2ème Partie

Ordonnancement

Prédictibilité temporelle des STR

Prédictibilité = assurer *a priori* que
toutes les tâches respectent leurs échéances

Les 2 piliers sont:

- l'analyse de **pire temps d'execution** (WCET – Worst Case Execution Time)
  - problèmes:
    - liés au programme (structure dynamique – boucles, branches…; appels externes – OS, MW…)
    - liés au matériel (cache, pipeline, exécution prédictive…)
  - méthodes: analyse statique de programmes, interprétation abstraite, analyse dynamique, hybride…
- **l'ordonnancement** des tâches

Ordonnancement (rappel)

Politique d'ordonnancement :

politique d'allocation des tranches de temps processeur

(i.e., les moments où une tâche est livrée au processeur / suspendue)

Objectif de l'ordonnancement : assurer le respect des échéances *de manière prouvable*

Une ***méthode d'ordonnancement*** est caractérisée par:

- la méthode effective de construction de l'*emploi du temps* (schedule) du processeur (online ou offline**)**
- le critère de test d'ordonnançabilité (offline) – prédiction du comportement « au pire cas »

Types de tâches

- Tâches périodiques
  - Déclanchées par le temps. Caractéristiques connues à l'avance.
  - T_i est caractérisée par:
    - p_i – période d'arrivée
    - c_i – temps de calcul au pire cas (WCET)
    - d_i – échéance *relative* à l'arrivée
  - Exemple : Guidage-Navigation-Contrôle (GNC) dans un AOCS (ex. Ariane-5)

- Tâches apériodiques
  - Déclanchées par un événement extérieur. Caractéristiques partiellement non-connues.
  - T_i est caractérisée par:
    - c_i – temps de calcul au pire cas (WCET)
    - d_i – échéance relative à l'arrivée
    - (éventuellement) a_i – temps d'arrivée / contraintes

- Tâches sporadiques
  - tâches apériodiques avec temps minimum entre arrivées connu.
  - Exemple : Séparation des étages dans un lanceur (Ariane-5)

Un modèle basique

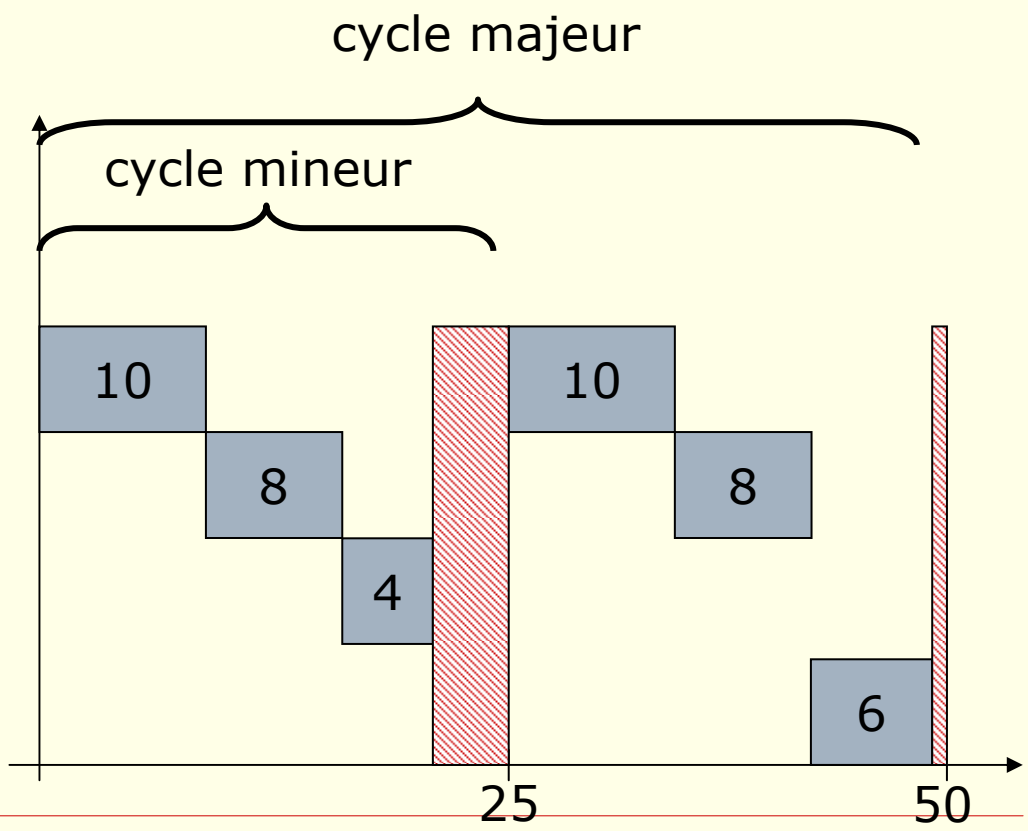
- Le système consiste d'un ensemble fixé de tâches
- Toutes les tâches sont périodiques
- les tâches sont indépendantes (pas de communication/synchronisation)
- on suppose un temps de changement de contexte zéro

Ordonnancement cyclique

- Conception concurrente, mais implantation séquentielle cyclique
- Les tâches sont mappées sur un ensemble de procédures

Exemple:

Tâche	P_t	C_t
A	25	10
B	25	8
C	50	4
D	50	6



Implémentation

⇒ interruption horloge: période = cycle mineur

```
while(true) {
    A(); B(); C();
    attendre interruption;
    A(); B(); D();
    attendre interruption;
}
```

Discussion

pour:

- pas de processus à l'exécution
  - espace mémoire partagé
  - pas besoin de protection de régions critiques – accès concurrent impossible
- exécution complètement déterministe

contre:

- impossible d'introduire des activités sporadiques
- cycles « longs » difficiles à gérer
- nécessité de découper des calculs longs sur plusieurs cycles mineurs ⇒ erreurs
- emploi de temps difficile à obtenir (NP-hard)

Ordonnancement préemptif ou non

- Ordonnancement non-préemptif
  - quand une tâche commence, elle va jusqu'au bout
  - … mais des fois c'est impossible de tenir les délais:

Tâche	P_t	C_t	D_t
A	1000	25	1000
B	10	1	2

Chaque fois que A exécute, B va rater 2 fois son échéance

- Ordonnancement préemptif
  - la transition préempt est permise;
    une tâche peut être interrompue et reprise plus tard
  - la préemption a lieu quand une tâche de *plus grande priorité* arrive
  - permet une meilleure occupation du processeur
  - …donc permet de rendre ordonnançable des systèmes qui ne le sont pas autrement
  - mais : implique une surcharge pour le changement de contexte

Ordonnancement préemptif à priorité fixe (FPPS)

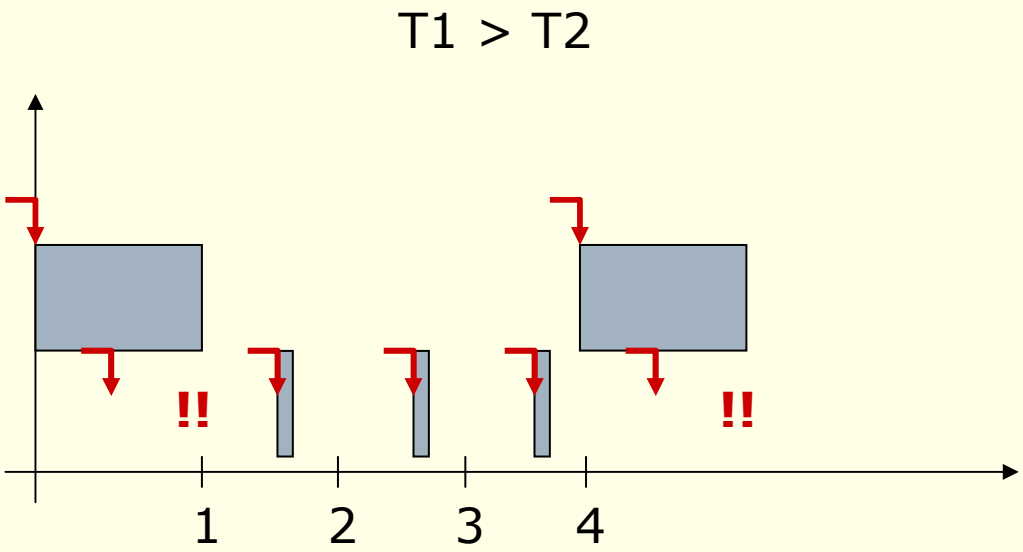
- l'approche la plus utilisée dans les systèmes réels
- chaque tâche a une priorité fixe, *statique*, *calculée off-line.*
  (Toutes les instances de la tâche ont la même priorité.)
- à tout moment, la tâche la plus prioritaire a le CPU
  - changements de contexte:
    - *préemption :* quand une tâche arrive, si elle est plus prioritaire que la tâche en cours
    - quand la tâche en cours se termine ($\rightarrow$ idle)

