Chapitre 3

Entrées Sorties

1. Entrées Sorties (ES) physiques
2. Transfert de données et de commandes
3. ES programmée
4. DMA et Channel
5. ES virtuelle
6. Correspondance ES Logique - Physique
7. Problèmes : recouvrement et technique de tampon

Entrées Sorties (ES)

⊡ Notion de base : *échange de signaux.*

→ e.g. imprimante : *prête, envoi données, fin, erreur out of paper, spool plein/vide, bourrage...*

⊡ Le SE gère les communications entre les périphériques (malgré les standard différents).

⊡ Les système d'ES : interface entre l'utilisateur et les périphériques (par commandes de haut niveau telles que *read, write, ...*).

→ Ces commandes provoquent un SVC (*superviser call*).

3.1 Entrées Sorties physiques (ES Φ)

$\boxdot$ Deux opérations de base :

sélection du périphérique, **transfert** des données

$\Rightarrow$ la sélection au niveau matériel ;

$\Rightarrow$ le transfert des données via le matériel et modules système ;

$\Rightarrow$ Plusieurs méthodes de transfert.

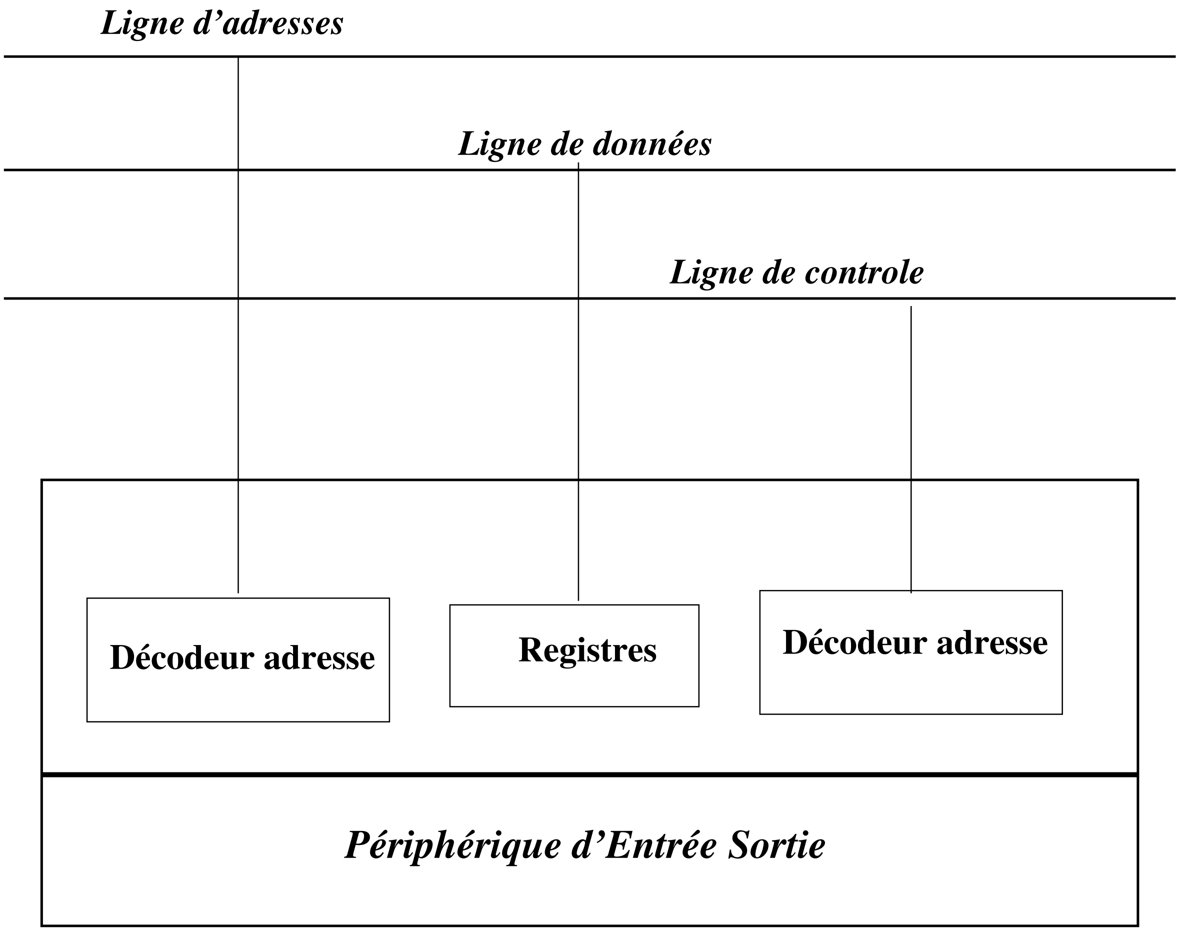


fig III.1 : périphérique et bus

3.2 Méthodes de transfert

⊡ ES programmées, DMA, Canal de commande, ...

3.2.1 ES programmées

⊡ Chaque périphérique : registres, zone mémoire.

⊡ Le SE transfère les données vers ces emplacements + exécution d'une instruction d'ES effective.

