Avant propos

Petit Plan

- 2 séances où les sujets principaux abordés seront :
  - Introduction aux Systèmes d'Exploitation
  - Phases d'une compilation et la compilation croisée
  - Introduction aux Noyaux (Kernel) des Systèmes d'Exploitation
  - Embarqué et la Robotique
  - Stratégies de contrôle d'un Robot : un exemple concret

- Dans ce document, les séctions signalées par (*) peuvent être lues dans un 2e temps (contraintes de temps de présentation).

Systèmes d'Exploitation (et Temps Réel)

Introduction aux
SYSTÈMES d'EXPLOITATION

Alexandre S. Saidi-Glandus

(ECL-3A)

(Alexandre.Saidi@ec-lyon.fr)

2009-2010

Introduction

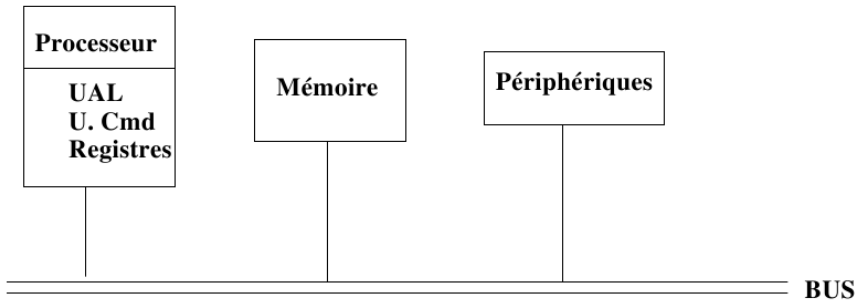
Un Système d'Exploitation (SE) : Ensemble de programmes qui réalise l'interface entre le matériel de l'ordinateur et les usagers.

• Beaucoup d'évolution ; forte orientation récente vers systèmes "User-friendly" et Graphiques, Multimédia, ...

• De nos jours, les SEs les plus courants (subjectif) :

➥ MacOS, Linux, Windows

➥ IBM, HP, SUN, ...

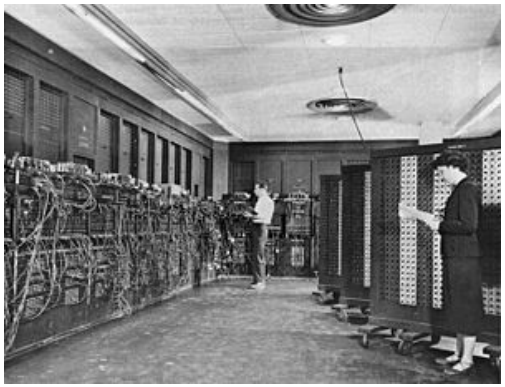


Petit Aperçu historique

- Les SE fortement liés à l'architecture des machines (physique / matériel)

  ◗ L'évolution du matériel ➙ évolution des SE

- On distingue quatre générations depuis les débuts (1946) à nos jours.
- ENIAC (Electronic Numerical Integrator and Computer, 1946) = l'ancêtre !



Aperçu historique

1. **Débuts (1946)**
   - Machines à tubes, un PC dans une salle !, peu de mémoire ;
   - Programme = séquence de bits rentrée à la main !
   - Un seul programme à la fois, Sorties sous forme de voyants lumineux
   - Langage Assembleur,
   - ...
   - Arrivent plus tard :
   - → le clavier Hexa, cartes/rubans perforées, compilateurs et langages évolués (Fortran, Cobol, ...)
2. **Transistors, traitement par lot (1955)**
3. **Circuits intégrés (IC) et la multi prog° (1965-80)**
4. **La miniaturisation et les réseaux d'ordinateurs (1980-..)**

Aperçu historique

1. **Débuts (1946)**
2. **Transistors, traitement par lot (1955)**
   - Industrialisation de (gros) ordinateurs : très chers,
   - Langages : Assembleur mais Fortran et Cobol dominent,
   - Programmes sur cartes, Dump en cas d'erreurs,
   - Utilisation peu efficace des ressources
   - Idée : traitement par lots et enchaînement par un *Moniteur résident* = *ancêtre des SEs*, ..
   - Grosse différence de vitesses entre E/S et UC .
   - Idée : déconnecter les deux et recouvrir E/S (par circuits dédiés, DMA) par une exécution en UC.
3. **Circuits intégrés (IC) et la multi prog° (1965-80)**
4. **La miniaturisation et les réseaux d'ordinateurs (1980-..)**

Aperçu historique

1. **Débuts (1946)**
2. **Transistors, traitement par lot (1955)**
3. **Circuits intégrés (IC) et la multi prog° (1965-80)**
   - Les IC remplacent les transistors ⇒ baisse des coûts
   - Problème de compatibilité : 1 hardware ⇒ 1 SE
   - Nouveaux matériels = nouveaux programmes.
   - Intel Lance 8008 (après 4004) : bon processeur, quel SE ?
   - IBM lance (avec OS/360) l'idée d'un unique SE qui s'adapte à tous matériels,
   - Harmonisation et Standardisation,
   - Arrivée des disques (plus rapide que les bandes et cartes),
   - Meilleur recouvrement (E/S - UC) grâce aux *accès par secteur*
   - Vers la Multiprogrammation ...
4. **La miniaturisation et les réseaux d'ordinateurs (1980-..)**

Aperçu historique

1. **Débuts (1946)**
2. **Transistors, traitement par lot (1955)**
3. **Circuits intégrés (IC) et la multi prog° (1965-80)**
   - **➥ Émergence de la Multi Programmation**
     - Pendant qu'1 prog attend une E/S, un autre programme en UC
     - On a l'impression de posséder la machine pour soi ! ➥ MULTICS
     - *Multics : multiplexed Information & computing service*
     - Miniaturisation des IC ➥ Mini ordinateurs ➥ DEC PDP-1 en 1961.
     - Prix : environ 120 000 $ (environ % 5 d'un IBM 7094).
     - UNICS sur PDP-7, puis rebaptisé UNIX, réécrit en C,
     - Implanté sur tous ordinateurs (VAX, SM90, SUN, ...)
4. **La miniaturisation et les réseaux d'ordinateurs (1980-..)**

Aperçu historique

1. **Débuts (1946)**
2. **Transistors, traitement par lot (1955)**
3. **Circuits intégrés (IC) et la multi prog° (1965-80)**
   ➥ **Émergence de la Multi Programmation**
4. **La miniaturisation et les réseaux d'ordinateurs (1980-..)**
   - Les micros ordinateurs (à partir de 1978 (Mac) puis 1982 (PC))
   - .... quelques années ...
   - Motorola (68xxx) , Intel 80/88, x86, ...
   - Systèmes interactifs, graphique, logiciels (MS DOS, IBM PC)
   - Développement des systèmes Multiprocesseurs, réseaux (ARPANET, SNA, TRANSPAC, ETHERNET...) pour relier les ordinateurs (éventuellement éloignés)
   - INTERNET, WEB , ... (Don't forget Minitel !)

