Ordonancement des Processus dans Unix

Rappels

- Le processeur est une **ressource** partagée
  - Comme les terminaux, la mémoire, l'interface réseau…
  - Le système d'exploitation doit en répartir l'utilisation
    - UNIX = partage du temps (time sharing)
      - Exécution concurente des processus
- Ordonnanceur Unix
  - Chargé d'assurer le partage du temps
    - Donne l'illusion du partage sur mono-processeur
    - Donne le processus par brève tranche de temps
      - quantum
    - Déclenche le changement de contexte
      - Quand le quantum est épuisé
      - Quand un processus plus prioritaire est prêt

Objectifs et Critères de Conception

- Politiques
  - Règles de décision : qui et quand ?
- Implémentation
  - Algo et Struct de Données (comment ?)
- Objectifs de la politique d'ordonnancement
  - Temps de réponse rapide pour appli. interactives
  - Débit élevé pour travaux en arrière plan
  - Eviter la famine ...
- Objectifs de l'implémentation
  - Minimiser le temps d'exécution de la politique
  - Prendre en compte les contraintes matérielles
    - Changement de contexte = vider caches, pipelines, ...

Interruption d'Horloge

- Mécanisme indispensable
  - préemption d'un processus quand le quantum **expire**
- **Horloge matérielle**
  - Programmable
    - Emission d'une intr. à intervalle de temps fixes
      - période appelée tick d'horloge
        - Unité de mesure du temps dans le noyau Unix
        - jiffies linux : nb ticks écoulés depuis boot
      - Nombre de ticks/seconde : variable/macro HZ
        - Typiquement HZ=100, mais autres valeurs possibles...
    - Interruption => traitant d'interruption (handler)
      - Priorité maximale (2e après power-failure)
        - Exécution rapide
        - Tâches limitées

Travail Typique d'un Traitant d'Horloge Unix

- Réarmer l'intr. d'horloge (si nécessaire)
- Mettre à jour stats d'utilisation du CPU
  - Pour le processus courant
- Exécuter des traitements pour l'ordonnanceur
  - Recalcul des priorités
  - Gestion de l'expiration d'un quantum…
- Traiter dépassement de quota CPU
  - Envoi SIGXCPU au processus courant si quota atteint
- Màj horloge "time-of-day" (jiffies sous linux)
- Gérer les appels de fonction différés
- Réveiller les processus systèmes (swapper, …)
- Gestion des alarmes

Appels de Fonctions Différés (*callout*)

- Mécanisme pour enregitrer un appel différé
  - Ex: API SVR4

```
int to_ID=timeout(void (*fn)(), /* fonction */
                  caddr_t arg, /* argument */
                  long delta); /* délai */
```

- Utilisés pour différentes tâches périodiques
  - Retransmission des paquets réseau
  - Tâches de l'ordonnanceur et du gestionnaire mémoire
  - Surveillance des périphériques
    - Evite interruptions perdues
  - Interrogation de périphs ne supportant pas interruptions
    - Polling

Implémentation(s) des Listes de *Callout*

- Contraintes
  - Favoriser les accès
    - Critiques car réalisés par le traitant d'horloge
- Solution 4.3 BSD
  - Liste triée par date de "mise à feu"
  - Délai de "mise à feu" **relatif** à l'élément précédent
  - Interruption d'horloge : décrémente délai de la tête de liste
- Alternatives
  - Délai de mise à feu absolu
  - Tourniquet temporel (timing wheel)
    - Nombre de listes fixée, la "roue" tourne à chaque tick

Alarmes

- 3 types d'alarmes
  - Temps réel
    - Relative au temps total écoulé (depuis le boot)
    - Notification via SIGALRM
  - Profilage
    - Mesure le temps (total) d'exécution du processus
    - Notification via SIGPROF
  - Temps virtuel
    - Mesure le temps d'exécution en mode utilisateur
    - Notification via SIGVTALRM

Implémentation des Alarmes

- BSD Univ : A.S. setitimer()
  - Supporte les 3 types d'alarmes
  - Spécifie le temps en micro-secondes
    - Conversion en ticks : perte de précision
- System V
  - alarm() : temps reel, temps exprimé en secondes
  - hrtsys() : précision en micro-secondes (compat BSD)
- VTALRM et PROFALRM
  - tick d'horloge porté au crédit du processus courant
    - Car décision au moment de l'intr. d'horloge
    - Perte de précision supplémentaire ...

