Principes des systèmes d'exploitation

Support de transparents
IUP MIAGE
Faculté de sciences -UNSA

N. Le Thanh

février 1995

Plan du cours

I- Partie 1 : Systèmes centralisés

I.1- Introduction

- qu'est ce qu'un système d'exploitation ?
- deux mot sur l'histoire
- principes des systèmes d'exploitation
- structure d'un système d'exploitation

I.2- Les processus

- introduction
- communication inter-processus
- problèmes classiques en communication inter-processus
- Ordonnancement des processus

I.3- La gestion de la mémoire

- gestion directe de la mémoire
- va-et-vient

Plan du cours

- mémoire virtuelle
- modélisation des algorithmes de pagination
- conception des systèmes paginés
- segmentation

I.4- Le système de fichiers

- fichiers
- catalogues
- mise en oeuvre du système de fichiers
- sécurité
- mécanismes de protection

I.5- Les entrées/Sorties

- principes du matériel
- principes du logiciel
- disques

Plan du cours

- horloges
- terminaux

I.6- Les interblocages

- ressources
- interblocage
- mécanismes de résolution

I.7- Etudes de cas

- Unix
- MS-Dos

Plan du cours

II- Partie 2 : Systèmes distribués

II.1- Introduction

- objectifs
- concepts matériels
- concepts logiciels
- bases de la conception des systèmes distribués

II.2- Communication dans les systèmes distribués

- couches de protocoles
- modèle client-serveur
- appels de procédures à distance
- communication de groupe

II.3- Synchronisation dans les systèmes distribués

- synchronisation d'horloge
- exclusion-multuelle

Page 5

Plan du cours

- Algorithmes d'élection
- transactions atomiques
- interblocage dans les systèmes distribués

II.4- Processus et processeurs

- processus légers
- modèles de systèmes
- allocation de processus
- ordonnancement dans les systèmes distribués

II.5- Les systèmes de fichiers distribués

- conception d'un système de fichiers distribué
- implémentation d'un système de fichiers distribués
- tendances des systèmes de fichiers distribués

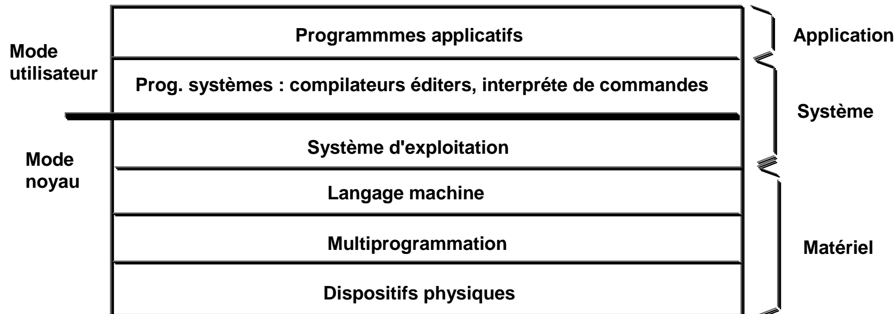
II.6- Etude de cas

- Windows-NT

Page 6

I- Partie 1 : systèmes centralisés
I.1- Introduction

I.1.1- Qu'est-ce qu'un système d'exploitation ?

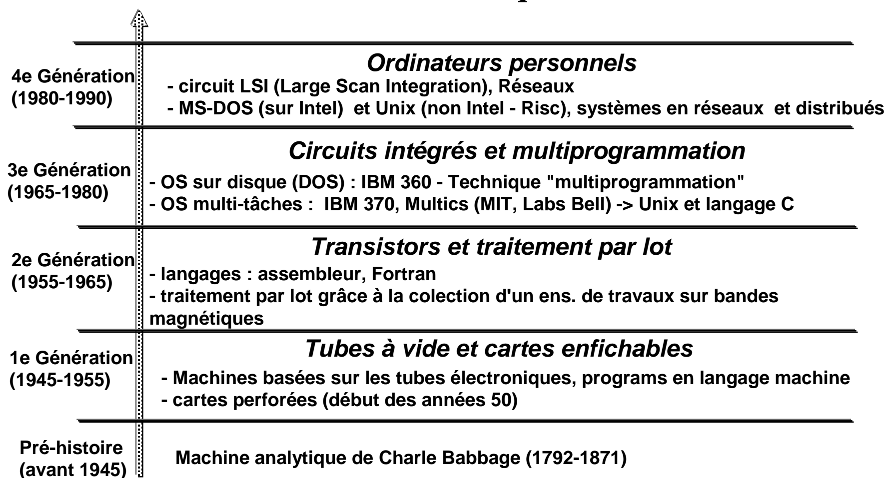


Deux fonctions principales
- MACHINE VIRTUELLE (MACHINE ÉTENDUE)
- GESTIONNAIRE DE RESSOURCES