⇒ Inconvénients :

- taille et nombre fixes (vol. données à trasférer,...)
- Accès concurrent

3.2.2 Accès direct à la mémoire (DMA)

⊡ Le SE fournit l'adresse de la zone mémoire (data) + la taille des données

⊡ Des circuits de contrôle spéciaux font le transfert **sans l'aide de l'UC**

⊡ Matériel adéquat de DMA ⇒ régistres spécifiques (adresse mémoire, taille, commande = type d'ES), ...

NB : même chose pour le disque (avec des registres supplémentaires).

⊡ Problème du DMA : accès simultané à un mot de mémoire (DMA et UC).

⇒ Solution :

- Exclusion mutuelle d'accès (verrou, etc.)
- Phénomène *cycle stealing* : le DMA "vole" des cycles à l'UC.

3.2.3 Canal de commandes (command channel ≃ DMA)

⊡ Les circuits du DMA remplacés par un processeur (contrôleur) autonome capable d'exécuter des programmes (= programmes du Canal).

⇒ Le SE fournit l'adresse du code à exécuter par le Canal (sans l'aide de l'UC).

⇒ le DMA = un Canal de commande qui exécute un programme d'une seule instruction de transfert.

3.3 périphériques virtuels d'Entrées Sorties

Constat :

⊡ Il y a différents types de périphériques : *clavier, console, lecteur, disque, souris, scanner, etc* ;

⊡ Chaque type existe en plusieurs modèles.

⇒ **Objectifs** :

⊡ Rendre l'utilisateur indépendant de ces particularités :

⇝ e.g. l'impression ne doit pas dépendre de la marque de l'imprimante.

⇝ Si l'imprimante doit être changée, on ne doit pas changer nos programmes ! !

⊡ Obtenir une indépendance au niveau élevé :

$\$cat\ f > fic$ ou $\$cat\ f > stdout$

Avec éventuellement peu de modification.

$\Longrightarrow$ abstraction du flot d'E/S.

$\Rightarrow$ Conséquence :

Ne pas s'adresser directement aux périphériques mais aux flots (périphériques d'ES **virtuels**).

$\Rightarrow$ C'est à dire :

- Correspondance virtuel-Φ assurée par le SE
- Gestion d'un *descripteur* de flots propres à chaque processus (PCB) , pour un processus :

Entrée 3	Disque 2	Actif
Sortie 5	Imp 2	Actif
Entrée 1	Disque 2	Inactif
...	...	...

- Dialogue SE $\longleftrightarrow$ périphérique Φ :

  ⊡ pilote (*device driver*) de périphériques

  ⊡ structure unique avec une table des particulrités de chaque périphérique

$\Rightarrow$ L'utilisateur utilise la même commande pour écrire sur des périphériques différents.

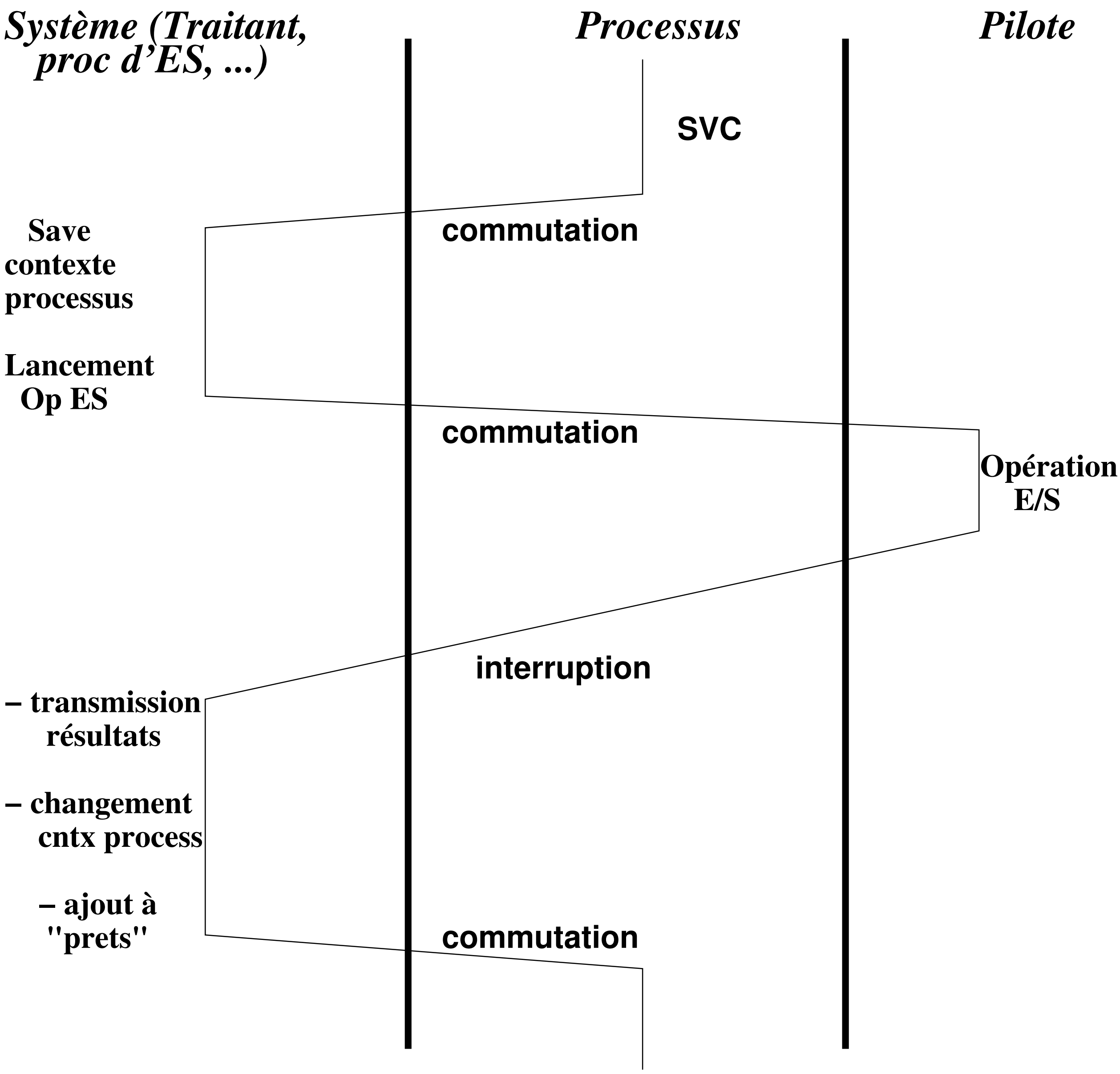
3.3.1 Correspondance ES Logique (virtuelle) $\leftrightarrow$ ES Φ

