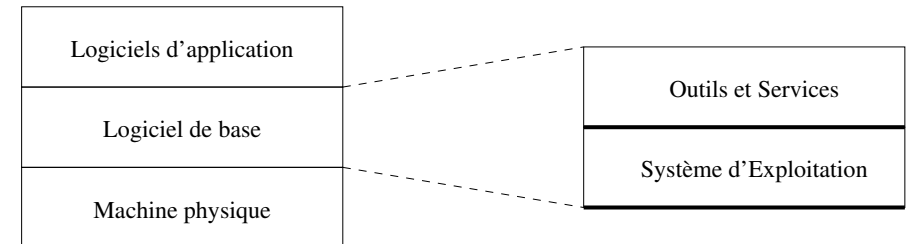


- Une première définition
- La gestion du temps
- Historique et évolutions
- Architecture interne d'un système d'exploitation

Un **système d'exploitation** est un **logiciel** destiné à **faciliter** et **optimiser** l'exploitation d'un ordinateur.

Structure en **couches** :



- **Sécurisation** : chaque couche isole les couches inférieures
- **Simplification** : gestion des périphériques
- **Abstraction** : remplace les objets réels par des objets virtuels
- **Optimisation** : allocation des ressources (CPU, mémoire, disques)
- **Partage** : multi-utilisateurs pour diminuer les coûts

Le système doit gérer les **différences de vitesse** et les **transferts** entre organes de la machine (entrée/sortie transparente).

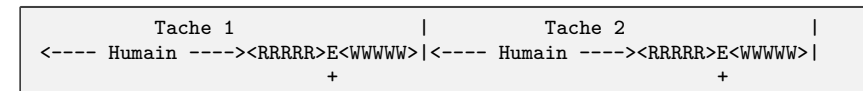
Une machine :

Organe	Rôles	Vitesse	Coût
Processeurs	Exécution	très rapide	très important
Mémoires	mémorisation temporaire	rapide	important
Périphériques	mémorisation définitive	très lent	faible

Un travail :

Partie	Acteur		Vitesse	%
Préparation	humain	H	extrêmement lent	80 %
Lecture des données	périphériques	R	très lent	10 %
Exécution	processeur	E	très rapide	2 %
Écriture des résultats	périphériques	W	très lent	8 %

- Le temps d'accès à la machine est structuré en période de 30 minutes
- Chaque période est allouée à un utilisateur
- À la fin de la période la machine est réinitialisée
- L'utilisateur doit attendre la période suivante

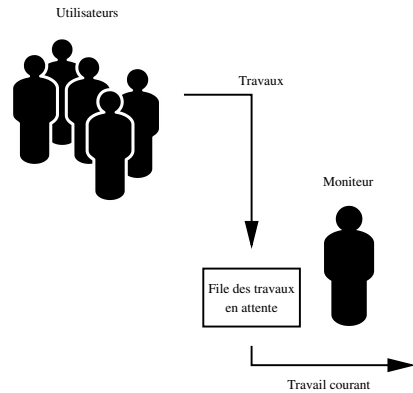


Avantage : Chaque utilisateur a la machine pour lui tout seul.

Inconvénients :

- Le travail doit être découpé en tranches de 30 minutes.
- La partie machine (20%) est faible.
- La partie processeur (2%) est très faible.

► **Moniteur humain de gestion des travaux** ◀



Il récupère les travaux, gère la **file d'attente** et enchaîne les exécutions.

```
T1 : <---- Humain ----><MM><RRRRR>E<WWWWW>
T2 :           <---- Humain ----><MM><RRRRR>E<WWWWW>
T3 :           <---- Humain ----><MM><RRRRR>E<WWWWW>
```

Conséquence : parallélisme entre temps humain et temps machine.

► **Début 50 : Le moniteur logiciel d'enchaînement des travaux** ◀

Ce **logiciel** enchaîne automatique les phases de

- compilation
- chargement en mémoire
- exécution

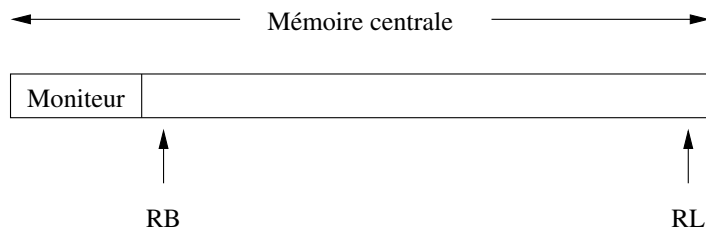
```
| chargement du moniteur compilateur
| compilation en mémoire du travail 1
| exécution du travail 1
| chargement du moniteur compilateur
| compilation en mémoire du travail 2
| exécution du travail 2
| ...
```

```
T1 : <---- Humain ----><RRRRR>E<WWWWW>
T2 :           <---- Humain ----><RRRRR>E<WWWWW>
T3 :           <---- Humain ----><RRRRR>E<WWWWW>
```

C'est le premier système d'exploitation : logiciel destiné à gérer la machine.

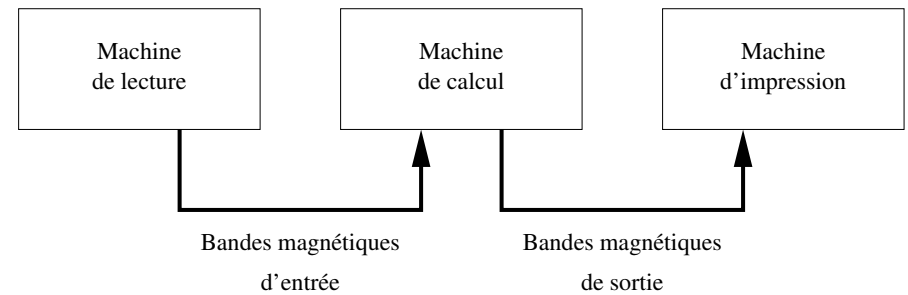
► **Milieu des années 50 : Le moniteur logiciel résidant** ◀

- enchaînement automatique des travaux
- protection de la mémoire
- limitation de durée
- supervision des entrées/sorties



► **Fin des années 50 : Le traitement par lots** ◀

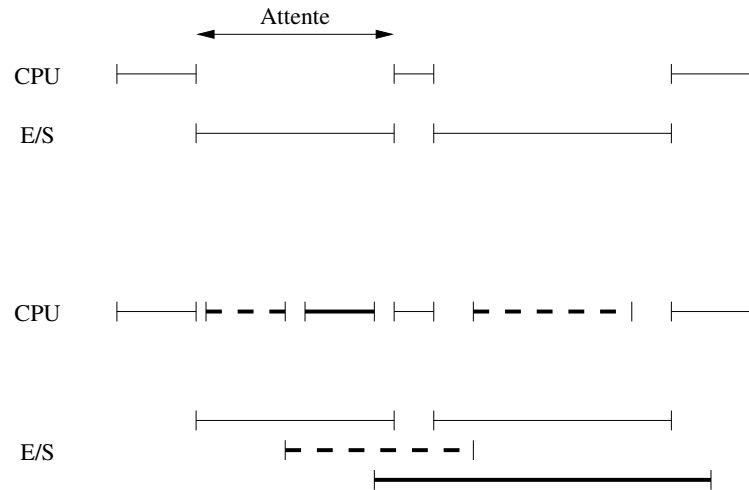
Apparition du **parallélisme** des tâches entre lecture, calcul et impression.



