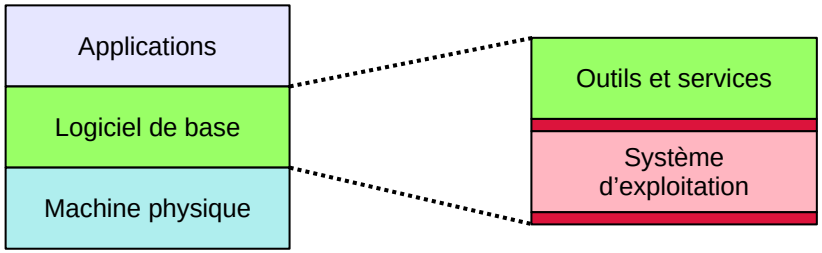
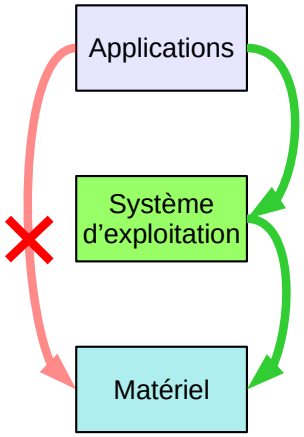
►► Système d'exploitation : première définition ◄◄

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

Structure en **couches** :

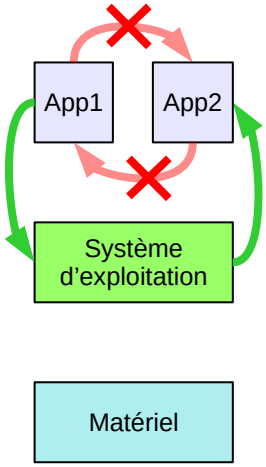


► Organisation en couches ◄



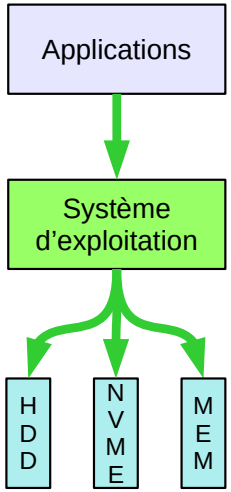
- **Sécurisation** : chaque couche isole les couches inférieures
- **Simplification** : gestion des périphériques

► Isolation des applications ◄



- **Sécurisation** : les applications sont indépendantes
- **Optimisation** : allocation des ressources (CPU, mémoire, disques)
- **Partage** : multi-utilisateurs pour diminuer les coûts

► Simplifications ◄



- **Simplification** : gestion des périphériques
- **Abstraction** : remplace les objets réels par des objets virtuels
  - ▷ **CPU** / Processus
  - ▷ **Mémoire** / Mémoire virtuelle
  - ▷ **Imprimante** / Queue d'imp.
  - ▷ **Machine** / Machine virtuelle

▸▸ La gestion du temps ◂◂

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

Une machine :

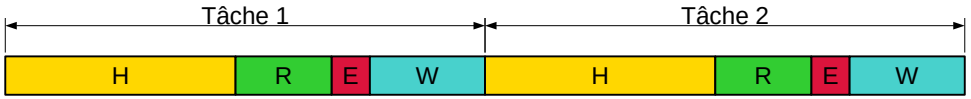
Organe	Rôles	Vitesse	Coût
Processeur	Exécution	très rapide	très important
Mémoire	mémorisation temporaire	rapide	important
Périphérique	mémorisation définitive	très lent	faible

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 %

▸ Années 50 : Organisation en porte ouverte ◂

- 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.

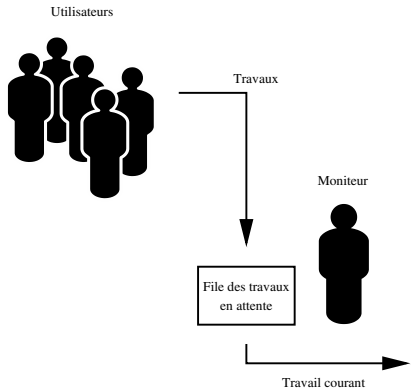
Inconvénients :

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

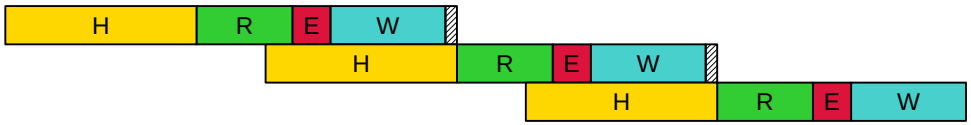
▸ Moniteur humain de gestion des travaux ◂

Le moniteur humain de gestion des travaux

- récupère les travaux,
- gère la **file d'attente**
- enchaîne les exécutions



Conséquence : parallélisme entre temps humain et temps machine.

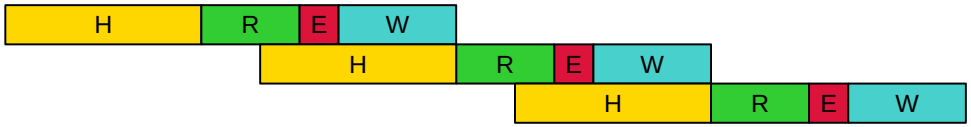


▸ Début 60 : Le moniteur logiciel d'enchaînement des travaux ◂

Ce **logiciel** enchaîne automatique les phases de

- compilation
- chargement en mémoire
- exécution

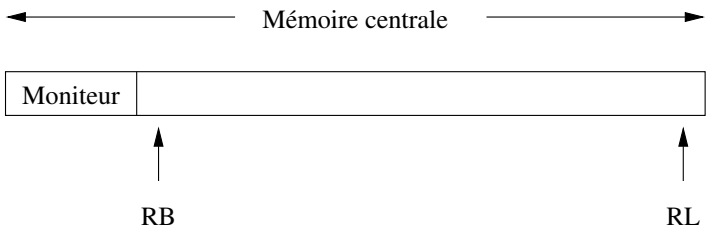
```
Travail 1 | chargement du moniteur compilateur
          | compilation en mémoire du travail 1
          | exécution du travail 1
Travail 2 | chargement du moniteur compilateur
          | compilation en mémoire du travail 2
          | exécution du travail 2
          | ...
```