$\boxdot$ read $\mapsto$ SVC $\mapsto$ Interruption (ES) $\mapsto$ Commutation $\mapsto$ Traitant

$\boxdot$ Le traitant d'interruption (Handler) :

- Reconnaît la cause de l'Interruption,
- Provoque une commutation vers la routine d'ES
- Cette routine cherche les données (registres) de l'opération (flot, type Op, Qté info, src / dest, etc.)
- Correspondance flot $\leftrightarrow$ périphérique Φ
- Attente si périphérique occupé (ajout PCB à la file d'attente de la ressource)
- Commutation vers le device driver
- Opération ES par DMA ou Canal
- Fin de l'ES $\mapsto$ Interruption (fin ES) $\mapsto$ Handler $\mapsto$ transmission résultats $\mapsto$ ajout PCB demandeur à "prêts"

$\Rightarrow$ Voir figure.



⇒ Ce shéma ne traite pas le recouvrement.

3.3.2 Problème liés aux ES

3 problèmes :

signaux simultanés, recouvrement , protection.

1- Traiement des signaux et Interruption simultanées :

- Une demande d'ES $\mapsto$ le SE s'assure que le périphérique est prêt.

- Contrôle via les registres d'état du périphérique (via bus de contrôle).

- Si pas prêt, le SE reçevra la réponse (un signal) du périphérique sur une ligne de contrôle (*LineIntr*).

- Plusieurs signaux dans le système (cf. priorité, masquage, désarmement).

- Les réponses (Interruptions) non traitées sont répétées par le périphérique jusqu'un signal d'acquittement (ligne *INTA*).

N.B. : Si *Canal* disponible, on peut soustraiter.

2- Différence de vitesses de traitement :

⊡ Les périphériques sont beaucoup plus lents que l'UC

⇒ UC lit 120 cartes/minute et les compile en 2 sec

⇒ L'UC est toujours en avance

⇒ Recouvrement phases calcul et ES par un tampon (à l'aide de DMA/Channel)

2.1 Technique de Tampon

⊡ Utilisée pour les accès disque / mémoire

⊡ But :

- Ne pas faire attendre l'UC
- Ne pas faire attendre les processus

⇒ Se servir d'un tampon pour :

- Passer des données à l'UC,
- Permettre à l'UC de lancer une ES à la fin d'une autre ES (cf. vos imprimantes)

⊡ Si l'UC a besoin de lire (ou de transmettre) des valeurs, il peut le faire via le tampon

⊡ Un tampon = une zone disque (ou une zone mémoire)

⊡ Tout processus dirige ses sorties (idem entrées) vers les tampons

⊡ Les contrôleurs (de disque/mémoire) doivent assurer ces transferts (sans l'UC).

