Université Mohamed Boudiaf de M'sila
Faculté de technologie
Département d'Électronique

Cours :

Systèmes Temps Réel (Real Time Systems)

Master2 Électronique des Systèmes EMbarqués (ESEM)

Dr. Youcef BRIK

Plan général :

- Introduction aux STR.
- Systèmes d'exploitation Temps Réel (RTOS).
- Gestion des tâches et ordonnancement.
- Gestion de la mémoire.
- Parallélisme, synchronisation et communication entre les processus.
- Programmation temps réel.
- Exercices corrigés.
- Exercices proposés.

Université de M'sila ©2021

Chapitre 01 :

Introduction aux systèmes temps réels

(Real time systems)

1. Définition :

Système : ensemble d' « activités » correspondant à un ou plusieurs traitements effectués en séquence ou en concurrence. Les traitements communiquent éventuellement entre eux. Le système est en interaction avec son environnement.

Système temps réel (STR): système dont le comportement dépend non seulement de l'exactitude des traitements effectués, mais également du temps où les résultats de ces traitements sont fournis, c.-à-d. qu'un retard dans la production d'un résultat est considéré comme une erreur.

Un STR dit critique s'il y a des conséquences d'une défaillance du système et concerne la sûreté de fonctionnement. Exemple : contrôle aérien.

Un STR est soumis à des contraintes temporelles, il n'est pas forcément rapide. L'échelle de temps peut varier selon le contexte.

Exemples :

- Réaction chimique (des heures),
- Chaine de fabrication (des minutes),
- Surveillance de centrales nucléaires (secondes),
- Suivi d'une missile/Radar (microseconde)
- Robotique,
- Fourniture d'images et son pour le multimédia,
- Suivi opératoire en milieu médical.

Dans un système temps réel, un résultat de calcul mathématiquement exact mais arrivant au-delà d'une échéance prédéfinie est **un résultat faux**.

Un système temps réel est composé d'un système contrôlé (procédé) et d'un système de contrôle :

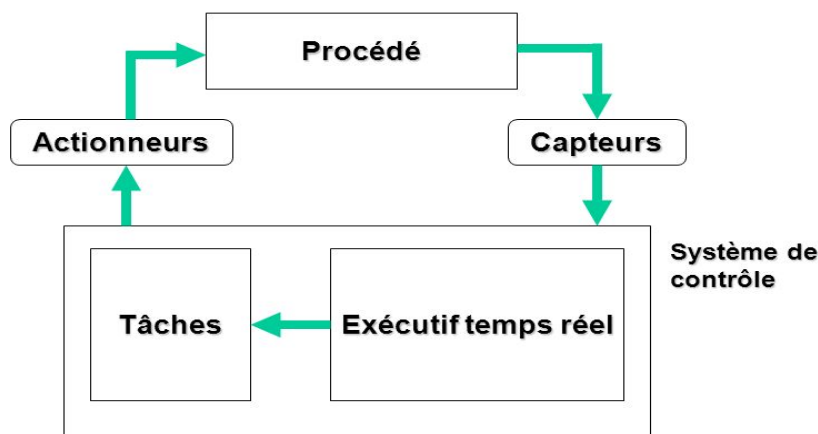


Figure 2.1. Cycle d'un système temps réel.

2. Classification des systèmes temps réels:

2.1. En termes d'interaction système/environnement :

2.1.1. Systèmes synchrones: A des moments déterminés par une référence de temps interne au système : systèmes pilotés par le temps (time driven system), programmés pour exécuter les actions/lectures à intervalles réguliers fixés par un timer (fonctionnement séquentiel).

Exemple: Feux rouge.

2.1.2. Systèmes asynchrones : A des moments déterminés par l'environnement lui-même: il attend les sollicitations et réagit à celles-ci: système piloté par les évènements (event driven system).

Exemple: Système de refroidissement d'un central électrique.

2.2. En termes de tolérance de dépassement:

2.2.1. Temps réel dur/strict (hard real time): ne tolère aucun dépassement de ces contraintes, ce qui est souvent le cas lorsque de tels dépassements peuvent conduire à des situations critiques ou des exceptions. Un système temps réel strict doit respecter des limites temporelles données même dans la pire des situations d'exécution possibles.

Exemples :

- Pilote automatique d'avion,
- Système de surveillance de centrale nucléaire.

2.2.2. Temps réel lâche/souple/mou (soft real time): s'accommode de dépassements des contraintes temporelles dans certaines limites. Un système temps réel souple doit respecter ses limites pour une moyenne de ses exécutions. On tolère un dépassement exceptionnel, qui pourra être compensé à court terme.

Exemples: - Visioconférence,

- mesure de température d'une chambre froide.

3. Caractéristiques d'un système temps réel :

3.1. Prévisibilité : Un STR doit être conçu tel que ses performances soient définies dans le pire cas.

Les performances de l'application doivent être définies dans tous les cas possibles de façon à assurer le respect des contraintes de temps. On parle de pire cas. Pour cela, il est nécessaire de connaître avec précision les paramètres des tâches : temps global de calcul de chaque activité, périodicité, date de réveil, etc.

3.2. Déterminisme : Enlever toute incertitude sur le comportement des activités :

- Dans un STR dur : on cherche à ce que toutes les échéances soient respectées.
- Dans un STR mou : minimiser le retard moyen des activités par exemple.

3.2.1. Sources de non-déterminisme:

- Charge de calculs (variations des durées d'exécution des activités),
- Entrées/sorties (temps de réaction, durée des communications),
- Interruptions (temps de réaction du système),
- Fautes et exceptions matérielles ou logicielles.

3.3. Fiabilité:

- Minimiser l'intervention humaine directe (systèmes temps réel embarqués),
- Conception tolérante aux fautes, pour garantir le comportement du système et de ses composants.

4. Limites des systèmes classiques pour le temps réel:

- politiques d'ordonnancement visent le partage équitable du temps d'exécution. Pas adaptées à des tâches plus critiques que d'autres,
- la gestion des E/S engendre de longues attentes (parfois non bornées),

- la gestion des interruptions n'est pas optimisée,
- les mécanismes de gestion de la mémoire virtuelle ne sont pas optimaux,
- les temporisateurs qui organisent le temps n'ont pas une résolution assez fine.

5. La structure d'un STR:

Un système TR est constitué de deux couches essentielles, la couche « matériel » et la couche « logiciel »

5.1. Hardware (matériels): peut avoir plusieurs configurations suivant l'importance de l'application et le cout à investir. Cependant, la structure de base est la même.

5.2. Software (logiciel): doit permettre l'exécution des tâches attribuées au calculateur, et de répondre en temps optimal à toutes les actions/lectures.

5.3. Compromis hardware/software:

- Capacités et rapidité (Mémoire, cpu, etc...)
- Algorithme (complexité)
- Gestion du hardware (ordonnancement, gestion de la mémoire, preémption).

Chapitre 02 :

Généralité sur les systèmes d'exploitation

(RTOS: Real Time Operating System)

1. Définitions :

Un système d'exploitation (**OS**) : est un ensemble de programmes spécialisés qui permet l'utilisation des ressources matérielles d'un ou plusieurs ordinateurs de façon optimale et efficace. Un OS fournit une interface homme-machine (IHM) permettant la communication entre l'utilisateur et les machines par les différents logiciels d'application.

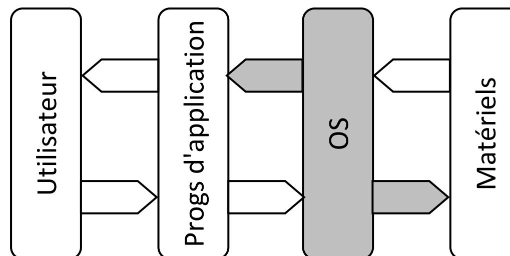


Figure 1.1. L'emplacement d'un OS.

2. Rôles d'un OS:

- **Gestion du processeur** : il gère l'allocation du processeur entre les différents programmes grâce à un algorithme d'ordonnancement.
- **Gestion de la mémoire vive** : il gère l'espace mémoire alloué à chaque application. En cas d'insuffisance de mémoire physique, le système d'exploitation peut créer une zone mémoire sur le disque dur, appelée «mémoire virtuelle».
- **Gestion des entrées/sorties** : il permet d'unifier et de contrôler l'accès des programmes aux ressources matérielles par l'intermédiaire des pilotes (gestionnaires de périphériques).
- **Gestion de l'exécution des applications**.
- **Gestion des fichiers** : il gère la lecture et l'écriture dans le système de fichiers et les droits d'accès aux fichiers par les utilisateurs et les applications.
- **Gestion des informations**.

3. Composantes des systèmes d'exploitation:

3.1. Le noyau (kernel) toujours en RAM d'OS, il contient les fonctionnalités critiques du SE: elles doivent:

- toujours être prêtes à l'utilisation
- traitement d'interruptions
- gestion d'UCT
- gestion de mémoire, fichiers, processus, des entrées sorties principales
- communication entre processus ... etc.

3.2. Interpréteur de commande/coquille (shell) permettant la communication avec le système d'exploitation par l'intermédiaire d'un langage de commandes, afin de permettre à l'utilisateur de piloter les périphériques.

3.3. Système de fichiers (file system), permettant d'enregistrer les fichiers dans une arborescence.

4. Notions importantes:

Processus: Suite continue d'opérations, d'actions constituant la manière de faire, de traiter quelque chose.

Programme: Ensemble d'instructions et de données représentant un algorithme qui peut être exécuté par un ordinateur.

Instruction: une opération élémentaire d'un processeur dans une architecture d'ordinateur.

Macro-instruction: instruction complexe, définissant des opérations composées à partir des instructions du répertoire de base d'un ordinateur.

Un système d'exploitation multitâche (*multithreaded*): lorsque plusieurs tâches (processus) peuvent être exécutées simultanément.

Un système préemptif: lorsqu'il possède un **ordonnanceur (planificateur)**, qui répartit, selon des critères de priorité, le temps entre les différents processus qui en font la demande (les interruptions).

Le multiprocessing: une technique qui consiste à faire fonctionner plusieurs processeurs en parallèle afin d'obtenir une puissance de calcul plus importante ou bien afin d'augmenter la disponibilité du système (en cas de panne d'un processeur). Un

système multiprocesseur gère le partage de la mémoire entre plusieurs processeurs, également il distribue la charge de travail.

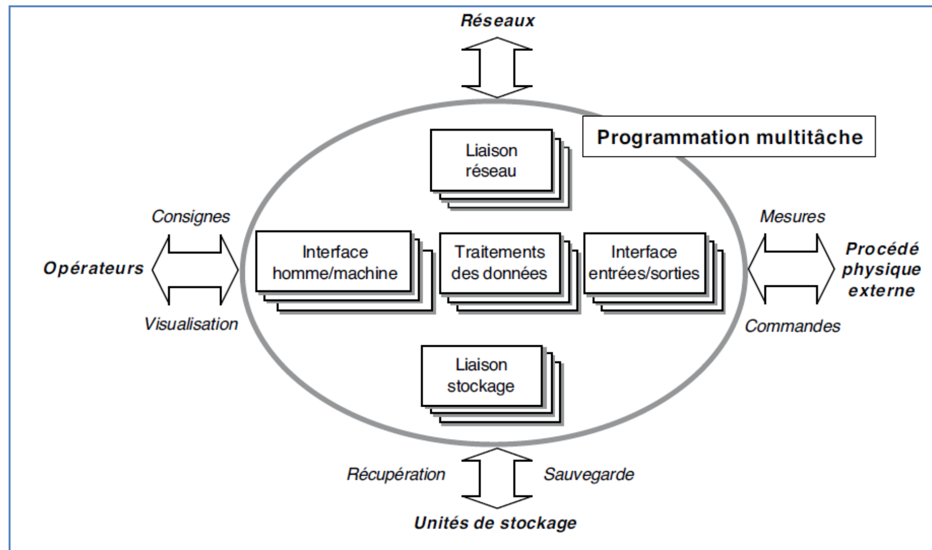


Figure 1.2. Architecture logicielle d'une application de contrôle-commande multitâche.

5. Types des OS:

5.1. OS généralistes:

- Windows
- Linux (Fedora, Debian, Ubuntu, etc...)

5.2. OS spécialisés:

- Android
- BlackberryOS
- iOS
- OSEK/VDX (Automobile)

5.3. OS temps réel: fonctionne de manière fiable selon des contraintes temporelles spécifiques, c'est-à-dire qu'il doit être capable de délivrer un traitement correct des informations reçues à des intervalles de temps bien définis. Voici quelques exemples de systèmes d'exploitation temps réel :

- RTLinux (hard Real Time Linux, USA);
- QNX (la société BlackBerry, USA/Canada);
- Xenomai (2001, USA);
- OpenVMS (la société *hp*, USA, utilisé dans le projet "*la ligne 14 métro Paris*");

- Micro-Itron (libre, Japan).
- RTEMS, FreeRTOS (libres et légers).

Remarque : Linux n'est pas TR, il peut être modifié pour être Temps Réel (PREEMPT-RT, Xenomai).

6. Différences entre OS et RTOS :

6.1. OS (Système d'exploitation ou système opératoire) :

1. Au démarrage, l'OS prend le contrôle,
2. Compilation + Edition des liens + Exécution du programme,
3. Multiprogrammation, multi-utilisateurs.

6.2. RTOS (Système d'exploitation temps réel) :

1. Edition des liens de l'application et du RTOS,
2. Au démarrage, l'application prend le contrôle et démarre le RTOS,
3. RTOS et l'application sont fortement couplés,
4. Aucune protection vis-à-vis de l'application → meilleure performance,
5. Services limités aux Systèmes embarqués → réduction de taille mémoire,
6. Configuration du RTOS : gestion des fichiers, pilotes des E/S, gestion de la mémoire, Outils,…

7. Fonction d'un RTOS :

Les principales fonctions d'un système d'exploitation pour la commande de processus en temps réel sont les suivantes:

1- l'action sur les dispositifs externes (convertisseurs, lecteurs de capteurs, contrôleurs de vannes, etc.);

2- la prise en compte du temps réel, en fournissant toute réponse dans un temps minimum;

3- la réaction aux évènements extérieurs, avec le minimum d'interventions humaines;

4- la gestion fiable des informations permettant un fonctionnement, même en cas de défaillances matérielles.

8. Exemples de quelques RTOS dans l'industrie :

8.1. RTEMS (Real-Time Executive for Multiprocessor Systems) est un RTOS libre pour les systèmes embarqués (http://www.rtems.com). Il permet de développer des programmes dans lesquels le temps de réponse et la réactivité sont des contraintes fortes (temps réel dur). Il a été porté sur de nombreux processeurs (ARM, i386, m68k, MIPS, PowerPC) et est très complet.

8.1.1. Ses Caractéristiques :

1. Conforme à la norme POSIX1003.1b (API).
2. Système multitâches incluant les multi-activités (threads).
3. Supporte les systèmes multiprocesseurs homogènes ou non.
4. Supporte le pilotage par événement.
5. Ordonnancement préemptif basé priorités.
6. Supporte l'ordonnancement RM (rate monotonic scheduling).
7. Communication et synchronisation entre tâches.
8. Héritage de priorités.
9. Gestion des réponses aux interruptions.
10. Supporte l'allocation dynamique de la mémoire.
11. Pile TCP/IP.

8.2. VxWorks : est aujourd'hui l'exécutif temps réel le plus utilisé dans l'industrie. VxWorks est fiable, généralement utilisé dans les systèmes embarqués et porté sur un nombre important de processeurs (PowerPC, 68K, ColdFire, MCORE, 80x86, Pentium, i960, ARM, StrongARM, MIPS, SPARC, NECV8xx,…).

Un point fort de VxWorks a été le support réseau (sockets, NFS,...) dès son apparition. VxWorks est également conforme à POSIX1003.1b.

8.2.1. Ses Caractéristiques :

1. Noyau TR multitâches préemptif avec ordonnancement round robin.
2. Faible temps de latence.
3. Mémoire protégée permet d'isoler les applications utilisateurs du noyau.

4. Support des multiprocesseurs.
5. Communication inter-tâches.
6. Implémente l'exclusion mutuelle par sémaphores.
7. Héritage de priorités.
8. Gestion de file de messages locaux et distribuées.
9. Système de fichiers.
10. Pile TCP/IP IPV.
11. Latence moyenne de 1.7 ms, latence max de 6.8 ms sur Pentium 200 MHz (http://www.windriver.com).

8.3. QNX : est un RTOS adapté aux applications critiques. Développé par QNX Software Systems, sa structure est de type Unix et il est compatible POSIX.