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

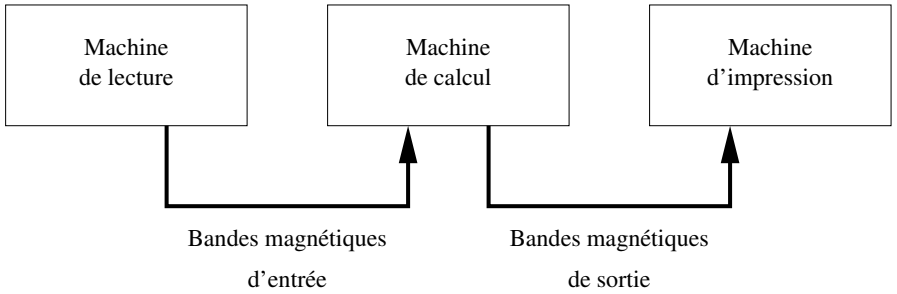
▶ Le moniteur logiciel résidant ◀

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

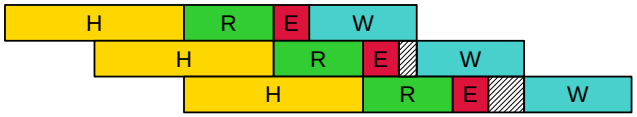


▶ Années 60 : Le traitement par lots ◀

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

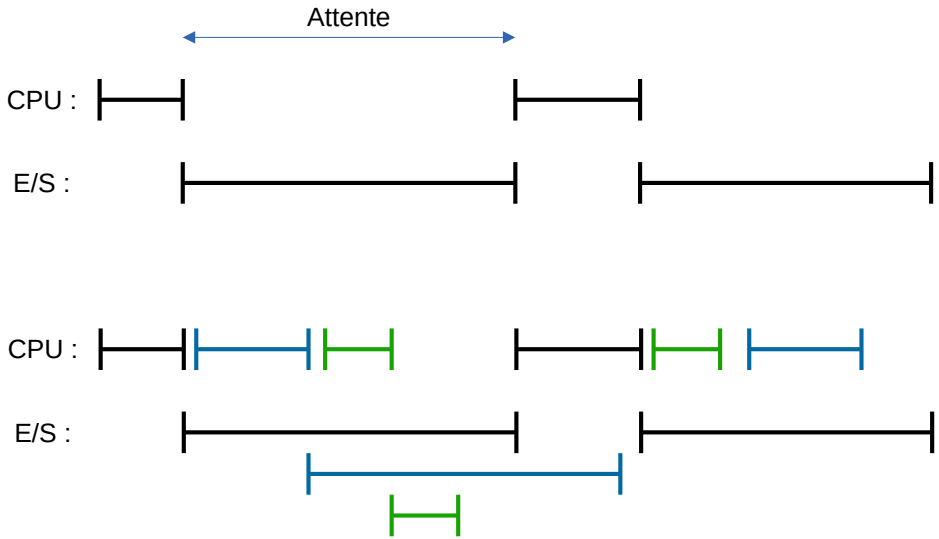


ML	MC	MI
Préparation du lot 1		
Préparation du lot 2	Exécution du lot 1	
Préparation du lot 3	Exécution du lot 2	Impression du lot 1



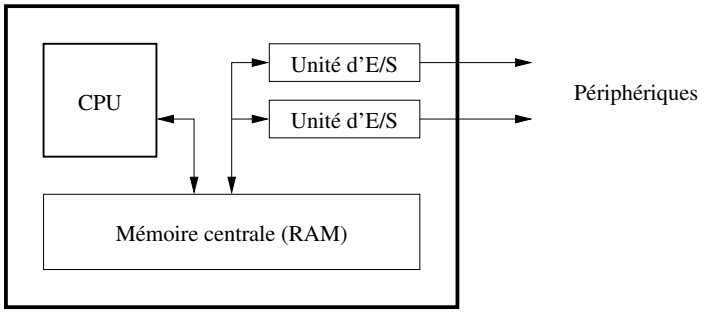
▶ Cycles de CPU et d'entrée/sortie ◀

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



▶ Milieu des années 60 : Multiprogrammation ◀

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

▶ Années 60/70 : Apparition des terminaux ◀

- 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% :

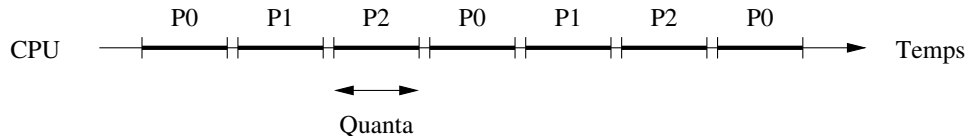
donc, sur 100 utilisateurs, **10 sont actifs**.

Hypothèse 2 : Les utilisateurs actifs réclament des actions simples :

prise en compte du travail interactif

▶ Années 60/70 : Le temps partagé ◀

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



Si

Quanta = 50 millisecondes **et** une requête $\leq$ 1 quanta **et** 10 demandes

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**.