Comment affecter les priorités

Exemple : la centrale nucléaire la moins chère au monde(*)

- tâche 1 : barres de contrôle (cœur du réacteur)
- tâche 2 : vanne du lave-vaisselle (cafeteria)

Tâche	P_t	C_t
T1	4	1
T2	1	0.02



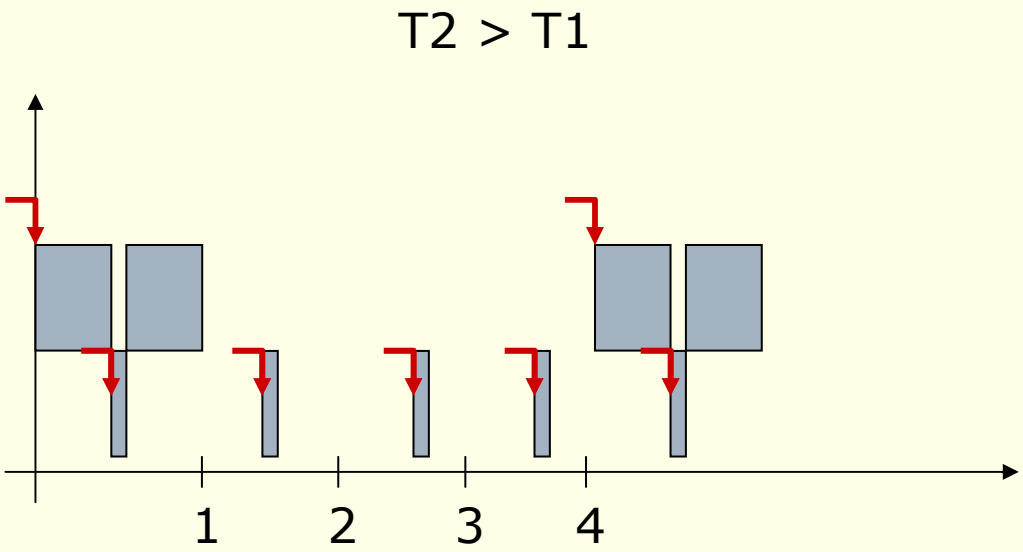
(*) thanks to D.C. Locke for this example

Comment affecter les priorités

Exemple : la centrale nucléaire la moins chère au monde(*)

- tâche 1 : barres de contrôle (cœur du réacteur)
- tâche 2 : vanne du lave-vaisselle (cafeteria)

Tâche	P_t	C_t
T1	4	1
T2	1	0.02



(*) thanks to D.C. Locke for this example

Comment affecter les priorités

Exemple : la centrale nucléaire la moins chère au monde(*)

- tâche 1 : barres de contrôle (cœur du réacteur)
- tâche 2 : vanne du lave-vaisselle (cafeteria)

Tâche	P_t	C_t
T1	4	1
T2	1	0.02

Conclusion:
La priorité **n'est pas basée sur l'importance**, mais uniquement sur les contraintes temporelles !

(*) thanks to D.C. Locke for this example

Notion d'ordonnanceur optimal

Définition: Un algorithme d'ordonnancement est appelé optimal (pour une classe de problèmes*) quand il produit un *emploi du temps processeur* faisable chaque fois qu'un autre algorithme d'ordonnancement peut le faire.

---

* classe de problèmes: par exemple, ensemble de tâches cycliques sans interactions, avec FPPS

Ordonnancement « Rate Monotonic » (RMS)

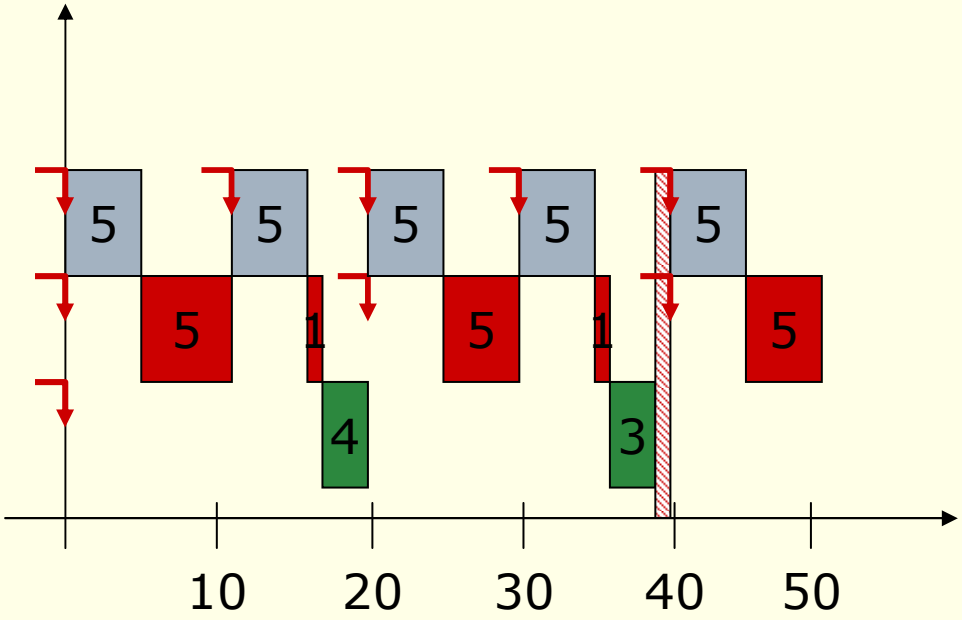
Résultat : Il existe une politique optimale avec priorités statiques, c'est **RMS :**

- affecter les priorités aux tâches dans l'ordre de leur périodes: les tâches avec des périodes plus courtes sont plus prioritaires.

Exemple

Tâche	P_t	C_t
A	10	5
B	20	6
C	50	7

A > B > C

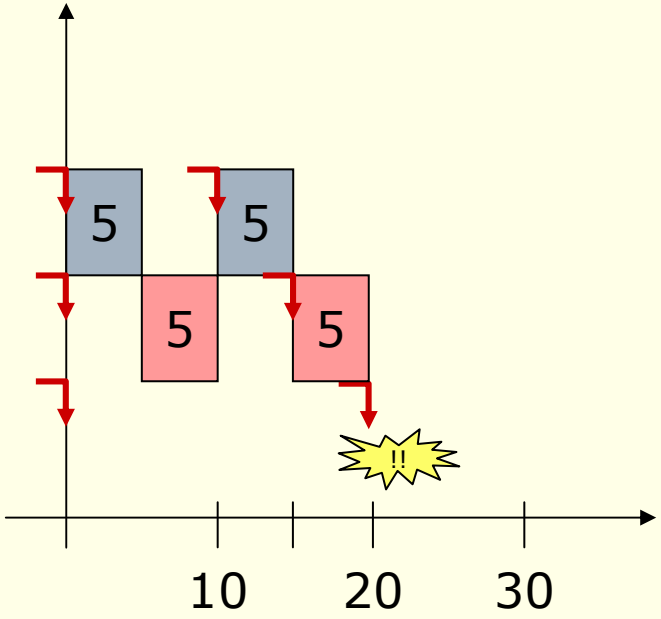


Ordonnançabilité ?

