Systèmes d'exploitation - Processus

Pierre Gançarski

DUT Informatique - S31

ATTENTION
Ces transparents ne sont qu'un guide du cours : de nombreuses explications et illustrations manquent.

De nombreuses précisions seront données au tableau et à l'oral pendant le cours.

Université de Strasbourg

IUT Robert Schuman
UNIVERSITÉ DE STRASBOURG

Plan

Rappel

Définition

- Un processus est une *abstraction*.
- Il exécute une liste d'instructions issue d'un programme.
- Programme : entité *statique*.
- Processus : entité *dynamique*.
- Concept central de tout SE multiprogrammé (multitâche).

Structure d'un processus

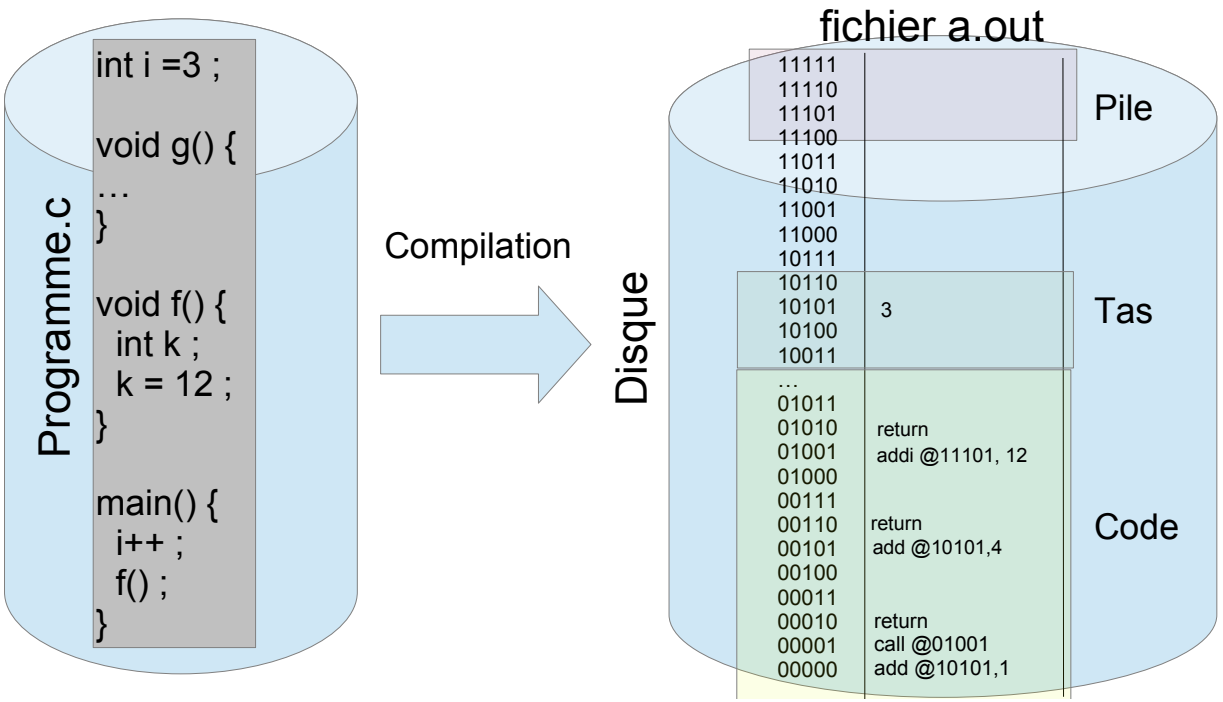
Compilation

- Pour être exécutable, un programme (fichier) doit être traduit :
  - Les instructions sont traduites en assembleur : add, call, ...
  - Les fonctions sont traduites en labels
  - les variables et labels sont traduits en adresses
  - ...
- Dans la cas d'un langage
  - *compilé* (C, Fortran...) , cette traduction est faite avant exécution : Le résultat de la traduction est stockée dans un fichier (ex : a.out)
  - *interprété* (PHP, Perl, Python...), la traduction se fait via un interpréteur qui traduit les lignes du code à la volée
  - *intermédiaire* (JAVA...) le code est préalablement traduit dans un langage intermédiaire (bytecode) proche du langage machine, permettant ainsi de préserver de bonnes performances.

Structure d'un processus

Compilation

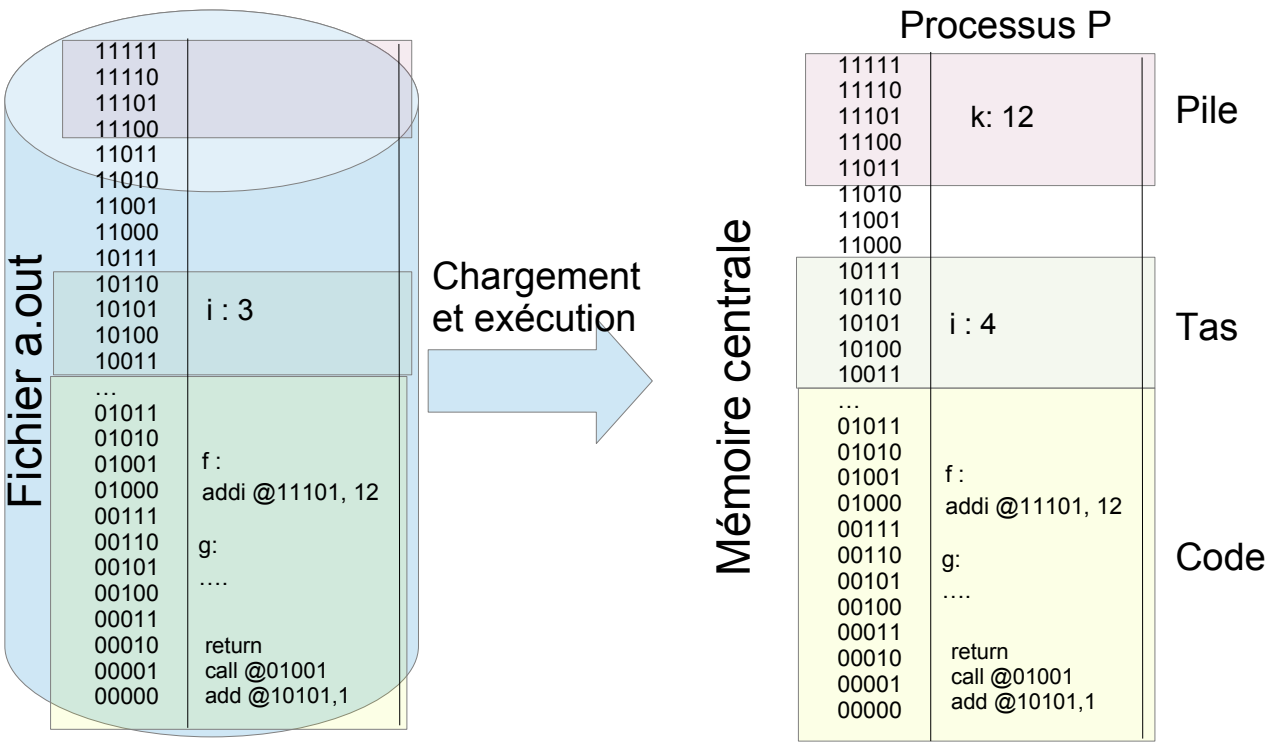
- Pour être exécutable, un programme (fichier) C doit être compilé :
  - Le compilateur va générer un fichier (ex : a.out)



Structure d'un processus

Exécution d'un processus

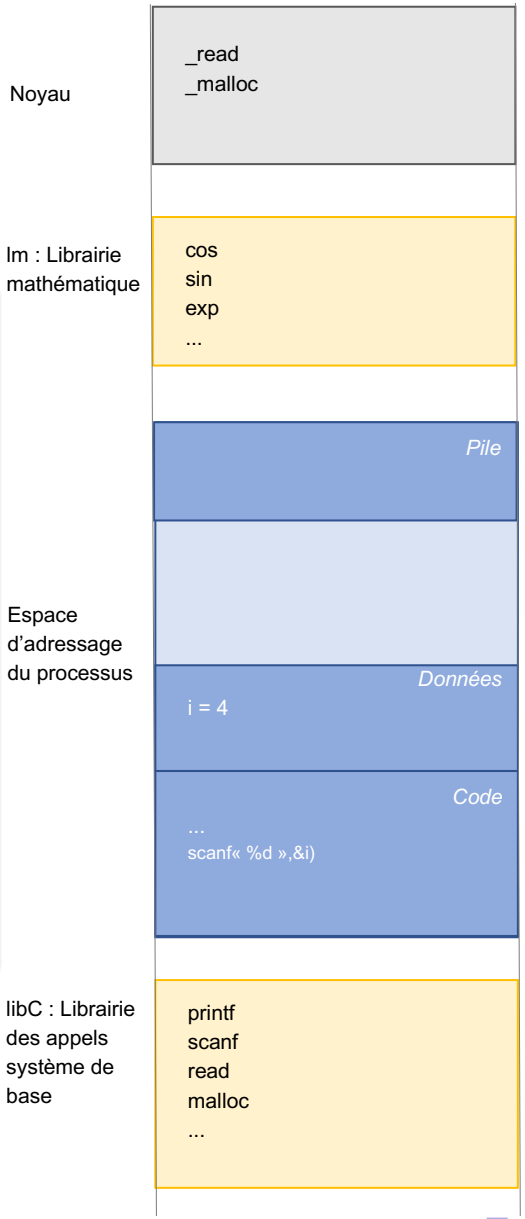
- Pour pouvoir être exécuté, un programme compilé doit être chargé en mémoire sous forme d'un processus :
  - le fichier exécutable (a.out) est chargée en mémoire
  - les instructions se déroulent une à une
    → **Rappel cours S21 :** compteur ordinal



