Système d'exploitation et programmation concurrente

Texte issu du cours de Guy Mazaré (2006)

Systèmes d'exploitation : Unix, Linux, MS-DOS, Windows, MacOS, GCOS (Bull), OS 3xx (IBM), VMS (DEC), VM (machine virtuelle).

Programmation parallèle :

- Programmation différente de la programmation séquentielle vue en première année.
- Ensemble d'activités en exécution simultanée.
- Notions de programme sans fin.
- Sert à faire des systèmes et d'accélérer les calculs (Activités simultanées communicantes et synchronisées)

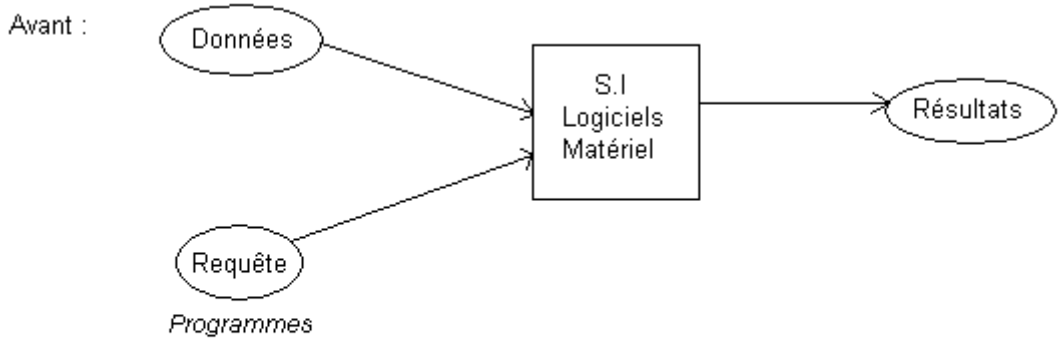
Livres référence :

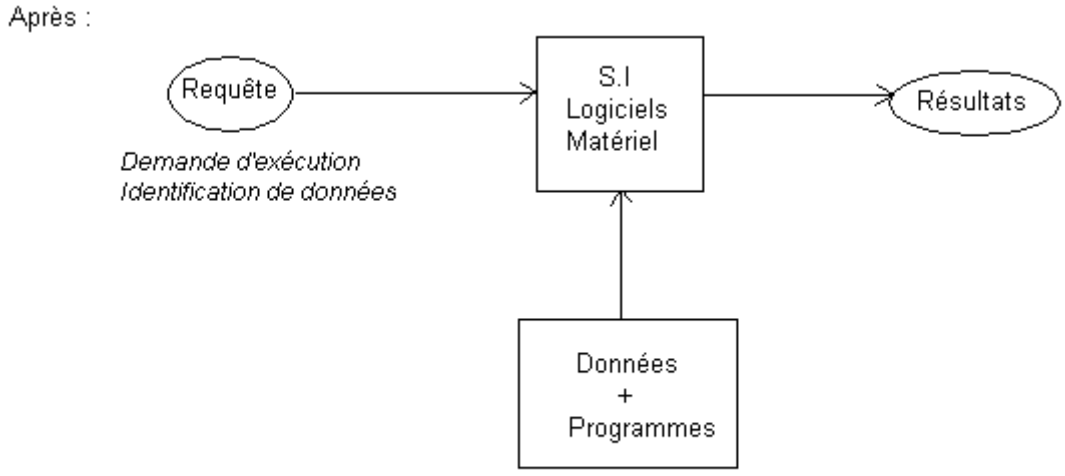
- *Modern Operating Systems / Systèmes d'exploitation (Andrew Tanenbaum), Prentie Hall 2001, Pearson Education France2003.*
- *Principe des systèmes d'exploitation des ordinateurs (Sacha Krakowiak)*

I) Introduction : fonctions et structure d'un système d'exploitation.

1. Fonctions - Généralités :

Système informatique :

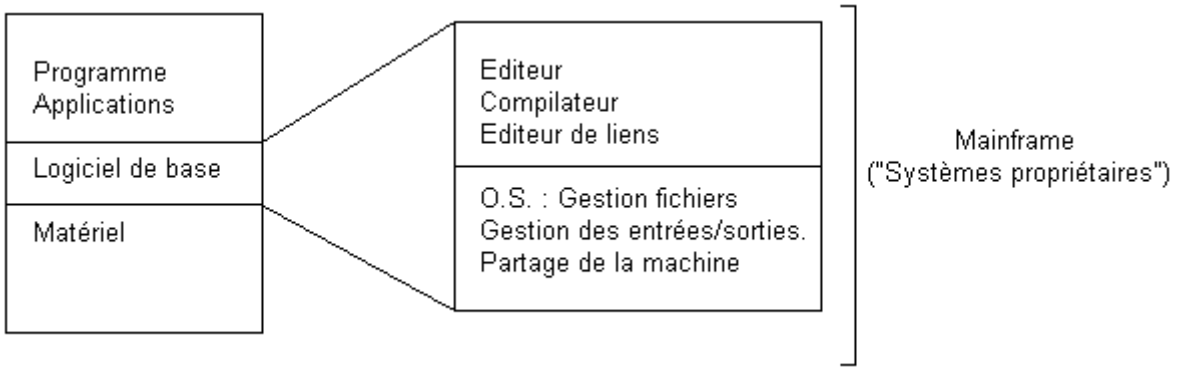




Fonctions :

- Exécution des programmes
  - Gestion
  - Calcul scientifique
- Gestion de l'information
  - Gestion des fichiers
- Construire de nouveaux programmes
  - Éditer
  - Compiler
  - Édition de liens
  - Lancer l'exécution

Composants du SI



Notion de Machine Virtuelle :

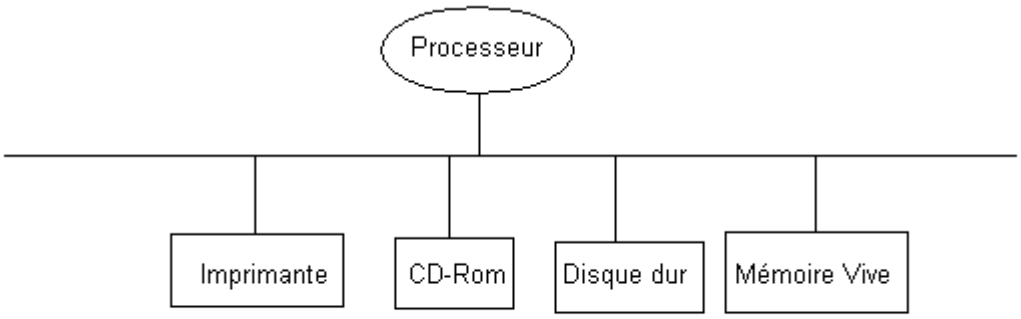
- Machine abstraite : ensemble de fonctions
- Partager une ressource (machine) : émuler plusieurs machines virtuelles sur une même machine

2. Exemples de Systèmes Informatiques

Le PC

Ordinateur individuel : Windows, MS-DOS

Configuration matérielle :



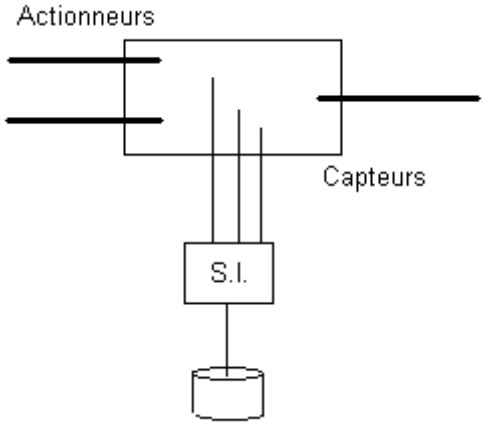
Services :

- Fichiers :
  - Créer
  - Supprimer
  - Renommer
  - Accéder
- Exécution d'applications (ex : éditer un fichier) et partage de programmes (langage de commandes).

