cour 03 : Réalisation de processus

1. Le principe des interruptions :

a. Rappel :

- **Une interruption:** est un mécanisme permettant d'interrompre temporairement l'exécution normale d'un programme ou d'un processus pour traiter un événement ou une condition spécifique. Les interruptions sont un élément essentiel des systèmes informatiques, car elles permettent de gérer des événements asynchrones et de réagir rapidement à des événements matériels ou logiciels.
- Les principes des interruptions :

1. **Mécanisme câblé** : Les interruptions sont un mécanisme câblé dans le matériel de l'unité centrale de traitement (UC) d'un ordinateur. Cela signifie qu'elles sont gérées directement par le matériel de l'ordinateur plutôt que par un logiciel.
2. **Déclenché par un événement extérieur à l'UC** : Les interruptions sont déclenchées par des événements externes à l'unité centrale de traitement (UC). Ces événements peuvent être des signaux matériels, des interruptions de périphériques, des erreurs matérielles, etc.
3. **Sauvegarde du mot d'état de programme** : Lorsqu'une interruption se produit, l'UC sauvegarde l'état actuel du programme en cours d'exécution. Cela inclut généralement des informations telles que les valeurs des registres et l'emplacement du compteur de programme.
4. **Charge le mot d'état de programme par une valeur associée à l'événement externe** : Après avoir sauvegardé l'état du programme, l'UC charge le compteur de programme avec une valeur associée à la nature de l'événement externe. Cette valeur pointe généralement vers l'emplacement mémoire où réside le programme qui doit être exécuté en réponse à l'interruption. Ce programme est souvent appelé "traitant de l'événement."
5. **Lancement de l'exécution du traitant de l'événement** : Une fois le compteur de programme chargé avec l'adresse du traitant de l'événement, l'UC commence à exécuter ce programme. Le traitant de l'événement est responsable de la gestion spécifique de l'interruption, de la prise en compte de l'événement et de l'exécution des actions appropriées.
6. **Possibilité de différer la prise en compte d'un événement (masquage)** : Dans certains cas, il peut être nécessaire de différer la prise en compte d'un événement. Par exemple, les interruptions peuvent être masquées temporairement pour garantir l'intégrité des opérations critiques ou pour s'assurer que certaines interruptions sont traitées en priorité par rapport à d'autres.

b. Utilisation des interruptions pour la gestion des processus

L'utilisation des interruptions pour la gestion des processus :

1. **Existence d'une horloge** : Le système dispose d'un dispositif d'horloge qui permet de décompter un intervalle de temps. Cette horloge génère régulièrement des interruptions à des intervalles prédéfinis, appelées interruptions d'horloge.

2. **Traitant de l'interruption horloge** : Lorsqu'une interruption d'horloge se produit, le processeur interrompt l'exécution normale du processus en cours et exécute une routine de gestion d'interruption spécifique appelée "traitant de l'interruption d'horloge."

3. **Rangement des registres** : Le traitant de l'interruption d'horloge commence par sauvegarder l'état du processus en cours, y compris les valeurs des registres et l'emplacement du compteur de programme, dans une structure de données appropriée. Cela permet de préserver l'état du processus afin qu'il puisse reprendre son exécution ultérieurement.

4. **Choix d'un nouveau processus** : Après avoir sauvegardé l'état du processus en cours, le traitant de l'interruption d'horloge choisit un nouveau processus à exécuter. Ce choix peut être basé sur des algorithmes de planification de processus qui déterminent quel processus doit être exécuté ensuite en fonction de priorités, de quotas de temps, etc.

5. **Lancement du nouveau processus** : Une fois le nouveau processus sélectionné, le traitant de l'interruption d'horloge charge l'état sauvegardé de ce processus, y compris les valeurs des registres et l'emplacement du compteur de programme, et lance l'exécution de ce processus. Le processeur reprend alors l'exécution à partir du point où le nouveau processus a été interrompu la dernière fois.