Structure de la mémoire

Trois niveaux de mémoire

- Pour s'exécuter un processus a besoin d'un environnement :
  - Du noyau noyau : adresse physique fixe
  - De librairies : contiennent du code partagé (ou partageable)
  - D'un espace d'adressage privé

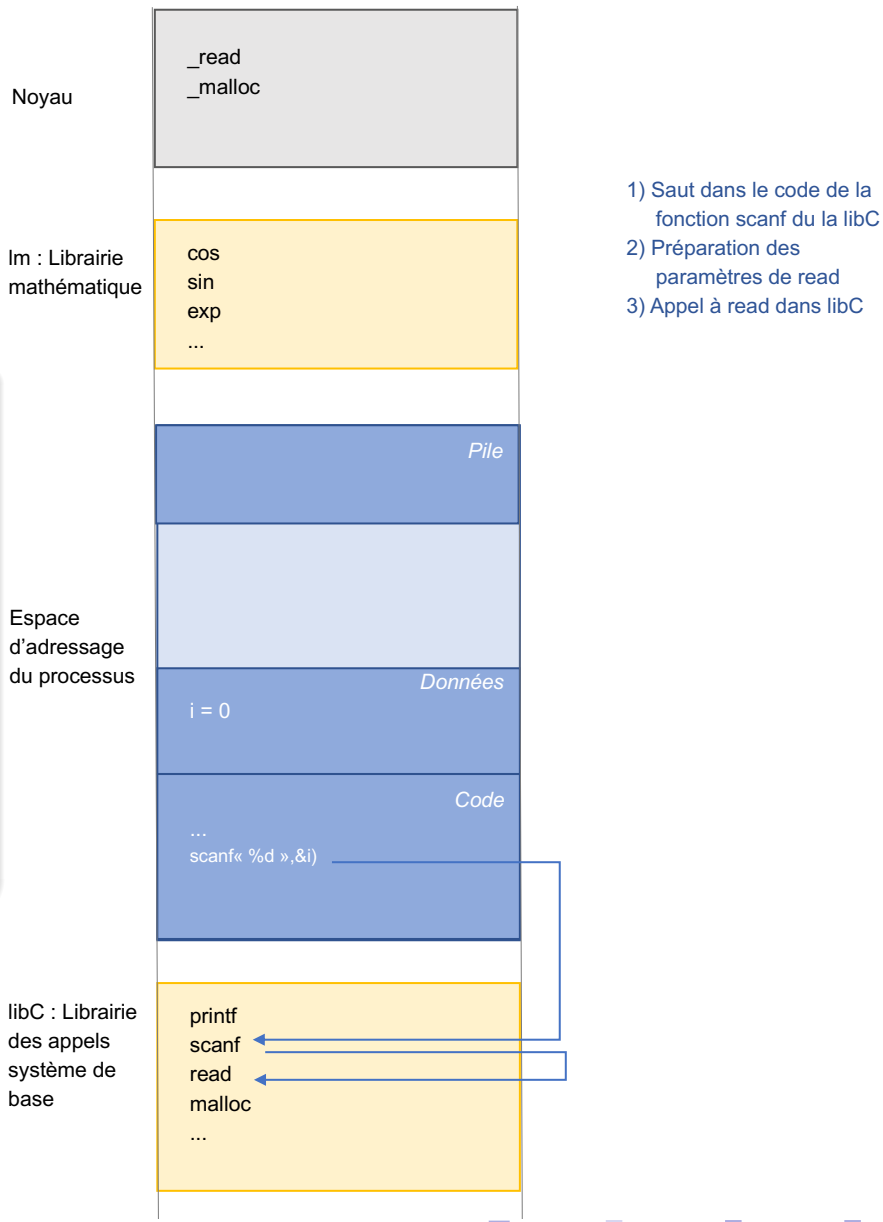


Structure de la mémoire

Trois niveaux de mémoire

- Accès à des ressources système –> *Appel système*
  - Fonction dont le code réside dans le noyau
  - Le processus passe en mode noyau

Ex : `scanf("%d",&i);`

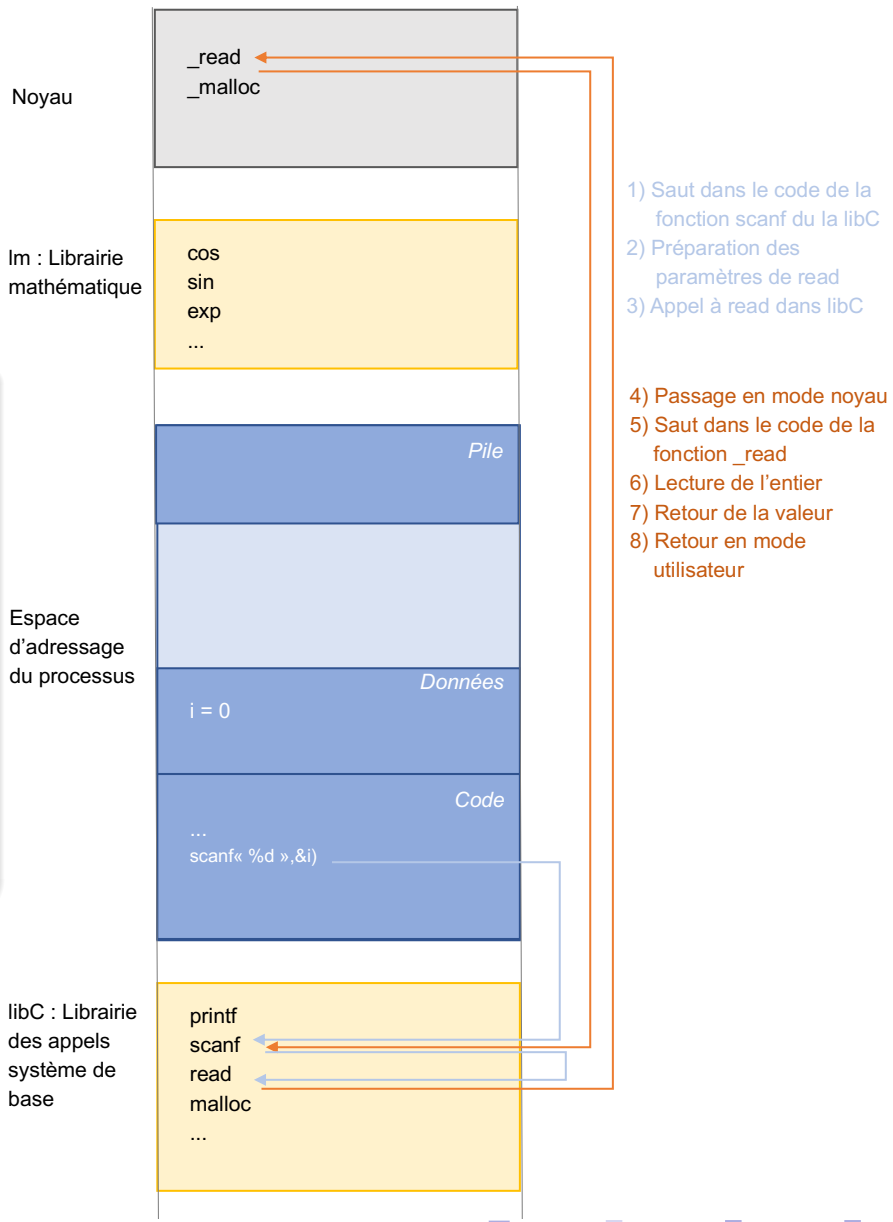


Structure de la mémoire

Trois niveaux de mémoire

- Accès à des ressources système –> *Appel système*
  - Fonction dont le code réside dans le noyau
  - Le processus passe en mode noyau

