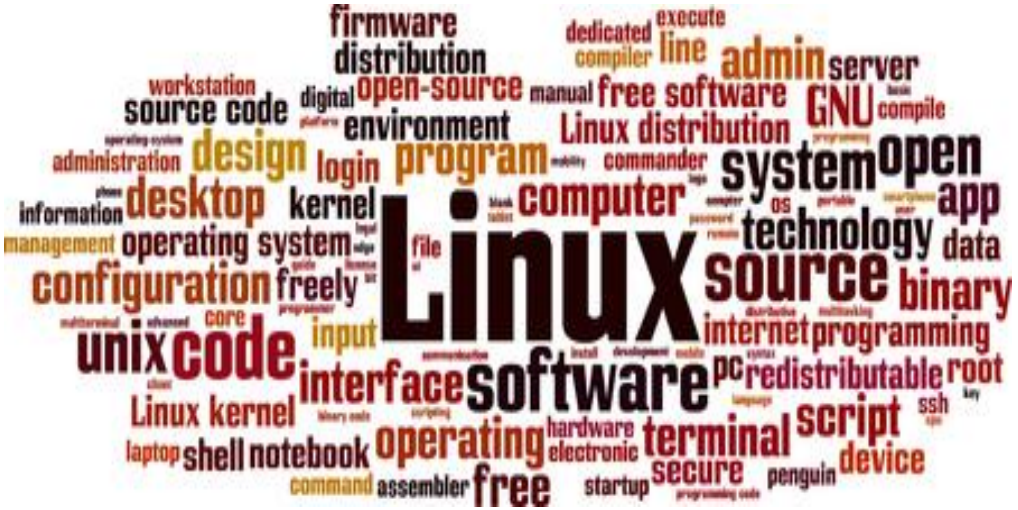


La gestion des processus

Génie Informatique, ENSAH
Y. EL MORABIT

Plan

- Introduction
- Processus
- Ordonnancement des processus
  - Ordonnancement monoprocesseur
    - Ordonnancement collaboratif
    - Ordonnancement préemptif
  - Ordonnancement multiprocesseurs
- Communication interprocessus
  - Synchronisation des processus
- Gestion des processus sous Unix

Introduction

- Les premiers SE datent des années 60-70 étaient simples, au point qu'ils ne peuvent pas exécuter plusieurs programmes en même temps.
  - Impossible de démarrer à la fois un navigateur internet, tout en écoutant de la musique.
  - Ne permettent de démarrer qu'un seul programme à la fois.
- SE **mono-programmés** ou **mono-tâche**
- Problème: Lors de l'accès à un périphérique, le processeur reste inutilisé, il doit attendre la fin de la communication avec le périphérique
- Solution: exécuter un autre programme pendant que le programme principal accède aux périphériques
- Un SE qui permet cela est un SE **multiprogrammé**

Introduction

- Les SE actuels vont plus loin et permettent d'exécuter plusieurs programmes "simultanément".
- Les SE de ce genre sont des **SE multitâches**.
  - En réalité ce n'est pas vrai, une machine monoprocesseur ne peut exécuter qu'une seule tâche à un instant donné
  - Le processeur est affecté, périodiquement, à chaque programme en cours d'exécution une durée infiniment petite, donne l'impression d'exécuter tous les programmes en même temps
- Comment ces SE gèrent la mémoire et les commutations entre programmes (appelés processus sur ces SE).

Processus

- Un processus est un programme en cours d'exécution qui a son propre environnement
  - Référencé par un identifiant unique PID
  - Sous Unix:
    - Commande ps pour déterminer les processus en cours d'exécution
    - Synopsis: ps [elf]
      - L'option -e permet d'afficher une liste de tout les processus en cours d'éxecution
      - L'option -l demande un affichage d'informations plus détaillées
      - L'option -f affiche la sous-liste des processus démarrés par vous
- Un programme peut être exécuté par plusieurs processus

Processus

- Création des processus
  - Quatre événements sont à l'origine de la création d'un processus
    - Initialisation du système
    - Exécution d'un appel système
    - Commande utilisateur
    - Initialisation d'un job en traitement par lot

Processus

- Création des processus
  - La création d'un processus se traduit toujours par un appel système déclenché à partir de :
    - Processus utilisateur
    - Processus système
    - Processus gestionnaire de traitements par lots
  - Sous Unix : fork()
  - Sous Win32 : Create_Process

Processus

- Fin d'un processus
  - Arrêt normal
    - Exécution de toutes les instructions du programme correspondant
  - Arrêt à cause d'une erreur (volontaire)
    - Prévue dans le programme (exit si un événement survient)
  - Arrêt à cause d'une erreur fatale
    - Erreur imprévisible (division par 0)
  - Arrêt par un autre processus

Processus

- Hiérarchie des processus
  - Chaque processus a un et un seul processus parent
    - Sous Unix le processus « init » est le seul qui n'a pas de parent
      - C'est le processus qui initialise le système
  - Un processus peut avoir des processus fils
  - Sous Unix, il est possible d'envoyer des signaux à toute une arborescence de processus
    - Demande d'arrêt par exemple
  - Le principe de hiérarchie n'existe pas sous Windows
    - Tous les processus sont égaux

Processus

- Principaux appels système de gestion des processus de la norme POSIX

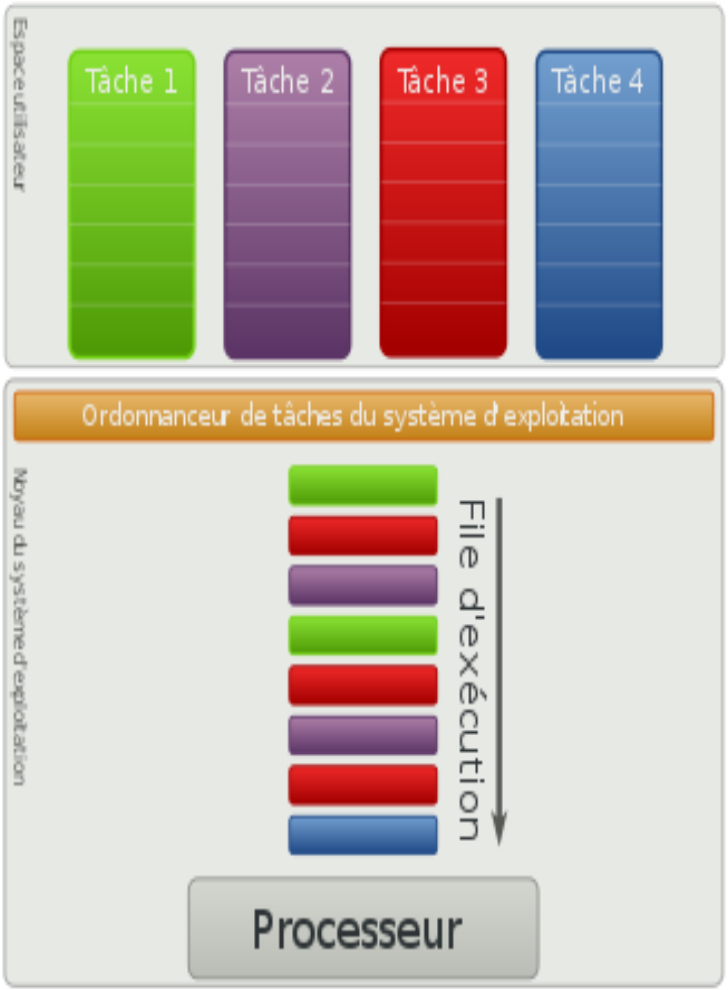
Appel système	description
pid = fork()	Crée un processus enfant identique au parent
pid = waitpid(pid, &stat, options)	Attend la terminaison d'un fils
exit (statut)	Termine l'exécution du processus et renvoie le statut
s = kill(pid, signal)	Envoie un signal à un processus

Processus

- Appels système de gestion des processus
  - Fork
    - Crée une copie conforme du processus appelant
      - Même code
      - Mêmes descripteurs de fichiers
      - …
      - Sauf la valeur retournée par fork
        - Nulle dans le nouveau processus (fils)
        - PID du fils dans le processus père
    - Les deux processus évolueront, en général, différemment
    - Le seul moyen de créer les processus

Allocation du processeur

- Le système d'exploitation utilise une technique simple pour permettre la multiprogrammation sur les ordinateurs à un seul processeur : **l'ordonnancement**
- Cela consiste à alterner entre les différents programmes à exécuter: en alternant assez vite, on peut donner l'illusion que plusieurs processus s'exécutent en même temps.