▶ Années 1980 : Les systèmes répartis ◀

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

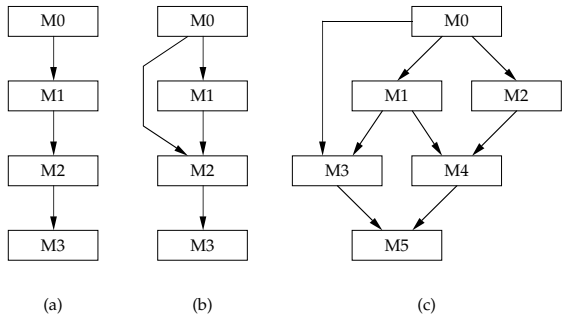
Exemple : Le **Cluster**. La somme de plusieurs machines est vue comme une seule machine (simplicité, évolutivité, robustesse).

▶▶ Architecture interne d'un système ◀◀

Une **machine** c'est

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

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

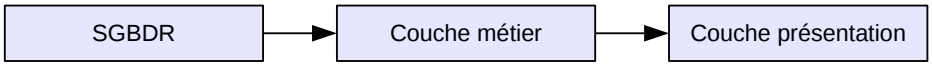


Avantages : Sécurité et modularité.

► Notion de couches dans les architectures logicielles ◄

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

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

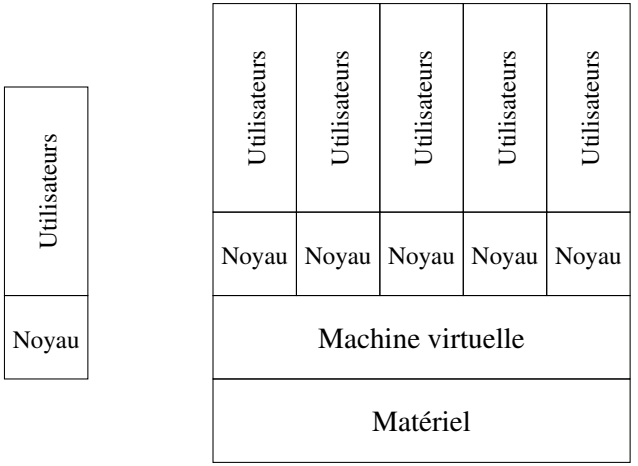


- une couche possède une spécification (interface d'entrée),
- une couche masque les couches inférieures.

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

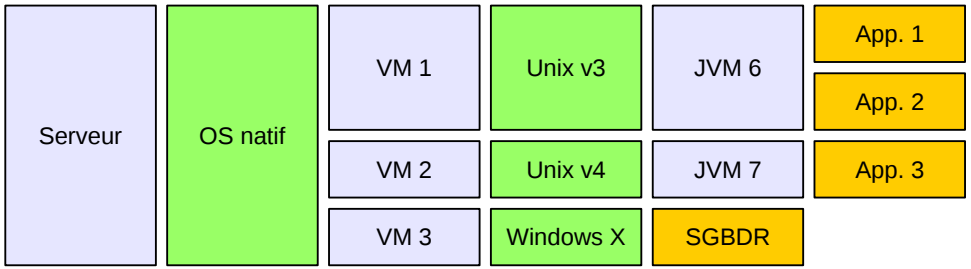
► Un premier exemple ◄

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



► Architectures virtualisées applicatives ◄

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

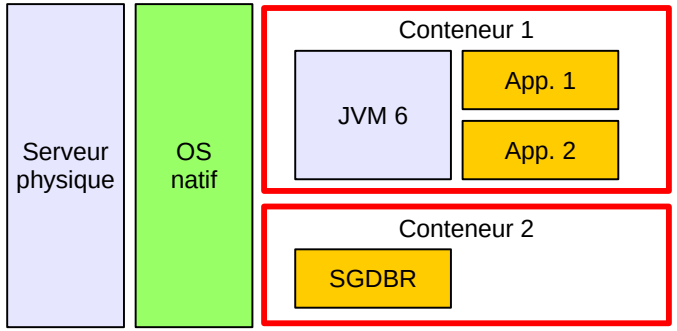


Avantages :

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

► Architectures à base de conteneurs ◄

Pour alléger l'utilisation des ressources, une machine physique peut être découpée en **conteneurs** avec une répartition **maîtrisée** des ressources.



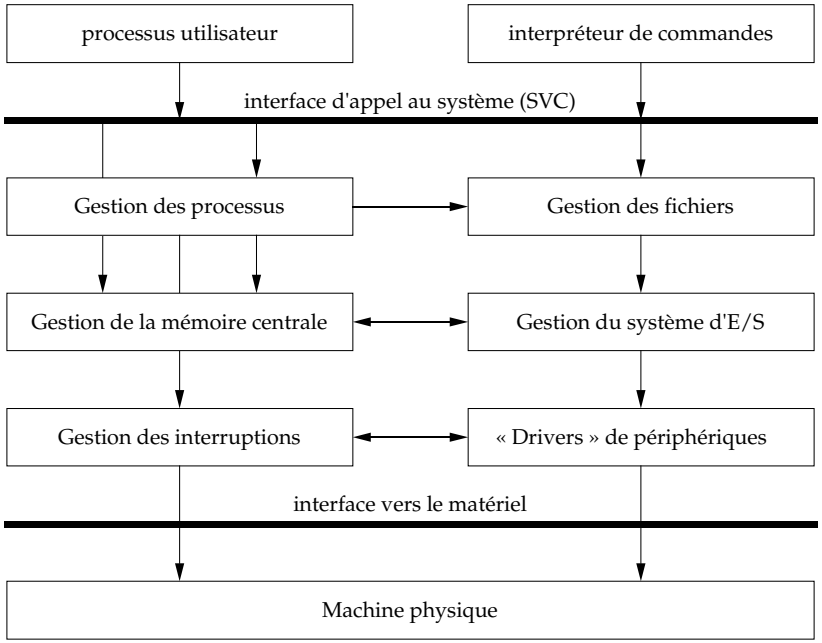
Avantages :