On le retrouve entre autres dans le développement de logiciels, le contrôle de robots industriels et les ordinateurs embarqués. Son micro-noyau Neutrino peu gourmand en ressource confère à QNX des capacités temps-réel très performantes avec un temps de latence très faible (0.55 µsec sur un Pentium III). Il dispose d'une interface graphique nommée Photon (version 4.x). Le temps de commutation de tâches est très faible (<1µsec).

8.3.1. Ses Caractéristiques :

1. Supporte la norme POSIX 1003.1b.
2. Micronoyau Neutrino.
3. Zone mémoire protégée (Run time Memory protection).
4. Multitâches préemptif.
5. Micro-GUI Photon.
6. Supporte les processeurs x86, SMP jusqu'à 8 processeurs.
7. Système de fichiers.
8. version actuelle : 6.4 (http://www.qnx.com/).

8.4. RTLinux : se présente sous la forme d'un micronoyau prenant place entre la machine réelle (matériel) et le noyau Linux. Il intègre un gestionnaire de tâches (qui dans la pratique est un module). Le noyau de Linux partage le temps processeur avec les autres tâches temps réel et devient une tâche du RTOS.

Il est utilisé pour des produits sensibles dans le domaine des télécommunications ou des applications militaires. RTLinux propose des systèmes de communication de type FIFO ou mémoire partagée entre les tâches temps réel (dans l'espace noyau) et l'espace utilisateur dans lequel s'exécutent habituellement les applications. Initialement produit libre, il est maintenant devenu un produit commercialisé par la société FSMLabs (http://www.fsmlabs.com).

8.4.1. Ses Caractéristiques :

1. Système préemptif, ordonnanceur à priorités fixes.
2. Supporte plusieurs processeurs (SMP).
3. Supporte les architectures x86, PPC Alpha (DEC), MIPS, StrongARM, …
4. Autorise la communication et synchronisation IPC (**I**nter **P**rocess **C**ommunication : tubes, files de messages, sémaphores, mémoire partagée,…).
5. Communication entre processus utilisateurs et noyau (tâches temps réel) par FIFO, mémoire partagée ou signaux.
6. Noyau Linux est alors une tâche de faible priorité (idle task).

Chapitre 03 :

Gestion des tâches / Ordonnancement

(Scheduling)

1. Définitions :

- **Tâche/Processus (Thread/Job/Task)** : Entité d'exécution. Instance dynamique d'un programme exécutable.
- **Tâches périodiques :** déclenchées régulièrement car leurs caractéristiques sont connues à l'avance.
- **Tâches apériodiques :** déclenchées par un évènement avec des caractéristiques partiellement non-connues à l'avance.
- **Tâches indépendantes** : ne présentent pas de relation de précédence et ne partagent pas des ressources critiques.
- **Tâches dépendantes** : suivant ou précédant d'autres tâches parce qu'elles se synchronisent ou communiquent entre elles.
- **Séquence valide :** si toutes les tâches respectent leur CT.

2. Paramètres de base d'une tâche :

Une tâche (généralement périodique) se caractérise par les paramètres temporels suivants (avec $0 \leq C \leq D \leq P$) :

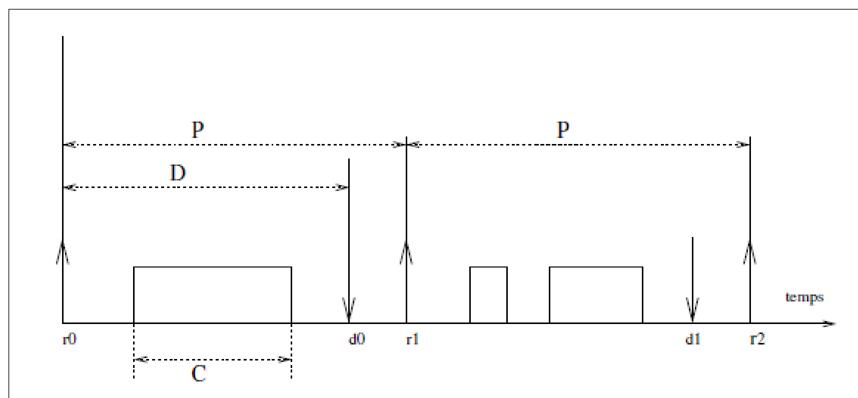


Figure 3.1. Diagramme temporel d'une tâche.

- r_0 : date de réveil, le moment de déclenchement de la 1ere requête d'exécution.
- C : durée max d'exécution,
- c_i : temps de calcul au pire cas (WCET: Worst Case Execution Time),
- D : délai critique (délai maximal acceptable pour son exécution),

- P : période,
- r_k : date de réveil de la k^{eme} requête de la tâche:
- $r_k = r_0 + k*P$, représentée par ↑
- d_k : échéance de la k^{eme} requête de la tâche (à contraintes strictes):
- $d_k = r_k + D$, représentée par ↓
- une tâche périodique à échéance sur requête ⇒ $D = P$, représentée par ↕.

3. Différents états d'une tâche:

- **Élue (courante) :** la tâche est en service, c.-à-d., en cours d'exécution,
- **Prête :** si la tâche dispose de toutes les ressources nécessaires à son exécution à l'exception du processeur,
- **Passive (en attente)** : si la tâche ne dispose pas de toutes les ressources nécessaires à son exécution, ou elle attente qu'un événement se produit pour pouvoir continuer,
- **Inexistante** : l'ordinateur ne sait pas que la tâche est existé (elle est en disque),
- **Bloquée (Existante**) : l'ordinateur sait que la tâche est existé mais elle n'a aucune priorité (hors file d'attente).

Les transitions entre les états prouvent être schématisées comme suit:

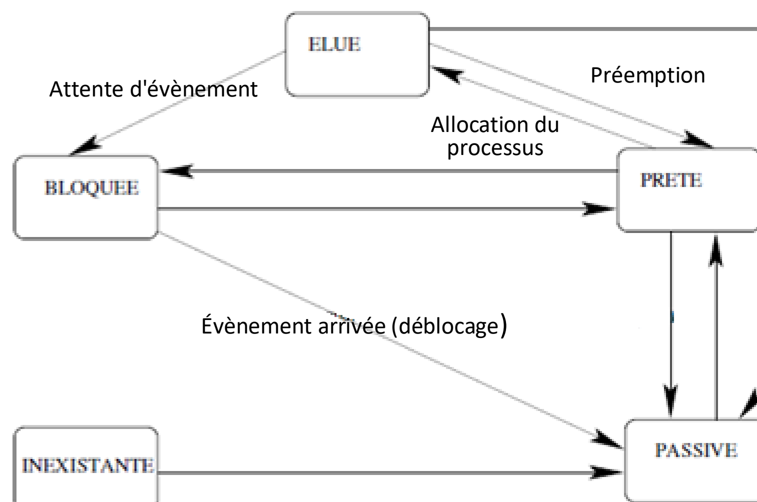


Figure 3.2. Transitions entre les états.

Remarque : Le dépassement d'une échéance est appelée faute temporelle.

4. Ordonnancement temps réel :

Planification de l'exécution des tâches/requêtes de façon à respecter les contraintes temporelles. Il détermine l'ordre d'allocation du processeur. L'ordonnancement a pour deux objectifs majeurs :

- *en fonctionnement normal* : respecter les contraintes temporelles pour toutes les requêtes.
- *en fonctionnement anormal* (*surcharge ou présence d'incidents → tâches supplémentaires suite à des anomalies : alarmes, fautes temporelles...*) : atténuer les effets des surcharges et maintenir un état cohérent et sécuritaire par l'exécution d'au moins les requêtes vitales.

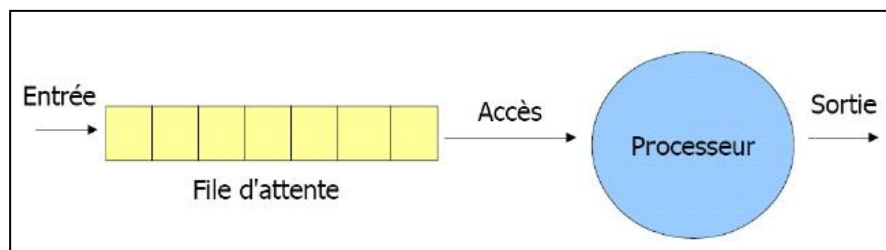


Figure 3.3. Ordonnancement des taches.

Le but de l'ordonnancement est de :

- Valider a priori la possibilité d'exécuter l'application en respectant ces contraintes.
- Permettre le respect de ces contraintes, lorsque l'exécution se produit dans un mode courant.
- En régime anormal : l'ordonnancement doit offrir une tolérance aux surcharges et permettre une exécution dégradée mais sécuritaire pour le procédé.

5. Typologie des algorithmes d'ordonnancement :

5.1. Ordonnanceur hors ligne (Offline) : l'algorithme connue la séquence complète de planification des tâches avant l'exécution.

- Avantage :
  - Très efficace,
  - Il convient les tâches périodiques.

- Inconvénient :
    - statique,
    - rigide,
    - ne s'adapte pas au changement de l'environnement.

5.2. Ordonnanceur en ligne (Online): capable à tout instant de l'exécution d'une application de choisir la prochaine tâche à ordonnancer, en utilisant les informations des tâches déclenchées à cet instant.

- Avantage :
    - dynamique,
    - permet l'arrivée imprévisible des tâches,
    - autorise la construction progressive de la séquence d'ordonnancement,
    - convient les tâches apériodiques.
- Inconvénient :
    - offre des solutions moins bonnes (car elle utilise moins d'informations),
    - entraîne des surcoûts de mise en œuvre.

5.3. Ordonnanceur préemptif (preemptive) : une tâche élue peut perdre le processeur au profit d'une tâche plus prioritaire (ou une interruption).

- Avantage :
    - dynamique,
    - s'adapte au surcharge ou changement de l'environnement.
- Inconvénient :
    - utilisable si toutes les tâches sont préemptibles.

5.4. Ordonnanceur non préemptif (non-preemptive) : une fois élue, la tâche ne doit plus être interrompue jusqu'à la fin de la requête.

- Avantage :
    - Rigide,
    - Facile à concevoir.
- Inconvénients :
    - ne s'adapte pas au surcharge ou changement de l'environnement.
    - absence de la notion interruption.

6. Critères d'ordonnancement :

- À maximiser :
    - le pourcentage d'utilisation du processeur.
    - le débit (nombre moyen de processus traités par unité de temps).
- À minimiser :
    - *le temps de réponse* : le temps entre une demande et la réponse.
    - *le temps de rotation* : durée moyenne qu'il faut pour qu'un processus s'exécute.
    - *le temps d'attente* : durée moyenne qu'un processus passe à attendre.

7. Performances des algorithmes d'ordonnancement :

- *Temps de rotation* = temps fin d'exécution - temps d'arrivée.
- *Temps d'attente* = temps de rotation - Durée d'exécution.
- *Temps moyen d'attente* = ∑ (temps d'attente) / nombre de processus.
- *Rendement* = ∑ (temps d'exécution) / nombre de processus.

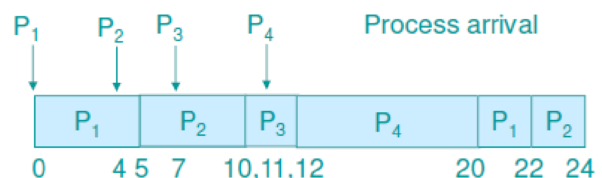
Exemple :

Utilisation de l'UCT:

- 100%

Temps de réponse (P3, P2):

- P3: 3 = (10 – 7)
- P2: 1 = (5 – 4)



Débit :

- 4/24

Temps de rotation (P3, P2):

- P3: 5 = (12 – 7)
- P2: 20 = (24 – 4)

Temps d'attente (P2):

- P2: 13 = (5 – 4) + (22 – 10)

8. Algorithmes d'ordonnancement les plus utilisés :

8.1. Ordonnancement FCFS (First Come First Served):

Dans cet algorithme ; connu sous le nom FIFO (First In, First Out), en français (premier arrivé, premier servi) ; les processus sont rangés dans la file d'attente des processus prêts selon leur ordre d'arrivée.

Exemple de FCFS :

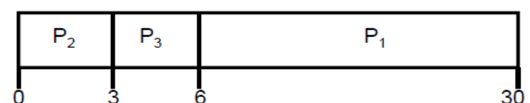
Processus	durée d'exécution
P1	24
P2	3
P3	3

Si les processus arrivent au temps 0 dans l'ordre: P1, P2, P3. Le diagramme Gantt est:

- Temps d'attente pour P1= 0, P2= 24, P3= 27
- Temps moyen d'attente: (0 + 24 + 27)/3 = 17
- Utilisation UCT = 100%
- Débit = nbre de processus / temps = 3/30 = 0,1 (3 processus complétés en 30 unités de temps)
- Temps moyen de rotation: (24+27+30)/3 = 27

Si les mêmes processus arrivent à 0 mais dans l'ordre *P2, P3, P1*. Le diagramme de Gantt est :

- Temps d'attente pour P1 = 6, P2 = 0, P3 = 3
- Temps moyen d'attente : (0 + 3 + 6)/3 = 3
- Utilisation UCT = 100%
- Débit = nbre de processus / temps = 3/30 = 0,1
- Temps moyen de rotation : (3+6+30)/3 = 13



- Avantages de FCFS :
  - Simple.
- Inconvénients :
  - le temps moyen d'attente peut varier grandement,
  - les tâches de faible temps d'exécution sont pénalisées si elles sont précédées dans la file par une tâche de longue durée.

***Exercice* :** répéter les calculs si :

- P2 arrive à temps 0,
- P1 arrive à temps 2,
- P3 arrive à temps 5.

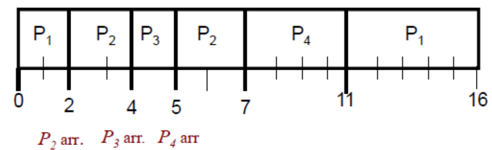
8.2. Ordonnancement SJF (Shortest Job First) (Plus Cours d'abord) :

Le processus le plus court part le premier.

8.2.1. SJF avec préemption: si un processus qui dure moins que le restant du processus courant se présente plus tard, l'UCT est donnée à ce nouveau processus (**SRTF** : shortest remaining-time first).

Exemple de SJF avec préemption (SRTF) :

Processus	Temps d'arrivée	durée d'exécution
P1	0	7
P2	2	4
P3	4	1
P4	5	4



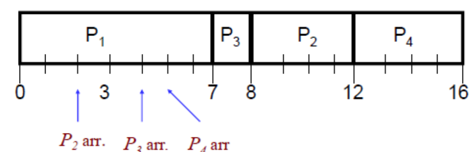
Temps moyen d`attente = (9 + 1 + 0 +2)/4 = 3

- *P1* attend de 2 à 11,
- *P2* de 4 à 5,

***8.2.2. SJF sans préemption** :* on permet au processus courant de terminer son cycle.

Exemple de SJF sans préemption :

Processus	Temps d'arrivée	durée d'exécution
P1	0	7
P2	2	4
P3	4	1
P4	5	4



Temps moyen d`attente = (0 + 6 + 3 + 7)/4 = 4

➢ Avantages de SJF :

❖ Remédie à l'inconvénient de FCFS,

❖ Minimise le temps de réponse moyen.

➢ Inconvénients :

❖ Pénalise les travaux longs,

❖ Elle impose d'estimer les durées d'exécution des tâches a priori (connues difficilement).

8.3. Ordonnancement basé sur les priorités (PRIO) :

Affectation d'une priorité à chaque processus, les processus sont rangés dans la file d'attente des processus prêts par ordre de priorité.

Problème de la famine : les processus moins prioritaires n'arrivent pas à s'exécuter pendant un temps indéterminé.

Solution : modifier/incrémenter graduellement la priorité d'un processus en fonction de son âge et de son historique d'exécution (*vieillissement*).

Pour les processus de même priorité, on peut choisir :

- FCFS,
- Tourniquet (avec la préemption).

Exemple de PRIO : supposons que les 4 processus arrivent au même temps (instant 0)

Processus	PRIO	durée d'exécution (*ms*)
P1	3	10
P2	1	1
P3	4	5
P4	2	3

Temps moyen d`attente = (4 + 0 + 14 + 1)/4 = 4,75*ms*

➢ Avantages de PRIO :

❖ adapté aux systèmes temps réels durs (strictes),

❖ répond efficacement aux interruptions et aux alertes.

- <u>Inconvénients :</u>
  - problème de la famine.