Ex : `scanf("%d",&i);`

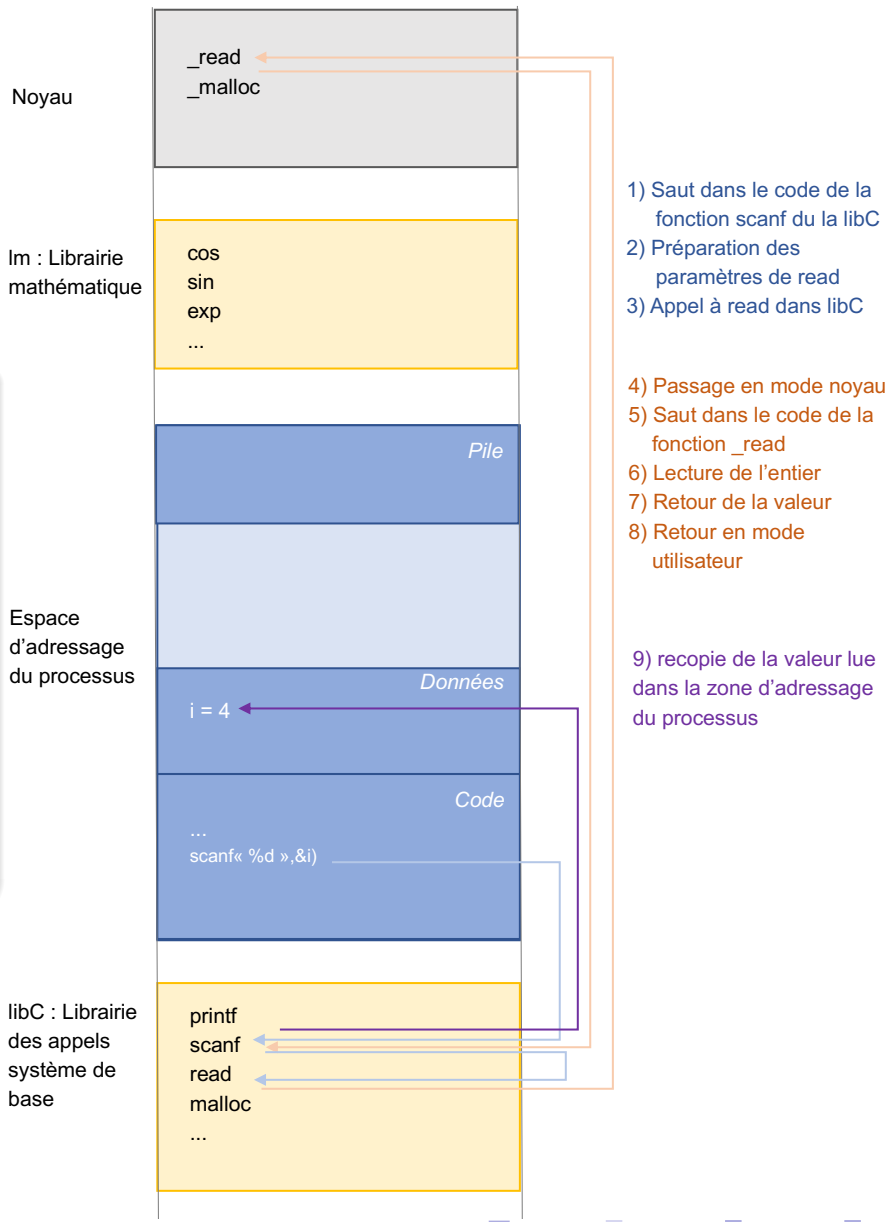


Structure de la mémoire

Trois niveaux de mémoire

- Accès à des ressources système –> *Appel système*
  - Fonction dont le code réside dans le noyau
  - Le processus passe en mode noyau

Ex : `scanf("%d",&i);`



Plan

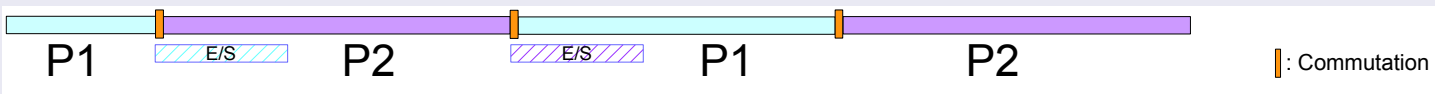
Temps partagé

Principe

- Lorsqu'un processus s'exécute sur un processeur, il peut être bloqué (lecture/écriture sur le disque, synchronisation de processus, ...)
  $\rightarrow$ le processeur ne fait rien ! !

- Idée : le processus bloqué est remplacé par un autre
  $\rightarrow$ Commutation de processus

$\Rightarrow$ Permet d'utiliser le processeur au maximum :

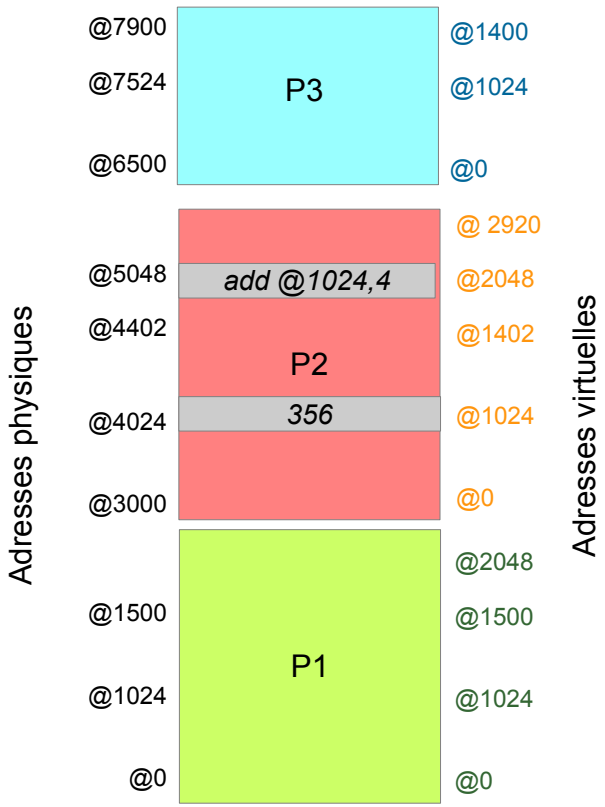


- Deux problèmes :
  - Comment charger plusieurs processus en mémoire ?
  - Comment commuter deux processus ?

Temps partagé

Gestion de la mémoire

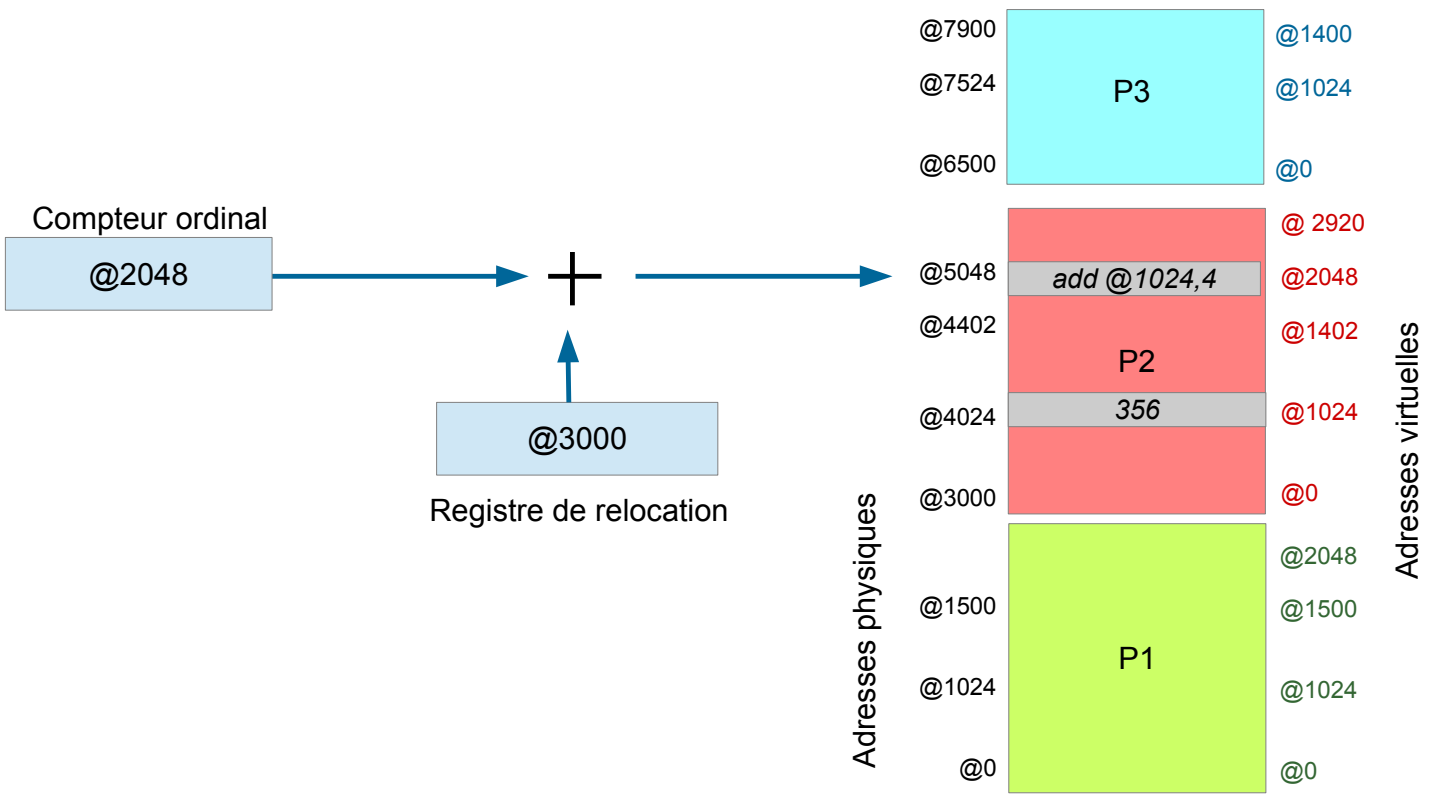
- Charge plusieurs processus en mémoire et gère leurs adresses
- Problème : Une adresse dans un processus est donnée par rapport au début du processus → adresse virtuelle $@_v$