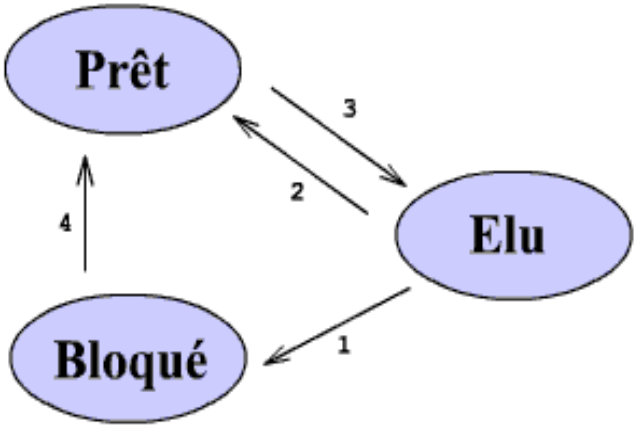


Allocation du processeur

- Avec cette technique, chaque processus peut être dans l'un des trois états :
  - Élu
    - En train de s'exécuter, c'est lui qui détient la CPU
  - Prêt
    - A besoin de s'exécuter et attend son tour pour la CPU
  - Bloqué
    - Attend une ressource ou un événement
    - Ne peut pas être candidat pour obtenir la CPU avant qu'il soit débloqué
- Les processus en état prêt sont placés dans une file d'attente, le nombre de programme dans cette liste a une limite fixée par le système d'exploitation.

Allocation du processeur

- La transition 1 rend le processus bloqué en attente d'une ressource ou d'un événement
- La transition 2 reprend la CPU au processus et la donne à un autre processus. Notre processus doit attendre son tour
- La transition 3 donne (redonne) la CPU à notre processus
- La transition 4 débloque le processus ce qui le rend candidat pour obtenir la CPU
- Un processus prêt ne peut pas devenir bloqué sans passer par l'état « élu »
- Un processus bloqué ne peut pas prendre la CPU sans passer par l'état « prêt »



Allocation du processeur

- Problématique:
  - Un ensemble de processus p1, p2, …, pn en état prêt et qui attendent le processeur et on ne dispose que d'un seul processeur

- Solution:
  - Ordonnancement des processus

Ordonnancement des processus

Ordonnancement monoprocesseur

- L'ordonnanceur peut jouer un rôle prépondérant pour améliorer les performances du système d'exploitation
  - Exemple:
    - Etant donnés deux processus P1 et P2
      - P1 : processus de rafraichissement de l'écran
      - P2 : processus d'envoi de courrier électronique
    - Retarder P1 (ralentir le rafraichissement de l'écran): l'utilisateur perçoit ce petit retard comme une dégradation du service
    - Par contre un petit retard de l'envoi du courrier électronique ne sera pas senti par l'utilisateur
    - L'ordonnanceur doit, normalement favoriser le processus P1

Ordonnancement monoprocesseur

- Les objectifs de l'ordonnanceur sont:
  - Assurer le plein usage de la CPU (agir en sorte qu'il soit le moins possible inactif)
  - Réduire le temps d'attente des utilisateurs
  - Assurer l'équité entre les utilisateurs

Ordonnancement monoprocesseur

- Types d'ordonnanceurs:
  - Ordonnancement non préemptif (sans réquisition)
    - Un processus est exécuté jusqu'à la fin sans suspension

  - Ordonnancement préemptif (avec réquisition)
    - L'exécution d'un processus est interrompue au bout d'un certain temps

Ordonnancement monoprocesseur

- Nous distinguons plusieurs algorithmes d'ordonnancement, les plus répandus sont:
  - Algorithmes d'ordonnancement non préemptif
    - First Input First Output (FIFO)
    - Shortest Job First (SJF)
    - Ordonnancement par priorités
  - Algorithmes d'ordonnancement préemptif
    - Ordonnancement de type tourniquet (Round Robin)
    - Ordonnancement par priorités

Ordonnancement monoprocesseur

- Un algorithme d'ordonnancement permet d'optimiser l'un des grandeurs temporelles suivantes:
  - Le temps d'exécution moyen
    - Décrit la moyenne des dates de fin d'exécution
    - TEM=$\sum_{i=0}^{n} TEi/n$, avec TEi=date fin-date arrivée
  - Le temps d'attente moyen
    - La moyenne des délais d'attente pour commencer une exécution
    - TAM=$\sum_{i=0}^{n} TAi/n$, avec TAi=TEi-temps d'exécution

Ordonnancement monoprocesseur

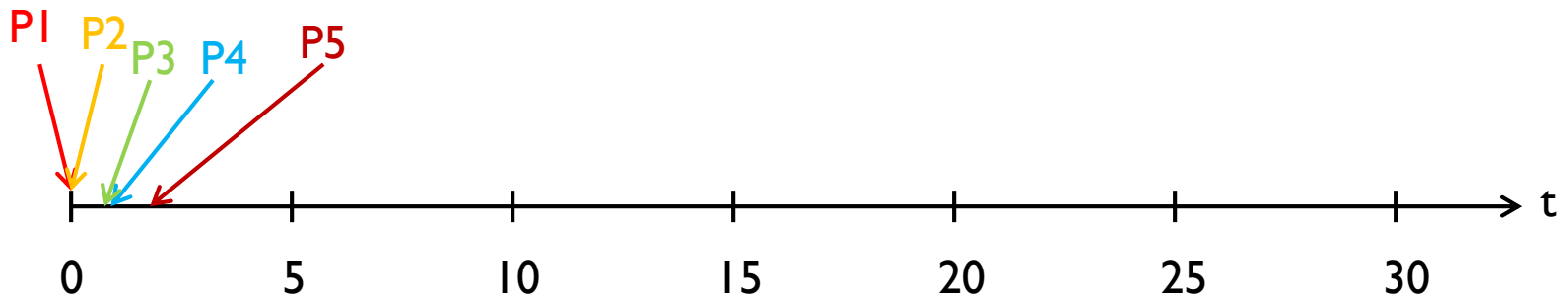
- Ordonnancement non préemptif
  - **Premier arrivé premier servi (FIFO)**
    - Les processus attendent dans une file, le premier arrivé est admis immédiatement et s'exécute tant qu'il n'est pas bloqué ou terminé. Lorsqu'il se bloque, le processus suivant commence à s'exécuter et le processus bloqué va se mettre au bout de la file d'attente.

    - Avantage
      - Algorithme facile à mettre en œuvre
    - Inconvénient
      - Les processus de faible temps d'exécution peuvent être pénalisés parce qu'un processus de longue durée les précède dans la file.