Remarque : l'algorithme d'ordonnancement PRIO peut avoir plusieurs variantes en fonction de la nature des priorités affectées aux tâches indépendantes périodiques. On peut citer quelques variantes :

8.3.1. Algorithme à Priorités fixes (RM : Rate Monotonic) : la tâche de plus petite période est la plus prioritaire. L'utilisation de cet algorithme est valable uniquement si les tâches sont à échéance sur requête (P=D).

- Test d'acceptabilité (condition suffisante): les tâches respectent leur CT si:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n\left(2^{1/n} - 1\right)$$

8.3.2. Algorithme Inverse Deadline (ID) ou Deadline Monotonic (DM) : La tâche la plus prioritaire est celle de plus petit délai critique.

- Test d'acceptabilité (condition suffisante): les tâches respectent leur CT si:

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n\left(2^{1/n} - 1\right)$$

8.3.3. Algorithme à priorité variable ou dynamique (EDF : Earliest Deadline First) : à chaque instant (i.e. à chaque réveil de tâche), la priorité maximale est donnée à la tâche dont l'échéance est la plus proche.

- Période d'étude $= [0\, , PPCM(P_i)]$ (départ simultané des tâches)
- Test d'ordonnançabilité :

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1 \text{ (Condition nécessaire)}$$

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq 1 \text{ (Condition suffisante)}$$

8.3.4. Algorithme à priorité variable (LLF : Least Laxity First) : la tâche à qui il reste le moins de marge s'exécute d'abord. Autrement dit, la priorité maximale est donnée à la tâche qui a la plus petite laxité résiduelle *L(t)*.

- Laxité dynamique résiduelle $L(t) = D(t) - C(t) = D + r - t - C(t)$
    - retard maximum pour reprendre l'exécution d'une tâche.

Exercice :

Considérons le système temps réel suivant :

Tâche	Temps de réveil (r)	Durée d'exécution (C)	Délai critique (D)	Période (P)
T1	0	3	7	20
T2	0	2	4	5
T3	0	2	9	10

1- Calculer la période d'étude de ce système.

2- Vérifier l'acceptabilité d'ordonnancement de ce système et tracer le diagramme de Gantt sur une période d'étude en utilisant **RM** puis **DM**.

3- Vérifier l'ordonnançabilité de ce système et tracer le diagramme de Gantt correspondant en utilisant **EDF** (D_3 = 8 au lieu de 9).

4- Étudier et tracer l'ordonnancement généré par **LLF** (D_3 = 8).

Solution :

On a bien n=3

1- Période d'étude = [0, PPCM (20, 5, 10)] = [0, 20].

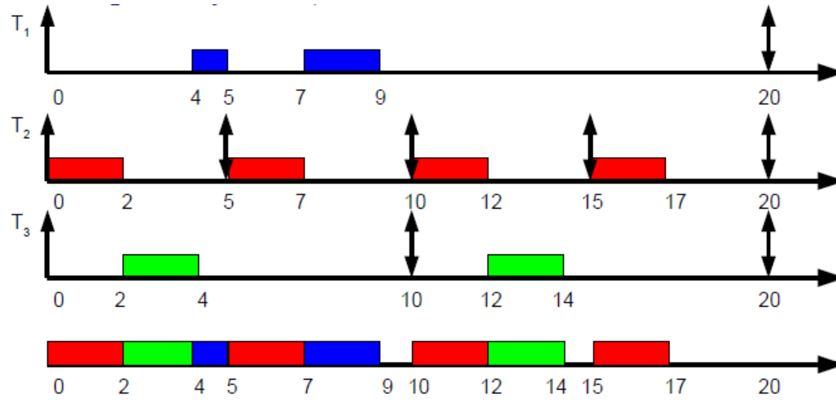
2- Algorithme de **RM** : ($Prio_{T2} > Prio_{T3} > Prio_{T1}$)

Remarque : ici les tâches sont à échéance sur requête ⇒ $D = P$.

$$\sum_{i=1}^{n} \frac{C_i}{P_i} = \frac{3}{20} + \frac{2}{5} + \frac{2}{10} = 0.15 + 0.4 + 0.2 = 0.75$$

$$n\left(2^{1/n} - 1\right) = 3\left(2^{1/3} - 1\right) = 0.779$$

Donc, les 3 tâches respectent leur CT car la condition suffisante est vérifiée $0.75 \leq 0.779$.

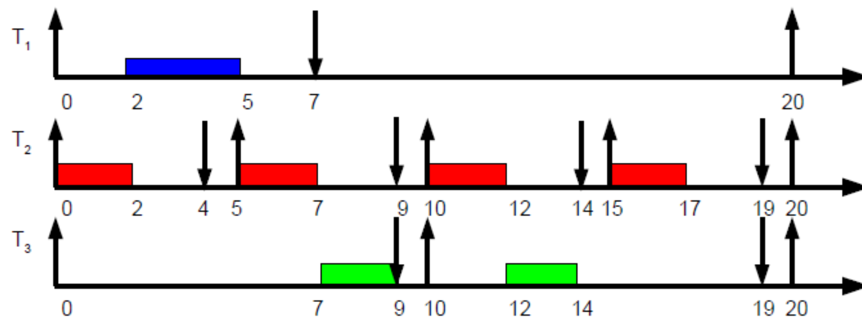


<u>Diagramme de Gantt généré par RM</u>

Algorithme de **DM** : ($\text{Prio}_{T2} > \text{Prio}_{T1} > \text{Prio}_{T3}$)

$$\sum_{i=1}^{n} \frac{C_i}{D_i} = \frac{3}{7} + \frac{2}{4} + \frac{2}{9} = 1.14 \geq 0.779$$

Donc, la condition suffisante n'est pas vérifiée. Cependant, le diagramme construit sur la période d'étude montre qu'elles sont ordonnançables sans faute temporelle.



<u>Diagramme de Gantt généré par DM</u>

3- Algorithme **EDF** :

La condition nécessaire est vérifiée mais la condition suffisante n'est pas vérifiée.

- À t=0 : au réveil des 3 tâches, T2 prioritaire, elle s'exécute pendant 2 unités.
- À t=2, T2 termine. C'est T1 la plus prioritaire. Elle s'exécute pendant 3 unités de temps.
- À t=5, T1 se termine et T2 se réveille de nouveau (car sa période 5), mais c'est T3 qui est cette fois la plus prioritaire car son échéance est 8, elle s'exécute pendant 2 unités.

- Ici, on voit que les priorités des tâches varient dans le temps : par exemple, à t=0, c'est T2 la plus prioritaire (que T1 et T3), mais à t=5, c'est T3 la plus prioritaire.

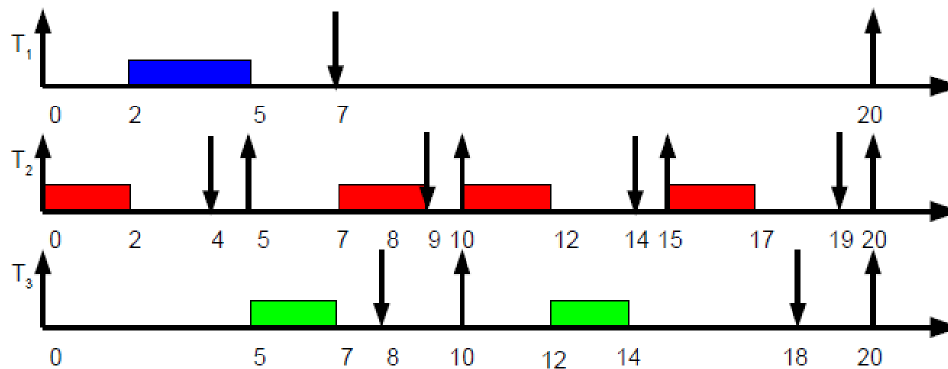


Diagramme de Gantt généré par EDF

4- Algorithme **LLF :**

- À t=0, les 3 tâches sont réveillées :

  Laxité de T1 = 7 - 3 = 4

  Laxité de T2 = 4 - 2 = 2

  Laxité de T3 = 8 - 2 = 6

  C'est donc T2 (la plus prioritaire) qui s'exécute.
- À t=1, L1 = 7-1-3= 3, L2= 4-1-1= 2, L3 = 8-1-2= 5 ⇒ T2 qui s'exécute.
- À t=2, T1 qui s'exécute (sa laxité [7-2-3] est plus petite que celle de T3 [8-2-2]).
- À t=3, L1 = 2, L3 = 3 ⇒ T1 qui s'exécute.
- À t=4, L1 = 2, L3 = 2 ⇒ T1 qui s'exécute (T1 est préférable, le retard de T3 est toléré tant quand son échéance est toujours respectée).
- À t=5, T2 se réveille de nouveau.

  L2 = 9-5-2 = 2 (2ème échéance - temps courant - durée exécution),

  L3 = 8-5-2 = 1. Donc T3 qui s'exécuter.
- À t=6, L2 = 1, L3 = 1 ⇒ T1 qui s'exécute.
- À t=7, L2 = 9-7-2 = 0 ⇒ T2 qui s'exécute.

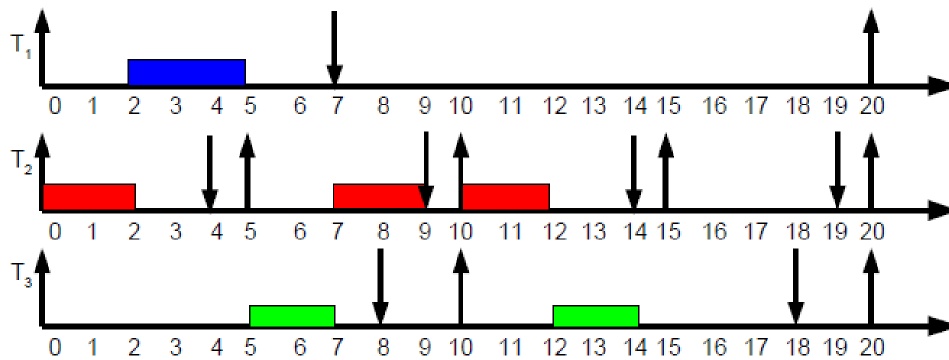


Diagramme de Gantt généré par LLF

8.4. Ordonnancement par Tourniquet/Tour de rôle (RR : Round Robin) :

Le processeur est alloué successivement aux différents processus pour une tranche de temps fixe ***Q*** appelé ***Quantum*** (tranche du temps).

L'efficacité de cet ordonnanceur dépend principalement de la valeur du quantum ***Q*** :

- ***Q*** assez petit augmente le nombre de commutation.
- ***Q*** assez grand augmente le temps de réponse du système → FCFS.

➢ Avantages de RR :

- ❖ Adapté aux systèmes temps partagé,
- ❖ garantit que tous processus sont servis au bout d'un temps fini,
- ❖ méthode préemptive,
- ❖ évite *la famine*.

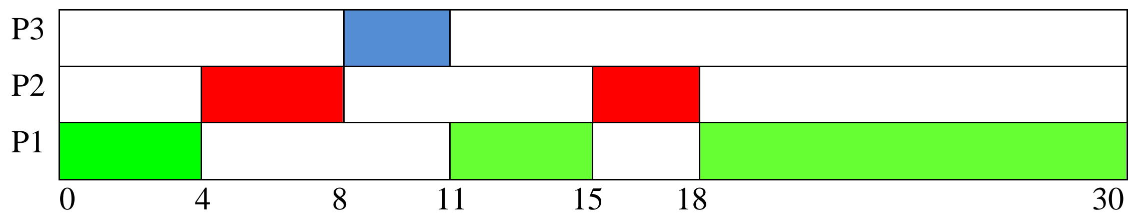
➢ Inconvénients :

- ❖ ***Q*** optimal est difficilement estimé.

Exemple de RR :

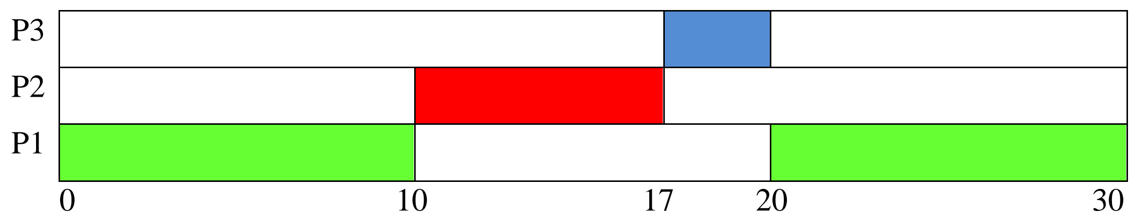
Processus	PRIO	Durée d'exécution (*ms*)
P1	1	20
P2	2	7
P3	3	3

A- avec *Quantum* = 4



- Temps moyen d'attente : (8 + 11 + 8)/3 = 9
- Temps moyen de rotation : (30+18+11)/3 = 19,66

A- avec *Quantum* = 10



- Temps moyen d'attente : (10 + 10 + 17)/3 = 12,33
- Temps moyen de rotation : (30+17+20)/3 = 22,33

Chapitre 04 :

Gestion de la mémoire

(Memory management)

1. Définitions :

- **Mémoire vive (centrale ou interne, RAM) :** un type de mémoire qui permet de stocker des informations provisoires. Son avantage majeur est sa capacité de lecture très rapide par rapport au disque dur et qui permet une utilisation fluide de l'ordinateur (son but est d'y accéder rapidement et provisoirement). Quand le système s'éteint, la RAM se vide.
- **Mémoire de masse (physique ou interne):** est une mémoire de grande capacité qui peut être lue et écrite par un système (Bande magnétique, disque dur, disque optique (CD, DVD, Blu-ray), disque magnéto-optique et mémoire flash).
- **Mémoire d'échange (SWAP) :** est une partie de la mémoire de masse d'un ordinateur utilisée par le système d'exploitation pour stocker des données qui, du point de vue des applications, se trouvent en mémoire vive. Son intérêt est de simuler sans coût une mémoire vive plus grande, avec une perte de vitesse limitée. C'est utile lorsque la mémoire vive est pleine.
- **Mémoire virtuelle :** Espace du disque dur interne d'un ordinateur qui vient seconder la mémoire vive, Elle se concrétise par un fichier d'échanges (fichier swap), lequel contient les données non sollicités constamment. La mémoire virtuelle, comme son nom l'indique, sert à augmenter artificiellement la mémoire vive.
- **Mémoire morte (ROM) :** Type de mémoire dont le contenu est accessible en lecture et non en écriture. Elle conserve les données en l'absence de courant électrique. Elle stocke les informations nécessaires au démarrage d'un système (BIOS, des équipements embarqués, des tables de constantes,...).
- **Mémoire flash :** est une mémoire de masse à semi-conducteurs réinscriptible, c'est-à-dire une mémoire possédant les caractéristiques d'une mémoire vive mais dont les données ne disparaissent pas lors d'une mise hors tension.
- **Mémoire cache :** est une mémoire plus rapide et plus proche du matériel informatique (processeur, disque dur). Son rôle est de stocker les informations les plus fréquemment utilisées par les logiciels et applications lorsqu'ils sont actifs. C'est

cet accès direct qui détermine les performances d'un programme car il économise des échanges incessants entre le processeur et la mémoire vive.

2. Rôle d’un gestionnaire de mémoire :

- création d’un processus.
- activation/désactivation d’un processus.
- suppression d’un processus.
- partage la mémoire disponible entre les processus (protection).
- cartographie la mémoire.
- alloue/dés-alloue de la mémoire dynamiquement pour les besoin d’un processus.
- assure la cohérence de la mémoire.
- optimise l’utilisation de la mémoire.

3. Organisation de la mémoire :

Une application ne peut s'exécuter que si ses instructions et ses données sont en mémoire physique (vive). Donc, si on désire exécuter plusieurs programmes simultanément dans un système, il faudra que tous ces programmes soient chargés dans la mémoire. L’OS devra donc allouer à chaque programme une zone de la mémoire où celui-ci sera chargé.

Le système prend à sa charge la gestion de la mémoire principale:

- il assure sa protection,
- il fournit l'information qui permet à chaque programme de s'exécuter (à quel endroit dans la mémoire ? → l'adresse et la capacité).

Le but d'une bonne gestion de la mémoire est d'augmenter le rendement global du système.

3.1. Monoprogrammation :

Le rôle le plus simple d'un gestionnaire de mémoire est d'exécuter un seul programme à la fois, en partageant la mémoire entre le programme et le système d'exploitation.

Figure 4.1. Architecture d'une mémoire physique avec monoprogrammation.