Exemple : ensemble de tâches qui n'est pas ordonnanceable
avec occupation du processeur < 100% (~ 83.4%)

Tâche	P_t	C_t	Occup. CPU
A	10	5	0.5
B	15	5	0.33
C	20	ε	$\varepsilon/20$

A > B > C



Critère statique d'ordonnançabilité RMS

Théorème : Un ensemble de n tâches périodiques indépendantes est ordonnançable par RMS *indifféremment de l'ordre d'arrivée* si :

$$\underbrace{\sum_{i=1}^{n} \frac{c_i}{p_i}}_{\textbf{Utilisation du processeur}} \leq n(2^{1/n} - 1)$$

Critère statique d'ordonnançabilité RMS

$$\sum_{i=1}^{n} \frac{c_i}{p_i} \leq \underbrace{n(2^{1/n} - 1)}_{\mathbf{U(n)}}$$

- la condition est restrictive:

  U(1) = 1 U(2) ≈ 0.828 U(3) ≈ 0.779

  U(4) ≈ 0.756 U(5) ≈ 0.743 U(6) ≈ 0.734

  pour n $\rightarrow \infty$, U $\rightarrow \ln 2$ ≈ 69%

Application du critère

Tâche	P_t	C_t
A	10	5
B	20	6

$$U_A = C_A / P_A = 0.5$$

$$U_B = C_B / P_B = 0.3$$

$$U = U_A + U_B = 0.8 < U(2)$$

$$\Downarrow$$

l'ensemble est ordonnançable

Application du critère

Tâche	P_t	C_t
A	10	5
B	20	6
C	50	7

$U_A = C_A / P_A = 0.5$

$U_B = C_B / P_B = 0.3$

$U_C = C_C / P_C = 0.14$

$U = U_A + U_B + U_C = 0.94 > U(3) \approx 0.779$

Améliorer le critère

- La condition $\sum_{i=1}^{n} \frac{c_i}{p_i} \leq n(2^{1/n} - 1)$ est suffisante, mais pas nécessaire
- Il existe un test précis:

Théorème: Si un ensemble de n tâches indépendantes avec priorités fixes affectées par RMS respecte la 1ère échéance de chaque tâche quand toutes les tâches sont démarrées en même temps (à T_0), alors l'ensemble est toujours ordonnançable.

Un cas particulier

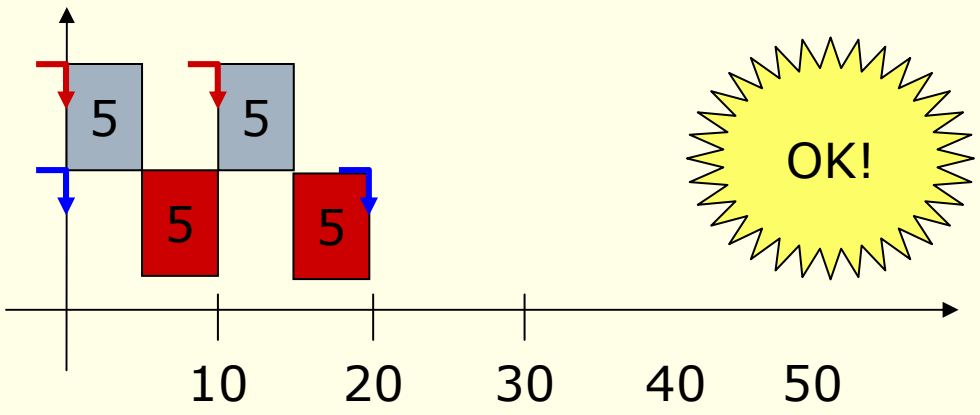
Tâche	P_t	C_t
A	10	5
B	20	10

$$U_A = C_A / P_A = 0.5$$

$$U_B = C_B / P_B = 0.5$$

$$U = U_A + U_B = 1$$

pourant :



Un cas particulier

Tâche	P_t	C_t
A	10	5
B	20	10

$$U_A = C_A / P_A = 0.5$$

$$U_B = C_B / P_B = 0.5$$

$$U = U_A + U_B = 1$$

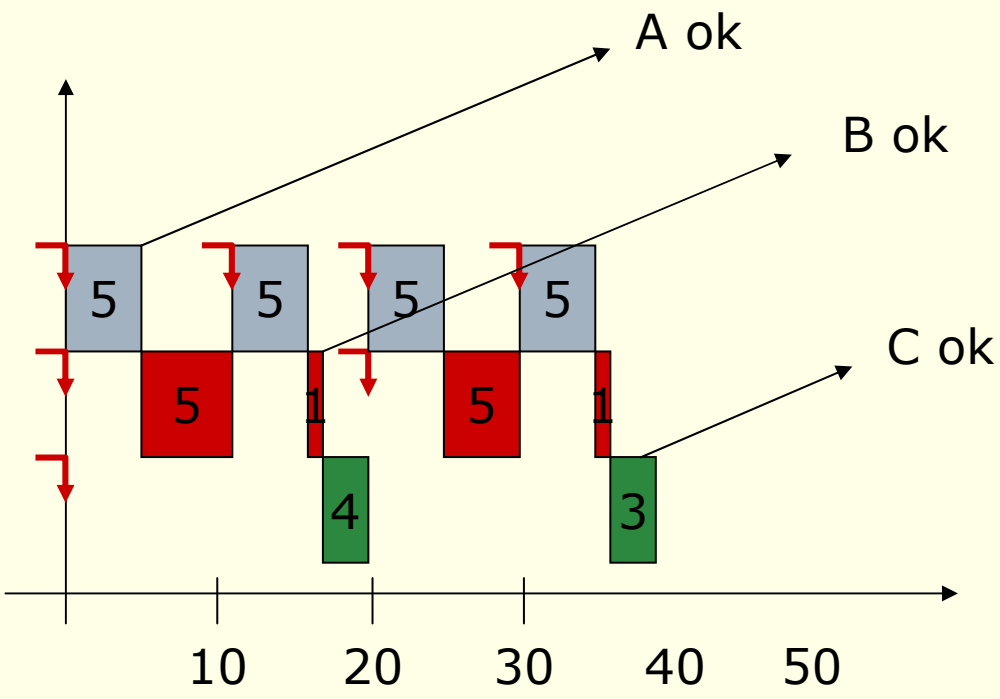
De manière générale : un ensemble de tâches périodiques **harmoniques** est ordonnançable si l'utilisation du processeur est ≤ 100%

Tâches harmoniques : $\{T_1, T_2, \ldots, T_n\}$ avec $P_1 \leq P_2 \leq \ldots \leq P_n$ et $\forall i,j,\ i<j \rightarrow P_i \mid P_j$

Application à l'exemple

Tâche	P_t	C_t
A	10	5
B	20	6
C	50	7

A > B > C



Rendre le critère algorithmique

Pour chaque tâche *i* nous allons calculer le temps de réponse R_i et le comparer a l'échéance D_i (cela fonctionne aussi pour $D_i \neq P_i$)

$$R_i = C_i + I_i$$

où I_i est le temps pendant lequel la tâche *i* est suspendue par des tâches plus prioritaires.

Calcul de R_i

Durant R_i, une tâche *j* > *i* est activée

$$\left\lceil \frac{R_i}{P_j} \right\rceil \text{ fois}$$

Chaque activation prend C_j temps

⇒ l'interférence par la tâche *j* est : $\left\lceil \frac{R_i}{P_j} \right\rceil C_j$

⇒ l'interférence totale I_i est

$$I_i = \sum_{j>i} \left\lceil \frac{R_i}{P_j} \right\rceil C_j$$

Calcul de R_i

L'équation qui caractérise R_i est donc:

$$R_i = C_i + \sum_{j>i} \left\lceil \frac{R_i}{P_j} \right\rceil C_j$$

$\Rightarrow$ R_i est le plus petit point fixe de la fonction monotone

$$F(X) = C_i + \sum_{j>i} \left\lceil \frac{X}{P_j} \right\rceil C_j$$

Calcul de R_i

On peut le calculer par itération:

$$W_0=C_i$$
$$W_1=F(W_0)$$
$$W_2=F(W_1)$$
$$\ldots$$
$$W_{k+1}=F(W_k)$$

on arrête quand $W_k=W_{k+1}$

$R_i=W_k$

L'algorithme

```
w = 0;
next_w = C_i;
while(w != next_w && next_w <= D_i) {
    w = next_w;
    next_w = F(w);
}
if(next_w <= D_i)
    « La tâche T_i est ordonnançable »
else
    « La tâche T_i n'est pas est ordonnançable »
```