Temps partagé

Gestion de la mémoire : traduction d'adresses

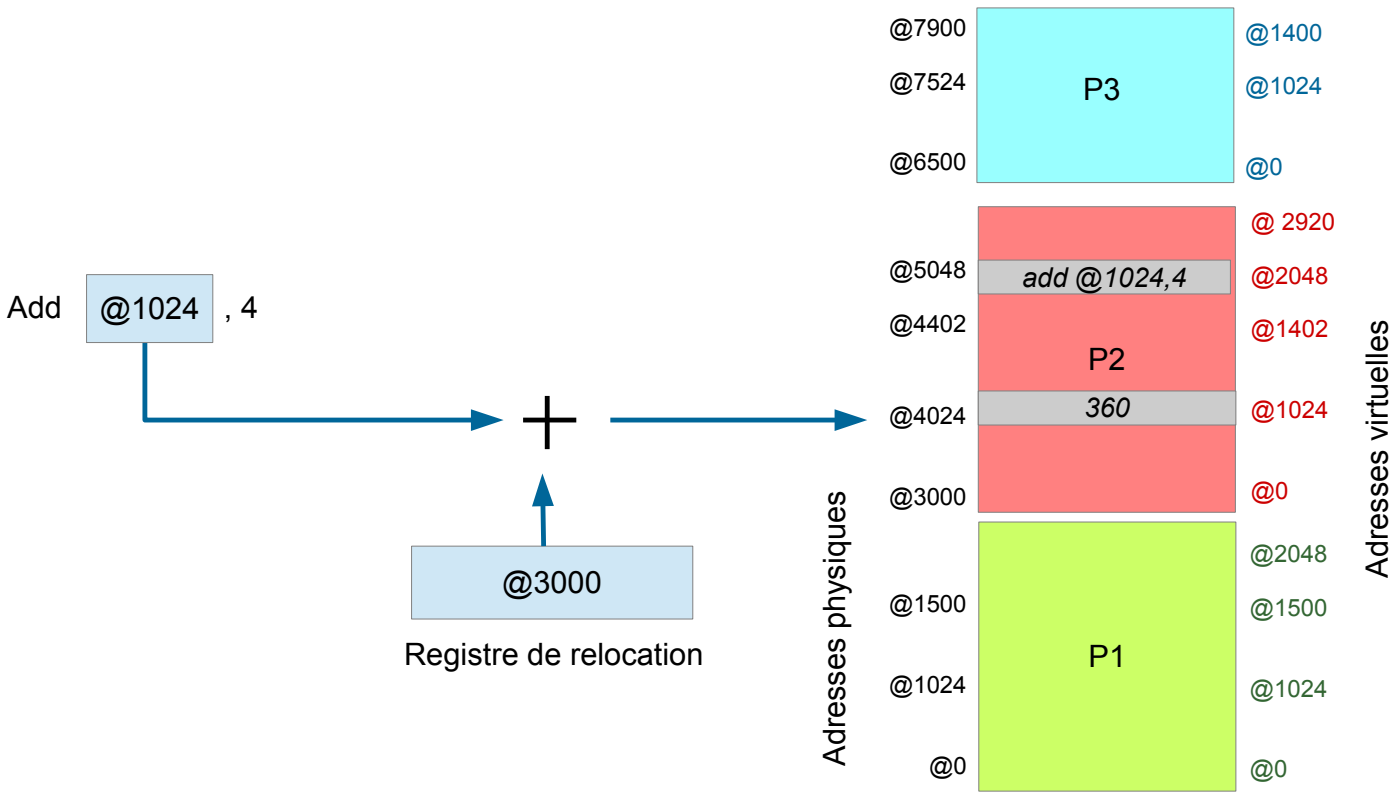
- Un registre de relocation donne l'adresse $@_r$ de début du processus
- L'adresse physique d'une adresse virtuelle $@_v$ est : $@_p = @_v + @_r$



Mémoire

Exemple : exécution de l'instruction $@_v2048$

- $@_v2048$ correspond à $@_p5048$ chargée dans le registre d'instruction
- L'adresse $@_v1024$ contenue dans l'instruction correspond à $@_p4024$ $\rightarrow$ le processeur modifie la case @4024



Exécution d'un processus

Gestion de la mémoire : adresses virtuelles vs physiques

Une adresse virtuelle DOIT être traduite en adresse physique afin de pouvoir être utilisée

- Cela ne peut se faire que à la volée pendant l'exécution de processus $\rightarrow$ une unité physique au sein de la CPU est chargée de cette traduction
- MMU : Memory Management Unit

Pour s'exécuter un processus a donc besoin :

- d'un compteur ordinal .
- des registres du processeur.
- d'une adresse de relocation
- et autres informations sur son environnement d'exécution (fichiers ouverts, ...)

Gestion des processus

Implémentation des processus

- Le système d'exploitation gère les processus au moyen d'une **table des processus**.
- Une entrée par processus (exemple) :

Contexte système	Contexte Mémoire	Contexte Utilisateur
Registres	**Adresse relocation**	Répertoire racine
Compteur ordinal	Pointeur sur le code	Répert. de travail
Mot d'état du programme (PSW)	Pointeur sur les données	Descripteurs fichiers
Etat du processus	Pointeur sur le pile	ID utilisateur
Priorité		ID groupe
Paramètres ordonnancement		
ID du processus		
Processus parent		
Groupe du processus		
Signaux		
Heure début		
Temps de traitement		
Temps de traitement du fils		
Heure de la prochaine alerte		

Gestion des processus

Exécution des processus

Supposons

- que plusieurs processus soient chargés en MC
- qu'il existe une **table des processus**.

Question : Comment ces différents processus vont-ils s'exécuter ?
Solution : Utilisation des contextes des processus et d'un ordonnanceur

Temps partagé

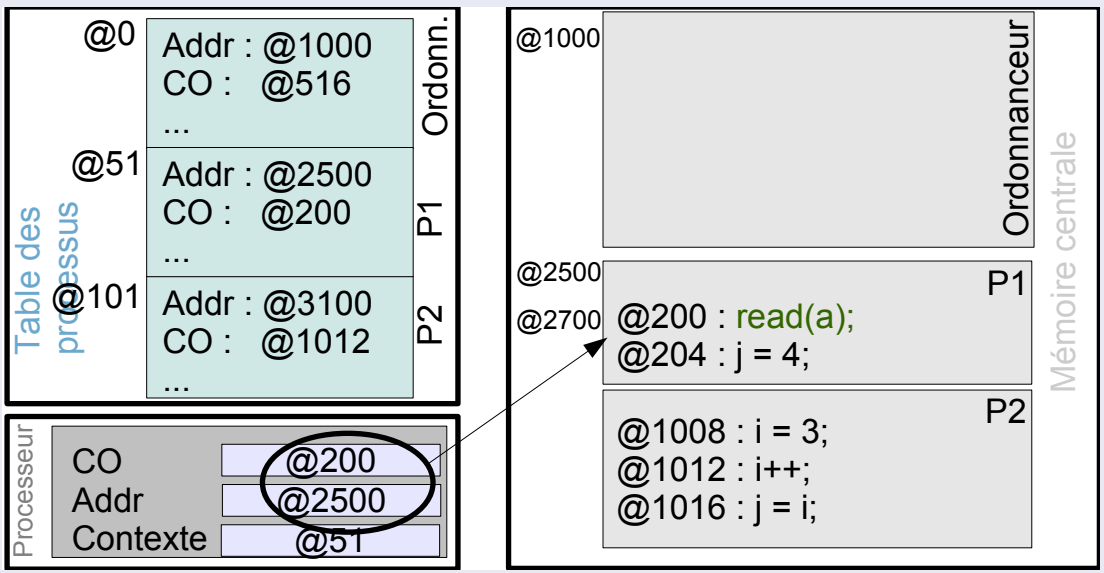
Commutation de processus

Exécution d'un instruction bloquante (`read`, `malloc`, `wait`...)