Ordonnancement monoprocesseur

- Ordonnancement non préemptif
  - **Premier arrivé premier servi (FIFO)**
    - Exemple:

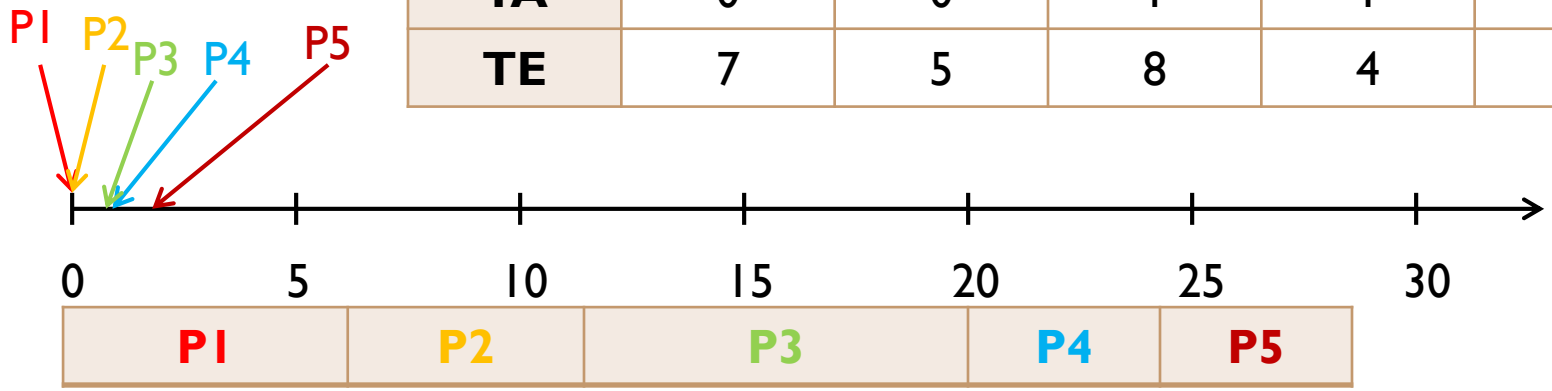
P	P1	P2	P3	P4	P5
TA	0	0	1	1	2
TE	7	5	8	4	3



Ordonnancement monoprocesseur

- **Premier arrivé premier servi (FIFO)**
  - Exemple:

P	P1	P2	P3	P4	P5
TA	0	0	1	1	2
TE	7	5	8	4	3



- $TEM=\frac{TER(P1)+TER(P2)+TER(P3)+TER(P4)+TER(P5)}{5}$
- $TER(P1) = T_F(P1) - T_A(P1) = 7 - 0 = 7$
- $TER(P2) = T_F(P2) - T_A(P2) = 12 - 0 = 12$
- $TER(P3) = T_F(P3) - T_A(P3) = 20 - 1 = 19$
- $TER(P4) = 23$
- $TER(P5) = 25$

$TEM=17,2$

Ordonnancement monoprocesseur

- **Premier arrivé premier servi (FIFO)**
  - Exemple:

P	P1	P2	P3	P4	P5
TA	0	0	1	1	2
TE	7	5	8	4	3

P2	P1	P3	P4	P5

- $TME = \frac{TER(P2)+TER(P1)+TER(P3)+TER(P4)+TER(P5)}{5}$
- $TME = 16{,}8 < 17{,}2$

Ordonnancement monoprocesseur

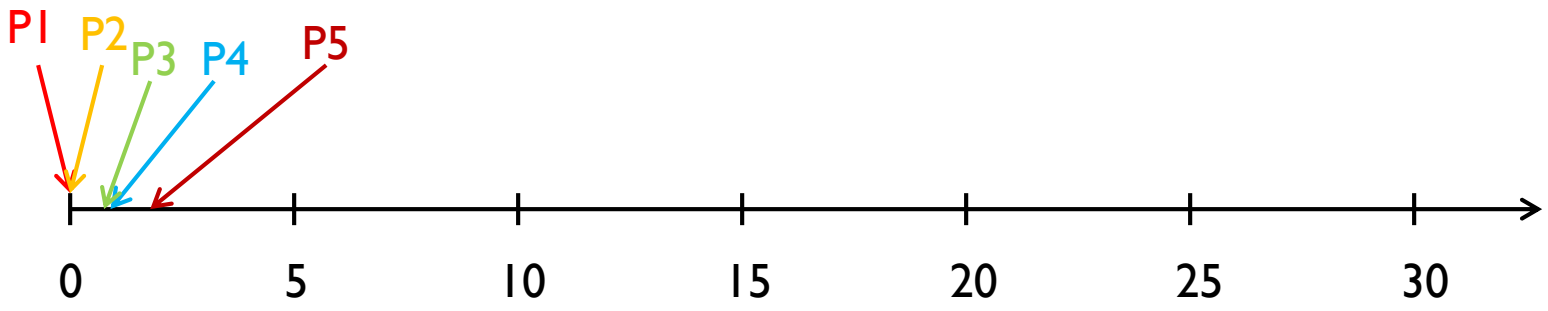
- Ordonnancement non préemptif
  - **Le plus court d'abord (SJF)**
    - Sera élu, le processus dont on suppose que le traitement sera le plus court
    - Dans le cas où plusieurs processus possède la même durée, l'algorithme FIFO sera appliqué sur ces processus

    - Avantage
      - Optimise le temps moyen d'attente des processus
    - Inconvénient
      - Si des processus courts arrivent sans cesse, les processus plus longs n'auront jamais le temps de s'exécuter (famine).

Ordonnancement monoprocesseur

- Ordonnancement non préemptif
  - **Le plus court d'abord (SJF)**
    - Exemple:

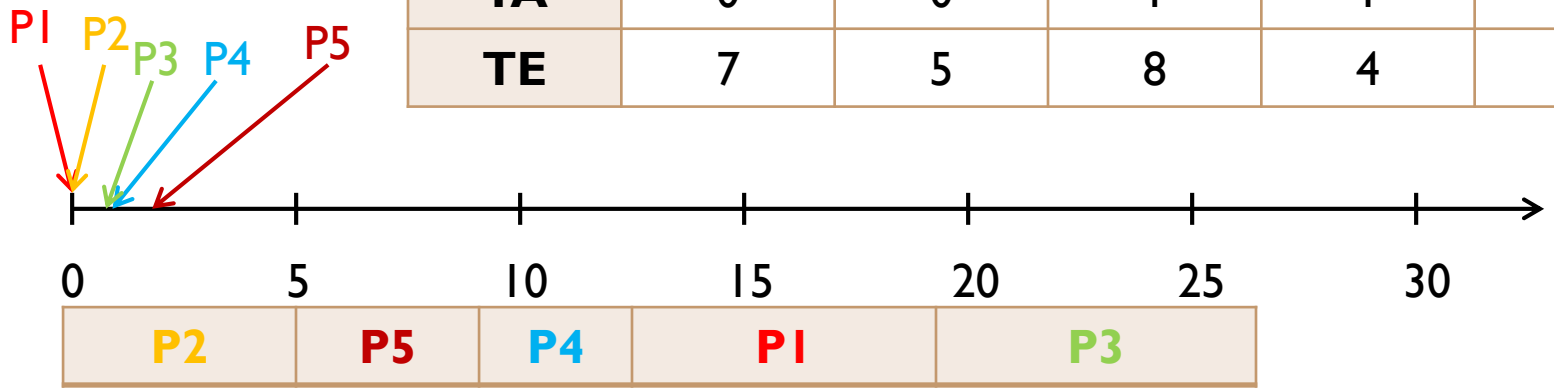
P	P1	P2	P3	P4	P5
TA	0	0	1	1	2
TE	7	5	8	4	3



Ordonnancement monoprocesseur

- **Le plus court d'abord (SJF)**
  - Exemple:

P	P1	P2	P3	P4	P5
TA	0	0	1	1	2
TE	7	5	8	4	3



- $TEM=\frac{TER(P1)+TER(P5)+TER(P4)+TER(P1)+TER(P3)}{5}$
- $TER(P2) = T_F(P2) - T_A(P2) = 5 - 0 = 5$
- $TER(P5) = T_F(P5) - T_A(P5) = 8 - 2 = 6$
- $TER(P4) = 11$
- $TER(P1) = 19$
- $TER(P3) = 26$