Analyse du temps de réponse: exemple

Tâche	P_t	C_t
A	7	3
B	12	3
C	20	5

$$\boxed{R_A = 3}$$

$$W_0^B = 3$$

$$W_1^B = 3 + \left\lceil \frac{3}{7} \right\rceil 3 = 6$$

$$W_2^B = 3 + \left\lceil \frac{6}{7} \right\rceil 3 = 6$$

$$\boxed{R_B = 6}$$

$$W_0^C = 5 \quad W_1^C = 5 + \left\lceil \frac{5}{7} \right\rceil 3 + \left\lceil \frac{5}{12} \right\rceil 3 = 11 \quad W_2^C = 5 + \left\lceil \frac{11}{7} \right\rceil 3 + \left\lceil \frac{11}{12} \right\rceil 3 = 14$$

$$W_3^C = 5 + \left\lceil \frac{14}{7} \right\rceil 3 + \left\lceil \frac{14}{12} \right\rceil 3 = 17 \quad W_4^C = 5 + \left\lceil \frac{17}{7} \right\rceil 3 + \left\lceil \frac{17}{12} \right\rceil 3 = 20 = W_5^C \quad \boxed{R_C = 20}$$

Exercice

Tâche	P_t	C_t
A	75	9
B	35	20
C	20	5

- est-ce ordonnançable par RMS ?
- quel temps de réponse pour chaque tâche ?
- si A avait besoin de plus de temps CPU, jusque où peut-elle s'étendre en FPPS ?
- jusque où A peut-elle s'étendre en EDF ?

DMS : généralisation de RMS

[cas où $d_i \leq p_i$]

Les tâches avec des échéances (relatives) plus courtes sont plus prioritaires

Critère basé sur l'utilisation CPU :

$$\sum_{i=1}^{n} \frac{c_i}{d_i} \leq n(2^{1/n} - 1)$$

- suffisant mais pas nécessaire: il existe des ensembles ordonnançables avec $\sum_{i=1}^{n} \frac{c_i}{d_i} > 1$!!
- critère nécessaire et suffisant: analyse du temps de réponse (la même qu'en RMS)
- algorithme optimal en priorités statiques

RMS & DMS : résumé

- algorithmes optimaux pour affecter les priorités statique en FPPS
- critère suffisant basé sur le taux d'occupation du processeur
- critère nécessaire et suffisant basé sur l'analyse du temps de réponse

Exercice

Soit le système comportant 4 tâches avec le caractéristiques suivantes :

Tâche	P_t	C_t	D_t
A	100	20	100
B	50	12	50
C	35	10	12
D	25	5	15

- quelle politique utiliser?
- quel ordre de priorité ?
- est-ce ordonnanceable?

Earliest Deadline First (EDF)

Objectif : obtenir une meilleure occupation du processeur.

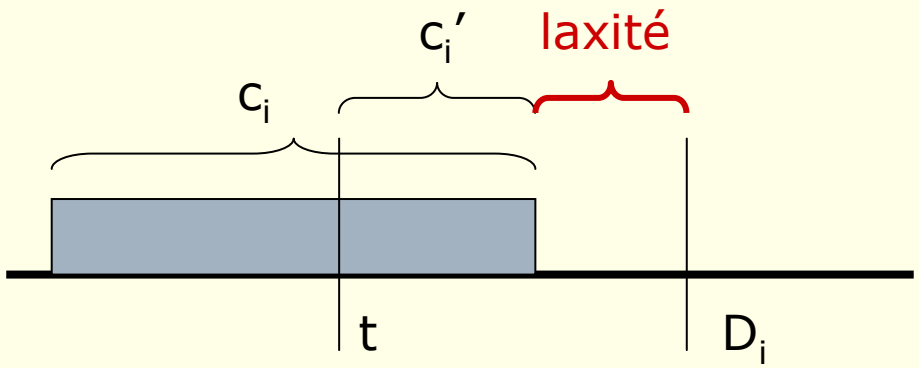
Politique EDF (définition) : à tout moment, la tâche qui a l'échéance la plus proche occupe le CPU.

Remarques:

- toutes les instances d'une tâche n'ont pas la même priorité -- priorité dynamique
- la priorité d'une tâche reste fixe par rapport aux priorités des tâches qui sont *déjà admises à exécution* quand elle arrive
- préemption : quand une tâche arrive, si elle a l'échéance plus proche que la tâche en cours

Une variante de EDF : LLF

- variante : LLF (Least Laxity First)
  - Laxité d'une tâche Ti = D_i – (t + c_i')
    où: D_i – deadline absolut
    t – temps courant
    c_i' – temps de calcul restant



- EDF & LLF sont des algorithmes optimaux d'affectation on-line de priorités pour FPPS

Critères d'ordonnançabilité EDF

[si $d_i = p_i$]

Condition *nécessaire et suffisante*: un ensemble de n tâches est ordonnanceable par EDF ou LLF si l'utilisation du processeur est inférieure à 100%

$$\sum_{i=1}^{n} \frac{c_i}{p_i} \leq 1$$

- difficile de faire mieux ! (sans déborder)

[si $d_i < p_i$]

Condition *suffisante mais pas nécessaire*:

$$\sum_{i=1}^{n} \frac{c_i}{d_i} \leq 1$$

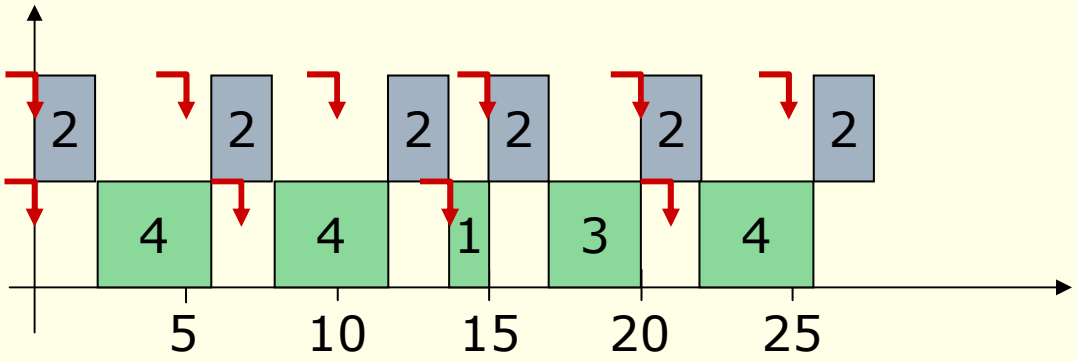
- analyse du temps de réponse plus difficile qu'en RMS

Comparaison RMS - EDF

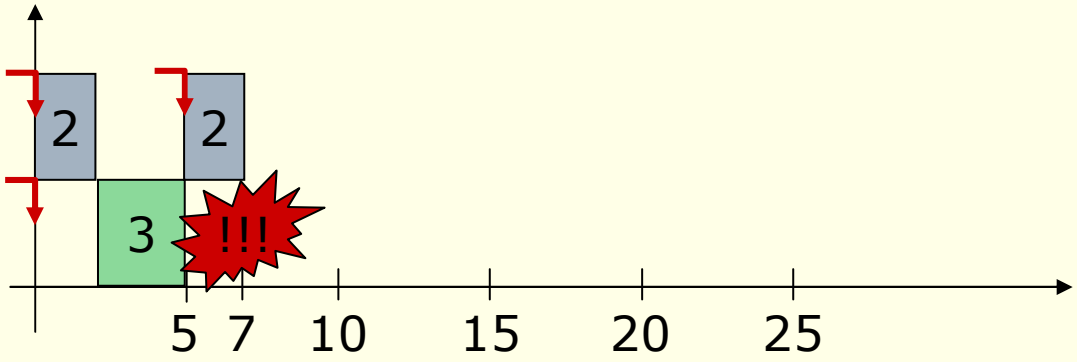
Tâche	P_t	C_t
A	5	2
B	7	4

Ordonnançable avec EDF, pas avec RMS !

