Leçon : Communication inter-processus

Définitions

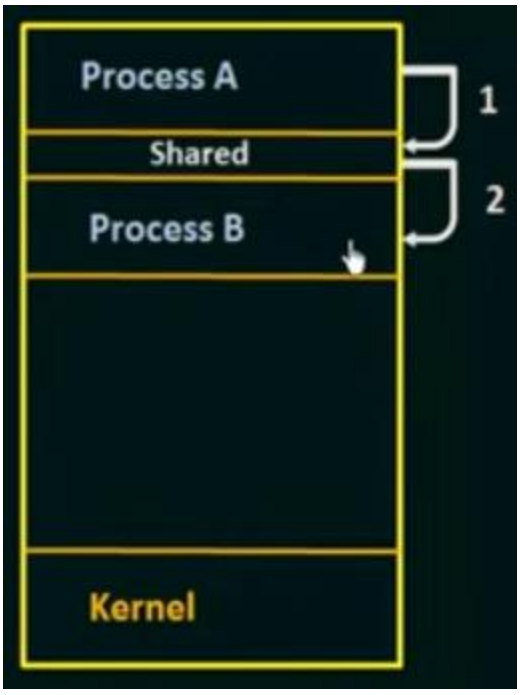
- Un processus indépendant : processus qui ne peut pas affecter ou être affecter pas d'autre processus s'exécutant de manière concurrente dans le système.
- Un processus coopératif : processus qui peut affecter ou être affecter pas d'autre processus s'exécutant de manière concurrente dans le système. Exemple : Tout processus qui partage les données avec un autre processus est un processus coopératif
- Communication inter-processus : façon dont le processus coopératif communique entre eux.

Méthodes de communication inter-processus

Il existe deux méthodes classiques de communication inter-processus : les variables partagées (shared variables) et l'échange de messages (message passing).

Variables partagées

La méthode des variables partagées consiste à établir une région en mémoire qui sera partagée par les processus coopératifs. Ainsi ces processus peuvent s'échanger des informations en lisant et modifiant des données dans la région partagée. Cependant cette méthode peut causer des problèmes de conflit d'accès.



Exemple : un processus A est dédié à la gestion d'une imprimante, un processus B désire imprimer un document. B doit communiquer à A le nom et l'emplacement du fichier qu'il veut imprimer. Supposons que plusieurs processus veulent imprimer leurs résultats sur une imprimante commune. Les noms des

fichiers à imprimer sont placés dans un tableau spécial fonctionnant comme une file : la variable tête qui indique le premier fichier dans la file, et la variable queue qui indique l'endroit d'insertion d'un nouveau fichier dans la file sont des variables partagées.

Supposons également que le processus B veut imprimer un fichier, nommé F_b. ces instructions sont les suivantes :

I1b : Charger le fichier F_b dans [queue]

I2b : queue=queue+1

Exemple de conflit d'accès : Deux processus B et C veulent placer au même moment leurs fichiers (F_b et F_c) dans la file d'impression. Supposons que l'ordonnancement s'exécute alternativement : I1b, I1c, I2b, I2c.

Problème : Le fichier F_b ne sera jamais imprimé, car il a été écrasé par F_c.

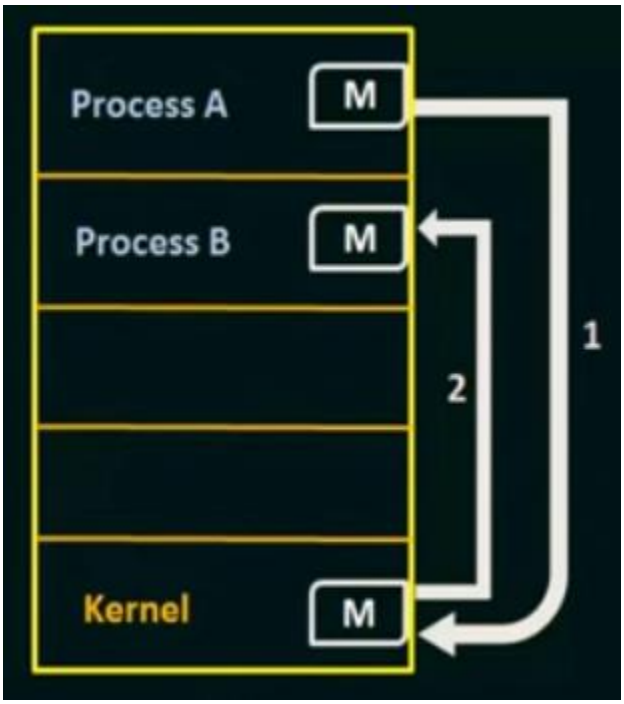
Source du problème : la variable partagée queue est lue et modifiée par deux processus à la fois.