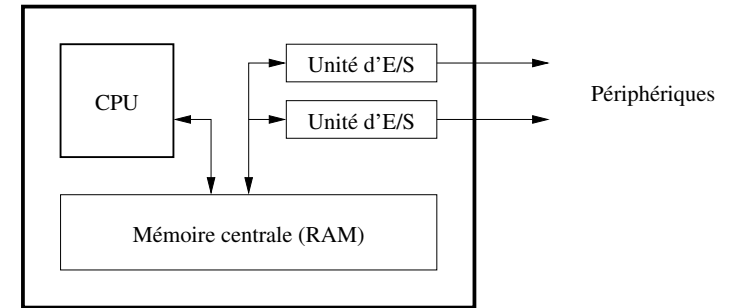
ML	MC	MI
Préparation du lot 1		
Préparation du lot 2	Exécution du lot 1	
Préparation du lot 3	Exécution du lot 2	Impression du lot 1

```
Tâche 1 : <---- Humain ----><RRRRR>E<WWWWW>
Tâche 2 :           <---- Humain ----><RRRRR>E<WWWWW>
Tâche 3 :           <---- Humain ----><RRRRR>E<WWWWW>
```

Chaque processus enchaîne des **cycles de CPU** (exécution de code) et des **cycles d'entrée/sortie** :



Multiprogrammation. Présence **simultanée** de **plusieurs programmes** en mémoire centrale.



Nouvelles caractéristiques :

- E/S tamponnées : définition d'un canal d'E/S,
- réimplantation du code,
- protection de la mémoire

- les terminaux sont connectés au serveur
- les utilisateurs travaillent devant le terminal
- ils partagent leur temps entre réflexion et action

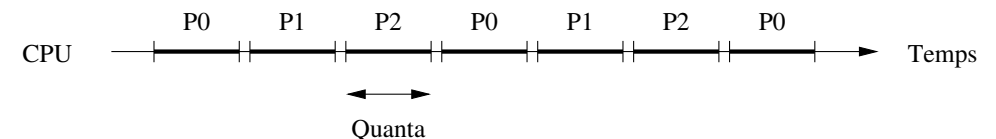
Hypothèse 1 : Le temps de réflexion est de 90% :

donc, sur 100 utilisateurs, **10 sont actifs**.

Hypothèse 2 : Les utilisateurs actifs réclament des actions simples :

prise en compte du travail interactif

Le temps d'exécution de la CPU est découpé en tranches :



Si

Quanta = 50 millisecondes **et** une requête ≤ 1 quanta

alors

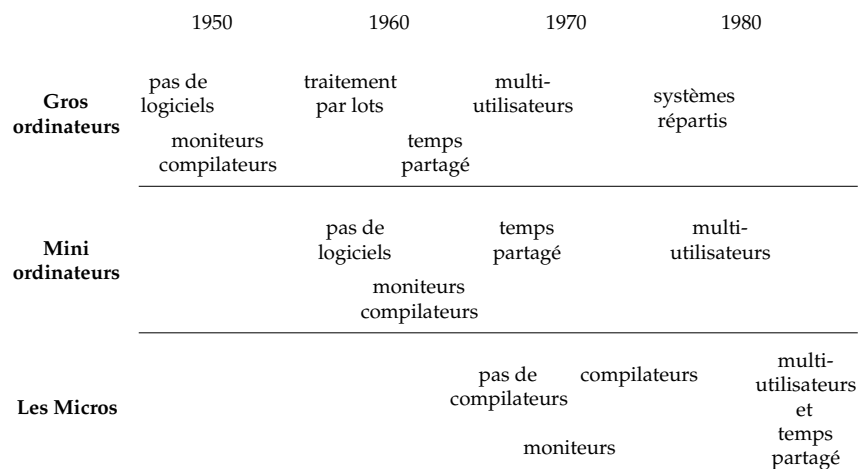
$$\text{Temps de réponse} = 10 \times 50 \text{ ms} = \frac{1}{2} \text{ s}$$

Contraintes :

- multiprogrammation,
- temps de commutation faible,
- possibilité d'interruption **propre**.

Répartition d'une activité entre **plusieurs machines** reliées par réseau.

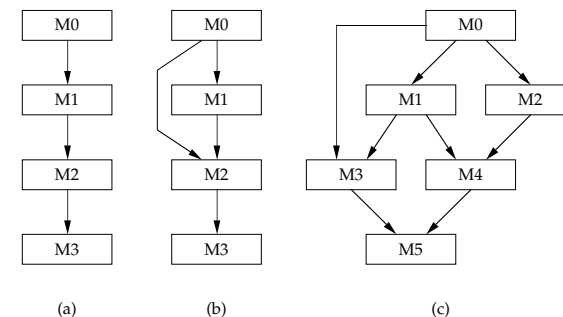
Exemple : Le **Cluster**. La somme de plusieurs machines est vue comme une seule machine (simplicité, évolutivité, robustesse).



Une **machine** c'est

- des objets,
- des actions possibles,
- des règles de composition des actions.

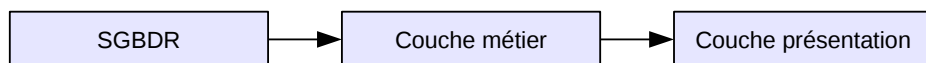
Dans une **structure en couches**, le système est conçu comme un empilement de machines.



Avantages : Sécurité et modularité.

Les **applications modernes** utilisent les **architectures en couches**.

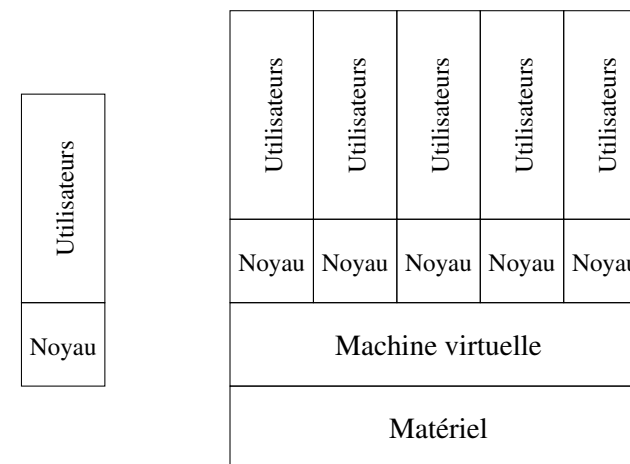
Un exemple : l'architecture applicative **3-tiers** :



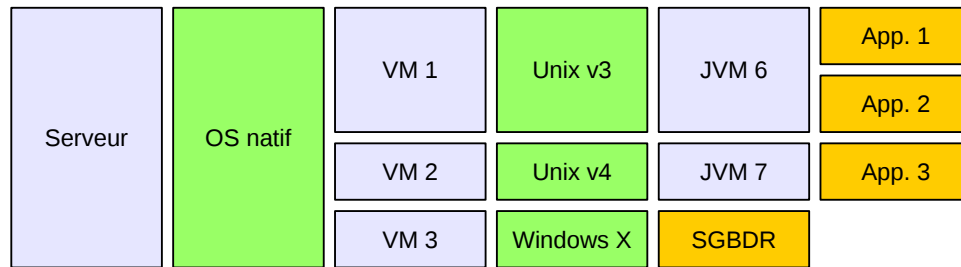
- une couche possède une spécification (interface d'entrée),
- une couche masque les couches inférieures.

