Systèmes temps-réels

Programmation multitâche et ordonnancement

Matthieu Lemerre

March 13, 2008

Programmation multitâche

Un système temps-réel dispose:

- De plusieurs ressources (CPU(s), port série, etc...)
- capables d'assurer une fonction à la fois

afin de remplir:

- Plusieurs fonctionnalités
- Un objectif fonctionnel = 1 ou plusieurs tâches
- Une tâche = 1 ou plusieurs fonctions

La mise en œuvre du système suppose une allocation des tâches sur les diverses ressources au cours du temps

Faire cette allocation au cours du temps, c'est ordonnancer les traitements

Intérêt d'une conception multitâche

- Initialement : optimiser l'utilisation du matériel
  Ex: paralléliser les I/O avec le CPU
  Donc, optimisation des temps d'execution moyen des diverses tâches à accomplir
- Traiter des évenements asynchrones
  (interruptions à des instants pas entièrement déterminés)
- Profiter des multiples ressources de calcul
  Architectures multiprocesseur

Intérêt d'une conception multitâche (2)

Gérer un objectif temps-réel global:

- Des échelles de temps différentes (les traitements courts passent avant les longs)
- Des degrés de criticité différentes (si on ne peut pas tout faire tourner, les traitements critiques passent en premier)

Objectifs souvent contradictoires:

- Maximiser le nombre de fonctionnalités correctement acquittées
- Minimiser le coût (dimensionnement)

Classification des tâches par importance

Critiques : doivent **toujours** être assurées (garantir des propriétés de **sûreté**)

Essentielles : doivent être assurées autant que possible, c'est à dire au moins de temps en temps (garantir des propriétés de **vivacité**)
Dans tous les cas, pour un système temps-réel, il faut assurer la **ponctualité** de tous les traitements

Optionnelles : ...

sûreté : Montrer qu'une chose ne peut pas se produire

- Ex: Telle assertion est toujours vérifiée

vivacité : Montrer qu'une chose se produira au bout d'un certain temps (indéfini)

- Ex: mon algorithme termine

ponctualité : Montrer que les traitements se terminent à temps

Programmation multitâche

Ordonnancer correctement un système, c'est:

- 100% des tâches critiques doivent respecter leurs contraintes (preuve)
- Pour les tâches essentielles: "best effort"

Les contraintes diffèrent:

- Ressources réquisitionnables (ex: CPU)
- non réquisitionnables (ex: DSP)
- multiples ou non

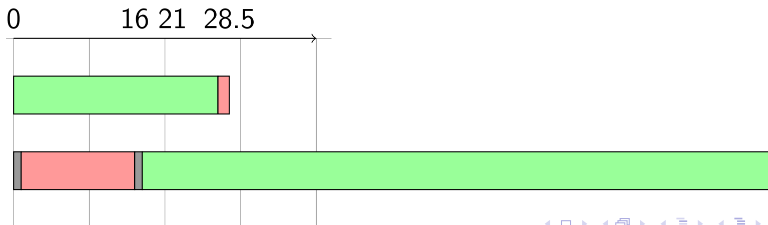
→ complexité du problème d'ordonnancement (NP-complet en général)

	tortue	lièvre
vitesse	1	10
durée de commutation	1	0
priorité	à l'échéance	au premier

2 travaux:

- Travail A: durée = 270 échéance 320
- Travail B: durée = 15 échéance 21

A arrive juste avant B...



Travail (job)

Attributs temporels d'un travail:

- Date de début au plus tôt (demande)
- Date de fin au plus tard (échéance)
- Durée d'exécution (besoin)

Notes:

- La durée d'exécution dépend de la vitesse du processeur!
- La durée d'exécution est théorique, (non constante en pratique) : emploi de WCET

Paramètres dérivés:

- Date de début au plus tard
- Durée jusqu'à la date de début au plus tard (laxité)
- date de fin au plus tôt (pb de réquisition)

Travail (job)

Attributs temporels d'un travail:

- Date de début au plus tôt (demande)
- Date de fin au plus tard (échéance)
- Durée d'exécution (besoin)