Échanges de messages

La méthode d'échange de message consiste à l'envoi et la réception de message aux processus coopératif.

En effet les mécanismes de communication par variables partagées conviennent bien à des processus qui tournent sur le même processeur ou sur plusieurs processeurs possédant une mémoire commune. Pour des systèmes distribués, i.e. constitués de plusieurs processeurs ayant chacun leur propre mémoire et reliés par un réseau local, la communication inter-processus peut se faire par un échange de message :

- Un processus peut envoyer un message par réseau à un destinateur précis : send(destination, message)
- Un autre processus peut « consommer » un message par : receive(source, message)



Problème lie à la communication inter-processus

Les processus coopérants sont confrontés à deux grands problèmes : la famine et l'inter-blocage (deadlock).

- Les inter-blocages surviennent lorsque deux ou plusieurs processus se bloquent dans l'attente d'un événement, sous le contrôle de l'autre processus bloqué. Une situation de blocage sur une ressource peut survenir si et seulement si toutes les conditions suivantes sont réunies simultanément dans un système :
    - Exclusion mutuelle : Un seul processus peut utiliser la ressource à un instant donné.
    - Attente et occupation : un processus détient actuellement au moins une ressource et demande des ressources supplémentaires qui sont détenues par d'autres processus.
    - Non préemption : une ressource ne peut être libérée que volontairement par le processus qui la détient.
    - Attente circulaire : chaque processus doit attendre une ressource qui est détenue par un autre processus, qui à son tour attend que le premier processus libère la ressource

- La famine survient lorsqu'un processus attend indéfiniment une ressource (qui est éventuellement occupée par d'autre processus)

Cas du problème du producteur consommateur

Il s'agit de partager entre deux processus (le producteur et le consommateur), une zone de mémoire tampon utilisée comme une file (variable partagée). Le producteur génère un élément de données, l'enfile sur la file et recommence ; simultanément, le consommateur retire les données de file.

- Famine : Si le producteur souhaite placer un nouvel élément dans le tampon alors que ce dernier est déjà plein. Le producteur devra attendre qu'au moins 1 consommateur consomme pour libérer le tampon. De la même façon, si le consommateur souhaite récupérer un élément dans le tampon, alors que celui-ci est vide, il devra attendre jusqu'à ce que le producteur ait placé quelque chose dans le tampon.

Cas du problème des philosophes

5 philosophes (un philosophe représente un processus) sont assis autour d'une table. Sur la table se trouvent 5 assiettes remplies de spaghettis et 5 fourchettes (les fourchettes sont des ressources partagées) ; pour manger les spaghettis il faut 2 fourchettes (les spaghettis sont très glissants). Si un philosophe n'a pas faim, il pense (exécute un programme). S'il a faim il essaye d'attraper 2 fourchettes voisines (accéder à 2 ressources) et de manger. Deux risques se présentent :

- Inter Blocage (deadlock, étreinte fatale) : Les 5 philosophes attrapent au même moment leur fourchette gauche et la gardent jusqu'à pouvoir attraper aussi leur fourchette droite, les 5 processus sont indéfiniment bloqués.
- Famine (starvation) : les 5 philosophes attrapent au même moment leur fourchette gauche, voient que leur fourchette droite n'est pas disponible, reposent leur fourchette gauche, et retente à nouveau d'attraper les 2 fourchettes. S'ils répètent ces tentatives toujours tous au même moment, aucun n'arrive à manger alors qu'il y a assez de spaghettis pour tous !

Synchronisation Interprocessus

Une section critique (SC) est une séquence d'instructions manipulant des données partagées par plusieurs processus et qui peut produire des résultats imprévisibles lorsqu'elle est exécutée simultanément par de différents processus.

Les problèmes liés à la communication interprocessus, peuvent se résume au contrôle de l'occupation de la zone critique, par un et un seul processus à la fois. Ainsi un model de solution est celui de l'utilisation de l'exclusion mutuelle.

L'exclusion mutuelle est une méthode qui permet de s'assurer que si un processus utilise une variable ou une ressource partagée, les autres processus seront exclus de la même activité. Pour assurer l'exécution des sections critiques des processus en exclusion mutuelle, des solutions ont été mise en œuvre.