Contrôle de procédés : « Process Control »

Programmes :

- Régulation (lancée à période fixe T)
- Enregistrer l'histoire du procédé
- Gestion des alarmes (traitement urgent) implique l'exécution d'un programme.



En train chaque programme, si le processeur est inutilisé, une tâche de fond est lancée.
Plusieurs activités : partager la machine (priorités)
Contraintes « temps réel » : « Systèmes critiques »

Types de systèmes :

- Ordinateur personnel
- Système de contrôle de procédé (temps réel)
- Système à transaction
  - Client/serveur
  - Architecture n-tiers
- Systèmes multiprogrammés traditionnels (pour mémoire)
- Système à « temps partagé » (Plusieurs utilisateurs connecté sur la même machine : telesun...).
  - Partage de ressources (processeur(s), mémoire centrale, mémoire de masse, périphérique) : de manière « spatiale » ou « temporelle » (time sharing)

3. Évolution (historique) des systèmes d'exploitation et des modes opératoires :

1) « L'ordinateur primitif »

Ne possède qu'une mémoire volatile et des périphériques tels qu'une imprimante ou un lecteur de cartes.
Les programmes d'entrées/sorties n'étais pas stockés mais à rentrer à la main.
Pour le démarrage de l'ordinateur on utilisait au début le pupitre pour initialiser les registres du processeur et les premiers programmes. On a vite évolué vers un système d'amorçage sur cartes (« bootstrap »).

OS :

- Initial Program Loader (IPL)
- Routines d'ES

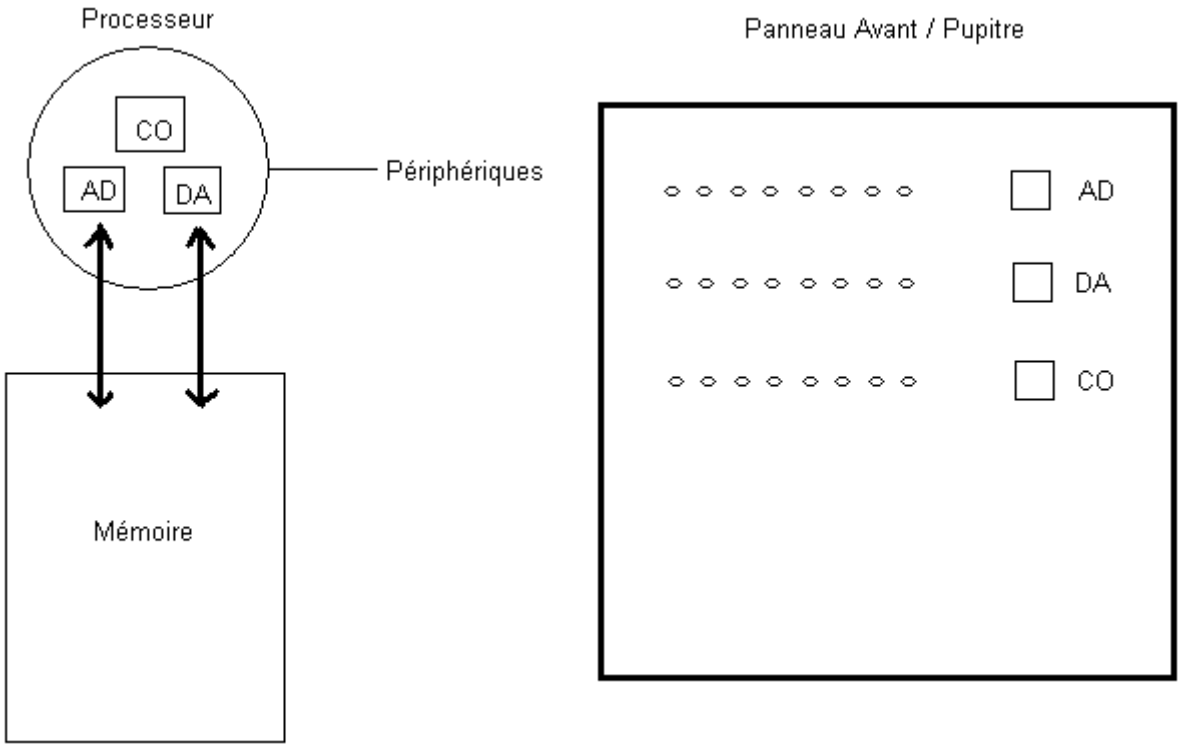
Moniteur d'enchaînement des travaux :

- Permet de changer d'utilisateur sans éteindre la machine
- Nécessite des cartes de contrôle

On a alors commencé à stocker en mémoire le moniteur d'enchaînement des travaux et les programmes d'entrée/sortie.

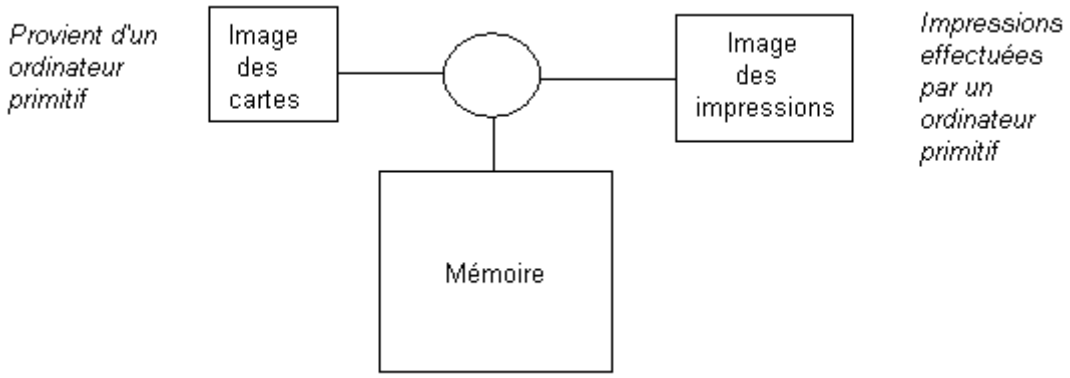
Les problèmes :

- Un utilisateur peut écraser le moniteur ou les sous-programmes d'E/S : On met en place un mécanisme de protection mémoire.
- Impossible d'arrêter un programme qui boucle, ou erreurs telles que la division par 0 : Mise en place d'un système d'horloge et de gestion des erreurs.
- Lecture de toutes les cartes par un programme utilisateur : Empêcher l'utilisateur d'utiliser le moniteur en séparant les instructions en deux catégories : privilégiées (superviseur) et standards. Ajout d'un bit pour le mode dans le processeur.