- Un seul système partagé
- mise en oeuvre plus simple

Inconvénients :

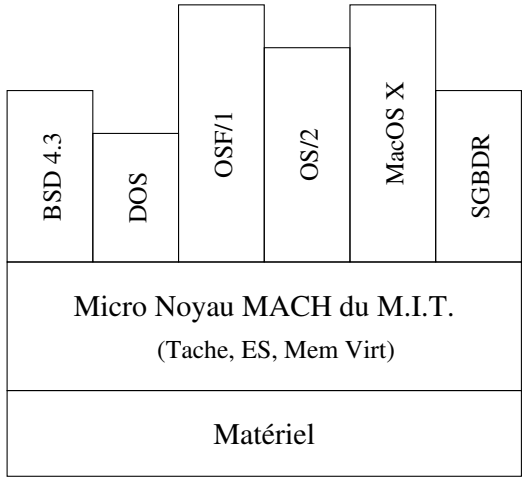
- environnements homogènes
- pas disponible sur tous les OS

► Composantes d'un système d'exploitation ◄



► Un deuxième exemple ◄

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



Avantages : **Portabilité**, **Qualité**.

Présentation des processus

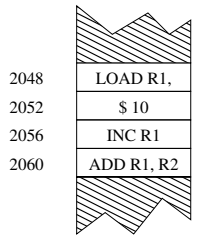
- Notion de programme
- Notion de machine
- Exécution d'un programme
- Processus

▶ Notion de programme ◀

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

```
void main() {
  ...
}
```

⇒



⇒

```
01001001011101101
10011111001010100
11101010001100101
```

$$\text{Instructions} = \begin{cases} \text{privilégiées} = \begin{cases} \text{Lancement des E/S} \\ \text{Contrôle des interruptions} \\ \text{Contrôle du matériel} \end{cases} \\ \text{ordinaires} = \begin{cases} \text{mouvement de données} \\ \text{structure de contrôle} \\ \text{calcul} \\ \text{appel du système d'exploitation} \end{cases} \end{cases}$$

▶ Description du processeur ◀

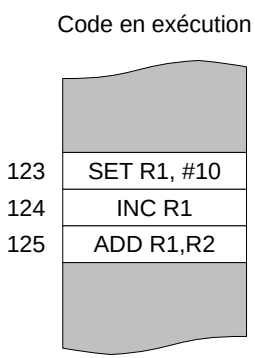
$$\text{Un processeur} = \begin{cases} \text{Mot d'état processeur (MEP)} = \begin{cases} \text{Registres spécialisés} \\ \text{Registres généraux} = \begin{cases} \text{Entier} \\ \text{Flottant} \\ \text{Adresse} \end{cases} \end{cases} \\ \text{Unité de calcul (UAL)} \end{cases}$$

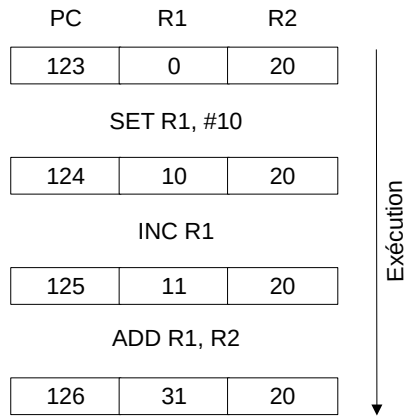
$$\text{Registres spécialisés} = \begin{cases} \text{compteur ordinal (PC : program counter)} \\ \text{mode d'exécution (MODE : utilisateur ou système)} \\ \text{pointeur de pile (SP : stack pointer)} \\ \text{masque d'interruptions} \\ \text{codes de conditions} \\ \ldots \end{cases}$$

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

▶ Exécution d'un programme ◀

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



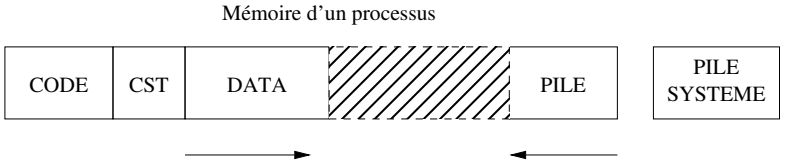


Notion de points **observables**.

▶ Notion de processus ◀

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

▶ Rendez-vous asynchrone ◀

Problème :

- **exécuter efficacement** un programme et, **en même temps**,
- **traiter rapidement** un **événement imprévisible (asynchrone)**.

Exemple :

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

▶ Attente active ◀

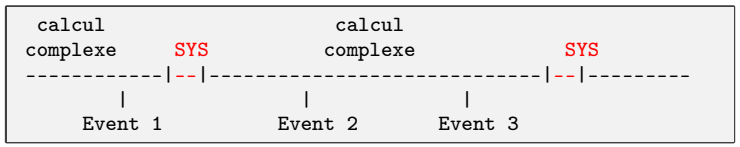
Solution : **l'attente active**. Soit `F` un indicateur qui signale l'événement

```
// Code utilisateur consommateur de CPU

...
pour i variant de 1 à 100000 faire
    faire_un_calcul_complexe();
    system_tasks();
fin pour
...
```

```
// Code système

fonction system_tasks() {
    si (F = 1) alors
        F := 0
        < traiter l'événement >
    fin si
}
```

Remarques :

- le test **explicite** est pénible et peu sécurisé
- perte de temps CPU
- risque de perte d'événement



▶ Interruption d'un processus : Principe ◀

- L'**interruption** est une opération
  - ▷ **indivisible** réalisée par la **CPU**,
  - ▷ **liée à une cause** identifiée par un numéro,
  - ▷ qui sauvegarde et change le `PC` et le `MODE`.

