Chapitre 3 LES PROCESSUS

1. ASPECTS GENERAUX DES PROCESSUS

Un processus est un programme qui s'exécute, ainsi que ses données, sa pile, son compteur ordinal, son pointeur de pile et les autres contenus de registres nécessaires à son exécution. Un processus fournit l'image de l'état d'avancement de l'exécution d'un programme.

Attention : ne pas confondre un processus (**objet dynamique** dont l'exécution peut être suspendue, puis reprise), avec le texte d'un programme, source ou exécutable.

1.1 Simultanéité, ressources

On appelle **simultanéité** l'activation de plusieurs processus au même moment.

Si le nombre de processeurs est au moins égal au nombre de processus, on parle de simultanéité totale ou vraie, sinon de pseudo-simultanéité.

pseudo-simultanéité : c'est par exemple l'exécution enchevêtrée de plusieurs processus sur un seul processeur. La simultanéité est obtenue par commutation temporelle d'un processus à l'autre sur le processeur. Si les basculements sont suffisamment fréquents, l'utilisateur a l'illusion d'une simultanéité totale.

Dans le langage de la programmation structurée, on encadre par les mots-clés **parbegin** et **parend** les sections de tâches pouvant s'exécuter en parallèle ou simultanément.

Avec un SE qui gère le temps partagé (pseudo-parallèlisme par commutation de temps), conceptuellement chaque processus dispose de son propre processeur virtuel. Ainsi, concrètement il n'existe qu'un seul compteur ordinal dont le contenu est renouvelé à chaque commutation. Conceptuellement, tout se passe comme si chaque processus disposait de son propre compteur ordinal.

Dans certains SE, il existe des appels système pour créer un processus, charger son contexte et lancer son exécution. Dans d'autres SE, un processus particulier (INIT sous UNIX) est lancé au démarrage de la machine. Il crée un processus par terminal. Chacun de ces processus attend une éventuelle connexion, et lorsqu'une connexion est validée, il lance un nouveau processus chargé de lire et d'interpréter les commandes de l'utilisateur (Shell sous UNIX). Chacune de ces commandes peut elle-même créer un nouveau processus, etc... On aboutit ainsi à une **arborescence de processus**.

De façon simplifiée, on peut imaginer un SE dans lequel les processus pourraient être dans trois états :

- **élu** : en cours d'exécution. Un processus élu peut être arrêté, même s'il peut poursuivre son exécution, si le SE décide d'allouer le processeur à un autre processus

- **bloqué** : il attend un événement extérieur pour pouvoir continuer (par exemple une ressource; lorsque la ressource est disponible, il passe à l'état "prêt")

- **prêt** : suspendu provisoirement pour permettre l'exécution d'un autre processus

Le modèle processus permet une approche claire du fonctionnement de l'ordinateur : processus utilisateurs, processus systèmes (processus terminaux, processus disques, etc...) qui se bloquent par défaut de ressource, qui passent de l'état élu à l'état prêt. Dans ce modèle, la couche la plus basse du SE est l'ordonnanceur (scheduler). Il est surmonté d'une multitude de processus. La gestion des interruptions, la suspension et la relance des processus sont l'affaire de l'ordonnanceur.

1.2 Réalisation des processus

Pour mettre en œuvre le modèle des processus, le SE gère une **table des processus** dont chaque entrée correspond à un processus. Chaque ligne peut comporter des informations sur un processus : son état, son compteur ordinal, son pointeur de pile, son allocation mémoire, l'état de ses fichiers ouverts, et d'autres paramètres.

Dans bien des SE, on associe à chaque périphérique d'E/S une zone mémoire appelée **vecteur d'interruptions**. Il contient les adresses des procédures de traitement des interruptions cf. ch. 9). Lorsqu'une interruption survient, le contrôle est donné à l'ordonnanceur qui détermine si le processus élu doit être suspendu ou bien (par exemple, s'il s'agit du processus traitant une interruption plus prioritaire) s'il doit poursuivre son exécution en empilant l'interruption qui vient d'arriver.

1.3 Mécanismes de commutation

- par interruptions

- par trap ou déroutement sur erreur : il s'agit d'une extension du mécanisme des interruptions. En cas de détection d'erreur interne au processus (ex : division par 0, erreur d'adressage), le contrôle est passé au SE en cas d'erreur mettant en cause son intégrité (ex : erreur d'adressage) ou à une fonction de traitement de l'erreur du processus courant.

