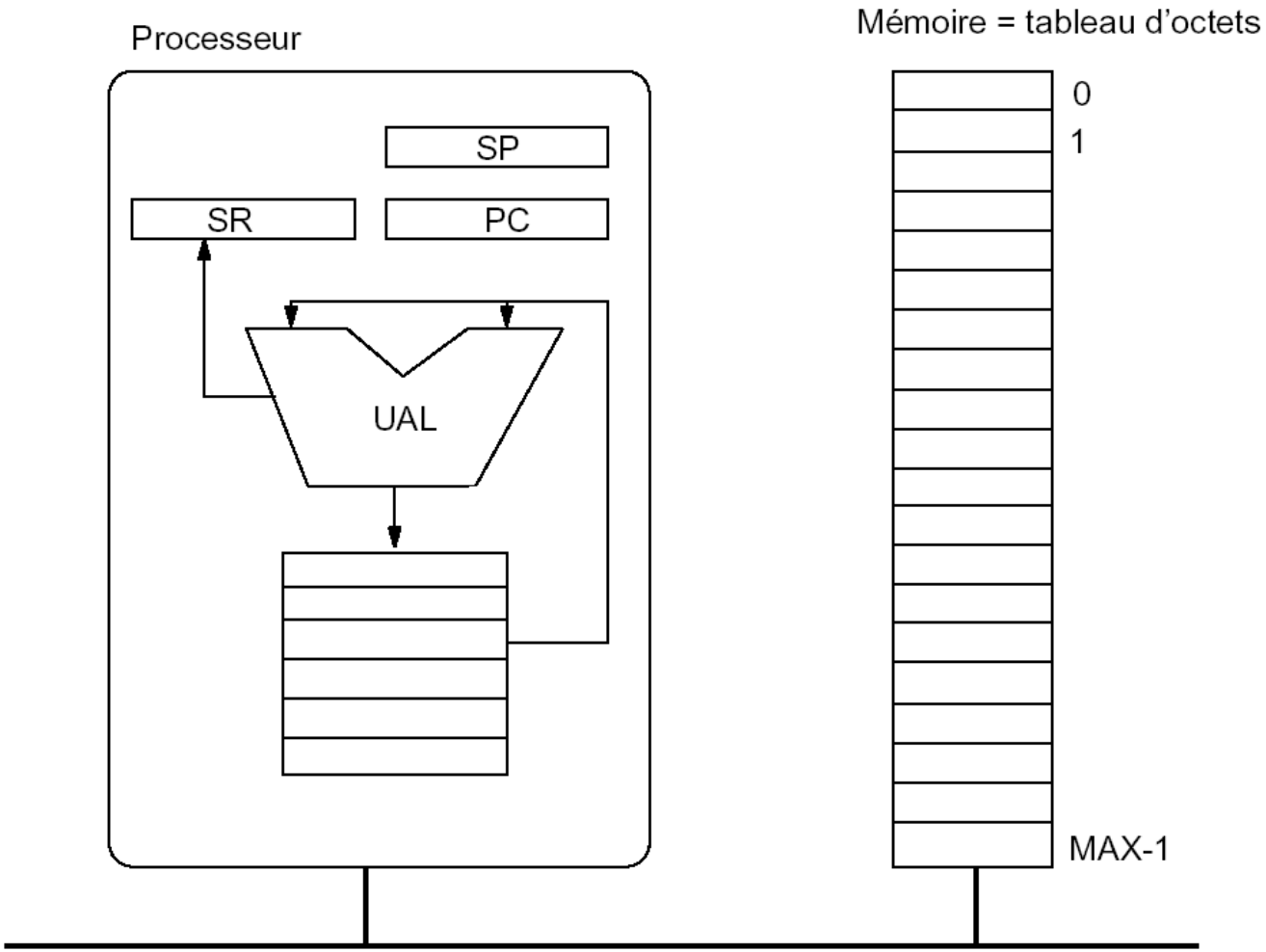
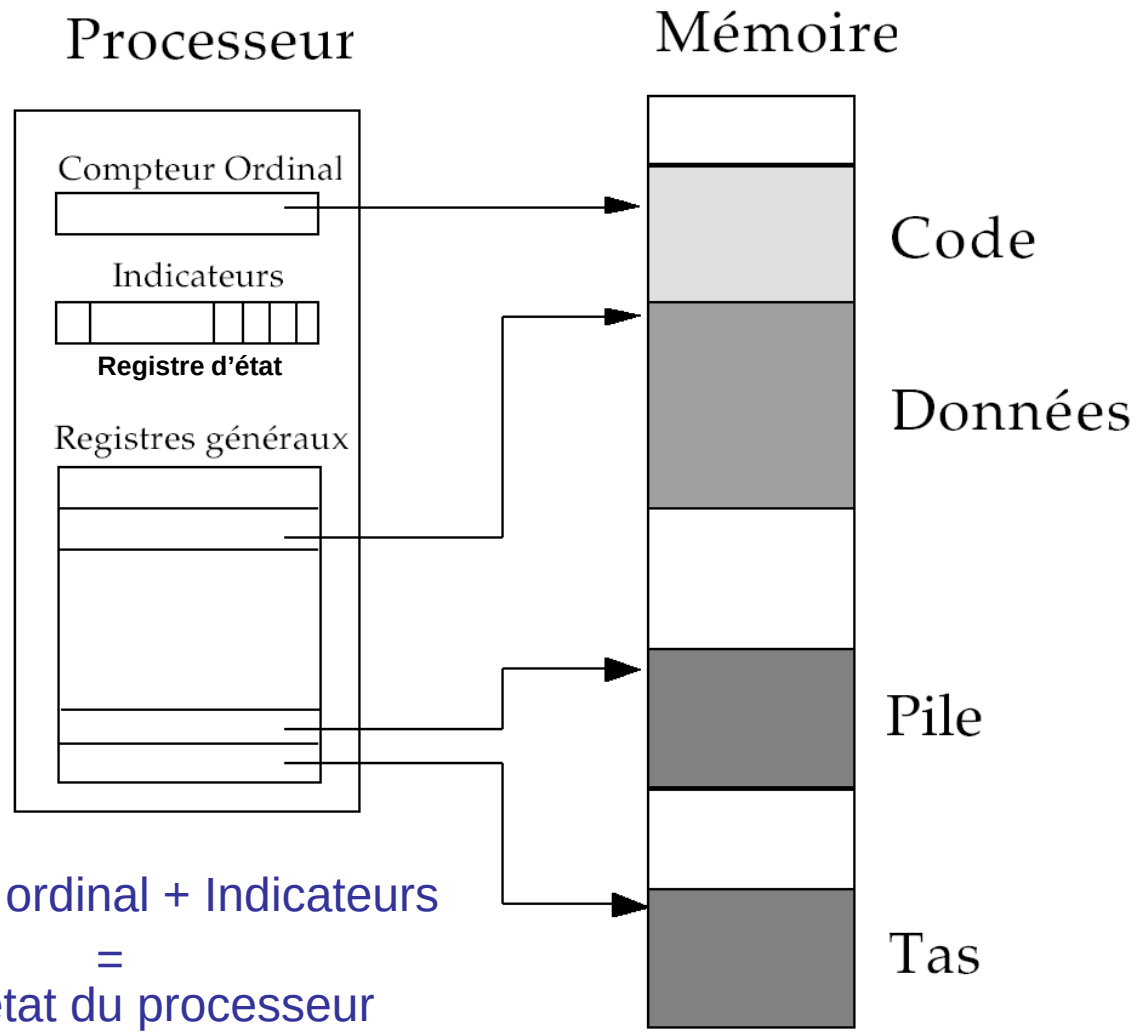
Mécanismes d’exécution

Gestion de processus

Mémoire et Processeur

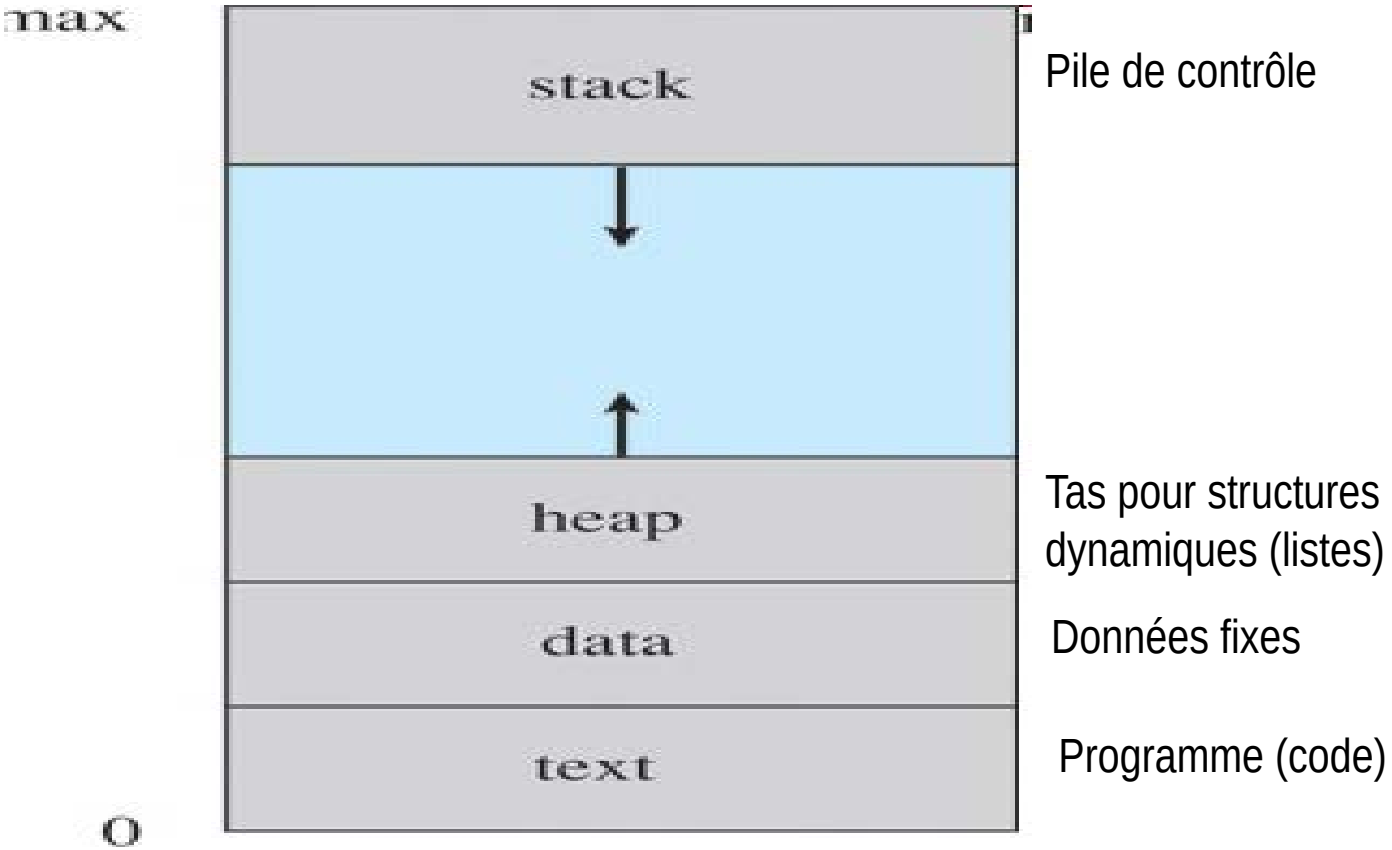


Modèle d'exécution d'une tâche



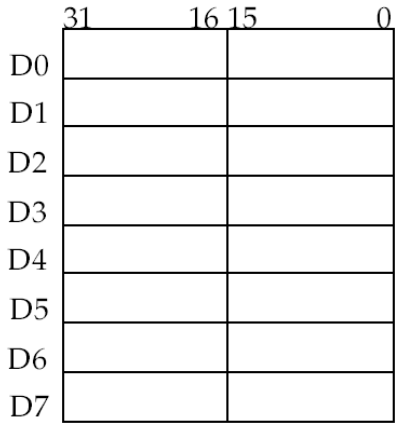
Compteur ordinal + Indicateurs
=
Mot d'état du processeur

