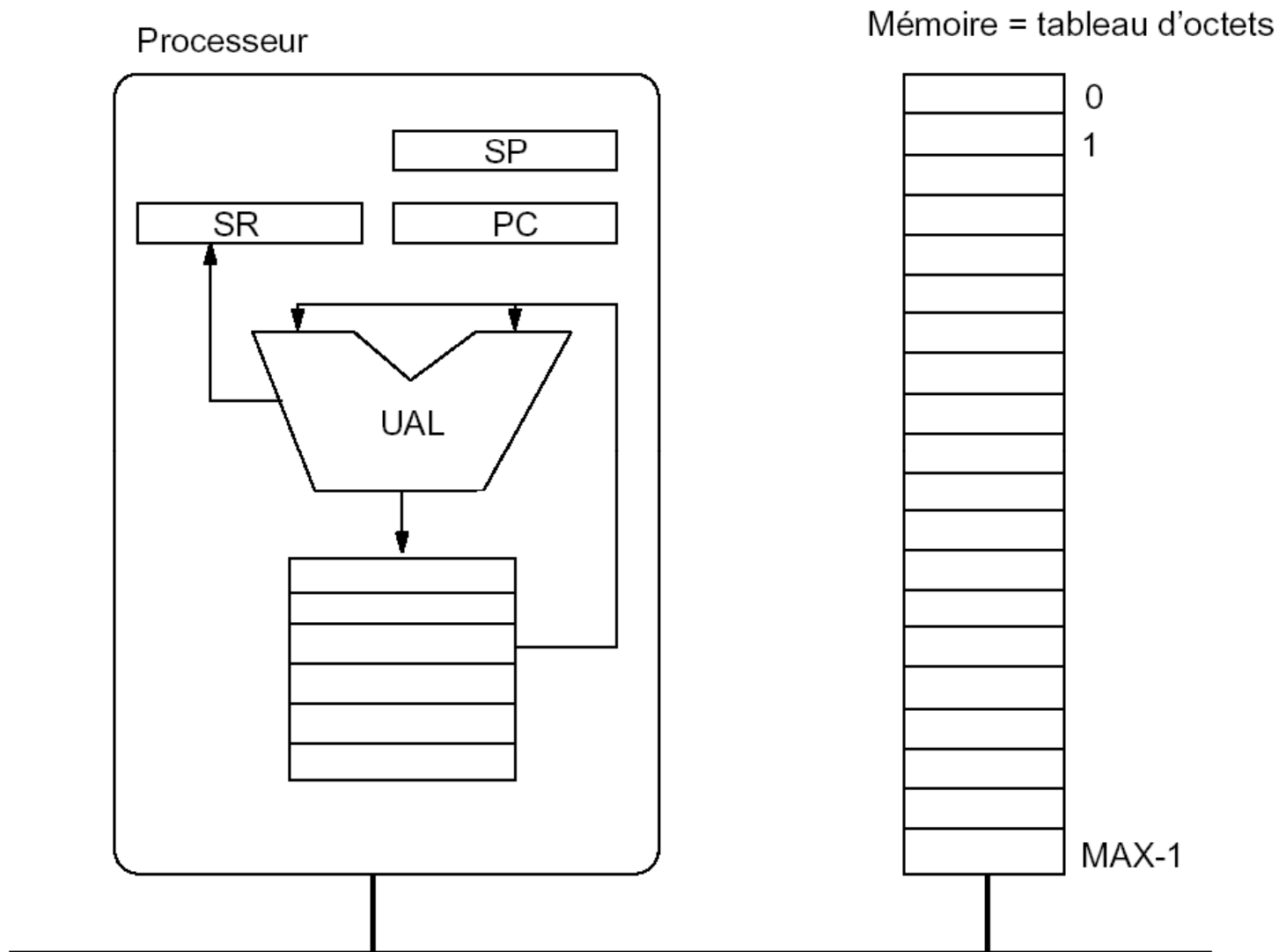


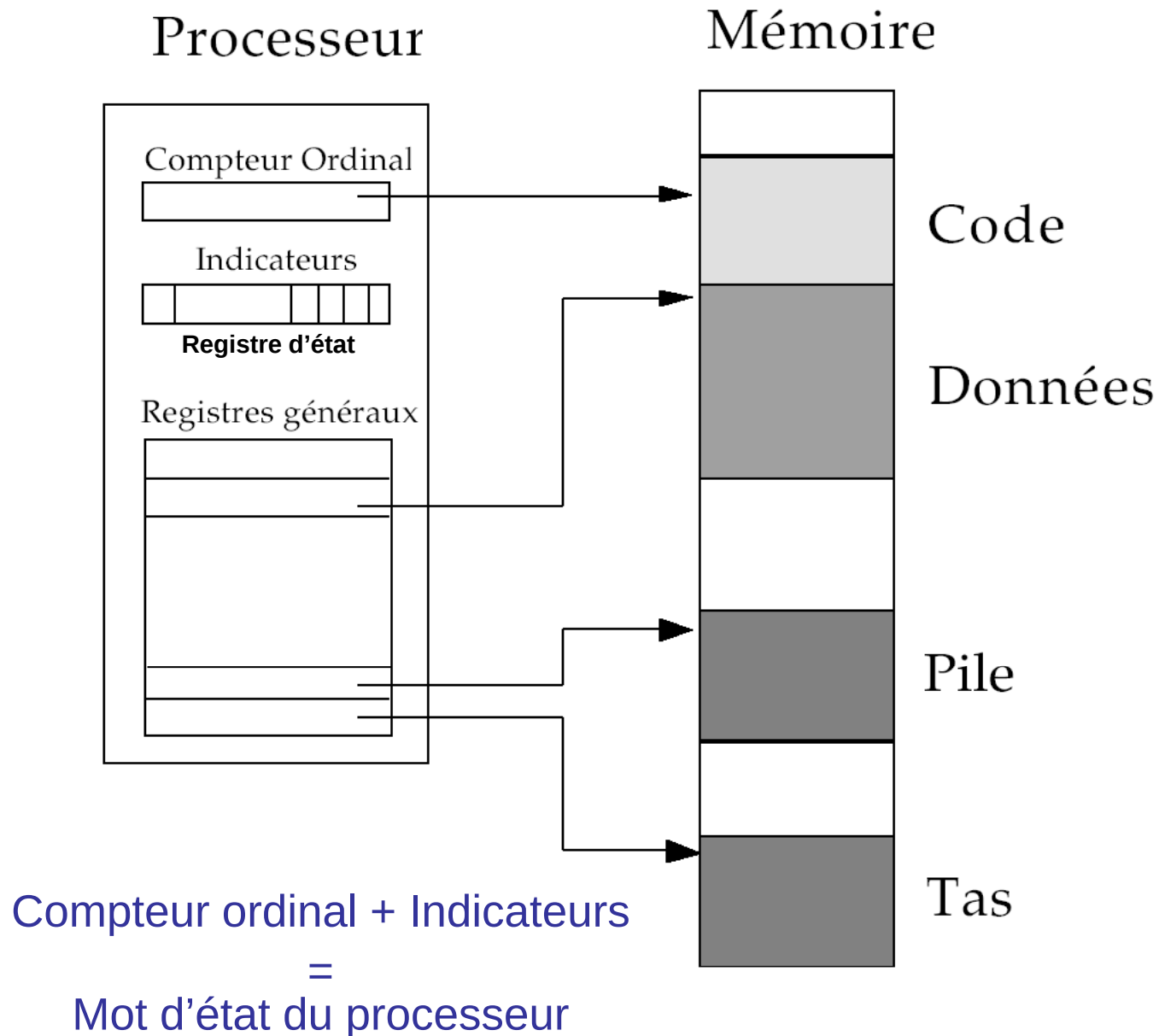
Mécanismes d'exécution

Du séquentiel au parallèle

Mémoire et Processeur

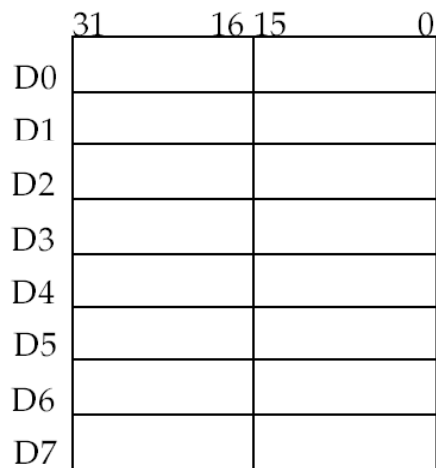


Modèle d'exécution

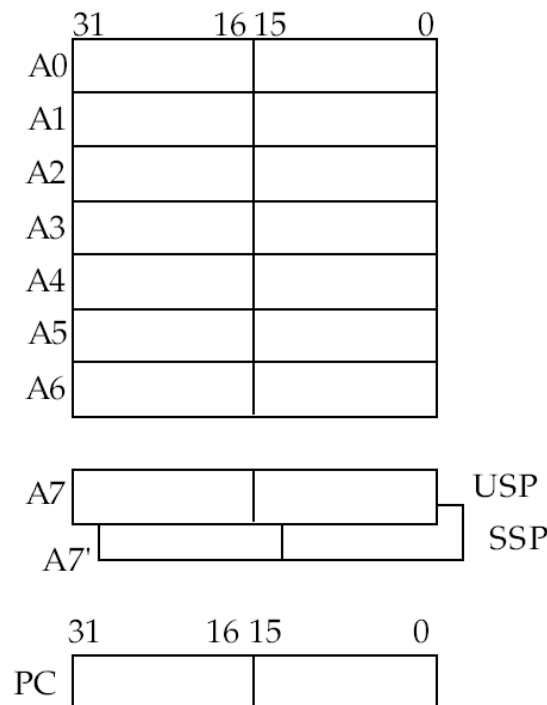


Exemple : le Motorola 68000

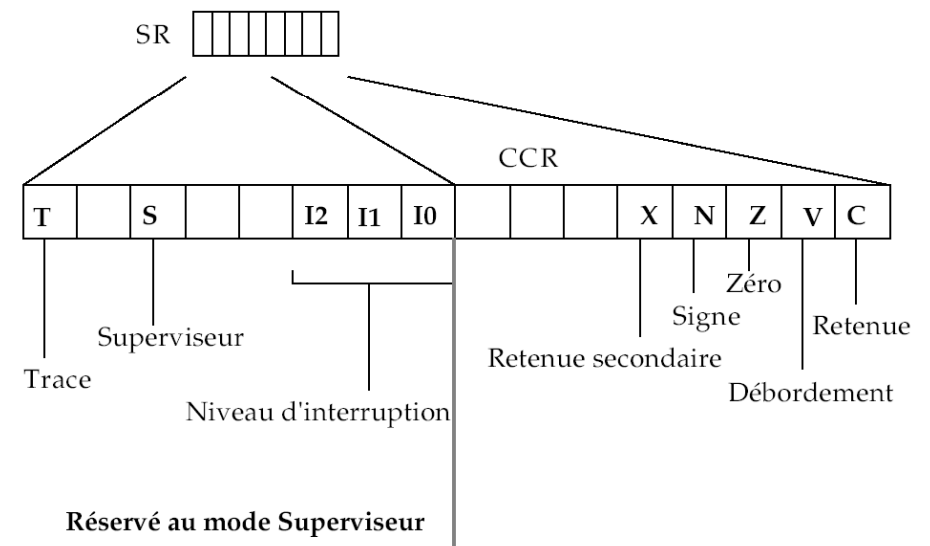
Registres
de donnée



Registres
d'adresse

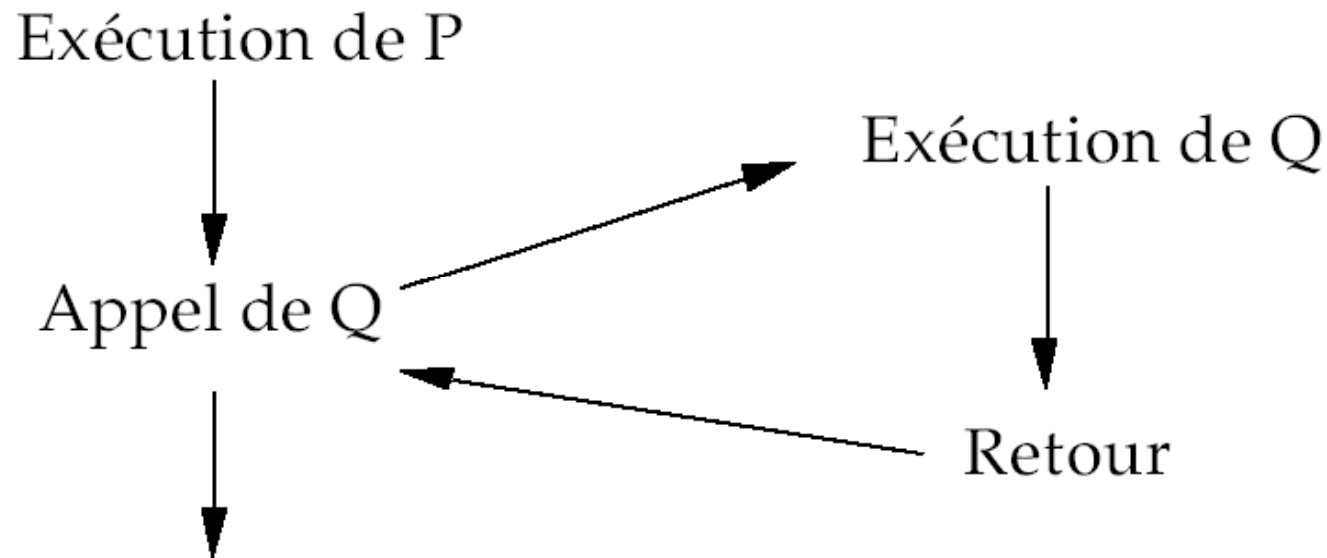


Registre d'état