RQ :

- `Le traitant de l’interruption horloge` a pour rôle l’exécution des trois opérations :
  - Rangement des registres
  - Choix d’un nouveau processus
  - Lancement du nouveau processus

2. Les sémaphores :

a.définition :

- Les sémaphores sont un mécanisme de synchronisation utilisé pour gérer l'accès à des ressources partagées ou pour coordonner l'exécution de plusieurs processus ou threads.
- Un sémaphore s est l’ensemble d’un compteur entier noté s.c et d’une file d’attente s.f
- Deux opérations qui sont exécutées en exclusion mutuelle
  - P(s) s.c--; si s.c<0 alors le processus qui exécute P est bloqué dans s.f
  - V(s) s.c++ ; si s.c<=0 alors activer un processus de s.f

b. Propriétés des sémaphores :

Les propriétés des sémaphores :

1. **Valeur initiale d'un sémaphore** (c0 >= 0) : Lorsqu'un sémaphore est créé ou initialisé, il reçoit une valeur initiale, généralement indiquée comme c0. Cette valeur initiale représente la quantité de ressources disponibles ou l'état initial de la ressource ou de la file d'attente contrôlée par le sémaphore. Un sémaphore ne peut pas avoir une valeur initiale négative.

2. **Valeur positive du sémaphore (s.c > 0)** : Si la valeur du sémaphore (s.c) est strictement positive, cela signifie qu'il y a des ressources disponibles pour être utilisées. Plus précisément, s.c représente le nombre de processus ou de threads qui peuvent exécuter avec succès une opération P (attente) sans se bloquer. Chaque exécution de P décrémente la valeur du sémaphore. Si s.c est supérieur à zéro, cela signifie que les ressources sont disponibles, et les processus peuvent les acquérir sans être mis en attente.

3. **Valeur négative du sémaphore (s.c < 0)** : Si la valeur du sémaphore (s.c) est négative, cela indique qu'il y a des processus en attente pour acquérir des ressources. Plus précisément, s.c représente le nombre de processus actuellement en attente (bloqués) pour effectuer une opération P. Chaque fois qu'un processus exécute P et ne peut pas acquérir de ressource (car s.c <= 0), il est mis en attente jusqu'à ce qu'une ressource devienne disponible.

Ces propriétés permettent de contrôler l'accès concurrent à une ressource partagée. Lorsqu'un processus souhaite acquérir une ressource, il utilise une opération P. Si s.c est positif, le processus obtient la ressource et décrémente s.c. Sinon, s.c devient négatif, indiquant que le processus est en attente. Lorsqu'un processus libère une ressource à l'aide de l'opération V, s.c est incrémenté, ce qui peut débloquer un processus en attente.

Cette gestion des valeurs positives et négatives du sémaphore permet de coordonner efficacement l'accès à des ressources partagées et de gérer les files d'attente de processus en attente d'accès à ces ressources. Elle est couramment utilisée dans la programmation concurrente et les systèmes d'exploitation pour éviter les problèmes de concurrence et de blocage.

c. exmples :

- **Eécriture->Lecture :**

```
s1 :sémaphore
    C0:= 0

Processus P1 :
Calcul ; -- écrire
V(s1) ; -- réveil

Processus P2 :
P(s1) ; -- wait
Traitement ;    -- lécture
```

RQ :

- cecei marcher car C0 = 0 si non la condition de syncronisation ne sera pas réspecte .

- **Rendez vous de deux processus :**

RQ :Exclusion mutuelle par sémaphore

- Sémaphore mutex vi :1
  - P(mutex) ; -- pour l'accés a la section critique
  - Section critique -- séction critiique
  - V(mutex) ; -- débloquag de processus en attente

d. Pb de Interblocage :

- **Pb:**

```
mutex1 := 1
mutex2 := 1

-- processus 1 :
P(mutex1);
P(mutex2);
V(mutex1);
V(mutex2);

-- processus 2 :
P(mutex2) ;
P(mutex1) ;
V(mutex1) ;
V(mutex2) ;
```

- **Explication :**

Supposons que vous avez deux mutex (sémaphores), `mutex1` et `mutex2`, ayant une valeur initiale de 1 chacun.

