Université Mohammed V- Rabat
École Mohammadia d'Ingénieurs
Département Génie Informatique

Filière Génie Informatique et Digitalisation

SYSTÈMES D'EXPLOITATION 2

Mr N. EL Faddouli (elfaddouli@emi.ac.ma)

2024-2025

1

Plan du cours

- Introduction et Rappels
- Gestion des processus: Synchronisation avec attente active
  - Définitions et rappels
  - Algorithmes avec attente active
- Gestion des processus: Synchronisation sans attente active
  - Les sémaphores
  - Les moniteurs
  - L'interblocage (Dead-lock)
- Gestion de la mémoire
  - Définitions et rappels
  - La mémoire virtuelle
- Les entrées/Sorties

2

Introduction et Rappel

Qu'est-ce qu'un SE ?

- Une couche logicielle, dont le rôle est de:

  - Gérer les ressources de la machine.

    Ressource : Tout ce qui est nécessaire à l'avancement d'un processus (processeur, mémoire, disques, périphériques, message d'un autre processus, réseau, …etc)

    Exemples:

    - Savoir quelles sont les ressources disponibles
    - Savoir qui utilise quoi, quand, combien, etc.
    - Allouer/Libérer les ressources efficacement.

  - Fournir aux programmes d'application une interface simplifiée avec les ressources sous forme de ***machine virtuelle***

    (*base pour le développement et l'exécution des programmes d'application.*)

Problématique

Afin que les programmes puissent s'exécuter de façon portable et efficace, il faut pouvoir gérer simultanément:

- La multiplicité des différentes ressources
- La complexité des composants de chacune d'elles, qui requiert la prise en compte de nombreux détails fastidieux, sources de bogues.

Exemple: Partage d'imprimante

Pour assurer un service d'impression sur une machine multi-utilisateurs, il faut:

- Verrouiller l'accès à l'imprimante afin que les flots de caractères à imprimer envoyés par les programmes ne s'entrelacent pas sur le papier.
- Gérer les tampons d'impression afin que les programmes puissent reprendre leur travail sans devoir attendre la fin de l'impression.

Buts d'un SE

Le problème : Gérer l'accès à des ressources coûteuses.

À tout instant, il faut:

- Connaître l'utilisateur d'une ressource donnée;
- Gérer l'accès concurrent à cette ressource;
- Pouvoir accorder l'usage (exclusif) à cette ressource;
- Éviter les conflits entre les programmes ou entre les usagers.

Buts d'un SE

Un SE a pour but de:

- Fournir un environnement où l'utilisateur puisse exécuter des programmes.
- Utiliser le matériel de façon efficace.
- Protéger le système et ses usagers de fausses manipulations.

Fonctionnalités d'un SE

Le SE fournit à l'utilisateur:

- Une vue uniforme des entrées/sorties;
- Une mémoire partageable;
- La gestion de fichiers et répertoires;
- La gestion des droits d'accès, sécurité, et du traitement des erreurs;
- La gestion des processus;
- La gestion des communications inter-processus.

Fonctionnalités d'un SE (suite)

En tant que gestionnaire de ressources, le SE doit permettre:

- D'assurer le bon fonctionnement des ressources;
- L'identification de l'usager d'une ressource;
- Le contrôle des accès;
- L'interruption d'une utilisation de ressource;
- La gestion des erreurs, …