Configuration typique de mémoire pour une tâche

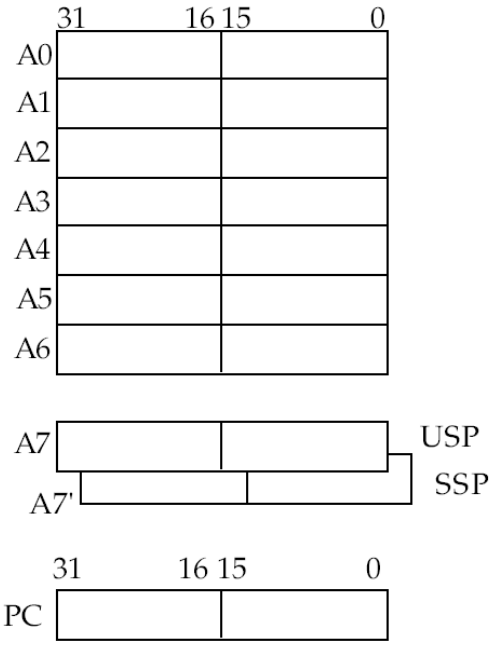


Exemple : le Motorola 68000

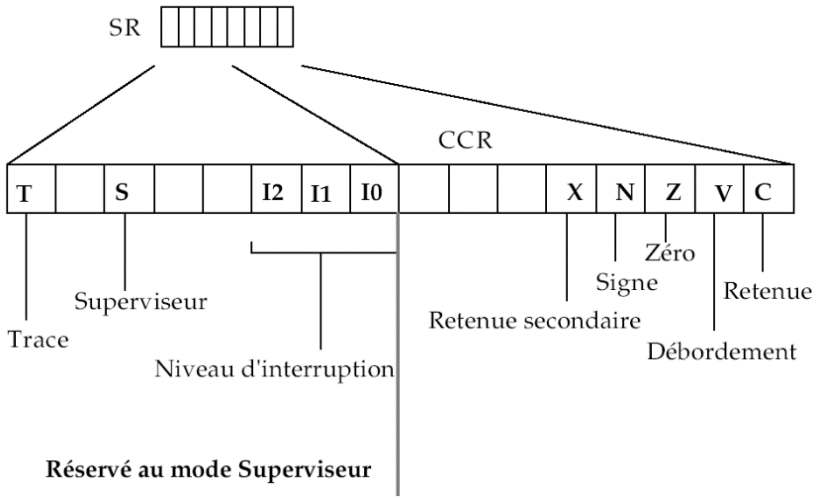
Registres de donnée



Registres d'adresse

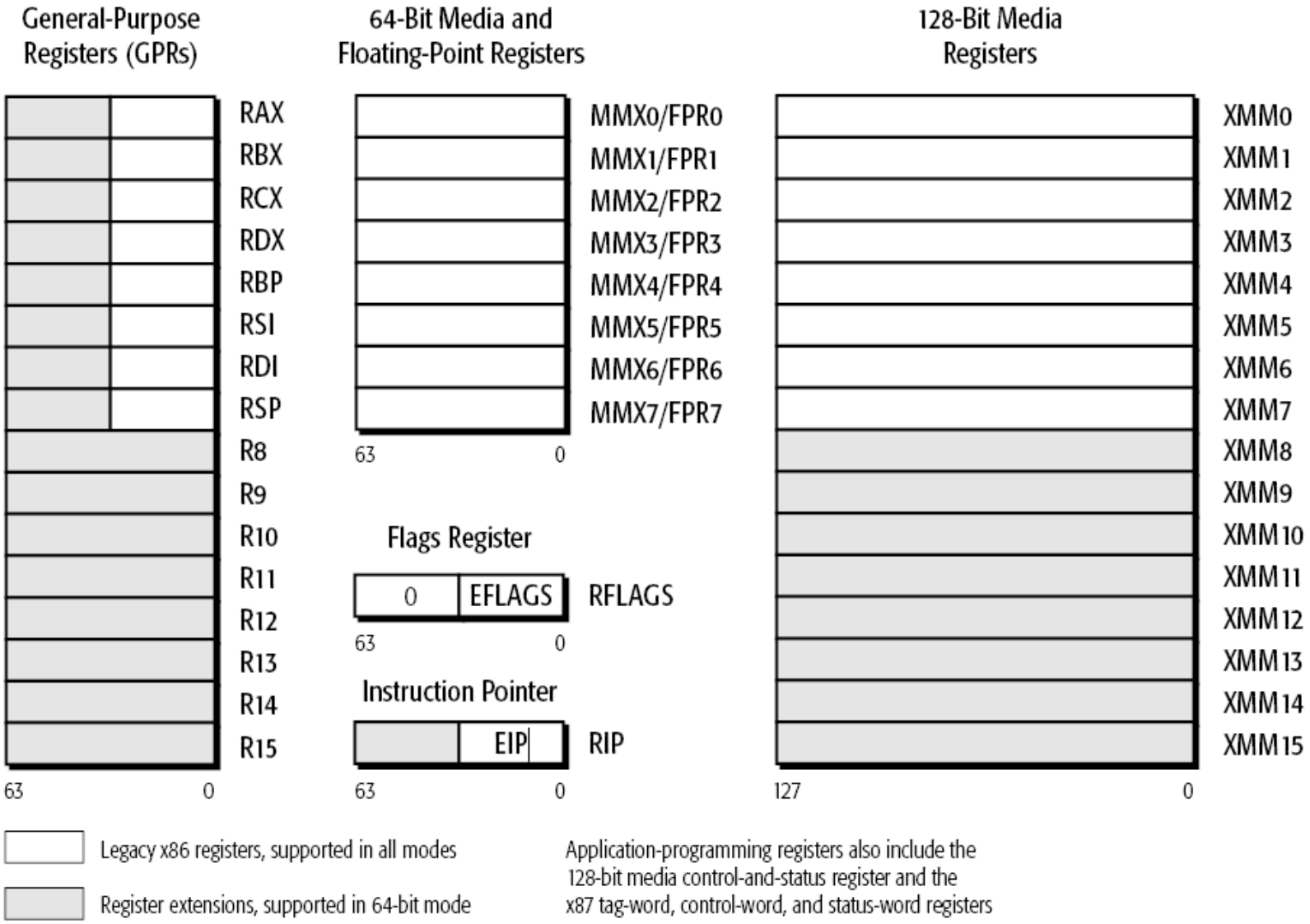


Registre d 'état

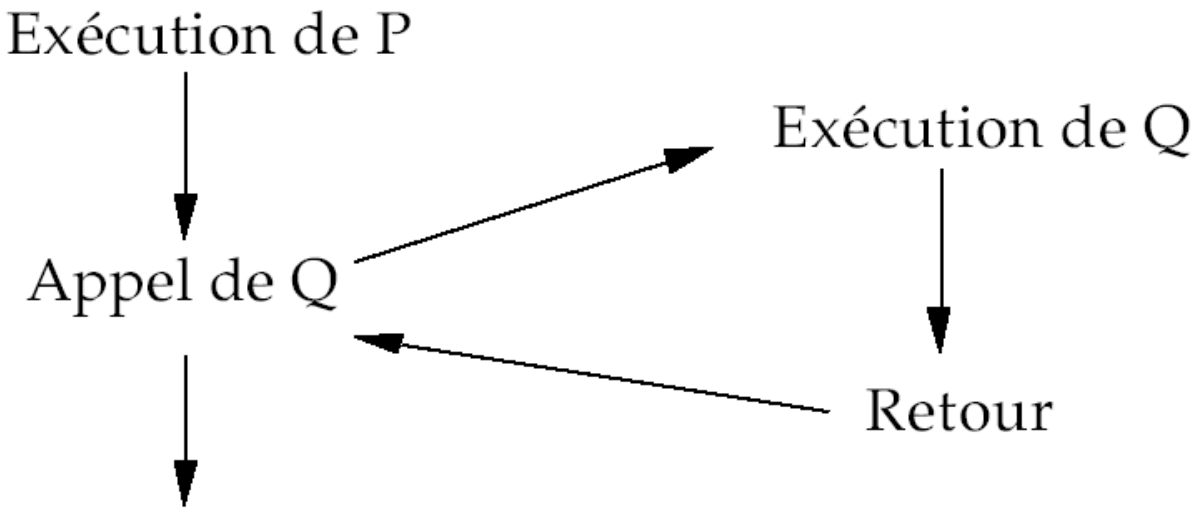


L'ensemble des valeurs des registres pour un processus donné est appelé le **contexte** du processus.