Notes:

- La durée d'exécution dépend de la vitesse du processeur!
- La durée d'exécution est théorique, (non constante en pratique) : emploi de WCET

Paramètres dérivés:

- Date de début au plus tard
- Durée jusqu'à la date de début au plus tard (laxité)
- date de fin au plus tôt (pb de réquisition)

Travail (job)

Attributs temporels d'un travail:

- Date de début au plus tôt (demande)
- Date de fin au plus tard (échéance)
- Durée d'exécution (besoin)

Notes:

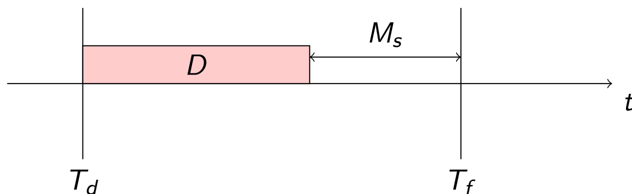
- La durée d'exécution dépend de la vitesse du processeur!
- La durée d'exécution est théorique, (non constante en pratique) : emploi de WCET

Paramètres dérivés:

- Date de début au plus tard
- Durée jusqu'à la date de début au plus tard (laxité)
- date de fin au plus tôt (pb de réquisition)

Marge statique

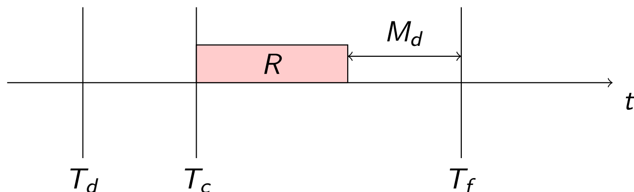
$$M_s = (T_f - T_d) - D$$



- $M_s \geq 0$
- Si $M_s = 0$, pas de choix

Marge dynamique

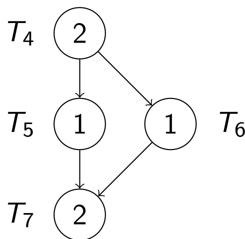
$$M_d = (T_f - T_c) - R$$



- M_d = laxité
- M_d évolue dynamiquement
  - Pour le travail active
  - Pour les travaux non actives

Relation entre les travaux

- graphe de dépendances/précedence (calcul des dates au plus tôt et au plus tard)



- concurrence explicite
- concurrence implicite (plages de temps qui se chevauchent)
- accès concurrents aux ressources, sérialisations...

Notion de tâche

Une tâche est une entité qui crée des des travaux (job release). On dit aussi que la tâche s'active.

Trois classes majeures:

- **Periodiques** : Crée un travail tous les P (> 0)
- **Sporadiques** : Créations de travaux séparées par un laps de temps minimal I
- **Aperiodiques** : Modèle le plus général (comprends les 2 précédentes).
  Les creations de travaux doivent quand même être connues (exemple: graphe)

Plan d'ordonnancement

Plan d'ordonnancement, ou ordonnancement: c'est une méthode de prévision pour l'allocation des ressources (phase de conception).

Un ordonnancement est dit optimal si toutes les contraintes temporelles sont respectées.

On dit qu'il y a surcharge, lorsque le volume des travaux à ordonnancer est tel que tous les plans d'ordonnancements conduisent au non-respect d'au moins une travail (il n'existe pas de plan optimal)

Un algorithme d'ordonnancement est dit optimal,si pour une classe de problèmes donnée,il produit des plans optimaux.

Types d'algorithmes d'ordonnancement

Statique : Le plan d'ordonnancement est décidé avant exécution (hors ligne).

Dynamique : Le plan d'ordonnancement varie pendant l'exécution. (en ligne)

- **Non préemptifs** : On ne peut pas interrompre l'exécution d'un programme, pour la reprendre ensuite
- **Préemptifs** : On peut.
  - **Priorité fixe** La priorité d'une tâche ne change pas à l'exécution (aussi appellé priorité statique)
  - **Priorité dynamique** La priorité d'une tâche varie dynamiquement.