Précision des Alarmes Temps Reel

- Résolution, précision ... confusion !!
- Durée exprimé en micro-secondes, certes ...
- Mais précision bien moindre !
  - Basé sur les ticks d'horloges
    - Si HZ=100, résolution min : 10 **mili-secondes**
  - Quand le temps est écoulé, le processus est **seulement** mis dans l'état prêt
    - L'exécution reprend lorsque l'ordonnanceur lui donne le CPU
  - Les noyaux UNIX usuels sont non-interruptibles
    - Le noyau termine ce qu'il a commencé avec le processus courant avant de donner la main à un autre

Ordonnancement Unix Traditionnel (4.3 BSD/SVR3)

- Cibles principales
  - Applications interactives
  - Batch
- Préemptif + Tourniquet
  - CPU donné au processus de plus haute priorité
  - Tourniquet entre processus de même priorité
  - Préemption immédiate (ne laisse pas terminer quantum)
  - Noyau non préemptible
    - Le processus (en mode noyau) peut laisser volontairement le processeur
      - Blocage sur ressource (ex: wait queue), alloc mémoire, etc
    - Autrement, il ne peut être préempter qu'au moment du retour en mode utilisateur (retour d'A.S.)

Priorité des Processus (Ordon. Traditionnel)

- Priorité = entier de 0 à 127
  - 0 à 49 : priorités réservées au noyau
  - 50 à 127 : priorités en mode utilisateur
  - La structure du processus contient les champs suivants
    - p_pri : priorité courante
    - p_usrpri : priorité en mode utilisateur
    - p_cpu : mesure l'utilisation récente du processeur
    - p_nice : coefficient de gentillesse donné par l'ut.
- Priorités utilisateur et noyau ??
  - En mode utilisateur : p_pri = p_usrpri
  - En mode noyau :
    - priorité boostée après endormissement/réveil ...

Pourquoi *booster* la Priorité Après un Réveil ?

- Donner la priorité aux processus qui doivent terminer un traitement en mode noyau
  - Ils consomment des ressources temporaires
    - Ex: données allouées dynamiquement (kmalloc)
- Donner la priorité aux processus IO-BOUND
  - Typiquement les processus interactifs
    - Passent leur temps à attendre un événement
- Implémentation
  - Le noyau associe une priorité de réveil au moment d'endormir un processus
  - Au réveil p_pri = priorité de réveil
  - Au retour en mode utilisateur p_pri = p_usrpri