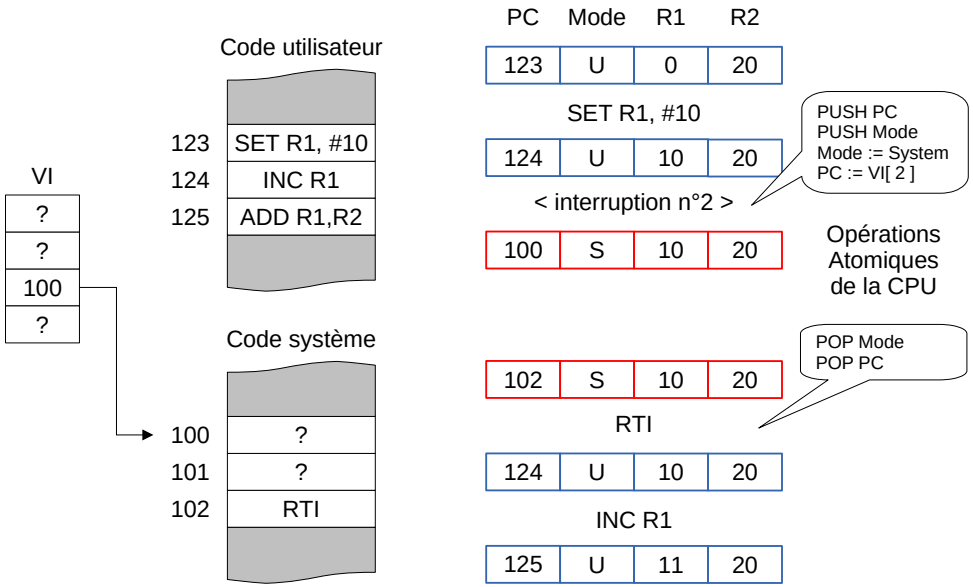
$$\text{interruption de cause } k = \begin{cases} \texttt{PUSH PC} & \texttt{// sauvegarde de PC} \\ \texttt{PUSH MODE} & \texttt{// sauvegarde de MODE} \\ \texttt{MODE := système} & \texttt{// passage en mode système} \\ \texttt{PC := vi[k]} & \texttt{// branchement sur vi[k]} \end{cases}$$

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

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

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

▶ Interruption d'un processus : Exemple ◀



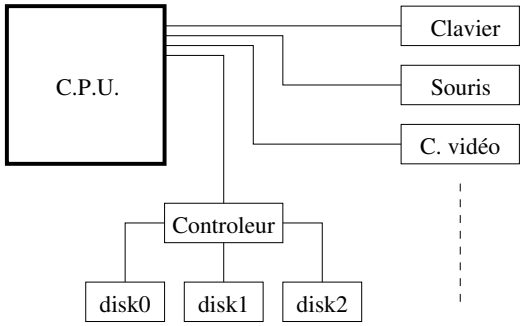
▶ Structure générale d'un traitant ◀

Le code exécuté après une interruption est appelé le **traitant** de cette interruption. Il a la structure suivante :

1. **sauvegarder** le contexte (registres)
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`)