- appel au superviseur (noyau du SE) : dans un SE multi-utilisateur multi-programmé, toutes les interruptions sont contrôlées par le superviseur. Une raison (pas unique) : l'événement qui interrompt un processus est peut être destiné à un autre; comme le SE doit garantir l'indépendance des processus, c'est lui qui doit récupérer l'événement pour le transmettre au processus destinataire. Ainsi, en cas d'interruption :

  - on sauve le mot d'état et le contexte du processus en cours et on passe en mode superviseur (ou noyau ou système)

  - le traitement de l'interruption est réalisé soit par le superviseur lui-même (horloge, coupure), soit par un processus spécifique (pilote de périphérique d'E/S), soit par le processus interrompu (erreur interne)

  - on élit un nouveau processus à exécuter (peut-être celui qui avait été interrompu)

2. MODELE DE REPRESENTATION DE PROCESSUS

2.1 Décomposition en tâches

On appelle **tâche** une unité élémentaire de traitement ayant une cohérence logique. Si l'exécution du processus P est constituée de l'exécution séquentielle des tâches T_1, T_2,..., T_n, on écrit :

$$P = T_1\ T_2 ... T_n$$

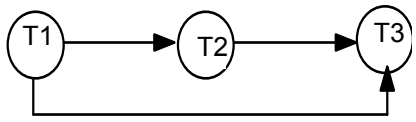
A chaque tâche T_i, on associe sa date de début ou d'initialisation d_i et sa date de terminaison ou de fin f_i.

Une **relation de précédence**, notée <, sur un ensemble E est une relation vérifiant :

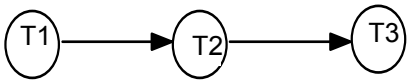
- $\forall$ T $\in$ E, on n'a pas T < T
- $\forall$ T $\in$ E et $\forall$ T' $\in$ E, on n'a pas simultanément T < T' et T' < T
- la relation < est transitive

La relation $T_i < T_j$ entre tâches signifie que f_i inférieur à d_j entre dates. Si on n'a ni $T_i < T_j$, ni $T_j < T_i$, alors on dit que T_i et T_j sont **exécutables en parallèle**.

Une relation de précédence peut être représentée par un graphe orienté. Par exemple, la chaîne de tâches S = ((T_1, T_2, T_3), ($T_i < T_j$ pour i inférieur à j)) a pour graphe :



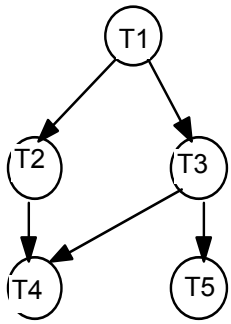
qu'on peut simplifier, en représentant le graphe de la plus petite relation qui a même fermeture transitive que la relation de précédence : c'est le **graphe de précédence** :



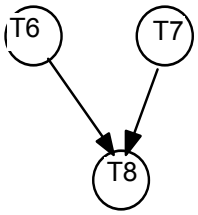
On munit l'ensemble des graphes de précédence de deux opérations :

- la composition parallèle : G_1 et G_2 étant deux graphes de précédence correspondant à des ensembles de tâches disjoints , G_1 // G_2 est l'union de G_1 et de G_2

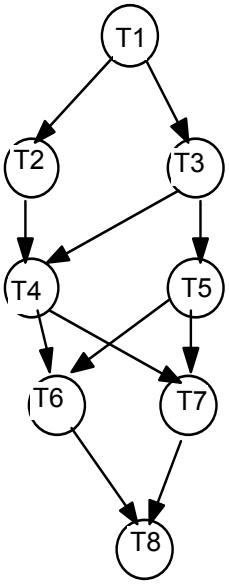
- le produit : G_1 * G_2 reprend les contraintes de G_1 et de G_2 avec en plus la contrainte qu'aucune tâche de G_2 ne peut être initialisée avant que toutes les tâches de G_1 ne soient achevées.