Caractéristiques d'un SE

- Temps de réponse
- Fiabilité
- Sécurité

Gestion des processus

Synchronisation avec attente active

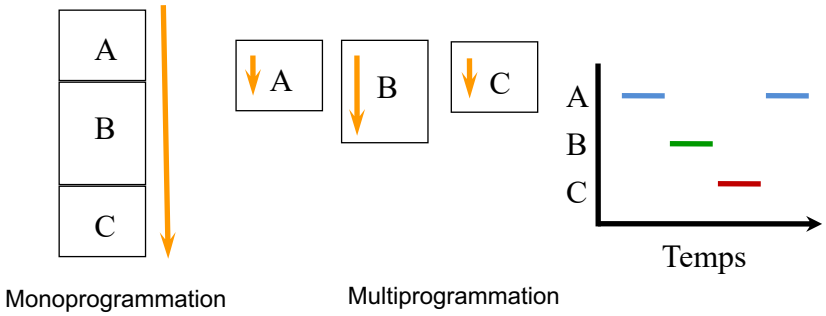
Processus

- **Processus**: c'est un programme ou une partie en exécution

  (*ensemble d'instructions réalisant une fonction donnée*).
- Chaque processus possède un PCB (*Process Control Block*)
  - Registres de CPU (*pointeur pile, pointeur code/compteur ordinal, pointeur données,…*)
  - Numéro du processus (PID), PPID, état, priorité
  - Liste des fichiers ouverts, liste des périphériques E/S utilisés,
  - Temps CPU utilisé,
  - …
- Les PCBs sont rangés dans une table de processus (***LinkedList*** dans la ***RAM***)
- Un processus est détruit en général à la fin de son exécution (*peut être détruit à la demande d'un autre processus ayant ce droit comme le père*)
- Lorsqu'un processus est détruit, son PCB et ses ressources sont libérés

Processus

- Processus: "Un programme qui s'exécute"
- Plusieurs processus peuvent être exécutés simultanément (*pseudo parallélisme*)



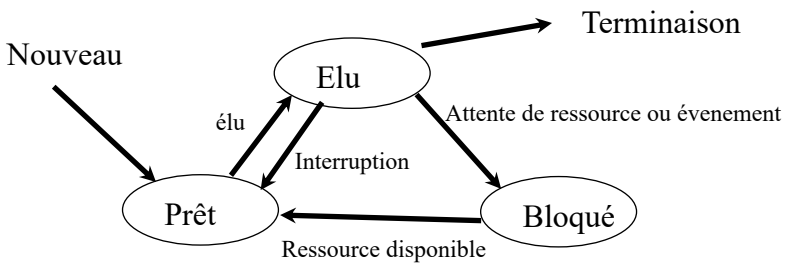
Etats des Processus

Etats d'un processus:

- **Prêt** : Suspendu provisoirement pour permettre l'exécution d'un autre processus.
- **Elu** : En cours d'exécution (Actif)
- **Bloqué**: Attend un événement (*libération d'une ressource, E/S, ….*) pour pouvoir continuer.

Etats des processus

- Changement d'état basé sur des événements



Synchronisation des processus

• Lorsque des processus s'exécutent en parallèle ou en pseudo-parallèle, ils peuvent:

- Être concurrents pour accéder à des ressources partagées:

S'ils sont exécutés sans précaution, on risque d'avoir des résultats imprévisibles.

→Assurer leur **synchronisation**.

- Échanger des informations, coopérer selon un certain **protocole** (*producteur-consommateur,…*) par l'intermédiaire de ressources partagées

→ Assurer la **communication** inter-processus

Gestion des processus

Multiprogrammation: présence simultanée en mémoire principale de plusieurs programmes.

Ordonnanceur (scheduler): se charge de l'ordonnancement des processus.

Section critique: une partie du code qui utilise une ou plusieurs ressources partagées.

Exclusion mutuelle: si une ressource est en cours d'utilisation par un processus, les autres processus ne peuvent pas y accéder, càd un seul processus à la fois est autorisé à utiliser la ressource partagée. (→*1 seul processus en S.C*)

Blocage mutuel (**Interblocage** ou **Deadlock**): c'est la situation dans laquelle aucun processus ne peut exécuter sa section critique: chaque processus attend indéfiniment que la section critique soit libérée.

Problème de Synchronisation

n processus {P_1, P_2, …,P_n} en concurrence pour utiliser des ressources (données, …) partagées.

Section critique: Segment de code dans lequel le processus peut accéder aux ressources partagées. Elle doit être exécutée de façon **atomique**.

Problème: Assurer que, quand un processus exécute sa section critique, **aucun** autre processus n'est autorisé à exécuter la sienne.