Temps partagé

Exécution d'un instruction bloquante (`read`, `malloc`, `wait...`)

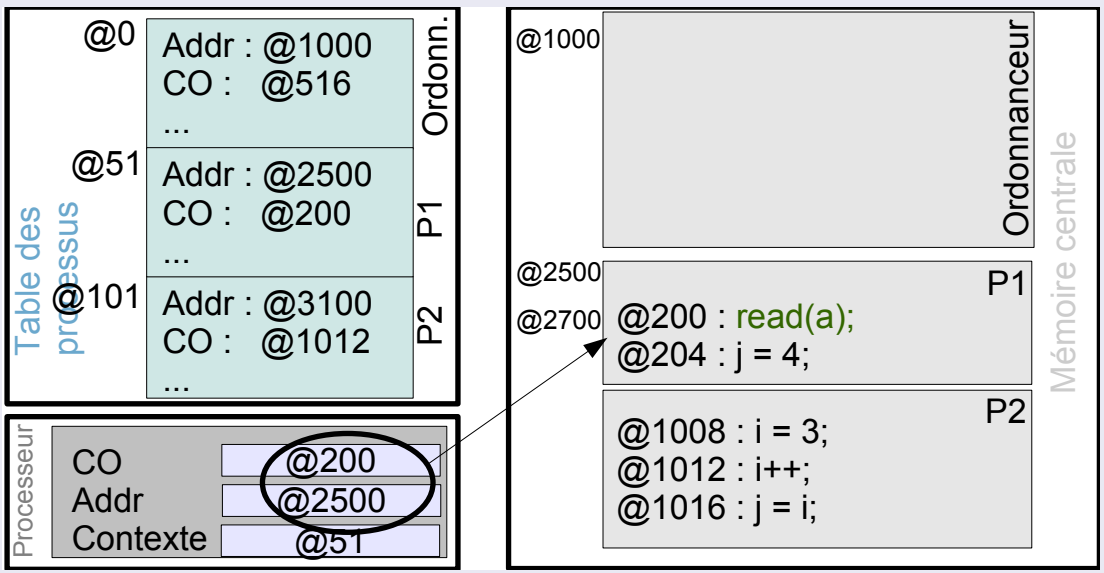
On suppose que P1 est en train de s'exécuter :
→ Son contexte est donc chargé sur le processeur. Son CO = @200



Temps partagé

Exécution d'un instruction bloquante (`read`, `malloc`, `wait...`)

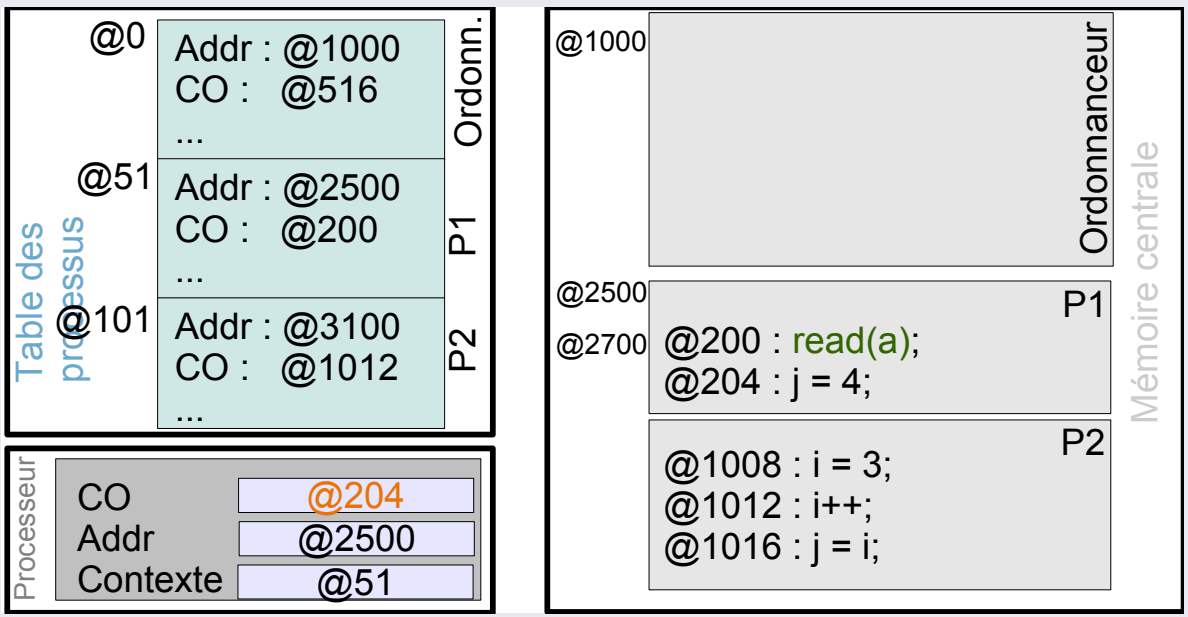
1. Exécution de l'instruction (P1) - blocage
   1.1 Lancement du `read(a)`



Temps partagé

Exécution d'un instruction bloquante (`read`, `malloc`, `wait...`)

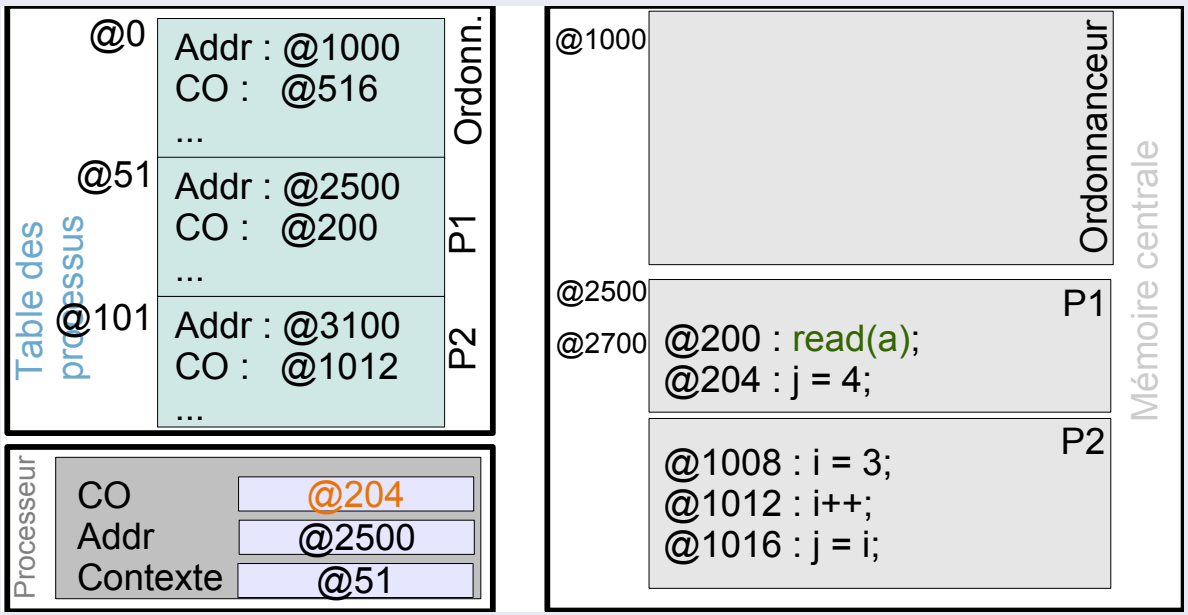
1. Exécution de l'instruction (P1) - blocage
   - 1.1 Lancement du `read(a)`
   - 1.2 Mise à jour du CO



Temps partagé

Exécution d'un instruction bloquante (`read`, `malloc`, `wait...`)

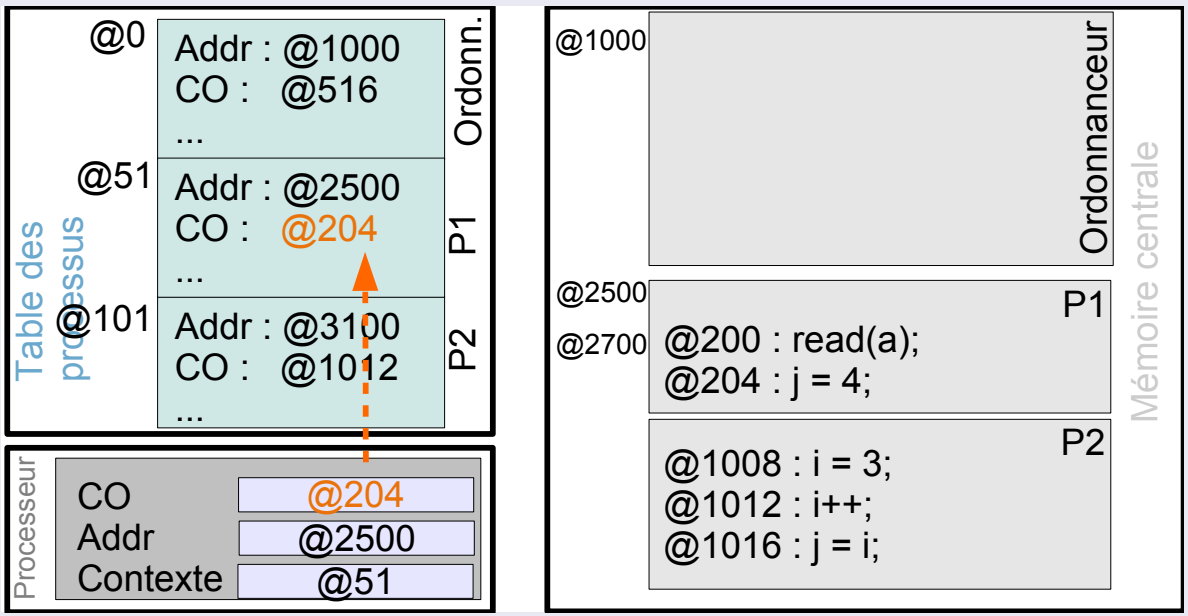
1. Exécution de l'instruction (P1) - blocage
   - 1.1 Lancement du `read(a)`
   - 1.2 Mise à jour du CO
   - 1.3 Blocage sur E/S : L'unité d'exécution émet une interruption vers le processeur