2) Traitement par lots

Apparition des bandes magnétiques bien plus rapides qui permet d'améliorer la vitesse de lecture de carte bien inférieur à la vitesse du processeur.



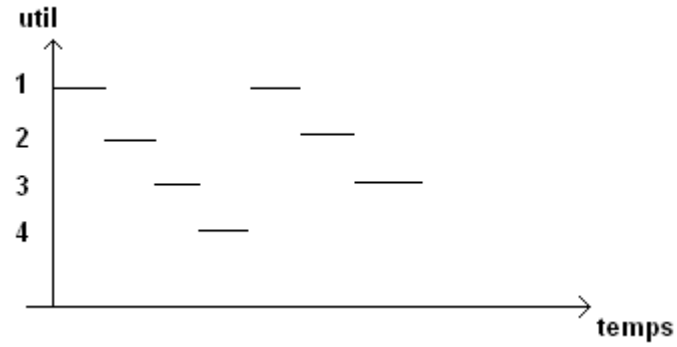
3) Multiprogrammation

Afin d'éviter les transferts de carte entre les ordinateurs, on crée des unités d'échange (UE) qui permettent d'écrire directement dans la mémoire de l'ordinateur les jobs à effectuer. Utilisation d'un disque dur pour stocker un grand nombre de jobs qui s'exécuteront plus tard.
Ceci a permis d'exécuter les jobs les uns à la suite des autres de manière efficace. On a également pu faire travailler des processeurs en parallèle dans la même machine.

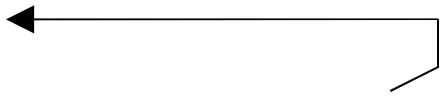
Gestion des activités parallèles

1. Introduction *:* Exemples d'activité parallèle

- Time Slice / Temps partagé



Time Slice



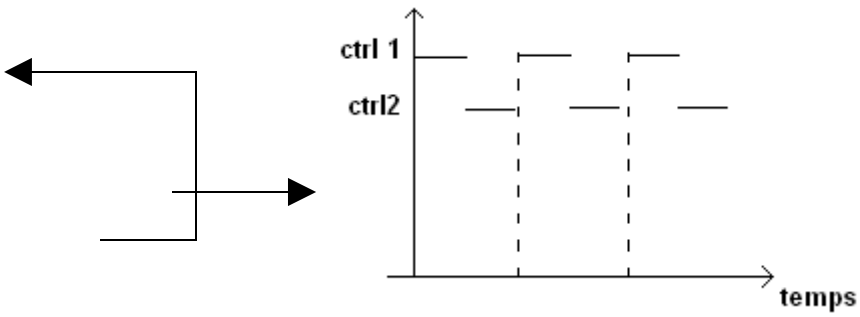
- Spools/Multiprogrammation : 1) Lire carte(i_carte) ; read
  Entrer i_carte dans tampon_entree

  si il y a de la place dans tampon_entree
  alors déposer i_carte
  sinon attendre
  déposer

  2) Sortir i2_carte de tampon_entree
  Transfert disque jobqueue

  sortir i2_carte
  si 1) attend
  alors la relancer

- Contrôle de procédé :



Synchronisation des activités.

h=heure
lire capteur
calculer la réaction
écrire actionneur
attendre(h+T)

- Code de l'activité
- « Contexte » = {données}

Définitions

- État de la machine = { valeurs de disque « mémoire » } *// Mémoire = Registre ou MC*
- Contexte (d'une activité) : { valeurs de mémoire qui la concerne }
- Instruction : fait passer la machine de l'état avant à l'état après grâce à son exécution. Ensemble d'instruction = programme.
- Processus séquentiel : ~exécution du programme (caractérisé par la suite des états).
  □ Vitesse imprévisible
- Système : { processus }

Parenthèse

Synchronisation : **L'exclusion mutuelle.**

Exemple 1 : Réserver

Entrer en SC
- lire nb_places_libres
- si nb_places_libres == 0 alors return (« Complet »)
- sinon nb_places_libres --
  et ajouter_liste(nom du passager)

Sortir de SC

SC = Section critique □ mécanisme d'exclusion mutuelle. Un processus au plus dans la section critique.

↓

multiprocesseur : Les accès mémoire ne peuvent pas avoir lieu en même temps Sans section critique, on a un problème (suivre l'ordre □ à la fin, on a nb_places_libres := nb_places_libres – 1 et pas -2 □ bug). Il faut donc introduire les indications de section critique :

1) Il y a au plus 1 processeur en train d'exécuter une section critique.
2) Aucun proc. ne reste en attente si la section critique est libre.
3) Pas de privilégié/statuts symétriques.

- Attente active :

α : si b=0
alors c=1
sinon aller à α

Algo de DEKKER :

Méthode 1 : booléen c = libre/occupé

Entrer en SC :
α : si c == occupé
alors aller à α
sinon c = occupé

Sortir de SC :
c = libre

□ Problème : il faut que l'entrée en SC soit dans la SC elle-même…

Méthode 2 : variable t = « tour » = 0 ou 1

Entrer en SC :
α : si t != i
alors aller à α

Sortir en SC :

t = 1-i

 Problème : si un processeur n'a pas besoin de la SC, alors l'autre ne peut plus l'avoir deux fois de suite.

Méthode 3 : tableau c[0..1] = libre/occupé

Entrer en SC :
α : si c[1-i]=occupé
alors aller à α
sinon c[i]=occupé
Sortir de SC :
c[i]=libre

Problème : si les deux font la comparaison en même temps, ça ne marche plus.

Méthode 4 :

Entrer en SC :
α : c[i] = V
si c[i-1]==V
alors si t != i
alors c[i] = F
aller à α

Sortir de SC :
t = i-1
c[i] = F

 Problème : s'il n'y a pas de conflit, le premier rentre, et le deuxième voit qu'il y a conflit mais c'est son tour, donc il rentre aussi…

Méthode de DEKKER :

α : c[i] = V
β : si c[j]
alors
si t == i
alors aller à β

Si c'est mon tour, i'insiste

c[i] = F
γ : si t != i
alors aller à γ.
aller à α

Sinon, annuler ma demande, attendre que l'autre ait fini, puis recommencer

Méthode de PETERSON :

c[i] = V
t=i
α : si c[i-1] et t == i
alors aller à α

c[i]=F

Processus : Vitesse imprévisible. On peut les assimiler à des processeurs virtuels.

2. Synchronisation

Exemple 1 : fin(ecrire(a)) < debut(lire(a)) (un processus P2 doit attendre qu'un processus P1 ait écrit la variable a avant de la lire)

Exemple 2 : barrière de synchronisation (rendez-vous)

Code de Pi : avant ; rv ; après ;

Quelque soit i,j, fin(avant i) < debut(après j)

Spécification opérationnelle des synchro.

- Point de synchronisation : endroit où il faut (peut-être) attendre.
- Condition associée : Variable(s) de synchronisation (globales)

« autorisation de passer le point de synchro (p.s.) = aut(,)
Exp 1 : aut(po) :arrivé = Vrai.
Exp 2 : aut(po) = (n=N)

Un premier exemple d'outil de synchro : L'événement / events.

Un événement e = { arrivé , non arrivé }. On définit attendre(e), qui attend que e soit à l'état « arrivé ».