L'ensemble des valeurs des registres pour un processus donné est appelé le **contexte** du processus.

Appel et retour de procédure



Séquence d'appel

Préparation des paramètres transmis à Q

Sauvegarde du contexte de P

Remplacement du contexte de P par le contexte de Q

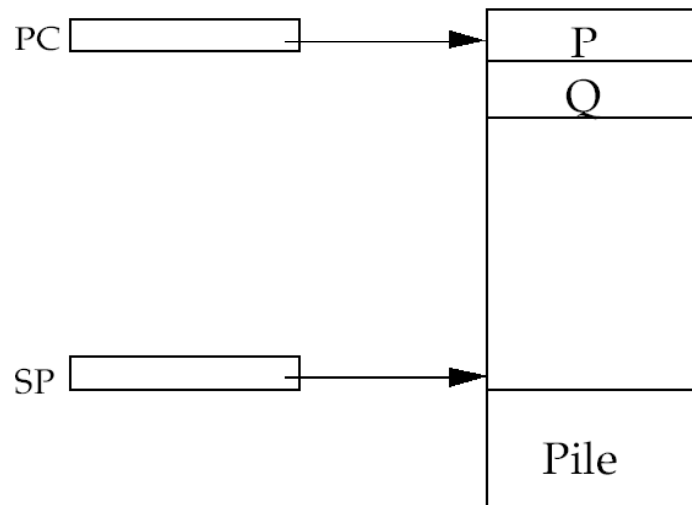
Retour

Préparation des résultats transmis par Q

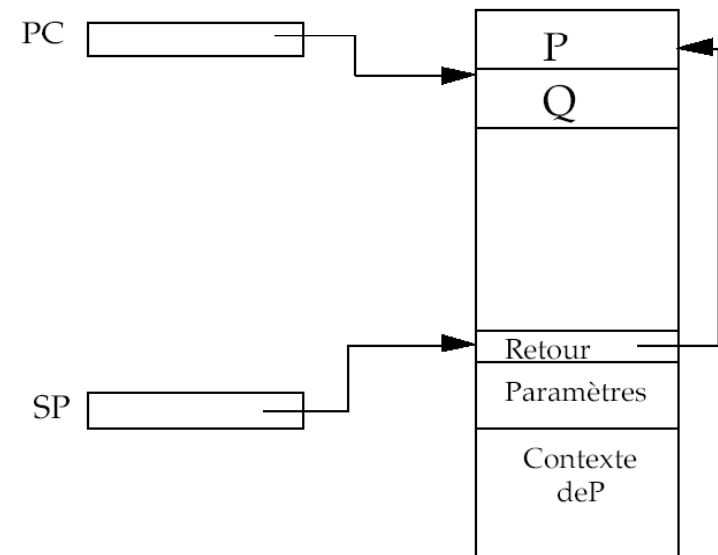
Restauration du contexte de P avant l'appel