G1



G2



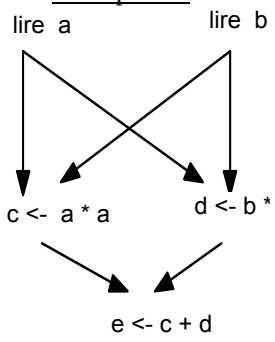
G1 * G2

2.2 Parallèlisation de tâches dans un système de tâches

La mise en parallèle s'effectue par la structure algorithmique :

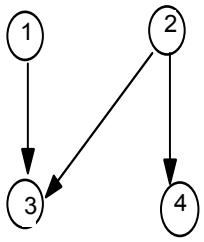
parbegin
.....
parend

exemple 1 :



```
soit   debut
           parbegin
                 lire  a
                 lire  b
           parend
           parbegin
                 c ← a * a
                 d ← b * b
           parend
           e ← c + d
       fin
```

exemple 2 : peut-on représenter le graphe suivant de tâches par un programme parallèle utilisant **parbegin** et **parend** ?



Pour augmenter le degré de multiprogrammation, donc le taux d'utilisation de l'UC, on peut exécuter en parallèle certaines tâches d'un processus séquentiel.

exemple : (système S0)

```
T1    lire  X
T2    lire  Z
T3    X ← X + Z
T4    Y ← X + Z
T5    afficher  Y
```

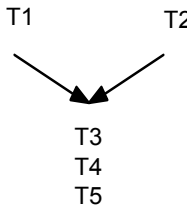
Le système S1 répartit les 5 tâches en 2 processus parallèles organisés chacun séquentiellement :

(S1)

P1	**P2**
T1	T2
T3	T4
T5	

S1 n'est pas équivalent à S0 (affichage pas forcément identique). S1 est **indéterminé**.

(S2)



Et S 2 ?

Méthode d'étude :

Supposons la mémoire composée de **m** cellules : $M = (C_1, C_2, ..., C_m)$

L'état initial du système est $S_0 = (C_1 (0), C_2 (0), ..., C_m (0))$
L'état k du système ($k \in [1 , l]$) est $S_k = (C_1 (k), C_2 (k), ..., C_m (k))$
après l'événement a_k du comportement $w = a_1 a_2 ....a_l$

Par exemple, considérons le problème précédent et $w = d_1 f_1 d_2 f_2 d_3 f_3 d_4 f_4 d_5 f_5$
avec M = (X, Y, Z) et $S_0 = (0 , 0 , 0)$

		d_1	f_1	d_2	f_2	d_3	f_3	d_4	f_4	d_5	f_5
X	0	0	α	α	α	α	$\alpha + \gamma$	$\alpha + \gamma$	$\alpha + \gamma$	$\alpha + \gamma$	$\alpha + \gamma$
Y	0	0	0	0	0	0	0	0	$\alpha + 2\gamma$	$\alpha + 2\gamma$	$\alpha + 2\gamma$
Z	0	0	0	0	γ	γ	γ	γ	γ	γ	γ
	0	1	2	3	4	5	6	7	8	9	10

A une tâche T, on associe :
- *son domaine de lecture* : $L = \{ C'_1, ..., C'_p\}$ ou ensemble des cellules lues par T
- *son domaine d'écriture* : $L = \{ C''_1, ..., C''_p\}$ ou ensemble des cellules écrites par T

écrire signifie plus généralement *modifier* l'état d'une ressource.

Dans l'exemple que nous développons, nous obtenons :

L1 = ∅ E1 = { X }
L2 = ∅ E2 = { Z }
L3 = { X , Z } E3 = { X }
L4 = { X , Z } E4 = { Y }
L5 = { Y } E5 = ∅

A une tâche T, on associe une fonction F_T de L dans E

Dans l'exemple que nous développons, $F_{T4} (x , z) = x + z$, x et $z \in N$

Sous-suite des valeurs écrites dans une cellule par un comportement w :

$V (X , w) = (0 , \alpha , \alpha + \gamma)$
$V (Y , w) = (0, \alpha + 2 \gamma)$
$V (Z , w) = (0, \gamma)$

2.3 Caractère déterminé d'un système de tâches. Conditions de Bernstein

Définition : un système de tâches S est **déterminé** si pour tous comportements w et w' et pour toute cellule C de M, on a : V (C, w) = V (C, w')

Avec l'exemple que nous traitons , soient $w = d_1 f_1 d_2 f_2 d_3 f_3 d_4 f_4 d_5 f_5$
et $w' = d_1 f_1 d_2 f_2 d_4 f_4 d_3 f_3 d_5 f_5$
$V (Y , w) = (0, \alpha + 2 \gamma)$ et $V (Y , w') = (0, \alpha + \gamma)$
Donc le système de tâches est non déterminé

Tout système séquentiel est déterminé

Il existe une relation entre le caractère déterminé d'un système et une propriété de non-interférence de tâches.