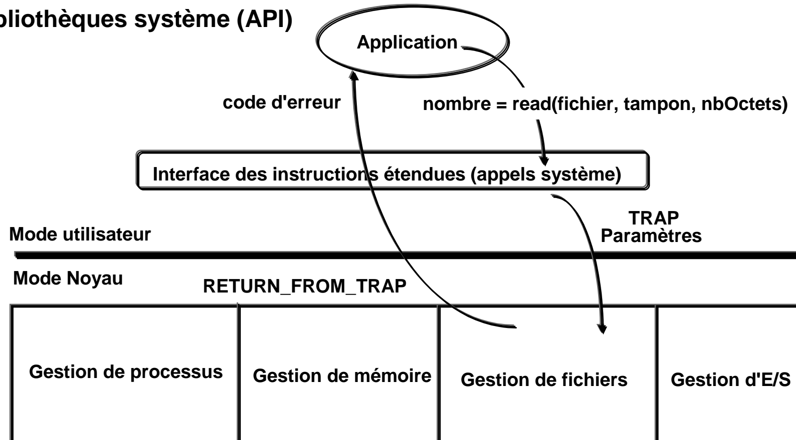
Page 7

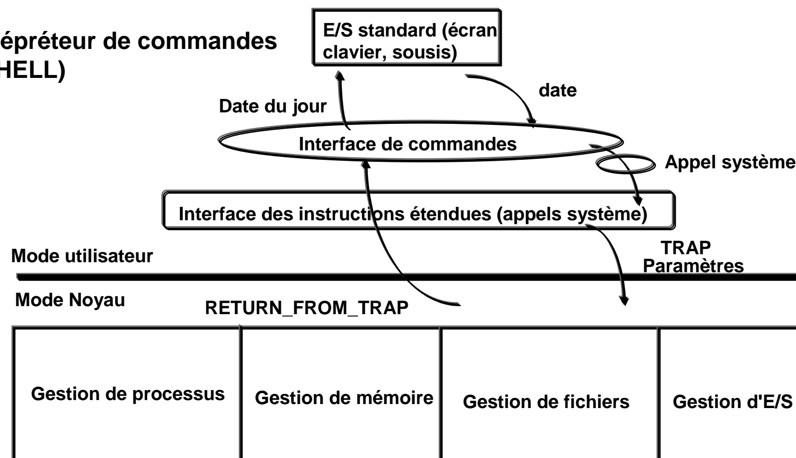
I- Partie 1 : systèmes centralisés
I.1- Introduction

I.1.2- Historique



Page 8

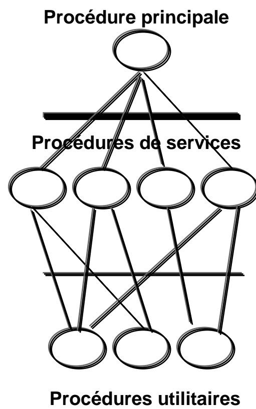

I- Partie 1 : systèmes centralisés
I.1- Introduction
I.1.3- Principes des systèmes d'exploitation
Bibliothèques système (API)
Application
code d'erreur
nombre = read(fichier, tampon, nbOctets)
Interface des instructions étendues (appels système)
Mode utilisateur
TRAP
Paramètres
Mode Noyau
RETURN_FROM_TRAP
Gestion de processus
Gestion de mémoire
Gestion de fichiers
Gestion d'E/S
Page 9

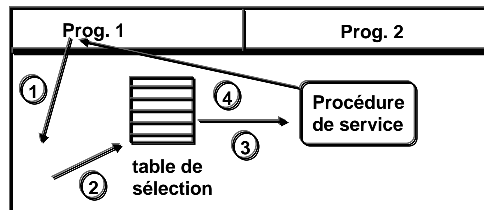

I- Partie 1 : systèmes centralisés
I.1- Introduction
I.1.3- Principes des systèmes d'exploitation
Intépréteur de commandes
(SHELL)
E/S standard (écran
clavier, sousis)
date
Date du jour
Interface de commandes
Appel système
Interface des instructions étendues (appels système)
Mode utilisateur
TRAP
Paramètres
Mode Noyau
RETURN_FROM_TRAP
Gestion de processus
Gestion de mémoire
Gestion de fichiers
Gestion d'E/S
Page 10

I- Partie 1 : systèmes centralisés
I.1- Introduction

I.1.4- Structure d'un système d'exploitation

I.1.4.1- Systèmes monolithiques





Structure = absence de structure

Le grand désordre !

I- Partie 1 : systèmes centralisés
I.1- Introduction

I.1.4- Structure d'un système d'exploitation

I.1.4.2- Systèmes à couches

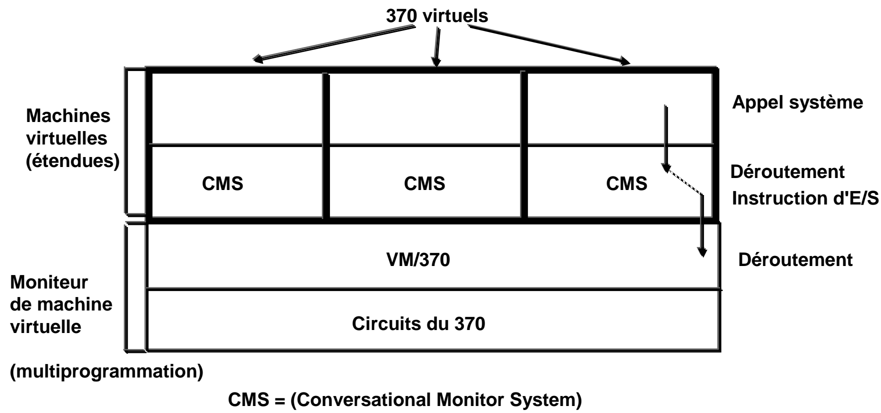
5	L'opérateur
4	Les programmes utilisateurs
3	Gestion d'Entrées / Sorties
2	Communication opérateur-processus
1	Gestion de la mémoire
0	Allocation de processeur et multiprogrammation

Les couches des systèmes d'exploitation THE (Dijkstra - 1968 - Pays-bas)