Calcul des Priorités

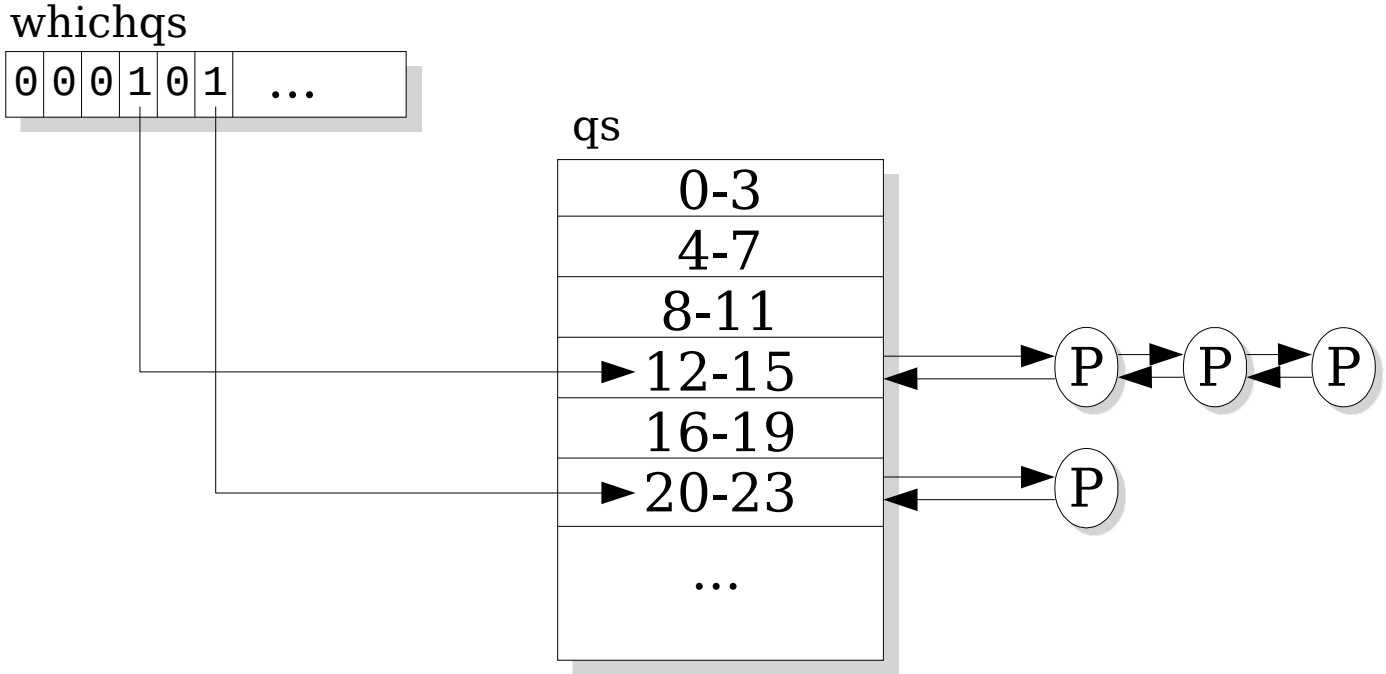
- Priorité en mode mode noyau
  - Valeur prédéfinie selon le type d'E/S
    - Terminal = 28
    - Disque = 20 ...
- Priorité en mode utilisateur
  - Dépend de deux facteurs :
    - Niveau de gentillesse : 0 à 39, 20 par défaut
      - Tâche en arrière plan automatiquement abaissée par le shell
    - Utilisation récente du CPU : p_cpu
      - facteur d'amortissement (oubli)
  - Recalcul de p_usrpri toutes les secondes
    - Càd tous les HZ ticks
    - p_usrpri = 50 + (p_cpu / 4) + (2 * p_nice)

Calcul de l'Utilisation Récente

- p_cpu recalculé toutes les secondes
  - p_cpu = p_cpu * **decay**
  - decay = facteur d'amortissement
    - Moyenne pondérée exponentielle => oubli progressif
      - Le temps consommé récemment a plus d'importance que le temps consommé il y a longtemps
- Valeur de decay dépend de l'implémentation
  - fixe = $\frac{1}{2}$ dans SVR3
    - Effet de bord indésirable : élève la priorité quand la charge augmente
  - variable dans 4.3 BSD
    - decay = (2*load_average) / (2*load_average +1)
      - On oublie moins le passé récent quand la charge augmente...
        - ... donc la priorité descend plus vite

Implémentation de l'Ordonnanceur (4.3BSD)

- 32 run queues
  - 4 priorité adjacentes par file
  - Chaque file = liste doublement chainée de structures de processus
  - whichqs : bitmap indiquant si chaque file est vide ou non