Exemples:

1) Plusieurs processus réalisant, en parallèle, l'impression d'un texte sur une imprimante qui constitue une ressource critique à utiliser par un seul des processus ➔ Allocation

2) Mise à jour d'un compte à l'aide de deux classes de processus:

- Processus créditeurs
- Processus débiteurs

Exemple: Mise à jour d'un compte bancaire

Processus créditeur

```
…
(1) S =Lecture_Solde();
(2) S = S+ 100
(3) Ecriture_Solde(S);
…
```

Processus débiteur

```
…
(1) M=Lecture_Solde();
(2) M= M- 50
(3) Ecriture_Solde(M);
…
```

solde du compte = 1000

En séquentiel: Solde valide=1050

Processus créditeur 1

(1)

(2) ……

(3)

Processus débiteur 1

(1)

(2)

(3)

→ Solde invalide = 1100

Exemple: Mise à jour d'un compte bancaire *(Suite)*

Exécution de deux processus débiteurs

solde du compte = 1000	**En séquentiel:** Solde valide=900

Processus débiteur1	Processus débiteur 2	
(1)		
(2)		
	(1)	→Solde invalide=950
	(2)	
(3)		
	(3)	

➔ **Si l'accès est concurrent, l'exécution n'est pas cohérente et les données (solde) sont invalides.**

➔ **Adopter une solution pour assurer l'exclusion mutuelle**

Conditions de validité des solutions de Synchronisation

- **Exclusion mutuelle:**

  Si un processus exécute sa section critique, ***aucun*** autre processus ne peut exécuter la sienne.

- **Inter-blocage**:

  Un processus qui n'est pas dans sa section critique ne peut pas bloquer les autres pour exécuter la leur.

- **Famine**:

  Aucun processus ne doit attendre trop longtemps (indéfiniment) avant d'entrer en section critique.

- **Aucune hypothèse** n'est faite sur les vitesses relatives des processus, ni sur le nombre de processeurs.

Solutions:

1. Logicielles
2. Matérielles

Protocole d'exécution de la Section Critique

• L'exécution de la section critique de plusieurs processus doit suivre le protocole suivant.

Processus P1;	Processus P2;	……..
{	{	
Section non critique;	*Section non critique;*	
{demande_entrée_section;}	{demande_entrée_section;}	
SC;	SC;	
{sortie_section;}	{sortie_section;}	
Section non critique;	*Section non critique;*	
}	}	

Solutions Logicielles (1)

Utilisation d'une variable Verrou

Verrou: Boolean

Verrou = 0

{entrée_section;}:

```
(1) While (verrou == 1);  // Attente
(2)  verrou = 1;          // Verrouillage
```

{sortie_section;}:

```
verrou = 0;          // Déverrouillage
```

Solution Fausse: la séquence $(1)_{P1}$; $(1)_{P2}$;$(2)_{P1}$;$(2)_{P2}$ autorise l'entrée simultanée contrairement au but recherché.

Solutions Logicielles (2)

Utilisation d'une variable verrou *(suite)*

Amélioration: verrou = 0;

(1) While (**verrou** != 0); *// Attente*

(2) **verrou** = pid; *// Verrouillage*

(3) if (**verrou** == pid)

Section critique

verrou=0;

La séquence précédente deviendra:

$(1)_{P1}$; $(1)_{P2}$;$(2\ ,\ 3)_{P1}$;$(2\ ,\ 3)_{P2}$

Inconvénients: Attente active , problème (conflit) d'accès à la variable partagée.

Solutions Logicielles (3)

Alternance: *// Lecteur et rédacteur partageant une case*

Tour= 1; *// C'est le rédacteur qui commencera en premier*

Processus 1 /*Lecteur*/

```
While (True)
{ While (Tour==1) ; // Attente
     SC_Lecteur(); // SC exécutée si Tour=0. Exemple: Lire(B) où B est un objet
  Tour=1;        // sortie de la SC, c'est le tour du rédacteur
   Section_Non_Critique(); // Exemple: Traitement de l'objet B
}
```

Processus 2 /*Rédacteur*/

```
While (True)
{ While (Tour==0) ; // Attente
     SC_Redacteur();    // SC exécutée si Tour=1. Exemple: Ecrire(B)
  Tour=0;               // sortie de la SC, c'est le tour du lecteur
  Section_Non_Critique();}
```

Solutions Logicielles (4)