1. Le premier processus (Processus 1) exécute les opérations suivantes :
   - `P(mutex1)`: Il tente de verrouiller `mutex1`. Étant donné que la valeur initiale de `mutex1` est de 1, le verrou est acquis avec succès, et la valeur de `mutex1` devient 0.
2. En même temps, le deuxième processus (Processus 2) exécute les opérations suivantes :
   - `P(mutex2)`: Il tente de verrouiller `mutex2`. Étant donné que la valeur initiale de `mutex2` est de 1, le verrou est acquis avec succès, et la valeur de `mutex2` devient 0.
3. Le Processus 1 exécute `P(mutex2)`: Il tente de verrouiller `mutex2`. Cependant, `mutex2` est déjà verrouillé par le Processus 2 (la valeur de `mutex2` est de 0), ce qui signifie que le Processus 1 est bloqué en attente de `mutex2`.

4. En même temps, le Processus 2 exécute `P(mutex1)`: Il tente de verrouiller `mutex1`. Cependant, `mutex1` est déjà verrouillé par le Processus 1 (la valeur de `mutex1` est de 0), ce qui signifie que le Processus 2 est bloqué en attente de `mutex1`.

À ce stade, les deux processus sont bloqués en attente de l'autre mutex, ce qui crée une situation de deadlock (interblocage). Aucun des processus ne peut progresser, car ils sont tous les deux en attente de l'autre verrou.

- **Solution :**

Pour résoudre le problème d'interblocage , vous devez réorganiser la manière dont les mutex sont acquis et libérés par les processus pour garantir qu'aucun interblocage ne se produit.

```
// init
mutx1 := 1
mutx2 := 2

// Processus 1
P(mutex1);
P(mutex2);
// Section critique du Processus 1
V(mutex2);
V(mutex1);

// Processus 2
P(mutex1);
P(mutex2);
// Section critique du Processus 2
V(mutex2);
V(mutex1);
```

3. Réalisation de l’exclusion mutuelle :

a. Définition de l'exclusion mutuelle :

- **section critique SC :** une zone de code qui ne doit être exécutée que par un seul processeur à la fois .
- **l’exclusion mutuelle :** : un concept qui garantit qu'à un moment donné, un seul entité concurrente (processus ou thread) peut accéder a la SC .

b. Les #ts méthodes :

1. **masquage des interruptions : ca du monoprocessuer**

L'assurance de l'exclusion mutuelle avec le masquage des interruptions est une technique de synchronisation couramment utilisée dans les systèmes monoprocesseurs (un seul processeur) pour garantir que les sections critiques du code s'exécutent sans être interrompues par d'autres processus ou par des interruptions matérielles. Cette technique repose sur la désactivation temporaire des interruptions pendant l'exécution des sections critiques.

i. **Interruptions dans un système monoprocesseur** :

- Dans un système monoprocesseur, plusieurs processus ou threads s'exécutent en utilisant la commutation de contexte.

ii. **Passage d'un processus à un autre** :

- Le passage d'un processus à un autre peut avoir lieu lorsqu'une interruption se produit. Par exemple, un processus en cours d'exécution peut être interrompu pour traiter une interruption (comme une minuterie) ou pour exécuter un autre processus (commutation de contexte).

iii. **Solution - Masquage des interruptions** :

- Pour assurer l'exclusion mutuelle, on peut désactiver temporairement les interruptions pendant l'exécution des sections critiques. Cela signifie que tant que les interruptions sont désactivées, aucune autre interruption ne peut avoir lieu, et par conséquent, aucun passage de processus ne peut se produire.
- Ainsi, la section critique s'exécute de manière ininterrompue jusqu'à ce qu'elle soit terminée, garantissant l'exclusion mutuelle pendant cette période.