Types d'algorithmes d'ordonnancement

- **Statique** : Le plan d'ordonnancement est décidé avant exécution (hors ligne).
- **Dynamique** : Le plan d'ordonnancement varie pendant l'exécution. (en ligne)
  - **Non préemptifs** : On ne peut pas interrompre l'exécution d'un programme, pour la reprendre ensuite
  - **Préemptifs** : On peut.
    - **Priorité fixe** La priorité d'une tâche ne change pas à l'exécution (aussi appellé priorité statique)
    - **Priorité dynamique** La priorité d'une tâche varie dynamiquement.

Types d'algorithmes d'ordonnancement

- **Statique** : Le plan d'ordonnancement est décidé avant exécution (hors ligne).
- **Dynamique** : Le plan d'ordonnancement varie pendant l'exécution. (en ligne)
  - **Non préemptifs** : On ne peut pas interrompre l'exécution d'un programme, pour la reprendre ensuite
  - **Préemptifs** : On peut.
    - **Priorité fixe** La priorité d'une tâche ne change pas à l'exécution (aussi appellé priorité statique)
    - **Priorité dynamique** La priorité d'une tâche varie dynamiquement.

Statique vs dynamique

Statique:

- \+ Prévision peut être infini
- \+ La connaissance du plan est totale (garantie,flot d'instruction unique)
  - $\rightarrow$ vérification, simulation (ponctualité, relations entre tâches)
- \- Rigide, nécessite une régularité des traitements
- \- Utilisation de mutex impossible, partage des ressources
- \- sous-optimale, peut-être inefficace (coût)

Utile:

- pour la programmation en boucle
- ou garanties d'allocation de temps CPU (ordonnancement par partitions)

Statique vs dynamique (2)

Dynamiques:

- + Flexibles
- + Meilleures décisions car mieux informés (connaissance de la vraie date de fin)...
- - Prédictibles? Instables
- - (a priori moins sûrs...) (besoin d'une preuve)

Modèles - hypothèses

Sur les tâches:

types	:	séquentielles, parallèles
relations	:	mutuellement dépendantes, indépendantes
valeurs	:	identiques/différentes, constantes/dépendantes de l'instant de terminaison (anytime)
abandon		si violation de contrainte: exigé, autorisé, interdit

Modèles - hypothèses

Sur les ressources:

multiplicité : 1 ou n, équivalentes ou non

mode d'acces : centralisé ou distribué (ressources mémoires)

requisition/preemption :

- possible toujours (avec ou sans pertes)
- possible par moment
- impossible

Modèles - hypothèses

Sur les lois evenementielles:

- Connues totalement ou partiellement (non temps-réel)
- Predeterminees ou statistiques

Sur les lois d'activations des tâches

- instants d'activations des tâches
- instants d'accès aux ressources
- instant délimitant les fenêtres de réquisitions possibles/impossibles

Un énoncé de problème = un choix parmi toutes ces hypothèses

Présentation de diverses classes de problème avec des solutions connues, souvent optimales (cadre hypothèses/contraintes)

Dans la suite:

H_1 = hypothèses issues de l'énoncé du problème, identifie la classe de problème

H_2 = condition sur les données quantitatives du problème (hypothèses de faisabilité à vérifier)

Algorithmes connus

Algorithmes de bases, les autres sont pour la plupart une combinaison d'entre eux.

- RMS
- FCFS
- RR
- HPF
- LLF
- EDF

Note: pour des tâches périodiques, on peut obtenir un ordonnancement statique en simulant un ordonnancement dynamique sur le PPCM des périodes!