$TEM=13,4$

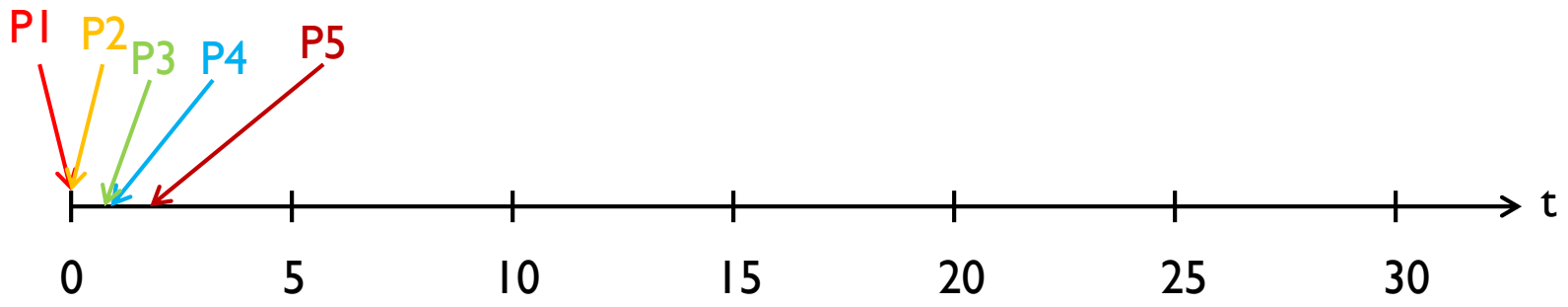
Ordonnancement monoprocesseur

- Ordonnancement préemptif
  - **Ordonnancement de type tourniquet (round robin)**
    - Tous les processus à ordonnancer sont au même pied d'égalité
    - Le processeur est attribué, à tour de rôle, à chacun des processus pour une durée de temps donnée (quantum)
    - Un processus, qui n'a pas terminé son exécution au bout de son quantum, se met de nouveau dans la file d'attente
    - Avantage
      - L'algorithme est simple et équitable
    - Inconvénient
      - Le choix du quantum est très déterminant pour les performances du système
        - Un quantum très petit provoquerait des basculements très fréquents
          - dégradation des performances
        - Un quantum très grand
          - temps d'attente est important pour les processus prêts

Ordonnancement monoprocesseur

- Ordonnancement préemptif
  - **Ordonnancement round robin**
    - Exemple:

P	P1	P2	P3	P4	P5
TA	0	0	1	1	2
TE	7	5	8	4	3



Ordonnancement monoprocesseur

- **Ordonnancement round robin**
  - Exemple: Q=2

P	P1	P2	P3	P4	P5
TA	0	0	1	1	2
TE	7	5	8	4	3

P1(2/5)	P2(2/3)	P3(2/6)	P4(2/2)	P5(2/1)
P1(2/3)	P2(2/1)	P3(2/4)	P4(2/0)	P5(1/0)
P1(2/1)	P2(1/0)	P3(2/2)		
P1(1/0)	P3(2/0)			

Ordonnancement monoprocesseur

- **Ordonnancement round robin**
  - Exemple:

P	P1	P2	P3	P4	P5
TA	0	0	1	1	2
TE	7	5	8	4	3

- $TEM = \frac{TER(P1)+TER(P2)+TER(P3)+TER(P4)+TER(P5)}{5}$
- $TER(P1) = T_F(P1) - T_A(P1) = 25 - 0 = 7$
- $TER(P2) = T_F(P2) - T_A(P2) = 22 - 0 = 12$
- $TER(P3) = 27 - 1 = 26$
- $TER(P4) = 17$
- $TER(P5) = 17$

$TEM = 21,4$

Ordonnancement monoprocesseur

- Algorithmes avec ou sans réquisition
  - **Ordonnancement par priorités**
    - En réalité, les processus ne sont pas de même niveau d'importance
      - Les processus système sont le pivot du système et par la suite doivent être les plus prioritaires
      - Un processus d'envoi de courrier n'est pas sensible à un petit retard
      - Un processus de traitement de texte peut être sensible au retard d'affichage des caractères saisis au clavier
    - Il est important de définir des priorités entre les processus

Ordonnancement monoprocesseur

- Ordonnancement par priorités
  - Principe
    - Attribution d'une priorité à chaque processus
    - Parmi les processus prêts, l'ordonnanceur choisit celui qui a la plus grande priorité
    - La priorité peut être
      - Statique : ne change pas le long de l'exécution
        - un processus de priorité supérieure est toujours prioritaire par rapport à un autre de prioritaire moindre
      - Dynamique
        - Le niveau de priorité du processus qui détient la CPU baisse dans le temps jusqu'à ce qu'un autre processus devient plus prioritaire et lui prend la CPU
        - De même le processus prêt se voit augmenter le niveau de priorité au fil du temps ce qui lui donne la chance d'avoir la CPU

Ordonnancement monoprocesseur

- Ordonnancement par priorités
  - Catégories de priorités
    - Le système définit plusieurs catégories de priorités
    - Regrouper les processus par catégorie
    - Les processus de la même catégorie ont le même niveau de priorité
      - L'ordonnancement adopté entre ces processus est celui de type Tourniquet
      - Les processus de la catégorie de priorité supérieure sont toujours prioritaires par rapport aux autres processus des autres catégories

Ordonnancement monoprocesseur

- Ordonnancement par priorités
  - Files d'attente multiples
    - Définition de plusieurs catégories de priorités
    - Le quantum du processeur n'est pas le même pour toutes les catégories. La catégorie de niveau de priorité inférieure a un quantum relativement grand que celui de la catégorie de priorité supérieure
    - Récompenser le retard des processus de priorité inférieure par une durée de détention de la CPU supérieure
    - Le processus qui termine sa durée et lâche le processeur passe à la catégorie inférieure

Ordonnancement monoprocesseur

Ordonnancement par priorités

- Exemple: à t=0, les 4 processus P1, P2, P3 et P4 demandent le processeur, en appliquant un ordonnancement avec priorité, sans réquisition, déterminer le temps moyen d'exécution des processus, avec l'indice de priorité faible correspond à une priorité haute.

P	P1	P2	P3	P4
Priorité	1	4	2	3
TE	5	2	5	4

- $TEM = \frac{TER(P1)+TER(P3)+TER(P4)+TER(P2)}{4} = 11,25$
- $TER(P1) = T_F(P1) = 5$
- $TER(P3) = 10$
- $TER(P4) = 14$
- $TER(P2) = 16$

Ordonnancement monoprocesseur

- Autres algorithmes d'ordonnancement
  - Ordonnancement garanti
    - Le principe de cet ordonnancement est que les processus reçoivent une garantie d'exécution en fonction d'une allocation équitable du temps CPU.
    - Le système maintient, pour chaque processus, un taux correspondant à la durée effective d'utilisation de la CPU par rapport à son quota (la période/nombre de processus)
    - Par exemple, si trois processus doivent s'exécuter simultanément et que le CPU est partagé équitablement, chaque processus doit obtenir environ **1/3 du temps CPU.**
    - Si un processus a déjà consommé plus que sa part garantie, il sera temporairement mis en attente pour que les autres puissent rattraper leur quota.
      - Un processus de taux ½ (a utilisé la moitié de son quota) est prioritaire par rapport au processus de taux 1,2 (a dépassé son quota de 20%)

Ordonnancement monoprocesseur

- Autres algorithmes d'ordonnancement
  - Ordonnancement par tirage au sort
    - Cette approche attribue des **tickets** aux processus et sélectionne aléatoirement l'un d'eux pour exécution.
    - L'ordonnanceur effectue, à chaque fois, un tirage au sort et le processus possédant le **ticket** tiré est exécuté.
    - Après une certaine durée (quantum de temps), un nouveau tirage est effectué.
    - Des processus coopératifs peuvent s'échanger leurs tickets

Ordonnancement monoprocesseur

- Autres algorithmes d'ordonnancement
  - Ordonnancement équitable
    - Le principe de l'ordonnancement équitable est de répartir le temps CPU à parts égales entre les utilisateurs plutôt qu'entre les processus
    - Si un utilisateur exécute plusieurs processus, ces processus doivent partager entre eux la part qui lui est allouée.
    - Par exemple, si deux utilisateurs ont des droits égaux et que l'un exécute un seul processus alors que l'autre en exécute trois, le premier processus **recevra 50% du CPU,** tandis que les trois autres se partageront les **50% restants (soit 16,6% chacun).**

Ordonnancement multiprocesseurs

- Les ordinateurs actuels sont équipés de plusieurs processeurs pour un traitement encore plus vite. On parle de processeur multicoeurs ou d'architecture multiprocesseur.
- Soit ils ont effectivement plusieurs processeurs connectés à la carte mère, soit ils ont un seul processeur qui comprend plusieurs unités de traitement indépendante dit multi cœurs.

