Les mécanismes d’exécution

Processus, ordonnancement

Université Grenoble-Alpes
Licence MIASHS

2022-2023

Introduction

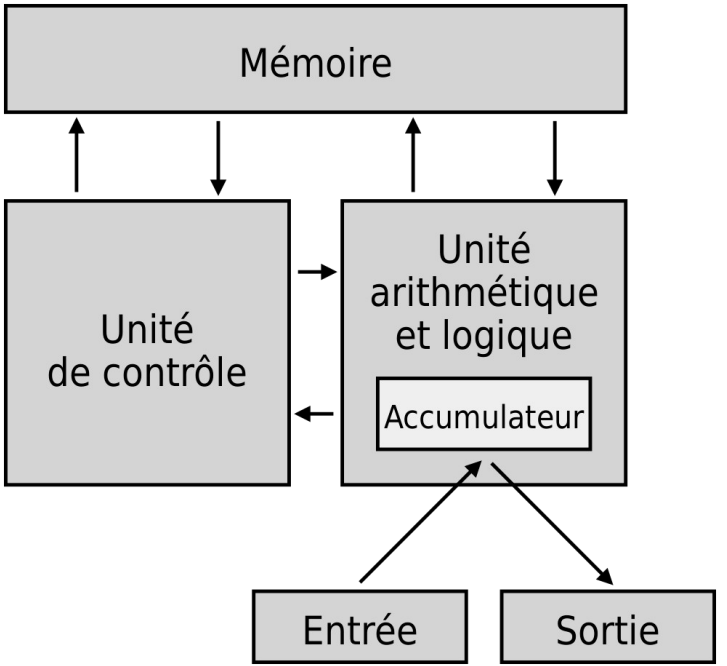
- La notion de processus est centrale
  - C’est une abstraction d’un programme en cours d’exécution
  - Elle permet de simuler des opérations concurrentes même si il n’y a qu’un seul CPU
  - Elle fournit une vue virtuelle de la mémoire, des CPU (plusieurs CPU virtuels avec un seul CPU physique)
  - Elle permet de donner « l’illusion » que plusieurs programmes s’exécutent en même temps (la multiprogrammation)

Principes de la multiprogrammation

- Il n’y a qu’un seul processus en cours d’exécution à la fois (si 1 seul processeur)
  - Un processus n’est pas exécuté en une seule fois
  - On doit être capable d’enregistrer l’état d’un processus pour y revenir plus tard
  - Il faut des stratégies pour répartir l’allocation du processeur aux processus

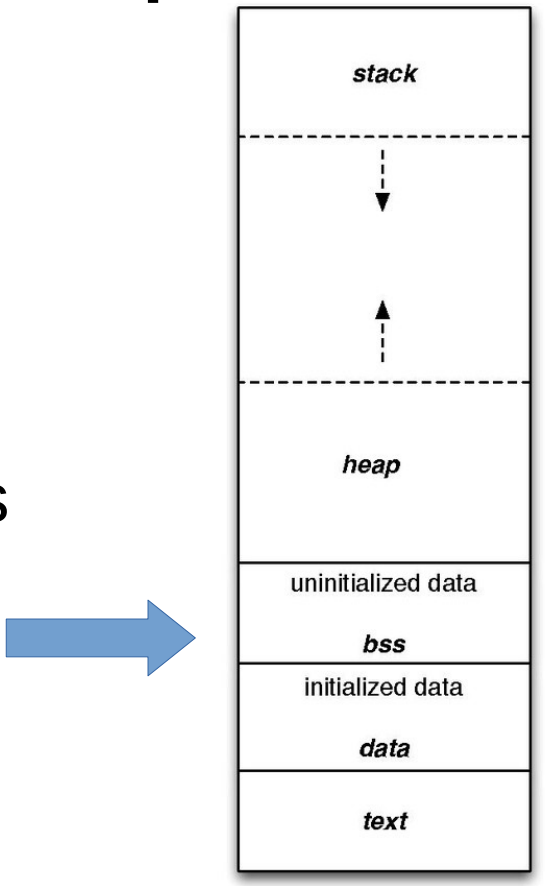
Modèle de processeur

- Instructions = opérations
  - Accès à la mémoire, opérations arithmétiques & logiques, contrôle (branchements)
- Registres (mémoire interne du processeur)
  - Accumulateur : pour stocker le résultat
  - **Registre d'état (PSW)** : zéro, retenue, dépassement, signe, etc.
  - Registre d'instruction (IR) : instruction en cours
  - **Compteur ordinal (PC)** : adresse de la prochaine instruction à charger
  - Pointeur de pile (SP) : prochain emplacement dispo dans la pile mémoire