Typologie des SEs

1- Système à soumission de travaux (système *batch*)

- A chaque instant ; 1 seule tâche est exécutée
- La tâche dispose de toutes les ressources
- Ex : micro-ordinateurs (primaires) ou un terminal relié à un hôte distant

2- Système mono/multi tâches

- Multi tâches : le temps d'utilisation du processeur est réparti entre plusieurs tâches
- Les ressources affectées alternativement aux tâches selon les priorités (mécanisme transparent aux utilisateurs)
- Ex. de mono tâche : CPM, DOS
- Ex. de multi tâches : Unix

Typologie des SEs (suite)

3- Système mono/multi utilisateur (Unix, Win, MacOs, ..)

- Un système mono utilisateur (mais mono ou multi tâches) traite les commandes d'un seul utilisateur.
- Un SE multi utilisateurs gère plusieurs utilisateurs ➙ le SE est multi tâches

4- Système mono/multi traitements

- Plusieurs processeurs : certains peuvent être spécialisés
- Multi tâches, multi utilisateur
- Répartition des tâches **entre les processeurs**, en fonction de la spécialité ou de la charge ⇒ optimisation des performances
- Unix est multi tâches, multi utilisateur, voire multi traitement

Typologie des SEs (suite)

De nos jours, on peut rouver sur un bureau :

- Parallélisme, Transputers et Multi cores..

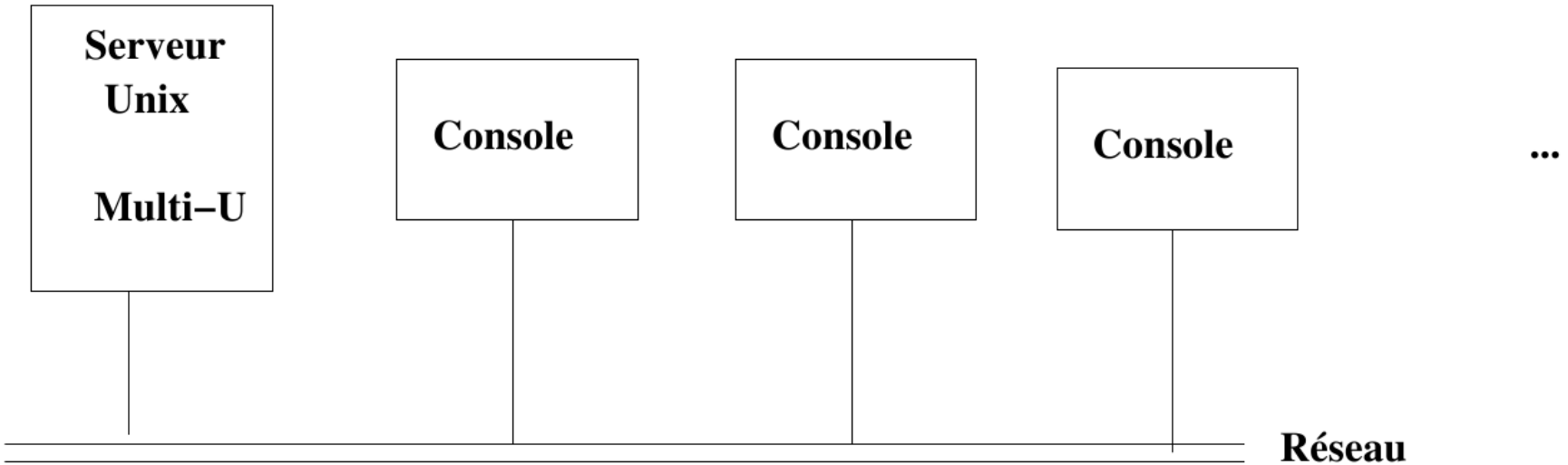
➙ Montée en force des GPU (cartes graphiques puissantes), ...

Ou dans sa poche :

-Systèmes Temps-Réels & Embarqués (contraintes de temps)

Typologie des SEs (suite)

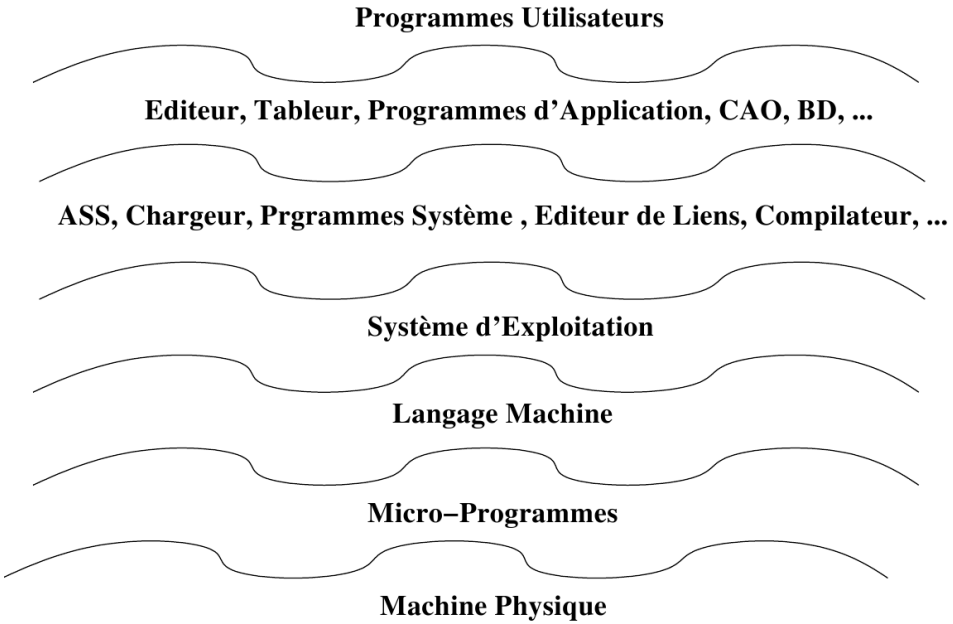
- Multi tâches = multi programmation ≠ multi traitement



- cf. à l'ECL, dans les salles

Objectifs et fonctions essentielles d'un S.E.

• Construire, sur la machine physique telle qu'elle est livrée, une machine VIRTUELLE plus facile d'emploi et plus conviviale.
• Prendre en charge la gestion complexe des ressources en optimisant leur utilisation et permettre leur partage entre les utilisateurs



• Les SEs modernes sont conçus sur une *architecture par couches*

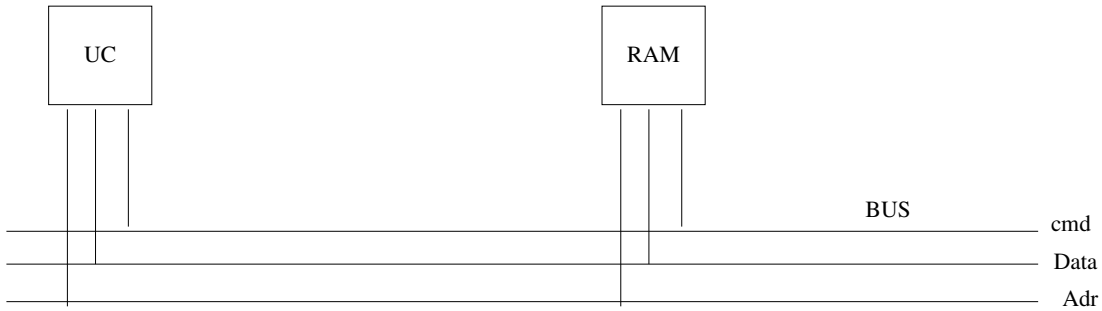
Objectifs et fonctions essentielles d'un S.E. (suite)