3. Premier outil de programmation de synchronisation : les moniteurs

```
moniteur toto ;
        var
         |
         |
         |
        proc a ;
         |
         |
         |
        proc b ;
         |
         |
        début
           initialisation
        fin
```

1. Les procédures sont toujours exécutées en exclusion mutuelle. ➔ Au plus un processus à la fois actif dans le moniteur.
2. Synchro :
   a. Les instructions de synchro portent sur la variable « condition »
   b. Condition : attendre/ signaler
      « Attendre » provoque la mise en attente qui l'exécute.
      « Signaler » relance un processus qui attend. (pas forcément FIFO)

Exemple 1 :

```
synch : moniteur
var
        arrivé : bool
        c : condition
procedure debutlire
        si arrive = F alors c.attendre ;
procedure finecrire
        arrive = V ;
        c.signaler
debut
        arrive = F
fin
```

Exemple 2 :

```
moniteur : barrière
        var : n : entier
              c : condition
procedure arriver ;
        n = n+1
        si n < N alors c.attendre ;
                c.signaler ;
        sinon // Les réveiller tous
                c.signaler ;
debut
        n = 0
fin
```

4. Schémas classiques de processus coopérants

1) Exclusion mutuelle :

Exemple : a = a + 1

```
        sect_crit : moniteur
        procedure add(n)
                n = n + 1
```

```
        P1                      P2
sec_crit.add(a)         sect_crit.add(a)
        ...                     ...
sec_crit.add(b)         sect_crit.add(b)
```

On ne peut modifier a et b en même temps.
Solution possible : faire un moniteur par variable.

Autre solution:

```
        P1                      P2
        ...                     ...
exclusion.entrer        exclusion.entrer
a = a + 1               a = a + 1
exclusion.sortir        exclusion.sortir
        ...                     ...
```

```
exclusion : type moniteur
        libre : bool
        c : condition

procedure entrer
        si non libre alors
                c.attendre;
        libre = F;

procedure sortir
        libre = V;
        c.signaler;

debut
        libre = V;
fin
```

En instanciant les exclusion (exclusion_a, exclusion_b ...), on peut modifier plusieurs variables en même temps.

```
        P1                      P2
        ...                     ...
exclusion_a.entrer      exclusion_b.entrer
a = a + 1               b = b + 1
exclusion_a.sortir      exclusion_b.sortir
        ...                     ...
```

2) Réservation de ressources multivaluées

```
...
reserver
...
liberer
...
```

Ressources binaires :
mono_valuée identique à une exclusion mutuelle

Ressources multi-valuées :
N exemplaires

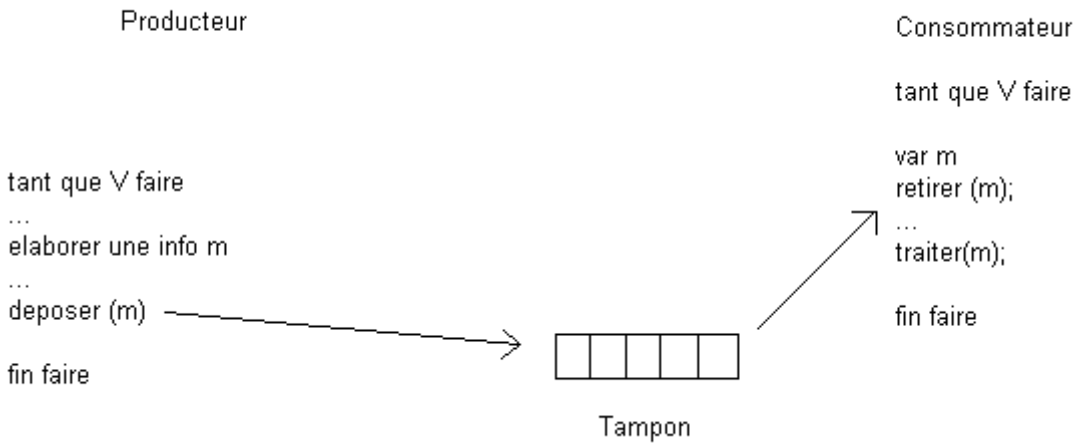
Processus : Demande de 5 exemplaires, doit en liberer 5. Si pas assez d'exemplaires, le processus attend.

```
reserver(5)
...
liberer(5)
```

```
ressource : moniteur
        nlibre : entier;
        c : condition;

procedure reserver (n)
tant que n>nlibre
        faire c.attendre; c.signaler;
nlibre = nlibre – n;

procedure liberer(n)
nlibre = nlibre + n;
c.signaler;

debut
        nlibre = N;
fin
```

3) Producteur/Consommateur

Utilisation d'un tampon (exemple : Production de pièces stockées dans le tampon pour être utilisées après)



Spécifications :
– Chaque message déposé est retiré
– Aucun message n'est retiré plusieurs fois
– On ne peut ni déposer dans un tampon plein, ni retirer dans un tampon vide

Tampon de taille N :

```
procedure t_entrer(m)
        T[queue] := m;
        queue := queue + 1 mod N;

procedure t_sortir(var m)
        m = T[tete]
        tete = tete + 1 mod N;
```

```
Tampon : moniteur
        n : entier
        C1, C2 : condition

procedure deposer(m)
si n = N
        alors C1.attendre;
n = n + 1;
t_entrer(m);
C2.signaler;

procedure retirer (var m)
si n = 0 C2.attendre;
n = n – 1;
t_sortir(m);
C1.signaler;

debut
        n = 0
fin
```

4) Client/Serveur

Le système marche également s'il y a plusieurs producteurs ou consommateurs.

- Client/serveur sans réponse (serveur d'impression par exemple).
- Client/serveur avec réponse déposée dans un autre tampon d'une seule case (boîte aux lettres privée indicié par le numéro du processus).

Lorsque le client souhaite avoir une réponse du serveur on utilise le numéro du processus client :
Chez le client : balpriv[moimeme].retirer(rep)
Chez le serveur : balpriv[req.p].deposer(rep)

Organisation d'un système (ensemble des processus)
- horizontale
- verticale

5. Composant défectueux

Tous les processus doivent être corrects.

1) Interblocage / Dead-lock

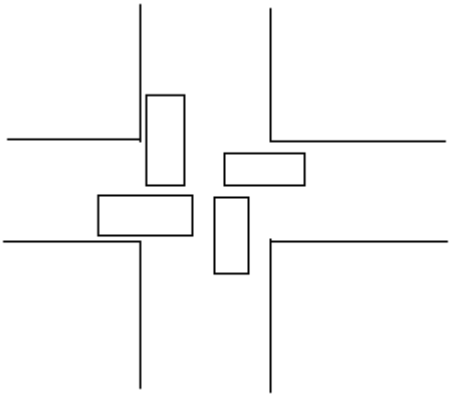
Exemple 1

Processus 1	Processus 2
…	…
reserver (R1)	reserver(R2)
…	…
reserver (R2)	…
…	…
…	reserver(R1)
liberer (R1)	…
liberer (R2)	liberer (R1)
…	liberer (R2)
	…

Exemple 2 : Ressources multivaluées (Chaque processus demande plus de ressource que disponible et les deux attendent)

Exemple 3 :

Carrefour routier :



Solutions ?

~~1. Accepter le risque d'interblocage~~
2. Détection et traitement de l'interblocage :
Detecter?