Avantage : capacité à développer en parallèle les couches. (diminution du temps de développement).

Les machines **virtuelles** d'IBM (VM)/CMS :



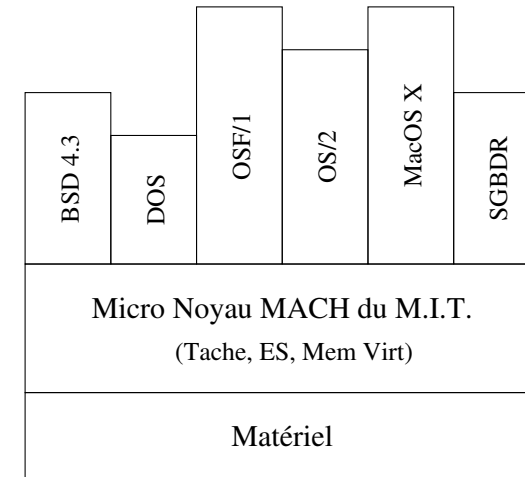
La **virtualisation** est très utilisée dans le déploiement des **applications**.



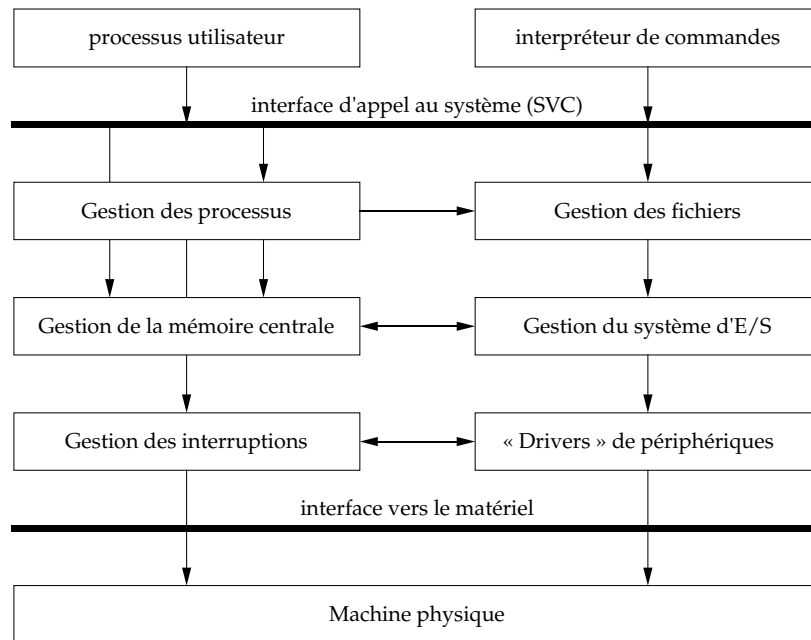
Avantages :

- définir des environnements d'exécution adaptés
- assurer l'évolutivité de son parc applicatifs
- améliorer la sécurité
- exploiter des architectures matérielles sophistiquées

L'architecture **Micro-noyau** par opposition aux structures **monolithiques** des systèmes



Avantages : **Portabilité, Qualité.**



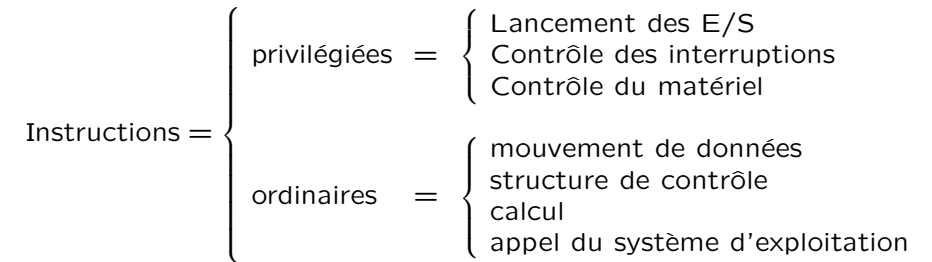
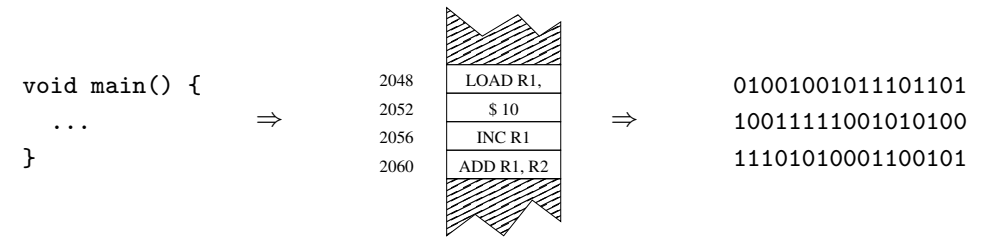
Présentation des processus

- Notion de programme
- Notion de machine
- Exécution d'un programme
- Processus

1

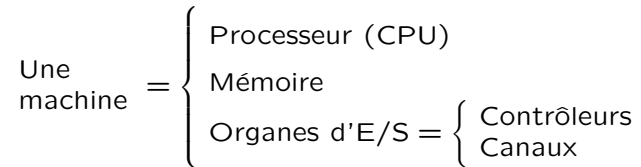
► Notion de programme ◀

Un **programme** est une suite d'instructions rangées en mémoire :



2

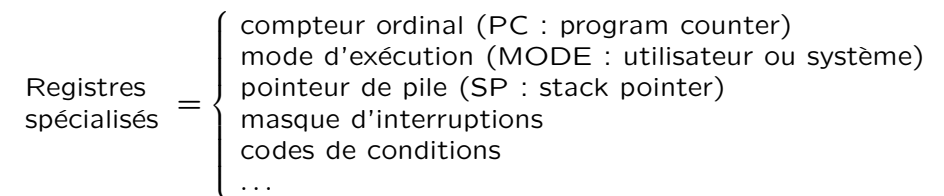
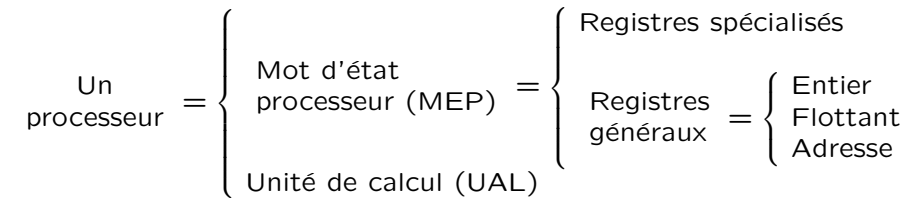
► Notion de machine ◀



L'**état d'une machine** c'est l'état de ses composants.