Un seul processus peut s'exécuter à la fois. Dès que l'utilisateur tape une commande, le système d'exploitation copie le programme demandé depuis le disque vers la mémoire et l'exécute.

3.2. Multiprogrammation :

Pour supporter la multiprogrammation et donc l'existence de plusieurs processus en mémoire principale, on peut distinguer deux grandes stratégies d'allocation mémoire: l'allocation de partitions contiguës et l'allocation de partitions non-contiguës.

3.2.1. Multiprogrammation et partitions multiples contigües :

L'espace mémoire est divisé en partitions. Chaque partition peut être allouée à un programme. La taille des partitions et donc leur nombre peuvent être fixes ou variables.

A. Partitions contiguës fixes: la mémoire est divisée en *n* partitions de tailles fixes (si possible inégales). Ce partitionnement se fait au démarrage du système. Quand un processus arrive, il peut être placé dans la file d'attente.

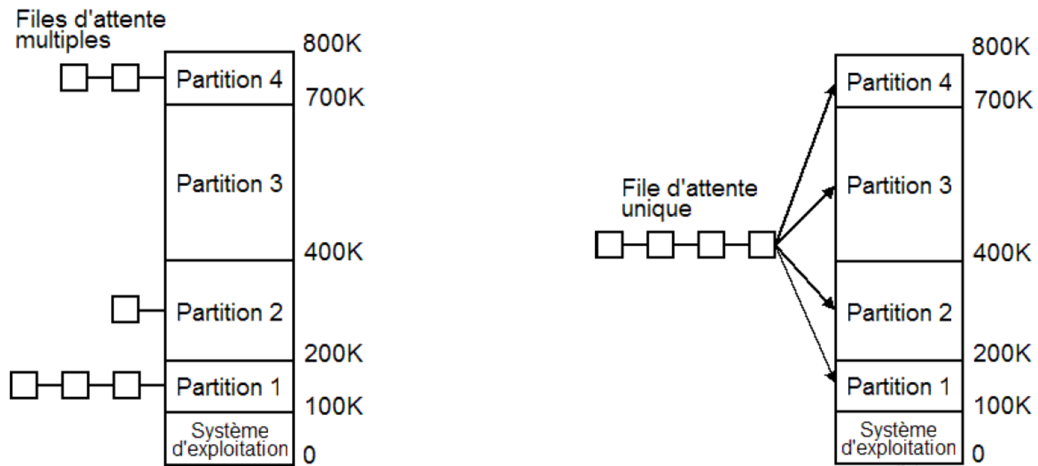
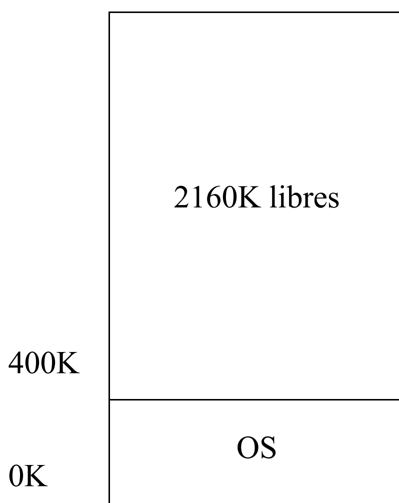


Figure 4.2. Files d'attente séparées et communes.

L'inconvénient des files d'attente séparées se présente lorsque la grande partition est vide tandis que celle d'une petite partition est pleine.

La solution → *file d'attente commune* : Dès qu'une partition devient libre (**trou**), toute tâche placée en tête de file d'attente et dont la taille convient peut être chargée dans cette partition vide et exécutée.

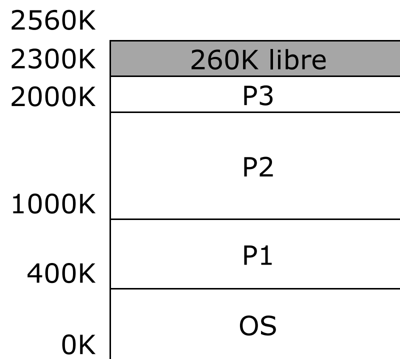
Exemple : L'état de la mémoire d'un système est décrit par la figure suivante. Le système a une file de processus décrit par le tableau suivant (État de la mémoire):

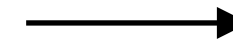


Processus	Mémoire	Temps
P1	600 K	10
P2	1000 K	5
P3	300 K	20
P4	700 K	8
P5	500 K	15

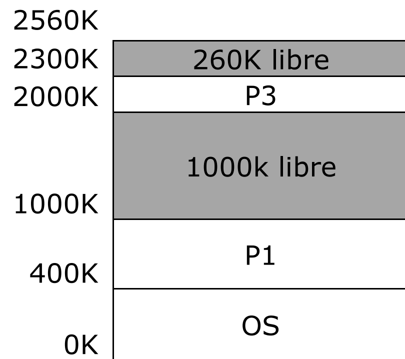
Les figures suivantes montre les différents états successifs de la mémoire après les entrées et les sorties de processus.

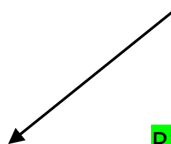
P1, P2 et P3 entrent:



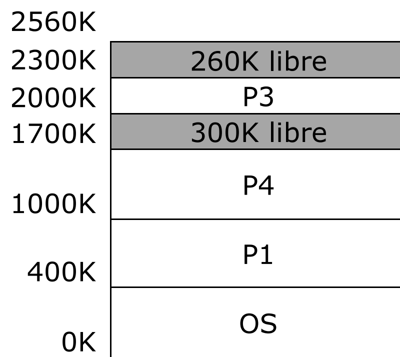


P2 termine:

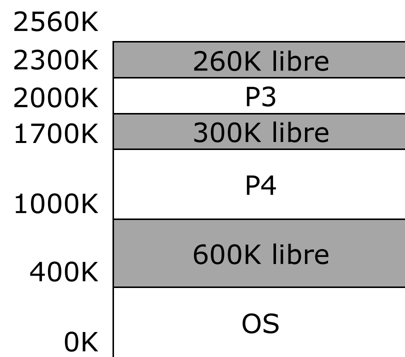


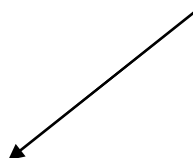


P4 entre:

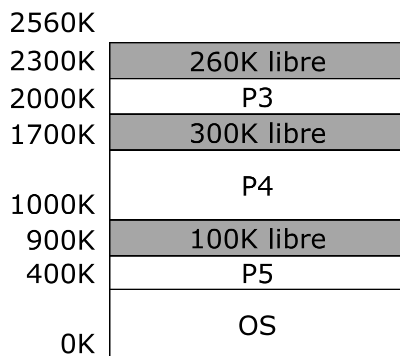


P1 termine:





P5 entre:



B. Partitions contiguës dynamiques

Dans cette stratégie la mémoire est partitionnée dynamiquement selon la demande. Lorsqu'un processus se termine sa partition est récupérée pour être réutilisée (complètement ou partiellement) par d'autres processus. Le gestionnaire de la mémoire doit garder trace des partitions occupées et/ou des partitions libres. On distingue les stratégies de placement suivantes:

- **Stratégie du premier qui convient (FF : First Fit):** On alloue au processus la première partition suffisamment grande.
- **Stratégie du meilleur qui convient (BF : Best Fit):** On alloue la plus petite partition dont la taille est au moins égale à celle du processus en attente.
- **Stratégie du pire qui convient (WF : Worst Fit):** On alloue au processus la partition de plus grande taille.

Remarque: les algorithmes BF et WF nécessitent le tri des partitions libres par ordre des adresses croissantes.

Exemple:

L'état de la mémoire d'un système est décrit par la figure suivante. On suppose qu'un processus P4 demande un espace mémoire de 80K.

Adresse	Contenu
2100K	
2000K	100K libre 3
1400K	P3
	400K libre 2
1000K	
900K	P2
700K	200K libre 1
400K	P1
	OS
0K	

En fonction de l'algorithme choisi, P4 occupe :

- Algorithme FF : Partition 1.
- Algorithme BF : Partition 3.
- Algorithme WF : Partition 2.

Exercice:

Dans un système de gestion mémoire à partitions variables, on constate que la liste des partitions libres "trous" est la suivante (dans l'ordre des adresses mémoire croissantes) :

10K 4K 20K 18K 7K 9K 12K 15K

On veut placer successivement des processus de volumes respectifs P1 (12K) P2 (10K) P3 (9K) dans la mémoire.

Indiquer, dans l'ordre des adresses croissantes, la nouvelle liste des trous en utilisant chacune des stratégies de placement : FF, BF, WF ?

Solution:

1) First Fit : utilisation de la première zone libre :

État initial	10K	4K	20K	18K	7K	9K	12K	15K
Placement de P1(12K)	10K	4K	P1+8K	18K	7K	9K	12K	15K
Placement de P2(10K)	P2	4K	P1+8K	18K	7K	9K	12K	15K
Placement de P3(9K)	P2	4K	P1+8K	P3+9K	7K	9K	12K	15K

2) Best Fit : meilleur ajustement :

État initial	10K	4K	20K	18K	7K	9K	12K	15K
Placement de P1(12K)	10K	4K	20K	18K	7K	9K	P1	15K
Placement de P2(10K)	P2	4K	20K	18K	7K	9K	P1	15K
Placement de P3(9K)	P2	4K	20K	18K	7K	P3	P1	15K

3) Worst Fit : on prend le plus grand emplacement libre :

État initial	10K	4K	20K	18K	7K	9K	12K	15K
Placement de P1(12K)	10K	4K	P1+8K	18K	7K	9K	12K	15K
Placement de P2(10K)	10K	4K	P1+8K	P2+8K	7K	9K	12K	15K
Placement de P3(9K)	10K	4K	P1+8K	P2+8K	7K	9K	12K	P3+6K

C. Va et vient (Swap)

Parfois la mémoire principale est insuffisante pour maintenir tous les processus courants actifs : il faut alors conserver les processus supplémentaires sur un disque et les charger pour qu'ils s'exécutent dynamiquement.

La stratégie *swapping* consiste à considérer chaque processus dans son intégralité. Le programme en cours doit être sauvegardé sur disque avant le chargement en mémoire principale de son successeur pour exécution.

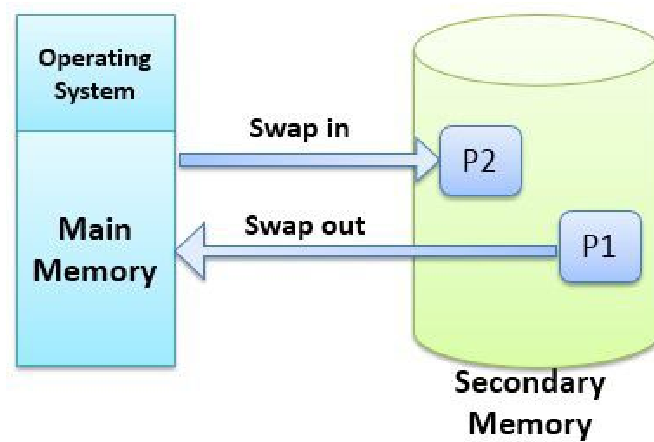


Figure 4.3. Principe de Swapping.

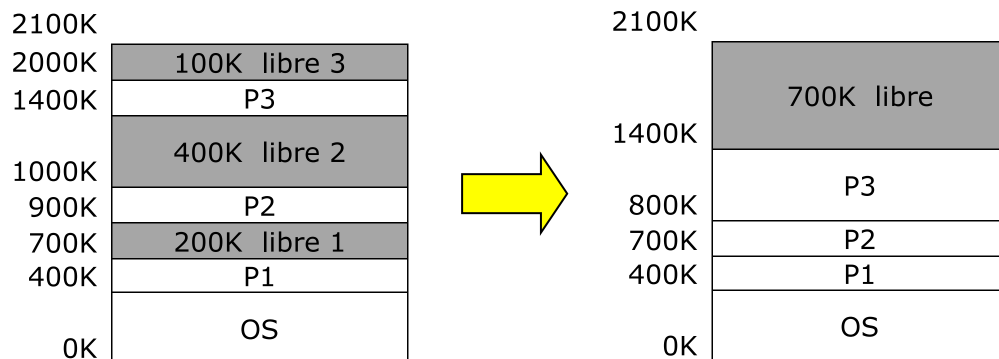
Problème de Fragmentation :

Les algorithmes d'allocation de la mémoire contiguë précédents, produisent la fragmentation de la mémoire. En effet, suite aux différentes entrées et sorties de processus, des fragments (fractions) séparées se forment dans la mémoire.

Si le processus P4 désire entrer dans le système en occupant 500K, il ne pourrait pas bien que l'espace total disponible est de 700 K.

Adresse	Contenu
2100K	
2000K	100K libre 3
1400K	P3
	400K libre 2
1000K	
900K	P2
700K	200K libre 1
400K	P1
0K	OS

Solution→ Défragmentation par compactage: le principe est de rassembler tous les trous en un seul bloc.



Inconvénient : L'opération de compactage est une opération très coûteuse pour l'OS.

3.2.2. Multiprogrammation et partitions multiples non contigües :

Nous avons vu que les techniques d'allocation associées conduisent à la fragmentation de la mémoire (de nombreuses petites zones libres inexploitables). Pour l'éviter, il faut pouvoir implanter un programme dans plusieurs zones non contiguës. Plusieurs techniques proposent d'allouer des espaces mémoire non-contiguës pour les processus : Pagination et Segmentation.

A. Pagination :

La pagination consiste à découper l'espace adressable, ou espace **virtuel**, en zones de taille fixe appelée **pages**. La mémoire réelle (RAM) est également découpée en **cases** (***frame*** en anglais) ayant la taille d'une page de sorte que chaque page peut être implantée dans n'importe quelle case de la mémoire réelle.

- Une adresse est divisée en deux parties :
  - un numéro de page *p* et
  - un déplacement à l'intérieur de la page *d*.
- La taille de la page (et donc de la case) est une puissance de 2 variant généralement entre 512 et 8192 octets selon les architectures.
- Si la taille de l'espace d'adressage logique est 2^m et la taille d'une page est 2^n :
  - Les *m-n* bits de poids fort d'une adresse paginée désignent le numéro de page,
  - Les *n* bits de poids faible désignent le déplacement dans le page.

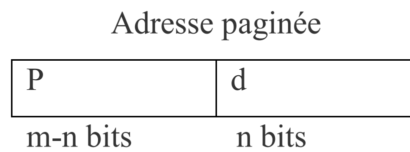


Table des pages : La traduction des adresses utilise une table des pages qui est située en mémoire centrale ou dans des registres. L'adresse réelle (*f,d*) d'un mot d'adresse virtuelle/logique (*p,d*) est obtenue en remplaçant le numéro de page *p* par le numéro de case *f* trouvé dans la $p^{ième}$ entrée.

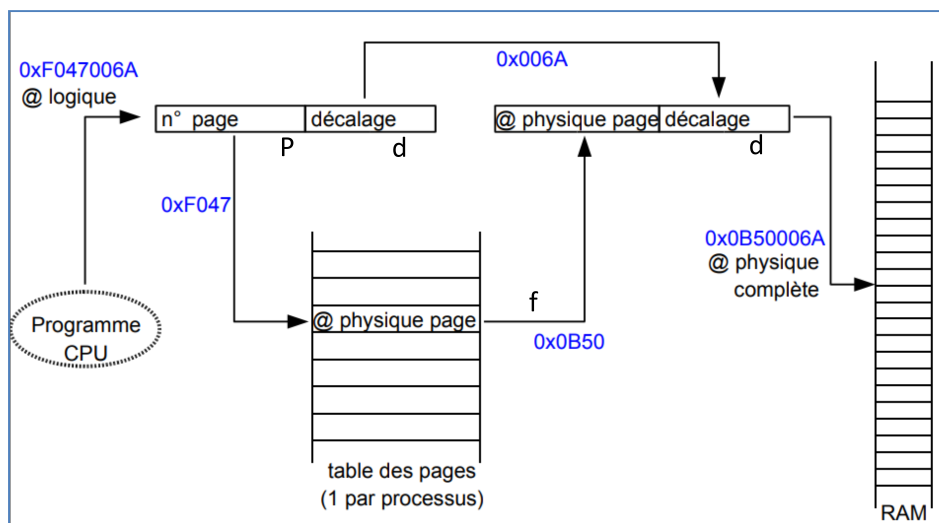


Figure 4.4. Traduction de l'adresse logique (virtuelle).