I- Partie 1 : systèmes centralisés
I.1- Introduction

I.1.4- Structure d'un système d'exploitation

I.1.4.3- Machines virtuelles

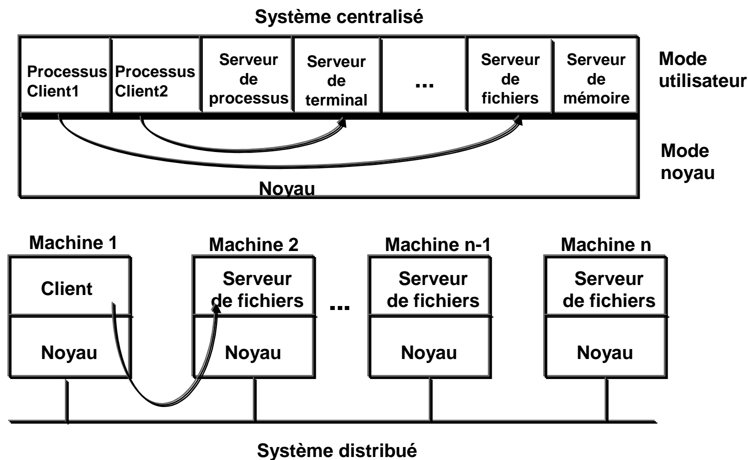


CMS = (Conversational Monitor System)

I- Partie 1 : systèmes centralisés
I.1- Introduction

I.1.4- Structure d'un système d'exploitation

I.1.4.4- Modèle Client / Serveur



I- Partie 1 : systèmes centralisés
I.1- Introduction

Exercices

1- Quelles sont les deux fonctions principales d'un SE ?

2- Qu'est-ce que la multiprogrammation ?

3- Quest-ce que le traitement par lot et le SPOOL ?

4- Quels sont les différences entre le système multiprogramme avec le traitement par lot et multiprogramme avec le traitement en temps partagé ?

5- Laquelle des instructions suivantes ne devrait être autorisée qu'en mode noyau ?
- masquer toutes les interruptions
- lire la date du jour
- modifier la date du jour
- changer la partition de la mémoire

6- Pourquoi l'interpréteur de commandes (shell) ne fait pas partie du SE ?

7- Le modèle Client/Serveur est courant dans les systèmes distribués. Peut-il être mise en oeuvre sur un seul ordinateur ?

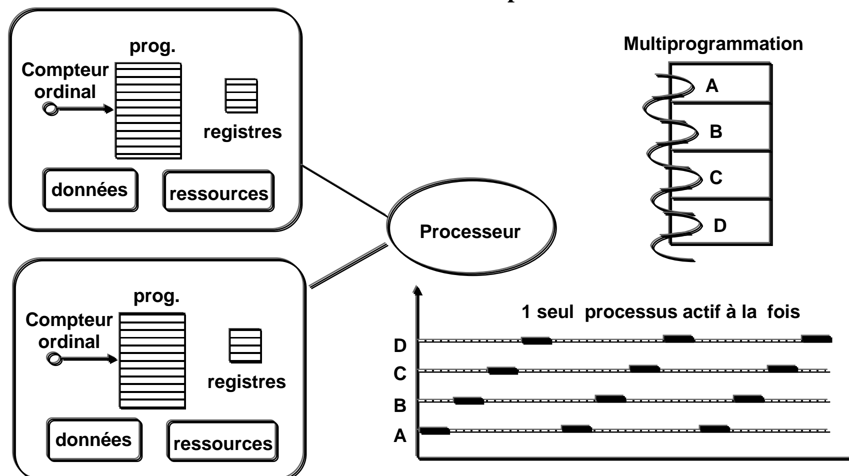
8- A quoi sert la table de processus ? est-elle nécessaire dans un ordinateur personnel qui n'exécute qu'un seul processus à la fois ?

Page 15

I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.1- Introduction aux processus

I.2.1.1- Modèle des processus



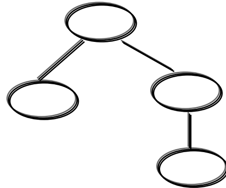
Page 16

I- Partie 1 : systèmes centralisés
I.2- Les Processus

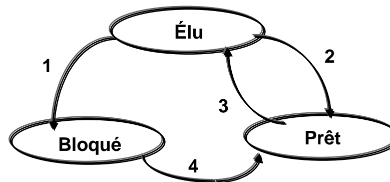
I.2.1- Introduction aux processus

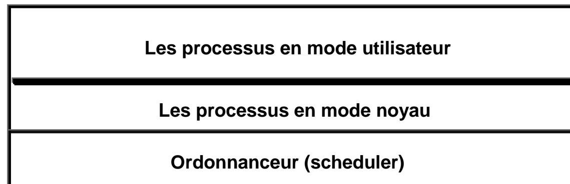
I.2.1.1- Modèle des processus

Hiérarchie de processus



Etats d'un processus



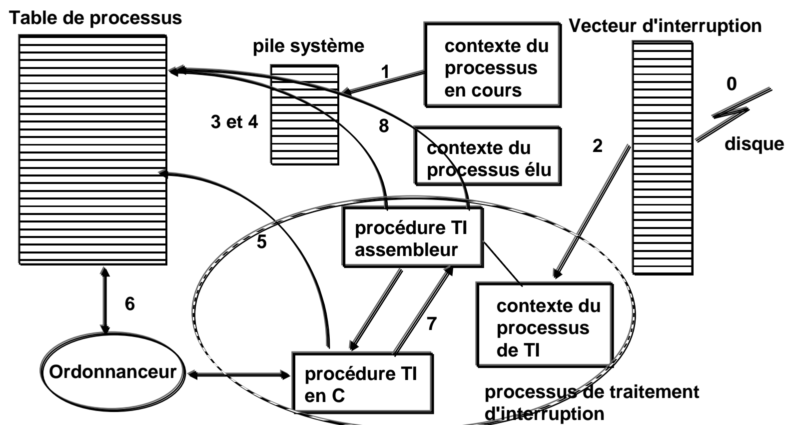


Page 17

I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.1- Introduction aux processus