Exemple de processeur 64 bits



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

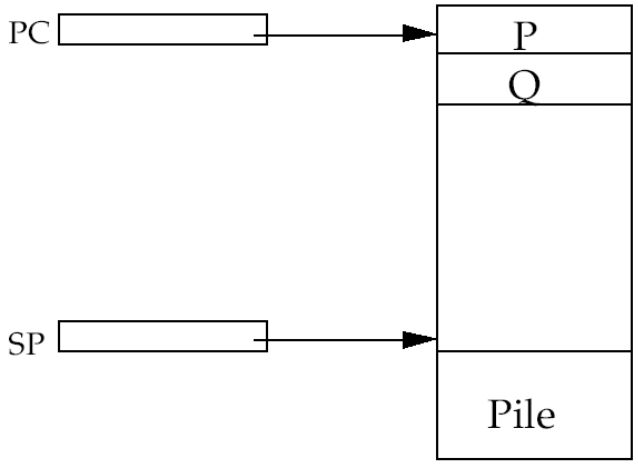
Retour

Préparation des résultats transmis par Q

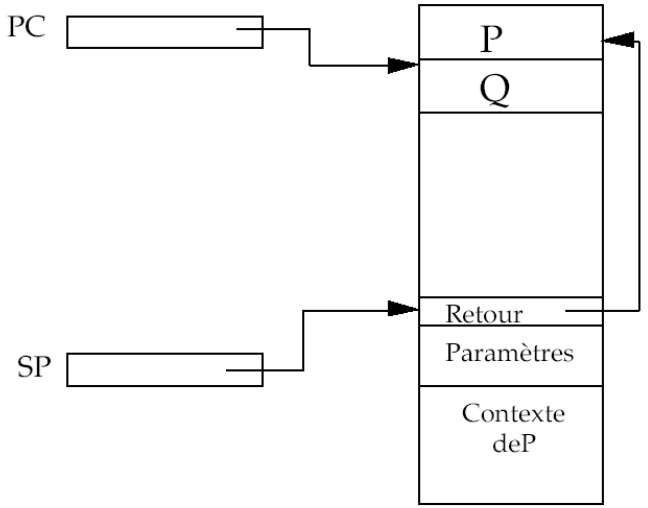
Restauration du contexte de P avant l'appel

Réalisation avec une pile

État avant l'appel

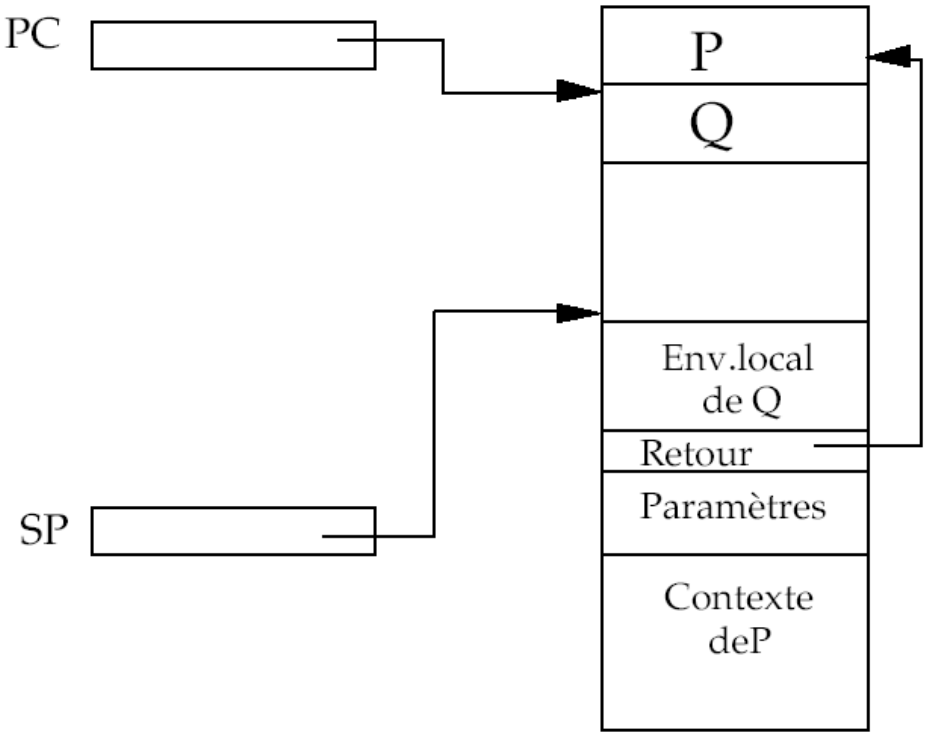


P : Préparation des paramètres et commutation

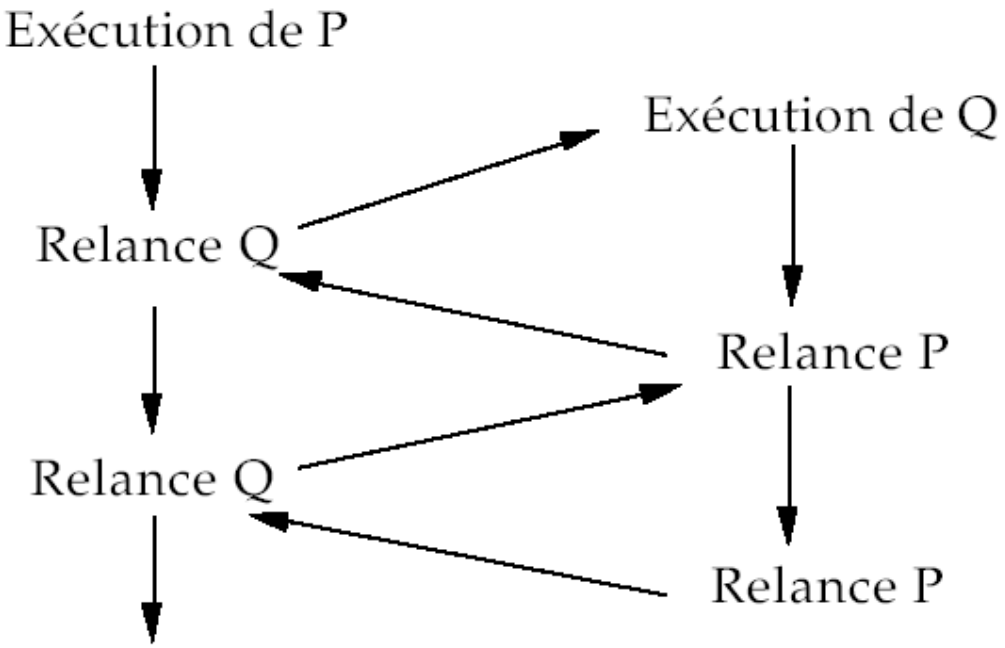


Réalisation avec une pile

Q : préparation de l'environnement local



Fonctionnement en co-routines



Relance (Resume) = Séquence d'appel

Préparation des paramètres transmis

Sauvegarde du contexte courant

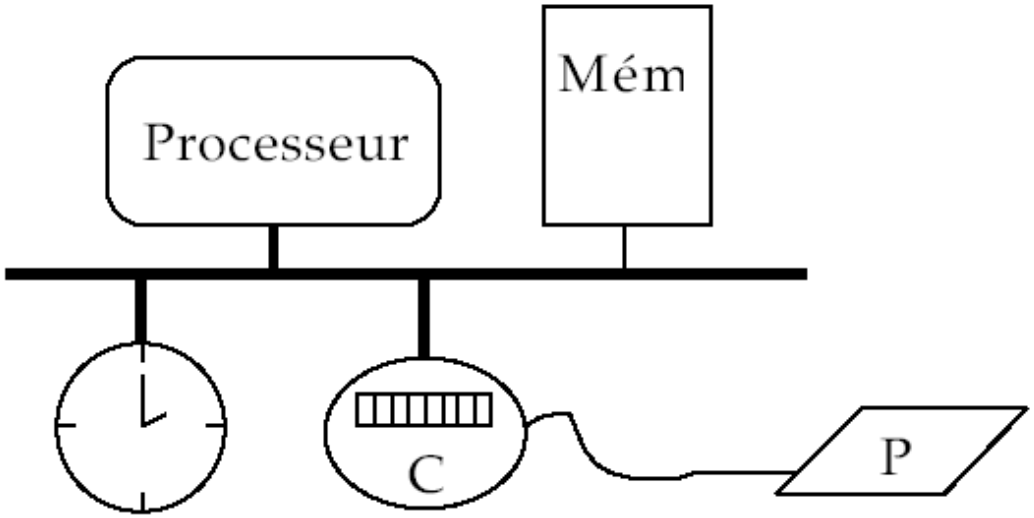
Remplacement de l'ancien contexte par le nouveau

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

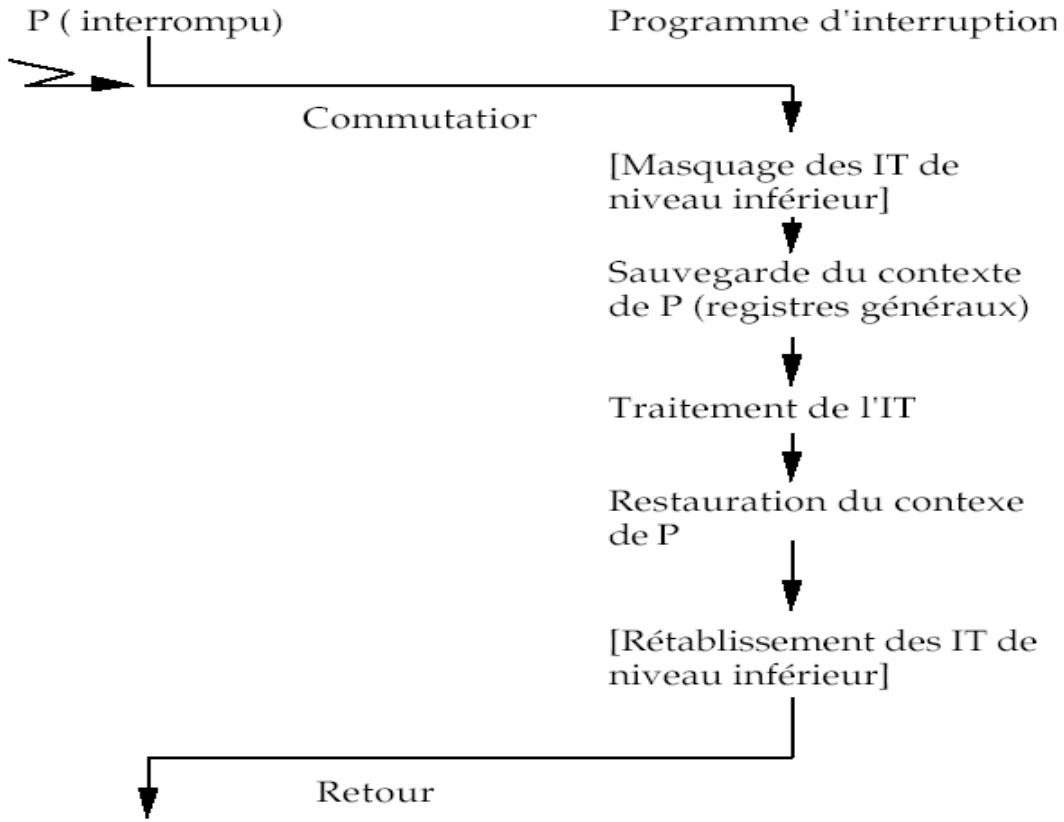
Mécanisme	Cause	Utilisation
Interruption	Extérieure à l'instruction en cours	Réaction à un évenement externe (E/S, horloge, sécurité)
Déroutement	Déclenché par l'instruction en cours	Traitement d'erreurs (débordement, division par 0, instruction illégale)
Appel au superviseur	Exception voulue, programmée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

Commutation de contexte
en cas d'exception

- 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
- 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)