Réalisation avec une pile

État avant l'appel

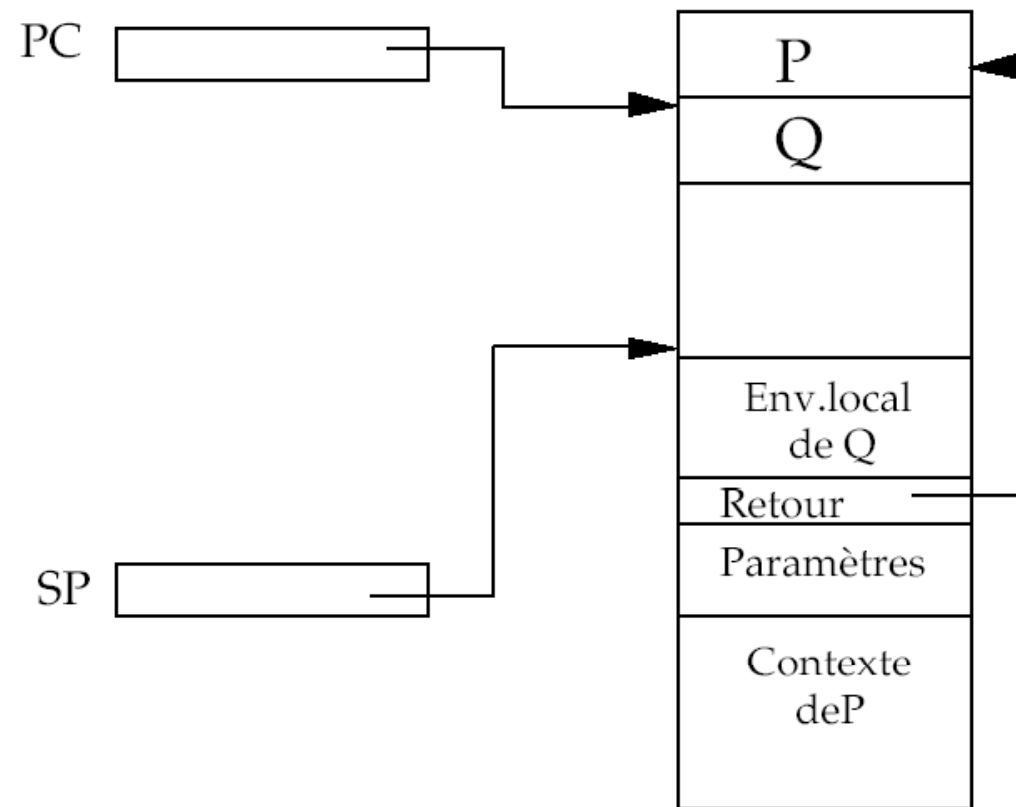


P : Préparation des paramètres et commutation

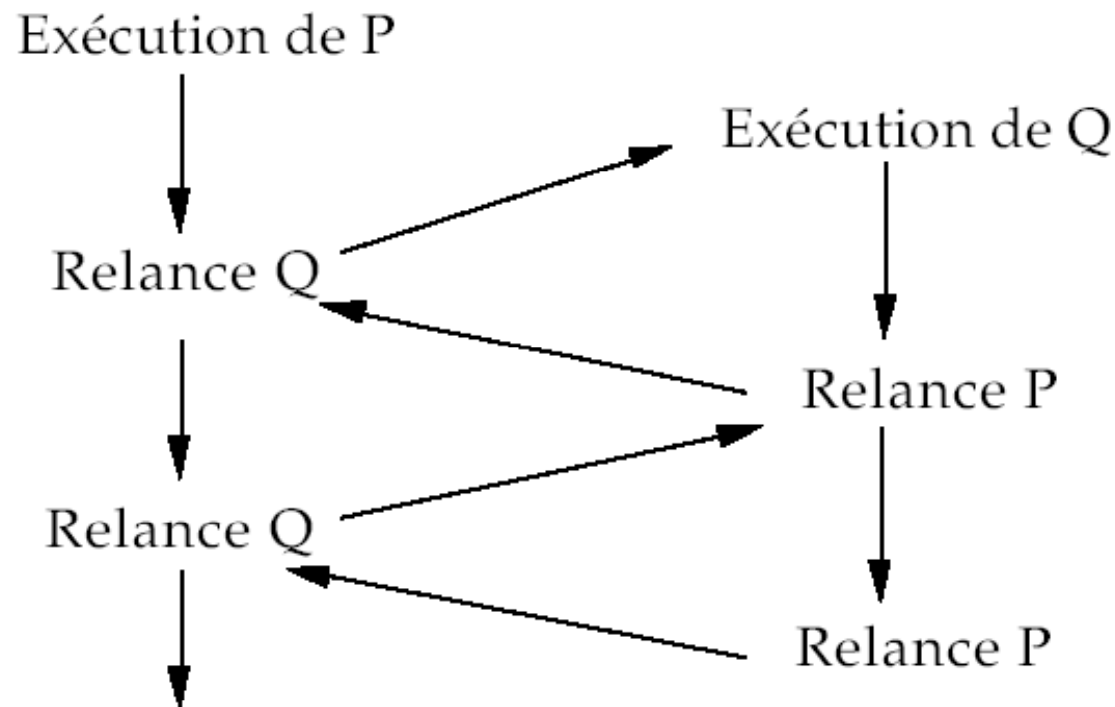


Réalisation avec une pile

Q : préparation de l'environnement local



Fonctionnement en co-routines



Relance (Resume) = Séquence d'appel

Préparation des paramètres transmis

Sauvegarde du contexte courant

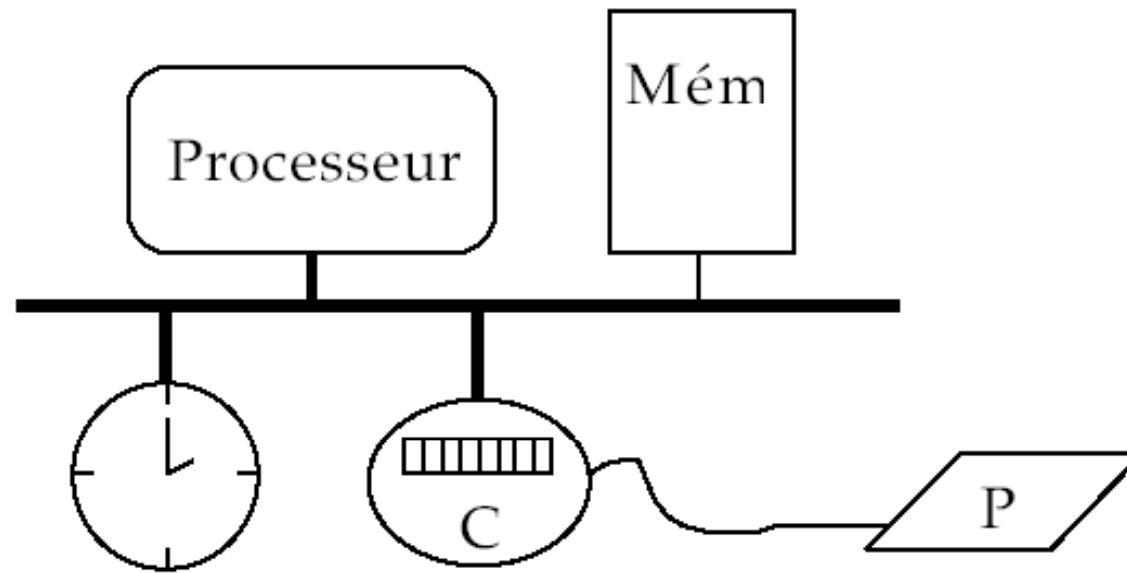
Remplacement de l'ancien contexte par le nouveau

Réalisations

Avec une pile ?

Avec contexte global

Activités synchrones



Entrées/Sorties par canal, Horloge,
Intervention externe, Traitement d'erreur

—▶ Nécessité de pouvoir interrompre le processeur

Activités synchrones

Mécanismes d'exception

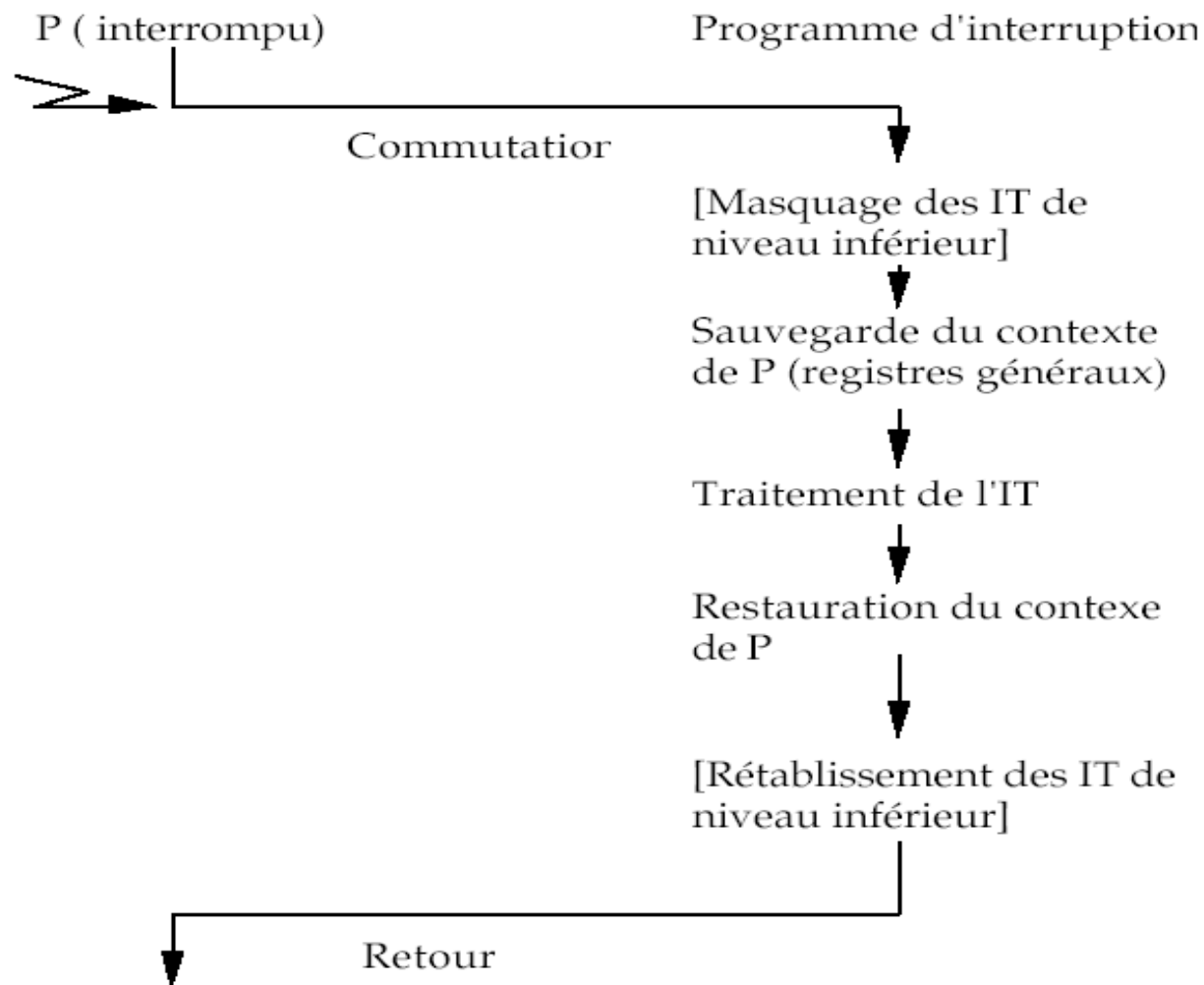
Mécanisme	Cause	Utilisation
Interruption	Extérieure à l'instruction en cours	Réaction à un événement externe (E/S, horloge, sécurité)
Déroutement	Déclenché par l'instruction en cours	Traitement d'erreurs (débordement, division par 0, instruction illégale)
Appel au superviseur	Exception voulue, programmée	Appel d'une fonction du système d'exploitation (passage en mode privilégié)

Commutation de contexte

- Principe
 - Sauvegarde du mot d'état du programme et des registres
 - Chargement d'un mot d'état et de registres à partir d'un emplacement spécifié
- 2 méthodes pour la sauvegarde
 - Emplacements fixes
 - Pile

Interruptions

- En général plusieurs niveaux (priorités)
- Masquables (armement, désarmement)
- Schéma d'un programme d'interruption :



Déroutements et appels au superviseur

Déroutements

Données incorrectes (débordement, division par 0)

Violation de protection (de la mémoire, du mode privilégié)

Instruction non-exécutable (code inconnu, erreur d'adresse,...)