Définition : deux tâches T et T' sont **non-interférentes** vis à vis du système de tâches S si :

- T est un prédécesseur ou un successeur de T'

ou - $L_T \cap E_{T'} = L_{T'} \cap E_T = E_T \cap E_{T'} = \varnothing$

(conditions de Bernstein)

Avec l'exemple que nous développons, dans le système S1, T3 et T4 sont interférentes car T3 n'est ni prédécesseur, ni successeur de T4 et d'autre part $L_4 \cap E_3 \neq \varnothing$ (vaut { X })

Théorème : Si un système S est constitué de tâches 2 à 2 non interférentes, alors il est déterminé pour toute interprétation

Si un système S est déterminé pour toute interprétation et si $\forall$ T , $E_T \neq \varnothing$, les tâches de S sont 2 à 2 non interférentes.

Nous sommes donc en mesure de construire, à partir d'un système S, un système S' comportant moins de contraintes de précédence. A condition bien sûr que les contraintes d'écriture dans les différentes cellules ne soient pas modifiées.

2.4 Parallélisme maximal

Définition : deux systèmes S et S' construits sur le même ensemble de tâches sont **équivalents** si :

- S et S' sont déterminés

et - pour tout comportement w de S, pour tout comportement w' de S', pour toute cellule C de M, on a : V (C, w) = V (C , w')

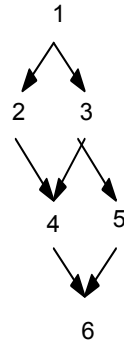
Autrement dit, la suite des valeurs écrites dans toute cellule par tout comportement de l'un ou l'autre système est unique.

Définition : un système S est de **parallélisme maximal** si :

- S est déterminé

et - son graphe de précédence G vérifie : la suppression de tout arc (T, T') entraîne l'interférence des tâches T et T'

Exemple : soit le système S3



M = (M1, M2, M3, M4, M5)

L1 = { M1 }	E1 = { M4 }
L2 = { M3, M4 }	E2 = { M1 }
L3 = { M3, M4 }	E3 = { M5 }
L4 = { M4 }	E4 = { M2 }
L5 = { M5 }	E5 = { M5 }
L6 = { M1, M2 }	E6 = { M4 }

Si l'on supprime l'arc (2 , 4), 2 n'est ni prédécesseur, ni successeur de 4,
mais $L2 \cap E4 = L4 \cap E2 = E4 \cap E2 = \varnothing$. Donc 2 et 4 ne deviennent pas interférentes.
Donc le système S3 n'est pas de parallélisme maximal.

Théorème : S = (E, <) étant un système déterminé, il existe un **unique** système S' de parallélisme maximal équivalent à S. S' est tel que : S' = (E , <') avec <' fermeture transitive de la relation :

$$R = \{ (T , T') \mid T < T' \text{ et } (L_T \cap E_{T'} \neq \varnothing \text{ ou } L_{T'} \cap E_T \neq \varnothing \text{ ou } E_T \cap E_{T'} \neq \varnothing) \text{ et } E_T \neq \varnothing \text{ et } E_{T'} \neq \varnothing \}$$

Algorithme : il découle du théorème :

- construire le graphe de R
- éliminer tous les arcs (T , T') redondants, c'est à dire tels qu'il existe un chemin de T à T' contenant plus d'un arc

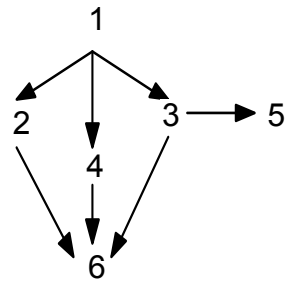
Exemple : avec le système S3 précédent : on démontre qu'il est déterminé

graphe de R : il comporte les arcs :