I.2.1.2- La réalisation des processus sous Unix



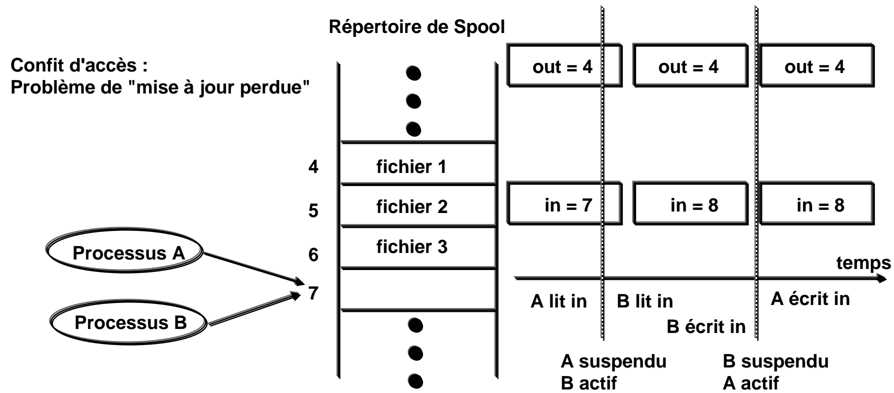
Page 18

I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.2- Communication inter-processus

(IPC = Inter-Process Communicaion)

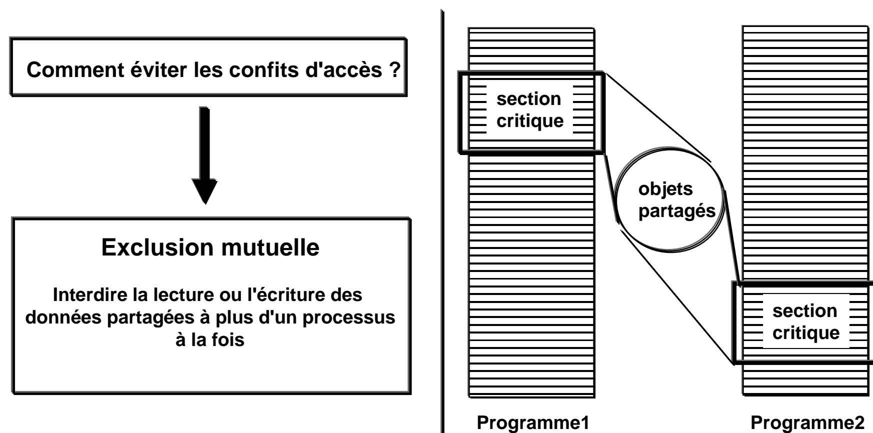
I.2.2.1- Accès concurrents



I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.2- Communication inter-processus

I.2.2.2- Sections critiques



I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.2- Communication inter-processus

I.2.2.3- Exclusion mutuelle par attente active

Le masquage des interruptions

Chaque processus masque les inter-ruptions avant d'entrer dans une section critique et les restaure à la sortie de la section

Les variables de verrouillage

Les processus partagent une variable commune (verrou) qui est initialisée à 0. Un processus doit tester le verrou avant d'entrer en section critique. Si le verrou vaut 0, il ne met à 1 et entrer en section critique, si non il attend qu'il repasse à 0

L'alternance

```
while (TRUE)
{
  while (tour != 0); /*attente*/
  section_critique ();
  tour = 1;
  section_non_critique
}
```

processus A

```
while (TRUE)
{
  while (tour != 1); /*attente*/
  section_critique ();
  tour = 1;
  section_non_critique
}
```

processus B

I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.2- Communication inter-processus

I.2.2.3- Exclusion mutuelle par attente active

Algorithme Peterson

```
#include "prototype.h"
#define FALSE        0
#define TRUE         1
#define N            2          /* nb de processus */

int tour;                       /* A qui le tour */
int interesse[N];               /* intitialiser à 0 */

void entrer_region(int process)  /* n° du processus 0 ou 1 */
{
  int autre                     /* n° de l'autre processus*/

  autre = 1 - process; /*l 'autre processus */
  interess[process] = TRUE;      /* on est intéressé */
  tour = process;                /* demande d'une entré */
  while (tour == process) && (interess[autre] == TRUE); /* attendre si l'autre est intéressé */
}

void quitter_region (int process)/* processus quitant */
{
  interesse[process] = FALSE;    /* sotir de la section critique */
}
```

I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.2- Communication inter-processus

I.2.2.4- Attente passive : primitives "dormir" et reveiller"

Problème d'attente active

A : priorité haute **B : priorité basse**

- A en attente (d'une opération d'E/S par exemple)
- B entre dans en section critique
- A passe à état prêt (opération d'E/S est terminée)
- A prend la main (priorité supérieure à B)
- A est bloquer dans la boucle car B n'est pas sorti de la section critique
- B ne sort jamais car A est à état actif