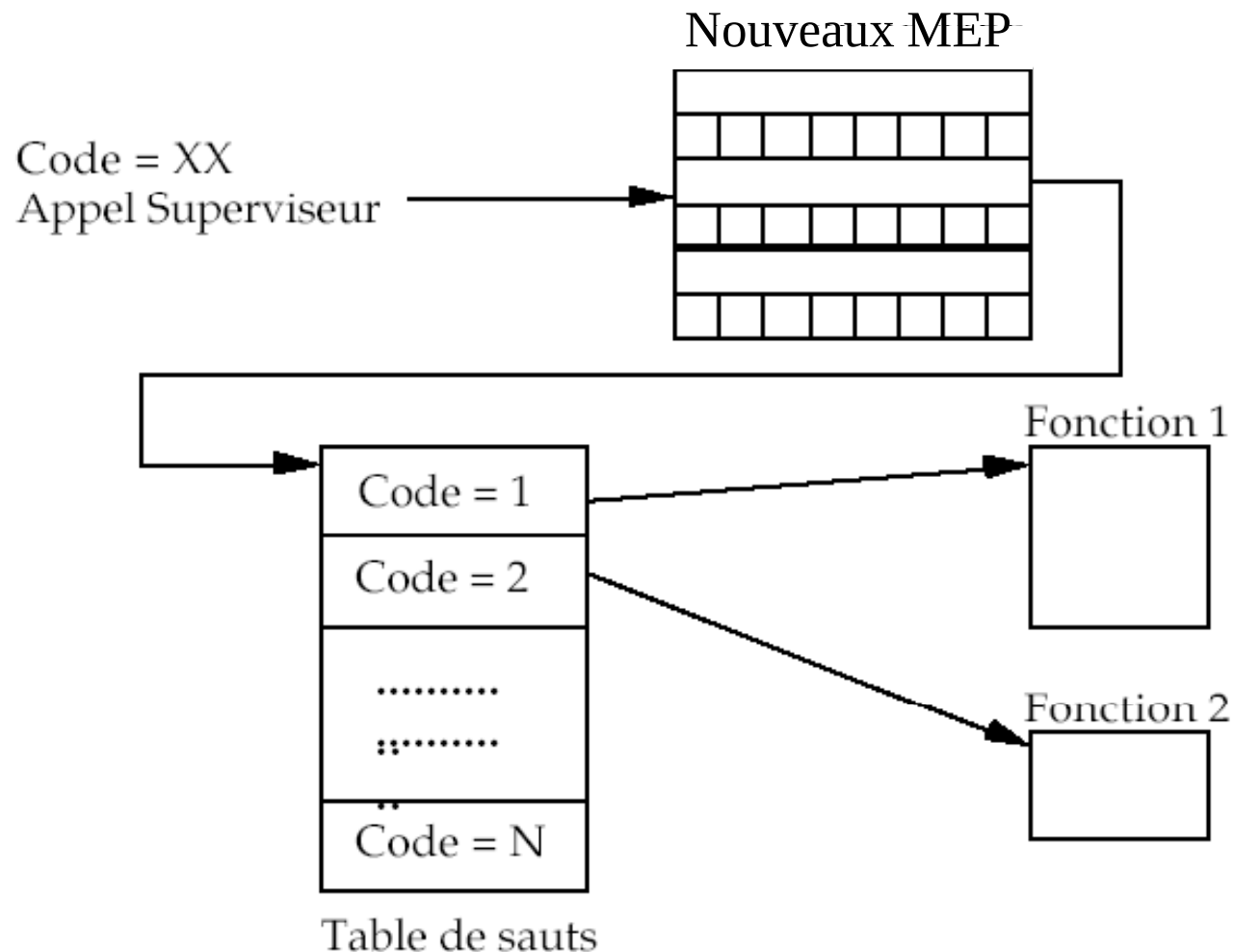
Appels au superviseur

Comme un appel de procédure mais avec des droits étendus (mode privilégié, IT masquées, droits d'accès)