2. **attente active :**

- L'idée consiste à utiliser des variables en mémoire pour décrire l'état des processus (présence d'un processus en section critique par exemple).
- Quand un processus veut entrer en section critique, il teste ces variables d'état et se met dans une boucle d'attente s'il n'est pas autorisé à y entrer.
- par exemple la tentative de solution ci-dessous:
    - Soit c une variable booléenne, initialement à faux ;
    - l'idée est de donner à c la valeur `true` si un processus est en section critique.
    - Le programme est :

```
bool c = false ;

void entree_Sc()
{
    label :
    if(c)
    {
        goto label ;
    }
}


// Section critique
c = true ;

// .... -- code
```

```
void sortie_sc()
{
    c = false ;
}
```

3. **test & set : Cas du multiprocesseur**

- Cette méthode est basée sur une instruction matérielle spéciale qui réalise une opération atomique permettant de tester et de définir une valeur en même temps.
- Elle est utilisée pour coordonner l'accès de plusieurs processus à une ressource partagée, garantissant que deux processus ne peuvent pas accéder simultanément à cette ressource.
- Dans un système multiprocesseur, le Test-and-Set est utilisé lorsque le masquage d'interruptions (interruption masking) n'est pas suffisant pour assurer l'exclusion mutuelle.
- L'instruction spéciale Test-and-Set permet d'effectuer deux opérations en une seule instruction atomique :
    1. Tester la valeur de la variable (booléenne) spécifiée. Si la valeur est "fausse" (0), le test renvoie "faux" (0), sinon, il renvoie "vrai" (1).
    2. Définir la valeur de la variable à "vrai" (1) de manière atomique, quelle que soit la valeur précédente.
- Cette combinaison de test et de définition atomiques garantit que deux processus ne peuvent pas modifier la variable en même temps, ce qui permet de gérer l'accès exclusif à la ressource partagée.
- Un processus qui souhaite accéder à la ressource effectue un Test-and-Set sur une variable partagée:
    - S'il obtient "faux" en retour, il sait qu'il a obtenu l'accès à la ressource.
    - Sinon, il doit attendre.

```
A <- 0
E:
    Test&Set A
    jnz E

// section critique


Sortie:
     move #0,A
```

4. `Réalisation du noyau` : PedagOS :

Objectif : description du fonctionnement d'un système complet appelé PedagOS

4.1.Les composantes de La machine POP : Petit Ordinateur Personnel

- **UC** : processeure Monoprocesseur
- **mémoire**
- **clavier**
- **écran**
- **disque**
- **registres généraux** : `reg`
- **mep:** c'est un registre qui stocke l'adresse prochiane de l'instruction qui serà exécute par le CPU .
- **Horloge:**
  - Un emplacement de mémoire noté `timer`
  - **les opérations :**
    - **Mise en route de l'horloge** = chargement dans timer d'une valeur positive
    - **Arrêt de l'horloge** : `timer <- 0` # masque des interruptions
    - **Décrémenté** : chaque milliseconde
    - **Déclenche une interruption** : si `timer = 0`

4.2 Les Modes de notre SE :

- **Mode Noyau (Mode Maître) :** Le mode noyau est le mode d'exécution qui donne au système d'exploitation un contrôle complet sur les ressources matérielles de l'ordinateur.
- **Mode Utilisateur (Mode Esclave) :** Le mode utilisateur est le mode d'exécution dans lequel les applications utilisateur s'exécutent. Dans ce mode, les applications n'ont qu'un accès limité aux ressources matérielles et ne peuvent pas effectuer d'opérations privilégiées directes. Elles doivent faire des appels système pour demander au noyau d'effectuer des opérations en leur nom.