Solution monoprocesseur : désactivation des interruptions

L'idée principale de cette solution est le masquage d'interruptions les commutations de processus qui pourraient violer l'exclusion Mutuelle des SC sont empêchées : chaque processus entrant dans la section critique désactive toutes les interruptions et les réactive à sa sortie de la section critique. Seule l'interruption générée par la fin du quantum de temps nous intéresse, il ne faut pas qu'un processus attende une interruption de priorité inférieure à celle générée par la fin du quantum de temps à l'intérieur de SC. Cette solution n'est pas applicable au sein d'un environnement multiprocesseur

Inconvénients :

• Les interruptions restent masquées pendant toute la SC, d'où risque de perte d'interruptions ou de retard de traitement.

• Une SC avec while(1) bloque tout le système

• Les systèmes ne permettent pas à tout le monde de masquer n'importe comment les interruptions

Variables de verrou

Un mécanisme proposé pour permettre de résoudre l'exclusion mutuelle d'accès à une ressource est le mécanisme de verrou (en anglais lock). Un verrou est un objet système représentée par une variable unique et partagée dont la valeur initiale est de 0. Cette solution consiste à résoudre le problème de partage de variables par l'utilisation du verrou. Un processus tentant à entrer en section critique doit tout d'abord tester la valeur du verrou si elle est égal à 0, le processus lui affecte 1 puis entre en SC ; sinon le processus doit attendre que le verrou passe de nouveau à 0. En sortant de la SC un processus doit libérer le verrou et ce par sa remise à 0. Si un ou plusieurs processus étaient en attente de ce verrou, un seul de ces processus est réactivé c'est lui qui reçoit le verrou et le met à 1. 1 : variable ou ressource partagée occupée : le verrou est non disponible => interdiction de l'entrée en SC. 0 : variable ou ressource partagée libre : le verrou est disponible et le processus peut l'acquérir =>autorisation de l'entrée en SC.

Alternance Stricte

L'allocation des ressources par alternance est une proposition au problème qui est simple et que nous proposons ici pour bien comprendre l'idée de l'attente active. Elle n'est pas applicable aux cas généraux, puisqu'elle suppose que les processus accédant aux ressources s'exécutent à la même vitesse, et font des accès en alternance. Elle repose sur un compteur tour qui indique l'identifiant de processus qui a le droit d'utiliser la ressource et qui est partagée entre les deux processus. Lorsqu'un processus a terminé sa section critique, il incrémente tour, modulo le nombre de processus, pour que sa valeur pointe sur le processus suivant. Lorsqu'un processus est en attente de l'allocation de la ressource, il exécute une boucle infinie. C'est le fait d'exécuter des commandes inutiles (i.e. la boucle infinie) qui forme la caractéristique principale de l'attente active; un processus en attente occupe le processeur avec des opérations inutiles. (a) : ce code est exécute par le processus 0, il entre en SC si tour=0. ◊ Le processus 0 teste la valeur de la variable tour ◊ Si tour est différente de 0, il est mis en attente et il n'est pas autorisé à entrer en SC. Au cours de cette attente il fait des tests de façon continue sur la valeur de tour

◊ Lorsque tour passe à 0, le processus 0 entre en SC ◊ Après avoir fini avec sa SC il met tour à 1 en faveur du processus 1 pour que ce dernier entre en SC. while (TRUE) { while (tour != 0) /* attente */ section_critique(); tour = 1; section_noncritique(); } (b) ce code est exécute par le processus 1 afin de vérifier s'il possède l'autorisation d'entrer en SC. while (TRUE) { while (tour != 1) /* attente */ section_critique(); tour = 0; section_noncritique(); } En dehors de l'hypothèse de l'alternance la solution présentée ne marche pas. Supposons, par exemple, que le processus (a) est un processus très rapide, et

qu'il vient de terminer sa section critique. (b) est très lent, est actuellement en section non critique, et n'aura pas besoin d'entrer en section critique avant très longtemps. (a) termine rapidement sa section non critique, et veut à nouveau entrer en section critique. Il ne pourra pas, puisque la variable tour n'est pas positionnée correctement. Ces tests sont très consommateurs en termes de temps processeur.