Problème Inversionde priorité

A et B sont bloqués

Principe d'attente passive

- Lorsque un processus est en état d'attente pour entrer en section critique, il se suspend et laisse la main aux autres processus

- Le processus sortant, doit reactiver le processus suspendu

- Primitive "Dormir" (SLEEP) : est un appel système qui suspend l'appelant en attendant qu'un autre processus le réveille

- Primitive "Réveiller" (WAKEUP) est un appel système pour réveiller un processus (en paramètre)

Page 23

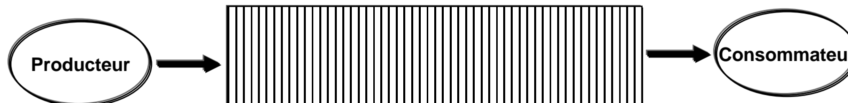
I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.2- Communication inter-processus

I.2.2.4- Attente passive : primitives "dormir" et reveiller"

Modèle du Producteur / Consommateur

Producteur



Consommateur

```
void producteur(void)
{
  int objet;
  while (TRUE)
  {
    produitr_objet(&objet);
    if (compteur == N) sleep();
    mettre_objet(objet);
    compteur = compteur + 1;
    if (compteur == 1) wakeup(consommateur)
  }
}
```

```
void consommateur(void)
{
  int objet;
  while (TRUE)
  {
    if (compteur == N) sleep();
    retirer_objet(objet);
    compteur = compteur - 1;
    if (compteur == N-1) wakeup(producteur);
    consommer_objet(objet);
  }
}
```

Page 24

I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.2- Communication inter-processus

I.2.2.4- Attente passive : primitives "dormir" et reveiller"

Problème du modèle du Producteur / Consommateur

Situation

- Le consommateur est mis en état attente par l'ordonnanceur au moment qu'elle a lu la valeur du compteur qui est à 0
- Le producteur est actif et produit un élément, il incrémente le compteur et reveiller le consommateur
- Le consommateur n'est pas en état dormi, donc l'appel WAKEUP est perdu
- Le consommateur est réactivé par l'ordonnaceur, il voit son compteur est à 0 et se suspend
- Le producteur continue de remplir le tampon et se suspend quand ceci est plein

Conséquence

Les deux processus dormiront pour toujour !!!

Page 25

I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.2- Communication inter-processus

I.2.2.5- Les sémaphores

SEMAPHORE

Un type d'entier qui permet de compter le nombre de réveils en attente

- un sémaphore a valeur 0 si aucun réveil n'a mémorisé et une valaur positive s'il y une ou plusieurs réveils en attente

Deux opérations atomiques indivisibles

DOWN (SLEEP généralisée)	UP (WAKEUP généralisée)
- décrémenter la valeur d'un sémaphore si cette derrière est supérieure à 0, puis poursuivre l'exécution normale - si la valeur du sémaphore est à 0, le processus appelant se met en attente	- incrémenter la valeur du sémaphore si cette derrière est supérieure à 0 - si la valeur est à 0, liberer un parmi les processus mis en attente sur ce sémaphore (la valeur du sémaphore reste 0)

Page 26

I.2.2- Communication inter-processus

I.2.2.5- Les sémaphores : solution pour le modèle P/C

```
semaphore mutex  = 1;  /* contrôle d'accès à la SC */
semaphore vide  =  N;  /* nb d'emplacements libres */
semaphore plein = 0;  /* nb d'emplacements occupés */
```

```
void producteur(void)
{
  int objet;

  while (TRUE)
  {
    produitr_objet(&objet);
    down(&vide);
    down(&mutex);
    mettre_objet(objet);
    up(&mutex);
    up(&plein);
  }
}
```

```
void consommateur(void)
{
  int objet;

  while (TRUE)
  {
    down(&plein);
    down(&mutex);
    retirer_objet(objet);
    up(&mutex);
    up(&vide);
    consommer_objet(objet);
  }
}
```


I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.2- Communication inter-processus

I.2.2.6- Les compteurs d'évènement

Type d'entier appellé Compteur_evt

Read(E) : donne valeur courante de E
Advance(E): incrémente de 1 (atomique)
Await(E,v): attend que E atteigne ou dépasse v

```
typedef int compteur_evt;

compteur_evt in = 0;
compteur_evt out = 0
```

```
void producteur(void)
{
  int objet, sequence = 0;

  while (TRUE)
  {
    produitr_objet(&objet);
    sequence = sequence +1;
    await(out, sequence - N);
    mettre_objet(objet);

    advance(&in);
  }
}
```

```
void consommateur(void)
{
  int objet, sequence = 0;

  while (TRUE)
  {
    sequence = sequence +1;
    await(out, sequence);
    retirer_objet(objet);

    advance(&in);
    consommer_objet(objet);
  }
}
```

I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.2- Communication inter-processus

I.2.2.7- Le moniteur

Modèle

- Un moniteur est un ensemble de procédures, de variables et de structures de données regroupées dans un module spécial

- Les processus externe peuvent appeller les procédures d'un moniteur mais ne peuvent pas accéder à la structure interne de données du moniteur

- propriété exclusion multuelle : un seul processus peut être actif dans un moniteur

- il dispose des variables de condition et deux opérations sur ces variables : WAIT pour mettre en attendre du processus appelant et SIGNAL pour réveiller un autre processus

Exemple

```
monitor exemple
  integer i;
  condition c;

  procédure producteur (x);
  ...
  end;

  procedure consommateur(x);
  ...
  end;

end moniteur;
```