4.3 Les interruptions :

- Vecteur d'interruption :
  - Contient les mots d'état des traitants
- Sauve le `mep` dans un emplacement fixe noté `ancien_mep`
- Charge dans `mep` l'entrée du vecteur d'interruption qui correspond à la cause de l'interruption
- On posséde une Instruction de retour d'interuption : `rti`

- `Masquage-démasquage` :
    - **Masquage** : Le masquage des interruptions est un mécanisme qui désactive temporairement les interruptions matérielles sur un processeur ou un cœur de processeur. Lorsque les interruptions sont masquées, le processeur ignore les demandes d'interruption et continue d'exécuter son programme actuel.
    - **Démasquage des interruptions** : Le démasquage des interruptions est le processus inverse du masquage. Lorsque les interruptions sont démasquées, le processeur redevient sensible aux demandes d'interruption. Cela signifie que les interruptions en attente d'être traitées seront maintenant traitées.
- `Appel système` :
    - pour le passage du `Mode escalve` au `mode maitre` : le SE lance une interruption
    - pour le retourt du `mode maitre` au `mode esclave` : le SE appéle l'instruction `rti`.

4.4. **Noyau de processus :**

- les composantes de la noyau de notre mini-os :

a. **États des processus :**

- **états :**
    - **Élu** : ce processus s'exécute
    - **Éligible** : attente de l'UC
    - **Bloqué** : attente dans une file de sémaphore
- **Transitions entre états :**
    - **Élu ---> Bloqué :** si P(sémaphore.cpt) < 0
    - **Bloqué ---> Éligible :** si un autre processus applique la fonction V sur le sémaphore .

b. **structure d'un processus :**

- Pour chaque processus, on conserve un contexte :
    - Registres généraux `reg`
    - Mot d 'état : `psw` l'instruction prochaine à éxecuter
    - Indicateurs `etat` : l'état d'un processus
    - Priorité `prio` : son ordre de priorité pour l'ordonnacement
    - Liens de chaînage
    - Chaque processus est identifié par un `pid` :
        - ona : `pid->reg`, `pid->prio` ...etc

```
struct descripteur_de_processus
{
    reg ; //registres généraux
    psw ; // mot d'état de
    programme
    etat ; // élu, éligible ou bloqué
    prio ; // priorité
    liens ;
}
```

c. **strucuture d'un sémaphores:**

- Pour chaque sémaphore :
    - Valeur du sémaphore `cpt`
    - Pointeur vers la file d'attente file `file`.
- Chaque sémaphore est identifié par un `sid` :
    - on les accées suivant : `sid->cpt`, `sid->file`

```
struct descripteur_de_semaphore
{
    cpt ; // valeur du semaphore
    file ; // file du semaphore
}
```

d. **files d'attente :**

- Une file d'attente par sémaphore
- Une file d'attente de l'UC
- Opérations standards sur une file `f` :
    - `entrer (pid,f)` // ajouter le processus pid a f .
    - `premier (f)` -> pid // return le tete de la file .
    - `sortir (f)` -> pid // return le queue de la file .
    - `vide (f)` -> booléen // return true si la file est vide .

e. **Allocation de l'unit centrale :**

- **ordonnaceur** : qui sera charger d'allouer l'UC et de decider à chaque instant le processus qui a le droit d'étre exécuter .
- le code de l'ordonnanceur se fonctionnne qq soit le mode Avec ou sans réquisition(on intermope le processus mais si il n'est pas d'accord).
- **Méthode du tourniquet :** (**quantum d'UC**, **f_eligibles**)
    - pour la gestion de l'UC `ordonnacment` on choisit la méthode de tourniquet
    - **Méthode du tourniquet :**
        - C'est une technique de gestion de processus qui permet de partager équitablement le temps CPU entre plusieurs processus en cours d'exécution
        - **fonctionnement :**
            - **Sélection des processus :**
                - Les processus prêts à s'exécuter sont mis dans une file d'attente circulaire `f_eligibles`.
                - **Le tourniquet** choisit le premier processus dans la file pour l'exécution.
            - **Exécution :**
                - Le processus sélectionné est autorisé à s'exécuter pendant une certaine période de temps appelée **quantum d'UC**.
                - Ce quantum est généralement court, typiquement de quelques millisecondes.
            - **Commutation :** Une fois que le quantum d'UC est écoulé ou que le processus se bloque (par exemple, en attendant une entrée utilisateur), le tourniquet interrompt ce processus et le met en fin de file d'attente.
            - **Sélection du processus suivant :** Le tourniquet sélectionne ensuite le processus suivant dans la file d'attente et lui alloue le CPU pour un quantum d'UC. Ce processus est également interrompu après la fin du quantum ou s'il se bloque.
            - **Répétition :** Ce processus de sélection, d'exécution et de commutation se répète en continu, ce qui garantit que chaque processus obtient sa part équitable du temps CPU.
        - **les avantages :**
            - L'avantage principal de cette méthode est qu'elle assure une utilisation équitable du CPU entre les processus actifs.
            - Elle est simple à mettre en œuvre et convient bien aux environnements multitâches.
        - **les inconvientées :**