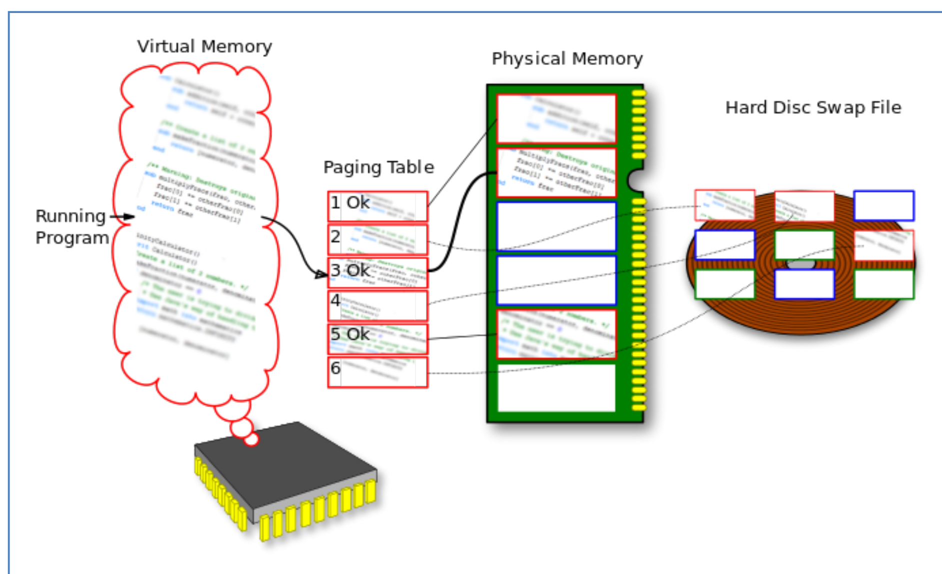


Figure 4.5. Principe de la pagination.

B. Segmentation :

La segmentation est analogue à la pagination sauf que la taille d'un segment est variable. L'espace d'adressage logique est divisé en un ensemble de segments. L'avantage de la segmentation par rapport à la pagination, est que les segments peuvent refléter une vision logique du programme. Par exemple chaque segment peut représenter un module de programme généré au moment de la compilation.

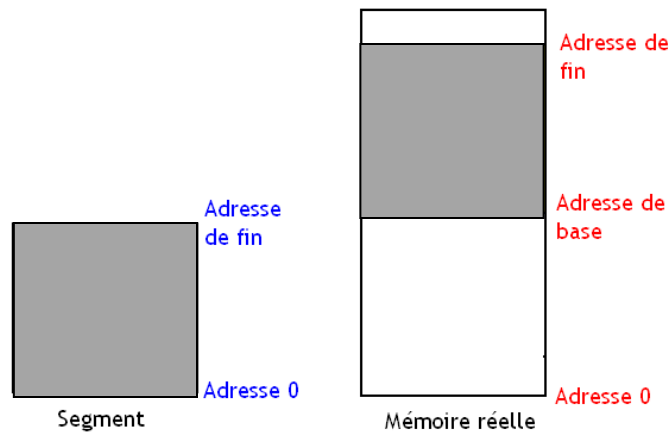


Figure 4.6. Principe de la segmentation.

Table de segments : la table de segment spécifie pour chaque segment deux valeurs :

- base : adresse début du segment en mémoire centrale,
- limite : taille du segment.

Une adresse logique (virtuelle) est dite segmentée. Elle comprend un numéro de segment (*s*) et un déplacement dans le segment (*d*). (*s*) est utilisé comme index dans la table de segments. La table de segments est généralement implantée par des registres rapides. Si $d >$ limite : alors erreur de débordement (famous Segmentation fault).

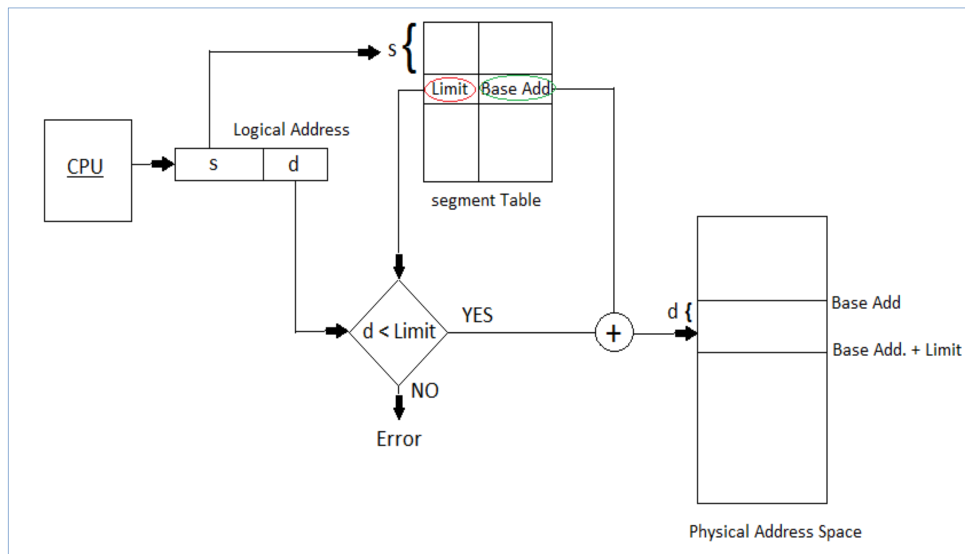


Figure 4.7. Traduction d'adresse d'une mémoire segmentée en utilisant la table des segments.

4. Gestion de la mémoire virtuelle:

La mémoire virtuelle est une technique autorisant l'exécution de processus pouvant ne pas être complètement en mémoire. La mémoire logique est plus grande que la mémoire physique.

→ Possibilité d'exécuter des programmes dont la taille est bien supérieure à celle de la mémoire physique.

→ Sur les ordinateurs actuels, l'espace mémoire virtuel projeté pour un processus peut atteindre plusieurs Go.

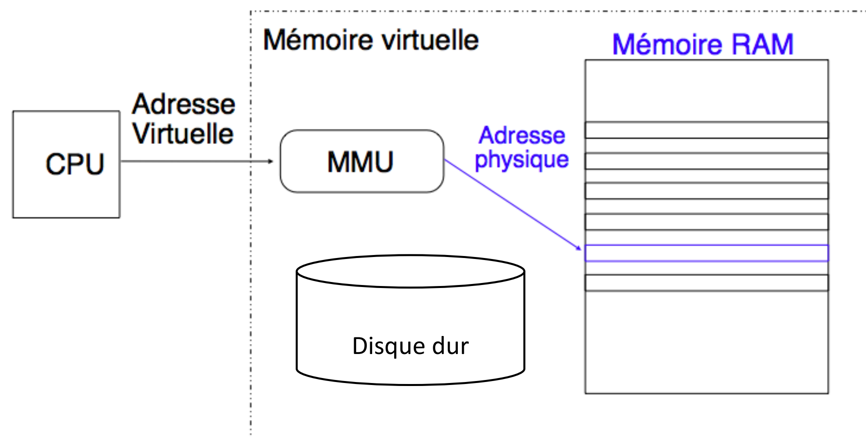


Figure 4.8. Principe de fonctionnement de la mémoire virtuelle.

L'allocation de mémoire consiste à concrétiser cette mémoire virtuelle par des supports physiques d'information tels que la mémoire principale (RAM), les disques magnétiques, etc. En dernier ressort, l'accès d'un processus à une information (de son espace virtuel) est concrétisé par l'accès d'un processeur physique à un emplacement de mémoire principale (RAM) adressable par ce processeur.

Principe: la taille de l'ensemble formé par le programme, les données et la pile peut dépasser la capacité disponible de mémoire physique. Le système d'exploitation conserve les parties de programme en cours d'utilisation dans la mémoire principale, et le reste sur le disque.

Dans un système paginé, les adresses générées par un programme sont appelées des adresses virtuelles et elles forment l'espace d'adressage virtuel. Ces adresses virtuelles ne vont pas directement sur le bus mémoire mais dans une unité de gestion mémoire (**MMU, Memory Management Unit**) qui fait correspondre les adresses virtuelles à des adresses physiques, en se basant sur une table de page.

Chapitre 05 :

Gestion du parallélisme et communication entre processus

(Parallelism and communication between processes)

1. Introduction :

La programmation Temps Réel fait souvent intervenir plusieurs tâches plus ou moins indépendantes qui constituent le système.

Il se pose alors un certain nombre de problèmes pour la réalisation d'une application TR.

On peut classifier ces problèmes en trois catégories :

1. **L'exclusion mutuelle :** accès concurrent à des ressources critiques.
2. **La synchronisation :** relation d'ordre entre les processus (tâches).
3. **La communication :** échange de données.

Donc, chaque application temps réel est souvent constituée de plusieurs tâches exécutées de façon concurrente :

- ces tâches ne sont pas indépendantes,
- besoin d'échanger des données,
- besoin de synchroniser les moments ou les données sont échangées.

2. Définitions :

- **Ressource :** Entité utilisée par un système informatisé pour un bon fonctionnement :
  - Physique : capteurs, actionneurs, périphériques ...
  - Logique : donnée dans une mémoire, code ...
- **Partage de ressources :** l’exécution en parallèle de plusieurs ressources en fournissant le même résultat qu’une exécution séquentielle (pas d’interférences).
- **Section critique (SC) :** est un ensemble d’instruction d’un programme qui peut engendrer des résultats imprévisibles lorsqu’elles sont exécutées simultanément par des processus différents (l’existence de section critique implique l’utilisation des variables partagées).
- **Ressource critique :** Accès concurrent par plusieurs processus à un objet (variable, table, fichier, mémoire, périphérique, …). Par exemple deux processus qui tentent d’envoyer simultanément un fichier sur l’imprimante ; le périphérique imprimante devient ressource critique pour eux.

- **Communication, coopération :** garantir que l'échange d'information entre tâches suit un protocole défini.

3. Exclusion mutuelle :

Le problème de l'exclusion mutuelle se pose lorsque plusieurs tâches demandent l'utilisation d'une ressource commune, alors qu'une seule tâche peut avoir accès à la ressource à la fois.
Pour réaliser une exclusion mutuelle qui est nécessaire pour résoudre le problème des accès concurrents, on admet que certaines contraintes doivent être respectées :

- **Le déroulement :** Le fait qu'un processus ne demande pas à entrer en SC ne doit pas empêcher un autre processus d'y entrer. Cette contrainte exclut les méthodes fondées sur un tour de rôle strict.

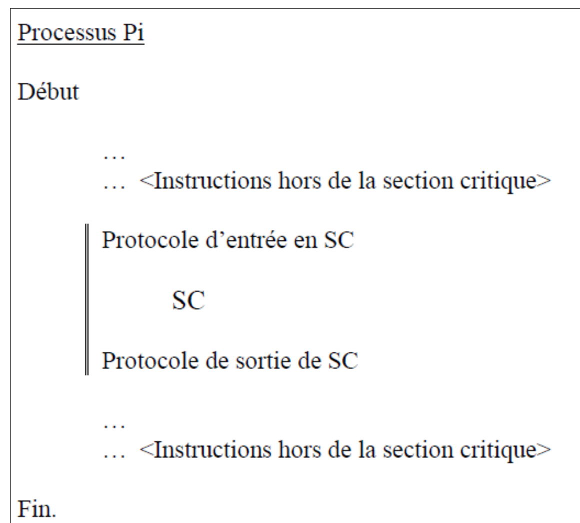
- **L'attente finie :** Si plusieurs processus sont en compétition pour entrer en SC, le choix de l'un d'eux ne doit pas être repoussé indéfiniment (pas de famine).

- Tous les processus doivent être **égaux** vis à vis de l'entrée en SC.

Voici donc le principe général d'une solution garantissant que l'exécution simultanée de plusieurs processus ne conduirait pas à des résultats imprévisibles :

1. Avant d'exécuter une SC, un processus doit s'assurer qu'aucun autre processus n'est en train d'exécuter une SC du même ensemble.
2. Dans le cas contraire, il ne devra pas progresser tant que l'autre processus n'aura pas terminé sa SC.
3. Avant d'entrer en SC, le processus doit exécuter un protocole d'entrée. Le but de ce protocole est de vérifier justement si la SC n'est occupée par aucun autre processus.
4. A la sortie de la SC, le processus doit exécuter un protocole de sortie de la SC. Le but de ce protocole est d'avertir les autres processus en attente que la SC est devenue libre.

La structure suivante résume ce principe de fonctionnement :



On peut réaliser l'exclusion mutuelle par trois méthodes :

1. Solutions logicielles :
    - Algorithme de Dekker (Pour 2 processus)
    - Algorithme du Boulanger (pour plusieurs processus)

2. Solutions matérielles :
    - Masquage des interruptions
    - L'instruction TEST-AND-SET
    - L'instruction SWAP
3. Sémaphores.

3.1. Sémaphores :

Les solutions matérielles et logicielles sont difficiles à mettre en œuvre pour des problèmes de synchronisation complexes. Dans cette section, nous définirons l'outil de synchronisation le plus connu qu'est **le sémaphore**. Les sémaphores ont été introduits par **Dijkstra** (informaticien hollandais) en 1965.

Un sémaphore possède une valeur entière ***S***, définie entre 2 primitives :

- ***Secure***(s) décrémentation de la valeur du sémaphore et blocage du processus appelant si la valeur est devenue < 0 → Prendre.

```
Primitive Secure(S)
Début
   S := S - 1;
   Si S < 0 Alors
      Bloquer le processus appelant et placer le dans la File F(s);
   Finsi
Fin.
```

- ***Release***(s) incrémentation de la valeur du sémaphore pouvant entrainer le déblocage d'un processus bloqué → Libérer.

```
Primitive Release(S)
Début
   S := S + 1;
   Si S <= 0 Alors
      Faire sortir le processus de la File F(s) et l'activer;
   Finsi
Fin.
```

De ce qui précède, on peut facilement proposer un schéma de synchronisation de *n* processus voulant entrer simultanément en SC, en utilisant les deux opérations ***Secure*** et ***Release***. En effet, il suffit de faire partager les *n* processus avec un sémaphore **mutex**, initialisé à 1, appelé sémaphore d'exclusion mutuelle.

Chaque processus *Pi* a la structure suivante :

```
Processus Pi
Début
      Secure(mutex);
      SC
      Release(mutex);
Fin.
```

Pour voir davantage l'efficacité des sémaphores comme outil de synchronisation, considérons l'exemple suivant : Deux processus P1 et P2 exécutent respectivement deux instructions S1 et S2.

```
Processus P1
Début
      S1;
Fin.
```

```
Processus P2
Début
      S2;
Fin.
```

Si on souhaite que S2 ne doit s'exécuter qu'après l'exécution de S1, nous pouvons implémenter ce schéma en faisant partager P1 et P2 un sémaphore commun S, initialisé à 0 et en insérant les primitives ***Secure*** et ***Release*** de cette façon :

```
Processus P1
Début
      S1;
      Release(S);
Fin.
```

```
Processus P2
Début
      Secure(S);
      S2;
Fin.
```

Comme ***S*** est initialisé à 0, P2 exécutera S2 seulement une fois que P1 aura appelé ***Release***(S).

Résumé sur les sémaphores :

Lorsqu'un processus désire acquérir une ressource, il exécute une opération **Secure**. Il est donc bloqué si aucune ressource n'est disponible.

Lorsqu'il libère la ressource, il exécute une opération **Release** qui signale la disponibilité de la ressource et débloque donc un processus éventuellement en attente.

Remarque : Théoriquement, un sémaphore peut être initialisé à n'importe quelle valeur entière, mais généralement cette valeur est positive ou nulle. Dans la plupart des cas pratiques, la valeur initiale = nombre de ressources disponibles.