Page 29

I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.2- Communication inter-processus

I.2.2.7- Le moniteur : solution Producteur/Consommateur

```
monitor ProducteurConsommateur
  condition plein, vide;
  integer compteur;

  procedure mettre;
  begin
    if compteur = N then wait(plein);
    mettre_objet;
    compteur := compteur + 1;
    if compteur = 1 then signal(vide);
  end;

  procedure retirer;
  begin
    if compteur = 0 then wait(vide);
    retirer_objet;
    compteur := compteur -1;
   if compteur = N -1 then signal(pliein);
  end;
 compteur := 0;
end monitor;
```

```
Procedure producteur;
begin
   while true do
   begin
     produire_objet;
     ProducteurConsommateur.mettre;
   end
end;


procedure consommateur;
begin
   while true do
   begin
     ProducteurConsommateur.retirer;
     utiliser_objet;
   end
end;
```

Page 30

I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.2- Communication inter-processus

I.2.2.8- L'échange de messages

Modèle

SEND (destination, &message) : envoyer un message à un destination (ANY pour toutes)

RECEIVE(source, &message) : recevoir un message d'une source (ou n'importe quelle source avec ANY), le récepteur peut se bloquer si aucun message n'est disponible

- Contrôle de perte de messages avec l'identification du message et le message d'acquittement
- Adaptation à un environnement distribué en nommant des unités :
  - processus@machine ou machine:processus
  - ou encore dans le cas de confusion de noms de machine :

    processus@machine.domaine
- Authentification souvent faite par de codage spécifique
- Inconvenient : performance

I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.2- Communication inter-processus

I.2.2.8- L'échange de messages : modèle Producteur/Consommateur

```
#define N 100
#define TAILLE 4

typedef int message[TAILLE]
```

```
void producteur(void)
{
  int objet;
  message m;

  while (TRUE)
  {
    produitr_objet(&objet);
    receive(consommateur,  &m);
    faire_message(&m, objet);
    send(consommateur, &m);
  }
}
```

```
void consommateur(void)
{
  int objet, i;
  message m;
  for (i=0; i < N; i++)
  send(producteur, &m);
  while (TRUE)
  {
    receive(producteur,  &m);
    retirer_objet(objet);
    send(producteur, &m);
    consommer_objet(objet);
  }
}
```

I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.2- Communication inter-processus

I.2.2.9- L'équivalence des primitives

SÉMAPHORE = MONITEUR = MESSAGE

Exemple d'utilisation de sémaphore pour le moniteur

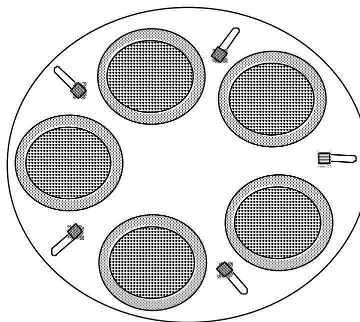
```
semaphore mutex = 1;            /* contrôler d'accès au moniteur */
void entrer_moniteur(void)
{
  down(mutex);                  /* un seul processus à l'intérieur à la fois */
}
void quitter_normalr(void)      /* quitter le moniteur sans signal */
{
  up(mutex);                    /* permettre à un autre processus d'entrer */
}
void quitter_signal(c)          /* quitter le moniteur avec signal */
semaphore c;                    /* variable de condition */
{
  up(c);                        /* permettre à un autre processus d'entrer */
}
void wait(c)                    /* dormir sur une condition */
semaphore c;                    /* variable de condition */
{
  up(mutex);                    /*permettre à un autre processus d'entrer*/
  down(c);                      /* permettre à un autre processus d'entrer */
}
```

Page 33

I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.3- Problèmes en Communication inter-processus

I.2.3.1- Le problème des philosophes



Page 34

I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.3- Problèmes en Communication inter-processus

I.2.3.1- Le problème des philosophes

```
#define N 5

void philosophe (int i)                /*n° de philosophes*/
{
  while (TRUE)
  {
    penser();                          /* il pense ! */
    P_fourchette(i);                   /* prend fourchette gauche */
    P_fourchette((i+1) mod n);         /* prend fourchette droite */
    manger();                          /* il mange ! */
    poser_fourchette(i);               /*poser fourchette gauche*/
    poser_fourchette((i+1) mod n);
                                       /* poser  fourchette droite */
  }
}
```

UNE SOLUTION ERRONÉE

Page 35

I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.3- Problèmes en Communication inter-processus

I.2.3.1- Le problème des philosophes : une solution

```
#define N
#define GAUCHE (i-1) mod N
#define DROITE (i+1) mod N
#define PENSE 0
#define FAIM     1
#define MANGE 2
typedef int semaphore;
int etat[N];
semaphore mutex = 1;
semaphore s[N];
```

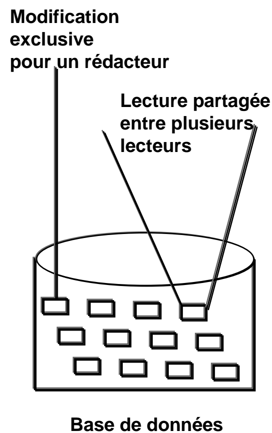
```
void philosophe (int i)
{
  while (TRUE)
  {
     penser();
    prendre_fourchettes(i);
    manger;
    poser_fourchettes(i);
  }
}

void prendre_fourchettes (int i)
{
  down(&mutex);
  etat[i] = FAIM;
  up(&mutex);
  down(&s[i]);
}
```

```
void poser_fourchettes (int i)
{
  down(&mutex);
  etat[i] = PENSE;
  test(GAUCHE);
  test(DROITE);
  up(&mutex);
}

void test(int i)
{
  if (etat[i] == FAIM &&
     etat[GAUCHE] != MANGE &&
     etat[DROITE] != MANGE)
  {
     etat[i] = MANGE;
     up(&s[I]);
  }
```

