal[X]);  }
void Finir(Id_Usine X) { V(Sem_Signal[(X+1) % 5]) ; }
```

// initialisation des semaphores

```
init_sem() {
   for (i=0; i<5; i++) {E0(Sem_Signal[i], 0); }
   V(Sem_Signal[0]);
}
```

Question 2. Depot

// declaration

```
SEM Nvide, Nplein, Mutex_Cons ;
typedef Id_Depot enum {0 , 1, 2, …, 7} ;
Rapport T[8];
Id_Depot Tete=0, Queue=0;

void Envoyer(Rapport X) {
   /* il n'y a qu'un seul proxy producteur à la fois, pas de concurrence
pour le dépôt */
   P(Nvide); T[Queue]= X; Queue = (Queue + 1) % 8; V(Nplein);
} // end Envoyer

Rapport  Retirer() {
   Rapport Y;
   P(Nplein);
   P(Mutex_Cons); Y = T[Tete]; Tete = (Tete + 1) % 8; V(Mutex_Cons);
   V(Nvide);
   return (Y) ;
} // end Retirer

init_sem() {
   E0(Nvide, 8); E0(Nplein, 0); E0(Mutex_Cons, 1);
}
```

Question 3. Allocation

On note que les 3 serveurs, qui ne sont pas synchronisés entre eux, peuvent retirer chacun un rapport pour l'usine 4. Par exemple le premier traite l'usine 4. Pendant ce temps le second traite l'usine 0, puis l'usine 2, puis commence à traiter l'usine 4 pour le rapport suivant de cette usine. Le troisième serveur traite l'usine 1 puis l'usine 3, puis l'usine 0, puis l'usine 1, puis l'usine 2, puis l'usine 3, puis l'usine 4 pour un nouveau rapport. Chaque serveur traite donc un rapport concernant des dates successives de l'usine 4. Cela conduit à demander 4*3 = 12 pages à l'allocateur.

/*les autorisations sont allouées une après l'autre et sont capitalisées par chaque demandeur. Le pire cas est celui où chacun des 3 serveurs a demandé X=4 pages et a reçu 3 autorisations, ce qui fait 9 pages déjà réservées. S'il n'y a que 9 pages, on est en interblocage. Avec 10 au minimum, on l'évite. */

//déclaration

```
#define Min 10
SEM N_Allouable; // à initialiser avec le nombre des ressources allouables


void Autoriser(Id_S I, int X) {
int Encore ;
  Encore = X; // sauf erreur, X est positif
  // demande X autorisations l'une après l'autre
  while (Encore !=0) { P(N_Allouable); Encore = Encore - 1; }
} //end Autoriser


int Rendre(Id_S I, int X)  {
int Encore ;
  Encore = X; //sauf erreur, X est positif
  // retourne X autorisations
  while(Encore !=0) { V(N_Allouable); Encore := Encore - 1; }
} // end Rendre


  init_sem(){
    E0(N_Allouable, Min);
  }
```

Question 4. Droit

Chacun des trois serveurs prend un rapport dans le Dépôt et le traite à son rythme. On peut avoir Serveur(0) qui traite un rapport pour l'usine 4 et que ce soit très long. Pendant ce temps, les Serveur(1) et Serveur(2) traitent les rapports suivants du Dépôt, soit ceux des usines 0, 1, 2, 3.

Supposons encore que Serveur(1) prenne ensuite le rapport suivant de l'usine 4 et que ce soit aussi long à traiter. Pendant ce temps Serveur(2) obtient les rapports suivants du Dépôt, ceux des usines 0, 1, 2, 3 et finit par avoir un troisième rapport de l'usine 4. Dans ce cas on aurait les trois Serveurs occupés à traiter trois rapports successifs concernant l'usine 4.

Comme ces trois Serveurs peuvent terminer leurs traitements pour l'usine 4 à tout moment et dans n'importe quel ordre, l'image qui sera visualisée n'est pas toujours la plus récente. Si on voulait que ce soit le cas, il faudrait éviter d'écraser une image par une image plus ancienne et utiliser la date de l'image pour cela.

// déclaration

```
SEM Mutex[5];
SEM Mutex_L, Mutex_A;
int NL= 0;

procedure Debut_Acces_Partiel(Id_Usine X) {
   P(Mutex[X]); // un seul accès pour l'usine X
   P(Mutex_L); /* accès concurrents avec des accès pour d'autres usines
que X */
   NL= NL + 1; if (NL = =1) { P(Mutex_A); }
   V(Mutex_L);
} //end Debut_Acces_Partiel

void Fin_Acces_Partiel() {
   P(Mutex_L); // fin d'accès concurrents
   NL= NL - 1; if (NL == 0) {V(Mutex_A); }
   V(Mutex_L);
   V(Mutex[X]); /* fin d'accès pour l'usine X, un autre accès pour X
peut se faire */
} //end Fin_Acces_Partiel

void Debut_Acces_ Total() { P(Mutex_A); }

void  Fin_Acces_ Total() {V(Mutex_A); }

init_sem(){
   for (I=0; I<5; I++) {E0(Mutex[I], 1); }
   E0(Mutex_L, 1); E0(Mutex_A, 1);
}
```