Exercice : 3 tâches doivent utiliser une ressource critique R, tel que :

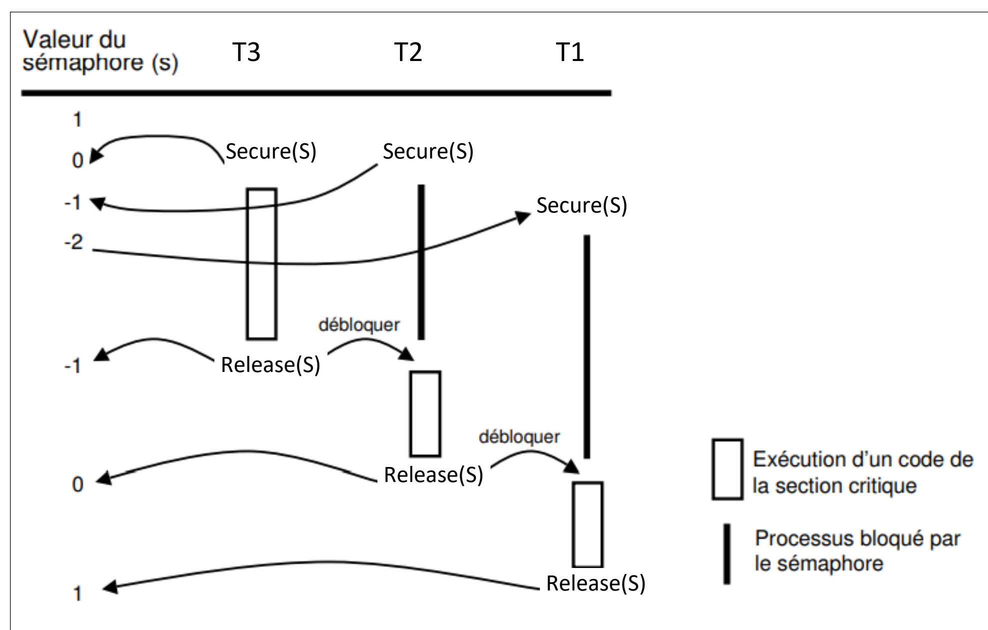
- T1 utilise R pendant 15 UT et demande R à l'instant t=12 UT.
- T2 utilise R pendant 10 UT et demande R à l'instant t=8 UT.
- T3 utilise R pendant 14 UT et demande R à l'instant t=0 UT.

Initialement, ***S***=1 et UT = unité de temps. On utilise un sémaphore **mutex** pour réaliser l'exclusion mutuelle. Que contient la file d'attente et quelle est la valeur de ***S***.

Solution :

Initialisation S=1 (**mutex**)

UT	R	F(s)	*S*
0	T3	Ø	0
8	T3	T2	-1
12	T3	T2 et T1	-2
14	T2	T1	-1
24	T1	Ø	0
39	Ø	Ø	1



4. Synchronisation :

La synchronisation permet de gérer des relations d'ordre de plusieurs tâches. Elle se réalise par sémaphores ou par un évènement.

4.1. Synchronisation par sémaphores :

Chaque tâche (T_i) est définit par un sémaphore **privé** (S_i) : $Ti \rightarrow S_i$

Seule la tâche propriétaire du sémaphore peut exécuter l'opération ***Secure***. Les autres processus ne peuvent exécuter que l'opération ***Release***. En général, la valeur initiale d'un sémaphore privé est 0.

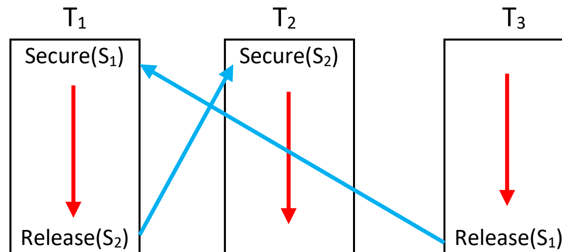
- Initialisation S_i =0,
- T_i ↔ exécute ***Secure***(S_i) seulement;
- T_j ↔ exécute ***Release***(S_i) avec $(i \neq j)$.

Exemple :

$T_3 \rightarrow T_1 \rightarrow T_2$

On définit :

- $S_1 \rightarrow T_1$ et $S_1 = 0$
- $S_2 \rightarrow T_2$ et $S_2 = 0$



4.2. Synchronisation par évènement :

Un évènement (e = event) est une information qui vienne d'une source matérielle (capteurs) ou logicielle (applications). Il peut prendre la valeur "***0 = faux***" ou "***1 = vrai***" et sur lequel 3 primitives sont définies : ***Wait*** = attendre, ***Signal*** = Déclencher, ***Reset*** = Réinitialiser.

```
Primitive Wait(e)
Début
   Si e = faux Alors
      Bloquer le processus dans la File F(e);
   Finsi
Fin.
```

```
Primitive Signal(e)
Début
   Si e = vrai Alors
      Débloquer le processus et placer le à l'état prêt;
   Finsi
Fin.
```

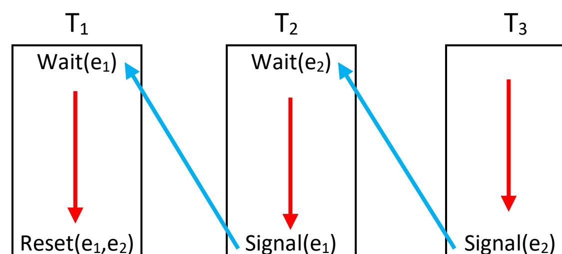
```
Primitive Reset(e)
Début
   e = faux;
Fin.
```

Exemple 01 :

$T_3 \rightarrow T_2 \rightarrow T_1$

Initialisation :

- $e_1 = 0$
- $e_2 = 0$



Exemple 02 : En utilisant les événements, la synchronisation des processus Clavier et Ecran s'exprime comme suit :

```
Var
    c: char;
    e=faux; % évènement initialisé par 0;

Process Clavier;
Var car_lu: char;
debut
    c:= car_lu; %lire un caractère dans la variable car_lu
    Signal(e);
End;

Process Écran;
Var car_à_écrire : char;
debut
    Wait(e);
    car_à_écrire := c;
    % écrire le caractère car_à_écrire sur l'écran.
end;

Begin
    Repeat
        Reset(e);
        Cobegin
            Clavier;
            ecran;
        Coend
    Forever
End
```

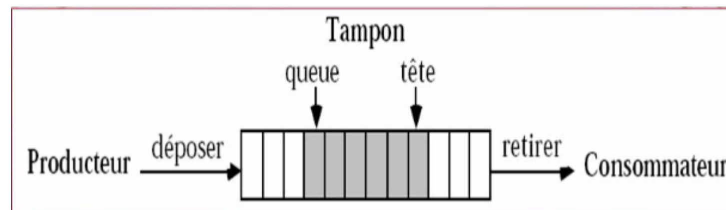
Le processus **Écran** est bloqué uniquement s'il exécute ***Wait(e)*** avant que le processus **Clavier** n'exécute ***Signal(e)***.

Une fois bloqué, le processus **Ecran** est débloqué lorsque le processus **Clavier** exécute ***Signal(e)***. La synchronisation est donc correctement résolue.

Remarque importante :

Les trois primitives (***Wait***, ***Signal*** et ***Reset***) sont utilisées seulement pour la synchronisation par évènement.

5. Problème de Producteur/Consommateur :



Les deux processus (**Producteur** et **Consommateur**) coopèrent en partageant un même tampon (un buffer) :

- Le producteur (**P**) produit des objets qu'il dépose dans le tampon.
- Le consommateur (**C**) retire des objets du tampon pour les consommer.

Problèmes/Conflits :

- Le producteur veut déposer un objet alors que le tampon est déjà plein ;
- Le consommateur veut retirer un objet du tampon alors que celui-ci est vide ;
- Le producteur et le consommateur ne doivent pas accéder simultanément au tampon.

Il y a deux problèmes :

- Les deux tâches ne peuvent pas avoir accès simultanément au buffer → **Exclusion mutuelle**.
- La tâche **C** accède le buffer après la production s'il est vide, tandis que la tâche **P** accède le buffer après la consommation s'il est plein → **Synchronisation**.

Variables et Initialisation :

```
#define N // nombre de places dans la file (taille de tampon)
typedef int sémaphore; //Les sémaphores partagés par tous les processus
sémaphore mutex = 1; // contrôle d'accès section critique (un seul processus en SC)
sémaphore vide = N; // nb. de places libres (la file est toute vide)
sémaphore plein = 0; // nb. de places occupées (aucun emplacement occupé)
```

```
void producteur() {
while (1) {
produire_objet (x); //produire un objet
Secure(vide); // on veut une place vide alors décrémenté des places libres
Secure(mutex); // on bloque la file (début section critique)
mettre_objet(x); // mettre l'objet en file (SC/RC)
Release(mutex); // libération de la file (fin section critique)
Release(plein); // incrémente des places occupées (un objet est à prendre)
}
}
```

```
void consommateur(){
while (1) {
Secure(plein); // attente d'un objet (décrémente des places occupées)
Secure(mutex); // début section critique
retirer_objet(x); // prendre l'objet courant (SC/RC)
Release(mutex); // fin section critique
Release(vide); // incrémente des places vides (une place est à prendre)
consommer_objet(x) //consommer l'objet courant
}
}
```

- ✓ Les processus **producteurs** produisent de l'information vers ces emplacements.
- ✓ Les processus **consommateurs** utilisent cette information et libère la place.
- ✓ Le système synchronise les deux types de processus pour ne pas perdre de données :
    - Bloquer un producteur si pas de place (plein) ou,
    - Bloquer un consommateur si pas d'information disponible (vide).

6. Communication entre tâches :

La communication entre les processus est un échange d'information (données, résultats,...). Elle se concrétise par deux catégories d'outils : les boites aux lettres et la mémoire partagée.

6.1. Boites aux lettres (Mail Box) :

Une boite aux lettres est une structure de données a pour objectif de recevoir ou transmettre des messages (des données) entre les tâches concurrentes et non synchronisées. On peut y écrire des messages et les récupérer.

- ➢ **Écrire message :** Put_mailbox(Identificateur, message)
- ➢ **Lire message :** Get_mailbox(Identificateur, message)

L'identificateur est une variable qui identifie la boite aux lettres. Après la lecture, le message est systématiquement effacé.

6.2. Mémoire partagée :

- Un moyen de communication très efficace entre les tâches :
    - ➢ implémente naturellement pour les processus ;
    - ➢ ne passe pas par le système d'exploitation (donc rapide).
- Mais dangereux :
    - ➢ accès concurrent à une même ressource.
- Elle nécessite l'utilisation des outils de synchronisation (sémaphores).

Chapitre 06:

Programmation temps réel

(RT programming)

1. Rappel :

- **Programmation parallèle :** Exécution d'un programme sur plusieurs unités (comme les Cœurs, processeurs, ordinateurs,…).
- **Programmation distribuée ou répartie :** Répartir l'exécution d'un programme sur plusieurs machines distinctes, ou programmation parallèle sur plusieurs machines.
- Le parallélisme dans l'exécution de logiciels peut se produire à quatre niveaux différents :
    - ***Niveau instructions machine :*** exécutant plusieurs instructions machine simultanément ;
    - ***Niveau instructions de code :*** exécutant plusieurs instructions de code source simultanément ;
    - ***Niveau unités :*** exécutant plusieurs unités (routines ou sous-programmes) simultanément.
    - ***Niveau programmes :*** exécutant plusieurs programmes simultanément.

2. Programmation concurrente :

La programmation concurrente est un paradigme de programmation tenant compte, dans un programme, de plusieurs contextes d'exécution (threads, processus, tâches) matérialisés par une pile d'exécution (stack) et des données privées

La concurrence est indispensable lorsque l'on souhaite écrire des programmes interagissant avec le monde réel ou tirant parti de multiples processeurs (multi-coeurs, clusters, cloud, ...) :

- Un processus est décomposé en threads.
- Un thread est une sorte d'une tâche légère qui s'exécute.
- Plus ou moins indépendamment des autres.

Donc, dans la **programmation parallèle** (ou répartie) les processus s'exécutent sur plusieurs processeurs. Par contre, les tâches sont gérées par un même processeur dans la **programmation concurrente**.

2.1. Types de concurrence :

- **Disjointe** : pas de communication entre les entités concurrentes.
- **Compétitive** : compétition pour l'accès à des ressources (CPU, E/S, mémoire, ...).
- **Coopérative** : coopération pour atteindre un objectif commun.

2.2. Avantages de la programmation concurrente :

- Optimiser l'utilisation du/des processeurs.
- Eviter de bloquer sur des entrées/sorties.
- Tirer avantage des architectures multi-cœurs.
- Augmenter le parallélisme : Tout programme faisant du calcul devrait être développé de manière concurrente.
- Attendre des événements de plusieurs entrées.
- Simplifier la structure d'un programme.
- Satisfaire des contraintes temporelles.

2.3. Comment implémenter la concurrence :

- Grâce à des mécanismes du langage de programmation :

  Avantage :

  - Les notions concurrentes de même que les constructions sont données par le langage
  - Détection d'une partie des erreurs à la compilation
  - Méthodologie de programmation imposée par le langage

  Désavantage :

  - Obligation d'utiliser un langage dédié qui est potentiellement peu répandu
  - Contraintes liées au langage choisi (pas forcément souhaitable)

  Exemples : ADA, Java, C++11, etc.

- Grâce à des bibliothèques : Le système d'exploitation offre une bibliothèque appelée système multitâche :

  Avantage :

  - Un langage quelconque peut profiter de la bibliothèque

Désavantage :

- La portabilité (dépendance au système cible).
- Débogage délicat.
- Pas de méthodologie de programmation imposée.

Exemple: langage C ou C++ avec bibliothèque *POSIX Threads* aussi appelée *Pthreads*.

❖ Solution intermédiaire : Un pré-compilateur gère l'implémentation des outils.

Exemple: C++ avec librairie *Qt*, c'est un code indépendant de la plateforme cible.

3. Gestion des aspects multithreads :

Un thread est un fil d'exécution, tandis qu'un processus est composé de plusieurs threads. Les threads s'exécutent en "parallèle" Sur un monoprocesseur (chacun son tour ou par entrelacement) ou sur un multiprocesseur. Dans tous les cas, l'ordonnanceur est responsable de l'ordre d'exécution

Problèmes:

- Accès à des ressources partagées.
- Echange de données.
- Séquentialité de l'exécution.

4. Séparation d'un programme en plusieurs threads :

Il existe plusieurs modèles pour décomposer un programme en plusieurs threads, on peut citer trois modèles les plus utilisés :

4.1. Modèle délégation (boss-worker model) : un thread principal s'occupe de répartir la charge de travail sur les threads travailleurs.

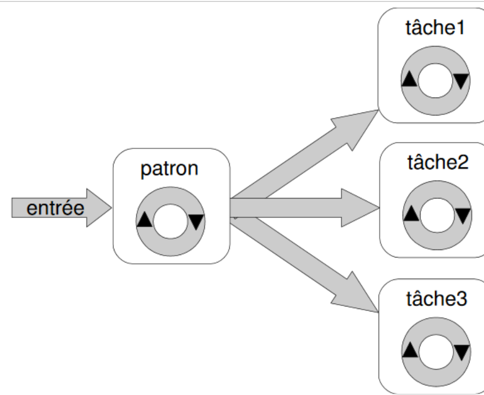


Figure 6.1. Modèle délégation.

Exemple :

```
void*patron(void*) {
boucle infinie {
       attend une requête
       switch (requete) {
          case requeteX: startThread( ... tacheX);break;
          case requeteY: startThread( ... tacheY);break;
          ...
                  }
                 }
                }
void*tacheX(void*) {
     exécuter le travail demandé, puis se terminer
}
void*tacheY(void*) {
     exécuter le travail demandé, puis se terminer
}
```

4.2. Modèle pair (peer model) : aucun thread n'est principal, tous étant égaux au niveau hiérarchique. Chaque thread est alors responsable de gérer ses propres entrées/sorties. La synchronisation entre thread risque fort d'y être nécessaire, afin que la tâche globale s'exécute correctement.

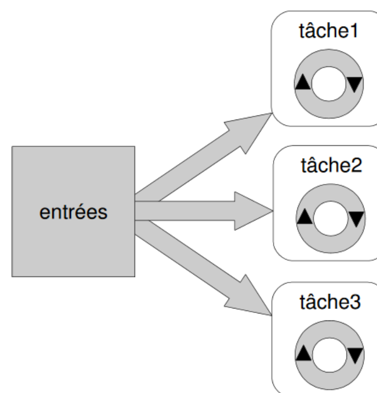


Figure 6.2. Modèle pair.

Exemple :

```
main() {
startThread( ... tache1);
startThread( ... tache2);
...
signale aux threads qu'ils peuvent commencer à travailler
}
tache1() {
attend le signal de commencement effectue le traitement, et
synchronise avec les autres threads si nécessaire
}
tache2() {
attend le signal de commencement effectue le traitement, et
synchronise avec les autres threads si nécessaire
}
```

4.3. Modèle pipeline (pipeline model) : Ce modèle est exploitable lorsque les conditions suivantes sont remplies :

- L'application traite une longue chaîne d'entrée ;
- Le traitement à effectuer sur ces entrée peut être décomposé en sous-tâches (étages de pipeline) au travers desquelles chaque donnée d'entrée doit passer ;
- Chaque étage peut traiter une donnée différente à chaque instant.