Solution de Peterson Peterson

offre un algorithme élégant et simple à l'exclusion mutuelle. Solution symétrique pour N processus. L'inter-blocage est évité grâce à l'utilisation d'une variable partagée Tour qui est utilisée de manière absolue et non relative; #define N 2 /* nombre de processus */ /* Les variables partagées par tous les processus */ int tour; /* à qui le tour */ int interesse[N]; /* initialisé à FALSE */ void entrer_region(int process) /* numéro du processus appelant */ /* ici process vaut 0 ou 1 */ { int autre; /* numéro de l'autre processus */

autre = 1-process; interesse[process] = TRUE; /* indiquer son intérêt */ tour = process; /* positionner le drapeau d'accès */ while(tour == process && interesse[autre] == TRUE); } void quitter_region(int process) /* numéro du processus appelant */ /* ici process vaut 0 ou 1 */ { interesse[process] = FALSE; /* indiquer la sortie de la section critique */ } Dans cette solution, chaque processus doit, avant d'utiliser des variables partagées, appeler entrer_region() en lui fournissant son identifiant de processus en paramètre. Cet appel ne retournera la main que lorsqu'il n'y a plus de risque. Dès que le processus n'a plus besoin de la ressource, il doit appeler quitter_region() pour indiquer qu'il sort de sa section critique et que les autres processus peuvent accéder à la ressource partagée. L'idée principale de l'algorithme est d'utiliser deux variables différentes, l'une indiquant que l'on s'apprête à utiliser une ressource commune (interesse[ ]), et l'autre (tour) qui, comme dans le cas de l'alternance, indique le processus qui utilisera la ressource au prochain tour. Lorsqu'un processus entre en section critique, il notifie les autres de ses intentions en positionnant correctement son entrée dans le tableau interesse[ ], puis il s'approprie le prochain tour. Tant que lui, est le prochain utilisateur (tour == process), mais l'autre processus a également l'intention d'utiliser (ou utilise déjà) la ressource, le processus courant attend que l'autre libère la ressource. En fin de section critique, en appelant quitter_region(), le processus sortant signale aux autres qu'il n'utilise plus la ressource. Il est à noter que toute la partie traitée dans ce cours concernant les exclusions mutuelles, repose fondamentalement sur le respect des règles d'utilisation de tous les processus voulant accéder aux ressources partagées. Ici, par exemple, on utilise un tableau partagé (interesse[ ]) dans lequel chaque processus indique son intention d'utiliser la ressource partagée. La règle implicite est, bien-sûr, que chaque processus n'écrit qu'à l'indice du tableau qui lui est dédié, et non pas aux autres endroits

L'attente passive

Les solutions déjà énoncées possèdent toutes l'inconvénient de faire appel à l'attente active : un processus souhaitant entrer en section critique vérifie s'il en a le droit sinon il entre en boucle d'attende de l'autorisation de l'entrée en SC. L'attente passive demande une intervention de la part du système

(qui doit par conséquent fournir les appels au noyau nécessaire), et évite une surcharge inutile du processeur. Il évite également le problème d'inversion de priorité, puisque l'idée de base consiste à basculer le processus en attente de l'accès à une ressource partagée en mode bloqué, jusqu'à ce que la ressource se libère, ce qui la fait basculer en mode prêt. Il existe une quantité importante de solutions pour implanter des méthodes d'attente passive. Pour remédier à ce problème, certaines primitives de communication inter-processus permettent de bloquer les processus au lieu de consommer du temps processeur lorsqu'ils n'obtiennent pas l'autorisation d'entrer dans leurs sections critiques. L'une des simples fonctionne avec la paire Sleep et Wakeup .Sleep et Wakeup prennent un paramètre qui indique le processus à bloquer ou réveiller