EDF



RMS



Comparaison RMS - EDF

- RMS/DMS sont des algorithmes optimaux avec priorités statiques
- EDF/LLF sont des algorithmes optimaux avec priorités dynamiques
- RMS : des nombreux résultats existent et sont *largement utilisés en pratique*.
- EDF offre des utilisations de processeur supérieures, donc une meilleure ordonnançabilité, mais il est *plus difficile à implémenter* et *instable en cas de surcharge*

Exercice

Tâche	P_t	C_t
A	75	9
B	35	20
C	20	5

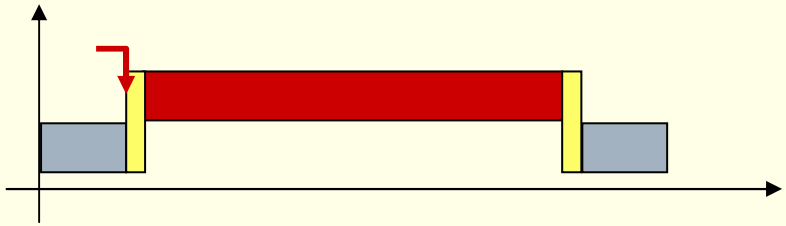
- est-ce ordonnançable par RMS ?
- quel temps de réponse pour chaque tâche ?
- si A avait besoin de plus de temps CPU, jusque où peut-elle s'étendre en FPPS ?
- jusque où A peut-elle s'étendre en EDF ?

Retour au modèle basique

- on suppose un temps de changement de contexte zéro
- Toutes les tâches sont périodiques
- les tâches sont indépendantes (pas de communication/synchronisation)

Compter le temps de changement de contexte

le pire cas:



Si S est le temps de changement de contexte, il faut compter l'utilisation du CPU par T ainsi:

$$U_T = \frac{c_T}{p_T} + \frac{2S}{p_T}$$

Retour au modèle basique

- on suppose un temps de changement de contexte zéro
- Toutes les tâches sont périodiques
- les tâches sont indépendantes (pas de communication/synchronisation)

Tâches sporadiques

- respectent un temps minimum entre arrivées P
- ont en général une échéance D << P

⇒ on peut les intégrer par exemple en utilisant DMS

Remarques:

- l'analyse du temps de réponse marche pour D < P
- l'analyse du temps de réponse marche pour n'importe quel ordre de priorités fixes

⇒ on peut vérifier off-line la faisabilité d'autres ordres

Tâches apériodiques

- pas de temps minimum entre arrivées

Solution 1: priorité minimale

- on ordonne les tâches *périodiques* selon DMS
- on affecte aux tâches *apériodiques* une priorité plus petite que celle de toutes les tâches périodiques

⇒ schéma sûr pour les tâches périodiques – n'affecte pas leur ordonnançabilité

Mais : ne garantit rien pour les échéances des tâches apériodiques

Tâches apériodiques

Solution 2: tâches serveur

- on alloue une tâche *serveur* avec:
  - une période P_s
  - une capacité de calcul C_s
  - une priorité π - en général la plus haute
- P_s et C_s sont choisies de façon à ce que les tâches périodiques restent ordonnançables
  ($\Rightarrow$ analyse de temps de réponse)
- à l'arrivée d'une tâche asynchrone:
  - s'exécute immédiatement ou selon π, *si la capacité restante est suffisante*
  - le temps d'exécution est *déduit de la capacité*
    (ne doit pas dépasser la capacité)
  - la capacité *se régénère* à chaque période P_s

Serveurs : discussion

- le CPU est *disponible immédiatement* pour les tâches asynchrones – si capacité suffisante
- en absence de tâches asynchrones
  - le CPU est dédié aux tâches périodiques
  - la capacité reste disponible le plus possible
- les serveurs protègent aussi contre l'arrivée trop fréquente des tâches sporadiques

Retour au modèle basique

- on suppose un temps de changement de contexte zéro
- Toutes les tâches sont périodiques
- les tâches sont indépendantes (pas de communication/synchronisation)

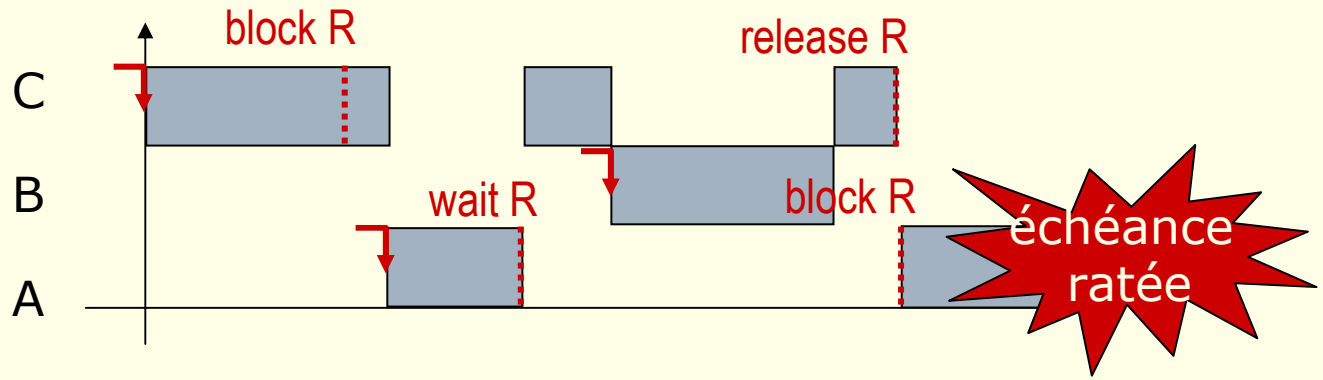
Interactions et blocage

- on considère les interactions par *ressources protégées* (e.g., moniteurs, protection par sémaphores, etc.)
  - ⇒ une tâche peut être suspendue en attente d'une action par une autre tâche

- **<u>anomalie</u>** : si A > B et A est bloqué par une ressource détenue par B. B s'exécute pendant que A attend. (inversion de priorité)

Exemple

A > B > C, ressource partagée R

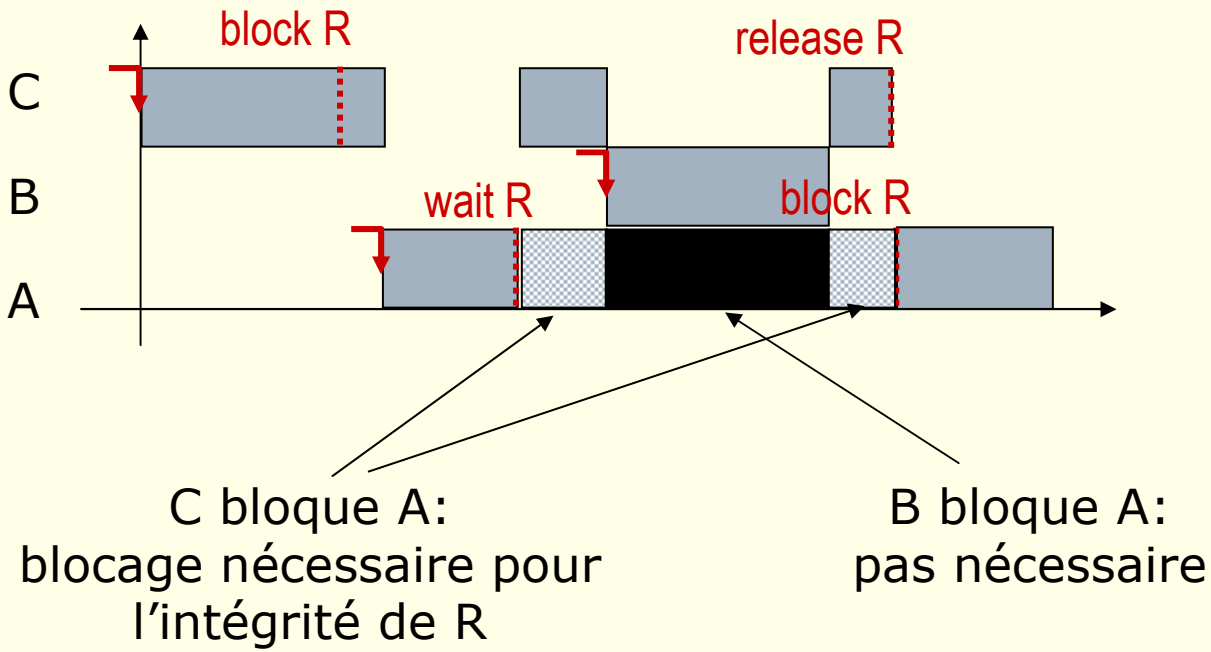


Un exemple réel

La mission Mars Pathfinder, 4 Juillet 1997

- système ordonnancé par RMS, sur VxWorks
- inversion selon le modèle précédent entre:
  - T1 tâche du bus
  - T2 tâche de communication sol
  - T3 tâche de recueil d'information météorologiques
  - R = le bus (mutex)
- corrigé à distance en activant PCP