Temps partagé

Exécution d'un instruction bloquante (`read`, `malloc`, `wait...`)

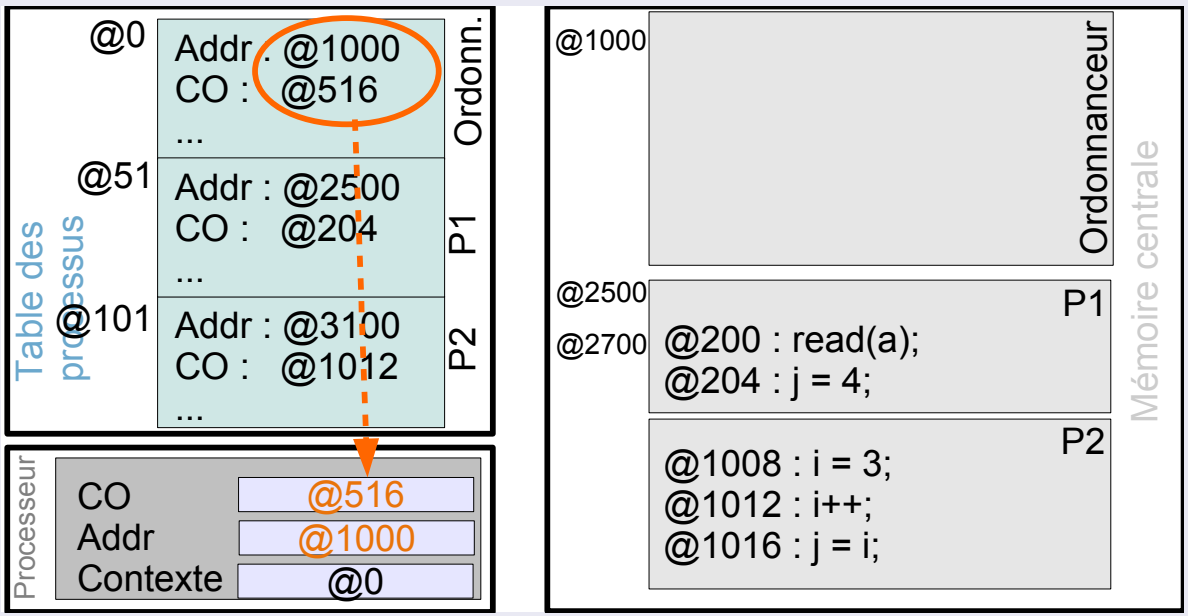
1. Exécution de l'instruction (P1) - blocage
2. Commutation de processus et ordonnancement
   - 2.1 Réception de l'interruption : le processeur sauvegarde le contexte courant



Temps partagé

Exécution d'un instruction bloquante (`read`, `malloc`, `wait...`)

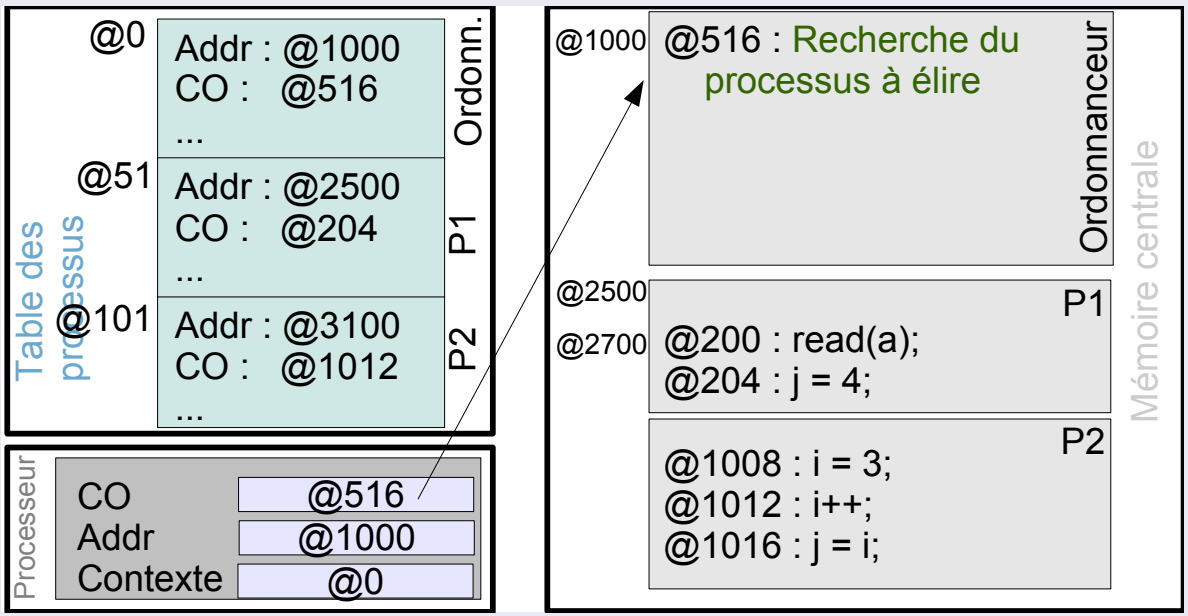
1. Exécution de l'instruction (P1) - blocage
2. Commutation de processus et ordonnancement
   - 2.1 Le processeur sauvegarde le contexte courant
   - 2.2 Le processeur charge automatiquement le contexte @0



Temps partagé

Exécution d'un instruction bloquante (`read`, `malloc`, `wait...`)

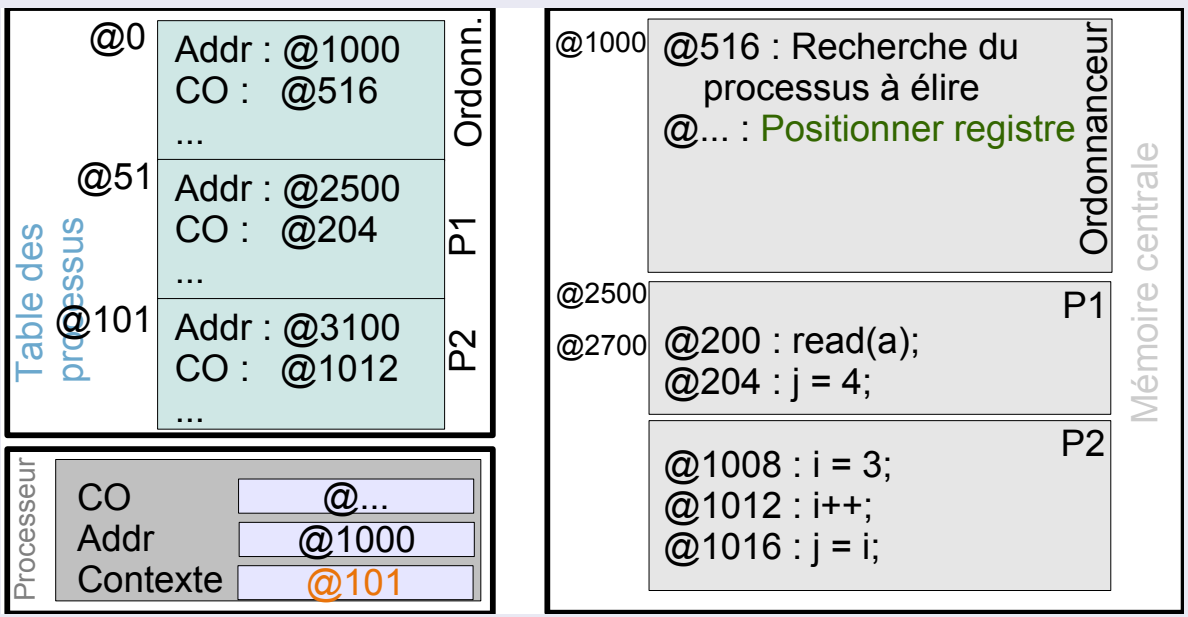
1. Exécution de l'instruction (P1) - blocage
2. Commutation de processus et ordonnancement
   - 2.1 Le processeur sauvegarde le contexte courant
   - 2.2 Le processeur charge automatiquement le contexte @0
   - 2.3 L'ordonnance définit le processus à élire (ex. P2 : contexte @101)