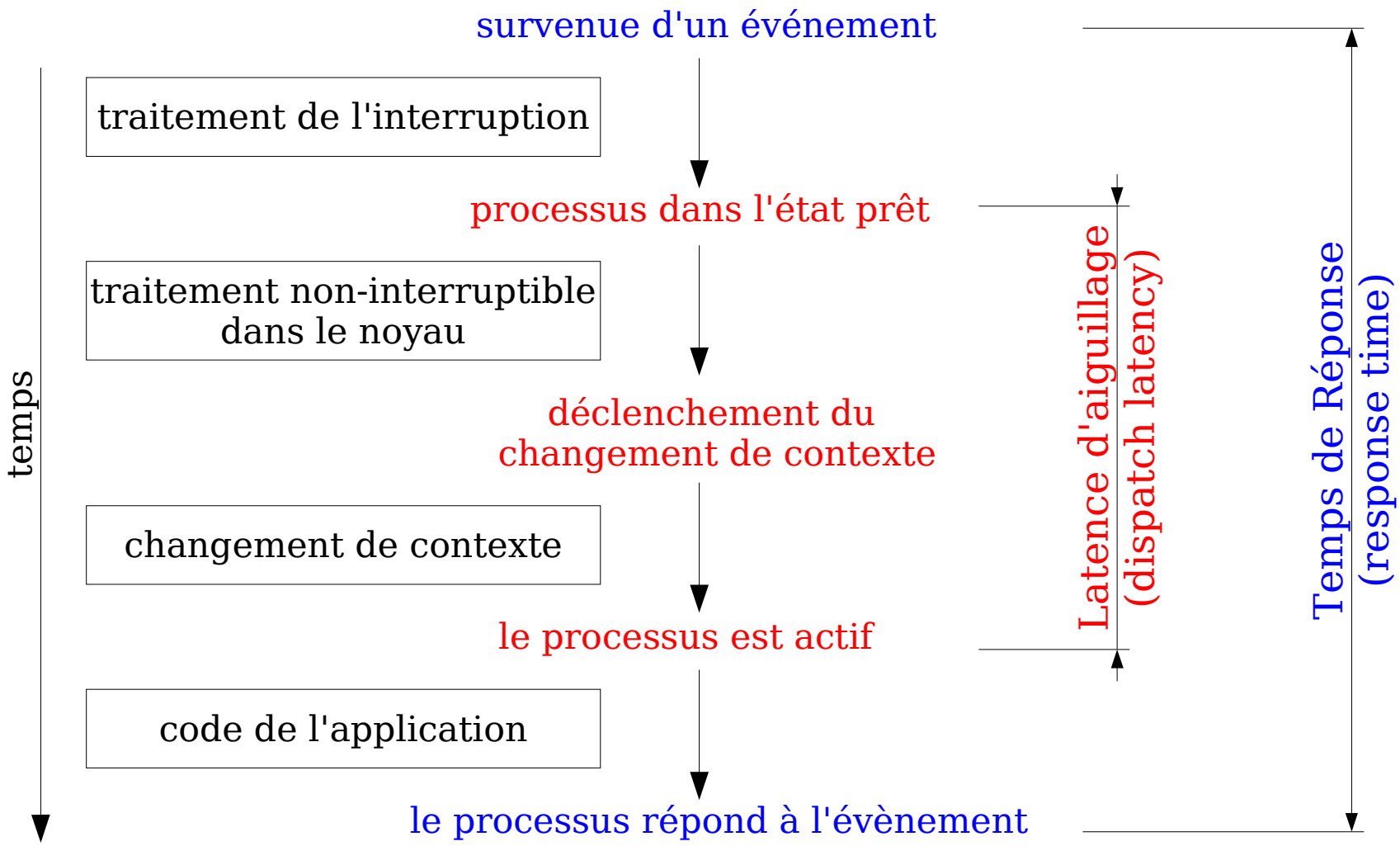
Analyse de l'Ordonnanceur Traditionnel

- Avantages
  - Simple et efficace pour un système généraliste
    - Càd mélangeant tâches intéractives et arrière-plan
  - Recalcul dynamique de priorités évite la famine
  - Favorise les tâches IO-bound
- Inconvénients
  - Mauvaise extensibilité (quand le processus augmente)
  - Impossible de réserver une partie de la ressource à un groupe de processus particulier
  - Pas de garantie du temps de réponse
  - Peu de contrôle sur le niveau de priorité (nice trop simpliste)
  - Problème d'inversion de priorité
    - un processus prioritaire peut attendre longtemps dans le noyau