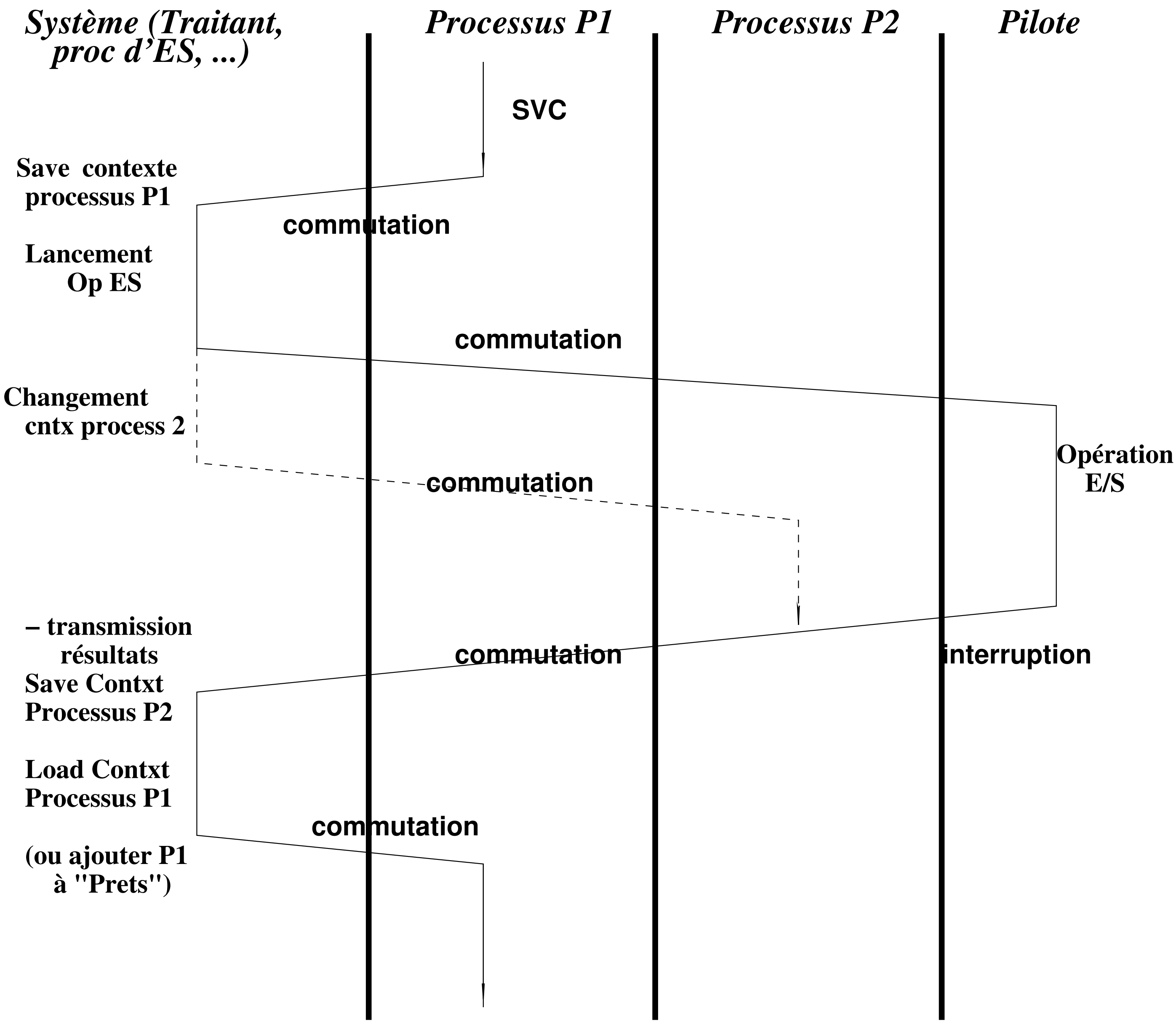
⊡ Cela évite de bloquer un processus dans les files d'attente.

⊡ Exemple : imprimer sans tampon = attendre la fin de l'impression.

$\Rightarrow$ le processus peut envoyer ses données à un tampon (spooler) et continuer son exécution.

∇ : inefficace si le processus fait beaucoup d'ES et peu de calcul (le tempon sera souvent vide/plein).

⊡ Schéma de recouvrement calcul - ES :


Système (Traitant, proc d'ES, ...)
Processus P1
Processus P2
Pilote
SVC
Save contexte processus P1
commutation
Lancement Op ES
commutation
Changement cntx process 2
commutation
Opération E/S
– transmission résultats
Save Contxt Processus P2
commutation
interruption
Load Contxt Processus P1
commutation
(ou ajouter P1 à "Prets")

3- Protection des opérations d'ES :

☒ Un programme ne doit pas influer sur les autres ;

☒ Les ES sont très critiques : on peut effacer les données des autres.

☒ On doit passer par SVC mais les manipulations directes sont possibles

☒ Règles pour éviter les problèmes :

- Seul le SE réalise les ES par des instructions privilégiées ;
- On prévoit aussi 2 modes : système (**SV**) et **user**.
- En mode système, on a accès à tout (zones, instructions, ...)
- En mode user, on a accès aux instructions (et zones) "publiques "⇒ pas de privilège.
- Le SE s'exécute en mode système et nos programmes en mode user
- Les instructions effectives d'ES de bas niveau sont des instructions privilègiées

☒ Pour la mise en place :

- dans chaque instruction : un bit (type instruction) et un bit (non accessible à User) pour le mode ;
- le matériel teste la conformité des 2 bits
- si problème : Interruption *violation instruction privilégiée* (déroutement)

Chapitre 4

Processus

- Elément de base du SE (cf. cours précédents)

- Modèles de présentation de processus

4.1 Modèles de présentation de processus

☒ Modèle simple : un processus après l'autre.

☒ Notion de *tâche* pour la modélisation des activités

☒ Une tâche : unité élémentaire de traitement

☒ On découpe un processus en tâhes :

$$P = T_1 T_2 T_3 ... T_n$$

Début et fin de Ti : *di* *et* *fi*

⇒ di : initialisation, lecture des paramètres en entrée, acquisition ressources, etc.

⇒ fi : écriture des résultats, libération des ressources, save des infos, etc.