3

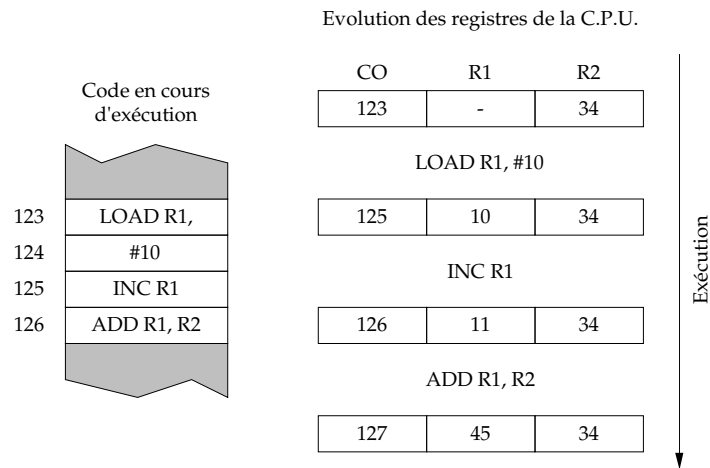
► Description du processeur ◀



- en mode **utilisateur** les instructions privilégiées sont interdites
- en mode **système** toutes les instructions sont utilisables

4

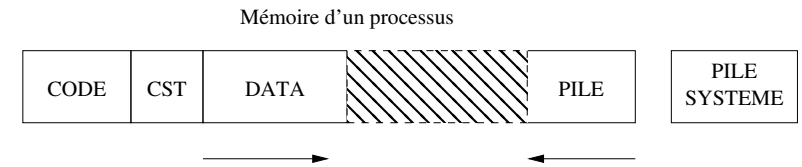
Une **exécution** c'est une évolution **discrète** de l'état de la machine.



Notion de points **observables**.

Un **processus** est un programme en cours d'exécution. Il est défini par :

- des zones mémoire (code, pile et données),



- la valeur des registres de la CPU (un MEP),
- un ensemble de ressources allouées,

Le tout forme le **contexte d'exécution** du processus.

Le mécanisme des interruptions

- Rendez-vous asynchrone
- Interruption d'un processus
- Interruption matérielle
- Les appels système
- Les déroutements

1

► Rendez-vous asynchrone ◀

Problème : Exécuter efficacement un programme et, en même temps, prendre en compte rapidement un événement imprévisible.

Exemple :

- **Le travail** :
 - ▷ exécuter un processus consommateur de CPU (pas d'entrée/sortie)
- **Les événements à gérer** :
 - ▷ arrivée de données sur la carte réseau
 - ▷ déplacement de la souris
 - ▷ activation du clavier
 - ▷ apparition d'une alarme
 - ▷ fin d'une lecture sur disque

2

► Attente active ◀

Solution : l'attente active. Soit **F** un indicateur qui signale l'événement

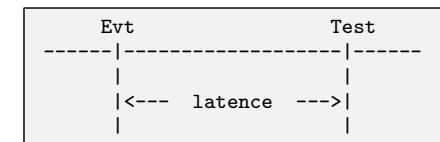
```
...  
si (F = 1) alors  
  F := 0  
  < traiter l'événement >  
fin si  
...
```

Remarques :

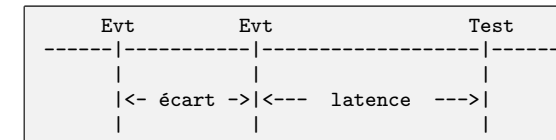
- le test **explicite** est pénible et peu sécurisé
- perte de temps processeur

3

- pas de réaction immédiate



- risque de collision



écart minimum entre deux événements > latence

4

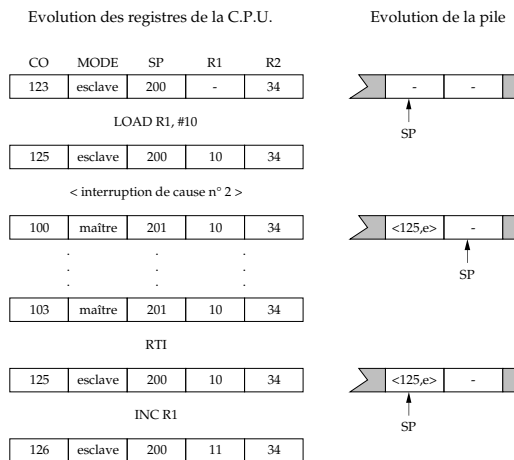
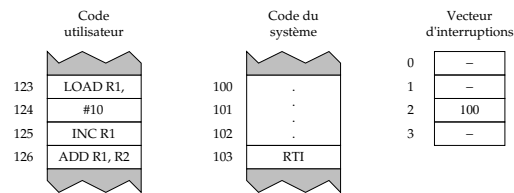
- L'**interruption** est une opération **indivisible** réalisée par le **processeur** et **liée à cause** identifiée par un numéro. Cette opération change le **PC** et le **MODE**.

$$\text{interruption de cause } k = \begin{cases} \text{PUSH PC} & // \text{ sauvegarde de PC} \\ \text{PUSH MODE} & // \text{ sauvegarde de MODE} \\ \text{MODE} := \text{ système} & // \text{ passage en mode système} \\ \text{PC} := \text{ vi}[k] & // \text{ branchement sur vi}[k] \end{cases}$$

Le **vecteur d'interruptions** (**vi**) est un table d'adresses.

- Les interruptions sont déclenchées **uniquement** sur les points observables (points **interruptibles**).
- reprise après interruption :

$$\text{instruction RTI} = \begin{cases} \text{POP MODE} \\ \text{POP PC} \end{cases}$$



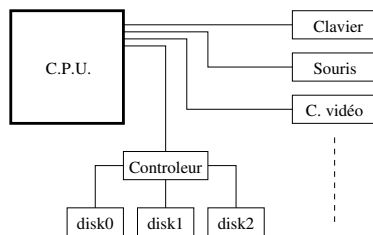
Le code exécuté après une interruption est appelé le **traitant** de cette interruption.

Les traitants ont la structure suivante :

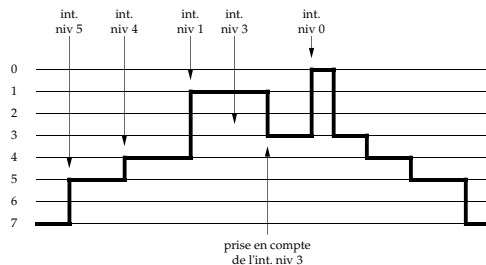
1. **sauvegarder** du contexte
2. **traiter** de la cause
3. **choisir** le prochain processus **P** à exécuter
4. **restaurer** le contexte de **P**
5. **relancer** le processus **P** (instruction **RTI**)

► Interruption matérielle : principe ◀

Un **signal électrique** est émis vers le processeur qui provoque une interruption.



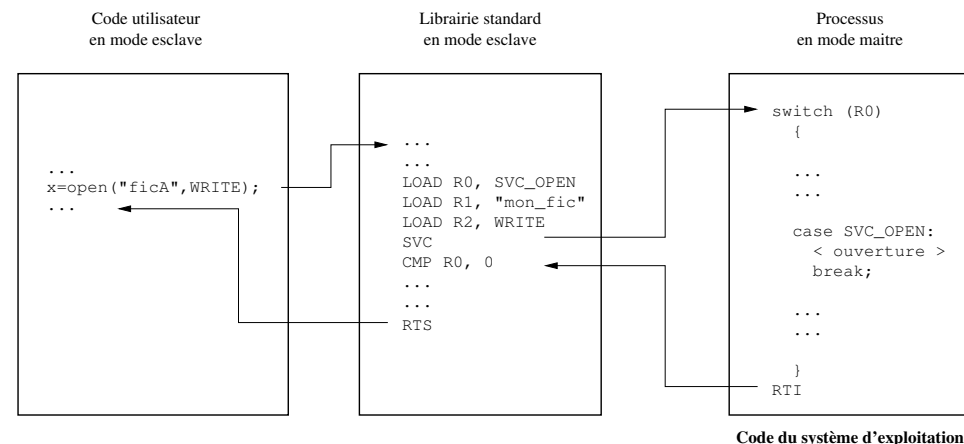
Le numéro associé à chaque interruption matérielle indique une **priorité** :



9

► Les appels systèmes : principe ◀

Les **trappes** (Trap) : Appel explicite du système par le biais d'une instruction qui **déclenche une interruption**.