Rate monotonic scheduling

Inventé par Liu & Layland

Hypothèse de modèle (H1)

- Algorithme à priorité fixe
- Préemption possible
- Chaque tâche est périodique
- Pas de dépendances entre tâches
- L'échéance est la période
- On affecte les priorités dans l'ordre inverse de la période

Hypothèse de faisabilité (H2)

- Il existe une condition suffisante (CS):
- Ordonnancement sûr si critère vérifié

Critère théorique

- n tâches
- C_i = durée d'exécution
- T_i = période = deadline de chaque travail

Analyse du taux d'utilisation (W) CPU:

$$\sum_i \frac{C_i}{T_i}$$

Si $W \leq U(n)$, RMS est optimal

$$\begin{aligned} U(n) &= n * (2^{1/n} - 1) \\ U(1) &= 1 \\ U(2) &= 0.83 \\ U(3) &= 0.78 \\ U(\infty) &= 0.69 = \log 2 \end{aligned}$$

Condition suffisante:

$$W = \sum C_i / T_i$$

$$U(n) = n * (2^{1/n} - 1)$$

CS: $W \leq U(n)$
CN: $W \leq 1$

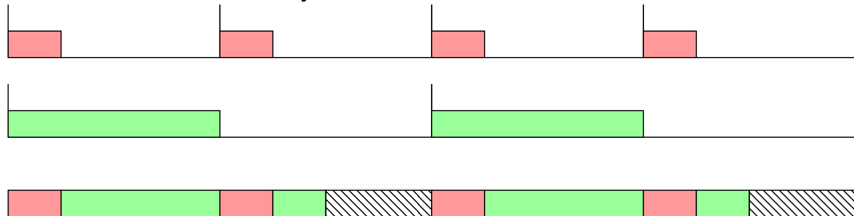
Pour W entre CS et CN:
Il faut réaliser le plan d'ordonnancement sur un horizon fini (le PPCM des périodes)

Exemple RMS

A: $C = 1 T = 4 \rightarrow P = 2$
B: $C = 4 T = 8 \rightarrow P = 1$

$W = 1/4 + 4/8 = 0.75$

CS: $W \leq 0.83$ donc le système est ordonnancable.



Note: on obtient un ordonnancement statique!

Conclusion RMS

Avantages:

- peut être étendu à une tâche apériodique (serveur sporadique)
- peut être étendu à la surcharge (les tâches de priorité haute ne sont pas impactées par les tâches de priorité basse)
- Simple à implémenter (décisions en O(1))

Inconvénients:

- Hypothèses très simples (rarement rencontrées)
- Famine si bug dans une tâche de haute priorité $\longrightarrow$ vérifier le temps pris par chaque tâche

High priority first

Les tâches ont une priorité fixe La tâche de plus haute priorité en mesure de s'exécuter prend la main (RMS est un cas particulier)

Exemple: une priorité par numéro d'interruption

- ++ très proche du HW (E/S asynchrones)
- + → Ordonnancement peut être fait en hard
- + ponctualité pour une tâche
- - pour les autres?
- - sûreté (contexte)
- - vivacité (si beaucoup de tâches de haute priorité)

- Changement possible des priorités en ligne (ex: UNIX)
- → amélioration des défauts (vivacité)
- On peut exercer des contrôles sur les temps de traitements, mais les analyses restent compliquées

Exemple

Tâche A: D = 2 P = 3

- Prête à s'exécuter
- mais requête à la tâche C pendant ses traitements (dépendance)

Tâche B: D = 2 P = 2

- Prête à s'exécuter

Tâche C: D = 1 P = 1

- En attente

Attention: inversion de priorité!
→ à cause de l'héritage de priorité, dans un gros système toutes les tâches se retrouvent avec la même priorité

Exemple

Tâche A: D = 2 P = 3

- Prête à s'exécuter
- mais requête à la tâche C pendant ses traitements (dépendance)

Tâche B: D = 2 P = 2

- Prête à s'exécuter

Tâche C: D = 1 P = 1

- En attente

Attention: inversion de priorité!
$\longrightarrow$ à cause de l'héritage de priorité, dans un gros système toutes les tâches se retrouvent avec la même priorité

Round Robin (tourniquet)

On partage le temps de façon “équitable”
Principe du tourniquet sur les tâches en mesure de s’exécuter
On donne la main à une tâche pour (au plus) k unités de temps
Avantages:

- Vivacité
- Paralléliser les IO et les traitements

Inconvénients:

- Ponctualité (temps long entre moments où la tâche recoit le processeur)

UNIX/Windows/etc..