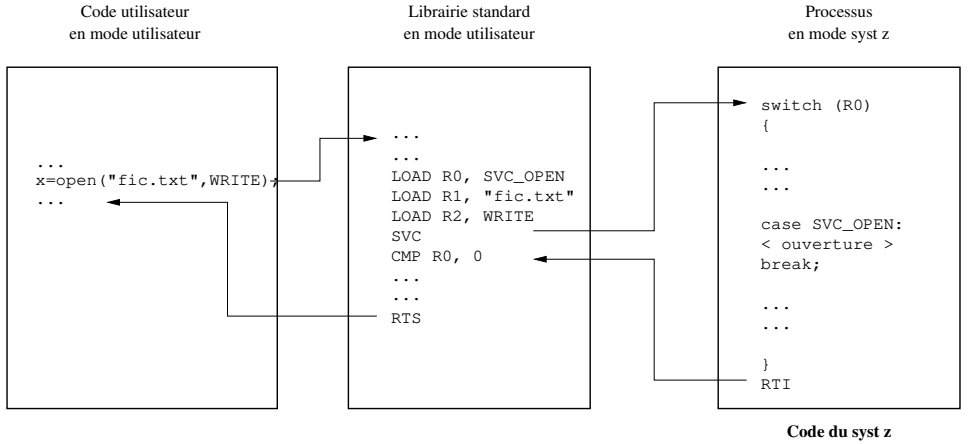
▶ Interruption matérielle : principe ◀

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



▶ Les appels système : principe ◀

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



▶ Les appels système : 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 système 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
```


▶ 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

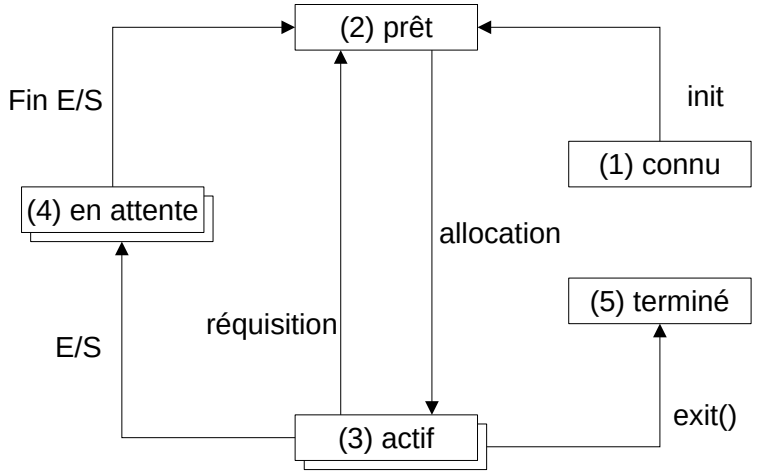
Utilisations :

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

Gestion des processus

▶ État d'un processus ◀

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



▶ Représentation d'un processus ◀

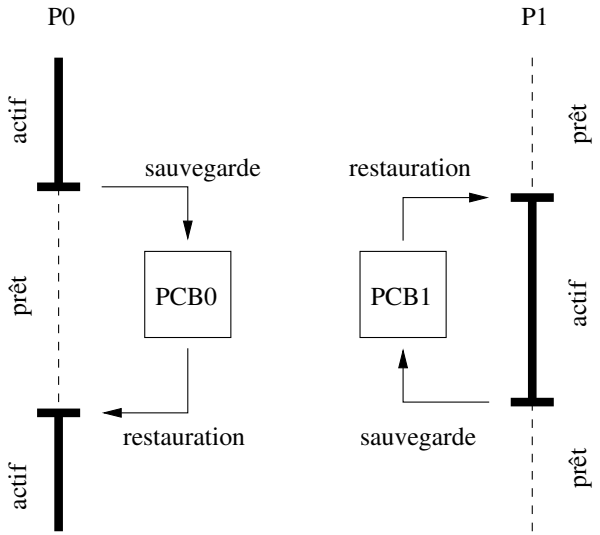
Pour chaque processus le système maintient :

- 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**).

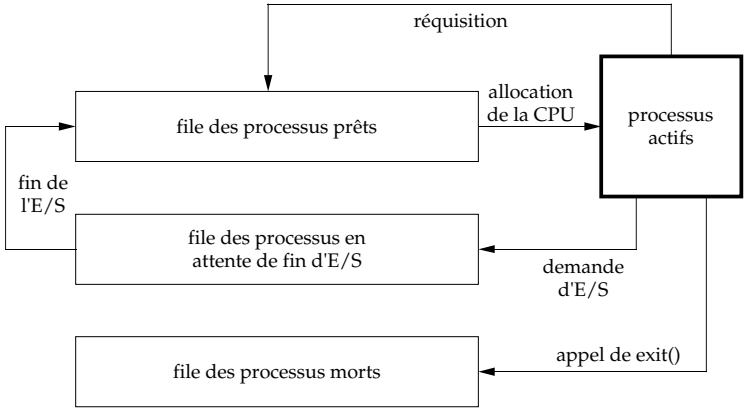
▶ Rôle du PCB ◀

Le rôle du PCB dans la commutation entre deux processus :



▶ Représentation des processus ◀

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



▶ Gestion des processus ◀

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 arret.

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)

▶▶ Allocation du processeur ◀◀

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,

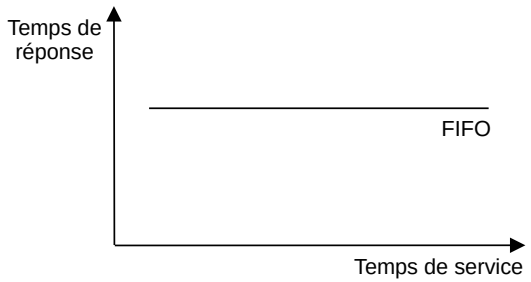
▶ Algorithme d'allocation FIFO ◀

FIFO (**First In First Out**) :

- premier arrivé, premier servi
- absence de réquisition
- croissance rapide du temps de réponse.

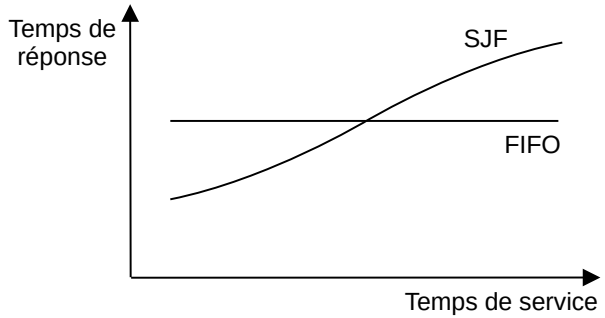
$$\text{temps de réponse} = (\text{temps d'attente} + \text{temps d'exécution})$$

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



▶ Algorithme d'allocation SJF / SRT ◀

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.

▶ Algorithme d'allocation HRRN ◀

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

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

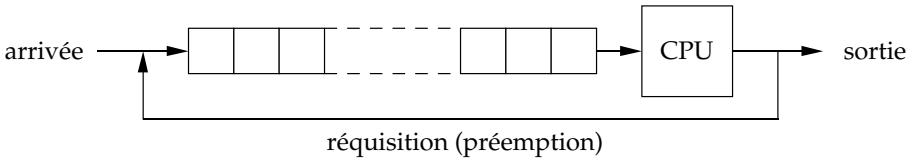
avec

- $w(t) = t - t_a$: 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.

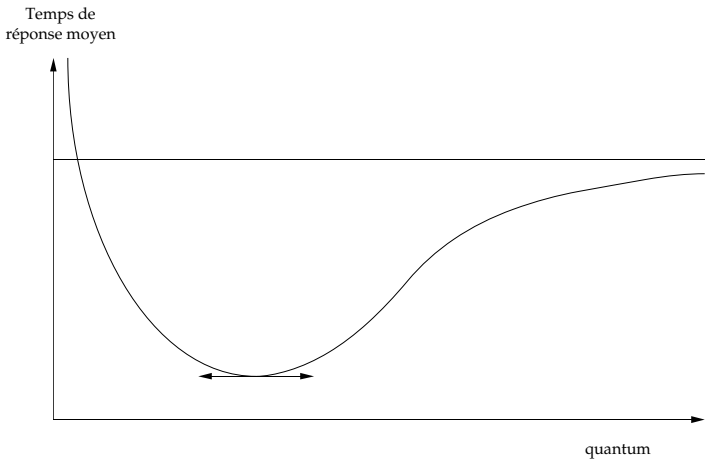
▶ Algorithmes d'allocation du tourniquet ◀

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

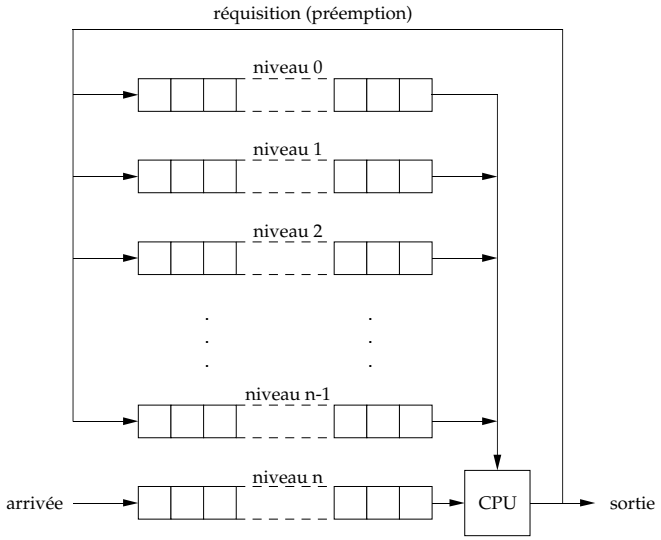


Comment fixer la durée du quanta ?

Temps de réponse = Nb de passages × taille du quanta × taille de la file FIFO



▶ Tourniquet Multi-niveaux ◀



▶ Tourniquet Multi-niveaux avec priorités ◀

Il existe des classes de processus :

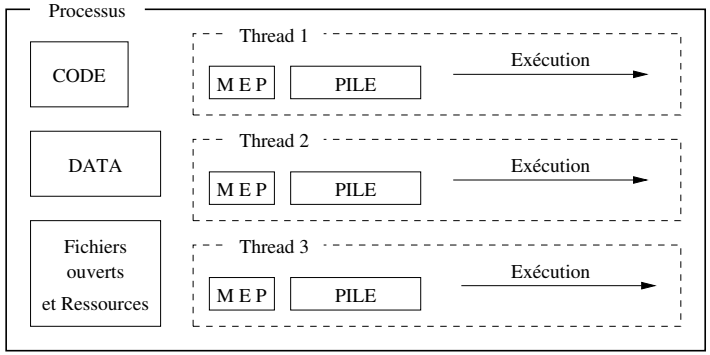
- processus systèmes,
- processus temps réel,
- processus interactifs,
- processus d'édition interactive,
- processus « batch ».

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

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

▶▶ Les « threads » (processus de poids léger) ◀◀

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**.

▶ Utilité des threads ◀

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