Le modèle de processus

- Chaque processus possède un état
  - La valeur du compteur ordinal (PC)
  - Le contenu des registres
- Et une image mémoire (segment de données)



La pile : stockage temporaire pour les param de fonction, retour, variables locales

Le tas : mémoire allouée dynamiquement

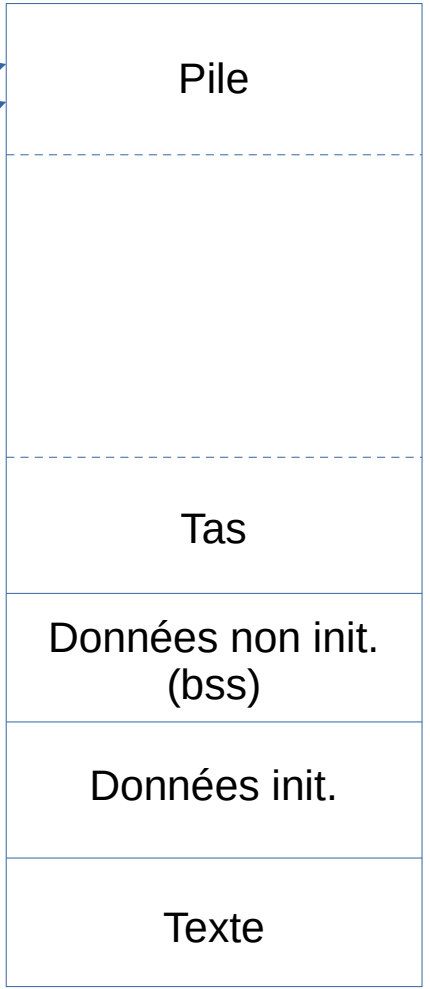
Les variables globales

Code exécutable

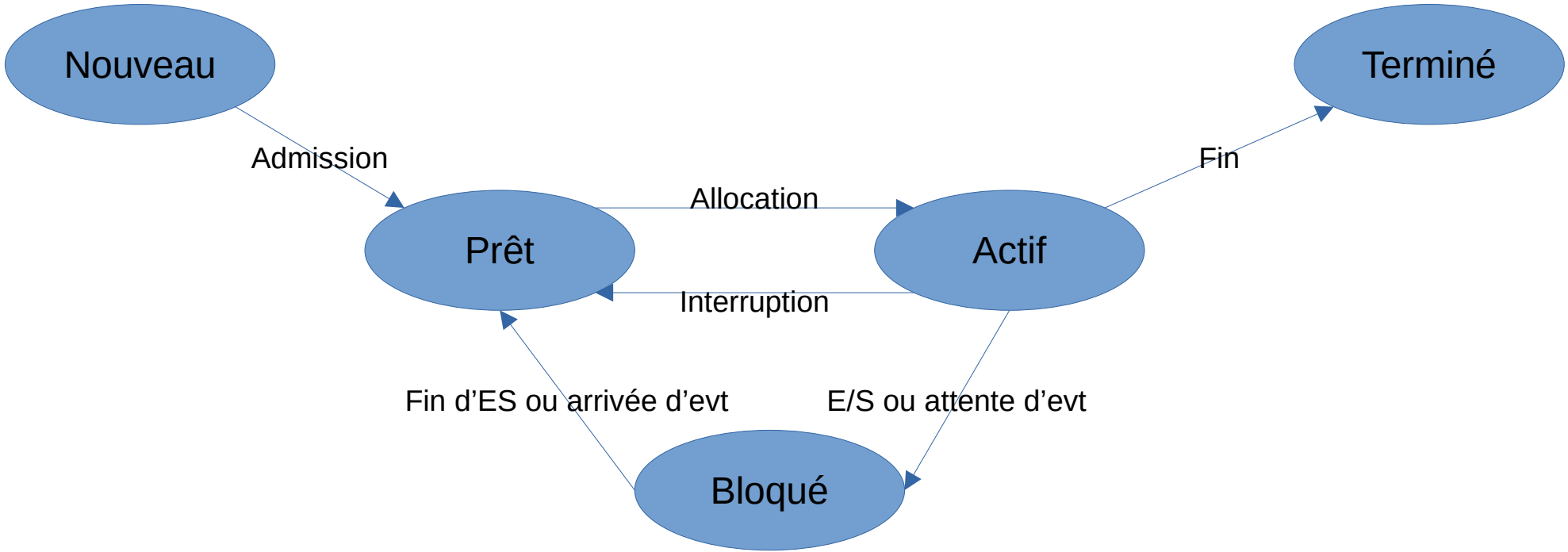
https://commons.wikimedia.org/wiki/File:Program_memory_layout.pdf