L'Ordonnanceur SVR4

- Conception revue
  - Ajoute le support des tâches temps réel
  - Sépare Politique et Implémentation
    - Définit la notion de classe d'ordonnancement
      - Une couche d'implémentation commune à toute les classes
      - Une couche d'implémentation spécifique à chaque classe
        - Principes de prog objet classique (comme VFS, mm, ...)
  - Fournit un meilleur contrôle aux processus
    - Priorité, politique d'ordonnancement
  - Permet l'ajout de nouvelles politiques de façon modulaire
    - Y compris de façon dynamique (sans redémarrer)
  - Limite la "latence d'aiguillage" (dipatch latency)

Temps de Réponse et Latence d'Aiguillage



Réduction de la Latence d'Aiguillage dans SVR4

- Cause principale : noyau non-interruptible
  - Solution : rendre le noyau interruptible
  - Pb : Implémentation du noyau plus délicate
    - Problèmes de concurrence, sections critiques, ...
- Solution intermédiaire SVR4
  - Introduire quelques **points de préemption**
    - Endroit "sûrs" dans le noyau
    - Avant des traitements potentiellement longs
    - Exemples :
      - Avant de parcourir les chemins dans l'arborescence
        - Vérification des permissions à chaque niveau
      - Dans open() avant de créer un nouveau fichier
      - Avant de libérer les pages mémoire d'un processus

La Couche d'Implémentation Commune de SVR4

- Définit 160 niveaux de priorité
  - Au lieu de 128 dans ordo. traditionnel
- Une file séparée par niveau
  - Au lieu d'une commune pour 4 niveaux
- Niveaux de priorité dans l'ordre croissant des valeurs
  - Au lieu de décroissant
    - 160 = prio max dans SVR4
    - 128 = prio min dans traditionnel
- Fournit une API pour insérer/retirer des processus dans chaque file
  - Insertion (en début ou en fin de file) + Retrait
  - Fonctions offertes aux couches d'implém. spécifiques

Les Classes d'Ordonancement SVR4

- Une Classe = implémentation d'une politique
  - Par défaut 3 classes définies dans SVR4
    - Temps réel : priorités 0 à 59
    - Système : priorités 60 à 99
    - Temps partagé : priorités 100 à 159
- Chaque classe décide (implémente)
  - Son propre calcul de priorité
  - Sa propre stratégie d'ordonancement (élection)
- Chaque processus appartient à une classe donnée
  - A la création : identique au processus parent
  - Ensuite : changement possible avec AS priocntl()

La Classe "Temps partagé" de SVR4

- Priorités dynmique avec tourniquet
  - Même stratégie que ordonancement traditionnel
  - Mais implémentation différente
    - La tranche de temps donnée dépend du niveau de priorité
      - Valeurs conservées statiquement dans une table
    - Plus la priorité est **faible**, plus la tranche est **grande**
- Implémentation dite "dirigée par les évènements"
  - La priorité change en réponse à certains événements
    - Augmente quand le processus est bloqué
      - Une table donne les paramètres pour chaque niveau de priorité
    - Diminue quand il épuise sa tranche de temps
    - Avantage : le calcul ne porte que sur le processus courant
      - Recalcul indépendant du nombre de processus

La classe "Temps Réel" de SVR4

- Priorité **fixe** avec tourniquet
  - La durée de la tranche de temps dépend du niveau de priorité
    - Plus la priorité est faible, plus la tranche est longue
- Priorité les plus élevées de toutes les classes
  - Supérieures aux priorités système
    - Mais le noyau ne peut être interrompu que sur un point de commutation
  - Les moins prioritaires sont obligés d'attendre...
    - Que l'élu se décide à s'endormir (à attendre un evt)
  - Il faut être root pour faire passer un processus dans cette classe (et fixer sa priorité)

Analyse de la classe "Temps réel" SVR4

- Favorise les tâches IO-bound
- Points de préemption réduisent la latence d'aiguillage
  - Mais pas aussi efficace qu'un vrai noyau préemptible
    - Inversion de priorité existe toujours (moindre que trad)
- Gros inconvénient : le réglage
  - Il faut remplir des tables avec différents paramètres en fonction de chaque niveau de priorité
  - Les expériences montrent que le réglage est difficile, voir impossible à trouver
    - Exemple : vidéo + tâche interactive
      - Pb : dans quelle classe mettre serveur X
        - Si vidéo + serveur X dans classe TR, alors tout le reste s'arrête (la souris ne bouge plus, le clavier ne répond plus...)