```
void travail() {
    int t;

    travail_1();
    if ((t = thread()) == 0) {
        travail_3();
        thread_exit();
    }
    travail_2();
    thread_join(t);
    travail_4();
}
```

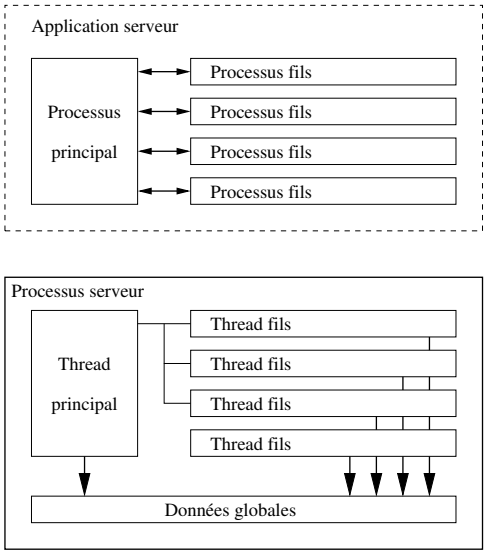
Trace de l'exécution

```
CPU0 : |<----- T1 ----->|<--- T2 --->|       |<----- T4 ----->|
CPU1 :                  |<------- T3 ------->|
```

Avantages :

- exploitation des machines multi-processeurs,
- récupération des temps d'E/S (même si un seul processeur),
- coopération entre threads.

►► Utilité des « threads » pour les serveurs ◄◄

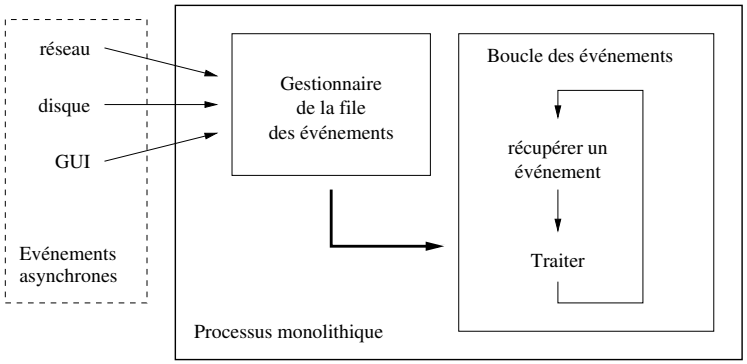


La **commutation** et la **communication** entre threads est une opération plus simple

► Les processus monolithiques ◄

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

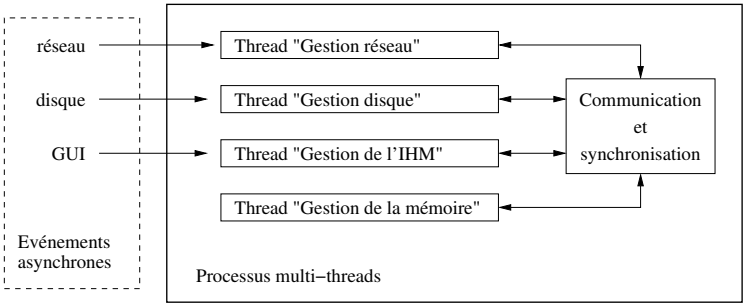
Solution **d'attente active** : E/S asynchrones (**non-bloquantes**) et structure avec **boucle d'événements**.



► Les processus multi-threads ◄

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.

▶▶ Présentation générale ◀◀

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

▶ Accès à des ressources partagées ◀

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 ;
```