Appel aux fonctions du système

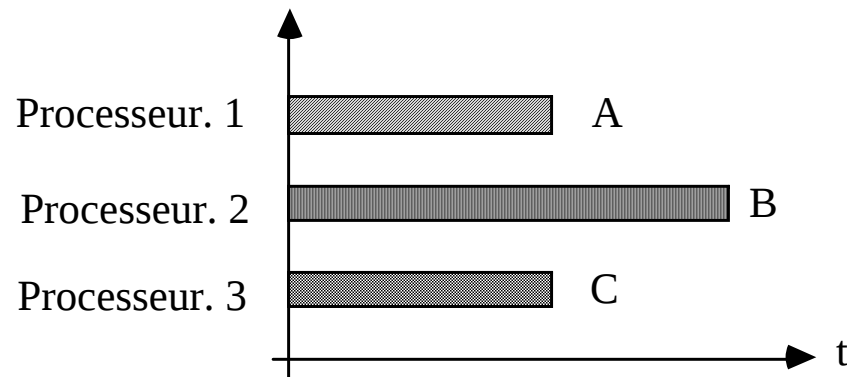
Déroutements et appels au superviseur

Appel aux fonctions du système

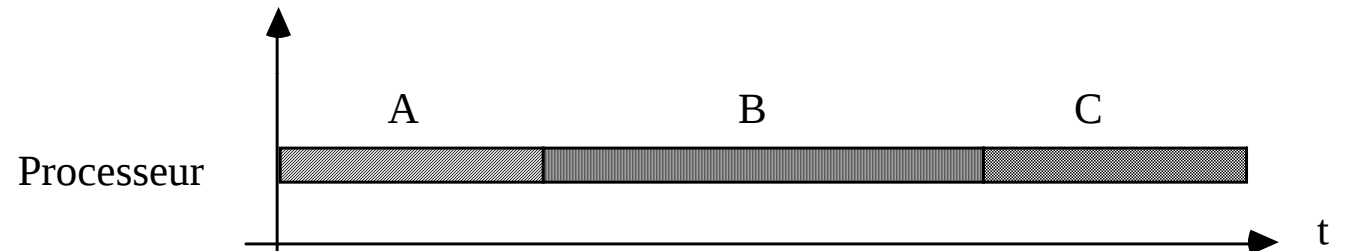


Le Multitâches

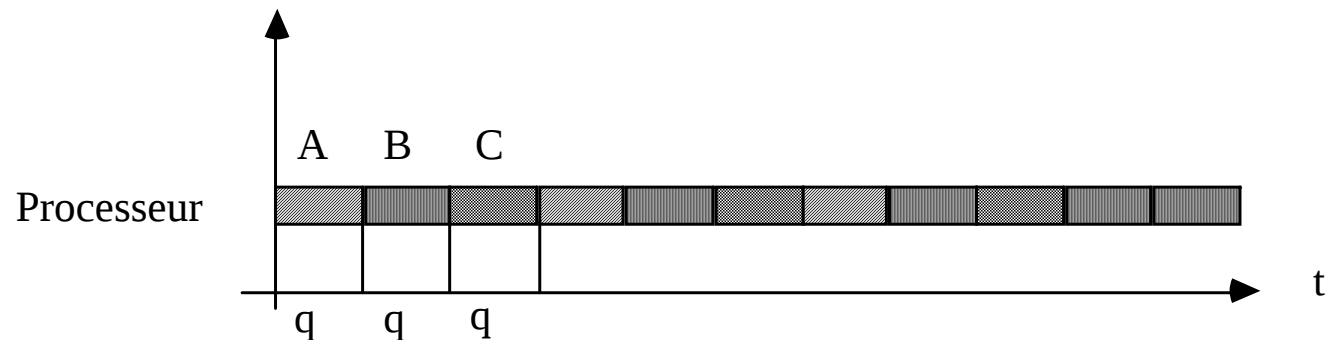
- Système multiprocesseurs



- Système Monoprocasseur, monotâche



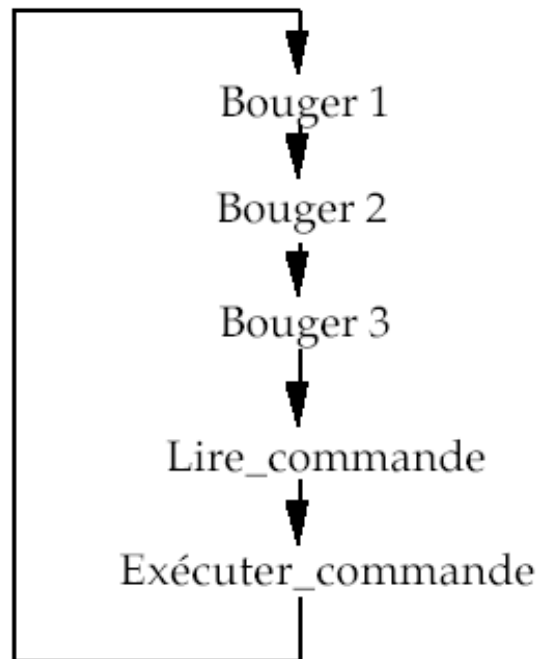
- Système Monoprocasseur, multi-tâches



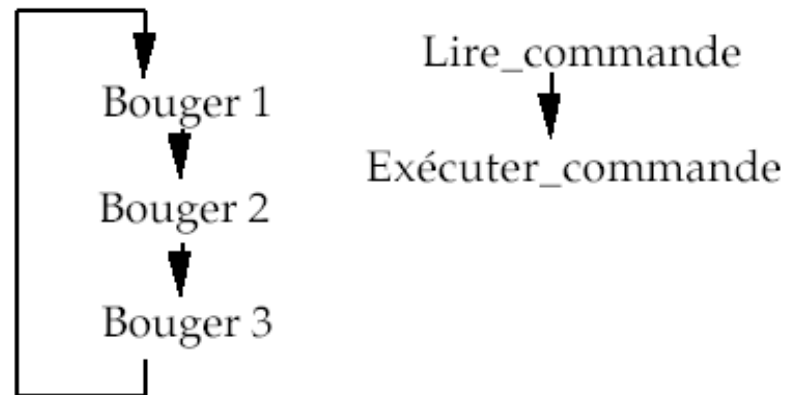
Exemples : jeux vidéos

Monotâche avec interruptions

Monotâche

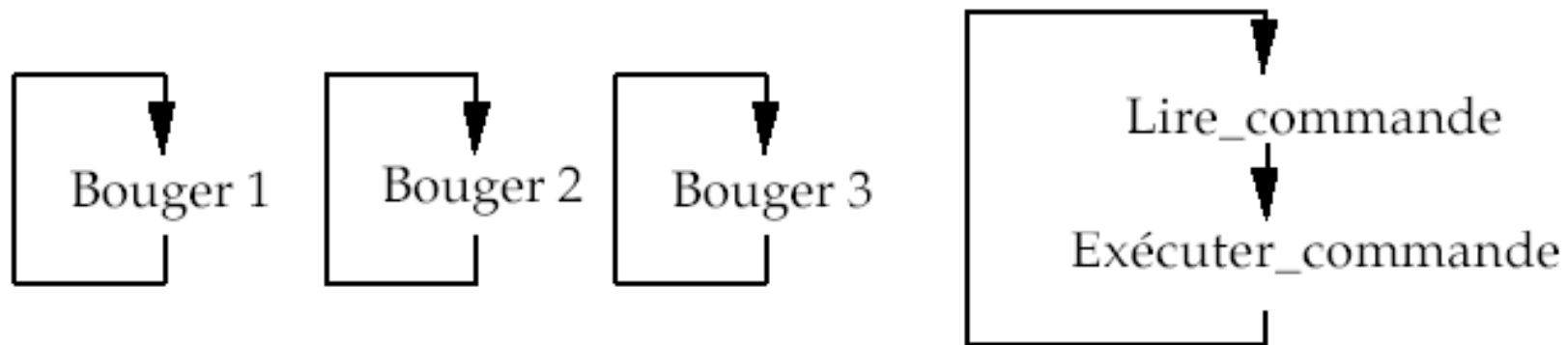


tâche principale/interruptions



Exemples : jeux vidéos

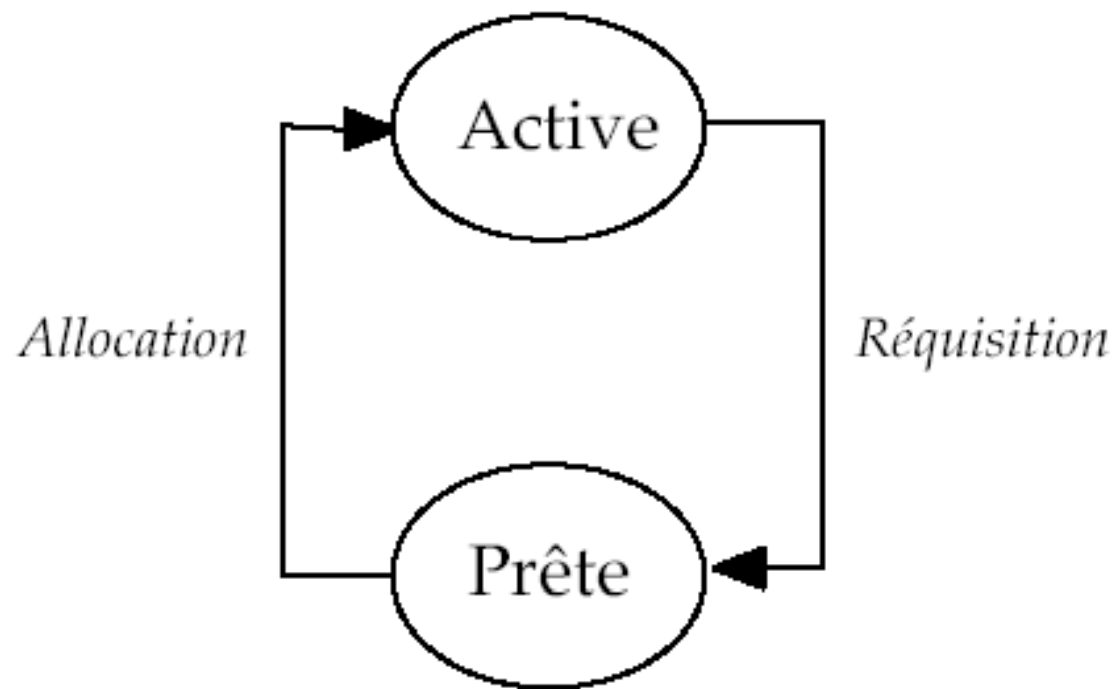
Multi-tâches



Souplesse
Partage du code

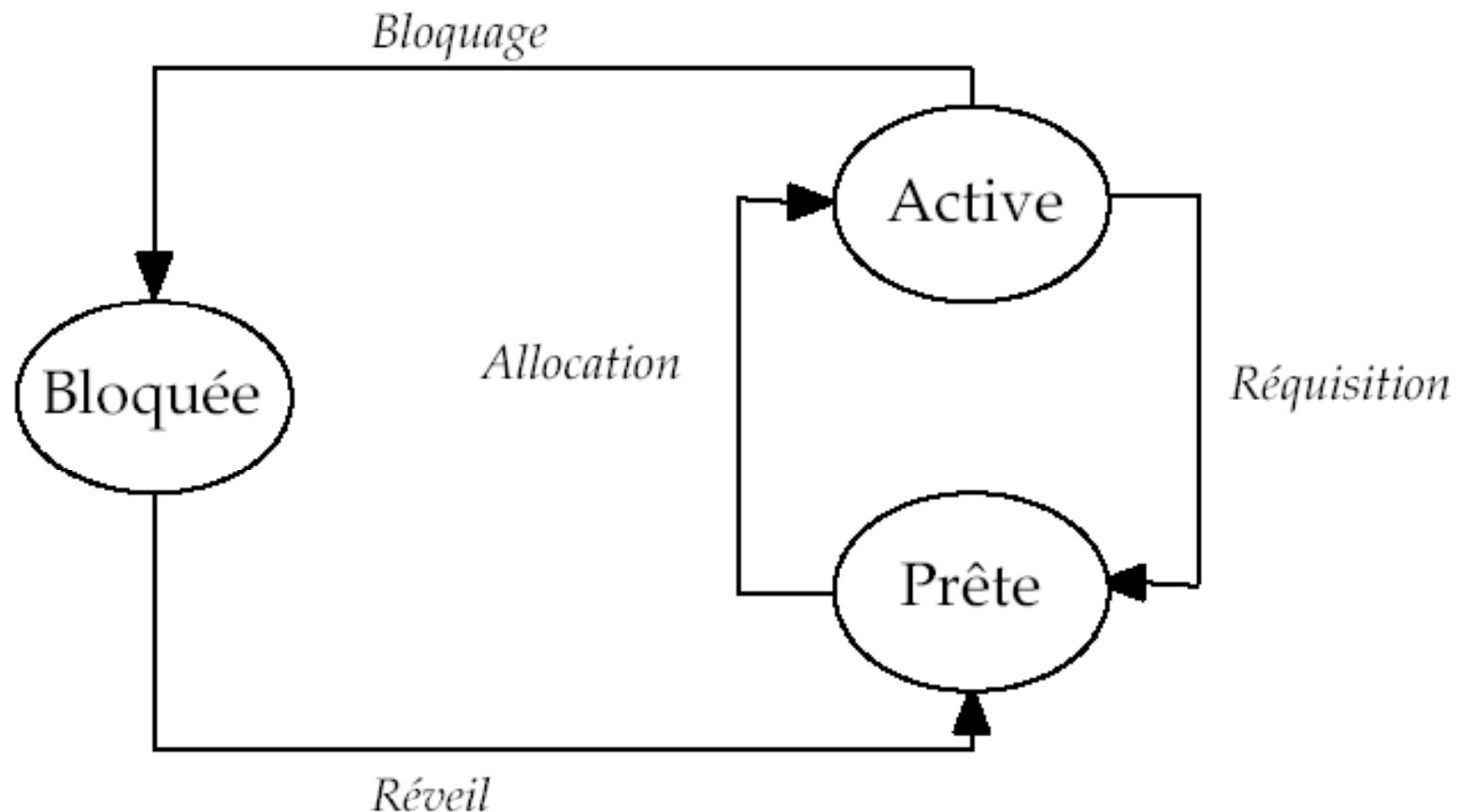
États d'une tâche

Contexte mono-processeur : une tâche active à la fois



États d'une tâche

Mais une tâche peut être bloquée en attente d'une E/S



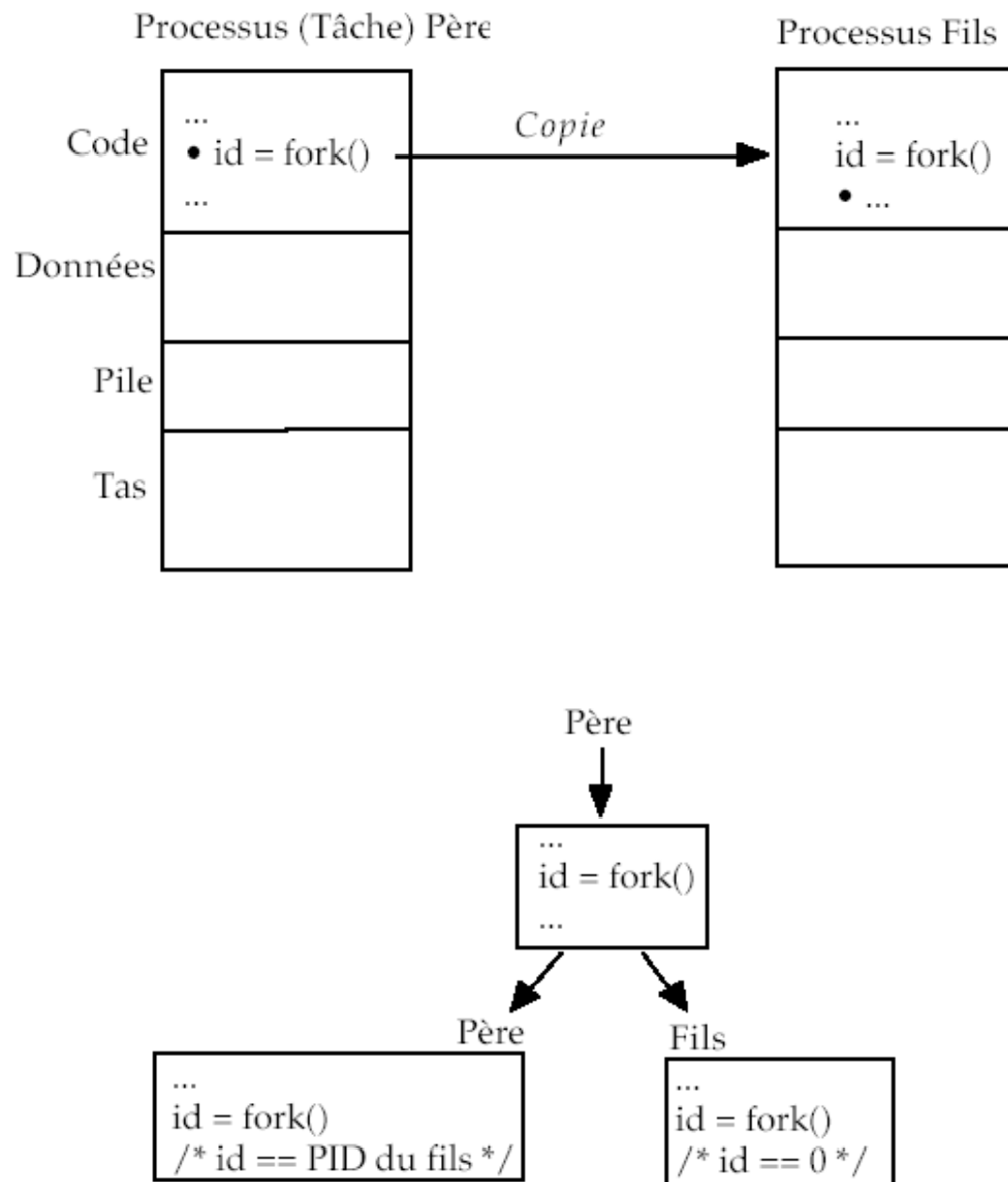
Création d'une tâche

Création dynamique (Unix, OS/2, ADA, ...)

Une tâche est toujours créée par une autre tâche

- Par copie
- Par chargement depuis un support

Exemple d'Unix



Exemple de programme C avec fork

```
/* -----
 * Schéma de création de processus.
 * Création d'un processus fils et test pour exécuter un code différent
 * dans le père et dans le fils.
 */
#include <stdio.h>           /* Pour perror */
#include <stdlib.h>          /* Pour exit */
#include <unistd.h>          /* Pour fork, getpid, getppid, sleep */
#include <sys/types.h>       /* Pour pid_t (fork, getpid, getppid) */
#include <sys/wait.h>        /* Pour wait */
int main(void)
{
    pid_t ident;
    ident = fork();