L'Ordonancement dans Solaris

- Conception encore nettement améliorée / SVR4
  - Noyau totalement interruptible (enfin !)
    - Réécriture complète avec implémentation multi-thread
  - Support Shared-Memory Multi-processor (SMP)
    - Facile avec un noyau interruptible ...
- Apporte des solutions à certains problèmes connus
  - Ordonancement caché
    - Les appels différés sont normalement exécutés sur le compte d'un processus non concerné
      - Solaris utilise un thread spécifique séparé pour les callouts
      - Les callouts des processus temps réel sont traités séparément
  - Inversion de priorité
    - Un processus peut se faire "doubler" ...

Inversionde Priorité : un Exemple

- P1(prio=10) détient ressource R1
- P2(prio=20) arrive et veut aussi R1
  - P2 est obligé d'attendre que P1 relâche la ressource
- P3(prio=15) arrive, il n'a pas besoin de R1
  - Il a une priorité plus forte que P1
  - Il n'a pas de raison d'attendre
  - Donc il interrompt P1 !
- Conclusion
  - P3 est passé devant P2 alors que sa priorité est plus faible
- Quelle solution ?
  - Sachant que P1 bloque P2, P3 devrait attendre ...

Héritage de Priorités dans Solaris

- Permet de résoudre le problème d'inversion de priorités
- Principe
  - Quand un processus P2 est bloqué par un autre P1
    - Si P2.prio > P1.prio
      - P1 **hérite temporairement** de la priorité P2.prio
      - P1 retrouve sa priorité initiale quand il libère la ressource
    - Sinon, on ne change rien
  - Quand P3 arrive avec P1.prio < P3.prio < P2.prio
    - En fait on a temporairement P3.prio < P1.tmp (=P2.prio)
    - Donc P3 ne peut pas doubler P2
- Limitations
  - Suppose d'identifier précisément le détenteur ...

Autres Stratégies : 1 – Partage Equitable (*fair-share*)

- Principe
  - Les processus sont organisés en groupes
    - Evéntuellement un seul processus
    - Ou plusieurs appartenant au même utilisateur ...
  - L'ordonanceur alloue une quantité donnée de CPU à chaque groupe
  - Quand un groupe n'utilise pas tout, le reste est partagé avec les autres groupes
- Analyse
  - Pratique dans les environnements ou on facture le temps d'utilisation
  - Offre une garantie forte de disponibilité aux processus exigeants

Autres Stratégies : 2 – Ordo. Dirigé par Date Limite

- Constat : les applications TR doivent souvent répondre dans un délai donné
- Principe :
  - Suppose qu'on sait quantifier la durée de calcul (CPU) nécessaire avant expiration de la date limite
  - Plus la date limite approche, plus la priorité augmente
    - En fonction de ce qu'il reste à faire (calculer) avant la date limite
  - Exemple d'Implémentation (FORTUNIX)
    - Différentes classes (temps réel fort/faible, batch, ...)
    - Dans certaines classes, priorité = proximité date limite
      - Le CPU est systématiquement donné à celui qui a la date limite la plus proche dans la classe la plus prioritaire

Autres Stratégies : 3 – Ordonanceur à 3 niveaux

- Constat : manque de contrôle d'admission
  - Les autres ordonnanceurs ne limitent pas le nombre de tâches
    - Ca peut aboutir à la surcharge
      - et donc au non respect des éventuelles contraintes temps réel ...
- Principe :
  - Différentes classes : périodique (ex: vidéo), apériodique mais avec contrainte de latence d'aiguillage, et arrière plan
  - Avant d'accepter une nouvelle tâche exigeante
    - Allocation de **toutes** les ressources nécessaire (CPU, bande passante disque, ...)
      - Si allocation réussie, la tâche est acceptée