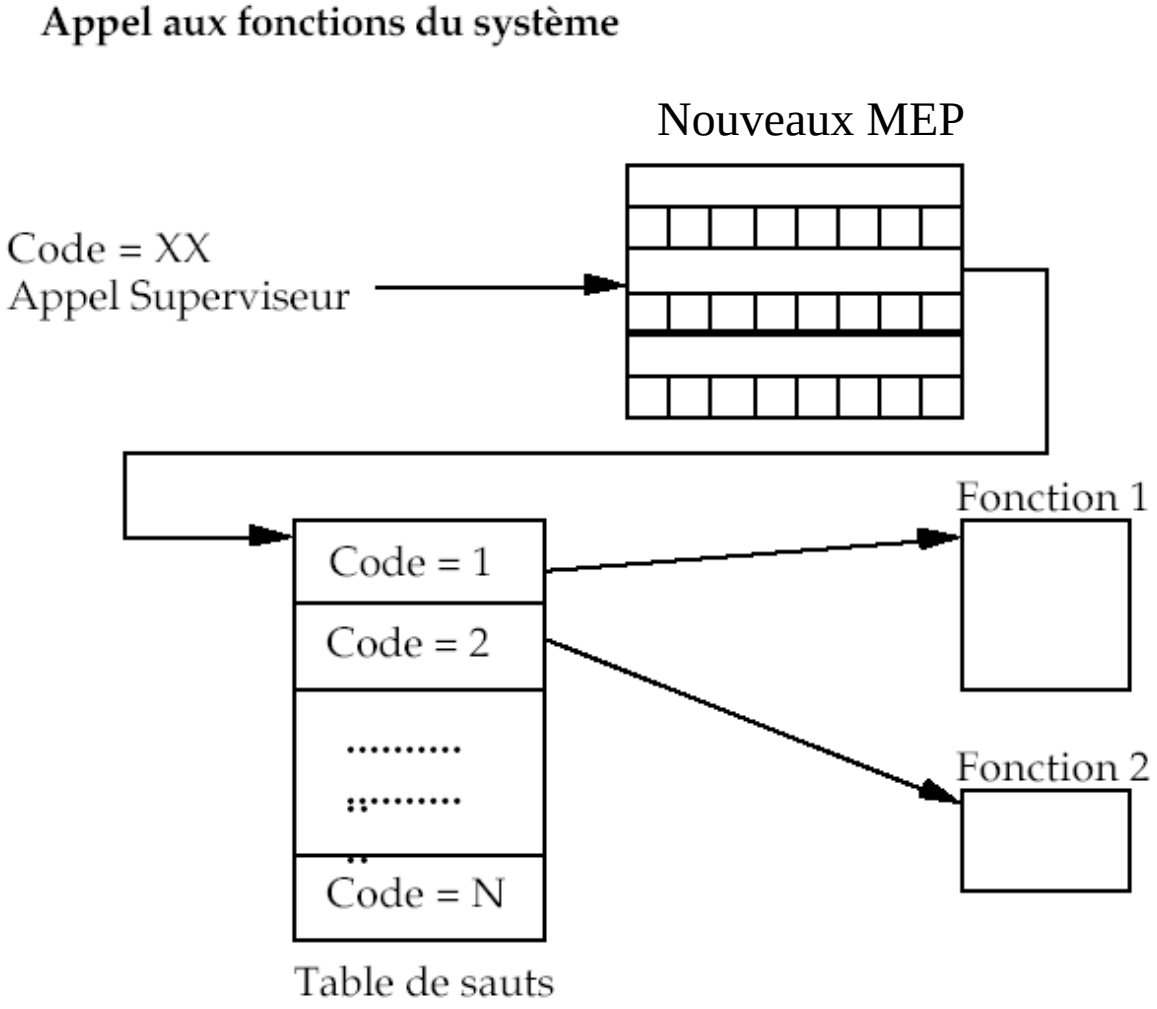
Instruction non-exécutable (code inconnu, erreur d'adresse,...)