☒ Une tâche Ti réalise une fonction Fi

☒ La suite $<d1f1d2f2...dnfn>$ est associée à $P = T1T2..Tn$

☒ **Un mot de tâche** :

toute suite construite sur $A = \{d1, f1, d2, f2, ..., dn, fn\}$

☒ Un processus peut être divisé en différentes séquences de tâhes.

⇒ on suppose qu'il n'y a pas d'ES entre di et fi

☒ Exemple :

$T : N := N + 1; \quad F(n) = n + 1; \quad T = T1\ T2\ T3$

$T = df = d1f1d2f2d3f3$ (dans le cas séquentiel simple)

$T1 : Load\ R, N$

$T2 : Add\ R, \#1$

$T3 : Store\ R, N$

4.1.1 Système de tâches et graphe de précédence

☒ Rappel : une relation de précédence sur un ensemble E notée '<' est une relation :

1. ∀ T ∈ E, on n'a pas T < T
2. $\forall T1, T2 \in E$, on n'a pas simultanément $T1 < T2$ et $T2 < T1$
3. la relation < est transitive : si $T1 < T2, T2 < T3 \Rightarrow T1 < T3$

☒ Le couple S=(E, <) est un **système de tâches**.

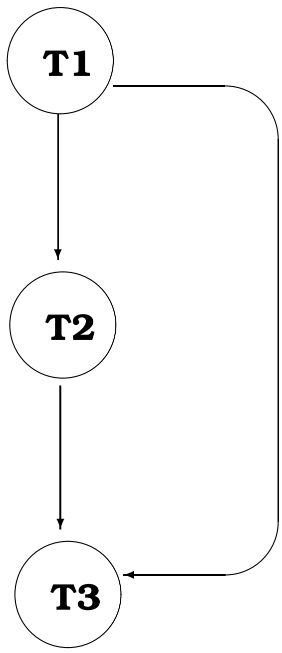
☒ Interprétation intuitive : $Ti < Tj$ ssi la terminaison de Ti a lieu avant l'initialisation de Tj.

☒ Si on n'a ni $Ti < Tj$ ni $Tj < Ti$, l'ordre d'exécution de Ti et de Tj n'a pas d'importance (peuvent s'exécuter en parallèle).

☒ On représente un processus séquentiel par un système de tâches dans lequel deux éléments quelconques sont toujoures en relation.

☒ **Exemple** (graphe non compacte) :

Il existe un arc entre Ti et Tj ssi $Ti < Tj$.



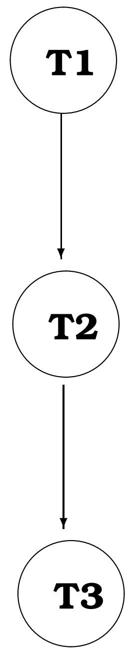
☒ Le graphe *compact* (ou le graphe de **précédences**) :

le graphe de la plus petite relation qui a la même fermeture transitive.

i.e. la relation '$\rightarrow$' telle que :

$Ti \rightarrow Tj$ SSI $Ti < Tj$ et $\forall$ Tk, on n'a pas $Ti < Tk < Tj$.

$\Rightarrow$ Pour l'exemple précédent, le graphe compacte est :



$\boxtimes$ Tj est *successeur* de Ti ssi $Ti \rightarrow Tj$.

$\boxtimes$ Une tâche sans *prédécesseur (successeur* d'aucune tâche) est dite *initiale.*

$\boxtimes$ Une tâche sans successeur est dite *terminale.*

$\boxtimes$ Ti et Tj sans aucune relation (exécutables en parallèle) sont dites *indépendantes.*

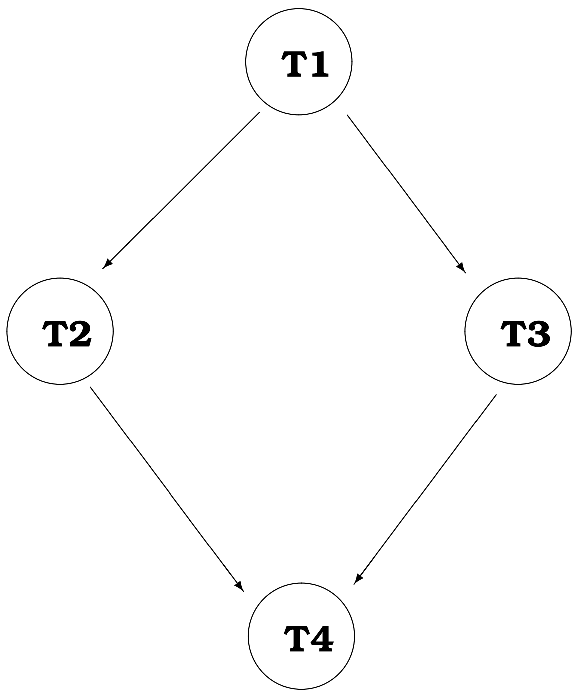
4.1.2 Langage des tâches

Rappel : mots de tâches construits sur l'alphabet $A = \{d1, f1, d2, f2, ..., dn, fn\}$

☒ Pour $T = T1T2T3$, $A = \{d1, f1, d2, f2, d3, f3\}$, on peut construire **6 !** mots.