10

► Les appels systèmes : Avantages ◀

- Changement de **mode** et nombre de points d'entrée est **limité**
- On obtient un **sas** qui isole le système d'exploitation
- Définition d'une **nouvelle machine** (ajout d'instructions)
- Une librairie standard offre
 - ▷ une interface système **agréable et simplifiée**
 - ▷ une interface **indépendante** du système d'exploitation
 - ▷ **l'implantation réelle** des appels systèmes en assembleur
- Deux modes d'exécution :

```
$ time bash -c 'for((i=0; i<500000; i++)); do true; done'
real 0m2,331s
user 0m2,330s
sys 0m0,000s
```

```
$ time cp fichier_1Go.txt copie.txt
real 0m7,682s
user 0m0,000s
sys 0m0,932s
```

11

► Déroutements : Principe ◀

L'objectif des déroutements est double :

- traitement **systématique** des erreurs ou des situations anormales,
- **protection** et bonne utilisation de la machine (les processus s'exécutent sous surveillance).

Si l'exécution d'une instruction **produit une erreur**, alors

→ il y a **interruption** et **branchement** vers le code de traitement des erreurs du système.

Les **causes** possibles sont :

- donnée incorrecte (division par 0)
- opération interdite (instruction privilégiée)
- instruction inconnue
- accès mémoire incorrect

Utilisations :

- mode TRACE (debugger)
- gestion des erreurs (**signal**)
- machine virtuelle

12

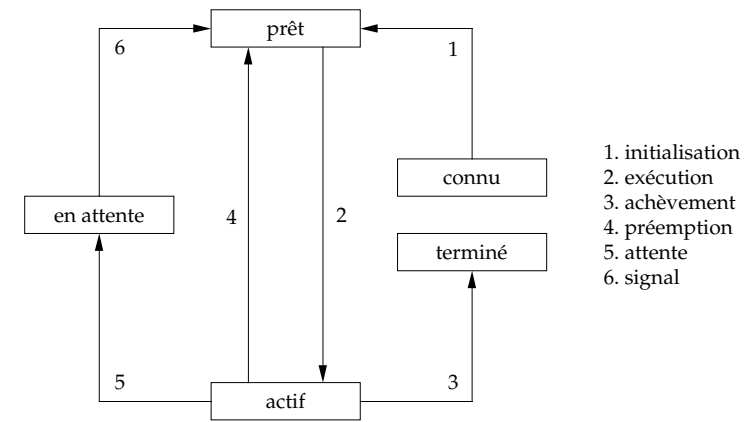
Gestion des processus

- État d'un processus
- Représentation d'un processus
- Les «threads» (processus de poids léger)
- Utilité des «threads»
- Implantation des «threads»

1

► État d'un processus ◀

Chaque processus est dans l'un des **états opérationnels** suivants :



2

► Représentation d'un processus ◀

Pour chaque processus le système maintient :

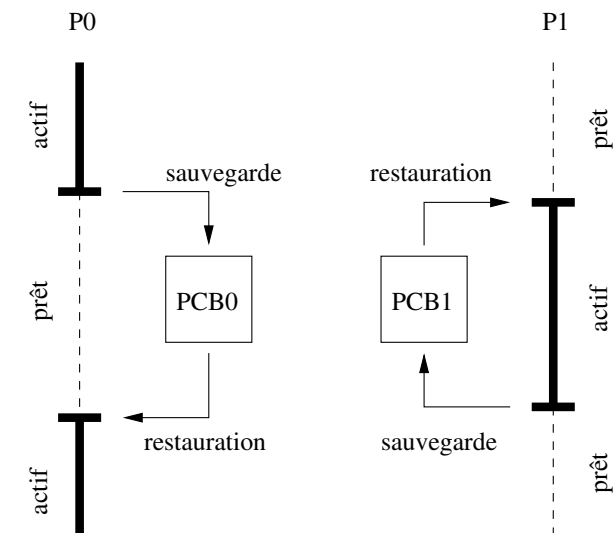
- un identifiant (`pid`)
- un état opérationnel
- un contexte d'exécution
- des informations diverses :
 - ▷ priorités
 - ▷ filiations
 - ▷ propriétaire
- des statistiques :
 - ▷ temps CPU
 - ▷ nombre d'E/S
 - ▷ nombre de défauts de page

Ces informations sont rangées dans un PCB (**Process Control Block**).

3

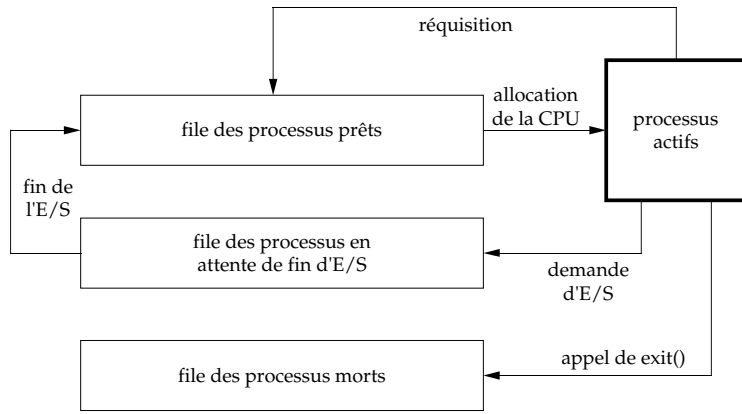
► Rôle du PCB ◀

Le rôle du PCB dans la commutation entre deux processus :



5

Les PCB sont rangés dans **des files** :



Création de processus :

- chaque processus peut **créer** des processus (ses fils),
- il est possible de **figer**, et de **reprendre** des processus.
- il est possible de **tuer** des processus ou demander leur arrêt.

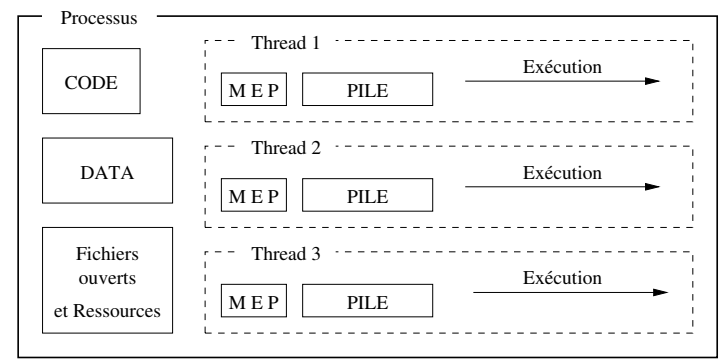
Communication entre processus :

- prévoir des **outils de communication** (fichier, tube, socket, mémoire partagée, envoi de messages)

Synchronisation de processus :

- prévoir des **outils de synchronisation** (verrou, sémaphore, moniteur)

Un **thread (fil)** est un programme en cours d'exécution qui partage son code et ses données.



Chaque thread a une pile d'exécution **autonome**.