Appels au superviseur

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

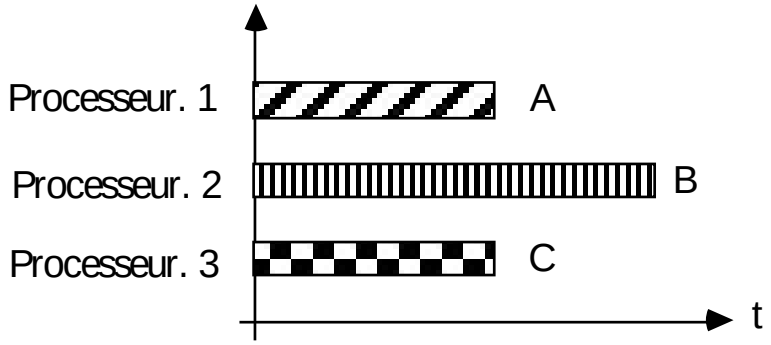
Appel aux fonctions du système

Déroutements et appels au superviseur

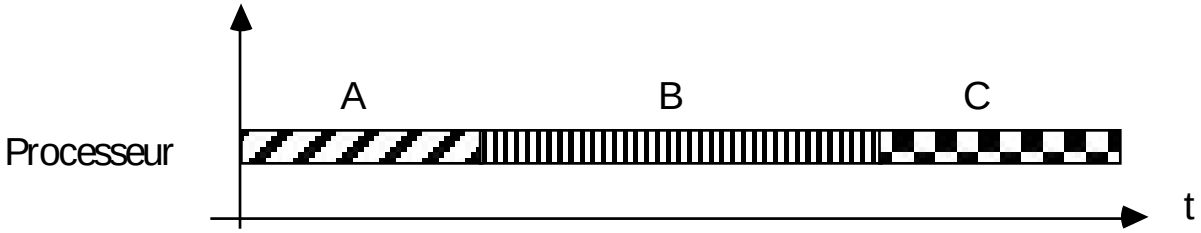


Le Multitâches

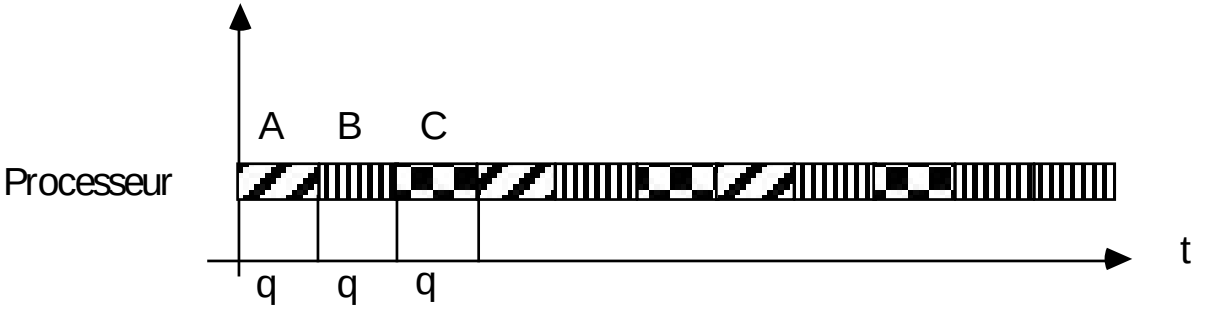
- Système multiprocesseurs



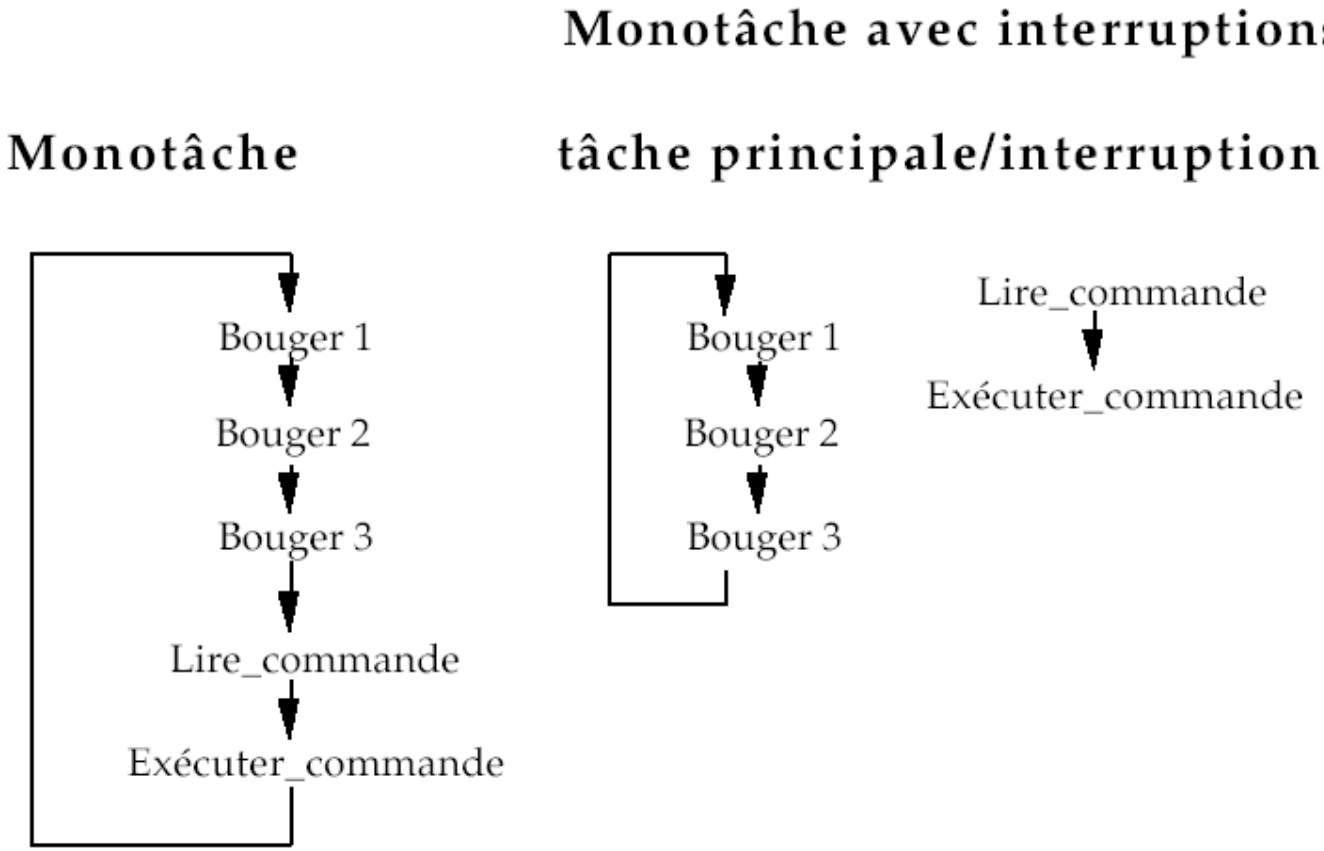
- Système Monoprocesseur, monotâche



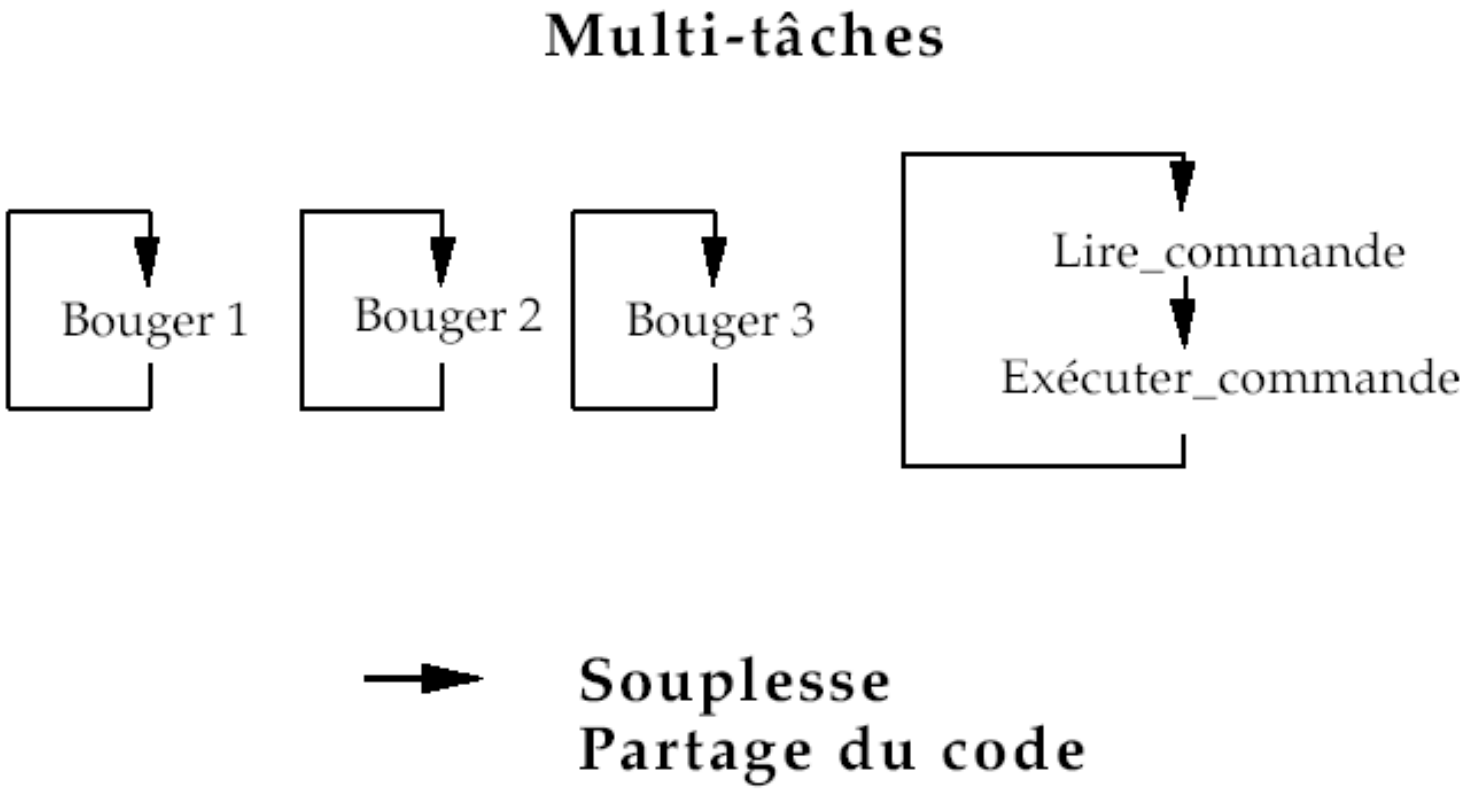
- Système Monoprocesseur, multi-tâches



Exemples : jeux vidéos

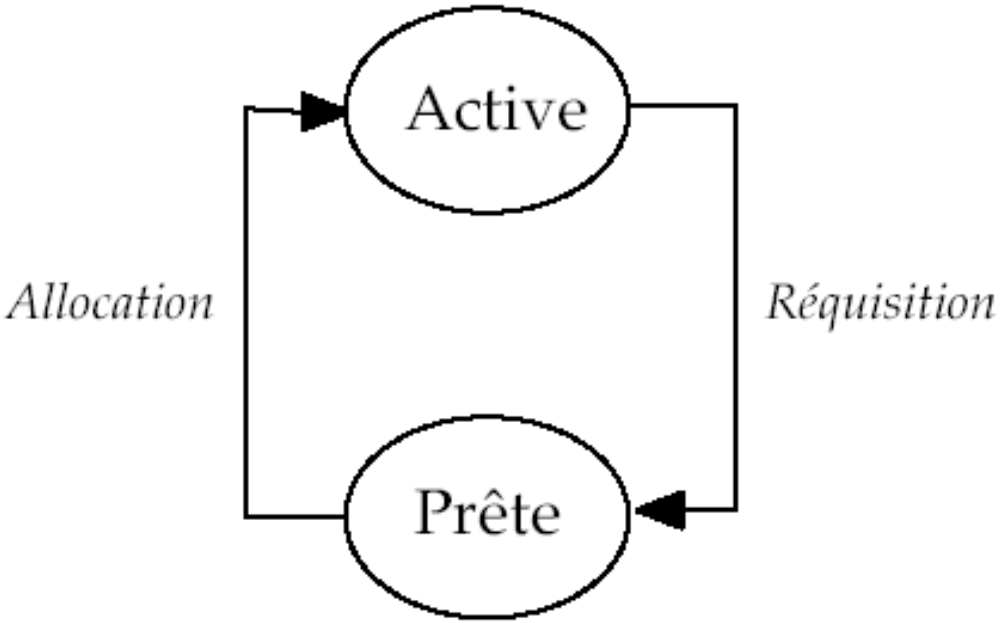


Exemples : jeux vidéos



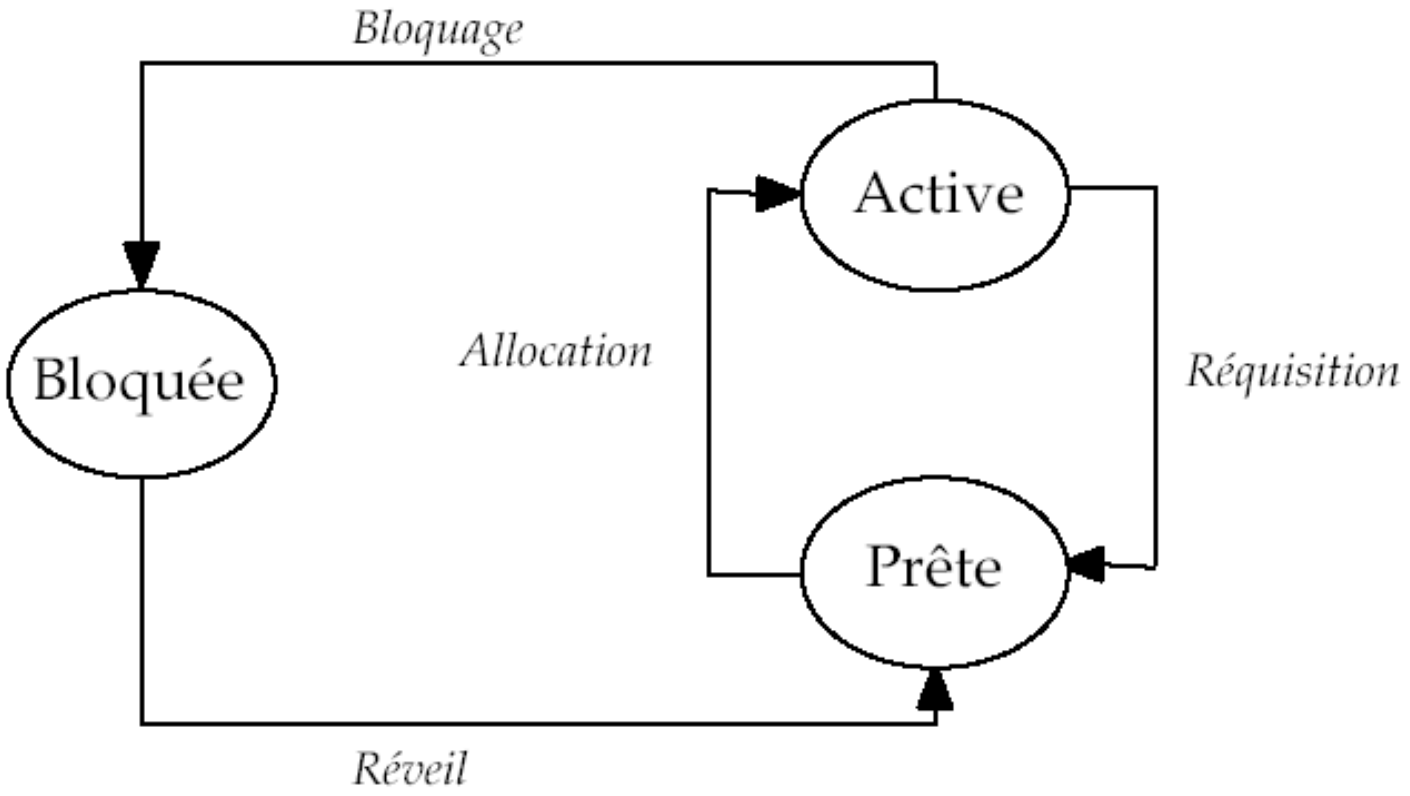
États d'une tâche

Contexte mono-processeur : une tâche active à la fois



États d'une tâche

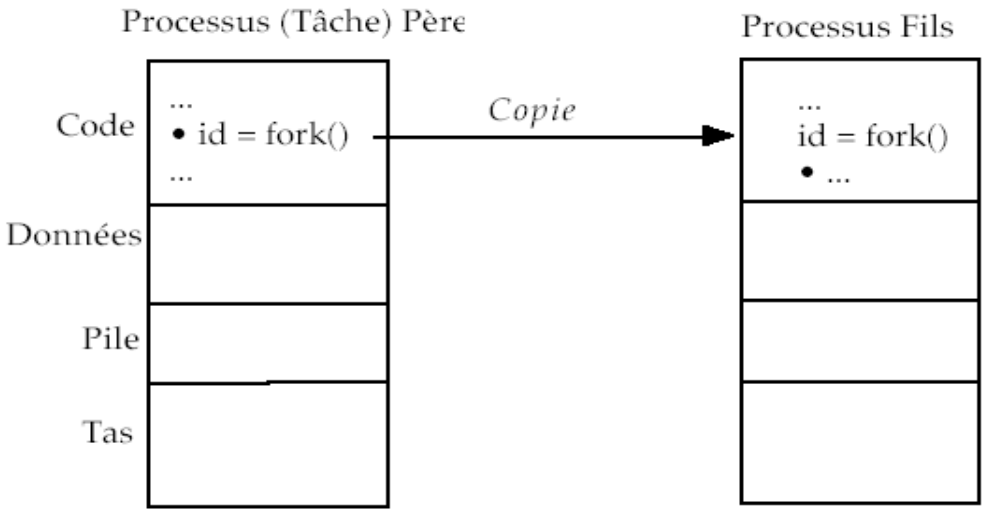
Mais une tâche peut être bloquée en attente d'une E/S