Qu'est-ce qui se passe lors de l'exécution de $N \leftarrow N + 1$ (sur une machine basique à Accumulateur)

➥ Une approche intuitive

- L'instruction est décomposée en Assembleur :

Load N
Add #1
Store N



Objectifs et fonctions essentielles d'un S.E. (suite)

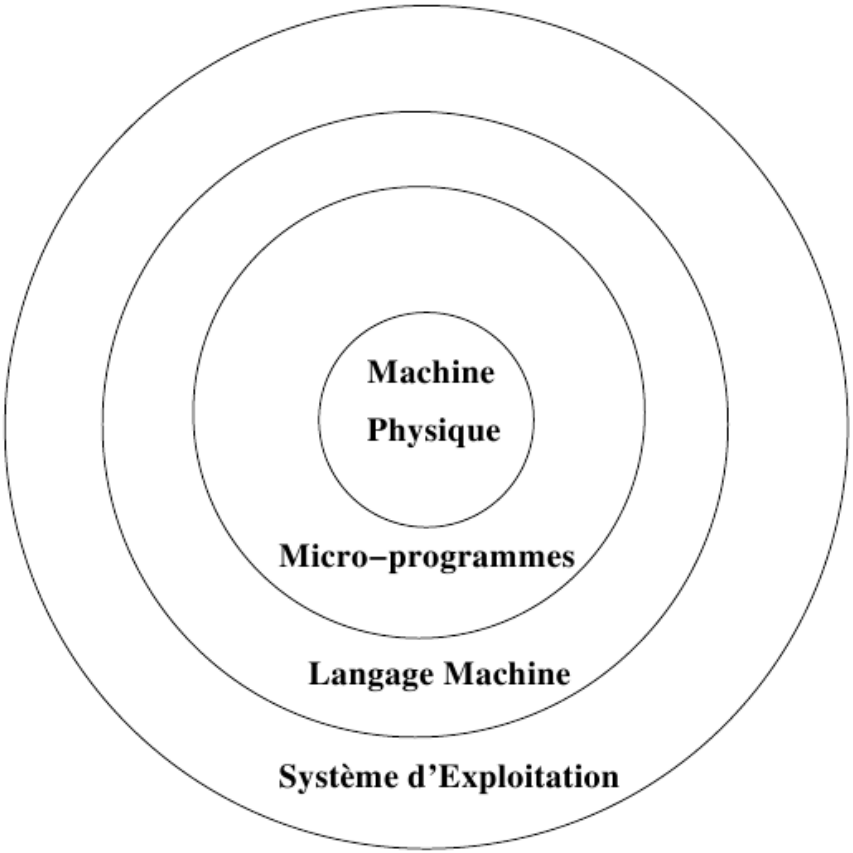
- *Load N* :
  - ➥ UC demande le Bus, dépose 1 cmd "Read RAM", Dépose @N, ...
  - ➥ MMU : reçoit demande lecture, récupère @N, dépose N sur le *Bus Data*
  - ➥ L'UC récupère ...

- La commande *read* : un exemple typique de l'intervention du SE

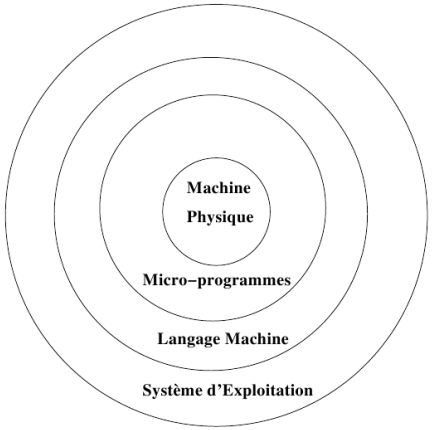
- Comment organiser les choses (UC ne peut pas attendre !) ?
  - Signal (Ready, Done, ..) vs. cycle et top d'Horloge.
  - Interruptions, réquisition de ressources (BUS), priorité, ...

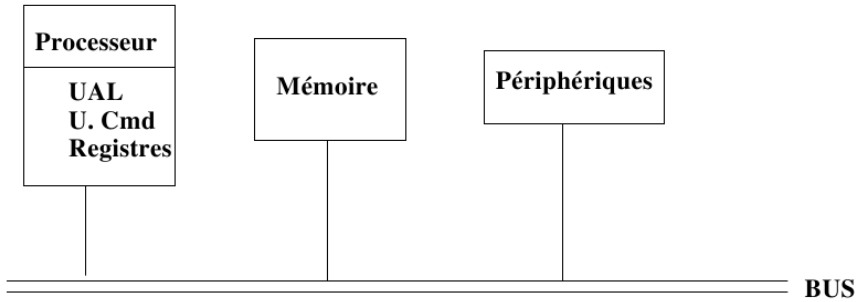
Objectifs et fonctions essentielles d'un S.E. (suite)

Couches fonctionnelles :



Objectifs et fonctions essentielles d'un S.E. (suite)





niveau 0 la machine Φ : *bus, mémoire, UAL, périphs phys., ...*

niveau 1 micro programmes : opérations réalisées directement par le matériel

niveau 2 langage machine : les instructions spécifiques à l'ordinateur.
➥ instructions interprétées par les micro programmes.

niveau 3 SE
➥ interpréteur d'instruction plus complexe qui correspond aux services supplémentaires offerts par SE.

Objectifs et fonctions essentielles d'un S.E. (suite)

- Sur ces couches, on peut définir une hiérarchie de langages :
  celui d'un niveau i est exécuté par une machine logique ou virtuelle M_i
  ➥ En fait, exécuté par les machines d'un niveau $<$ i

Exemple : 1 prog. Pascal $\Rightarrow$ Ass $\Rightarrow$ μ-progs. $\Rightarrow$ Machine Φ
➥ *Le programme Pascal est exécuté par une machine virtuelle PASCAL.*

N.B. : Le découpage en couches ne doit pas être figé (sauf les plus basses)
➥ A certaines époques, certaines fonctions étaient confiées aux logiciels et puis à d'autres époques, aux matériels.

- Normalement, toute fonc. logicielle peut être confiée au matériel et vice versa.
  ➥ Le choix est d'ordre économique ou d'efficacité (technique).
  ➥ De nos jours, les fonctions d'un SE sont plutôt exécutées par le matériel.

L'approche **fonctionnelle** d'un SE permet une vue indépendante de son implantation.

Éléments de Base d'un SE

Processus : notion de base introduite par MULTICS

➤ une abstraction de l'activité d'un processeur .

• Dans un système multi programmé avec un seul processeur (ressource non partageable), le *processus* permet d'exprimer qu'un *programme* est en cours d'exécution ; bien qu'il ne progresse pas e.g. il est en attente du processeur).