- elle n'est pas efficace en termes de performance, car les commutations fréquentes entre les processus introduisent un certain surcoût en termes de temps de commutation.

4.5 : **code complet : en Pseudo C**

```
entrer (pid,f); // ajouter le processus pid a f .

premier (f) -> pid ; // return le tete de la file .

sortir (f) -> pid  ;// return le queue de la file .

vide (f) -> booléen  ;// return true si la file est vide .

struct descripteur_de_processus
{
    reg ; //registres généraux
    psw ; // mot d'état de
    programme
    etat ; // élu, éligible ou bloqué
    prio ; // priorité
    liens ;
}

struct descripteur_de_semaphore
{
    cpt ; // valeur du semaphore
    file ; // file du semaphore
}

// Changement d'état

void ranger_proelu()
{
    proelu->reg =reg ;
    proelu->psw = ancien_mep ;

    /* le mot d'état du
    programme interrompu n'est plus dans mep, mais
    a été rangé dans ancien_mep par le matériel

    */
}


void lancer_exec(proelu)
{
    reg = proelu->reg ;
    ancien_mep = proelu->psw ;
    rti ; //relance l'exécution à partir de ancien_mep
```

```
}


// Traitant de l'interruption horloge

void traitant_it_horloge ()

{
    ranger_proelu() ; /* sauvegarde registres et mep */

    proelu->etat = eligible ;

    entrer (proelu,f_eligibles) ;

    lancer_processus_suivant() ;
}



// Définition et lancement du  prochain processus

void lancer_processus_suivant()
{
    proelu = sortir (f_eligibles) ;
    proelu->etat = elu ;
    lancer_horloge(quantum) ;
    lancer_exec(proelu) ;
}



//  Traitant de l'appel système

void traitant_svc()
{
    // seuls P et V sont possibles
    switch (r0) // r0 code de l'appel système
    {
    case CODE_P : executer_p (r1) ; break ;
    case CODE_V : executer_v (r1) ; break ;
    default : exec_erreur ;
    }
}



// Traitement de P(s)
void executer_p(s)
{
    s->cpt -- ;

    if (s->cpt >=0 )
    {
        rti ;
```

```
    }
    else
    {
        ranger_proelu() ;

        proelu->etat = bloque ;

        entrer (proelu,s->file) ;

        lancer_processus_suivant() ;
    }
}


// Traitement de V(s)

void executer_v(s)
{
    ranger_proelu() ;

    proelu->etat = eligible ;

    entrer (proelu,f_eligibles) ;

    s->cpt ++ ;

    if (s->cpt <=0 )
    {
        paux = sortir (s->file) ;

        paux->etat = eligible ;

        entrer (paux, f_eligibles) ;
    }

    lancer_processus_suivant() ;
}



int main(void)
{
    // Initialisation :
    vect_int[SVC]=(&traitant_svc,maître, masqué) ;

    timer = 0 ;

    vect_int[HORLOGE]=(&traitant_it_horloge, maître, masqué) ;

    /* il faut initialiser aussi les files de processus */


    return 0 ;
}
```