☒ Certains mots seront éliminés selon les contraintes de précédences.

☒ Exemple :



☒ Dans cet exemple, les mots suivants sont acceptables :

$\boxed{d1f1d2d3f3f2d4f4 \quad | \quad d1f1d3d2f2f3d4f4 \quad | \quad d1f1d3f3d2f2d4f4}$

$\Rightarrow$ Ces mots satisfont la relation de précédence exprimée par le graphe.

☒ Définition des mots du langage L(S) :

- Soit $S = (E,<)$ et $A = \{d1, f1, d2, f2, ..., dn, fn\}$,

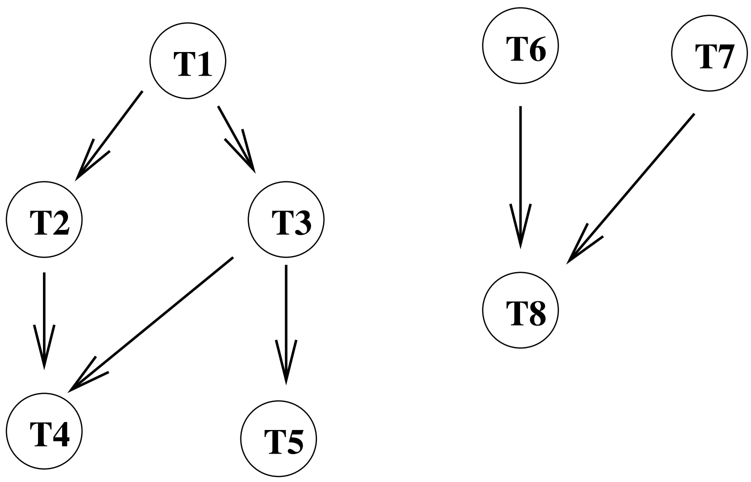
on associe au système de tâches S un ensemble fini de mots ω définis sur l'alphabet A par :

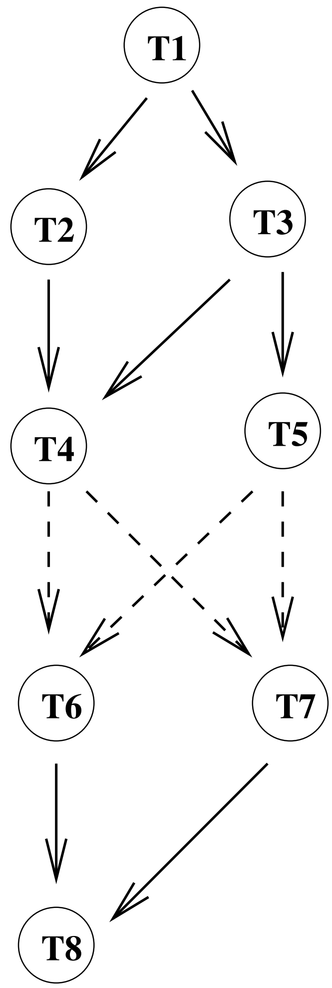
– $\forall\ Ti \in E$, ω contient exactement une occurrence de di et une de fi ;

– $\forall\ Ti \in E$, di est avant fi (di figure à gauche de fi)

$\Rightarrow$ Chaque mot décrit un **comportement** possible de S

☒ Le produit de deux systèmes de tâches $S1$ et $S2$ (leur exécution $squentielle$ notée $S1.S2$) s'obtient en joignant chaque sommet terminal de $S1$ à tout sommet initial de $S2$.





Composition séquentielle de S1 et S2

☒ La composition parallèle ($S1||S2$) reprend simplement les deux graphes séquentiels.

☒ Le langage $L = L(S1).L(S2) = L1.L2$ (concaténation des langages des systèmes $S1$ et $S2$) est tel que :

$$\boxed{L = \{\omega = \omega 1 \omega 2, \omega 1 \in L1, \omega 2 \in L2\}}$$

☒ Le langage $L = L(S1)||L(S2) = L1||L2$ représente l'exécution parallèle des deux systèmes $S1$ et $S2$. Il s'obtient en énumérant toutes les combinaisons possibles d'exécution parallèle de chaque tâche de S1 avec celles de S2.

4.1.3 Spécification dans les langages évolués

☒ Séquence : séparé par ' ;' dans un bloc $begin - end$:

$$\boxed{begin \text{ T1 ; T2 ; } end}$$

☒ Parallèle : séparé par ' ;' dans un bloc $parbegin - parend$:

$$\boxed{parbegin \text{ T1 ; T2 ; } parend}$$

☒ En notation textuelle, on écrira également $(T1\ T2)$ ou $(T1; T2)$ (pour séquentiel) et $(T1||T2)$ ou $(T1//T2)$ pour parallèle.