• Exemple : cuisine, livre recettes, gâteau, ... une dame

- recette : l'algorithme (code du programme)
- la dame : le processeur
- les ingrédients : données du programme
- lire la recette, trouver les ingrédinets et cuisiner (processus) un gâteau

  ... On attend le gateau ... ! mais :

→ On sonne ➙ Interruption, priorité, contexte, repris, ...

Éléments de Base d'un SE (suite)

- Aller Ouvrir la porte fait suspendre l'activité précédente.
- Chaque processus a son propre programme (recette, ouverture porte)
- La dame reprendra la cuisine après avoir répondu (porte)

1 processus : une abstraction de l'activité d'un processeur
➥ une activité qui possède un code (programme), des données (In/out) et un état courant.

- **Un programme** =

  entité statique associée à une suite d'instructions (recette) ;

- On distingue un programme de son exécution ⇒ notion d'*état*

- **Un processus** =

  entité dynamique associée à la suite des actions (cuisiner) *réalisées par un programme lors d'une exécution particulière.*

➧ Cette suite est indépendante du nombre de fois et des instants où le processus a été mis en attente de processeur.

Éléments de Base d'un SE (suite)

• Dire qu'un processus P est parvenu à l'action "lire la case M" signifie :
- soit le processeur est réellement entrain d'exécuter l'instruction correspondant à l'action,
- soit P est mis en attente, il exécutera l'action quand il aura le processeur,
- soit P a exécuté l'action et a été mis en attente (pour la suite).

- *début d'un programme* = la 1ère instruction (sur le papier !)
- *début d'exécution d'un programme* = l'exécution de la 1ère instruction (début du processus)
- La suite d'instructions exécutées ne permet pas de décrire entièrement comment le programme est exécuté (détails).
  ➥ Information d'**Etat**.

Eat d'un processus

Exemple :

Une maman prépare une recette avec ses ingrédients (lait, oeufs, farine, sucre,...).

- La recette = l'algorithme (suite d'instructions)
- la maman = le Processeur
- les ingrédients = les données
- L'activité de ce processeur est de lire la recette, trouver les ingrédients et faire cuire le gâteau.
- Soudain, le fils de la dame arrive avec une piqûre d'abeille,
- La maman marque l'endroit où elle se trouve dans la recette (l'état du processeur est ainsi sauvegardé)
- Elle cherche un livre de soins et commence à soigner son fils.
  Les soins : selon un autre algorithme, d'autres données, ... (le processeur est le même)

Eat d'un processus (suite)

• Le processeur (la dame) est passé d'un processus (la cuisine) à un autre, plus prioritaire (soins).

➥ Chaque processus a son propre programme.

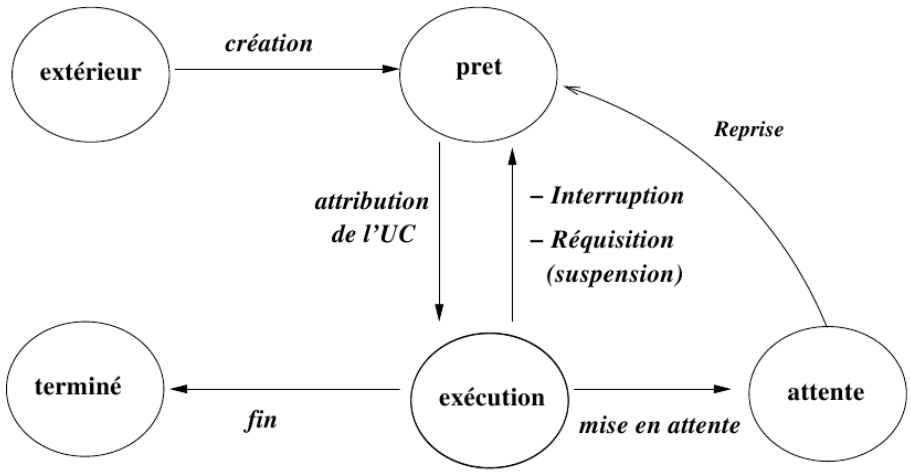
➥ Lorsque la piqûre d'abeille aura été soignée, la dame reprendra sa recette là où elle l'avait abandonnée (commutation vers l'ancien contexte).

> Un processus est une activité d'un certain type qui possède un programme (code), des données en entrée et en sortie ainsi qu'un état courant.

Etats possibles d'un processus

- Un processus ... est soit
  - réellement exécuté (possède l'UC) ;
  - prêt à être exécuté (en attente du processeur) ;
  - en attente de ressource/événement (e.g. fin d'un E/S) ;

Diagramme (simplifié) d'états :



Etats possibles d'un processus (suite)

- Les états importants : "en exécution", "prêt", "en attente" (d'autre ressource que le processeur) , "terminé", "extérieur", ...
- Il y a des états associés : "bloqué", "suspendu",...
- On pourra aussi distinguer les causes d'attente et l'attente en mémoire centrale/secondaire,...
- Les états sont modifiés par le SE, sous l'effet d'un **événement**.
- Les événements peuvent être internes au processus :
  ➥ une demande d'E/S fait passer de l'état "en exécution" $\Rightarrow$ "en attente".
- Ils peuvent être externes (depuis le SE) :
  ➥ l'attribution d'une ressource fait passer de l'état "en attente" $\Rightarrow$ "prêt".

• La nécessité de conserver un *état* pour tout processus
• L'état d'un processus permet sa reprise
• Les données (de reprise) qui caractérisent un processus sont stockés dans une structure appelée *Processus Control Bloc (PCB)*

Opérations sur les processus

Réalisés par le SE :

- *créer / détruire*
- *mettre en attente / réveiller*
- *suspendre / reprendre*
- *modifier la priorité*

Création Processus :

• Un processus peut en créer un autre (père - fils)

• Le fils peut à son tour en créer d'autres

➙ graphe du père et ses descendants = 1 structure partiellement ordonnée.

• On appelle parfois "job" l'ensemble du processus père et ses descendants

• Au **boot**, un processus spécial (*init*) est l'ancêtre des autres

.

Opérations sur les processus (suite)

- **Exemple d'Unix : *fork***
  - Permet de duppliquer le processus exécutant (copie identique)
  - Renvoie 0 au fils et le numéro d'ID du fils au père
  - Copie le code et les données du père
  - On utilise en général un "exec" pour exécuter un code différent (fichier)

- "exec" remplace le code et les données du processus (qui l'exécute) par ceux du fichier chargé
- le processus reste le même mais le programme exécuté change
- "fork" + "exec" permettent l'exécution en parallèle du père et du fils.

Opérations sur les processus (suite)

Example

```
idfils = fork () ;
if (idfils == 0)                    // en est dans le fils
    exec(fichier disque)
else                                 // on est dans le père
    <continuer la travail du père>
```

- Selon les systèmes, on peut visualiser la liste des processus :
  - ➙ Unix (MacOs, Linux, etc.) : *ps*
  - ➙ Windows : passer par l'interface graphique
  - ➥ On peut en arrêter (*kill, signal*)

- Exemple "fork_exec2" :
  le père crée un fils et lui passe un message, puis les deux affichent des messages.