Page 36

I.2.3- Problèmes en Communication inter-processus

I.2.3.2- Le problème des lecteurs et des rédacteurs



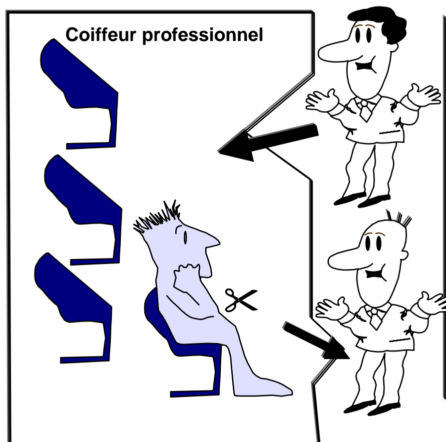

```
typedef in semaphore;

semaphore mutex = 1;
semaphore bd = 1;
int rc = 0;


void redacteur (void)
{
  while (TRUE)
  {
    creer_données();
    down(&bd);
    rc = rc + 1;
    écrire_données();
    up(&bd);
  }
}
```

```
void lecteur (void)
{
  while (TRUE)
  {
    down(&mutex);
    rc = rc + 1;
    if (rc==1) down(&bd);
    up(&mutex);
    lire_base_de_données();
    down(mutex);
    rc = rc -1;
    if (rc==0) up(&bd);
    up(&mutex)
    utiliser_données_lues;
  }
}
```


I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.3- Problèmes en Communication inter-processus

I.2.3.3- Le problème du coifeur endormi




```
#define CHAISES 5
typedef in semaphore;

semaphore mutex = 1;
semaphore coiffeurs = 0;
semaphore clients = 0;
int attente = 0;
void coiffeur(void)
{
  while (TRUE)
  {
    down(&clients);
    down(&mutex);
    attente = attente - 1;
    up(&coiffeur);
    up(&mutex);
    couper_cheveux();
  }
}
```

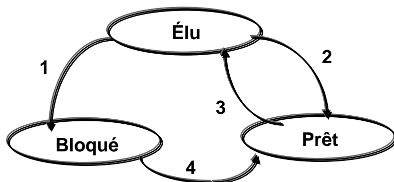
```
void cclient(void)
{
  down(&mutex);
  if (attente < CHAISES)
  {
     attente = attente + 1;
     up(&clients);
     up(&mutex);
    down(&coiffeur);
    obtenir_coupe();
  }
  else
  {
     up(&mutex);
  }
}
```

I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.4- Ordonnancement des processus

Introduction

Etats d'un processus



- Dans un système muti-utilisateurs, l'ordonnanceur s'occupe le choix d'un processus à exécuter

- Il distribue le temps du processeur à un ensemble de processus d'une manière optimale

OBJECTIFS

1- Equité : s'assurer que chaque processus reçoit sa part du temps processeur

2- Efficacité : utiliser le temps du processeur à 100%

3- Temps de réponse : minimiser le temps de réponse pour les utilisateurs en mode interactif

4- Temps d'exécution : minimiser le temps d'attente des utilisateurs qui traivaillent en traitement par lot

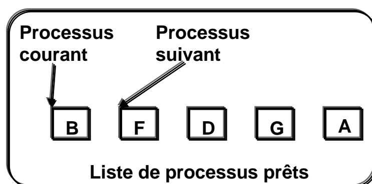
5- Rendement : maximiser le nombre de travaux effectués dans une unité de temps

I- Partie 1 : systèmes centralisés
I.2- Les Processus

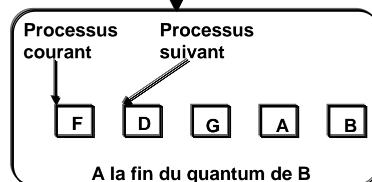
I.2.4- Ordonnancement des processus

I.2.4.1- Ordonnancement circulaire (tourniquet)

(round robin)



Liste de processus prêts



A la fin du quantum de B

Description

- Chaque processus possède un intervalle de temps, appelé QUANTUM, pendand lequel il est autirisé à s'exécuter

- Si un processus s'exécute toujours au bout de son quantum, le processeur est réquisitionné et alloué à un autre processus

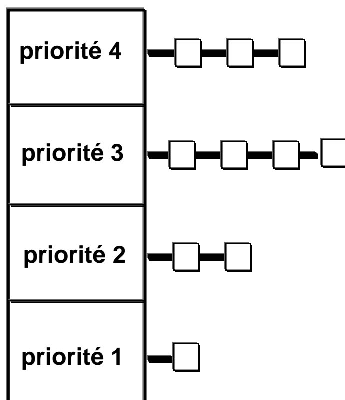
- Si en revanche, le processus se bloque ou se termine avant la fin de son quantum, le processeur est immédiatement alloué à un autre processus

- L'ordonnanceur doit mémoriser une liste des processus prêts, lorsque son quantum est épuisé le processus en cours est mis à la fin de la liste

I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.4- Ordonnancement des processus

I.2.4.2- Ordonnancement avec priorité



Description

- Chaque processus a une priorité et on lance le processus prêt dont la priorité est la plus élevée