    if (ident == -1)
    {
        perror("fork");
        return EXIT_FAILURE;
    }
    /* A partir de cette ligne, il y deux processus */
    printf("Cette ligne sera affichée deux fois\n");
    if (ident == 0)
    {
        /* Code exécuté uniquement par le fils */
        printf("Je suis le fils\n");
    }
    else
    {
        /* Code exécuté uniquement par le pere */
        printf("Je suis le père\n");
    }
    return EXIT_SUCCESS;
}
```

Exemple de programme C avec fork

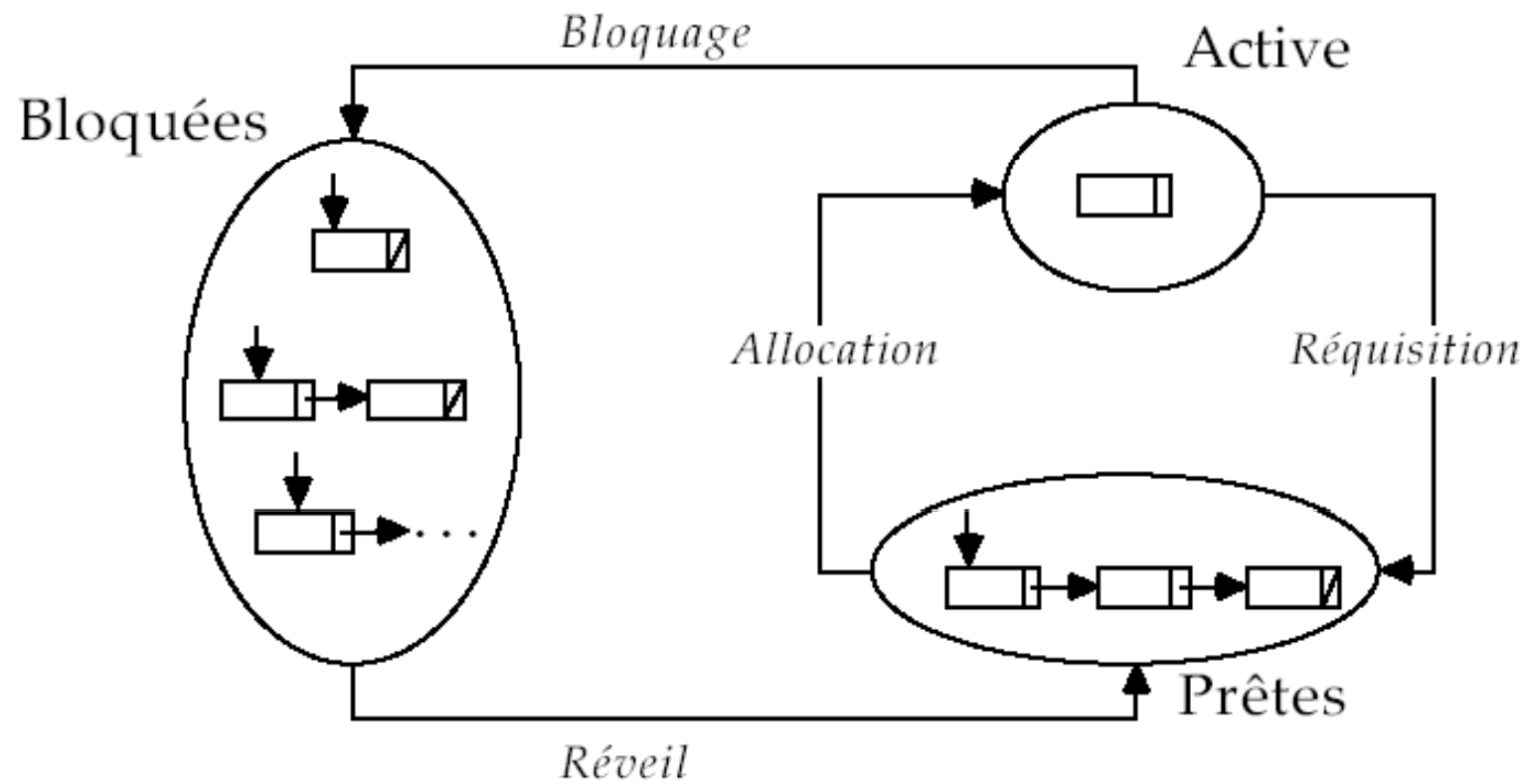
```
/* -----  
 * Schéma de création de processus.  
 * Création d'un processus fils et test pour exécuter un code différent  
 * dans le père et dans le fils.  
 */  
#include <stdio.h>                /* Pour perror */  
#include <stdlib.h>               /* Pour exit */  
#include <unistd.h>               /* Pour fork, getpid, getppid, sleep */  
#include <sys/types.h>            /* Pour pid_t (fork, getpid, getppid) */  
#include <sys/wait.h>             /* Pour wait */  
int main(void)  
{  
    pid_t ident;  
    ident = fork();  
    if (ident == -1)  
    {  
        perror("fork");  
        return EXIT_FAILURE;  
    }  
    /* A partir de cette ligne, il y deux processus */  
    printf("Cette ligne sera affichée deux fois\n");  
    if (ident == 0)  
    {  
        /* Code exécuté uniquement par le fils */  
        sleep(2); /* le fils dort pendant 2 secondes */  
        printf("Je suis le fils\n");  
    }  
    else  
    {  
        /* Code exécuté uniquement par le pere */  
        printf("Je suis le père\n");  
    }  
    return EXIT_SUCCESS;  
}
```