Opérations sur les processus (suite)

Attendre / Réveil

- "en attente" : demande de ressource (autre que l'UC) non disponible ;
- "réveil" : quand le S.E. peut lui affecter les ressources
  ➥ le processus passe à l'état "prêt".

Exemple :

Unix donne la priorité au processus qui fait une E/S par rapport à celui qui va faire une demande d'E/S.

➥ permet de garder le processeur et le périphérique plus souvent occupés
➥ maximise le parallélisme.

Opérations sur les processus (suite)

Suspendre / reprendre

- similaire à une mise en attente (save contexte)
- permet de donner “son tour” à chaque processus.
- à la repise ⇒ “prêt” ou “en attente” (selon l’état avant la suspension)
- **Equité** : ne pas suspendre un processus trop souvent/trop long temps ;
- un processus en attente de données (d’un autre processus) peut être suspendu (Attente ⇒ suspension)
- le SE peut suspendre un processus (pour efficacité : trop de processus, performances, ..)

Changement de priorité

- Utilisé par l’ordonnanceur (augmentation/baisse)

➥ Gestion des priorités et Famine / Equité

Recouvrement Etats des processus

- Le SE gère ses processus, en particulier en **phases de calcul et d'E/S**
- Un processus (v. un programme) : phases de calculs + des E/S
- Phase de calcul extrêmement courte, E-S trop longue (statistiques).
  ➥ Cause : différence de vitesse UC/périphériques.

Example

$ cat f1 f2 | grep "main"
➥ *"grep"* calcule sur des sorties de *"cat"*

➥ Deux processus (faire *ps* pour les voir)
➥ "grep" (à l'état "prêt") risque d'attendre les données de "cat" $\Rightarrow$ attente
➥ La lecture du "pipe" est comme l'attente au clavier.

Example (attente longue !)

$ cat f1 - | grep "main"
➥ "grep" attend "cat" qui attend le clavier (paramètre "-" de "cat").

Mécanismes de Base d'un SE

- Ressources
- Ordonnancement
- Commutation d'état
- Interruptions

Ressources

- **Une ressource** : élément qui contribue à la progression des processus.
- Deux types de ressouces : matérielle et logicielle :
  - Ressource matérielle (hardware) : UC, mémoire, réseau, ...
  - Ressource Logicielle (software) : les applications, compilateurs, ...

Remarques :

- Le *dispatcheur, chargeur, ordonnanceur, ...* sont des ressources logicielles.
- Un **processus** est une ressource logicielle (peut être appelé).

- Une ressource est décrite par un **TDA : classe ressource.**
- Un **objet ressource** (instance) : e.g. une case mémoire.
- Opérateurs du TDA ressource : *demander, allouer, utiliser, libérer*
- La demande (via une commande du SE), l'allocation et la libération sont faites par le SE.

Ressources (suite)

Types d'utilisation de ressources :

- Ressouce **réutilisable** : on peut la demander, libérer, redemander, ... (e.g. mémoire)
- Ressources **non réutilisable** : disparaissent après utilisation (e.g. un message).

- Gestion des ressources par des tables de ressources
- Etats : *disponible* , *allouée* (à qui ?), ..
- A chaque ressource est associée une **file d'attente** (de PCB)
- **Réquisition** de ressource : ressource demandée par un processus prioritaire.
  ➥ Sauvegarde de l'état pour la restoration.

Ordonnancement

• Dans un système multi programmé, l'ordonnanceur définit l'ordre d'attribution d'une ressource.
  ➥ e.g. l'UC + sa durée d'utilisation à un processus .
  ➥ Bon algo → bon taux d'utilisation → performances.

• Un ordonnanceur par ressource (e.g. pour la mémoire, les périphériques, ..).

• Le plus important des ordonnanceurs : **Scheduleur** (UC).

Utilisation de la Files d'attente de'une ressource :
• Demande d'une ressource non disponible ⇒ file d'attente
• Quand la ressource est/devient disponible :
  ➥ Sélection d'un processus par l'ordonnanceur.
    ➧ Le processus va dans la file "prêt" (éligibles).

Types d'ordonnanceurs*

Un ordonnanceur par ressource.

- Ordonnanceur à **long terme** (job scheduler) :
  - gère la demande d'entrée dans le système
  - la demande peut être refuseé/différée (trop de processus, perf., ..)
- Ordonnanceur à **court terme** (de l'UC) :
  - Choisit dans "prêt" le prochain
  - L'ordonnanceur est activé (au moins) pendant la phase de calcul du processus en cours (env. 10 ms.)
  - l'ordonnanceur à long terme est appelé moins souvent (env. 10 s.)
- Ordonnanceur à **moyen terme** (medium term scheduler) :
  - Si swapping (disque ↔ mémoire), le **medium term scheduler** décide quel processus, rangé sur le disque, peut accéder à la mém. centrale (MC).
  - Dans ce cas, un autre ordonnanceur gère les accès aux disques.

• Une fois le processus élu connu, l'ordonnanceur de l'UC (à court terme) appelle un autre processus système nommé **distributeur (dispatcher)** dont le rôle est de donner effectivement l'UC au processus élu.

Algorithmes d'ordonnancement

- But : augmenter les performances et le taux d'utilisation
- Dépend de la ressource
- Différents types d'algorithmes

On les verra plus loin.

Autres Mécanismes de base

Deux mécanismes de base implantés par matériel :
- La commutation du mot d'état (partie du contexte)
- Les interruptions

Commutation de contexte :
• Pour être exécuté, un programme et ses données sont chargés en MC puis les instructions sont transférées de la MC vers l'UC.

• L'UC contient des circuits (UAL) et des registres
• Utilisation juducieuse pour diminuer les accès mémoire (cf. optimisation dans les compilateurs).

Importance du contexte :
• A tout instant, un processus est caractérisé par :
- Un programme et des données (= contexte en mémoire) ;
- Un ensemble de registres de l'UC = mot d'état (= contexte de l'UC)

Le PCB contient ces informations propres à chaque processus.

Autres Mécanismes de base (suite)

Les registres de l'UC :

- Registre d'instruction : instruction en cours
- Compteur Ordinal (PC) : l'adresse de la prochaine instruction
  ⇒ La valeur du PC est modifiée pendant l'excution de l'instruction en cours pour pointer la prochaine .
  ⇒ L'exécution d'une instruction provoque des transferts UC ↔ MC.
  ⇒ Registre d'adresse : l'adresse de la cellule mémoire impliquée
  ⇒ Registre de données : l'informations à transférer
- Autres registre important : registres d'état de l'UC.
  ➥ indiquent : actif / inactif , mode super / user,
  - codes des commandes
  - infos sur l'UC et sur le processus en cours : zone mémoire accessible, droits d'accès, priorité, ...
  ⇒ L'ensemble de ces registres d'état = **mot d'état** (**PSW**)
  ⇒ Le registre d'état fait partie du mot d'état.