Un processus est composé de **threads**.

```

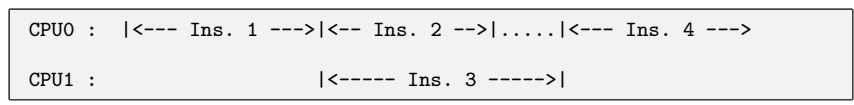
begin
  Instruction 1
  co-begin
    Instruction 2
    Instruction 3
  co-end
  Instruction 4
end
    
```

```

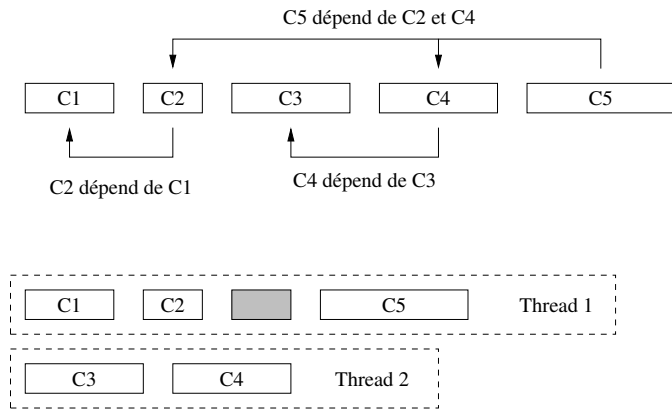
void travail() {
  int t;

  instruction1();
  if ((t = thread()) == 0) {
    instruction3();
    thread_exit();
  }
  instruction2();
  thread_join(t);
  instruction4();
}
    
```

Trace de l'exécution :

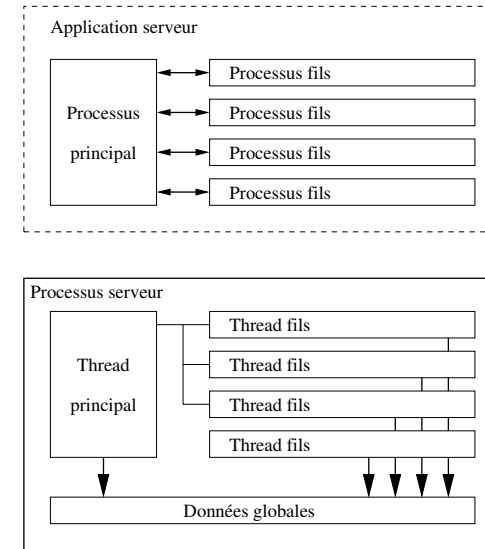


Étude des dépendances et découpage :



Avantages :

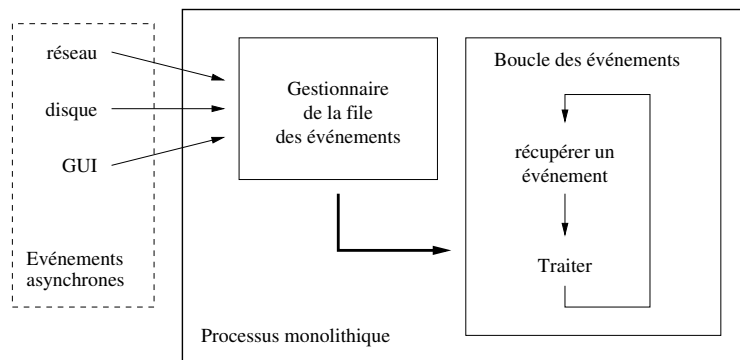
- récupération des temps d'E/S,
- exploitation des machines multi-processeurs,
- coopération entre threads.



La **commutation** et la **communication** entre threads est une opération plus simple

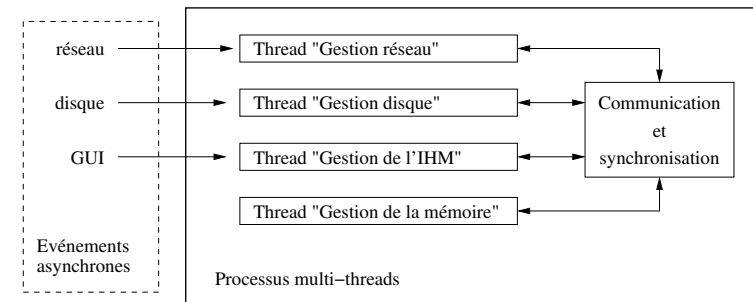
Il existe un **conflit** entre entrées/sorties synchrones (**bloquantes**) et interface homme/machine (**IHM**).

Solution **d'attente active** : E/S asynchrones (**non-bloquantes**) et structure avec **boucle d'événements**.



Il existe un **autre solution** basée sur

- le découpage en plusieurs **threads**,
- une utilisation des E/S **synchrones**,
- un module de **communication**.



Avantages :

- plus grande simplicité du code,
- indépendance entre les modules.

Une **interruption** provoque une perte de la CPU. Nous pouvons donc redistribuer le processeur.

Objectifs :

- équité,
- débit maximum
- maximum de processus interactifs,
- rationaliser la gestion des ressources,
- favoriser les «bons» processus,
- améliorer les temps de réponse moyen,

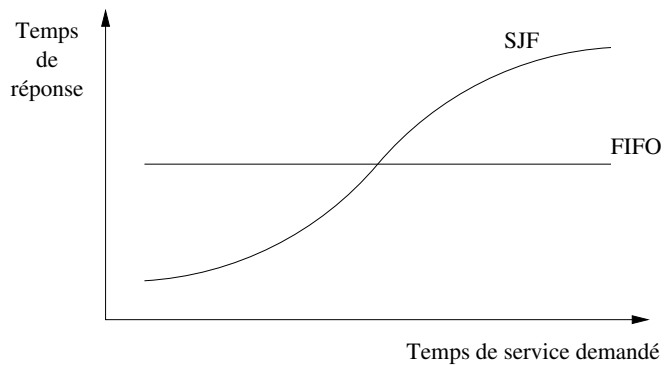
FIFO (First In First Out) : absence de réquisition et croissance rapide du temps de réponse.



$$\text{temps de réponse} = (\text{temps d'attente} + \text{temps d'exécution})$$

$$\text{temps d'attente} = (\text{taille de la file FIFO} \times \text{durée moyenne d'exécution})$$

SJF (Short Job First) : absence de réquisition et risque de privation.



SRT (Shortest Remaining Time) : c'est SJF + un mécanisme de réquisition.

On choisit le processus qui a le temps restant d'exécution le plus petit. L'arrivée d'un nouveau processus peut interrompre le processus courant.

HRN (Highest Response Ratio Next) : Le choix est basé sur le ratio

$$p(t) = \frac{w(t) + t_s}{t_s}$$

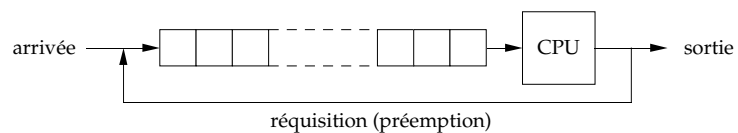
avec

- $w(t) = t - t_a$: temps d'attente,
- t_a : date d'arrivée,
- t_s : temps d'exécution estimé.

Si $w(t) = 0$, les travaux les plus courts (t_s) sont privilégiés.