Ordonnancement multiprocesseurs

- Les processeurs multi cœurs contiennent plusieurs *cœurs*, chaque cœur pouvant exécuter des instructions de manière autonome, ce qui signifie un système multi cœurs peut effectuer plusieurs tâches en parallèle, améliorant ainsi les performances globales du système.

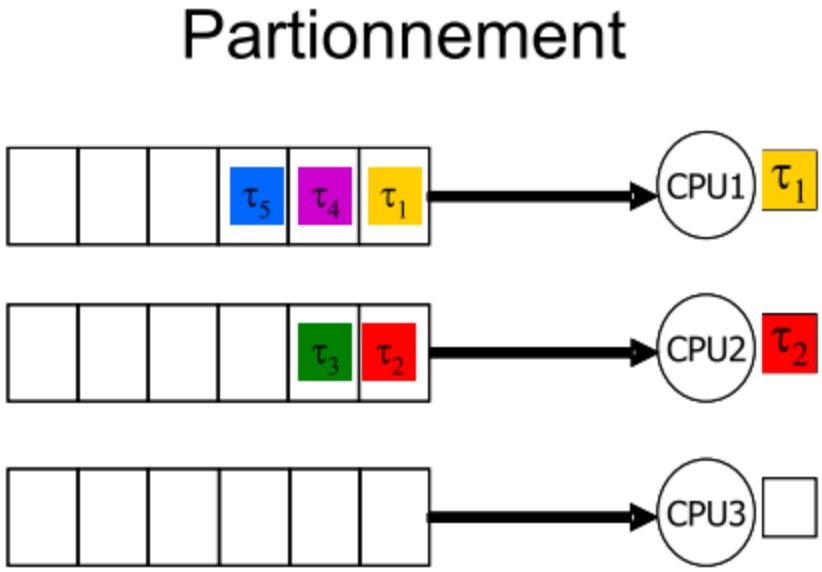
- L'idée derrière les processeurs multi cœurs et de mieux utiliser la ressource matérielle tout en réduisant la consommation énergétique

Ordonnancement multiprocesseurs

- L'ordonnancement multiprocesseur peut être effectué en utilisant:
  - Stratégie d'ordonnancement par partitionnement
  - Stratégie d'ordonnancement globale
  - Stratégie d'ordonnancement hybride (semi-partitionnement) obtenue par combinaison de la stratégie globale et par partitionnement.

Ordonnancement multiprocesseurs

- Stratégie par partitionnement
  - Consiste à partitionner l'ensemble des n tâches en m sous-ensembles disjoints, puis d'ordonnancer chaque sous-ensemble de tâches sur un processeur pj avec une stratégie d'ordonnancement "locale" monoprocesseur.
  - Il y a un ordonnanceur par processeur. Les tâches allouées aux processeurs ne sont pas autorisées à migrer d'un processeur à l'autre, la préemption ne peut entraîner de migration.

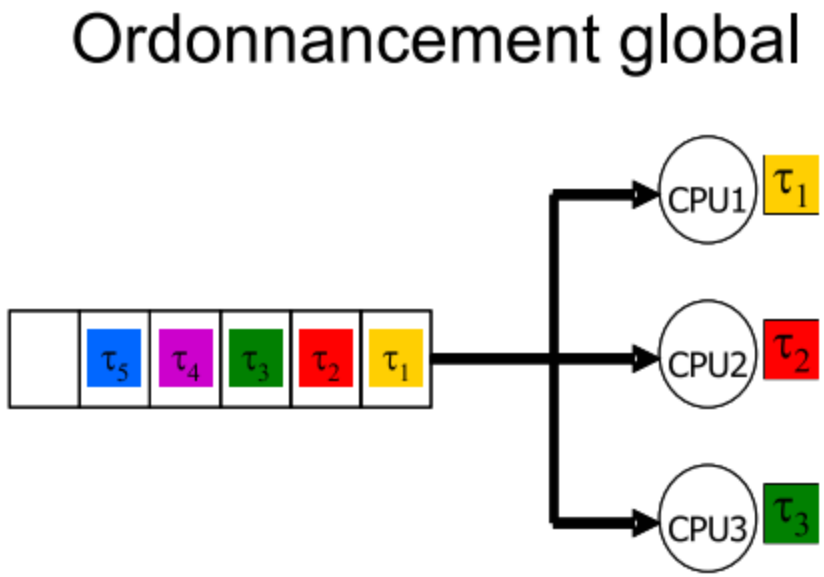


Ordonnancement multiprocesseurs

- Stratégie par partitionnement
  - Avantage
    - Peut ramener le problème d'ordonnancement multiprocesseur en plusieurs problèmes d'ordonnancement monoprocesseur pour lesquels, les algorithmes précédents peuvent être réutilisés sans problème et il suffit de rajouter un algorithme de répartition des programmes sur les différents CPU.
  - Inconvénient
    - Utilisation moins optimale des ressources (processeur libre).

Ordonnancement multiprocesseurs

- Stratégie globale
  - Applique globalement un algorithme d'ordonnancement sur l'ensemble de l'architecture multiprocesseur
  - Toutes les tâches sont dans la même queue des tâches prêtes qui est partagée par l'ensemble des processeurs.
  - Dans cette queue les m tâches les plus prioritaires sont sélectionnées pour être exécutées sur les m processeurs.



Ordonnancement multiprocesseurs

- Stratégie globale
  - Une tâche peut commencer son exécution sur un processeur, être préemptée par l'arrivée d'une nouvelle tâche plus prioritaire et reprendre son exécution sur un autre processeur
  - Dans la stratégie globale, nous avons un seul ordonnanceur, et préempter une tâches revient éventuellement à la faire migrer vers un autre processeur.
  - C'est ce phénomène de migration de tâches qui caractérise cette stratégie

Ordonnancement multiprocesseurs

- Stratégie globale
  - Avantage
    - Permettre une meilleure utilisation des processeurs.
  - Inconvénient
    - Le coût de migration des tâches est non négligeable.

Ordonnancement multiprocesseurs

- Stratégie par semi-partitionnement
  - Elle est obtenue par combinaison de la stratégie par partitionnement et de la stratégie globale.
  - Chaque tâche est allouée à un processeur pour toute son exécution, cependant si une tâche ne peut s'exécuter entièrement sur aucun processeur, son exécution est partagée sur deux ou plusieurs processeurs dont la charge maximale n'est pas atteinte (les migrations sont autorisées pour ce type de tâches).

Communication interprocessus

Présentation

- Sur une plateforme multiprogrammée, les processus ont généralement besoin de communiquer pour compléter leurs tâches.
- Un processus est dit indépendant, s'il n'affecte pas les autres processus ou ne peut pas être affecté par eux.
  - Un processus qui ne partage pas de données avec d'autres processus est indépendant.
- Un processus est dit coopératif s'il peut affecter les autres processus en cours d'exécution ou être affecté par eux.
  - Un processus qui partage des données avec d'autres processus est un processus coopératif.