Temps partagé

Exécution d'un instruction bloquante (`read`, `malloc`, `wait...`)

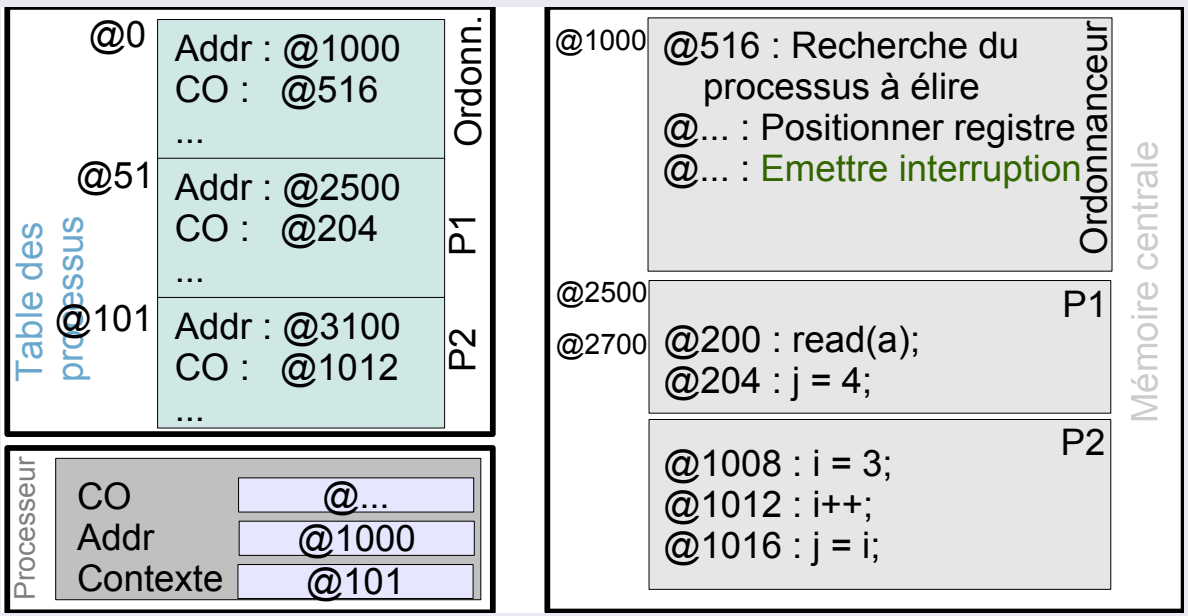
1. Exécution de l'instruction (P1) - blocage
2. Commutation de processus et ordonnancement
   - 2.1 Le processeur sauvegarde le contexte courant
   - 2.2 Le processeur charge automatiquement le contexte @0
   - 2.3 L'ordonnance définit le processus à élire (ex. P2 : contexte @101)
   - 2.4 L'ordonnanceur positionne l'adresse du contexte à charger



Temps partagé

Exécution d'un instruction bloquante (`read, malloc, wait...`)

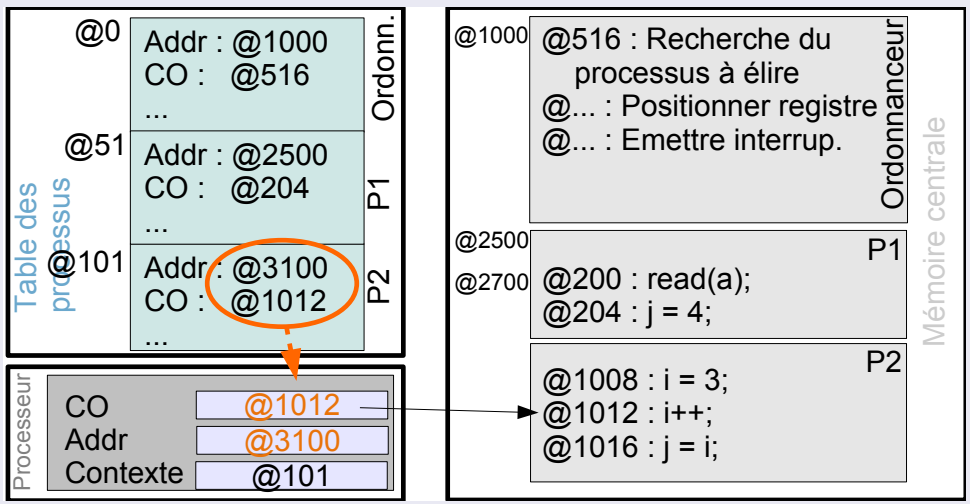
1. Exécution de l'instruction (P1) - blocage
2. Commutation de processus et ordonnancement
   - 2.1 Le processeur sauvegarde le contexte courant
   - 2.2 Le processeur charge automatiquement le contexte @0
   - 2.3 L'ordonnance définit le processus à élire (ex. P2 : contexte @101)
   - 2.4 L'ordonnanceur positionne l'adresse du contexte à charger
   - 2.5 L'ordonnanceur émet une interruption vers le processeur



Temps partagé

Exécution d'un instruction bloquante (`read`, `malloc`, `wait...`)

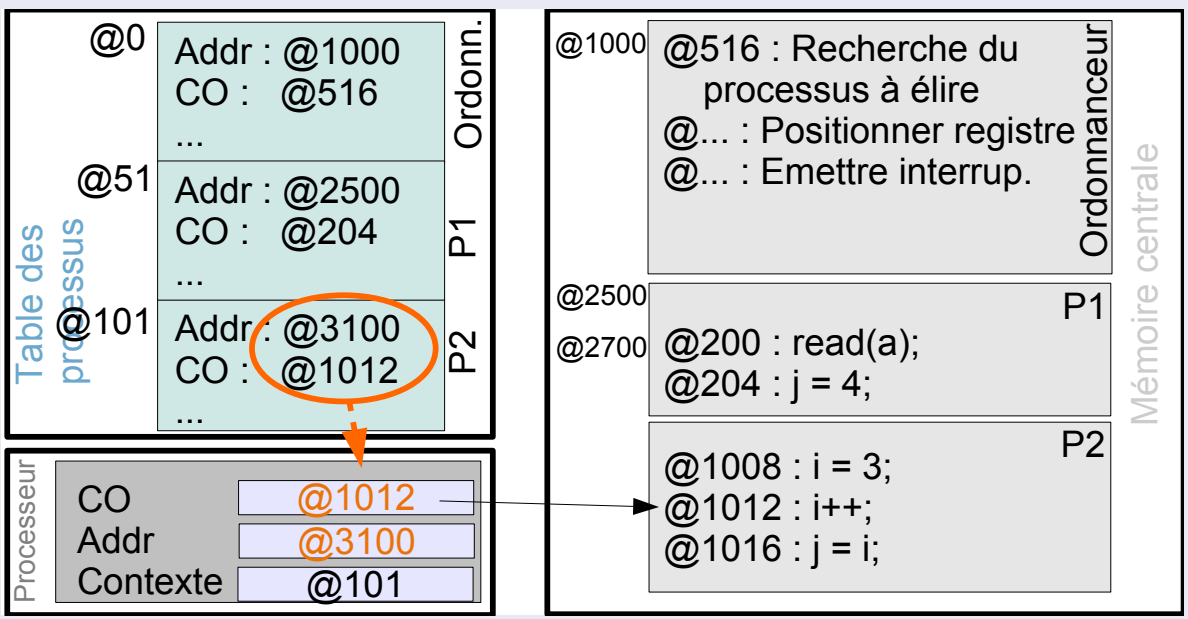
1. Exécution de l'instruction (P1) - blocage
2. Commutation de processus et ordonnancement
   - 2.1 Le processeur sauvegarde le contexte courant
   - 2.2 Le processeur charge automatiquement le contexte @0
   - 2.3 L'ordonnance définit le processus à élire (ex. P2 : contexte @101)
   - 2.4 L'ordonnanceur positionne l'adresse du contexte à charger
   - 2.5 L'ordonnanceur émet une interruption vers le processeur
   - 2.6 Le processeur charge le nouveau contexte



Temps partagé

Exécution d'un instruction bloquante (`read`, `malloc`, `wait...`)

1. Exécution de l'instruction (P1) - blocage
2. Commutation de processus et ordonnancement
3. Le processus élu (P2) (re)commence son exécution



Temps partagé

Exécution d'un instruction bloquante (`read`, `malloc`, `wait...`)

1. Exécution de l'instruction - blocage
2. Commutation avec ordonnancement
3. Exécution du processus élu

→ Lorsque l'E/S est finie, le processus est "simplement" remis dans la liste des processus prêts à s'exécuter

Problème : Un processus qui ne fait pas d'E/S monopolise le processeur

Temps partagé

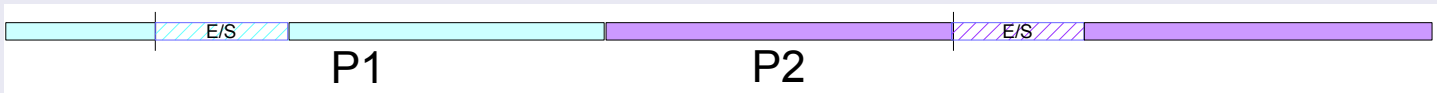
Préemption

- Pour partager équitablement le temps processeur :
  - Une horloge émet une interruption à intervalle régulier : *quantum*
  - A chacun de ces tops, le processeur agit comme pour une E/S
  - → Commutation de processus (appel à l'ordonnanceur et commutation de contextes)
- Un processus sans E/S restera au plus un **quantum** sur le processeur à chaque fois.