Exemple de programme C avec fork

```
/* -----  
 * Schéma de création de processus.  
 * Création d'un processus fils et test pour exécuter un code différent  
 * dans le père et dans le fils.  
 */  
#include <stdio.h>                /* Pour perror */  
#include <stdlib.h>               /* Pour exit */  
#include <unistd.h>              /* Pour fork, getpid, getppid, sleep */  
#include <sys/types.h>           /* Pour pid_t (fork, getpid, getppid) */  
#include <sys/wait.h>            /* Pour wait */  
int main(void)  
{  
    pid_t ident;  
    ident = fork();  
    if (ident == -1)  
    {  
        perror("fork");  
        return EXIT_FAILURE;  
    }  
    /* A partir de cette ligne, il y deux processus */  
    printf("Cette ligne sera affichée deux fois\n");  
    if (ident == 0)  
    {  
        /* Code exécuté uniquement par le fils */  
        sleep(2); /* le fils dort pendant 2 secondes */  
        printf("Je suis le fils\n");  
    }  
    else  
    {  
        /* Code exécuté uniquement par le pere */  
        wait(NULL); /* le père attend la fin de l'un de ses fils */  
        printf("Je suis le père\n");  
    }  
    return EXIT_SUCCESS;  
}
```

Gestion du processeur

États d'une tâche, Files d'attente



Gestion du processeur

Ordonnanceur (*scheduler*)

Ensemble des algorithmes utilisés pour faire les transitions entre tâches.

Distributeur (*dispatcher*)

Plus particulièrement chargé de l'allocation.

Politique (ou stratégie) d'ordonnancement

Préemptif ou non préemptif

Lié à la gestion des files d'attente

Stratégies d'ordonnancement

Temps partagé	Temps réel
Rendre le système agréable à utiliser : • tâches interactives • algorithmes complexes, avec vieillissement des priorités, régulation de la charge	Le système doit être efficace et sûr : • hiérarchie de tâches (priorités) • souplesse (adaptation à de nombreuses applications, même très contraintes)

Non préemptif	Préemptif
Exécution de la tâche courante jusqu'à appel du noyau ou interruption externe Le noyau décide ou non de commuter la tâche active ➤ Facile à réaliser ➤ Peu fiable ➤ Très bon rendement	La tâche est de toute façon interrompue en fin de quantum Commutation en fonction des priorités ➤ Plus difficile à réaliser ➤ Plus fiable ➤ Meilleure prise en compte des tâches prioritaires ➤ Rendement affaibli

Temps de latence aux interruptions

Durée maximum pendant laquelle les interruptions sont masquées.
Critère important des applications temps réel à cause des tâches de sécurité.

Unix : temps de commutation $\pm 1\text{ms}$, temps de latence non précisé

Noyau temps réel : temps de commutation 40 à 250 μs , temps de latence inférieur à 100 μs

Critères de qualité de l'ordonnancement

Efficacité/Rendement : le maximum de temps doit être consacré à l'application

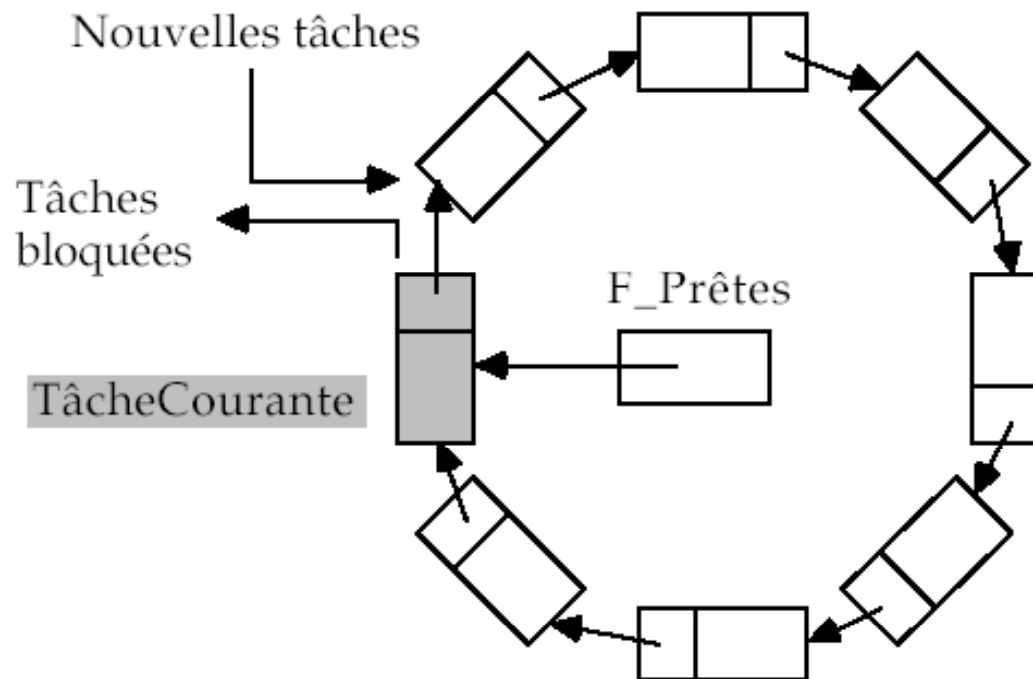
Temps de réponse : le plus faible possible (lié au temps de latence)

Impartialité : partage équitable entre tâches

Débit : le plus de tâches possibles dans un temps donné

Tourniquet : ordonnancement circulaire sans priorité

La file des tâches prêtes est circulaire et gérée en FIFO.

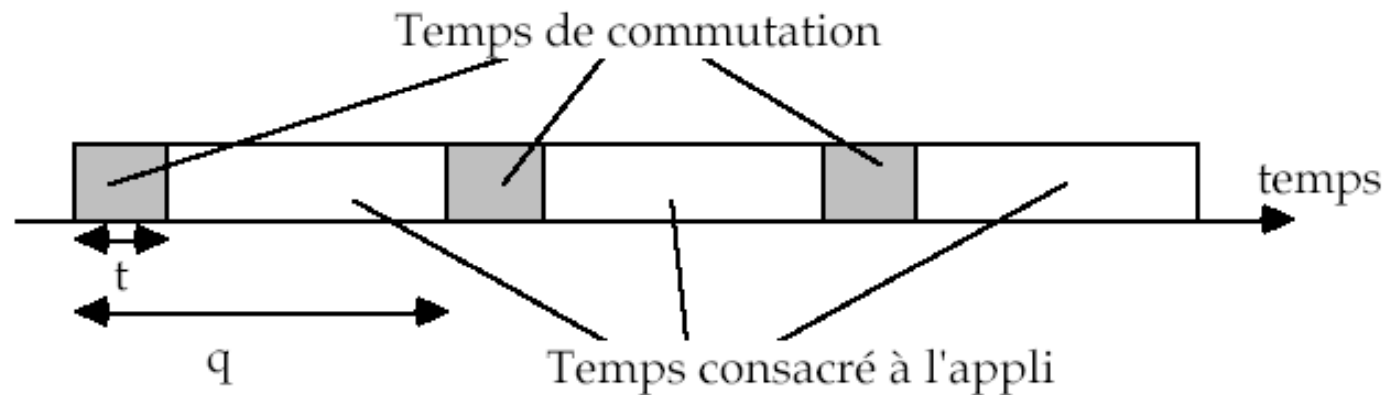


Sans réquisition : très efficace s'il y a peu de tâches qui se bloquent souvent (E/S)

Avec réquisition : c'est une méthode impartiale pour les tâches de même priorité.