Présentation

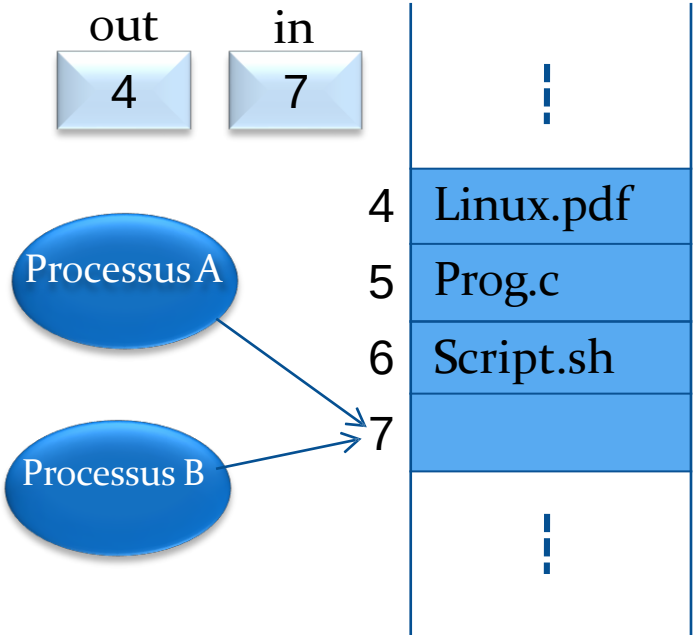
- La communication interprocessus est assurée généralement via des ressources partagées
- Une ressource désigne toute entité dont un processus a besoin pour s'exécuter:
  - Ressource matérielle (processeur, imprimante, etc)
  - Ressource logicielle (variable)
- Trois phases pour l'exploitation d'une ressource par un processus:
  - Sollicitation de la ressource
  - Utilisation de la ressource
  - Libération de la ressource
- Une ressource est dite critique lorsque des accès concurrents à cette ressource peuvent mené à un état incohérent

Conditions de concurrence

- Exemples illustratifs: Spooler d'impression
  - Principe
    - Un processus qui veut imprimer un fichier, entre son nom dans un répertoire spécial : répertoire du spooler
    - Le démon d'impression (spooler) inspecte périodiquement son répertoire, s'il y a un fichier il l'imprime et le supprime du répertoire
    - Le répertoire du spooler possède un grand nombre d'entrées numérotées : 0, 1 ,2 … chacune pouvant accueillir un nom de fichier.
    - Le spooler utilise deux variables partagées : out et in
      - Out : pointe vers le prochain fichier à imprimer
      - In : pointe vers la prochaine entrée libre
      - Les variables out et in peuvent être stockées dans un fichier

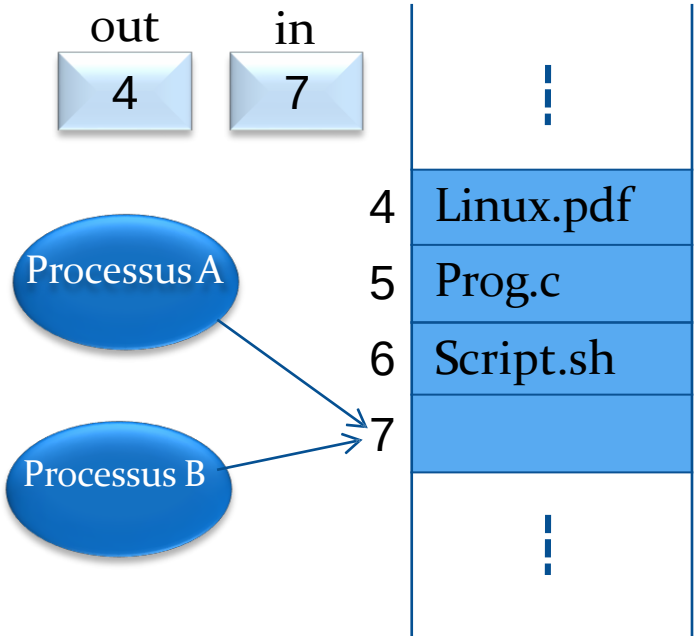
Conditions de concurrence

- Exemples illustratifs: Spooler d'impression
- Problème : deux processus A et B tentent d'imprimer en même temps
  - Le processus A lit la valeur de la variable in (7) et la mémorise dans une variable locale
  - Avant d'insérer le nom de son fichier dans l'entrée 7, son quantum a été épuisé , la CPU passe à un autre processus
  - Le processus B qui obtient, à son tour, la CPU lit la valeur de in (toujours 7) et la mémorise dans sa variable locale. Ensuite, il insère le nom de son fichier dans l'entrée 7 et actualise la variable in (8)



Conditions de concurrence

- Exemples illustratifs: Spooler d'impression
- Problème : deux processus A et B tentent d'imprimer en même temps
  - Quand la CPU revient au processus A, il reprend son exécution au point où la CPU lui a été retirée (i.e. pour lui la valeur de in est 7). Ensuite, il insère le nom de son fichier dans l'entrée 7 et actualise la variable in (8)
    - Pour le spooler tout est cohérent
  - Le fichier du processus B ne sera jamais imprimé, il a été écrasé par le fichier du processus A



Conditions de concurrence

- Exemples illustratifs: un compte bancaire doit être créditer de 5000dh et débité de 100dh

  - Un processus est chargé d'ajouter 5000dh

```
solde=lire(compte);
solde = solde + 5000;
ecrire (compte, solde) ;
```

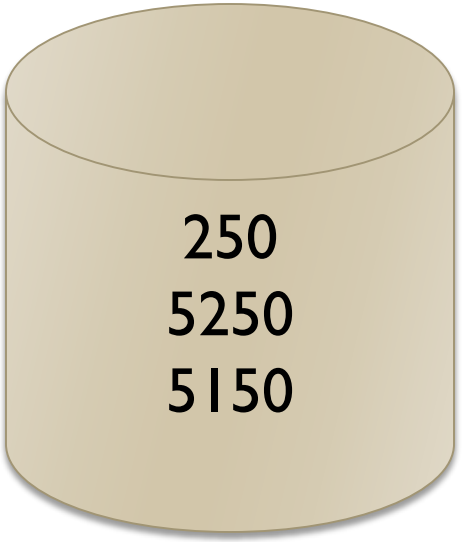
  - Un autre processus est chargé de soustraire 100dh

```
solde=lire(compte);
solde = solde - 100;
ecrire (compte, solde) ;
```

Conditions de concurrence

- Exemples illustratifs: un compte bancaire doit être créditer de 5000dh et débité de 100dh
  - Scénario 1: Exécution séquentiel
    - Le processus créditeur s'exécute avant le processus débiteur

```
solde=lire(compte);
        solde=250
solde = solde + 5000
        solde = 250+5000=5250
ecrire (compte, solde)
        compte = 5250
solde=lire(compte);
        solde=5250
solde = solde – 100
        solde = 5250 -100=5150
ecrire (compte, solde)
        compte = 5150
```

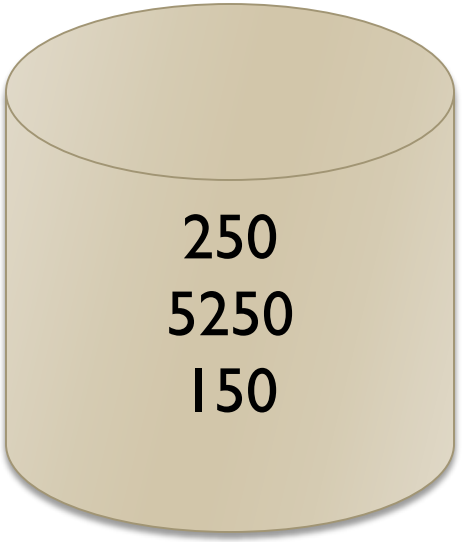


Compte

Conditions de concurrence

- Exemples illustratifs: un compte bancaire doit être créditer de 5000dh et débité de 100dh
  - Scénario 2: Exécution parallèle
    - Les processus créditeur et débiteur s'exécutent en même temps

```
solde=lire(compte);
        solde=250
solde = solde – 100
        solde = 250 -100=150
            solde=lire(compte);
                solde=250
            solde = solde + 5000
            solde = 250+5000=5250
            ecrire (compte, solde)
                compte = 5250
ecrire (compte, solde)
        compte = 150
```



Compte

Comme le montre cet exemple, le résultat final n'est pas un résultat cohérent (le client a perdu 5000dh dans cet affaire).