- A tout instant, un processus est caractérisé par :
  Un programme (code) + Données + PSW

Commutation du mot d'état

Importance du PSW et la commutation :

- Scénario : plusieurs processus en mémoire, prêts à être exécutés
- A un instant donné de son exécution, un processus P est caractérisé par ses contextes (mémoire et UC)
- Si un autre processus Q reprenanit son exécution, il changerait le PSW.
- Pour pouvoir reprendre un processus, il faut sauvegarder ses contextes ;
- La commutation de mot d'état permet de :
  - sauvegarder l'état de P dans une zone précise (vers le PCB) ;
  - charger un nouvel état (de Q) depuis son PCB
- La commutation même provoque l'exécution d'un autre processus (PC)
- Une commutation peut demander à Q de compléter, avant son début, la sauvegarde de P.

Interruptions

- Une interruption (it°) = déroutement
  ➥ Une commutation de mot d'état provoquée par un signal généré par le matériel (v. aussi **trap**)

- Ce signale est la conséquence d'un événement (evt) qui peut être :
  - **interne** au processus en cours (e.g. demande d'E/S) et donc résultant de son exécution ;
  - **externe** au processus, e.g. un evt. venant d'un autre processus (signal), d'un périphérique d'E/S, de l'opérateur, alarme, capteur, ...

- Le signal modifie un indicateur (e.g. 1 bit/ 1 mot/ 1 vecteur) consulté régulièrement par le système (examen après chaque instruction) ;

- On peut avoir plusieurs indicateurs (1 par type d'evt $\Rightarrow$ plusieurs causes)
  ➥ En général : plusieurs bits du PSW $\Rightarrow$ vecteur d'it°
- Le système détermine la cause de l'it°
  ➥ à chaque cause est associé un **niveau** d'it°

Interruptions (suite)

- il y a en général au moins 3 types (niveaux) d'it° :
  - **HardWare** : les it° externes au processus (e.g. pannes, opérateur, ...)
  - **SoftWare :**
    ◗ les déroutements (erreur, div/0, overflow) liées à l'instruction en cours
    ◗ les appels système = SVC (e.g. une demande d'E/S).

Interruptions (suite)

- Les it° sont traitées selon leur niveau (= priorité, importance)
  - ⏵ Exemple des niveaux sur IBM 360/370 (5 est le plus élevé) :

    niveau 5 : erreurs matérielles

    niveau 4 : déroutement (soft, trap)

    niveau 3 : appels systèmes

    niveau 2 : it externes

    niveau 1 : E / S

- Lors d'une it° ⇒ reconnaissance + commutation de contexte
  - ⏵ commutation décidée selon le niveau de l'it°
- Le processus en cours est suspendu ou mis en attente (selon le type et la cause) ;
- Chargement du mot d'état provoque l'exécution d'un **traitant** (*it handler*) ;
  - ⏵ un traitant associé à chaque type d'it°

Interruptions (suite)

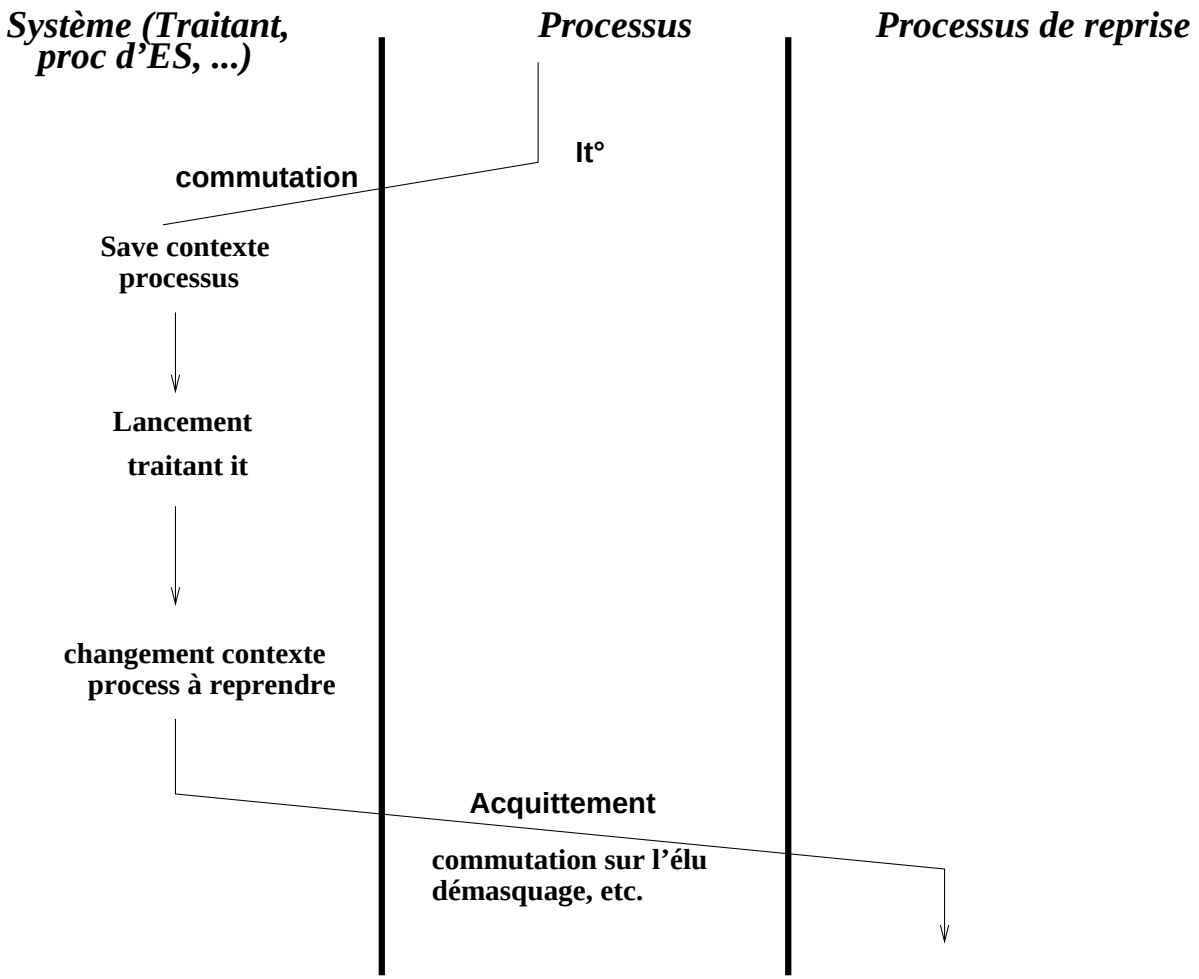
- Exemple des niveaux et les traitants sous Unix :
  - niveau 0 : it Horloge (handler : clockintr)
  - niveau 1 : it disque (hendler : diskintr)
  - niveau 3 : it console (ttyintr)
  - niveau 4 : it autre périphérique (devintr)
  - niveau 5 : appel system (sottintr)
  - niveau 6 : autre it (otherintr)

- Le traitant peut compléter la sauvegarde du contexte de l'ancien processus ;
  - si le même processus doit reprendre (e.g. cas d'appel de l'opérateur), on ne sauvegarde pas le contexte mémoire.
  - le minimum à sauvegarder (par matériel) est le PSW, le reste peut être fait par programme (e.g. par it handler) ;
  - le traitant s'exécute puis l'ordonnanceur charge le contexte d'un processus (peut être l'interompu) ;

  ➥ Cette opération sera **l'aquittement** de l'it°.