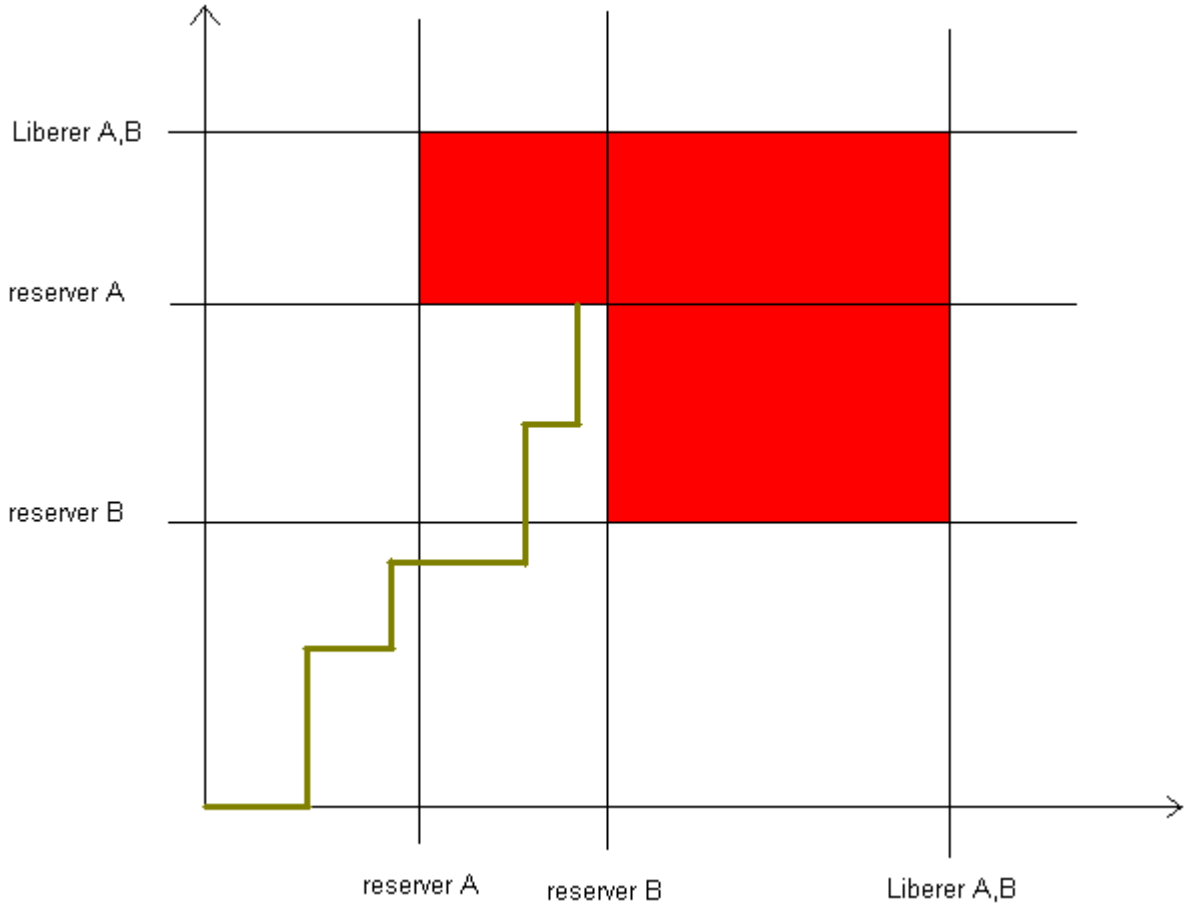
- processus (démons) surveillant l'ensemble des processus (inactivité, trouver un cycle dans le di-graphe)
- lors de sa formation : chaque demande de réservation de ressource avec attente.

Traiter?

Ressource pré-emptible (pré-empter : réquisitionnée)

3. Éviter les interblocages

Problème : cela demande de connaître le futur



4. Systèmes sans risques d'interblocage

Définir un ordre total sur les ressources et les demander dans cet ordre.

1) Privation

Un processus se fait toujours passer devant et n'a jamais son tour.

6. Deuxième outil : les sémaphores

sémaphore s
    s.c : compteur entier Z
    s.f : une file de processus en attente

P(s) :
s.c := s.c – 1
si s.c < 0 alors
    entrer(moi-même, s.f)
    bloquer le processus « moi-même »

Remarque : Gestion des files
    entrer (objet, file)
    sortir (var objet, file)
    vide (file) return bool
    premier (file) return objet
    suivant (objet, file)

V(s) :
s.c = s.c + 1
si s.c <= 0 alors
    sortir (p, s.f)
    debloquer (p)

np, nv sont respectivement le nombre d'appels à P et à V
nf : le nombre de processus qui n'ont pas été arrêtés.

I] **s.c = s.c0 + np(0) + nv (0)**
II] si s.c >= 0 alors card (s.f) = 0
   si s.c < 0 alors card (s.f) = - s.c
   card (s.f) = max (0, -s.c)
III] **nf(s) = min (np(s), s.c0 + nv(s))**

1) Exclusion mutuelle

s.c initialisé à s.c0 = 1

```
...
entrer s.c;
...
...                 s.c = 0
...
sortir s.c;
...
```

2) L'évènement

s.c0 = 0

```
...          ecrire (a)
lire (a)     ecrire (a)
P (s)        V (s)
```

3) Producteur / Consommateur

s1.c0 = N et s2.c0 = 0

Producteur	Consommateur
	var m
P(s1)	P(s2)
t_entrer(m)	t_sortir(m)
V(s2)	V(s1)

Attention : ne marche qu'avec un seul consommateur.

Solution :
On veut empêcher plusieurs manipulations du tampon à la fois.
sémaphore s1, s2, mutex

Producteur	Consommateur
	var m
P(s1)	P(s2)
P(mutex)	**P(mutex)**
t_entrer(m)	t_sortir(m)
V(mutex)	**V(mutex)**
V(s2)	V(s1)

Trop restrictif : on ne peut sortir et entrer en même temps

Solution :
sémaphore s1, s2, mutex1, mutex2

Producteur	Consommateur
	var m
P(s1)	P(s2)
P(mutex1)	**P(mutex2)**
t_entrer(m)	t_sortir(m)
V(mutex1)	**V(mutex2)**
V(s2)	V(s1)

4) Barrière de synchro

s.c initialisé à s.c0 = 1

```
P(mutex)
n := n + 1;
si n = N alors
        pour i = 1 à N-1 faire V(s)
V(mutex)
P(s)
```

Introduire un sémaphore privé pour chaque processus : spriv[1..nbprocessus]

```
P(mutex)
n = n + 1
si n = N alors
        pour k = 1 à N
                n = n – 1
                V (spriv[k])
V(mutex)
P(spriv[i])
```

6. Gestion dynamique des processus

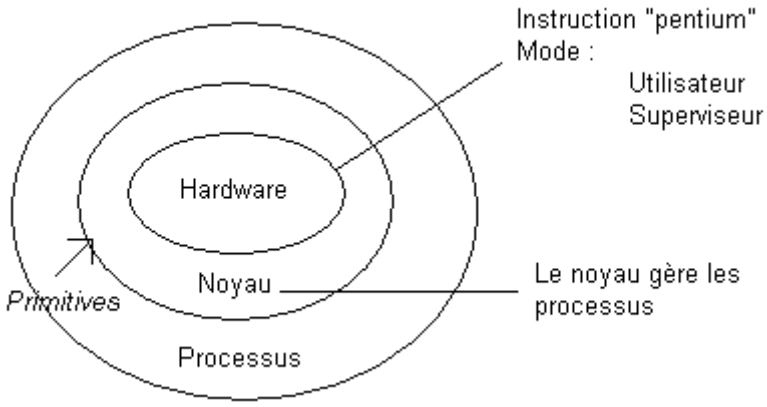
creerprocessus (adprogr, var pfils, paramètres)