(1 , 2) car	$L1 \cap E2 = \{ M1 \}$
(1 , 3)	$L3 \cap E1 = \{ M4 \}$
(1, 4)	$L4 \cap E1 = \{ M4 \}$
(1 , 6)	$E6 \cap E1 = \{ M4 \}$
(2 , 6)	$L2 \cap E6 = \{ M4 \}$
(3 , 5)	$E3 \cap E5 = \{ M5 \}$
(3 , 6)	$L3 \cap E6 = \{ M4 \}$
(4 , 6)	$L4 \cap E6 = \{ M4 \}$

Les arcs (1,5) , (2,4) , (3,4) et (5,6) ne sont pas présents

L'arc (1 , 6), redondant, peut être supprimé.
D'où le système de parallélisme maximal équivalent à S3 :



3. ORDONNANCEMENT DES PROCESSUS

L'ordonnanceur (scheduler) définit l'ordre dans lequel les processus prêts utilisent l'UC (en acquièrent la ressource) et la durée d'utilisation, en utilisant un algorithme d'ordonnancement. Un bon algorithme d'ordonnancement doit posséder les qualités suivantes :

- équitabilité : chaque processus reçoit sa part du temps processeur
- efficacité : le processeur doit travailler à 100 % du temps
- temps de réponse : à minimiser en mode interactif
- temps d'exécution : minimiser l'attente des travaux en traitement par lots (batch)
- rendement : maximiser le nombre de travaux effectués par unité de temps

L'ensemble de ces objectifs est contradictoire, par exemple le 3ème et le 4ème objectif.

On appelle **temps de traitement moyen** d'un système de tâches la moyenne des intervalles de temps séparant la soumission d'une tâche de sa fin d'exécution. On appelle **assignation** la description de l'exécution des tâches sur le ou les processeurs. A chaque tâche T_i du système de tâches, on associe deux réels :

- t_i : sa date d'arrivée
- τ_i : sa durée

Il existe deux familles d'algorithmes :

- **sans réquisition (ASR) :** le choix d'un nouveau processus ne se fait que sur blocage ou terminaison du processus courant (utilisé en batch par exemple)

- **avec réquisition (AAR) :** à intervalle régulier, l'ordonnanceur reprend la main et élit un nouveau processus actif (algorithmes 3.2 à 3.7)

3.1 ASR

On utilise :

- soit l'ordonnancement dans l'ordre d'arrivée en gérant une file des processus. Cet algorithme est facile à implanter, mais il est loin d'optimiser le temps de traitement moyen

- soit l'ordonnancement par ordre inverse du temps d'exécution (PCTE) : lorsque plusieurs travaux d'égale importance se trouvent dans une file, l'ordonnanceur élit le plus court d'abord. On démontre que le temps moyen d'attente est minimal par rapport à toute autre stratégie **si toutes les tâches sont présentes dans la file d'attente au moment où débute l'assignation**. En effet, considérons 4 travaux dont les durées d'exécution sont a, b, c, d. Si on les exécute dans cet ordre, le temps moyen d'attente sera : (4a + 3b + 2c + d)/4. Comme a possède le plus grand poids (4), il faut donc que a soit le temps le plus court, etc...

Exemple : soient 5 tâches A, B, C, D et E de temps d'exécution respectifs 2 , 4 , 1 , 1 , 1 arrivant aux instants 0 , 0 , 3 , 3 , 3. L'algorithme du plus court d'abord donne l'ordre A , B, C , D , E et une attente moyenne de 2,8. L'ordre B , C , D , E , A donnerait une attente moyenne de 2,6

τ_i peut aussi être remplacé par la durée estimée e_i de l'exécution de la tâche T_i. On peut par exemple calculer les durées estimées selon une méthode de moyenne pondérée, à partir d'une valeur initiale e_0 arbitraire :

$$e_i = \alpha * e_{i-1} + (1 - \alpha) * \tau_{i-1}$$, avec souvent $\alpha = 0,5$

3.2 Ordonnancement circulaire ou tourniquet (round robin)

Il s'agit d'un algorithme ancien, simple et fiable. Le processeur gère une liste circulaire de processus. Chaque processus dispose d'un quantum de temps pendant lequel il est autorisé à s'exécuter. Si le processus actif se bloque ou s'achève avant la fin de son quantum, le processeur est immédiatement alloué à un autre processus. Si le quantum s'achève avant la fin du processus, le processeur est alloué au processus suivant dans la liste et le processus précédent se trouve ainsi en queue de liste.