⇒ Exemple :

begin

    <u>parbegin</u> **Lire(a) ; Lire(b) ;** <u>parend</u>

    <u>parbegin</u> **C := a*a ; d := b*b ;** <u>parend</u>

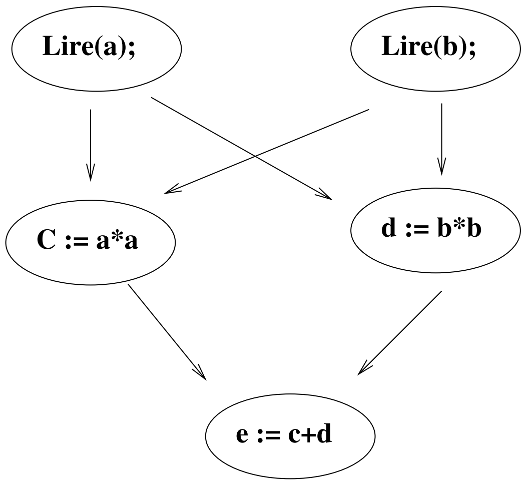
    e := c+d ;

end

⇒ Une notation textuelle :

((Lire(a) | | Lire(b)) (C := a*a | | d := b*b) e := c+d)

$\Rightarrow$ qui correspond au graphe :



$\boxtimes$ Exemple 2 :

T1 : $N := 0;$ **T2 :** $N := N + 1;$ **T3 :** $N := N + 1;$ **T4 :** $affiche(N);$

$\Rrightarrow$ Hypothèse d'exécution du système S : $T1(T2||T3)T4$

```
begin

    T1 ;

    parbegin T2 ; T3 parend

    T4

end
```

$\Rightarrow$ Problème : pas toujours la même valeur de N affichée.

$\Rightarrow$ Exemple de comportement problématique de S :

$$\boxed{\omega = d1\ f1\ d2\ d3\ f2\ f3\ d4\ f4}$$

On montre ce problème en détaillant le code.

Rappel : les détails du code (par exemple) de T2 : N :=N+1 :

d2 : Load R, N		% charger la variable N dans R
	Add R, #1	% ajouter 1 à R
f2 : Store R, N		% ranger R dans la case mémoire N

Dans :

T1 : $N := 0;$ **T2 :** $N := N + 1;$ **T3 :** $N := N + 1;$ **T4 :** $affiche(N);$

$\Rightarrow$ Pour la partie $(T2||T3)$ du système S, on peut obtenir le comportement $...d2d3f2f3...$ ($T1$ et $T4$ ne posent pas de problème) :

d2 : Load R, N		% R=0 ; **Ici, préemption (T2 commutée)**
d3 : Load R1, N		% R1=0 ; **T2 reprend**
	Add R, #1	% R=1
	Add R1, #1	% R1=1
f2 : Store R, N		% N = 1
f3 : Store R1, N		% N=1

Remarque : pour le comportement $...d2d3f2f3...$, d'autres combinaisons (d'instruction de base) sont possibles.

$\boxdot$ Une conclusion :

- Utiliser $parbegin...parend$ avec précaution.

- Utiliser les mécanismes de protection et de partage.

4.1.4 Autres formalismes et modèles

$\Rightarrow$ Automates d'états finis (AEF) et Réseaux de Petri (RdP).

$\Rightarrow$ Mettre un exemple de chaque formalismes

Chapitre 5

Interaction de processus

Résultats principaux :

☒ Les processus exécutés en parallèle peuvent interagir mutuellement :

– Ils coopèrent, partagent des données

– Sont concurrents pour acquérir des ressources

☒ La base : communication de différentes façons :

– **Dans un système centralisé** : par variables partagées

– **Dans un système réparti** (sans mémoire commune) : par messages (non instantanés)

– Les messages sont aussi utilisés dans un système centralisé

☒ Interactions mal contrôlées ⇒ Problèmes de blocage, etc.

☒ **Problème-1** : accès concurrent aux ressources.

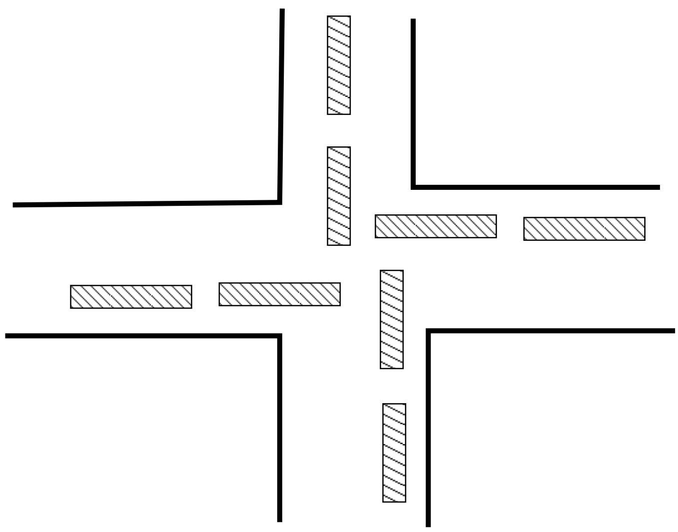
Vu dans l'exemple précédent (des tâches et accès concurrent à la mémoire).

- Selon l'ordre des accès, on a des résultats différents.

- Problème inhérent aux systèmes multiprogrammés (commutation possible à tout moment).

⇒ Quand cela est possible, partir d'un comportement séquentiel et produire un système équivalent avec certaines parties exécutées en parallèle.

☒ **Problème-2** : blocage :



☒ Chaque processus détient une ressource et est en attente d'une ressource prise par un autre.