Exemple : Unix
...
pid = fork ; // Execution parallèle à partir de la ligne suivante
si pid = nil alors FILS
sinon PERE

finprocessus (exit en UNIX)

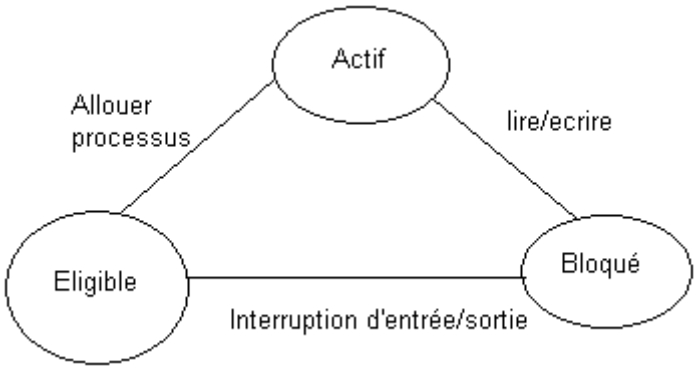
tuerprocessus (pid)

7. Implémentation des processus et des synchronisations : le noyau du système d'exploitation.



Primitives : P, V
creerprocessus / finprocessus
lire/écrire : synchrones (périphérique)

Etats d'un processus



Le noyau est séparé des processus dans la mémoire.

Allouer processus : politique de choix du processus éligible
- FIFO
- files par priorités
- ...

Chaque processus est associé à un *bloc de contrôle processus* (bcp)
processus p --> bcp[p]

- Reg
- Etat
- CO
- ...

Prologue :
bcp[elu].Reg = REG
bcp[elu].etat = sauvegarde_etat
bcp[elu].co = sauvegarde_co

Contexte du processus : valeurs sauvegardées dans beaucoup de processus élu (actif)

Appels de sous programmes (call en pentium)
Appels superviseur : SVC
Processus:

```
        ...
        ...
P(s)  --> Appel de la primitive noyau P(s)
        ...
        ...
Lire( )
        ...
        ...
```

Implémentation des primitives :

P(s) :
prologue
contrôle
s.c = s.c – 1
si s.c <= 0 alors sortir (p, f_eligible) // p est le processus élu
allouerprocesseur

allouerprocesseur :
elu = premier (f.eligible)
REG := bcp[elu].Reg
chargermep (bcp[elu].etat, bcp[elu].co) // mep = Registre d'état + CO
// ETAT := bcp[elu].etat
// ETAT := bcp[elu].co

V(s) :
prologue
contrôle
s.c = s.c + 1
si s.c <= 0 alors
sortir (p, s.f)
entrer(p, f_eligible)
allouerprocesseur

Lire/Ecrire (n°periph, param) :
prologue
contrôle
préparer les paramètres de transfert
Sio
sortir (p, f_eligible)
actif[n°periph] = p
allouerprocesseur

IT --> Traitement d'IT_finES (n°periph) :
prologue
vérifier E/S terminée correctement
si OK alors
    entrer (actif[n°periph], f_eligible)
allouerprocesseur

créerprocessus (adprog, var pfils, param) :
prologue
contrôle
i = allouer_bcp
bcp[i].Reg = bcp[elu].Reg
bcp[i].etat = bcp[elu].etat
bcp[i].co = adprog
entrer(i, f_eligible)
bcp[elu].Reg.eax = i
allouerprocesseur

Le noyau est une section critique
    => à exécuter en exclusion mutuelle !

1.En monoprocesseur, noyau non interruptible
    => Mode masqué
2.En multiprocesseur

Calcul parallèle

http://www-id.imag.fr/Laboratoire/Membres/Roch_Jean-Louis/perso_html/enseignement.html

1. Motivations

– Fournir plus de puissance, besoin de calcul (ex : météo)
– Packaging technologique (à une technologie donnée, multiplier les processeurs)

2. Architecture parallèles de machines

– Parallélisme interne à un processeur
  – Parallélisme bit/mot
  – « Overlay » : Le processeur prépare l'opération suivante.
  – Pipeline : On prépare les N opérations suivantes.
    Exemple :
    pour i = 1 à N
      y[i] = x[i] * a[i] + b[i]

    op1 : fetch a[i]
    op2 : fetch x[i]
    op3 : *
    op4 : fetch b[i]
    op5 : +
    op6 : store

– Processeurs parallèles
    Dans notre exemple, chaque processeur peut calculer une occurrence de la boucle.
    Amélioration au cours des années : Chaque processeur aux processeurs voisins : en ligne, en tableau voire en tores (le premier est relié au dernier) ou en hypercube (« connection machine »)

– Multiprocesseurs à mémoire partagée centrale unique
    Multiprocesseurs à mémoire partagée mais chaque processeur a son propre cache. Le problème du bus unique est compensé par le cache des processeurs. Ou utilisation du crossbar, connexion à toutes les mémoires (complexité n^2)

– Multiprocesseurs à mémoire partagée non-uniforme
    Chaque processeur a sa propre mémoire mais peut accéder à celle des autres

– Multiprocesseurs / Clusters
    Interconnexion entre les différents processeurs

– Systèmes répartis

3. Multiprocesseurs à mémoire partagée

Programmation des multiprocesseurs

Principe : indépendance du code par rapport au nombre de processeurs
Outils logiciels :
– Système : Gestion des processus / thread par le noyau
– Applications : décrire des « tâches » parallèles, émulées chacune par un thread.
  – Algorithme parallèle : expliciter le parallélisme ou détecter le parallélisme (compilateur)
  – CreerProcessus / FinProcessus

Organisations parallèles :
– Parallélisme de données
    pour i = 1 à n
        creerprocessus(add, i)
    pour i = 1 à n
        balrep.retirer()

```
procedure add (i)
        y[i] = x[i] + ...
        balrep.deposer()
        finprocessus
```

– Parallélisme fonctionnel et dépendances

```
        ex :
        x := y + 1
        z := y+ z + 10*t
        w := x + z
```

Opérations itératives --> « Produit itéré »

ex : Calcul en arbre pour le produit des a_i en (log(N) au lieu de N, si l'on a N/2 processeurs)

```
a[0..N-1] of int
int prod (ideb, ifin)
begin
        if ifin = ideb + 1alors
                return ( a[ideb] * a[ifin] )
        sinon
           in = (ideb + ifin) / 2
           r1 = prod (ideb, ifin)
           r2 = prod (in+1, ifin)
           return (r1 * r2)
```

Complexité parallèle

Études de complexité

En séquentiel :
– Algorithme A(n) : ensemble de n données
– Complexité = f(n)

Opération élémentaire :
– Un ou deux accès à la mémoire
– Une opération (comparaison, arithmétique)