- Pour empêché les processus de priorité élevé de s'exécuter indéfiniment, l'ordonnanceur diminue la priorité du processus élu à chaque impulsion d'horloge (à chaque interruption d'horloge)

- Si cette priorité devient inférieure à celle du deuxième processus de plus haut priorité, la commutation a lieu

- Les priorités peuvent être statiques ou dynamiques

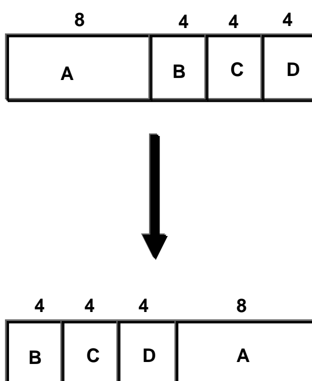
- Les processus de même priorité sont regroupés et gérés souvent par le mécanisme de tourniquet

Page 41

I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.4- Ordonnancement des processus

I.2.4.3- Ordonnancement du plus court d'abord



Description

- L'algorithme conçu pour les systèmes de traitement par lot, dans lesquels on connait à priori le temps d'exécution de chaque tâche

- l'algorithme cherche un ordonnancement des tâches qui minimise le temps moyen d'exécution des tâches :

$$\text{Min} \left(\left(\Sigma_i \; i \times t_i \right) \right)$$

où t_i est longeur de la tâche T_i

- Cela conduit à un ordonnencement par l'ordre croissant de longueur des tâches (disponibles en même temps)

- On peut appliquer aux systèmes interractifs en ajoutant une mesure dynamique de temps d'exécution

Page 42

I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.4- Ordonnancement des processus

I.2.4.4- Autres mécanismes

Ordonnancement garanti

- L'approche basée sur le principe de garantir à l'usager une certaine performance. Le système essaye de tenir ses promesses

- Exemple : "s'il y a n utilisateur connectés alors chacun reçoit 1/n de la puissance du processeur"

- On peut appliquer ce principe aux systèmes temps réels où il faut impérativement respecter certain délai

- Exemple : "Un processus qui est le plus proche à la limite de temps fixée pour lui est prioritaire sur les autres"

Décision et mécanisme

- L'approche consiste à séparer *mécanisme* et *décision* : l'algorithme d'ordonnancement est d'une certaine manière paramétrable et les paramètres sont fournis par les processus utilisateurs

- Cette approche est adapté dans un contexte des hiérarchies de processus : un processus peut avoir un ou plusieurs processus fils. En connaissant ses fils, il peut fournir à l'ordonnanceur des paramètres indiquant les priorités de ses fils (en exécution concurrante)

- Exemple : le processus principale d'un SGBD peut avoir plusieurs fils et participer à contrôler l'ordonnancement de ses fils

Page 43

I- Partie 1 : systèmes centralisés
I.2- Les Processus

I.2.4- Ordonnancement des processus

I.2.4.5- Ordonnancement à deux niveaux

Mémoire	disque
a, b , c, d	e, f , g, h
e, f , g, h	a, b , c, d
b , c, f, g	a, d, e, h

Description

- Une partie des processus est stockée sur le disque, et l'autre partie des processus est résidente dans la mémoire

- L'ordonnanceur de bas niveaux gère les processus dans la mémoire, il utilise un des mécanismes précédement présentés

- L'ordonnanceur de haut niveau gère le va-et-vient entre le disque et la mémoire : Après certain temps, il recompose de deux sous-ensembles en retirant certains processus de la mémoire et les remplacant par certains autres processus parmi ceux setrouvant sur le disque

- L'algorithme est adapté dans un contexte où la mémoire cetrale est relativement petite

Page 44

I- Partie 1 : systèmes centralisés

I.2- Les Processus

I.2.5- Exercices

1- **Qu'est-ce qu'un confit d'accès ?**
2- **Quelle est la dofférence entre l'attente active et le bloquage ?**
3- **La solution du problème des philosophes mettait la variable d'état à FAIM dans la procédure poser_fourchettes. Expliquez pouquoi ?**
4- **Suposons que dans la procédure poser_fourchettes, la variable etat[i] soit fixée à PENSER après les deux appels à test au lieu d'avant . Que passera-t-il dans le cas de 3 philosophes ? Même question pour 100 philosophes**
5- **La solution du problème du coiffeur endormi peut-elle être généralisée pour plusieurs coiffeurs ? Si oui comment ?**
6- **Un ordonnanceur circulaire mémorise la liste des processus prêts et chaques processus n'apparaît qu'une fois dans la liste. Que se passera-t-il si un processus apparaissait deux fois ? Y a-t-il des cas où il faut le mettre ?**
7- **La pluspart des ordonnanceurs circulaires utilise un quantum fixe. Donnez un argument en faveur d'un petit quantum et un autre en faveur d'un grand quantum**
8- **On utilise l'algorithme de veillissement avec a = 1/2 pour prévoir les temps d'exécution. Les quatre dernières exécutions ont données dans l'ordre 40, 20, 40 et 15 ms. Quel temps d'exécution suivante peut-on prévoir pour l'exécution suivante**
9- **Le problème de la boulangerie (Lamport 1974) : Les n vendeurs d'une boulangerie vendent des pains et des gâteaux. Chaque nouveau client prend un numéro et attend jusqu'à ce que son numéro soit appelé. On appelle un numéro suivante chaque fois qu'un vendeur se libère. Ecrivez une procédure à l'attention des vendeurs et une autre l'attention des clients**