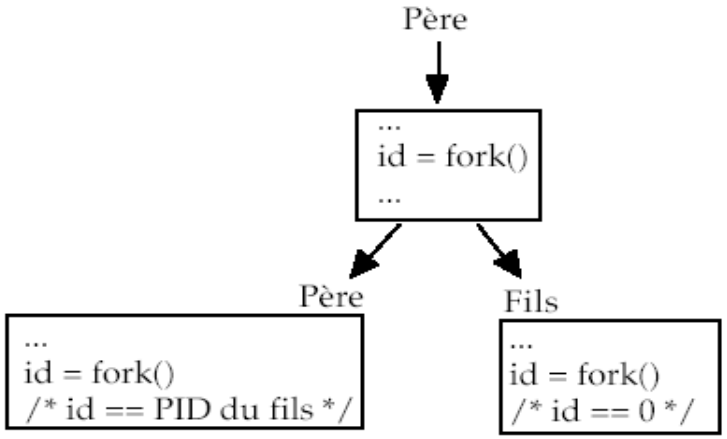


Tâche = Processus

- Processus: **un programme en cours d'exécution**
  - Possède des ressources de mémoire, périphériques, etc.
- Les processus peuvent créer d'autres processus, formant une hiérarchie
  - Par copie (instruction fork ou équivalente)
  - Par chargement depuis un support
- Sous UNIX chaque commande engendre la création d'un processus

Création d'une tâche sous UNIX





Exemple de programme C avec fork

```c
/* ----------------------------------------------------------------------
 * 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; } // pb système
  /* 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

```c
/* ------------------------------------------------------------------------
 * 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; } // pb système
  /* 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

```c
/* ------------------------------------------------------------------------
 * 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 */
    ident = wait(NULL); /* le père attend la fin de l'un de ses fils */
    printf("Je suis le père, mon fils %i est terminé\n" , ident);
  }
  return EXIT_SUCCESS;
}
```

Terminaison de processus

- Un processus exécute sa dernière instruction
  - pourrait passer des données à son parent
  - ses ressources lui sont enlevées
- Le processus parent termine l'exécution d'un fils pour raisons différentes
  - le fils a dépassé ses ressources disponibles
  - le fils n`est plus requis
- Le parent peut être le Système d'Exploitation

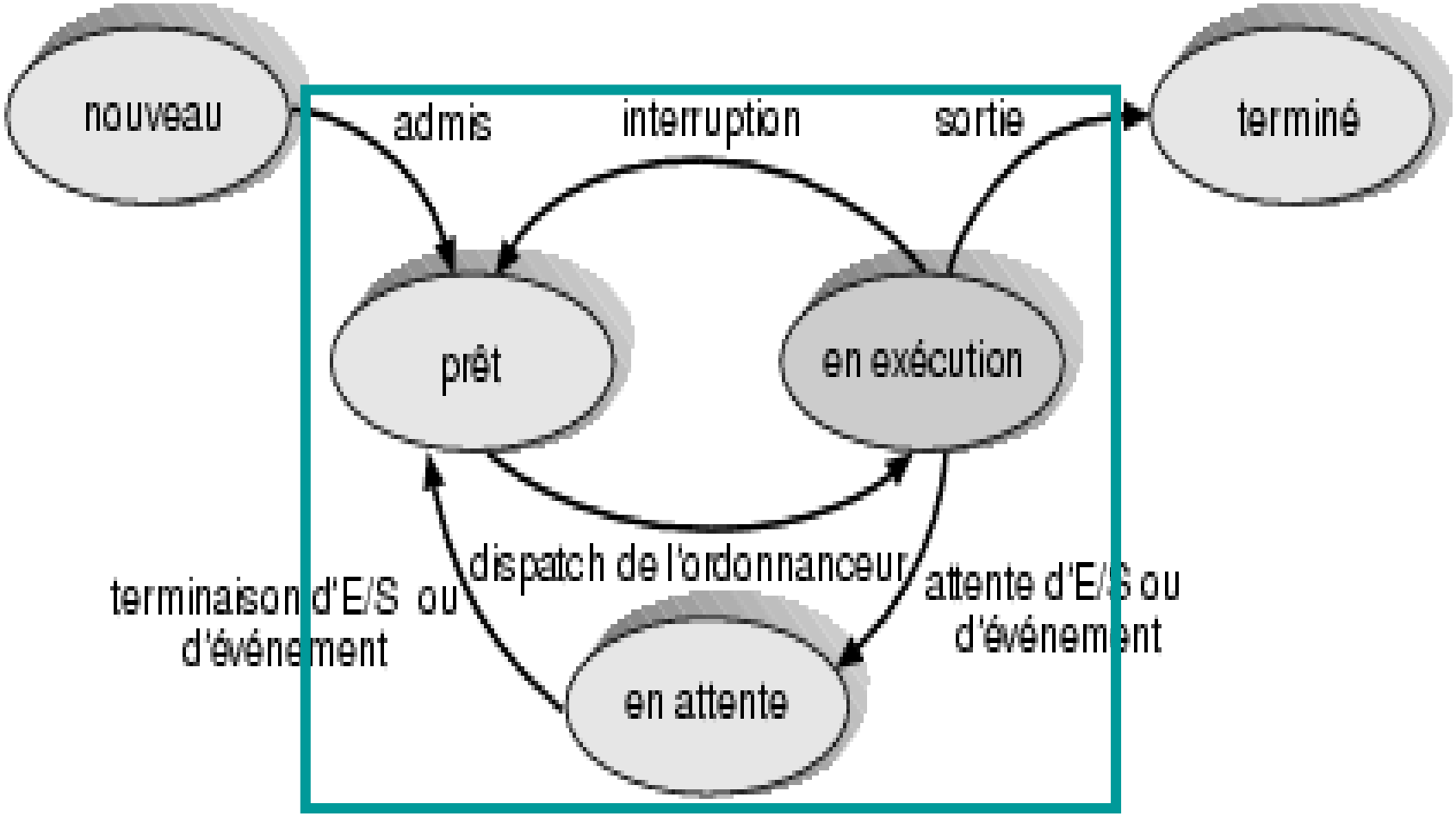
États d’un processus

- Au fur et a mesure qu’un processus s’exécute, il change d’état :
  - nouveau: le processus vient d’être créé
  - exécutant-running: le processus est en train d ’être exécuté par l ’UC
  - attente-waiting: le processus est en train d ’attendre un événement (p.ex. la fin d ’une opération d ’E/S)
  - prêt-ready: le processus est en attente d’être exécuté par l ’UC
  - terminé: fin d’exécution

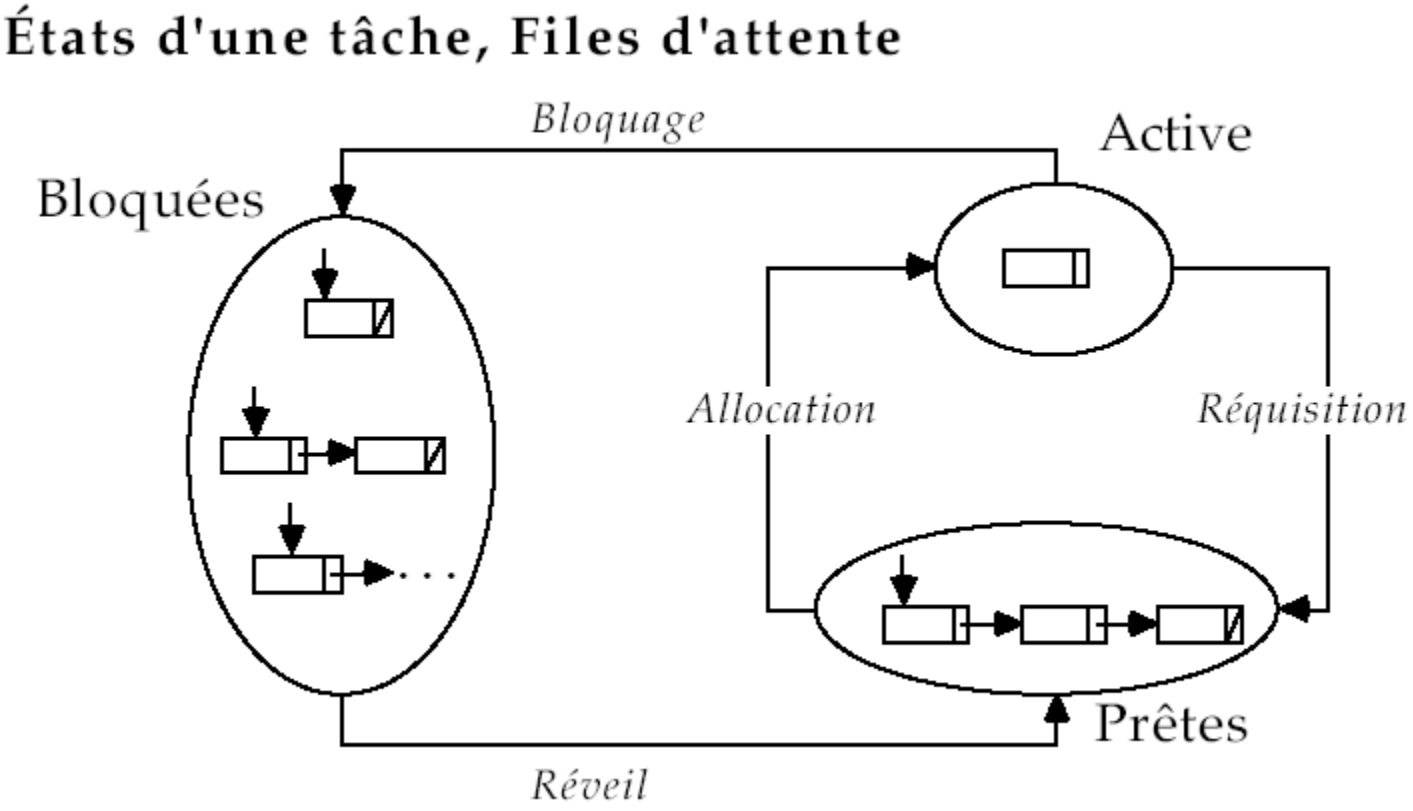
Ordonnanceur de l’UC

- Une UC est une ressource qui peut être affectée tantôt à un processus, tantôt à un autre
- Quand une UC se libère, un programme est invoqué qui décide quel processus lui sera affecté
- Ce programme est parfois appelé *gestionnaire de tâches,* scheduler en anglais
- Nous l’appellerons l’ordonnanceur