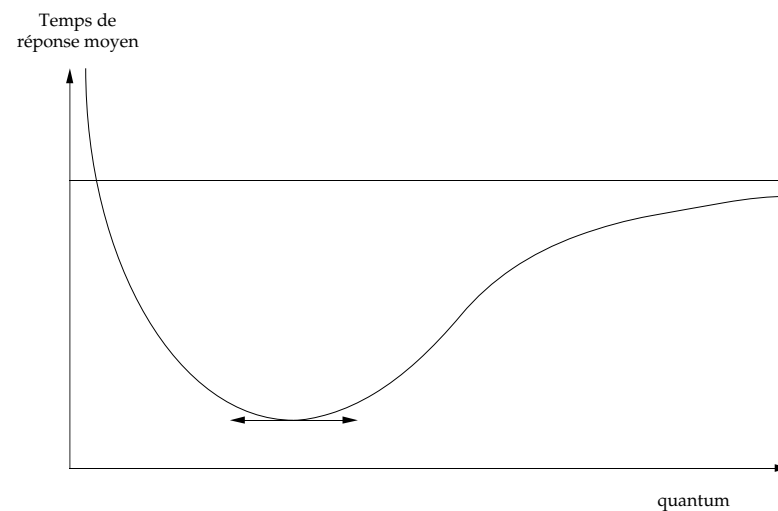
► Algorithmes d'allocation du tourniquet ◀

Tourniquet (round robin) : c'est FIFO plus un mécanisme de réquisition.



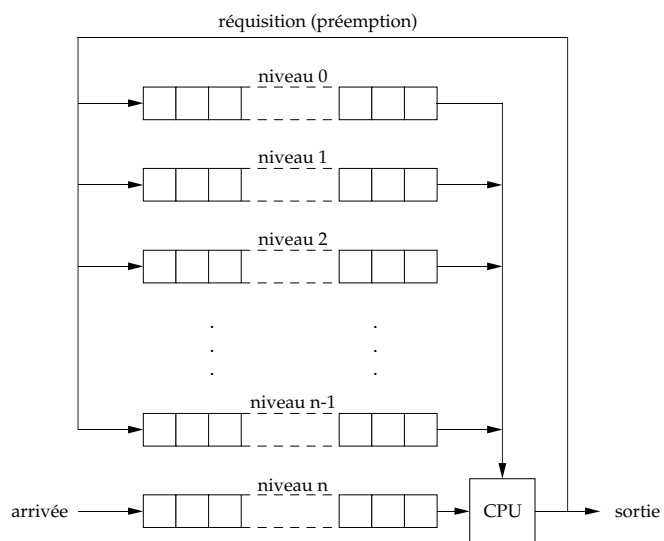
19

Comment fixer la durée du quanta :



20

► Tourniquet Multi-niveaux ◀



21

► Tourniquet Multi-niveaux avec priorités ◀

Il existe des classes de processus :

- processus systèmes,
- processus temps réel,
- processus interactifs,
- processus d'édition interactive,
- processus «batch».

Chaque classe est **absolument** prioritaire sur celles de niveau inférieur.

A l'intérieur de chaque classe, les processus sont rangés dans un système de files multi-niveaux.

22

Le problème :

- Accès à des ressources partagées
- Les sections critiques
- Le problème de l'exclusion mutuelle

Solutions d'attente active :

- Une solution d'attente active
- Solution matérielle

Solutions de manipulation des processus :

- Les verrous
- Les sémaphores
- Le producteur et le consommateur
- Les sémaphores à messages
- Les régions critiques

Soit une pile **partagée** par plusieurs threads :

```

pile = structure
| sommet : entier ;
| data : tableau [ 1 .. Max ] de entier ;

procédure empiler( var p : pile ; d : entier )
| p.sommet := p.sommet + 1 ;
| p.data[ p.sommet ] := d ;
    
```

Il est possible que nous ayons :

```

thread1 : empiler( p, 10 )
thread1 : p.sommet := p.sommet + 1 ;
— — interruption — —
thread2 : empiler( p, 20 )
thread2 : p.sommet := p.sommet + 1 ;
thread2 : p.data[ p.sommet ] := 20 ;
      :                               :
— — retour au thread 1 — —
thread1 : p.data[ p.sommet ] := 10 ;
    
```

- Les ressources qui posent des problèmes sont dites **critiques**.
- Les portions de code qui manipulent ces ressources critiques sont appelées des **sections critiques**.

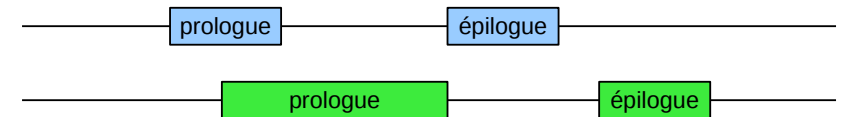
Forme des programmes :

⟨initialisation⟩	exécuté une seule fois
:	
⟨prologue⟩	identique pour tous les processus
⟨ section critique ⟩	au plus un processus
⟨épilogue⟩	identique pour tous les processus

Conflit avec deux threads. Le prologue et l'épilogue encadrent la section critique :



Le problème est réglé en augmentant la durée d'exécution du prologue :



Contraintes :

- Les processus ne sont pas bloqués sans raison (absence de **privation**).

La procédure empiler devient :

```

procédure empiler( var p : pile; d : entier )
  (prologue)
  |
  |   p.sommet := p.sommet + 1;
  |   p.data[ p.sommet ] := d;
  |
  (épilogue)
    
```

Nouvelle exécution :

```

thread1 : empiler( p, 10 )
thread1 : (prologue)
thread1 : p.sommet := p.sommet + 1;
      — — interruption — —
thread2 : empiler( p, 20 )
thread2 : (prologue)
      — — blocage du thread 2 — —
      :
      :
      :
      — — retour au thread 1 — —
thread1 : p.data[ p.sommet ] := 10;
thread1 : (épilogue)
      :
      :
      :
      — — reprise du thread 2 — —
thread2 : p.sommet := p.sommet + 1;
thread2 : p.data[ p.sommet ] := 20;
thread2 : (épilogue)
    
```

Soit `libre` une variable `partagée` de type booléenne :

```

<init>      (1) libre := vrai;
            :
            :
<prologue> (2) si (libre = faux) aller en (2)
            (3) libre := faux
            :
            :
            (4) <section critique>
            :
<épilogue> (5) libre := vrai
    
```

Critique 1 : Consommation de temps processeur.

Critique 2 : L'exclusion mutuelle n'est pas toujours respectée :

2 <interruption> 2' 3' 4' ... <retour au rouge> 3 4 ...

On introduit une nouvelle instruction `Test and Set` pour

- garantir l'atomicité d'une modification,
- construire une solution basée sur l'attente active valable en multi-processeurs :

Définition de `Test and Set` :

```

instruction TAS(var m : entier , var verrou : entier )
  (bloquer la case mémoire verrou)
  |
  |   m := verrou
  |   verrou := 0
  |
  (débloquer la case mémoire verrou)
  CO := CO + (taille de l'instruction TAS)
    
```

Codage de l'exclusion mutuelle avec une variable partagée `mutex` et une variable privée `p`.

```

<initialisation> (1) mutex := 1;
<prologue>      (2) répéter
                (3)   | TAS(p, mutex);
                (4) jusqu'à (p = 1)
                :
                :
                (5) <section critique>
                :
<épilogue>      (6) mutex := 1;
    
```

Critiques :

- Consommation de temps processeur.
- Risque de privation (cela peut s'arranger).
- Le processeur doit garantir l'exclusion mutuelle.
- C'est une solution utilisable uniquement sur des `séquences brèves`.

Objectifs :

- ne plus perdre de temps CPU,
- simplicité de la solution.