La commutation de processus (overhead) dure un temps non nul pour la mise à jour des tables, la sauvegarde des registres. Un quantum trop petit provoque trop de commutations de processus et abaisse l'efficacité du processeur. Un quantum trop grand augmente le temps de réponse en mode interactif. On utilise souvent un quantum de l'ordre de 100 ms.

3.3 Ordonnancement PCTER

Il s'agit d'une généralisation avec réquisition de l'algorithme PCTE : à la fin de chaque quantum, on élit la tâche dont le temps d'exécution restant est minimal (PCTER = plus court temps d'exécution restant). Cet algorithme fournit la valeur optimale du temps moyen de traitement pour les algorithmes avec réquisition.

3.4 Ordonnancement avec priorité

Le modèle du tourniquet suppose tous les processus d'égale importance. C'est irréaliste. D'où l'attribution de priorité à chaque processus. **L'ordonnanceur lance le processus prêt de priorité la plus élevée**. Pour empêcher le processus de priorité la plus élevée d'accaparer le processeur, l'ordonnanceur abaisse à chaque interruption d'horloge la priorité du processus actif. Si sa priorité devient inférieure à celle du deuxième processus de plus haute priorité, la commutation a lieu.

Une **allocation dynamique de priorité** conduit souvent à donner une priorité élevée aux processus effectuant beaucoup d'E/S. On peut aussi allouer une priorité égale à l'inverse de la fraction du quantum précédemment utilisée. Par exemple un processus ayant utilisé 2 ms d'un quantum de 100 ms, aura une priorité de 50.

A chaque niveau de priorité, on associe une liste (classe) de processus exécutables, gérée par l'algorithme du tourniquet. Par exemple, supposons 4 niveaux de priorité. On applique à la liste de priorité 4 l'algorithme du tourniquet. Puis, quand cette liste est vide, on applique l'algorithme à la liste de priorité 3, etc... S'il n'y avait pas une évolution dynamique des priorités, les processus faiblement prioritaires ne seraient jamais élus : risque de famine.

Exemple : VAX/VMS de DEC et OS/2 d'IBM

3.5 Ordonnancement avec files multiples

On associe une file d'attente et un algorithme d'ordonnancement à chaque niveau de priorité. Si les priorités évoluent dynamiquement, le SE doit organiser la remontée des tâches.

Exemple : UNIX dont le mode d'ordonnancement sera présenté dans un chapitre ultérieur

3.6 Ordonnancement par une politique

S'il y a n utilisateurs connectés, on peut garantir à chacun qu'il disposera de 1/n de la puissance du processeur. Le SE mémorise combien chaque utilisateur a consommé de temps processeur et il

calcule le rapport : *temps processeur consommé/temps processeur auquel on a droit* . On élit le processus qui offre le rapport le plus faible jusqu'à ce que son rapport cesse d'être le plus faible.

3.7 Ordonnancement à deux niveaux

La taille de la mémoire centrale de l'ordinateur peut être insuffisante pour contenir tous les processus prêts à être exécutés. Certains sont contraints de résider sur disque.

Un ordonnanceur de bas niveau (*CPU scheduler*) applique l'un des algorithmes précédents aux processus résidant en mémoire centrale. Un ordonnanceur de haut niveau (*medium term scheduler*) retire de la mémoire les processus qui y sont resté assez longtemps et transfère en mémoire des processus résidant sur disque. Il peut exister un ordonnanceur à long terme (*job scheduler*) qui détermine si un processus utilisateur qui le demande peut effectivement entrer dans le système (si les temps de réponse se dégradent, on peut différer cette entrée).