En parallèle : Modèle PRAM

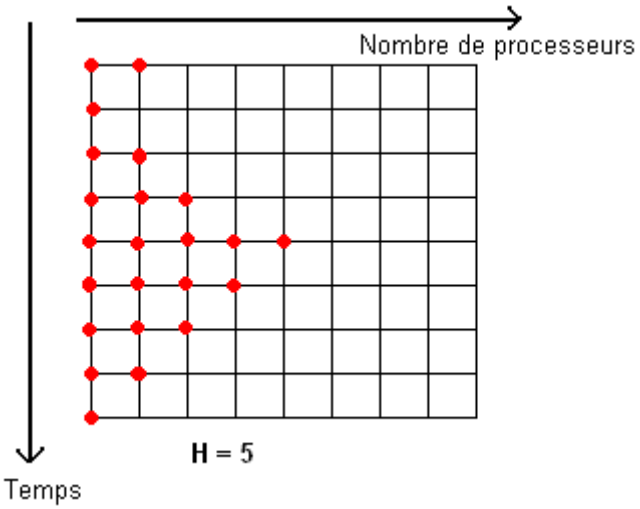
Les processeurs sont reliés à une RAM commune
Différents sous-modèle :
– Concurrent Read Concurrent Write (CRCW)
– Concurrent Read Exclusive Write (CREW)
– Exclusive Read Exclusive Write (EREW)

En séquentiel :
$T_s(A_n) = f(n)$

En parallèle :
$T_{//}(A_n') = f'(n)$
H(An') = le hardware = le nombre de processeurs nécessaire dans l'étape la plus exigeante.

Accélération : $a(An) = Ts / T_{//}$

Notion de travail : $W_{//} = T_{//} \times H$



En séquentiel, H = 1 et Ws = Ts

Efficacité : $e(An) = Ws/W_{//} = Ts / (T// \times H) = a/H$

Algorithme parallèle :
- **Efficace** : $T = O(\log^{\alpha}(n))$ avec $\alpha \rightarrow O(1)$
  $W_{//} = Ts \times O(\log^{\alpha'}(n))$
- **Optimal** : $T = O(\log^{\alpha}(n))$ avec $\alpha \rightarrow O(1)$
  $W_{//} = O(Ts)$

Algo de classe NC :
$T = O(\log^{\alpha}(n))$
$H = O(n)$

Exemple : produit itéré
$Ts = O(n)$
$T_{//} = O(\log_2(n))$
$H = O(n)$
$W_{//} = O(n \log(n))$

$Ws = Ts = O(n)$

Algorithme efficace mais non optimal.

<u>Principe de Brent</u> :
- Modus Ponens
- Pour tout m Є {1, ..., H(An) }
  An peut être executé en $O(m \times T_{//}(An))$
  sur $O (H(An) / m)$ processus

Application au produit itéré :
Il faut diviser par $\log_2(n)$ le nombre de processeurs
Étape 1 : Chaque processeur effectue $m = \log_2(n)$ produits
$n / \log_2(n)$ processeurs
Étape 2 : Ancien produit itéré des $n/\log_2(n)$ résultats.

	T	H
Étape 1	$O(\log_2(n))$	$O(n/\log_2(n))$
Étape 2	$O(\log_2(n/\log_2(n)))$	$O(n/\log_2(n))$
Résultat total	$O(\log_2(n))$	$O(n/\log_2(n))$

Tri naïf :

On calcule le rang de chaque élément.
En séquentiel, on a $Ts = O(n^2)$

On cherche une algorithme avec n^2 processeurs qui font les n^2 comparaisons :

	T	H
Étape 1	$O(1/\log_2(n))$	$O(n^2/\log_2(n))$
Étape 2	$O(\log_2(n))$	$O(n^2/\log_2(n))$
Étape 3	$O(1)$	$O(n)$
Résultat total	$O(\log_2(n))$	$O(n^2/\log_2(n))$

$T_{//} = O(\log_2(n))$
$W_{//} = n^2$ qui est moins bien qu'en séquentiel

Addition d'entiers :

Utilisation de « Diviser pour Régner » :

adentier(c, R <- A + B)

	T	H
Étape 1 : A -> (A_H, A_L) B -> (B_H, B_L)	$O(1)$	$O(n)$
Étape 2 : C_H,R_H <- adentier(A_H+B_H) C'_H,R'_H <- adentier(A_H+B_H+1) C_L,R_L <- adentier(A_L+B_L)	$T(n) = T(n/2)$ $\log_2(n)$	$H(n) = 3H(n/2)$ $H(n) = 3^{\log2(n)} = n^{1,58}$
Étape 3 : si $c_L = 1$ alors return (C'_H,R'_H,R_L) sinon return (C_H,R_H,R_L)	$O(1)$	$O(n)$
Résultat total	$O(\log_2(n))$	$O(n^{1,58})$

Systèmes et applications répartis

1. Introduction

Qu'est-ce qu'un « Système Réparti » ?

Architecture physique :
Réseau de communication entre plusieurs machines
- Dédiés
- Locaux
- Généraux (internet)

Approches / Motivations :
Se connecter à d'autres systèmes :
- FTP
- MAIL
- Web

Adapter la structure du S.I. à la réalité géographique
- Production
- Bancaire

Réalisation de « systèmes distribués » en installant des « serveurs » spécialisés
- Impression
- Disque

Exécution de programmes / application à l'aide de plusieurs processeurs.

2. Outils de base

- Les systèmes d'exploitation hétérogènes
- Communication

Les réseaux :
Interconnexion deux machines puis N machines (Notion d'adressage)
Routage

Couches du système de communication
- Niveau physique
- Niveau lien liaison
  - Protocole de contrôle d'erreur
  - Protocole de contrôle de flux
  - Niveau réseau + paquet
  - Niveau transport

Communication de processus à processus :
- Longue durée
- Mode datagramme

Communication entre les niveaux via des boîtes aux lettres

Processus Envoyer / Recevoir
Niveau 4
Niveau 3 : routage
Niveau 2
Niveau 1

Services :
Datagramme : « envoyer une lettre »
Ouvrir connexion – Recevoir (m) – Envoyer (m) – Fermer Connexion

Modèle Client / Serveur :

```
processus Serveur
        Tant que Vrai faire
        bal.retirer(req) --> recevoir(req,cs)
        traiter(req)
        balpriv[req.p].deposer( ) --> envoyer(req,cs)
        fin tant que
fin Serveur

processus Client
        ...
        prepare(req)
        bal.deposer(req) --> envoyer(req,sc)
        ...
        balpriv[moi-meme].retirer(req) --> recevoir(req,sc)
        ...
fin Client
```

Appel de procédure à distance :
Lors de l'appel d'une procédure côté client (talons/stub), on déroule plusieurs étapes :
- Emballage (marshalling).
- Envoi de la demande d'exe
- Attente de réception
- Déballage (unmarshalling)

Côté serveur : Un processus veilleur qui lance des processus pour exécuter les procédures qu'on lui demande.

Middleware :
- Outils
  - Corba
  - JAVA-RMI
  - .NET

Fonctions :
- Décrit l'interface
- Désignation de la procédure appelée
- Compiler les talons et lancer le processus serveur

2. Algorithmes distribués et synchronisation

Nouveaux problèmes :
Processus locaux sur chaque site
- Se synchroniser localement (P,V sur sémaphores, moniteurs)
- D'un site à l'autre, par ménage : envoyer(p,m) et recevoir(p,m)
- Les systèmes répartis nécessitent des synchronisation.