Définition des verrous :

Verrou : **structure**
 | **booléen** *libre* := **vrai**
 | **file fifo** *processus* := {}

Un **verrou** est une structure de donnée **partagée** du système d'exploitation.

Pour un verrou donné, les procédures s'exécutent en **exclusion mutuelle**.

```

procédure prendre(Verrou v)
    | si (v.libre = faux ) alors
    |   | <soit P le processus appelant>
    |   | <entrer P dans la file v.processus>
    |   | <suspendre le processus P>
    |   | sinon
    |   |   | v.libre := faux ;
    |   | fin si
    | fin si

procédure libérer(Verrou v)
    | si <la file v.processus est vide> alors
    |   | v.libre := vrai ;
    |   | sinon
    |   |   | v.libre := faux ;
    |   |   | <sortir un processus Q de la file v.processus>
    |   |   | <réveiller le processus Q>
    |   | fin si
    | fin si
    
```

Le code de l'exclusion mutuelle s'écrit

```

<initialisation>   (1)  mutex := nouveau Verrou ;

<prologue>        (2)  prendre(mutex) ;
                   |
                   |  <section critique>
                   |
                   |  <épilogue>
<épilogue>        (3)  libérer(mutex) ;
    
```

Soit deux processus qui partagent **deux ressources** :

```

<initialisation>   (1)  mutex1 := nouveau Verrou ;
                   (2)  mutex2 := nouveau Verrou ;

<rouge>           (10)  prendre(mutex1) ;
                   (11)  | prendre(mutex2) ;
                   (12)  |   | <section critique>
                   (13)  |   | libérer(mutex2) ;
                   (14)  | libérer(mutex1) ;

<bleu>            (20)  prendre(mutex2) ;
                   (21)  | prendre(mutex1) ;
                   (22)  |   | <section critique>
                   (23)  |   | libérer(mutex1) ;
                   (24)  | libérer(mutex2) ;
    
```

Il y a blocage pour **10** <interruption> **20 21** <blocage> **11** <blocage>

Un verrou ne sait pas compter... donc, il faut remplacer le drapeau par un compteur (Dijkstra).

Un **sémaphore** est une structure de donnée **partagée** du système d'exploitation.

Sémaphore : **structure**

```
compteur : entier ;
processus : file FIFO ;
```

procédure init(**var** s : Sémaphore; *compteur_initial* : entier)

```
vérifier que compteur_initial ≥ 0 ;
s.compteur := compteur_initial ;
s.processus := {} ;
```

Pour un sémaphore donné, les deux procédures ci-dessous s'exécutent en **exclusion mutuelle**.

procédure P(**var** s : sémaphore)

```
s.compteur := s.compteur - 1
si (s.compteur < 0) alors
    <soit P le processus appelant>
    <entrer le processus P dans la file s.processus>
    <suspendre le processus P>
fin si
```

procédure V(**var** s : sémaphore)

```
s.compteur := s.compteur + 1
si (s.compteur ≤ 0) alors
    <sortir un processus Q de la file s.processus>
    <reprendre le processus Q>
fin si
```

P pour *proberen* (tester) ou *wait*
V pour *verhogen* (incrémenter) ou *signal*

Soient **trois processus** qui exploitent la même ressource :

	P_1	P_2	P_3
(1)	P(s)	⋮	⋮
(2)	⋮	P(s)	⋮
(3)	⋮	⋮	P(s)
(4)	V(s)	⋮	⋮
(5)	⋮	V(s)	⋮
(6)	⋮	⋮	V(s)

La trace de l'exécution donne :

	action	(2, { })	P_1	P_2	P_3
(1)	P(s)	(1, { })	SC	A	A
(2)	P(s)	(0, { })	SC	SC	A
(3)	P(s)	(-1, { P_3 })	SC	SC	S
(4)	V(s)	(0, { })	A	SC	SC
(5)	V(s)	(1, { })	A	A	SC
(6)	V(s)	(2, { })	A	A	A

Il existe un tampon de taille **limité** entre le producteur et le consommateur.

Algorithme du producteur :

```
répéter
    <produire un message>
    <le déposer dans le tampon>
jusqu'à ...
```

Algorithme du consommateur :

```
répéter
    <prélever un message depuis le tampon>
    <le consommer>
jusqu'à ...
```

Le consommateur consomme si le tampon n'est pas vide :

N_{Plein} : sémaphore = $(0, \{\})$

producteur :

répéter
 | \langle produire un message
 | \langle le déposer dans le tampon
 | $V(N_{Plein})$;
jusqu'à ...

Consommateur :

répéter
 | $P(N_{Plein})$;
 | \langle consommer
jusqu'à ...

Définition des sémaphores :

N_{Plein} : sémaphore = $(0, \{\})$
 N_{Vide} : sémaphore = $(n, \{\})$

Le producteur produit si le tampon n'est pas plein :

répéter
 | $P(N_{Vide})$;
 | \langle produire un message
 | \langle le déposer dans le tampon
 | $V(N_{Plein})$;
jusqu'à ...

Le consommateur consomme si le tampon n'est pas vide :

répéter
 | $P(N_{Plein})$;
 | \langle consommer
 | $V(N_{Vide})$;
jusqu'à ...

Définition :

séma-mesg : **structure**
 | compteur : entier;
 | demandeurs : file FIFO de processus;
 | messages : file FIFO de messages;

Procédures :

procédure P_m (**var** s : séma-mesg ; **var** m : données)
 | $s.compteur := s.compteur - 1$
 | **si** ($s.compteur < 0$) **alors**
 | | \langle soit P le processus appelant
 | | \langle entrer le processus P dans la file $s.demandeurs$
 | | \langle suspendre le processus P
 | **fin si**
 | \langle sortir m de la file $s.messages$

procédure V_m (**var** s : séma-mesg ; m : données)
 | \langle entrer m dans la file $s.messages$
 | $s.compteur := s.compteur + 1$
 | **si** ($s.compteur \leq 0$) **alors**
 | | \langle sortir un processus Q de la file $s.demandeurs$
 | | \langle reprendre le processus Q
 | **fin si**

► Les régions critiques ◀

Présentation :

```
région p quand (condition)  
| <région critique>  
fin de région
```

Exécution de la région critique en **exclusion mutuelle** sur la variable **p** quand la condition est **vérifiée**.

22

Soit une pile partagée par plusieurs threads :

```
pile = structure partagée  
| sommet : entier ;  
| data : tableau [ 1 .. Max ] de entier ;
```

```
procédure empiler( var p : pile ; d : entier )  
| région p quand (sommet < Max)  
|   sommet := sommet + 1 ;  
|   data[ sommet ] := d ;  
fin de région
```

```
procédure dépiler( var p : pile ; var d : entier )  
| région p quand (sommet > 0)  
|   d := data[ sommet ] ;  
|   sommet := sommet - 1 ;  
fin de région
```

23

► Une version en Java ◀

```
public class Stack {  
  
    final int values[];  
    int counter = 0;  
  
    public Stack(int size) { this.values = new int[size]; }  
  
    synchronized public void push(int value) throws InterruptedException {  
        // attendre que la pile ne soit pas pleine  
        while (counter == values.length) { wait(); }  
        values[counter++] = value;  
        notifyAll();  
    }  
  
    synchronized public int pop() throws InterruptedException {  
        // attendre que la pile ne soit pas vide  
        while (counter == 0) { wait(); }  
        notifyAll();  
        return values[--counter];  
    }  
}
```

24