Exemple

```c
#include <stdio.h>
#include <stdlib.h>

int nb=10;
int res;

int main(int argc, char* argv[]) {
    // un pointeur vers un tableau d'entier
    int* tab;
    int i;

    // allocation d'un tableau de nb entiers
    tab = (int*) malloc(sizeof(int)*nb);

    for (i=0 ; i<nb ; i++) {
        tab[i]=i;
    }
    return 0;
}
```



Le cycle de vie d’un processus
- ses états -



Bloc de contrôle de processus

- Contient toutes les infos permettant de démarrer ou redémarrer un processus
  - sauvegardé à chaque passage « actif » vers « prêt » ou « bloqué »
  - restauré au passage « prêt » vers « actif »

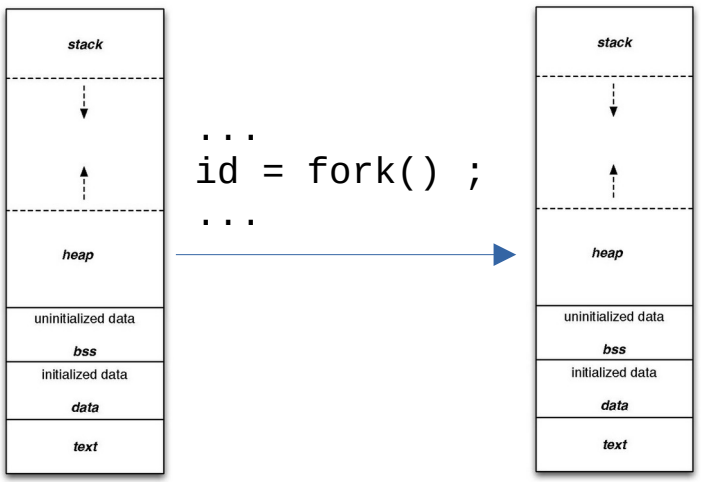
- Process bloc control », PCB :
  - L'identifiant du processus (PID), du parent (PPID), de l'utilisateur (UID)
  - Les valeurs des registres processeur
  - Le compteur ordinal (PC)
  - Le pointeur de pile (SP)
  - L'espace d'adressage (mémoire virtuelle)
  - La listes de descripteurs de fichiers (fichiers ouverts)
  - Information d'ordonnancement
  - … (ca dépend des systèmes)

La création des processus

- Cela revient à « lancer un programme »
  - Par clic, en tapant une commande dans le shell
  - Certains processus sont lancés au démarrage
- La création du processus est une opération système (exécutée en mode noyau)
  - Réalisée par l'appel système `fork()` sous Unix

La création des processus (cont.)

- Chaque processus est créé par un père
  - Hiérarchie de processus (vu précédemment)
- L'appel fork créé un processus fils, clone du processus père qui l'exécute
  - Même image mémoire, etc.
  - Par contre, ils ne partagent rien



Exemple de fork

```
#include <stdio.h>
#include <sys/types.h> /* pour pid_t */
#include <unistd.h> /* Pour fork, getpid, getppid, execlp */
#include <sys/wait.h>

int main() {
    pid_t pid;
    /* Créer un fils par clonage */
    pid = fork();
    // A patir d'ici j'ai deux processus
    if (pid < 0) { /* En cas d'erreur */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) {  /* Cas du fils */
        fprintf(stdout, "Je suis le fils");
        execlp("/bin/ls","ls",NULL);
    }
    else { /* cas du père */
        fprintf(stdout, "Je suis le père");
        wait(NULL); // attend que le fils ait fini
        printf("Fils terminé");
    }
    return 0;
}
```

Terminaison d’un processus

- Un processus se termine
  - lorsqu’il a exécuté sa dernière instruction
  - Par appel à la fonction système `exit(n)` ou n est le code de retour
  - La terminaison d’un processus libère toute les ressources du processus
- Le code de retour est envoyé au processus père qui le récupère via l’appel système `wait(&nb)`
  - Entre le moment où le processus est terminé et le moment où la fonction `wait()` du père est appelé, le processus est considéré comme un zombie
    - Il n’a plus d’image mémoire mais il a encore un PCB en mémoire (où est stocké) le code de retour