Diagramme de transition d'états d'un processus



Gestion du processeur



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 vieillisement 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	La tâche est de toute façon interrompue en fin de quantum
Le noyau décide ou non de commuter la tâche active	Commutation en fonction des priorités
↗ Facile à réaliser	↘ Plus difficile à réaliser
↘ Peu fiable	↗ Plus fiable
	↗ Meilleure prise en compte des tâches prioritaires
↗ Très bon rendement	↘ 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 ± 1ms, temps de latence non précisé

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

Critères de qualité de l'ordonnancement

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

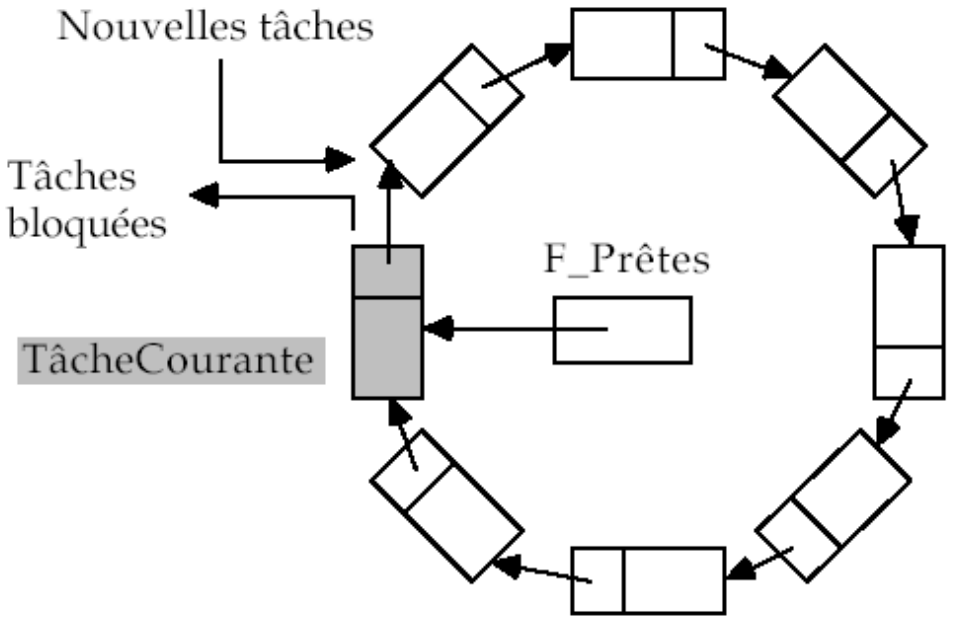
Temps de réponse : le plus faible possible (lié au temps de latence)

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.

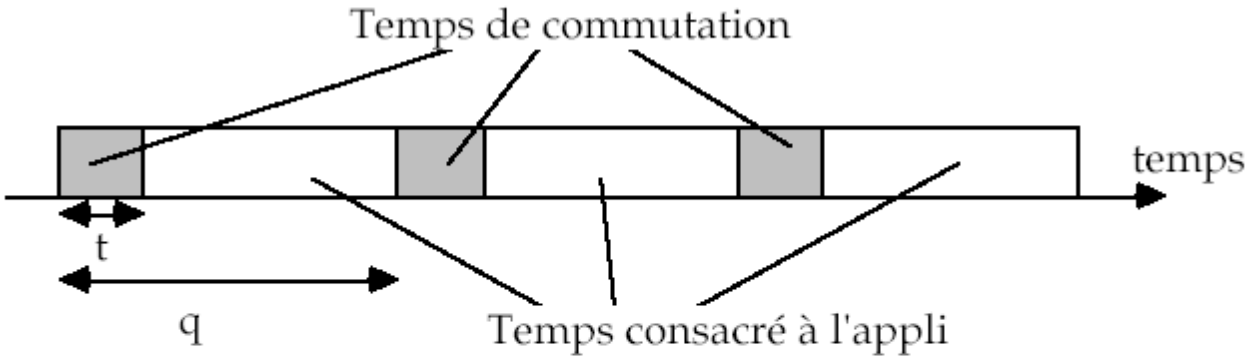


Sans réquisition : **très efficace s'il y a peu de tâches qui se bloquent souvent (E/S)**

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

Tourniquet : choix du quantum

Compromis entre débit et efficacité



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

Exemple : t = 1ms

E = 0.8 si q=5ms

E = 0.98 si q = 50ms

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

Exemple :

D = 200 si q=5ms

D = 20 si q = 50ms

Meilleur débit = meilleur temps de réaction

Meilleure efficacité = temps total d'exécution plus court

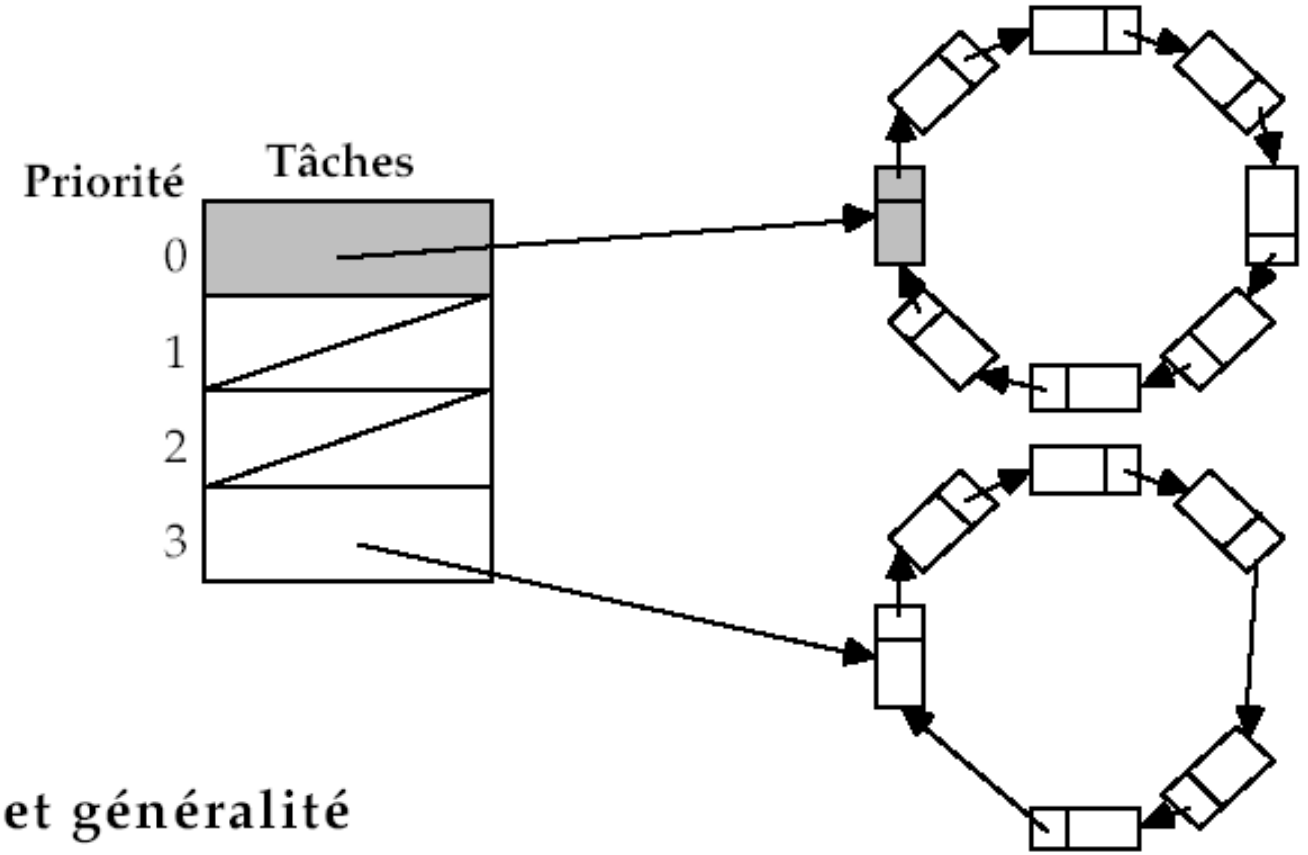
Bien sûr il faut que t soit le plus petit possible.

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'à un 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