Discussion

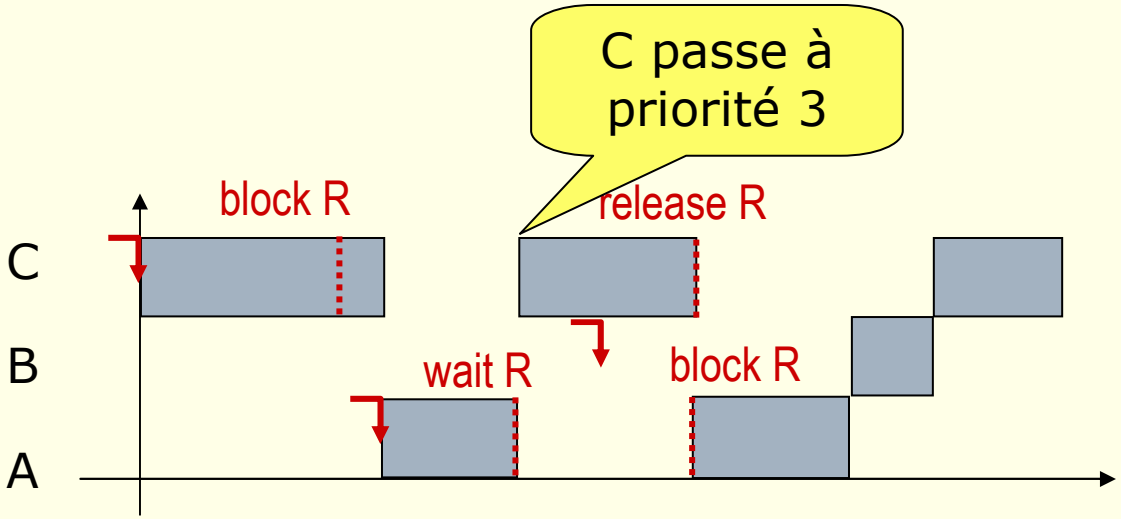


⇒ nécessité de politiques pour minimiser le temps de blocage et *le rendre déterministe*

Priority inheritance protocol

Quand une tâche T_H est bloquée en attente d'une ressource utilisée par une tâche T_L , alors T_L hérite temporairement la priorité de T_H.

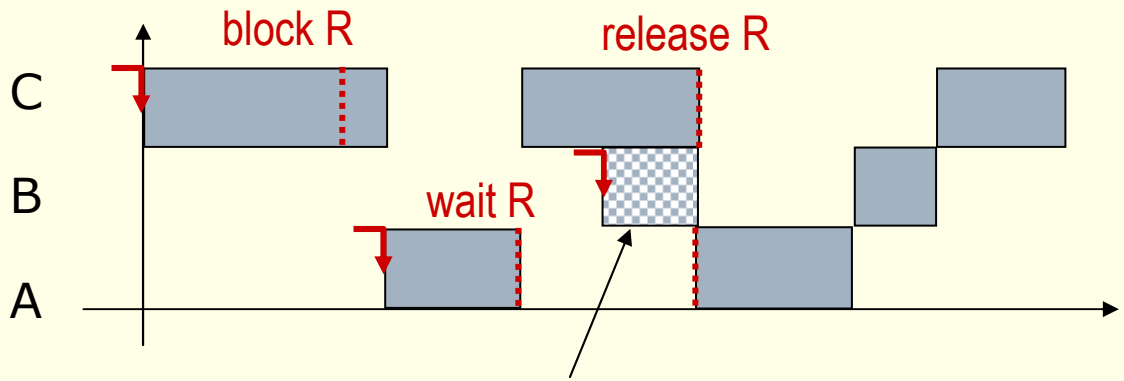
Quand T_L libère la ressource, elle reprend sa priorité normale.



L'héritage doit être transitif: si C devient bloqué par D pendant qu'il bloque A, D hérite la priorité de A !

Calcul du temps de blocage

- avec PIP, une tâche peut subir un blocage alors qu'elle n'utilise pas de ressource partagée :



B est bloqué alors qu'un processus de moindre priorité s'exécute (C)

- Soit :

$$\text{util}(k,i) = \begin{cases} 1, & \text{si la ressource k peut être utilisée par une tâche } <i \text{ } \textbf{et} \text{ par une tâche} \geq i \\ 0, & \text{sinon} \end{cases}$$

Calcul du temps de blocage

On suppose que pour toute ressource *k*, le temps de calcul entre acquisition et libération est C(k) (quelque soit la tâche qui)

- e.g., *k* est une opération de moniteur et C(k) est son WCET

Le temps de blocage maximum pour une tâche de priorité *i* est:

$$B_i = \sum_{k \in K} util(k,i) C(k)$$

Critère basé sur l'utilisation

On peut adapter le critère *suffisant* basé sur l'utilisation du CPU pour prendre en compte le temps de blocage B_i, en augmentant l'occupation du processeur de B_i/p_i

Théorème: Soit $T_1 > T_2 > \ldots > T_n$ un ensemble de tâches avec priorités affectées en RMS. L'ensemble est ordonnançable **si**

$$\forall i \in \{1,\ldots,n\}. \frac{c_1}{p_1} + \frac{c_2}{p_2} + \ldots + \frac{c_{i-1}}{p_{i-1}} + \frac{c_i + B_i}{p_i} \leq i(2^{1/i} - 1)$$

Analyse du temps de réponse

$$R_i = C_i + B_i + I_i$$

$$R_i = C_i + B_i + \sum_{j>i} \left\lceil \frac{R_i}{P_j} \right\rceil C_j$$

$$w_i^0 = C_i + B_i \qquad w_i^1 = C_i + B_i + \sum_{j>i} \left\lceil \frac{w_i^0}{P_j} \right\rceil C_j \qquad \cdots$$

Exercice

Tâche	P_t	C_t
A	75	9
B	35	20
C	20	5

→ peut utiliser k, C(k)=5

→ peut aussi utiliser k

- calculer le temps de réponse au pire cas de chaque tâche
- le système est-il ordonnançable?

Priority ceiling protocols

- caractéristiques de PIP:
  - une tâche peut être bloquée plusieurs fois
  - possibilité de blocage transitif
  - estimation pessimiste du temps de blocage

⇒ protocoles améliorés (priority ceiling) assurant que:
  - une tâche peut être bloquée une seule fois
  - le blocage transitif est évité
  - les deadlocks sont évités et l'exclusion mutuelle est assurée par le protocole même

Priority ceiling classique

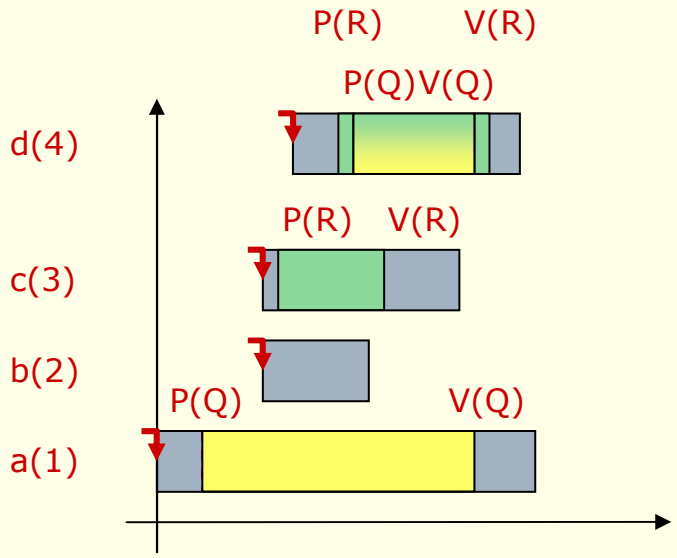
- basé sur le protocole d'héritage de classique (PIP) : une tâche hérite la priorité d'une tâche plus haute qu'elle bloque