Donc, Un thread attend les données du précédent et les transmet ensuite au suivant.

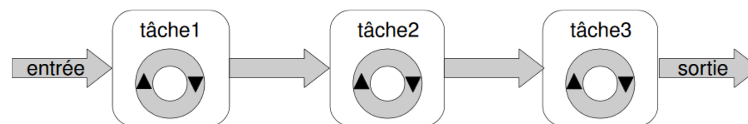


Figure 6.3. Modèle pair.

Exemple :

```
main()
{
startThread( ... etage1);
startThread( ... etage2);
...
startThread( ... etageN);
...
}
```

5. Langage de Programmation temps réel :

5.1. Ada : conçu initialement par l'équipe de CII-Honeywell Bull en réponse à un cahier des charges établi par le département de la Défense des États-Unis en 1980.

Avantage :

- Le langage intègre des notions de concurrence
- Certaines vérifications peuvent être faites à la compilation
- Robuste

Inconvénient :

- Moins utilisé que les autres en pratique

5.2. Java : est un langage de programmation orienté objet présenté officiellement en 1995 au sein de la société *SunWorld*.

Avantage :

- Orienté objet
- Mécanismes de synchronisation natifs

Inconvénient :

- Langage interprété
- Donc: légèrement plus lent que C/C++
- Définition un peu légère du fonctionnement des primitives

5.3. C/C++ (POSIX) : Utilisation de la bibliothèque POSIX

Avantage :

- Très utilisé dans le monde (notamment embarqué)
- Code compilé (rapidité)

Inconvénient :

- Pas de mécanismes de synchronisation natifs.

Systèmes Temps Réel
(Real Time Systems)

Exercices corrigés

et

Exercices proposés

Questions de cours :

1. Citer au moins 3 rôles d'un OS ?
2. Quelle est la déférence entre Instruction et Macro-instruction ?
3. Que veut dire une séquence de tâches valide ?
4. Pour qu'un ordonnanceur soit très performant, il doit maximiser .... et également minimiser........................ (Compléter les critères).
5. Quel type d'algorithme d'ordonnancement qui souffre de la famine ?
6. La surveillance d'une centrale nucléaire nécessite un système temps réel souple : **Oui/Non.**
7. Le Memory Management Unit (MMU) traduit une adresse virtuelle en adresse logique : **Oui/Non.**
8. Les sémaphores privés sont toujours initialisés par Si = -1 : **Oui/Non.**
9. Citer au moins 3 rôles d'un gestionnaire de mémoire ?
10. Quelle est la déférence entre *Shell* et *Kernel* dans un OS ?
11. Un sémaphore **mutex** est initialisé par S = 1 dans un système qui contient deux ressources critiques (Oui/Non avec justification)**.**

Réponses aux questions :

1. **Rôles d'un OS** : 1- Gestion du processeur, 2- Gestion de la mémoire vive, 3- Gestion des entrées/sorties, 4- Gestion de l'exécution des applications, 5- Gestion des fichiers, 6- Gestion des informations.
2. **Instruction** : une opération élémentaire d'un processeur. **Macro-instruction** : instruction complexe, définissant des opérations composées à partir des instructions du répertoire de base d'un ordinateur.
3. **Une séquence de tâches valide** : si toutes les tâches respectent leur Contrainte Temporelle.
4. Pour qu'un ordonnanceur soit très performant, il doit maximiser le pourcentage d'utilisation du processeur et le débit. Également, il doit minimiser le temps de réponse, le temps de rotation et le temps d'attente.
5. L'ordonnancement basé sur les priorités (**PRIO**) souffre de **la famine**, car les processus moins prioritaires n'arrivent pas à s'exécuter pendant un temps indéterminé.
6. **Non**, La surveillance d'une centrale nucléaire nécessite un système temps réel dur.
7. **Non**, Le Memory Management Unit (MMU) traduit une adresse virtuelle (logique) en adresse physique.
8. **Non**, Les sémaphores privés sont toujours initialisés par Si = 0.
9. Au moins 3 rôles d'un gestionnaire de mémoire :
   - création d'un processus.
   - activation/désactivation d'un processus.
   - suppression d'un processus.
   - partage la mémoire disponible entre les processus (protection).
   - cartographie la mémoire.
   - alloue/dés-alloue de la mémoire dynamiquement pour les besoin d'un processus.
   - assure la cohérence de la mémoire.
   - optimise l'utilisation de la mémoire.

10. **Shell** : assure la communication avec le système d'exploitation par l'intermédiaire d'un langage de commandes, afin de permettre à l'utilisateur de piloter les périphériques.
    Kernel : représentant les fonctions fondamentales du système d'exploitation telles que la gestion de la mémoire, des processus, des fichiers, des entrées-sorties principales, et des fonctionnalités de communication.
11. **Non**, Le sémaphore de ce système est initialisé par S = 2, parce qu'il possède 2 ressources critiques.

Exercice N°1 (Gestion des tâches) :

Soit un système temps réel avec 3 tâches ayant les paramètres suivants :

Tâche	Temps de réveil (r)	Durée d'exécution (C)	Délai critique (D)	Période (P)
T1	0	2	4	6
T2	0	3	8	8
T3	0	1	3	4

Partie I : Dans cette partie, on s'intéresse uniquement à la zone grise de tableau → Ti(ri, Ci).

1- Tracer les chronogrammes de **SJF** sans préemption et **RR** (*Q* = 2) ?

2- Pour chaque cas, calculer le temps moyen d'attente et le temps moyen de rotation ?

3- Déduire le meilleur ordonnanceur ? Justifier ta réponse ?

Partie II : Maintenant, on considère tout le tableau → Ti(ri, Ci, Di, Pi).

4- Calculer la période d'étude de ce système.

5- Ce système est-il ordonnançable avec **RM** ? Vérifier avec le diagramme de Gantt ?

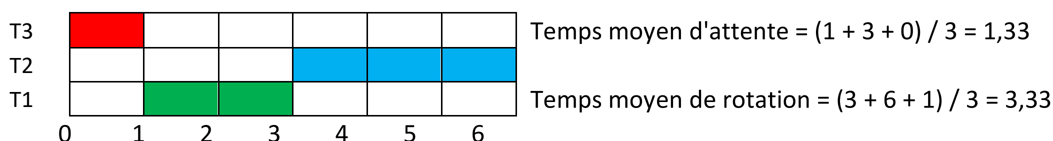
6- Même question pour **EDF** ?

Correction exercice N° 01 :

Partie I : Ti(ri, Ci).

Tâche	Temps de réveil (r)	Durée d'exécution (C)
T1	0	2
T2	0	3
T3	0	1

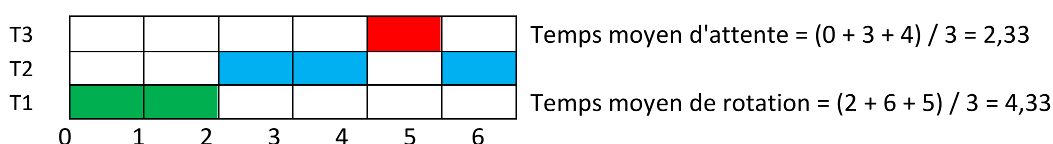
Le chronogramme de **SJF** sans préemption :



Temps moyen d'attente = (1 + 3 + 0) / 3 = 1,33

Temps moyen de rotation = (3 + 6 + 1) / 3 = 3,33

Le chronogramme de **RR** (*Q* = 2) :



Temps moyen d'attente = (0 + 3 + 4) / 3 = 2,33

Temps moyen de rotation = (2 + 6 + 5) / 3 = 4,33

3- Le meilleur ordonnanceur est le **SJF** sans préemption, car le temps moyen d'attente et le temps moyen de rotation correspondants sont les plus petits.

Partie II : Ti(ri, Ci, Di, Pi).

Tâche	Temps de réveil (r)	Durée d'exécution (C)	Délai critique (D)	Période (P)
T1	0	2	4	6
T2	0	3	8	8
T3	0	1	3	4

4- La période d'étude = [0, PPCM(P1, P2, P3)] = [0, PPCM(6, 8, 4)] = [0, 24].

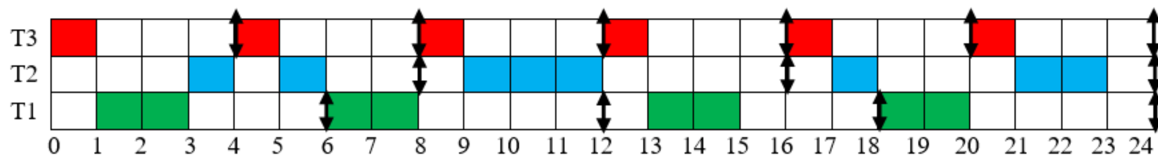
5- Test d'ordonnançabilité avec **RM** :

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n\left(2^{1/n} - 1\right) \rightarrow \text{Condition suffisante.}$$

$$\sum_{i=1}^{n} \frac{C_i}{P_i} = \frac{2}{6} + \frac{3}{8} + \frac{1}{4} = 0{,}958$$

$$n\left(2^{1/n} - 1\right) = 3\left(2^{1/3} - 1\right) = 0.779$$

Donc, la condition suffisante n'est pas vérifiée, on trace le diagramme de Gantt pour voir s'il y a des éventuelles fautes temporelles. On a T3 → T1 → T2.



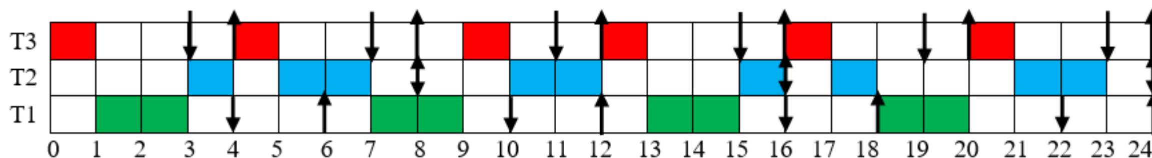
Il y a une faute temporelle dans la première période de la tâche T2, elle s'est exécutée pendant 2 UT seulement tandis que C2 = 3 UT. Donc, ce système n'est pas ordonnançable selon **RM**.

6- Même question pour **EDF** :

$$\sum_{i=1}^{n} \frac{C_i}{P_i} = 0.958 \leq 1 \text{ (La condition nécessaire est vérifiée)}$$

$$\sum_{i=1}^{n} \frac{C_i}{D_i} = 1.2 > 1 \text{ (La condition suffisante n'est pas vérifiée)}$$

On trace le diagramme de Gantt :



Exercice N°2 (Gestion des tâches) :

On considère les processus suivants, définis par leur durée (réelle ou estimée), leur date d'arrivée et leur priorité :

Tâche (P_i)	Date d'arrivée (r)	Durée d'exécution (C)	Priorité
P1	0	9	3
P2	2	7	3
P3	2	4	1
P4	4	8	2
P5	6	2	4

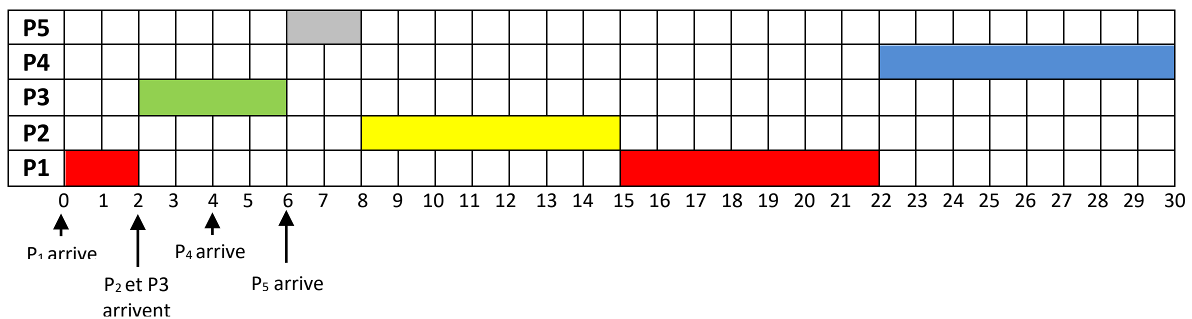
1- Dessinez un diagramme de Gantt correspondant au résultat d'un ordonnancement **SJF préemptif** et indiquez le temps d'attente moyen.

2- Dessinez un diagramme de Gantt correspondant au résultat d'un ordonnancement **Round Robin** avec un quantum de temps fixé à 2 et indiquez le temps d'attente moyen.

3- Quel est le meilleur algorithme ?

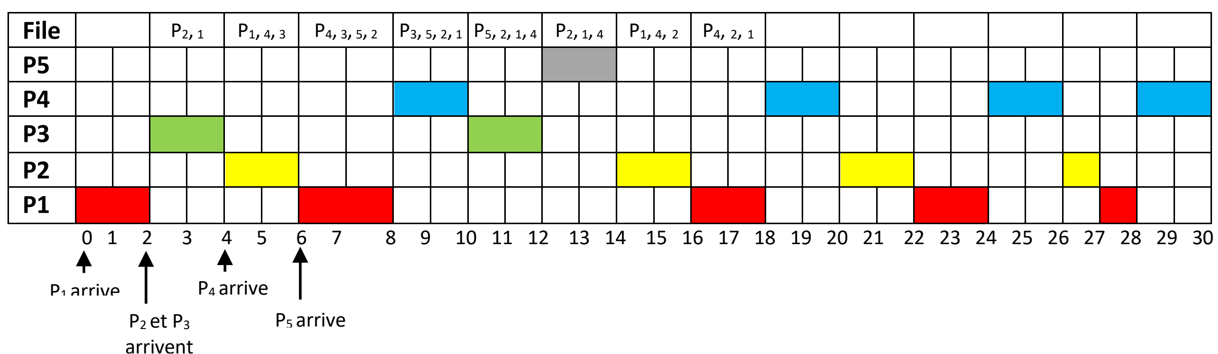
Correction Exercice N°2 :

1- SJF préemptif :



Temps moyen d'attente : (13 + 6 + 0 + 18 + 0) / 5 = 7.4

2- RR (Q=2) : dans ce cas on ignore la priorité



Temps moyen d'attente : (19 + 18 + 10 + 18 + 6) / 5 = 14.6

3- dans les deux cas, c'est le SJF préemptif le meilleur.

Exercice N°3 (Pagination) :

On suppose un espace d'adresses logiques de huit pages de 1024 octets chacune, représenté dans une mémoire physique de 32 cadres de pages. Combien de bits comporte l'adresse logique ? L'adresse physique ? Expliquez.

Correction Exercice N°3 :

$1024 = 2^{10}$ donc 10 bits pour l'offset dans la page, $8 = 2^3$ donc 3 bits pour le numéro de page et $32 = 2^5$ donc 5 bits pour les cadres de pages.

- Adresse logique = 13 bits.
- Adresse physique = 15 bits.

Exercice N°4 (Pagination) :

On suppose un système de 2048 Ko de mémoire haute organisé avec des pages de 8Ko. Décrivez le système d'adressage logique. Quelle est la taille maximum de la table des pages ? Expliquez.

Correction Exercice N°4 :

2048 Ko / 8 Ko = $2^{11} / 2^3 = 2^8$ pages. Pour adresser une page, il faut 1 octet.
On a donc une table des pages qui peut faire 2^8 octets = 256 o.

Exercice N°5 (Gestion de la mémoire) :

On se place dans un système de mémoire de 1700 Ko de mémoire haute (c'est-à-dire au-delà de la partie utilisée par l'OS) répartie en cinq partitions de 100Ko, 500Ko, 400Ko, 300Ko et 600Ko (dans cet ordre).
On suppose que le système d'exploitation doit allouer des processus de taille 212Ko, 417Ko, 112Ko et 426Ko (dans cet ordre). Pour chacun des algorithmes suivants, donnez l'allocation obtenue et le taux de fragmentation :

- First-Fit (prochain bloc libre)
- Best-FIt (plus petit bloc libre)
- Worst-Fit (plus grand bloc libre)

Quel algorithme utilise le plus efficacement la mémoire sur cet exemple ?
Calculer le Taux de fragmentation (espace libre/espace total) pour chaque cas ?