Les sémaphores

Le contrôle de la synchronisation par l'utilisation d'un type de données abstrait appelé sémaphore a été proposé par Dijkstra en 1965. Les sémaphore sont très facilement mis en œuvre dans les systèmes d'exploitation et constituent une solution générale pour contrôler l'accès aux sections critiques. Un sémaphore est une variable entière non négative à partir de laquelle sont définies deux opérations élémentaires : P et V (d'après les termes néerlandais proberen : tester et verhogen : incrémenter). Struct semaphore { Int count ; ProcessQueue queue ; } ; Void P(semaphore s){ if(s.count>0){ s.count=s.count-1; else s.queue.Insert(); //Bloque ce processus } Void V(semaphore s){ s.count=s.count+1 ; if(!s.queue.empty()) s.queue.Remove() ;//libère un processus //bloqué sur 's', s'il existe. } Il se présente comme un distributeur de jetons, mais le nombre de jetons est fixe et non renouvelable: les processus doivent restituer leur jeton après utilisation. S'il y a un seul jeton en circulation, on retrouve le verrou. Les opérations sémaphores sont parfois également connues sous le nom de Up (haut) et Down(bas) ou Wait et Signal ♣ P(s) ou wait correspondant à sleep permet à un processus d'obtenir un jeton, s'il y en a de disponibles. Si aucun n'est disponible, le processus est bloqué. (P(S) : si S=0 alors mettre le processus en attente, sinon S ← S-1) Wait(S) If(S=0) then sleep() ; S-- ; ♣ V(s) ou signal correspondant à wakeup permet à un processus de restituer un jeton. Si des processus étaient en attente de jeton, l'un d'entre eux est réactivé et le reçoit. (V(S) : S ← S+1 ; réveiller un (ou plus) processus en attente) Signal(S) Wakeup() ; S++ ; Une fois instanciée, la valeur compteur d'un sémaphore peut recevoir toute valeur non négative. Avec une valeur initiale de 1, un sémaphore assure mutuellement un accès exclusif si P est exécuté avant d'entrer dans une section critique et si V l'est à la sortie. En initialisant la valeur initiale du sémaphore sur n, P et V peuvent être utilisés pour autoriser jusqu'à n processus dans leurs sections critiques. Un sémaphore binaire est un sémaphore dont le compteur ne peut admettre que les valeurs 1 ou 0. Void P(semaphore s){ if(s.count==1) s.count=0 ; else s.queue.Insert() ; // Bloque ce processus } void V(semaphore s){ if(s.queue.empty()) s.count=1 ; else s.queue.Remove() ; ;//libère un processus //bloqué sur 's', s'il existe. } Pour distinguer plus clairement les types de sémaphores, précisons que ceux peuvent prendre des valeurs non négatives peuvent être appelés sémaphores généraux ou sémaphores compteur

On arrive à l'exclusion mutuelle en initialisant une variable sémaphore à un, en exécutant une opération P avant d'entrer dans la section critique et en exécutant une opération V après avoir quitté la section critique. À la création d'un sémaphore, il faut décider du nombre de jetons dont il dispose. On voit que si une ressource est à un seul point d'accès (critique), le sémaphore doit avoir initialement 1 jeton, qui sera attribué successivement à chacun des processus demandeurs. Si une ressource est à n points

d'accès, c'est-à-dire, peut être utilisée par au plus n processus à la fois, il suffit d'un sémaphore initialisé avec n jetons. Dans les deux cas, un processus qui cherche à utiliser la ressource, demande d'abord un jeton, puis utilise la ressource lorsqu'il a obtenu le jeton et enfin rend le jeton lorsqu'il n'a plus besoin de la ressource.

Solution avec échanges de messages Certains ont estimé que les sémaphores sont de trop bas niveau. Ils ont proposé un mode de communication inter-processus qui repose sur deux primitives qui sont des appels système : - send (destination , &message) - receive (source , &message), où source peut prendre la valeur générale ANY Généralement, pour éviter les problèmes dans les réseaux, le récepteur acquitte le message reçu. L'émetteur envoie à nouveau son message s'il ne reçoit pas d'acquittement. Le récepteur, s'il reçoit deux messages identiques, ne tient pas compte du second et en tire la conclusion que l'acquittement s'est perdu. Dans le contexte d'un mode client-serveur, le message reçu contient le nom et les paramètres d'une procédure à lancer. Le processus appelant se bloque jusqu'à la fin de l'exécution de la procédure et le message en retour contient la liste des résultats de la procédure. On parle d'appel de procédure à distance. On peut proposer une solution au problème producteur-consommateur par échange de messages avec les hypothèses suivantes : - les messages ont tous la même taille - les messages envoyés et pas encore reçus sont stockés par le SE dans une mémoire tampon - le nombre maximal de messages est N - chaque fois que le producteur veut délivrer un objet au consommateur, il prend un message vide, le remplit et l'envoie. Ainsi, le nombre total de messages dans le système reste constant dans le temps - si le producteur travaille plus vite que le consommateur, tous les messages seront pleins et le producteur se bloquera dans l'attente d'un message vide ; si le consommateur travaille plus vite que le producteur, tous les messages seront vides et le consommateur se bloquera en attendant que le producteur en remplisse un