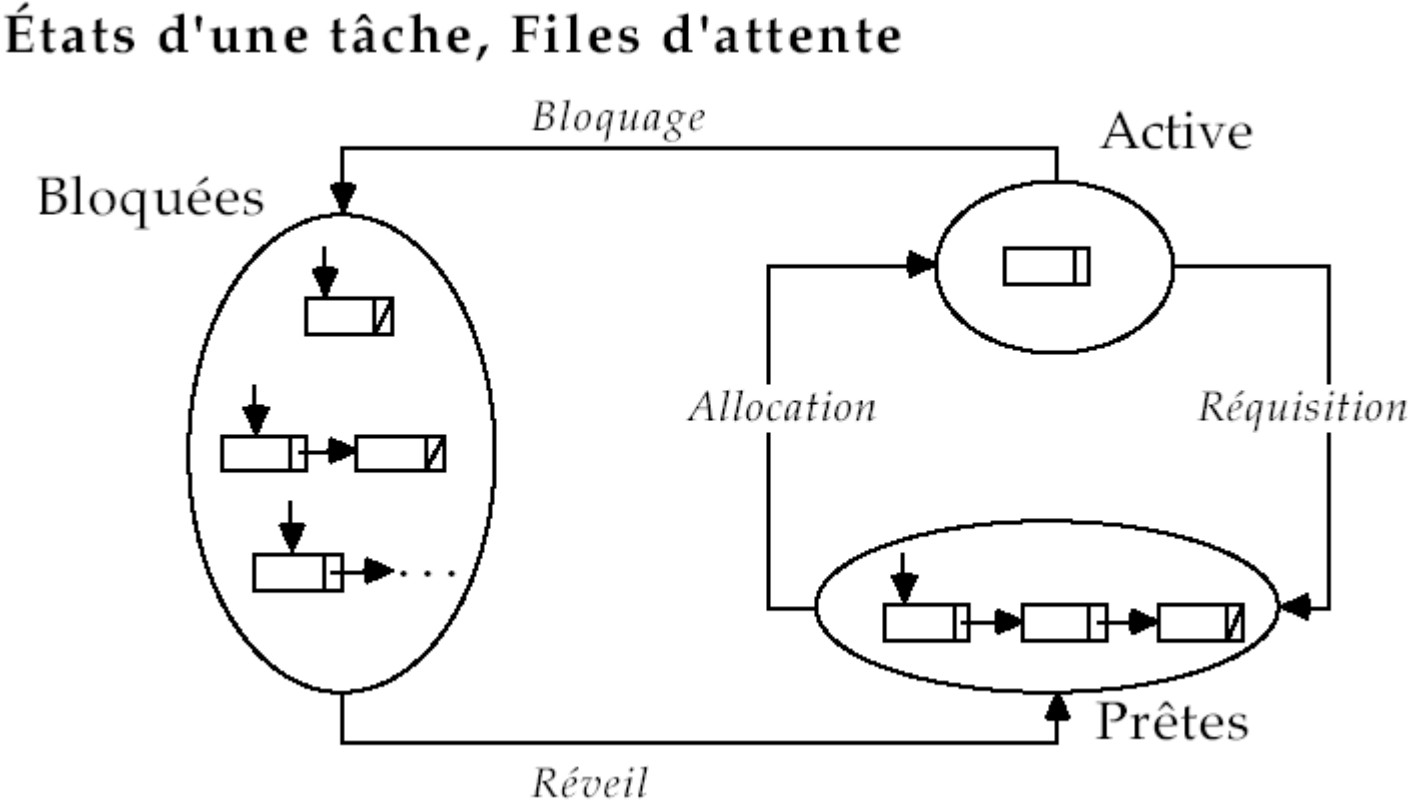
La gestion de l’exécution

- Il faut pouvoir arrêter et redémarrer un processus
  - Commutation de contexte, allocation du CPU
  - Rôle du « dispatcher »
- Il faut gérer des files d’attentes
  - Pour l’accès au CPU ou autres périphériques
- Réaliser une stratégie d’ordonnancement du CPU au processus prêts
  - Rôle du « scheduler »

Commutation de contexte

- Pour passer d’un processus à un autre
- Principe (simplifié) :
  - Sauvegarde du PCB du processus à interrompre
  - Chargement du PCB du processus à exécuter
- C’est généralement réalisé via des interruptions
  - Un mécanisme matériel du processeur
- Une commutation prend un certain temps...
  - C’est la latence de commutation (dispatcher latency)