L'ordonnanceur de haut niveau prend en compte les points suivants :
- depuis combien de temps le processus séjourne-t-il en mémoire ou sur disque ?
- combien de temps processeur le processus a-t-il eu récemment ?
- quelle est la priorité du processus ?
- quelle est la taille du processus ? (s'il est petit, on le logera sans problème)

3.8 Ordonnancement de chaînes de tâches

Supposons que l'on ait r chaînes de tâches à exécuter en parallèle : $C_1, C_2, ..., C_r$
avec : $C_i = T_{i1}\ T_{i2}\ T_{i3}\ ....\ T_{i\ ki}$, c'est à dire que la chaîne C_1 comprend k1 tâches, la chaîne C_2 comprend k2 tâches, etc... On note : k1 + k2 + ... + kr = n (nombre total de tâches).
On note : τ_{ij} la durée de la tâche T_{ij}

Il existe un algorithme produisant une assignation de temps moyen de traitement minimal :

pour chaque chaîne C_i , calculer la durée moyenne d'exécution (pas de traitement) ε_{ip} de chaque sous-chaîne C'_{ip} , pour p = 1 à ki
tant qu'il existe des tâches non assignées
déterminer j tel que : $\varepsilon_{j\,pj} = \min \{ \varepsilon_{m\,pm} \mid m = 1 , 2 , ... , n \}$
ajouter à l'assignation les tâches de la sous-chaîne $C'_{j\,pj}$ et les supprimer de la chaîne C_j
recalculer $\varepsilon_{j\,pj}$ pour la nouvelle chaîne C_j
fin tant que

Exemple : $C_1 : \tau_{11} = 9, \tau_{12} = 6 , \tau_{13} = 3 , \tau_{14} = 5 , \tau_{15} = 11$
$C_2 : \tau_{21} = 4 , \tau_{22} = 1 , \tau_{23} = 13$
$C_3 : \tau_{31} = 8 , \tau_{32} = 4 , \tau_{33} = 1 , \tau_{34} = 11$

On détermine le minimum : $\tau_{22} = 2{,}5$, donc : l'assignation T_{21} , T_{22}
puis $\tau_{33} = 4{,}33$; on complète l'assignation avec T_{31} , T_{32} , T_{33} ; etc ...
D'où l'assignation finale :

$$T_{21} , T_{22} , T_{31} , T_{32} , T_{33} , T_{11} , T_{12} , T_{13}, T_{14}, T_{15}, T_{34}, T_{23}$$

BIBLIOGRAPHIE

J. BEAUQUIER, B. BERARD, Systèmes d'exploitation, Ediscience, 1993

A. TANENBAUM, Les systèmes d'exploitation, Prentice Hall, 1999

Exercices sur le chapitre 3

1. Donner et comparer les assignations produites par les algorithmes FIFO, PCTE, tourniquet avec un quantum de 1, PCTER dans l'exemple suivant :

ordre d'arrivée des tâches	T1	T2	T3	T4	T5	T6	T7
durée	7	4	6	1	2	4	1
date d'arrivée	0	0	1	1	1	2	2

2. Sur un ordinateur, l'ordonnanceur gère l'ordonnancement des processus par un tourniquet avec un quantum de 100 ms. sachant que le temps nécessaire à une commutation de processus est de 10 ms, calculer le temps d'exécution moyen pour :

ordre d'arrivée des tâches	T1	T2	T3	T4	T5	T6	T7
durée	700	400	600	100	200	400	100
date d'arrivée	0	0	100	100	150	200	200

   Si l'on définit le rendement du processeur comme le rapport *temps pendant lequel l'UC exécute les processus/temps total de traitement*, calculer le rendement en ce cas.

3. Un SE utilise 3 niveaux de priorité (numérotés par ordre croissant). Le processus se voit affecter un niveau fixe. Une file de processus est attachée à chaque niveau. Chaque file est gérée par un tourniquet avec un quantum de 0,5. Un tourniquet de niveau n n'est activé que si toutes les files de niveau supérieur sont vides.

   Que peut-il se passer ?

   Donner l'assignation pour :

ordre d'arrivée des tâches	T1	T2	T3	T4	T5	T6	T7
durée	7	4	6	1	2	4	1
date d'arrivée	0	0	1	1	1	2	2
priorité	2	3	1	2	3	1	2

   Maintenant, on suppose que la priorité n'est pas fixe. Toutes les 2 unités de temps, tout processus n'ayant pas disposé de l'UC monte d'un niveau, alors que ceux en ayant disposé 2 fois en descendent. Donner la nouvelle assignation.