⇒ Le SE doit prendre des mesures pour débloquer.

☒ Les méthodes :

– détecter un blocage

– éviter un blocage

– résoudre un blocage

☒ La modélisation par système de tâches ne fournit pas assez d'information pour contrôler l'évolution de celui-ci.

⇒ Accès concurrent : dans l'exemple des tâches, il faut connaître la suite des valeurs écrites dans chaque cellule de mémoire.

⇒ Interblocage : connaître la quantité de ressources demandées, allouées, etc.

On associe un **état du système** au système de tâches

☒ Soit :

- le système $ST = \{E, <\}$,

- le comportement $\omega \in L(S) = \{a_1, ..a_l\}, a_i \in \{d_i, f_i\}$

- une séquence S$_E$ d'états associée à ω :

$S_E = s_0, ..s_l$ avec s_k = l'état de a_k

- a_k est l'état du système après l'événement a_k qui fait passer le système de l'état s_{k-1} à l'état s_k = une **transition**

⇒ Les méthodes de gestion des interactions s'appuient sur les états pour contrôler le système (son état).

5.0.5 Systèmes *déterminés* et interférent

☒ Système **déterminé** : pour tout comportement, les résultats (contenu de la mémoire) sont identiques.

⇒ C'est le cas dans un processus séquentiel

⇒ La suite des valeurs écrites par un comportement est indépendante du comportement.

☒ Système **interférent** : les tâches (les instructions) sont interdépendantes.

⇒ Exemple : deux processus P1 et P2 :

processus P1 :	**processus P2 :**
T_{1_1} : $Lire(X)$	T_{2_1} : $Lire(Z)$
T_{1_2} : $X := X + Z$	T_{2_2} : $Y := X + Z$
T_{1_3} : $affiche(Y)$	

☒ C'est un système **interférent** (lecture et modification des variables partagées) :

⇒ Pb. d'exécution parallèle de $X := X + Z$ et $Y := X + Z$

⇒ Plusieurs comportements possibles avec des résultats différents (système **indéterminé**)

⇒ Le seul comportement (déterminé) pour l'exemple précédent : $\boxed{(T_{11}||T_{21})T_{12}T_{22}T_{13}}$ (seules les 2 lectures en ||)

Propriété non-interférence↔ Caractère déterminé

Vérifier si les tâches d'un système sont interférentes

☒ Pour chaque tâche T, on détermine les ensembles L_T et E_T (les variables lues et/ou par modifiées par T) ;

⇒ On dira que dans le système de tâches S, deux tâches T1 et T2 sont **non interférentes** si :

- ou bien T1 est un prédécesseur / successeur de T2 ; - ou bien $L_{T1} \cap E_{T2} = L_{T2} \cap E_{T1} = E_{T1} \cap E_{T2} = 0$

Ces conditions sont appelées les **conditions de Bernstein**

Dans l'exemple précédent :

processus P1 :	**processus P2 :**
T_{1_1} : $Lire(X)$	T_{2_1} : $Lire(Z)$
T_{1_2} : $X := X + Z$	T_{2_2} : $Y := X + Z$
T_{1_3} : $affiche(Y)$	

$\Rightarrow$ On a :

- $L_{22} = \{X, Z\}$ et $E_{12} = \{X\}$

- T_{1_2} et T_{2_2} sont **interférantes** car T_{1_2} n'est pas prédécesseur / successeur de T_{2_2} et $L_{22} \cap E_{12} \neq 0$

☒ **Un résultat important** :

Pour un système de tâches S,

1. Si S est constitué de tâches 2 à 2 **non interférentes** alors S est **déterminé** pour toute interprétation (hypothèse d'exécution) ;

2. Si S est **déterminé** pour toute interprétation et si pour chaque tâche T de E, le domaine d'écriture $E_T \neq 0$, alors les tâches de S sont deux à deux **non interférantes**.

$\Rightarrow$ Pour vérifier si S est déterminé pour une interprétation, il est inutile de calculer les suites des valeurs écrites en mémoire pour chaque comportement (long et indécidable) : il y a des cas où une tâche qui n'écrit pas en mémoire interfère avec les autres sans qu'il soit possible de le détecter.

$\Rightarrow$ Il suffit de vérifier que les tâches de S sont deux à deux **non interférantes** (algorithme simple).

Fin Problème tâches déterminées/interférantes => accès concurrentes, possibilité d'exécution || .

5.0.6 Systèmes équivalents et parallélisme maximal

☒ On peut trouver pour S **déterminé** un système S' **équivalent** (mêmes comportements et mêmes résultats en mémoire => même suites d'écritures).

☒ Un système de **parallélisme maximal** est un système déterminé (même comportement pour toute interprétation) dont le graphe de précédence G vérifie :
La suppression d'un arc (T1,T2) de G entraîne l'interférence des tâches T1 et T2 .

5.1 Blocage - à compléter

Caractéristiques :

1. Cas de ressources qui ne peuvent pas être partagées. Quand un processus a besoin d'une ressource détenue par un autre, il doit attendre la libération de la ressource.

2. Les ressources ne peuvent pas être réquisitionnées.

3. Un ensemble de processus est en attente circulaire

Résultats :

TO be continued...