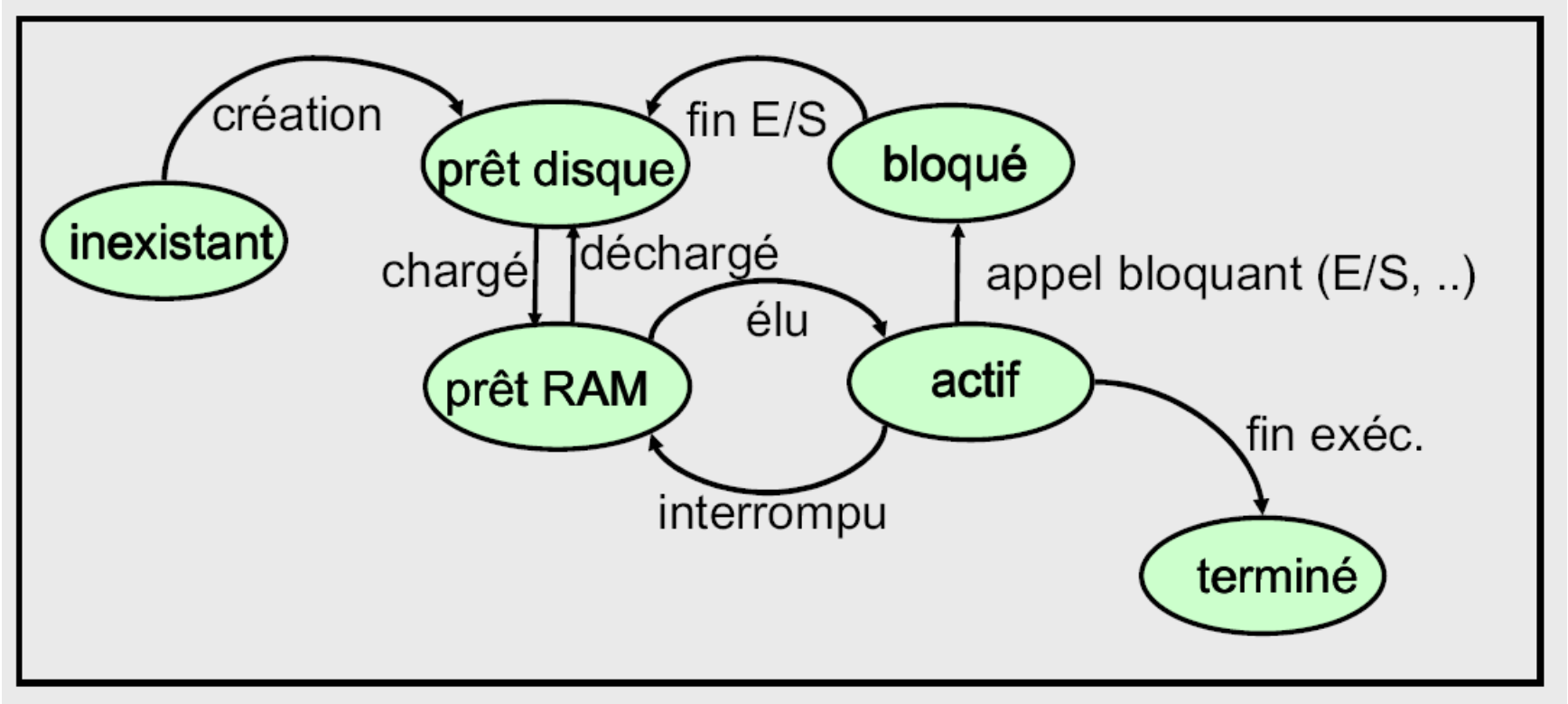
une tâche par niveau = priorité pure

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

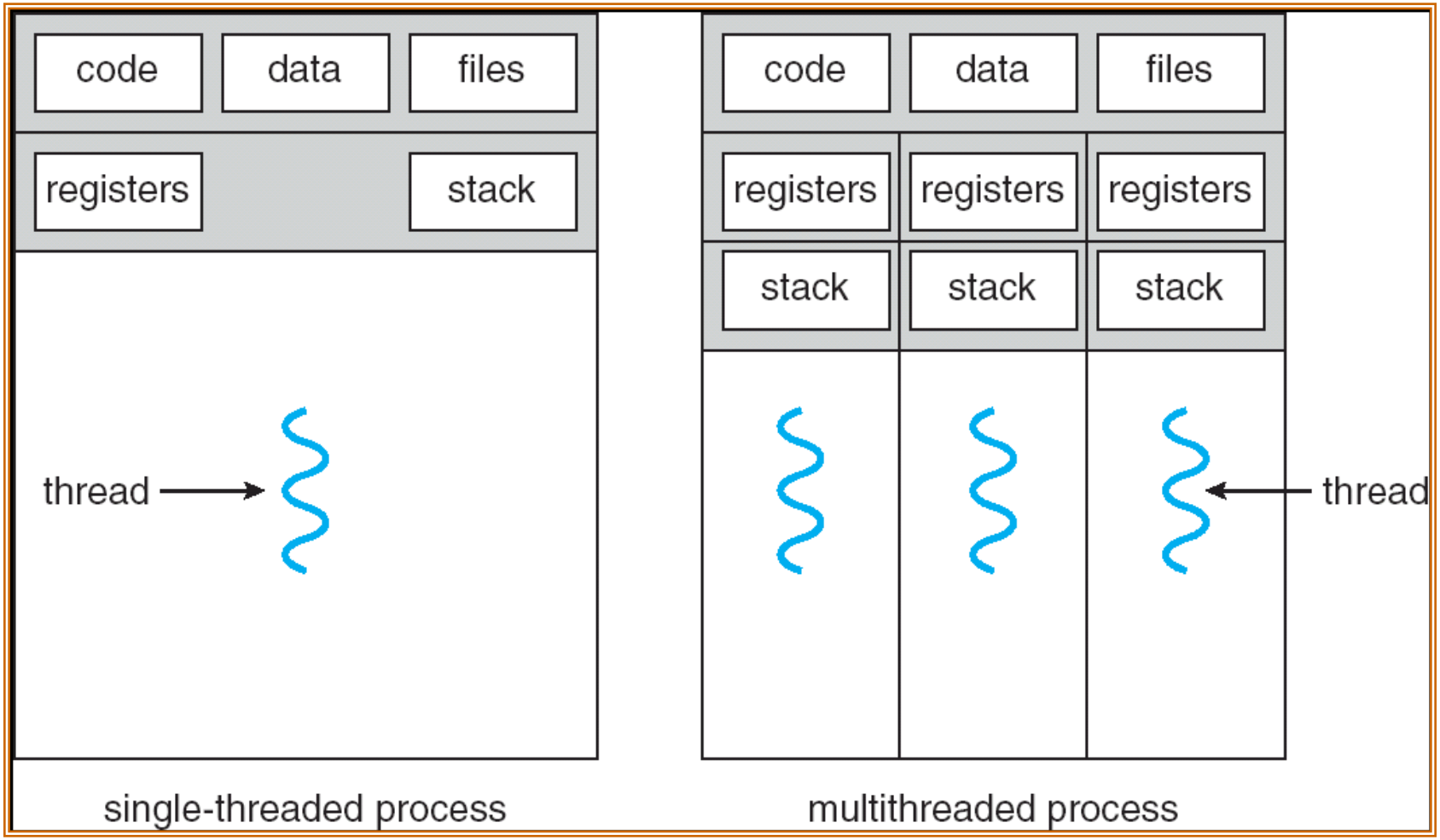
Systèmes à temps partagé de type Unix : l'algorithme d'ordonanncement est plus complexe :

- on privilégie les tâches courtes et les tâches interactives
- la priorité diminue avec le temps
- la durée du quantum alloué peut elle aussi diminuer

Notion de thread

- Un thread est une subdivision d`un processus
  - Un fil de contrôle dans un processus
- Les différents threads d'un processus partagent l'espace adressable et les ressources d'un processus
  - lorsqu'un thread modifie une variable (non locale à lui), tous les autres threads voient la modification
  - un fichier ouvert par un thread est accessible aux autres threads (du même processus)

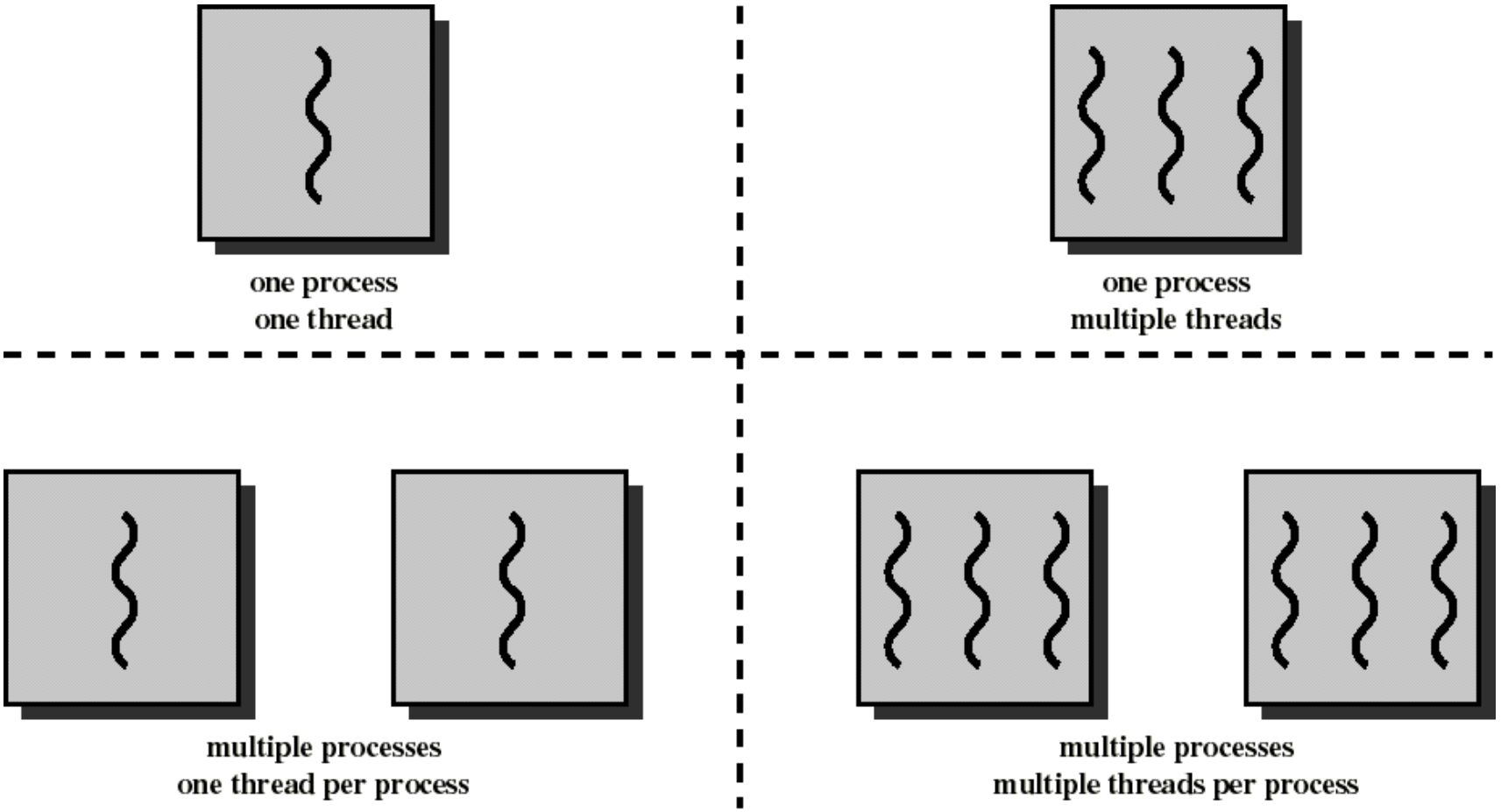
Processus à un thread et à plusieurs threads



Exemple

- Le processus MS-Word peut impliquer plusieurs threads:
  - Interaction avec le clavier
  - Rangement de caractères sur la page
  - Sauvegarde régulière du travail fait
  - Contrôle orthographe
  - Etc.
- Ces threads partagent tous le même fichier et données

Treads et Processus



Processus

- Possède sa mémoire, ses fichiers, ses ressources, etc.
- Accès protégé à la mémoire, fichiers, ressources d’autres processus

Thread

- Possède un état d’exécution (prêt, bloqué…)
- Possède sa pile et un espace privé pour les variables locales
- A accès à l’espace adressable, fichiers et ressources du processus auquel il appartient, en commun avec les autres threads du même processus

Pourquoi les threads

- Réactivité: un processus peut être subdivisé en plusieurs threads, par exemple l'un dédié à l'interaction avec les usagers, l'autre dédié à traiter des données

  L'un peut s'exécuter pendant que l'autre est bloqué

- Utilisation de machines multiprocesseurs: les threads peuvent s'exécuter en parallèle sur des UC différentes

La commutation entre threads est moins coûteuse que la commutation entre processus

- Un processus possède mémoire, fichiers, autres ressources
- Passer d'un processus à un autre implique de sauvegarder et rétablir l'état de chacun
- Passer d'un thread à un autre *dans le même processus* est bien plus simple, implique de sauvegarder seulement :
  - les registres de l'UCT,
  - la pile du thread,
  - les variabless locales du thread

Autres intérêts des threads

- La création et terminaison de nouveaux threads dans un processus existant sont aussi moins coûteuses que la création ou terminaison d’un processus
- La communication entre threads est plus simple qu’entre processus : mémoire partagée entre les threads