Correction Exercice N°5 :

On a les processus P1(212Ko), P2(417K), P3(312K) et P4(426K) :

1) First fit :

État initial	100K	500K	400K	300K	600K
Placement de P1(212K)	100K	P1+288K	400K	300K	600K
Placement de P2(417K)	100K	P1+288K	400K	300K	P2+183K
Placement de P3(312K)	100K	P1+288K	P3+88K	300K	P2+183K
Placement de P4(426K)	Impossible				

2) Best fit :

État initial	100K	500K	400K	300K	600K
Placement de P1(212K)	100K	500K	400K	P1+88K	600K
Placement de P2(417K)	100K	P2+83K	400K	P1+88K	600K
Placement de P3(312K)	100K	P2+83K	P3+88K	P1+88K	600K
Placement de P4(426K)	100K	P2+83K	P3+88K	P1+88K	P4+174K

Taux de fragmentation = (100+83+88+88+174)/(100+500+400+300+600) = 0.28

3) Worst fit :

État initial	100K	500K	400K	300K	600K
Placement de P1(212K)	100K	500K	400K	300K	P1+388K
Placement de P2(417K)	100K	P2+83K	400K	300K	P1+388K
Placement de P3(312K)	100K	P2+83K	P3+88K	300K	P1+388K
Placement de P4(426K)	Impossible				

L'algorithme qui utilise le plus efficacement la mémoire est : **Best Fit**.

Exercice N° 6 (Gestion des tâches et de mémoire) :

On considère un système disposant de 16 MB de mémoire physique, avec la partie résidente du système sur 4 MB. On suppose que l'exécution de chaque processus se compose d'un temps processeur suivi d'une demande d'E/S. On suppose de plus que les processus n'attendent pas pour leur E/S (par exemple, ils utilisent tous un périphérique différent). Le tableau suivant donne un exemple de séquences de tâches pour le système :

Instant *t*	Processus	Taille	Temps CPU	Durée E/S
0	A	3 MB	9 ms	2 ms
4	B	5 MB	6 ms	9 ms
6	C	5 MB	4 ms	4 ms
8	D	4 MB	2 ms	6 ms
10	E	1 MB	4 ms	3 ms

On suppose qu'un processus chargé y séjournera jusqu'à la fin de son exécution.
Donnez les états d'occupation de la mémoire aux différentes étapes de traitement de ces processus, sous les hypothèses suivantes :

- Partitions fixes de tailles 6 MB, 4 MB, 2 MB et 4 MB (pour le système) ;
- Le mode d'allocation des trous utilise l'algorithme meilleur ajustement (**Best Fit**);
- Le répartiteur de haut niveau (chargement de processus en mémoire) fonctionne selon **FCFS** ;
- Le répartiteur de bas niveau (vers microprocesseur) fonctionne selon **SJF sans préemption**.

Correction Exercice N°6 :

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Microp	A	A	A	A	A	A	A	A	A	B	B	B	B	B	B	D	D	E	E	E	E				C	C	C	C					
4 Mo	Réservée pour le système d'exploitation																																
2 Mo											E	E	E	E	E	E	E	E	E	E	E	E	E	E									
4 Mo	A	A	A	A	A	A	A	A	A	A	A	D	D	D	D	D	D	D	D	D	D	D	D										
6 Mo					B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	C	C	C	C	C	C	C	C	
File D'attente							C	C	C D	C D	C D	C	C	C	C	C	C	C	C	C	C	C	C	C									

A t =0, A arrive et est chargé dans la partition de 4 Mo, il y séjournera jusqu'à la fin de son exécution (t=9+2).
A t = 4, B arrive et est chargé dans la partition de 6Mo. Il y séjournera jusqu'à la fin de son exécution.
A t= 6, C arrive et ne peut être chargé en mémoire. Il est mis en attente dans la file (du répartiteur de haut niveau). La file contient (C).
A t= 8, D arrive et ne peut être chargé en mémoire. Il est mis en attente dans la file (du répartiteur de haut niveau). La file contient (C D)
A t= 9, A libère le processeur. L'exécution de B est entamée. B libérera le processeur à t= 9+6 et la partition à t = 9+6+9.
A t=10, E arrive et est chargé dans la partition de 2 Mo. Il y séjournera jusqu'à la fin de son exécution.
A t= 11, l'exécution de A se termine, la partition de 4 Mo devient libre. D est chargé dans cette partition et y séjournera jusqu'à la fin de son exécution. La file contient (C).
A t= 15, B libère le processeur qui est alloué au plus court d'abord c-à-d D. D libérera le processeur à t=15+2 et la partition à t=15+2+6.
A t= 17, D libère le processeur qui est alloué au plus court d'abord c-à-d E. E libérera le processeur à t=17+4 et la partition à t=17+4+3.
A t=21, E libère le processeur mais tous les processus en mémoire sont en attente de fin d'E/S.
A t = 23, D libère la partition de 4 Mo.
A t = 24, B et E libèrent les partitions de 6 Mo et 2 Mo, C'est chargé dans la partition de 6 Mo.
A t = 28, C libère le processeur.
A t = 32, C libère la partition de 6 Mo.

Exercice N° 7 : (Concours Doctorat Université Oum Elbouagui 2019)

Les mesures prélevées sur un système donné ont montré que la durée moyenne d'exécution d'une tâche était de **T** avant que ne se produise sur les E/S. Un changement de tâche a besoin de délai de **S** que l'on peut considérer comme une perte de temps. Pour un algorithme d'ordonnancement de type tourniquet avec quantum de **Q**.

Donner une formule pour exprimer l'efficacité du processeur (facteur de son utilisation) pour chacun de ces cas suivants :

1) Q>T
2) S<Q<T
3) Q=S
4) Q proche de 0.

Correction Exercice N°7 :

T : Durée moyenne de la tâche.

S : Retard de changement (durant les commutations)

Q : quantum de Tourniquet

Efficacité de processeur (eff_p) = temps utile / temps total

Cas 1 (Q > T) :

Dans ce cas y a pas des commutations de tourniquet => eff_p = $\frac{T}{T+S}$

Cas 2 (S < Q < T) :

Pour exécuter T, on a besoin de T/Q commutations (interruptions) de Tourniquet et à chaque changement y aura un rtard S => le total de retard = S. $\frac{T}{Q}$

Donc eff_p = $\frac{T}{T+S.\frac{T}{Q}} = \frac{Q}{Q+s}$

Cas 3 (Q = S) :

eff_p = $\frac{Q}{Q+s} + \frac{Q}{Q+Q} = 1/2$ ça veut dire 50% d'efficacité de processeur.

Cas 4 (Q plus proche de 0) :

eff_p = $\frac{0}{0+s} = 0$ ça signifie que le processeur ne fonctionne pas (y a une perte de temps pour rien)

Exercice N° 8 (Pagination) :

Dans un système paginé, les pages font 256 octet en mémoire et on autorise chaque processus à utiliser au plus 4 cadres de la mémoire centrale. On considère cette table des pages avec la tâche P1 :

Page	0	1	2	3	4	5	6	7
Cadre	011	001	000	010	100	111	101	110
Présente	**oui**	non	**oui**	non	non	non	**oui**	non

1) De combien de mémoire vive dispose ce système ?

2) Calculez les adresses réelles correspondant aux adresses virtuelles suivantes :

240, 546, 1578, 2072

3) Que se passe-t-il si P1 génère l'adresse virtuelle 770 ?

4) On considère l'adresse virtuelle suivante : 0000 0000 0000 0111.

Sachant que les 4 bits de poids fort désigne le numéro de page et que 12 bits suivants représentent le déplacement dans la page, donnez l'adresse physique correspondant à cette adresse (exprimée en binaire).

Correction Exercice N°8 :

1) Comme les cadres sont numérotés sur 3 bits, il y a 2^3= 8 cadres. Taille d'un cadre = taille d'une page donc la mémoire physique comporte 8*256 = 2Ko.

2) La conversion d'une adresse virtuelle en adresse réelle est réalisée de la façon suivante :

a- Calcul du numéro de la page et du déplacement dans la page.
b- Recherche dans la table de pages de l'entrée qui correspond à la page de façon à en déduire le numéro du cadre.
c- L'adresse physique (réelle) est obtenue en ajoutant le déplacement à l'adresse physique de début du cadre.

Voici le détail des calculs pour les adresses demandées :

- 240 = 0*256 + 240 → page = 0 et déplacement = 240. D'après la table des pages, cadre= 3. Adresse physique = 3*256 + 240 = 1008.
- 546 = 2*256 + 34 → page = 2 et déplacement = 34. D'après la table des pages, cadre= 0. Adresse physique = 0*256+ 34 = 34.
- 1578 = 6*256 + 42 → page = 6 et déplacement = 42. D'après la table des pages, cadre= 5. Adresse physique = 5*256 + 42 = 1322.
- 2072 est en dehors de l'espace d'adressage virtuel du processus (2048 mots).

3) 770 = 3*256 + 2. Il s'agit d'une adresse située dans la page 3. D'après la table des pages, cette page n'est pas présente en mémoire (un défaut de page).

4) D'après la table de pages, cette page se trouve dans le cadre 011. L'adresse physique s'obtient donc simplement en substituant aux 4 bits de poids fort de l'adresse virtuelle les 3 bits du numéro de cadre : 0011 0000 0000 0111.

Exercice N°9 : (Sémaphores, Inversion de priorité, Producteur/Consommateur)

Dans cet exercice on souhaite montrer l'impact d'une inversion de priorité sur l'ordonnancement d'un jeu de tâche. Soient trois tâches définies par les paramètres suivants :

	r_i	C_i	P_i
T1	0	2	6
T2	0	2	8
T3	0	4	12

Les délais critiques sont égaux aux périodes (D_i = P_i), On utilise **RM** pour ordonnancer les tâches (mode préemptif).

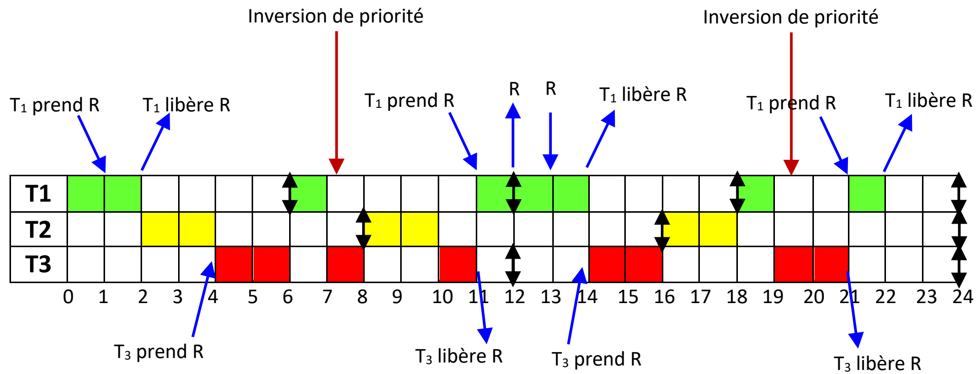
La tâche T1 produit des données alors que T3 les consomme. Ces données sont stockées dans une mémoire commune que T1 et T3 se partagent cette ressource (R) par l'accès en exclusion mutuelle. T1 accède à la ressource durant la deuxième unité de temps de sa capacité. T3 accède à la ressource durant la totalité de sa capacité.

1. Tracer sur la période d'étude le diagramme de Gantt généré par l'ordonnanceur RM.
2. Indiquer les moments d'accès exclusif à la ressource et les moments de sa libération.
3. Est-ce qu'il y a une inversion de priorité ? Si oui, indiquer où.

Correction Exercice N°9 :

Algorithme de RM : (T1 ↦ T2 ↦ T3)

Période d'étude =[0 – PPCM(6, 8, 12)] = [0 - 24].



Exercice N° 10 (Gestion des tâches) :

Soit un système temps réel avec 3 tâches ayant les paramètres suivants :

Tâche	Temps de réveil (r)	Durée d'exécution (C)	Délai critique (D)	Période (P)
T1	0	2	4	6
T2	0	**X**	8	8
T3	0	1	3	4

(X est un nombre entier positive)

1. Déterminez sous quelle condition sur **X** ce système est ordonnançable selon RM puis EDF.
2. Nous souhaitons maintenant introduire **m** copies de la tâche T2 en parallèle :
   Déterminez la condition obtenue sur **X**, en fonction de **m**, pour que ce nouveau système soit ordonnançable selon EDF ?
3. Pour **X**=3, tracer le diagramme de Gantt avec l'ordonnanceur RR (**Q** = 2) ? Calculer le temps moyen d'attente et le temps moyen de rotation ?

Correction exercice N° 10 :

1- **Ordonnançabilité selon RM :**

Condition suffisante : $\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n\left(2^{1/n} - 1\right) \Leftrightarrow \frac{2}{6} + \frac{X}{8} + \frac{1}{4} \leq 3(2^{\frac{1}{3}} - 1)$

$$\Leftrightarrow \frac{3X+14}{24} \leq 0.779$$

$$\Leftrightarrow X \leq 1.56$$

Ordonnançabilité selon EDF :

A. Condition nécessaire : $\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1 \Leftrightarrow \frac{3X+14}{24} \leq 1$

$$\Leftrightarrow X \leq 3.33$$

B. Condition suffisante : $\sum_{i=1}^{n} \frac{C_i}{D_i} \leq 1 \Leftrightarrow \frac{3X+20}{24} \leq 1$

$$\Leftrightarrow X \leq 1.33$$

2- Ordonnançabilité selon EDF avec **m** copies de T2 :

A. Condition nécessaire : $\sum_{i=1}^{n+m-1} \frac{C_i}{P_i} \leq 1 \Leftrightarrow \frac{2}{6} + \frac{mX}{8} + \frac{1}{4} \leq 1$

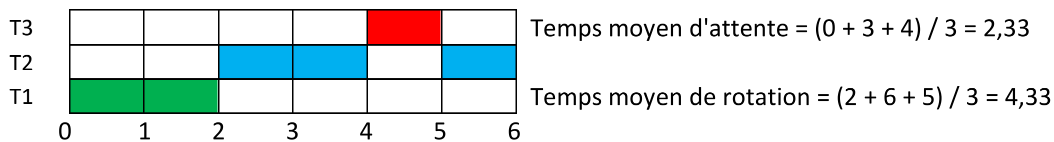
$$\Leftrightarrow X \leq \frac{3.33}{m}$$

B. Condition suffisante : $\sum_{i=1}^{n+m-1} \frac{C_i}{D_i} \leq 1 \Leftrightarrow \frac{3mX+20}{24} \leq 1$

$$\Leftrightarrow X \leq \frac{1.33}{m}$$

3- Le diagramme de Gantt pour **RR** (*Q* = 2 et X=3) :

Tâche	Temps de réveil (r)	Durée d'exécution (C)
T1	0	2
T2	0	X=3
T3	0	1



Exercices à corriger

Exercice N°11 :

Soit un système temps réel avec 3 tâches ayant les paramètres suivants :

Tâche	(r)	(C)	(D)	(P)
T1	0	2	5	6
T2	0	2	4	8
T3	0	4	8	12

1- Calculer la période d'étude de ce système.

2- Ce système est-il ordonnançable avec **RM** et **DM** ? Vérifier avec le diagramme de Gantt ?

3- Quel algorithme à priorités fixe vous semble le plus adapté pour cette application ?

4- Même question pour **EDF** et **LLF** ?

Exercice N°12 : (Concours Doctorat Université Oum Elbouagui 2019)

Un système d'arrosage automatique doit arroser trois types de plantes :

- les plus fragiles qui doivent être arrosées pendant 10 minutes, toutes les 40 minutes,
- une deuxième catégorie qui doit recevoir de l'eau pendant 20 minutes, toutes les heures,
- enfin, des plantes d'un troisième type qu'il faut arroser toutes les 80 minutes,

pendant 20 minutes. L'arrosage peut se faire de façon fractionnée, c'est-à-dire s'interrompre et reprendre.

Question A :

On cherche une solution pour le partage de l'eau entre ces différentes variétés de plantes :

1. Définir la liste des tâches à accomplir,
2. Puis, pour les stratégies RM et EDF :
   - Calculer l'ordonnançabilité de ces tâches,
   - Donner un schéma d'utilisation du système d'arrosage à partir du temps 0.

Question B :

On veut maintenant se servir du système d'arrosage pour nettoyer les allées qui desservent les plantations. On décide de faire cet entretien pendant 10 minutes toutes les heures.
Cet entretien est-il possible pendant les arrosages : avec RM, pourquoi ?
avec EDF, pourquoi ?

Question C :

Pour nettoyer toutes les allées, il faut 20 minutes. Si l'entretien commence 1h30 après le début de l'arrosage des plantes,
Pourra-t-on avoir complètement nettoyé les allées :

1. au bout d'une heure ?
2. après 100 minutes ?
3. conclure la meilleure durée pour nettoyer toutes les allées ?