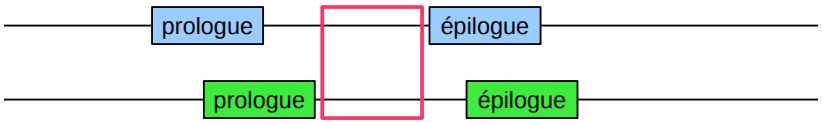

▶ Le problème de l'exclusion mutuelle ◀

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

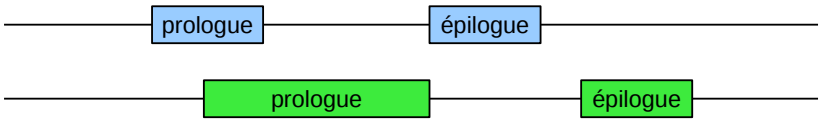
Forme des programmes :

⟨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 :



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



Contraintes :

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

▶ Nouvelle formulation ◀

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⟩
```


▶ Une solution d'attente active ◀

On se donne `libre` une variable **partagée** de type booléenne.

```
⟨initialisation⟩  (1)  libre := vrai ;          // la SC est libre

⟨prologue⟩        (2)  répéter                  // Attendre la liberté
                  (3)      ne rien faire
                  (4)  jusqu'à (libre = vrai)
                  (5)  libre := faux            // La SC est prise

                  (6)  ⟨section critique⟩

⟨épilogue⟩        (7)  libre := vrai            // Libérer
```

Critique 1 : Consommation de temps processeur.

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

2 3 4 ⟨interruption⟩ 2 3 4 5 6 ... ⟨interruption⟩ 5 6 ...

▶ Solution matérielle ◀

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 :

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


▶ Utilisation de Test and Set ◀

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

```
⟨initialisation⟩  (1)  mutex := 1 ;

⟨prologue⟩        (2)  Soit p une variable privée
                  (3)  répéter
                  (4)      TAS(p, mutex) ;
                  (5)  jusqu'à (p = 1)

                  (6)  ⟨section critique⟩

⟨épilogue⟩        (7)  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**.

▶ Les verrous ◀

Objectifs :

- ne plus perdre de temps CPU,
- simplifier 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.

▶ Les verrous (2) ◀

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

```
procédure prendre(Verrou v) en exclusion mutuelle
    | 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
```

```
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
```


▶ L'exclusion mutuelle avec les verrous ◀

Le code de l'exclusion mutuelle s'écrit

```
⟨initialisation⟩    (1)   mutex := nouveau Verrou ;
```

```
⟨prologue⟩          (2)   prendre(mutex) ;
                    (3)       ⟨section critique⟩
⟨épilogue⟩          (4)   libérer(mutex) ;
```


▶ Difficultés des verrous ◀

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⟩

Solution : ordonner les prises de verrous.

▶ Les sémaphores ◀

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 := {} ;
```


▶ Agir sur les sémaphores ◀

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*

▶ Utilisation des sémaphores ◀

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

▶ Le problème du producteur et du consommateur ◀

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 ...
```

▶ Codage du producteur/consommateur ◀

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

NPlein : sémaphore = $(0, \{\})$

producteur :

répéter
  ⟨produire un message⟩
  ⟨le déposer dans le tampon⟩
  V(NPlein) ;
jusqu'à ...

Consommateur :

répéter
  P(NPlein) ;
  ⟨consommer⟩
jusqu'à ...

▶ Codage du producteur/consommateur (2) ◀

Définition des sémaphores :

NPlein : sémaphore = $(0, \{\})$
NVide : sémaphore = $(n, \{\})$

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

répéter
  P(NVide) ;
  ⟨produire un message⟩
  ⟨le déposer dans le tampon⟩
  V(NPlein) ;
jusqu'à ...

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

répéter
  P(NPlein) ;
  ⟨consommer⟩
  V(NVide) ;
jusqu'à ...

▶ Les sémaphores à messages ◀

```
Sémaphore-à-messages : structure
    disponible : sémaphore = nouveau sémaphore(0,{}) ;
    messages : file FIFO de messages = {} ;
```

```
fonction receive( var sm : Sémaphore-à-messages ) : données
    P(sm.disponible) ;
    msg := sm.messages.sortir() ;
    renvoyer msg ;

procédure send( var sm : Sémaphore-à-messages ; msg : données )
    sm.messages.ajouter(msg)
    V(sm.disponible)
```

Remarque : L'une des deux files est toujours vide (voir les deux).

Exemple :

Les tubes : un exemple de Sémaphore à messages

```
$ find /usr | fgrep /doc | fgrep java | wc -l
```


▶ Les régions critiques (ou moniteurs) ◀

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**.

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
```

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


▶ 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];
    }
}
```