Tourniquet : choix du quantum

Compromis entre débit et efficacité



$$\text{Efficacité } E = \frac{q - t}{q} = \text{rapport } \frac{\text{temps consacré à l'appli}}{\text{temps total}}$$

Exemple : $t = 1\text{ms}$

$$E = 0.8 \text{ si } q = 5\text{ms}$$

$$E = 0.98 \text{ si } q = 50\text{ms}$$

$$\text{Débit } D = \frac{1}{q} = \text{nombre de tâches traitées par seconde}$$

Exemple :

$$D = 200 \text{ si } q = 5\text{ms}$$

$$D = 20 \text{ si } q = 50\text{ms}$$

Meilleur débit = meilleur temps de réaction

Meilleure efficacité = temps total d'exécution plus court

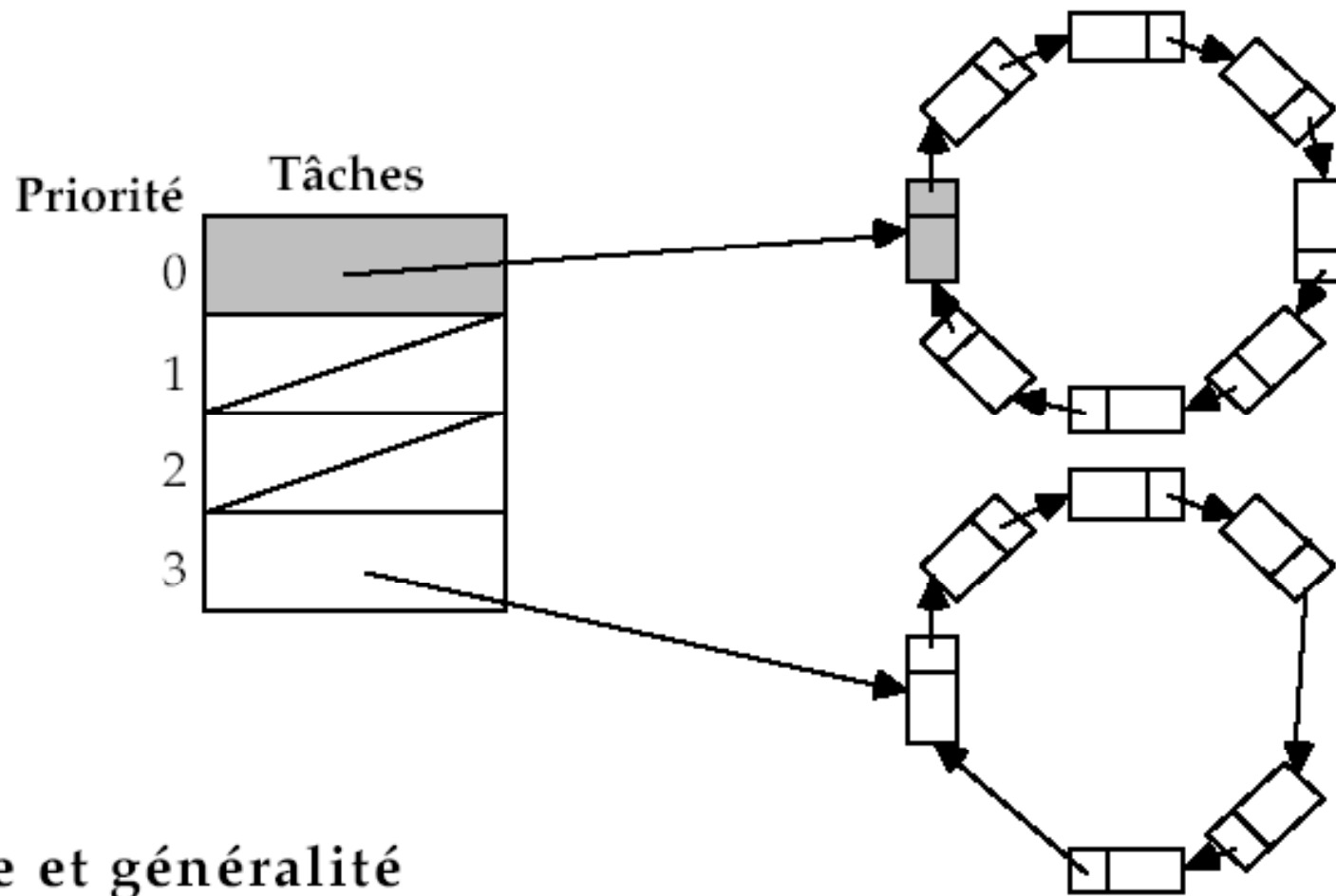
Bien sûr il faut que t soit le plus petit possible.

Priorité pure

- Toutes les tâches ont une priorité, et pour une priorité on a au plus une tâche
- La préemption présente peu d'intérêt : la tâche la plus prioritaire s'exécute jusqu'à sa fin ou jusqu'au blocage.
- L'ordonnancement est donc lié aux interruptions externes

Méthode mixte : tourniquet multi-niveaux

On a quelques niveaux de priorité mais on peut avoir plusieurs tâches au même niveau de priorité.



Souplesse et généralité

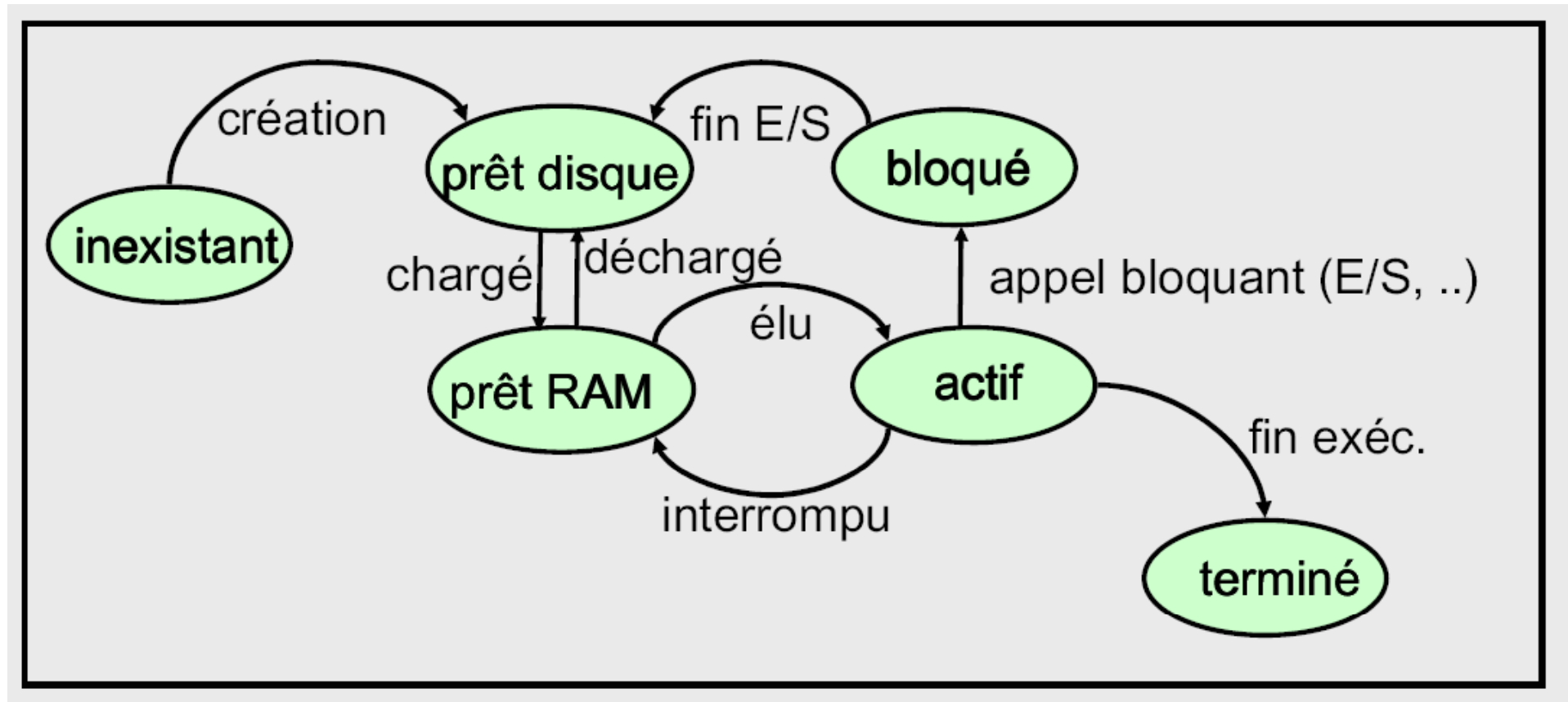
une tâche par niveau = priorité pure

toutes les tâches au même niveau = tourniquet simple

Algorithmes d'ordonnancement à plusieurs niveaux

- **Ensemble des processus prêts trop important pour tenir en mémoire centrale**
- **Certains sont déchargés sur disque, ce qui rend leur activation plus longue**
- **Le processus élu est toujours pris parmi ceux présents en mémoire centrale**
- **En parallèle, on utilise un deuxième algorithme d'ordonnancement pour gérer les déplacements des processus prêts entre le disque et la mémoire centrale**

Algorithmes d'ordonnancement à plusieurs niveaux



Autre méthode d'ordonnancement

Systèmes à temps partagé de type Unix : l'algorithme d'ordonnancement est plus complexe :

- on privilégie les tâches courtes et les tâches interactives

- la priorité diminue avec le temps

- la durée du quantum alloué peut elle aussi diminuer