Interruptions (suite)

• Illustration : Arrivée d'it° ➙ traitant ➙ chargement de contextes + commutation (= aquittement).



Priorité, masquage et désarmement*

- Une/plusieurs it° ⇒ plusieurs indicateurs positionnés
  - pour une it° = le niveau ⇒ la priorité
  - le SE traite la plus prioritaire (niveau le +élevé d'abord)
- Pendant l'exécution d'un handler, une it° peut arriver !
  - si ceci se répète, le SE ne progresse plus !
- Il faut retarder / annuler la prise en compte d'une it°
  - Technique : **masquage** et **désarmement** de niveau
  - Un masque d'it° : la suite de valeurs des indicateurs.

  ⇒ On retarde la prise en compte des it° d'un/plusieurs niveaux
  → masquage de ce(s) niveau(x).

  ⇒ Si l'on ne veut pas les traiter → désarmement
- Le masquage fait par un indicateur du PSW consulté à l'arrivée d'une it°
  ⇒ **peut-on me déranger** ? ? Dépend du niveau et du masque.
- L'indicateur du niveau à 1 ⇒ niveau masqué
- Le démasquage : on remet l'indicateur à sa valeur d'origine.

Priorité, masquage et désarmement* (suite)

• Le masque se modifie avec les commutations.

• Pendant l'exécution d'un traitant d'un niveau N, on peut masquer les it° de niveau inférieur.

• Quand le traitant fait l'aquittement (appel de l'ordonnanceur), la nouvelle commutation restore (démasque) les niveaux masqués.
- Les it° survenues pendant le traitement peuvent être sauvegardées et traitées
- Elles sont sauvegardées (save au niveau hard possible) sauf si désarmement (= on n'en veut pas !)

• On peut complètement supprimer la prise en compte d'un niveau
$\Rightarrow$ désarmement du niveau (+ réarmement ultérieur)

• Pour désarmer : un indicateur de PSW $\Rightarrow$ désarmer le niveau

• Dans les processeurs, certaines it° ne sont pas masquables (car trop déterminantes)

Priorité, masquage et désarmement* (suite)

Remarques :

• Certains déroutements (= it° logicielles , e.g div/0) ne peuvent pas être masqués (car lié à l'instruction en cours)

→ si on les traite plus tard → on plante !

• D'autres (non masquables) peuvent être désarmés

• N.B. : pour désarmer :

➥ comme un "return" dans le traitant → pas de save ni traitement

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* (suite)

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).

Entrées Sorties physiques (ES Φ)

- Deux opérations de base : **sélection** du périphérique, **transfert** des données

⇒ la sélection au niveau matériel ;

⇒ le transfert des données via le matériel et modules système ;

⇒ Plusieurs méthodes de transfert.

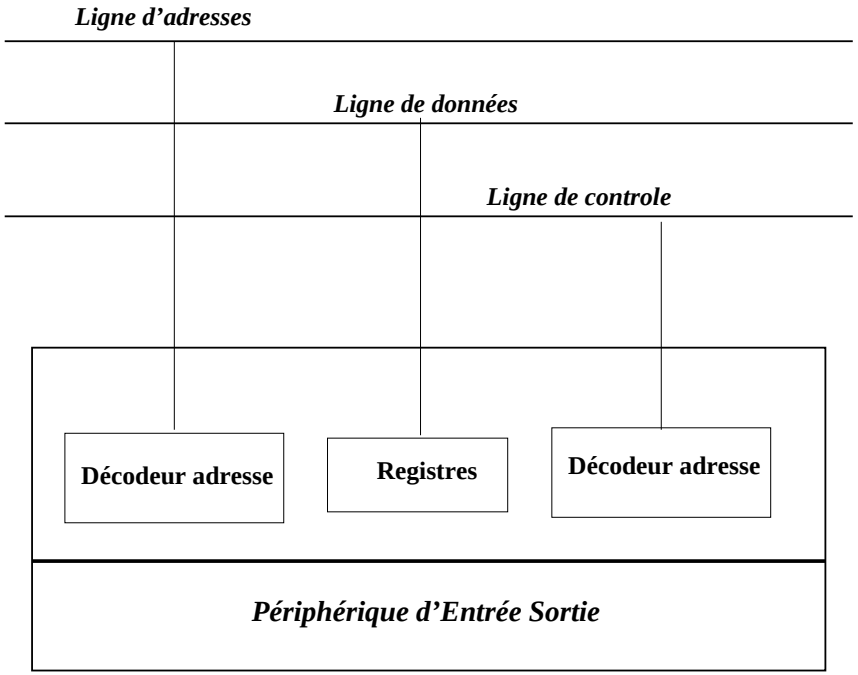


fig III.1 : périphérique et bus

Méthodes de transfert

Méthodes de transfert :

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

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

Méthodes de transfert (suite)

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.

Méthodes de transfert (suite)

Canal de commandes (command channel $\simeq$ DMA)

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

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

$\Rightarrow$ le DMA = un Canal de commande qui exécute un programme d'une seule instruction (de transfert).

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 :*

$\rightsquigarrow$ *e.g. l'impression ne doit pas dépendre de la marque de l'imprimante.*

$\rightsquigarrow$ *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.*

Périphériques virtuels d'Entrées Sorties (suite)

⇒ Conséquence :
Ne pas s'adresser directement aux périphériques mais aux flots (périphériques d'ES **virtuels**).
⇒ C'est à dire :