- PCP rajoute une condition pour l'acquisition des ressources:
  - chaque ressource a un plafond (ceiling) = la priorité de la plus haute tâche qui peut l'utiliser (calcul statique!)
  - une tâche peut acquérir une ressource R ssi:
    - R n'est pas déjà détenue par une autre tâche
    - la priorité dynamique de la tâche est supérieure au plafond des toutes les ressources déjà bloquées par **d'autres** tâches

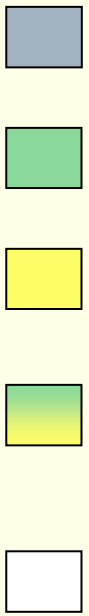
Un exemple

l'ensemble de tâches (sans ordonnancement)

- P(R) = acquisition de R, V(R) = libération de R



legende:



exécution sans ressource

exécution avec R acquis

exécution avec Q acquis

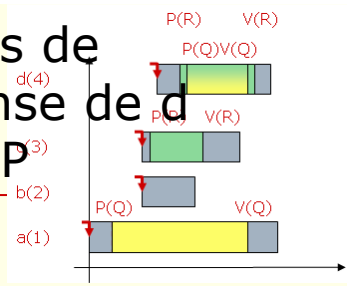
exécution avec R et Q acquis

tâche suspendue (préemptée)

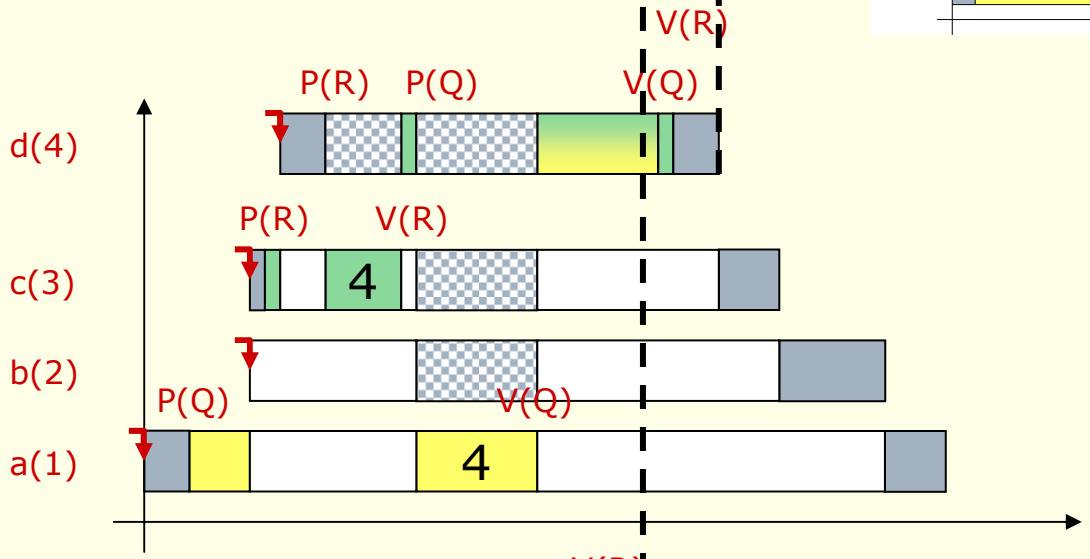
Un exemple

temps de réponse de d en PCP

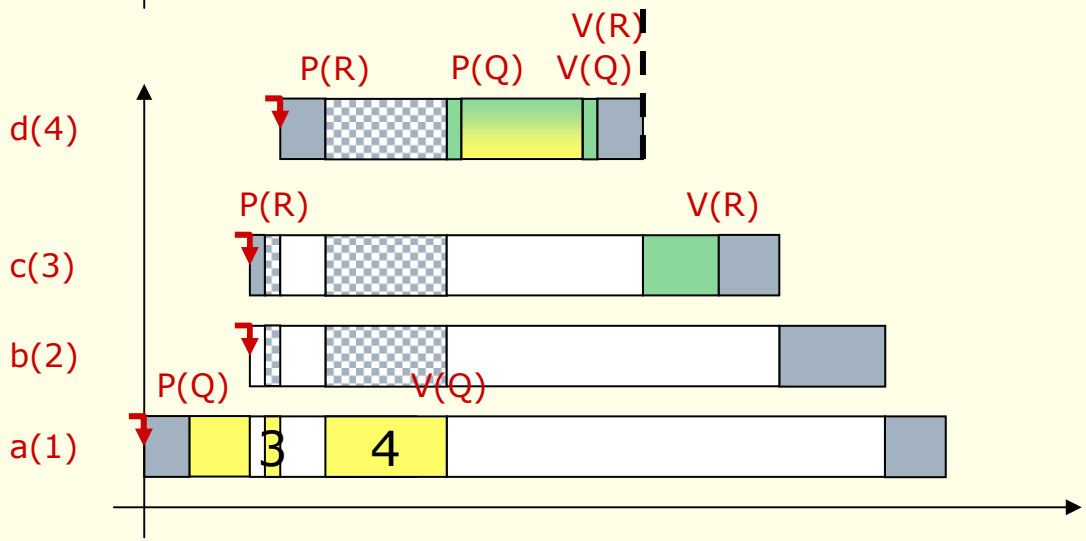
temps de réponse de d en PIP



exécution avec PIP :



exécution avec PCP :



Temps de blocage

- un seul blocage par exécution d'une tâche

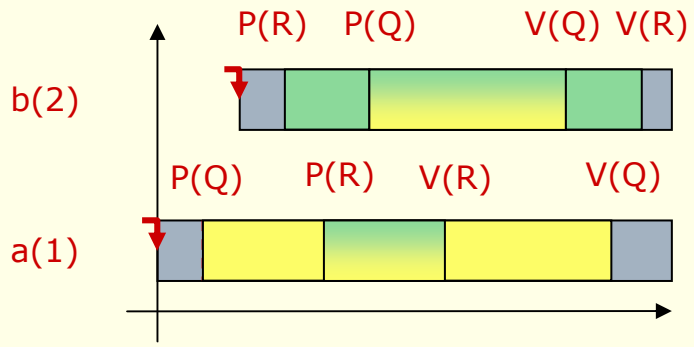
$$B_i = \underset{k \in K}{MAX}\ util(k,i)C(k)$$

au lieu de

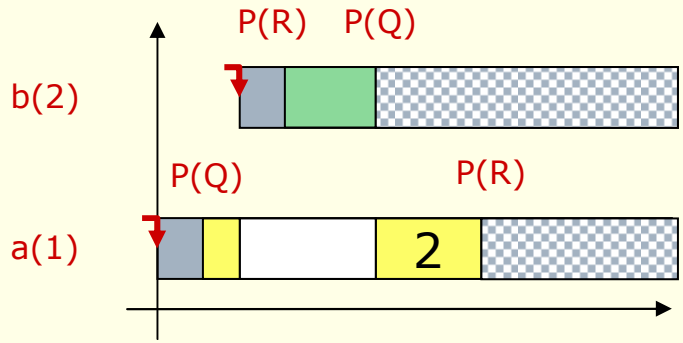
$$B_i = \sum_{k \in K} util(k,i)C(k)$$

Éviter les deadlocks

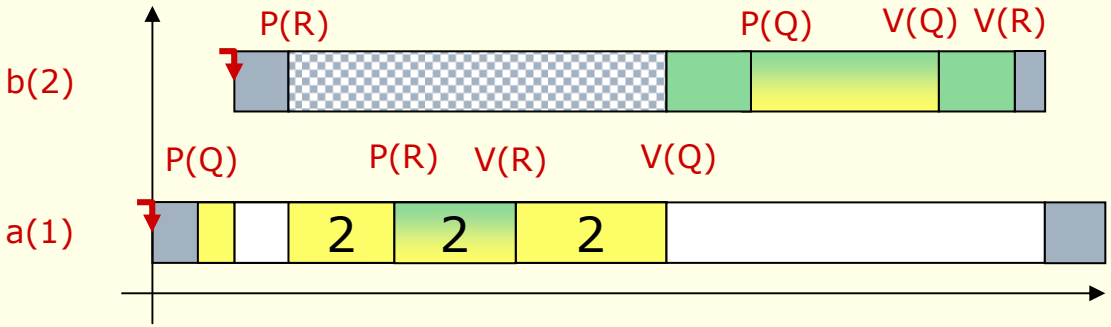
tâches (sans ordonnancement) :



en PIP :
(ou sans protocole)



en PCP :