Synchronisation des processus

- Pour garantir la cohérence, il faut implémenter des mécanismes de synchronisation

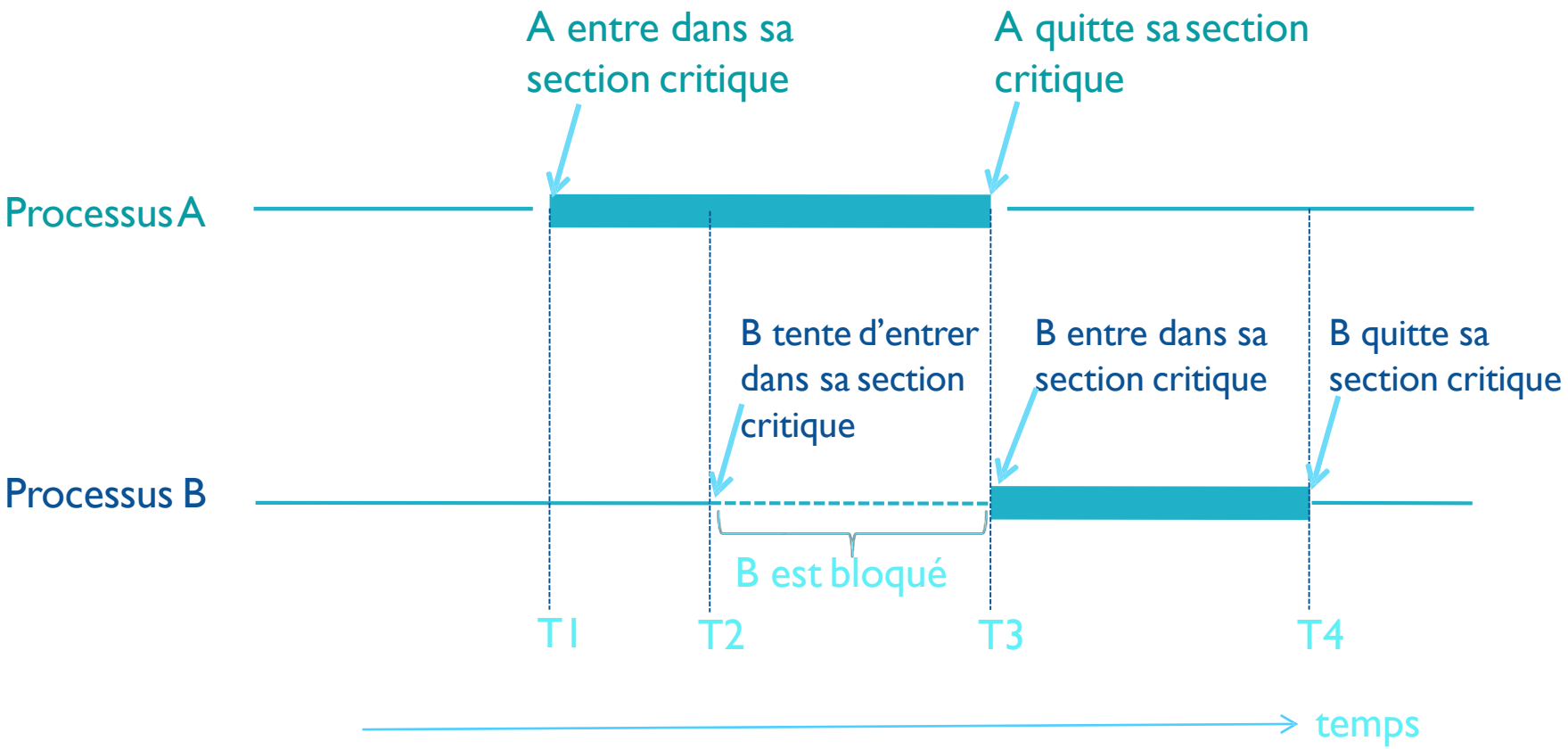
Synchronisation des processus

- Comment un processus passe des informations à un autre processus?
- Comment éviter des conflits entre des processus concurrents qui s'engagent dans des activités critiques?
- Comment synchroniser les processus dépendants?

Section critique

- Le problème dans les programmes précédents est dû aux conflits d'accès au même ressource. Ces accès sont en lecture et en écriture. Evidemment, il ne peut y avoir de conflit si les accès aux données ne sont qu'en lecture seule.
- On appelle section critique la partie d'un programme où se produit le conflit d'accès
- Comment éviter ces conflits d'accès ?
  - Il faut trouver un moyen d'interdire la lecture ou l'écriture des données partagées à plus d'un processus à la fois
  - Dans les exemples précédents, il faut obliger le 2ème processus à attendre la terminaison du 1er avant d'exécuter à son tour la même procédure
- Solution
  - L'exclusion mutuelle est une méthode qui assure qu'un seul processus est autorisé d'accéder à une ressource partagée, les autres processus seront exclus de la même activités

Section critique et Exclusion mutuelle



Critères de bonne synchronisation des processus

- Quatre conditions sont nécessaires pour assurer une bonne synchronisation des processus :

1. Exclusion mutuelle : deux processus ne doivent pas être en même temps en section critique
2. Attente bornée (Famine) : un processus ne doit pas attendre trop longtemps avant d'entrer en section critique
3. Progression (Interblocage) : un processus en dehors d'une section critique ne doit pas empêcher les autres d'y entrer
4. Aucune hypothèse : Il ne faut pas faire d'hypothèse quant à la vitesse de processus ou le nombre de processus

Algorithmes de synchronisation

- Plusieurs manières d'assurer la synchronisation
  - Synchronisation par attente active:
    - Masquage des interruptions
    - Variables de verrouillage
    - Alternance stricte
    - Solution de Peterson
  - Synchronisation par attente passive:
    - Le sommeil et l'activation
    - Les sémaphores
    - Les moniteurs
    - Echange de messages
    - Barrières

Synchronisation par attente active

- Un processus désirant entrer dans une section critique doit être mis en attente jusqu'à la section critique devient libre
- Un processus quittant la section critique doit le signaler aux autres processus
- Protocole d'accès à une section critique :

  <Entrer_Section_Critique> /* attente si SC non libre */
  <Section_Critique> /* Un seule processus en SC */
  <Quitter_Section_Critique>
- L'attente peut être :
  - Active : la procédure entrer_Section_Critique est une boucle dont la condition est un test qui porte sur des variables indiquant la présence ou non d'un processus en Section critique
  - Non active : le processus passe dans l'état endormi et ne sera réveillé que lorsqu'il sera autorisé à entrer en section critique
- Que contiennent les procédures entrer_Section_Critique et quitter_Section_Critique ?

Synchronisation par attente active

- 1ère Solution : Masquage des interruptions
  - Lorsque un processus entre en section critique il doit désactiver les interruptions. Et Lorsqu'il quitte la section critique, il doit restaurer les interruptions
    - l'interruption horloge qui permet d'interrompre un processus lorsqu'il a épuisé son quantum (temps CPU), serait ignorée, alors le processeur ne peut pas basculer d'un processus à un autre
  - Solution dangereuse en mode utilisateur s'il oublie de restaurer les interruptions
  - Si le système est multiprocesseur, la désactivation des interruptions concerne seulement une CPU, les autres peuvent accéder la mémoire partagée
  - La désactivation des interruptions peut être intéressante pour les processus en mode noyau, car ces processus en général sont très courts

Synchronisation par attente active

- 2ème Solution : la variable de verrouillage
  - Un verrou (lock) est une variable binaire partagée qui indique la présence d'un processus en Section Critique :
    - si verrou = 0 alors la section critique est libre
    - si verrou = 1 alors la section critique est occupée
  - Procédure entrer_Section_Critique ()

```
void entrer_Section_Critique () {
while (verrou == 1) ;        /* attente active */
verrou=1 ;                   /* Processus en SC*/}
```

Procédure quiter_Section_Critique()

```
void entrer_Section_Critique () {
        verrou=0 ;
        }
```

  - Problème
    - La variable verrou est partagée => pour y accéder, les conditions de concurrence peuvent se produire
    - Un processus qui a testé le verrou et le trouve à 0, perd le processeur avant de remettre le verrou à 1.
    - Un deuxième processus teste aussi le verrou (toujours à 0) et le remet à 1 ensuite, il entre dans sa section critique
    - Le premier processus, qui reprend le processeur avant que le deuxième ne termine sa section critique, remet le verrou à 1(il est déjà à 1) et commence sa section critique
    - Résultat : l'exclusion mutuelle n'est pas assurée, les deux processus se trouvent en même temps dans la section critique!!