Exemple 1 : Atelier de soudure
Plusieurs robots effectue des soudures sur une chaîne de montage, ils ne peuvent pas tous souder en même temps car le système électrique ne le permet pas.
Résolution du problème par N-Exclusion (exclusion mutuelle tolérant N processus à la fois)

Exemple 2 : Système de gestion de fichiers réparti.
Les répertoires sont dupliqués localement. Les manipulations doivent donc soit être faites en exclusion mutuelle

Exclusion mutuelle :
Solution centralisée :
Un serveur d'exclusion mutuelle qui traite les requêtes des différents clients. Solution peu utilisée car si le serveur plante, plus rien ne peut être exécuté.

Jeton circulant sur anneau virtuel :
Les machines sont liées les unes les autres pour former une boucle. Chaque machine a un numéro unique, elle connaît la machine suivante dans l'anneau.
La machine qui a le jeton peut exécuter le processus. Une fois qu'elle a terminée elle donne le jeton au suivant.

Problème d'ordonner dans le temps les évènements :
Pas d'ordre total, pas d'ordre temporel absolu dans le système.
A chaque action on incrémente une horloge virtuelle, afin que toutes les machines du système soient synchronisées.

Système de gestion de fichiers

1) Présentation Générale

Intérêt des fichiers : stocker l'information dans une mémoire permanente. Les dispositifs ont une grande capacité mais un grand temps d'accès. Il faut nous permettre de conserver et de pouvoir retrouver l'information (Storage & Retrieval System).
Structurer l'information interne en fichier, structure logique : flot d'octet, enregistrement.

Fonctions :
- Créer
- Supprimer
- Dupliquer
- lire (buffer, emplacement dans le fichier)
- écrire (buffer, emplacement dans le fichier)

2) Organisation des catalogues, descripteurs, droits d'accès...

Un catalogue en un niveau :
« Désignation logique » : descripteur (localisation, organisation du fichier).

Un catalogue à deux niveaux :
Permet d'avoir le même nom de fichier pour deux utilisateurs différents.

Catalogues arborescents (Système unix par exemple) :
- Protection
- Liens possibles

Descripteur de fichiers :
- Localisation physique (les secteurs disques concernés)
- Organisation
- Droits d'accès
- Organisation logique des fichiers
- Nature de l'application associée
- Date création / modification ...

Descripteur local de fichiers :
ouvrir (...)
lire (...)
écrire (...)

Flot d'octet :
Les fichiers sont considérés comme une suite d'octets.

Programme simple pour copier un fichier.

```
/* Programme de copie de fichiers. La vérification des erreurs et leur
   ➥renvoi sont très basiques */
#include <sys/types.h>                  /* Définitions de types */
#include <fcntl.h>                      /* Définit des constantes O_RDONLY… */
#include <stdlib.h>                     /* Prototypes des appels système */
#include <unistd.h>                     /* Prototypes des appels système */
int main(int argc, char *argv[])        /* Prototype C ANSI */
#define TAILLE_BUFFER 4096              /* Utilisation d'une mémoire de
                                           ➥4 096 octets */

#define MODE 0700                       /* Mode de protection du fichier
                                           ➥(rw-rw-rw-) */

int main(int argc, char *argv[])        /* argc : nombre d'arguments
                                           ➥argv : pointeurs sur arguments */
{
   int df_src;                          /* Descripteur du fichier source */
   int df_cib;                          /* Descripteur du fichier cible */
   int nb_in,nb_out;                    /* Valeurs lues, écrites */
   char zone[TAILLE_BUFFER];            /* Zone mémoire */
   if (argc != 3) exit(1);              /* Nombre de paramètres incorrect */
   /* Ouvrir le fichier source et créer le fichier cible */
   df_src=open(argv[1],O_RDONLY); /* Ouverture du fichier source en
                                     ➥lecture seule */
  if (df_src<0) exit(2);               /* Impossible d'ouvrir le fichier
                                          ➥source */

  df_cib=creat(argv[2],MODE);          /* Création du fichier cible */
  if (df_cib<0) exit(3);               /* Impossible de créer le fichier
                                          ➥cible */
  /* Tout s'est bien passé, boucle de copie */
  while (TRUE) {                       /* Boucle infinie */
     /* Lecture depuis le fichier source */
     nb_in=read(df_src,zone,TAILLE_BUFFER;
     if (nb_in<=0) break;              /* Quitter en fin de fichier */
     /* Écriture dans le fichier cible */
     nb_out=write(df_cib,zone,nb_in);
     if (nb_out<=0) exit(4);           /* Quitter en cas d'erreur */
  }
  close(df_src);                       /* Fermeture du fichier source */
  close(df_cib);                       /* Fermeture du fichier cible */
  if (nb_in==0) exit(0);               /* Sortie sans erreur */
  else exit(5);                        /* Sortie avec une erreur en lecture*/
```

Mode d'accès séquentiel :
Accès direct : pointeur vers le premier octet du fichier à lire.
Sick (fid, position, nboctets)

3) Organisation logique des fichiers

Exemple : fichier employés
nom, prénom, etc ...
lire ou écrire : un enregistrement
mode d'accès : séquentiel, accès direct, séquentiel indexé.

4) Organisation physique des fichiers

Gestion du support secondaire (disques)
Disques : secteur numérotés de 1 à N.
Adresse des secteurs physiques : n°drive, n°cylin dre.

Passage de l'adresse logique à l'adresse physique.
Secteurs / blocs contigus : Un bloc contient 1, 2, 4 ou 8 secteurs consécutifs.

Problème d'allocation de blocs de taille variable : émiettement de la mémoire.

Mode d'accès :
- Création : réservation
- Écriture initiale
- Lire / écrire : écraser

Blocs chaînés : Les blocs libres sont chaînés entre eux (comme dans le TP1).
Intérêt : facilité d'allocation, augmenter la taille, insertion d'information...

Tables et implantation de fichiers (TIF), i-node unix

Gestion Mémoire

1) Introduction

Gestion de la mémoire centrale par rapport à la protection.

2) Système mono-utilisateur

Le recouvrement (overlay) permet d'éviter le dépassement mémoire lors du chargement d'un programme.
Les différentes procédures viennent se charger dans la mémoire les unes après les autres.
C'est une gestion mémoire explicite, c'est à dire gérée par le programmeur.

3) Système multiprogrammé

La mémoire est partitionné : chaque utilisateur ainsi que le système a son propre espace mémoire réservé.

4) Mémoire virtuelle

Notion de code translatable : Un code est translatable lorsqu'il n'y a pas d'adresses absolues, il peut alors être placé n'importe où dans la mémoire.

Problème : le code généré par le compilateur n'est pas translatable la plupart du temps.

Chaque programme a sa mémoire virtuelle ce qui permet d'exécuter le code dans n'importe quel bloc de mémoire. La mémoire virtuelle sera complètement indépendante de la mémoire physique : plusieurs blocs non contiguë, bloc sur le disque dur...
Elle est vue par l'utilisateur comme un bloc de l'adresse 0 à l'adresse N (modèle linéaire) ou comme une suite de segment (système, code, données statiques, pile...)