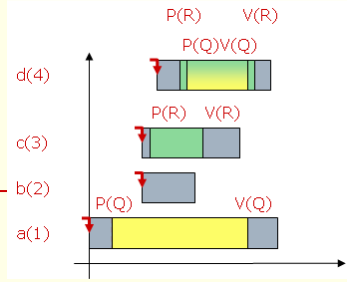


Une variante: PCP immédiat

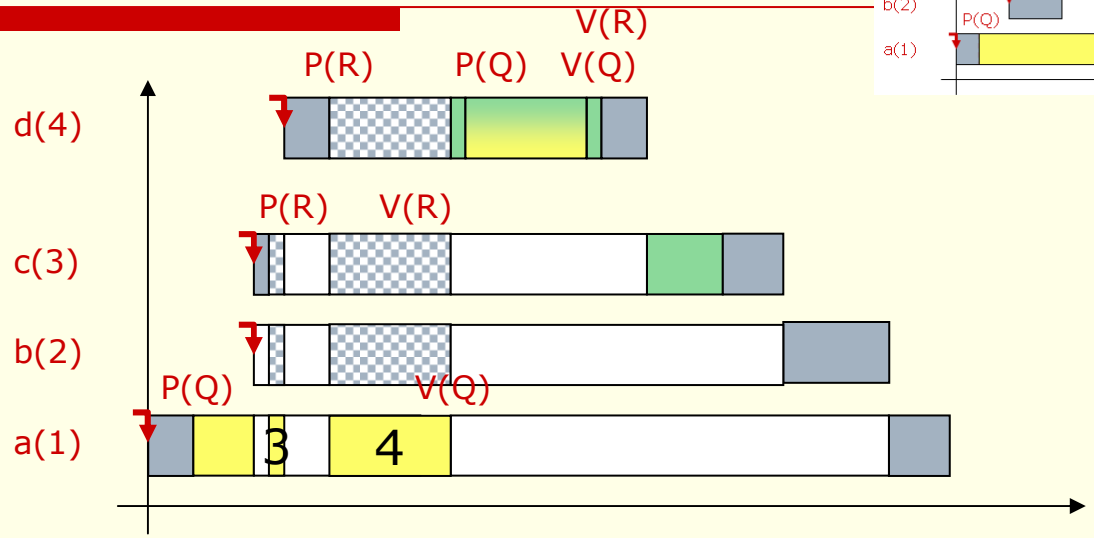
Définition : chaque fois qu'une tâche acquière une ressource, la priorité de la tâche est levée au niveau plafond de la ressource.

- l'acquisition de R se fait comme d'habitude, sans condition supplémentaire sur la priorité de la tâche

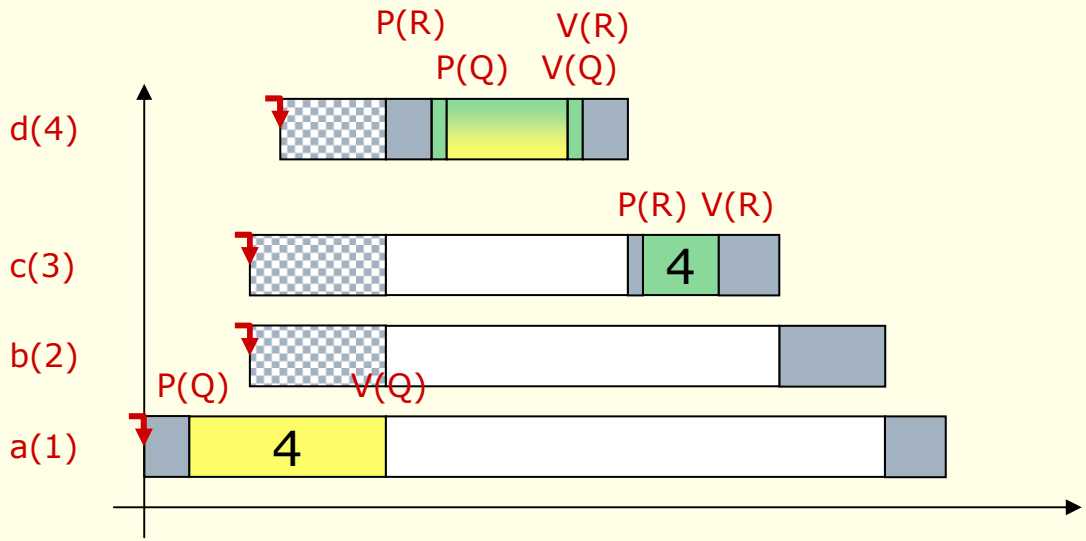
Exemple



exécution avec PCP :



exécution avec PCP immédiat :



PCP classique vs. PCP immédiat

- même temps de blocage au pire cas: $B_i = \underset{k \in K}{MAX}\, util(k,i)C(k)$
- PCP immédiat plus facile à implémenter
  (pas d'héritage de priorité, pas de test de priorité au moment de l'acquisition)
- le blocage survient toujours 1 fois, au début de l'exécution de la tâche
  $\Rightarrow$ moins de changements de contexte
- plus de changements de priorité dans PCP immédiat que dans PCP classique
  (à chaque acquisition de ressource, pas uniquement en cas de blocage)
- PCP immédiat implémenté en POSIX, RT Java, …

Ordonnancement : résumé

- une politique d'ordonnancement est caractérisée par
  - un algorithme pour allouer la ressource partagée (p.e. CPU)
  - une technique pour prédire le comportement au pire cas de la politique

- ordonnancement cyclique
  - le système est programmé comme un ensemble de procédures
  - les appels sont groupés dans des cycles mineurs (déclanchés par le temps). Au bout de plusieurs cycles mineurs, le schéma se répète (cycle majeur)
  - beaucoup d'inconvénients, résolus par des politiques préemptives à base de priorités

Ordonnancement : resume

- politiques à base de priorités fixes
  - affectation optimale des priorités : RMS, DMS
  - critère suffisant d'ordonnançabilité simple
  - analyse de temps de réponse (critère nécessaire et suffisant) flexible, peut prendre en calcul
    - processus sporadiques
    - temps de blocage causé par la synchronisation
    - temps de changement de contexte
    - etc.
  - implémentées en POSIX, Ada, RT Java, …

- politiques à base de priorités dynamiques
  - politiques optimales en mono-processeur: EDF, LLF
  - meilleure occupation du processeur
  - critère nécessaire et suffisant simple quand $D \leq P$
  - implémentation plus complexe, instabilité en cas de surcharge,…

Aller plus loin

- en priorités fixes, l'analyse de temps de réponse peut être étendue pour couvrir:
  - imprécision du moment d'arrivée des tâches (*jitter*)
  - échéances arbitraires, en particulier D > P
  - déphasage des tâches périodiques
  - …
- ordonnancement avec plusieurs niveaux de criticité
- ordonnancement en milieu multi-processur
  - EDF, RMS ne sont plus optimaux
  - anomalie du multiprocesseur : une tâche qui finit *plus vite que prévu* peut faire déborder une autre !!

STR : conclusions

- systèmes dont la correction est définie par les résultats et aussi par les moments où ces résultats doivent être disponibles (échéances)

- haute criticité ⇒ besoins stricts de **correction**, **sûreté** de fonctionnement, **prédictibilité**

- techniques employées
  - programmation concurrente, synchronisation (causes : environnement parallèle, système réparti)
  - programmation des aspects temporels (attente, interruption par temporisation,…)
  - gestion des échéances par une politique d'ordonnancement temps réel, prédiction du comportement au pire cas

STR : aller plus loin

Techniques employées (pas couvertes dans le cours)

- architectures matérielles prédictibles;
  estimation du pire temps d'exécution (WCET)
- méthodes et langages d'analyse
  - formalisation des besoins et des propriétés du système
- méthodes et langages de conception
  - langages dédiés: langages à base de flots de données, langages à base d'automates ou commandes gardées, langages synchrones / asynchrones,…
- techniques de validation et certification
  - vérification formelle
  - sélection de cas de test
  - génération automatique de tests
  - …