Algorithme de Dekker (1965): Deux variables partagées: tour, demande

```
Type   identité = (P1, P2);
Var   tour: identité   INIT(P1);  // tour initialisée à P1 → P1 peut entrer en SC
 //Le tableau demande initialisé à Flase → Les deux processus ne veulent pas entrer en SC
 demande: Array[P1..P2] of Boolean     INIT(2:FALSE);
Procédure  Entrer_SC (A_Moi, A_Lui: identité)
Begin
(1) demande[A_Moi] := TRUE;            // je veux entrer
(2) WHILE demande[A_Lui] DO Begin  // tant que l'autre le veut aussi…
(3)  IF  tour != A_Moi  THEN  Begin  // si ce  n'est pas mon tour…
(4)     demande[A_Moi] := FALSE;   // je renonce …
(5)     WHILE  tour!=A_Moi DO  NOP ;  // jusqu'à ce que ce soit mon tour
(6)     demande[A_Moi] := TRUE;  // …puis je réaffirme ma demande
    END {if}
  END {while}
End;
```


Solutions Logicielles (5)

Algorithme de Dekker *(suite)*

```
Procédure  Sortir_SC (A_Moi, A_Lui: identité)
Begin
      demande[A_Moi] := FALSE;  // je ne veut pas entrer
      tour := A_Lui;   // C'est le tour de l'autre processus
End;
```

P1 exécute: Entrer_SC(P1,P2);
SC
Sortir_SC(P1,P2);

P2 exécute: Entrer_SC(P2,P1);
SC
Sortir_SC(P2,P1);

Solutions Logicielles (6)

Algorithme de Peterson (1981)

Exclusion mutuelle entre deux processus numérotés 0 et 1

Var Tour: 0..1; *// variable de tour **non initialisée***

Intéressé: ARRAY[0..1] Boolean; *// intention d'entrer dans la SC*

Initialisation: Interéssé [0] = Interéssé [1] = FALSE

Processus i: // i= 0 ou 1

(1) Intéressé [i] :=TRUE; *//veut entrer*

(2) Tour:=i; *// est passé en dernier*

(3) while (Tour!=i AND Intéressé[(*i*+1) mod 2]=TRUE); //Attendre son tour

(4) *Section_Critique();*

(5) Intéressé [i]:=FALSE;

Processus i

	Intérêt de (*i*+1) mod 2 : **True**	**False**
Tour = i	**Ok**	**Ok**
≠ i	**Non**	**Ok**

Solution matérielle Test and Set (TAS)

L'instruction **TAS**:

Function **TAS** (b boolean): Boolean

```
Begin
   TAS := b;
   b := TRUE;
End
```

C'est une opération **indivisible** (atomique) dont l'argument est une variable qui sera mémorisée dans un registre.

Utiliser TAS pour entrer en SC: Initialisation verrou= **FALSE**

```
While TAS(&verrou); //Attente
      Section_Critique();
verrou :=FALSE;     // Sortie de la section critique
```

➔ **Problème**: Attente active

Les primitives Sleep & Wakeup (1)

- Au lieu de l'attente active (= consommation de la CPU) le processus s'endort (se bloque): primitive **sleep** *(Linux: pause())*
- Un processus peut réveiller un autre processus :

  primitive **wakeup** *(Linux: kill (pid, SIGCONT))*

Exemple: Le problème de producteur-consommateur à **n** cases

Le producteur :
- peut remplir une case libre s'il y en a, sinon il s'endort en attente d'être réveillé par le consommateur.
- après avoir produit (*remplissage d'une case*), il réveille le consommateur si une seule case est pleine.

Le consommateur :
- peut vider une case pleine s'il y en a, sinon il s'endort en attente d'être réveillé par le producteur.
- après avoir consommé, il réveille le producteur si une seule case est vide.

Les primitives Sleep & Wakeup (1)

```
/* Un tampon de N cases est partagé entre les deux processus */
#define  N  100
int compteur = 0; /* Nombre courant d'objets */
Producteur ()
{ objet  a;
  while (1){
    Produire_Objet(&a);
    if( compteur == N) sleep ();
          Stocker_Objet(a);
    compteur ++;
    if( compteur == 1)
      wakeup ( consommateur );
          }
}
```

```
Consommateur ()
{ objet  a;
  while (1){
    if( compteur ==0) sleep ();
          Retirer_Objet(&a);
    compteur --;
    if( compteur == N -1)
      wakeup ( producteur );
    Consommer_Objet(a);
          }
}
```

Pas d'attente active.

Problème si perte du signal wakeup (arrive avant sleep).

Exercices