Versions évoluées avec temps différent selon la tâche (2s A, puis 1s B)
En pratique, RR est couplé avec HPF

First Come/First Served

Très simple et très rudimentaire

Avantages

- Vivacité (si les tâches se terminent)
- Non préemptif

Inconvénients

- Ponctualité
- Sécurité

Utile pour garder un ordre de traitement implicite (pour les E/S)
Ex: spooler d'impression
Note: FCFS est souvent la stratégie "par défaut" pour beaucoup de ressources (ex: mémoire ou ports TCP/IP sous UNIX)

Least Laxity First

Hypothèse de modèle (H1)

- Priorité change dynamiquement
- tâche apériodique
- pas de dépendances entre tâches
- échéances connues
- durées d'exécution connues
- la priorité est l'inverse de la laxité
- préemption

Hypothèse de faisabilité (H2)

- Optimal si pas de surcharge

Earliest Deadline First

Hypothèse de modèle (H1)

- Dynamique
- Tâche apériodique
- Pas de dépendance entre tâche
- échéances connues
- durées inconnues
- La priorité est l'inverse de l'échéance relative
- préemption

Hypothèse de faisabilité (H2)

- Optimal si pas de surcharge

Combinaisons

Exemple de combinaisons:

- Ordonnanceur UNIX
- Systèmes à partitions:
  - Avionique IMA (Partitions ordonnancées statiquement + ordonnancement à priorité fixe)
  - OSEK/PikeOS (Priorités relatives, mélange sporadique/périodique)

Cas multiprocesseur

En monoprocesseur: algorithmes optimaux "en ligne" (EDF, LLF)
En multiprocesseur: nécessité de "connaitre le futur" pour ordonnancements optimaux!
(Global EDF non optimal)
Différentes stratégies:

- Partitionnement (migrations interdites)
- Algorithmes globaux (pas forcément mieux!)

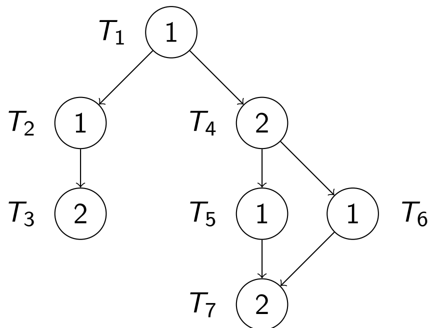
Différence théorie/pratique

Hypothèses précédentes presque jamais vérifiées:

- L'indépendance des traitements (souvent, mutex ou graphes de dépendances)
- Preemption: prends un certain temps
- Intégration de tâches critiques/non critiques: souvent surcharge

Hypothèses de cas concret:

- Graphe de dépendance
- $K = 2$ (plus d'une ressource
- pas de réquisition



Temps minimum : 6

Toutes les tâches doivent être terminées en 6 unités de temps

- FCFS
- $\sum_i T_i = 10$

T_1	T_2	T_3	T_3	T_6	T_7	T_7
–	T_4	T_4	T_5	–	–	

	Date de début au plus tôt	Date de fin au plus tard	Marge
T_1	0	1	0
T_2	1	4	2
T_3	2	6	2
T_4	1	3	0
T_5	3	4	0
T_6	3	4	0
T_7	4	6	0

- Pas de marge négative
- Marges nulles = chemin(s) critique(s)

- Préallocation de T_7, T_6, T_5, T_4

	T_4	T_4	T_5	T_7	T_7
			T_6		

- Préallocation de T_7, T_6, T_5, T_4
- Placement de T_1

T_1	T_4	T_4	T_5	T_7	T_7
			T_6		

- Préallocation de T_7, T_6, T_5, T_4
- Placement de T_1
- Placement de T_3 (rappel: pas de préemption)

T_1	T_4	T_4	T_5	T_7	T_7
			T_6	T_3	T_3

- Préallocation de T_7, T_6, T_5, T_4
- Placement de T_1
- Placement de T_3 (rappel: pas de préemption)
- Placement de T_2 (plusieurs choix possibles)

T_1	T_4	T_4	T_5	T_7	T_7
–	T_2	–	T_6	T_3	T_3