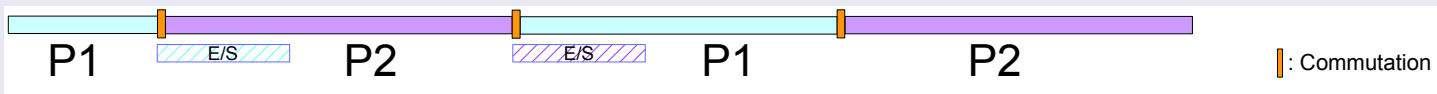
Systèmes préemptifs

Temps partagés

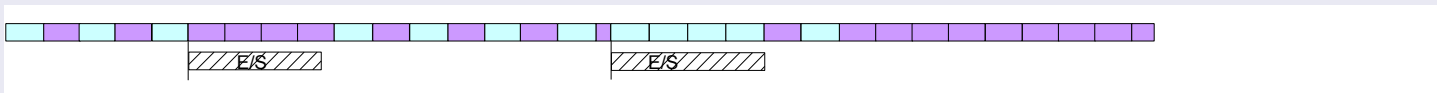
- Exécution séquentielle (un processeur) :



- Commutation sur E/S (un processeur) :



- Commutation sur E/S et quantum (pseudo-parallelisme) :



Algorithme ordonnancement

- C'est l'algorithme utilisé par l'ordonnanceur (scheduler) pour choisir parmi les processus dans l'état `prêt` celui qui doit être élu.
- Il en existe un grand nombre : dépend du système d'exploitation

Systèmes préemptifs

Durée du quantum

Supposons qu'un changement de contexte dure 1 ms.

- Le quantum est défini à Q = 4 ms
  - Le proc. passe 1/(4+1) = 20% de son temps à changer de contexte !
- Le quantum est défini à Q = 100 ms.
  - Le proc. passe 1% de son temps à changer de contexte (1/101 ms)
  - Mais les processus ne faisant pas d'E/S sont fortement favorisés
  - Risque de délai de latence : Supposons que 10 processus $P_0, ...P_9$ arrivent alors que le processus actif est à moitié de son quantum
    $\rightarrow$ P_0 attendra 50ms, P_1 : 150 ms, ... P_9 : plus d'une seconde !

$\Rightarrow$ Le quantum dépend du système (serveur ou PC) : 10 à 40 ms.

Plan

Ordonnancement : Généralités

Objectif

- Commun à tous les systèmes :
  - Equité : chaque processus doit posséder un temps processeur équitable.
  - Application de la politique : faire en sorte que la politique est bien appliquée.
  - Equilibre : faire en sorte que toutes les parties du système soient occupées.
- Évaluation :
  - Capacité de traitement : nombre de tâches terminées à l'heure (À optimiser)
  - Délai de rotation ou temps moyen d'exécution : temps moyen entre la *soumission* d'une tâche et sa terminaison :(À minimiser)
  - Taux d'occupation du CPU : (À maximiser).

Ordonnancement non préemptif

Premier arrivé - premier servi (FIFO)

- La première tâche démarre immédiatement.
- Quand une tâche arrive, elle est placée dans une file d'attente.
- Lorsque le processus actif se bloque, il est placé à la fin de la file d'attente : le premier processus de la file est exécuté.
- Algo. simple mais désavantage pour les proc. qui font bcp d'E/S.

Ordonnancement préemptifs

Tourniquet (round robin)

- L'ordonnanceur maintient une liste des processus exécutables prêts : il choisira toujours le premier de la liste
  - À sa création le processus est mis dans cette liste (généralement en bonne position)
  - Un processus qui termine son quantum est placé en queue de liste.
  - Un processus qui fait une E/S est sorti de cette liste : il y sera remis (généralement en bonne position) quand l'E/S sera finie
- Algorithme simple à implémenter.
- Très utilisé mais avec prise en compte de priorité.

Ordonnancement et mémoire

Ordonnanceurs bas et haut niveau

- Si le processus élu n'est pas en mémoire → Il doit être chargé en mémoire (swapp) → Temps de commutation bcp plus élevé :
  - Périodiquement, l'ordonnanceur de *haut niveau* retire de la mémoire des processus qui y sont restés assez longtemps
  - Il les remplace par des processus qui sont restés sur le disque assez longtemps.
  - L'ordonnanceur de *bas niveau* n'élit que des processus en mémoire.

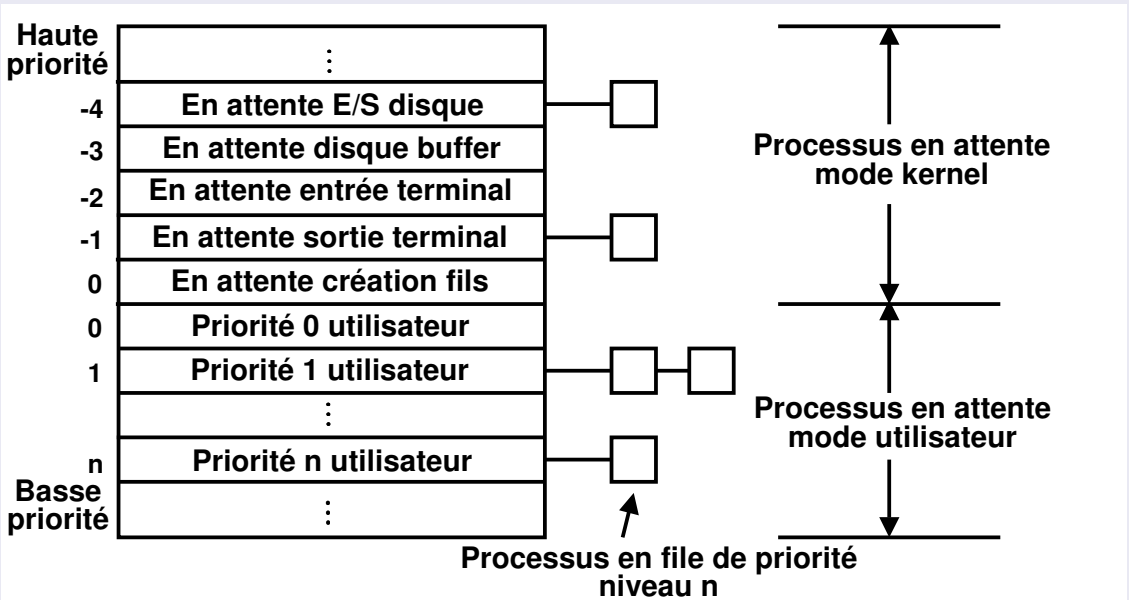
Ordonnancement multiniveau

Priorités

- Certains processus doivent être plus réactifs que d'autres.
- Chaque processus possède une priorité : le processus le plus prioritaire est choisi.
- Pour éviter que les processus prioritaires ne s'exécutent indéfiniment (situation de *famine*)
  - La priorité est recalculée dynamiquement en fonction des temps passés en E/S et du temps CPU consommé(Voir TD)
- En général trois classes (Linux) :
  - Temps réel FIFO (SCHED_FIFO) : pas de préemption sauf pour un autre processus de la classe SCHED_FIFO de priorité plus élevée.
  - Temps réel (SCHED_RR) : tourniquet et priorité dynamique
  - Normal avec priorité noyau/utilisateur (SCHED_OTHER) et différents niveaux (files d'attentes multiples)

Ordonnancement multiniveau

Cas de UNIX - Files d'attente multiples



Ordonnancement multiniveau

Cas de Windows

- 32 niveaux de priorité, divisés en deux classes
  - Temps réel (16-31) : priorité fixe, chaque niveau géré par un tourniquet
  - Variable (0-15) : les processus migrent d'un niveau à l'autre en fonction de la consommation de leur quantum, et du type d'E/S qu'ils effectuent → favorise les processus interactifs (E/S clavier, souris, écran...)
- Sur un système à N processeurs
  - Les N-1 processus les plus prioritaires ont chacun un processeur
  - Tous les autres processus se partagent le dernier processeur