Synchronisation par attente active

- 3ème Solution : l'alternance stricte
  - On utilise une variable partagée (tour) qui mémorise le numéro du processus autorisé à entrer en section critique
  - La SC de Pi est exécuté ssi turn=i
  - Pi est occupé à attendre si Pj est dans la SC
  - Exemple d'utilisation pour 2 processus :

```
Processus P1
while (true) {
while (turn != 1) ; /* attente active */
      entrer_SC;
      turn=2;
      quitte_SC;
}
```

```
Processus P2
while (true) {
while (turn != 2) ; /* attente active */
            entrer_SC;
            turn=1;
            quitte_SC;
 }
```

  - Avantage : fonctionne pour l'exclusion mutuelle
  - Inconvénient : problème de **famine**, un processus possédant tour, peut ne pas être intéressé immédiatement par la section critique

Synchronisation par attente active

- Solution de Peterson
  - Combine l'alternance stricte et la variable verrou
  - Elle se base sur les 2 fonctions entrer_SC et quitter_SC
  - Chaque processus doit, avant d'entrer dans sa SC appeler la fonction entrer_SC en lui fournissant en paramètre son numéro
  - À la fin de la SC il doit appeler quiter_SC pour indiquer qu'il quitte sa section critique et autorise l'accès à l'autre processus
  - Un processus peut enchaîner deux (ou plusieurs) exécutions de la section critique, alors que l'autre processus n'en a pas exécuté une.

```
Processus P0
while (true) {
entrer_SC(0) ;
    SC_P0();
quitter_SC(0)
}
```

```
Processus P1
while (true) {
entrer_SC(1) ;
    SC_P1();
quitter_SC(1);
}
```

Synchronisation par attente active

- Solution de Peterson
  - Pour le cas de deux processus P0 et P1 :

```
#define FAUX  0
#define VRAI   1
#define N      2                                   /* nombre de processus*/
int tour ;                                         /* à qui le tour */
int interesse[N] ;                                 /* initialisé à FAUX (0)*/
void entrer_Section_Critique (int process)         /* n° de processus : 0 ou 1*/
{
int autre ;                                        /* l'opposé du processus*/
autre = 1-process ;
interesse[process]=VRAI ;                          /* indiquer qu'on est intéressé */
tour = process ;                                   /* définit l'indicateur */
while (tour == process && interesse[autre] == VRAI);   /* boucler, sinon le processus
                                                   intéressé va exécuter sa section SC*/
}
void quitter_Section_Critique(int process)
{
interesse[process]=FAUX ;
}
```

Synchronisation par attente active

- Solution de Peterson

  - Pourquoi l'exclusion mutuelle est assurée par cette solution?
    - Réponse : Considérons la situation où les deux processus appellent « entrer_Section_Critique() » simultanément. Les deux processus sauvegarderont leur numéro dans la variable tour. La valeur sauvegardée en dernier efface la première. Le processus qui entrera en SC est celui qui a positionné la valeur tour en premier.

  - Cette solution malgré qu'elle fonctionne bien, elle présente un gros inconvénient : elle est basée sur l'attente active.

Synchronisation par attente active

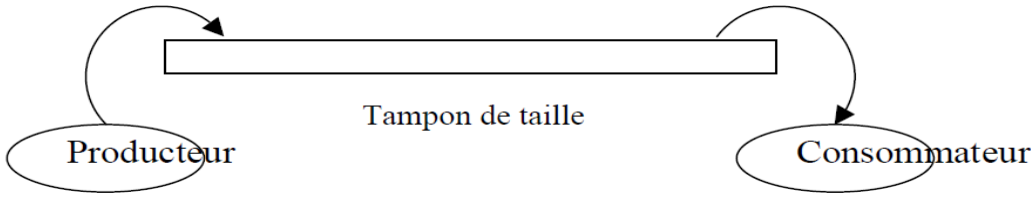
- Limitations
  - Toutes les solutions de cette catégorie obligent le processus en attente d'entrer dans sa section critique à rester actif
    - Le processus reste dans une petite boucle
  - Cette approche est très consommatrice en temps processeur
    - Le processus consomme, inutilement, sa part du processeur
- Solutions évitant l'attente active:
  - Le sommeil et l'activation
  - Les sémaphores
  - Echange de messages
  - Barrières

Synchronisation par attente passive

- Sommeil et activation
  - Principe
    - Un processus qui ne pouvant pas entrer en section critique, passe dans un état endormi, et sera réveillé lorsqu'il pourra y entrer.
  - Nécessite un mécanisme de réveil
  - Le SE fournit deux appels système :
    - Sleep (dormir) qui bloque le processus appelant
    - Wakeup (réveiller) qui réveille le processus donné en argument

Sommeil et activation

- Application des primitives sleep et wakeup au module Producteur/consommateur



- Les deux processus partagent un même tampon
- Le producteur produit des objets qu'il dépose dans le tampon
- Le consommateur retire des objets du tampon pour les consommer

Sommeil et activation

- Producteur/consommateur
  - Deux processus se partagent un tampon (buffer) de taille fixe. Le premier (producteur) écrit des informations dans le tampon et le deuxième (consommateur) y récupère les informations
  - Le producteur se bloque quand le tampon devient plein
  - Le consommateur se bloque quand le tampon se vide
  - Principe
    - N étant la taille fixe du tampon
    - Les deux processus se partagent une variable « compteur» correspondant au nombre d'éléments dans le tampon
    - Consommateur
      - Si compteur = 0 alors le consommateur entre en sommeil
      - Si compteur = N-1 alors le consommateur réveille le producteur
    - Producteur
      - Si compteur = N alors le producteur entre en sommeil
      - Si compteur = 1 alors le producteur réveille le consommateur

Sommeil et activation

- Code des processus Producteur/consommateur

```
1.  #define N 100                          /* nbre d'emplacement ds tampon */
2.  int compteur = 0 ;                     /* nbre d'objets ds tampon */
3.  void producteur () {
4.  while (TRUE) {
5.        produire_objet(…) ;
6.        if (compteur == N)       sleep () ;
7.        mettre_objet(…) ;
8.        compteur = compteur + 1 ;
9.        if (compteur == 1)       wakeup(consommateur) ; }
10. }
11. void consommateur () {
12. while (TRUE) {
13.       if (compteur == 0)       sleep() ;
14.       retirer_objet(…);
15.       compteur = compteur – 1 ;
16.       if (compteur == N-1)     wakeup (producteur) ;
17.       consommer_objet(…) ;}
18. }
```

Sommeil et activation

- Analyse de cette solution :
  - Etat de concurrence avec problème producteur-consommateur
    - Tableau vide, consommateur vient de lire que compteur vaut 0 (ligne 13)
    - Ordonnanceur change de processus avant que le consommateur soit en veille, exécutant producteur
    - Producteur ajoute objet au tableau, incrémente compteur (lignes 7 et 8)
    - Comme compteur vaut maintenant 1, réveille consommateur (ligne 9)
    - Mais consommateur n'est en veille et le signal de réveil est perdu
    - Prochaine exécution du consommateur, il va se mettre en veille
    - Producteur va remplir le tableau et aussi se mettre en veille
    - Les deux processus sont en veille, attendant l'un après l'autre
  - Essence du problème: signal de réveil envoyé avant que l'autre processus soit en veille
    - Courses entre processus doivent être bien encadrées
    - Préemption à moments arbitraires génère des bogues parfois très difficiles à trouver