Les files d’attentes



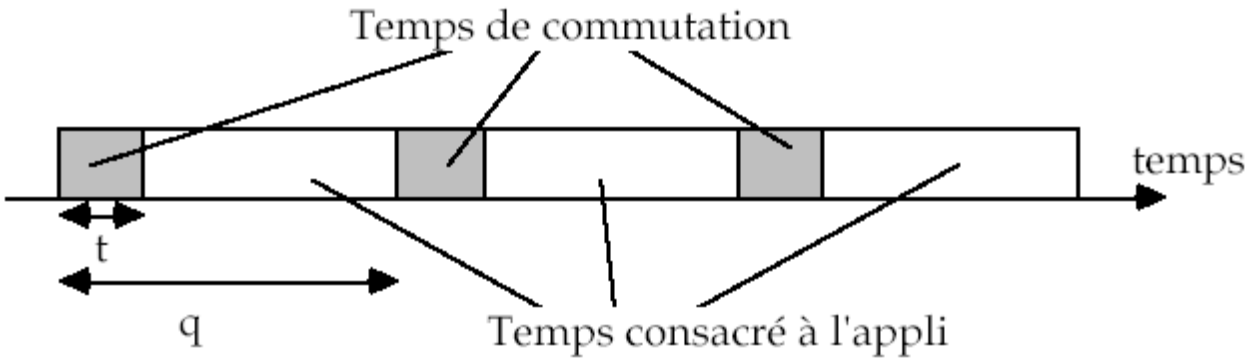
L’ordonnancement

- C’est le rôle du scheduler
  - Il doit choisir un processus parmi tous ceux sont prêts et lui allouer le CPU
- Il existe plusieurs stratégies
  - Non Préemptif
    - Laisse un processus actif tant qu’il ne bloque pas (état attente) ou qu’il n’est pas terminé (ou que le processus laisse volontairement passer son tour)
    - Utilisé dans les systèmes en mode « batch »
  - Préemptif
    - Laisse un processus dans l’état actif un temps maximun fixé.
    - Nécessite d’avoir une interruption d’horloge
    - Utilisé dans tous les systèmes d’exploitation interactifs « modernes »

Buts des algorithmes d’ordonnancement

- L’équité entre les processus
- Maximisation
  - de l’utilisation du CPU
  - du rendement (tâches terminées / unité de temps)
- Minimiser
  - Les temps d’exécution des tâches
  - Le temps d’attente (dans l’état prêt)
  - Le temps de réponse (surtout dans les systèmes interactifs)

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.

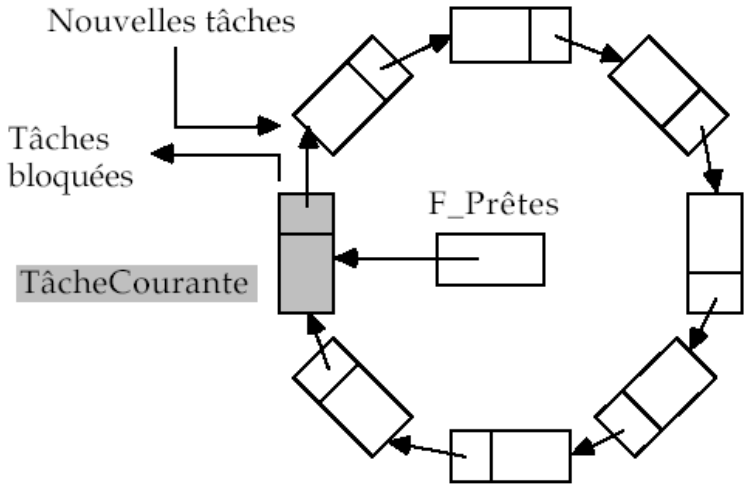
Quelques algo d’ordonnancement (non préemptifs)

- Premier arrivé, premier servi (FCFS)
  - Facile à implémenter, mais pb lorsque qu’une tâche prend trop de temps…
- Le plus court d’abord (Shortest Job First)
  - Ca serait bien mais on ne sais pas a priori le temps que va mettre un processus à s’exécuter…
  - Variante : le plus court restant.

Le Tourniquet

- Round Robin
  - FCFS en version préemptive
  - On change de processus après un certain temps
    - Quantum de 10ms à 100ms

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.**

La priorité pure

- Chaque processus est associé à une priorité
- L’ordonnanceur choisi la tâche la plus prioritaire
- PB de monopolisation du CPU
  - On décrémente la priorité à chaque interruption d’horloge
  - Le processus laissera donc sa place au bout d’un certain temps

Files d’attente multi-niveau

- Approche mixte : tourniquet + priorité

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