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

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*

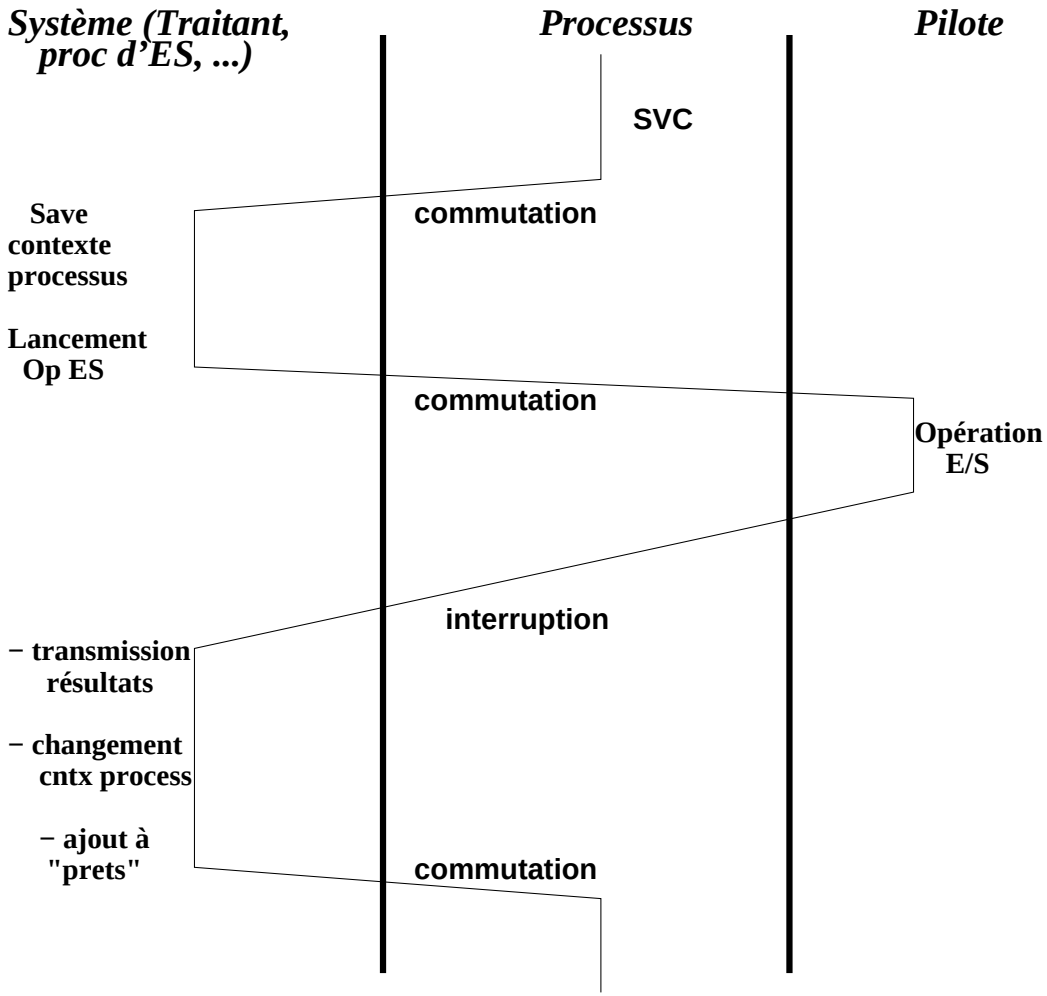
⇒ *L'user utilise la même commande pour écrire sur des périphs différents.*

Correspondance ES Logique (virtuelle) ↔ ES Φ

- read ↦ SVC ↦ Interruption (ES) ↦ Commutation ↦ Traitant
- Le traitant d'interruption (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 ↔ 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 ↦ Interruption (fin ES) ↦ Handler ↦ transmission résultats ↦ ajout PCB demandeur à "prêts"

Correspondance ES Logique (virtuelle) ↔ ES Φ (suite)



⇒ Ce shéma ne traite pas le recouvrement.

Correspondance ES Logique (virtuelle) ↔ ES Φ (suite)

Problème liés aux ES

3 problèmes : signaux simultanés, recouvrement, protection.

1- Traiement des signaux et Interruption simultanées :

- Une demande d'ES ↦ 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*).*

Correspondance ES Logique (virtuelle) ↔ ES Φ (suite)

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)*

Technique de Tampon

- Utilisée pour les accès disque / mémoire
- But :

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

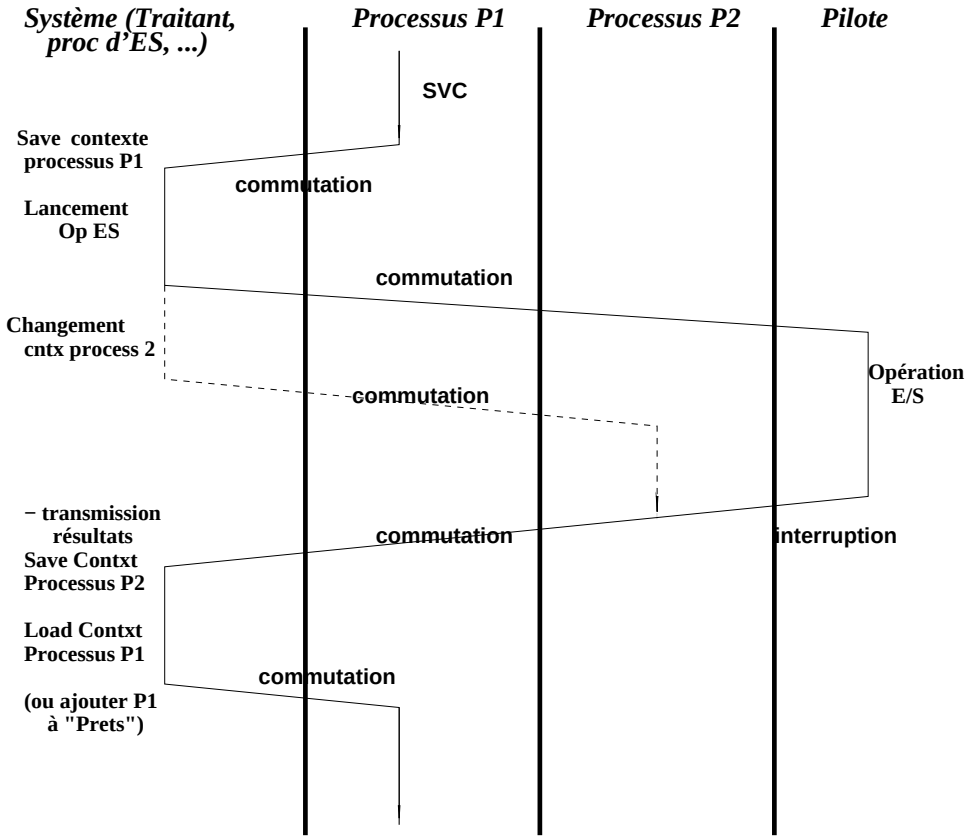
⇒ Se servir d'un tampon pour :

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

Correspondance ES Logique (virtuelle) ↔ ES Φ (suite)

• 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.
⇒ 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 :

Correspondance ES Logique (virtuelle) ↔ ES Φ (suite)



Correspondance ES Logique (virtuelle) ↔ ES Φ (suite)

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

Correspondance ES Logique (virtuelle) ↔ ES Φ (suite)

- 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)

Processus*

Elément de base du SE

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 T_i : d_i ... f_i

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

f_i : écriture des résultats, libération des ressources, save des infos, etc.

- Une tâche T_i réalise une fonction F_i
- La suite $< d_1 f_1 d_2 f_2 ... d_n f_n >$ est associée à $P = T_1 T_2 .. T_n$

Processus* (suite)

Un mot de tâche : toute suite construite sur $A = \{d_1, f_1, d_2, f_2, ..., d_n, f_n\}$

- Un processus peut être divisé en différentes séquences de tâhes.
  ➥ Pour simplifier, on suppose qu'il n'y a pas d'E/S entre d_i et f_i

Example

$T : N := N + 1; \qquad F(n) = n + 1; \qquad T = T_1\ T_2\ T_3$
$T = d \ldots f = d_1 f_1 d_2 f_2 d_3 f_3$ (dans le cas séquentiel simple)

$T1 : Load\ R, N$
$T2 : Add\ R, \#1$
$T3 : Store\ R, N$

Processus* (suite)

Système de tâches et graphe de précédence

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

1. $\forall$ T $\in$ 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 :

$T_i < T_j$ ssi la terminaison de T_i a lieu avant l'initialisation de T_j.

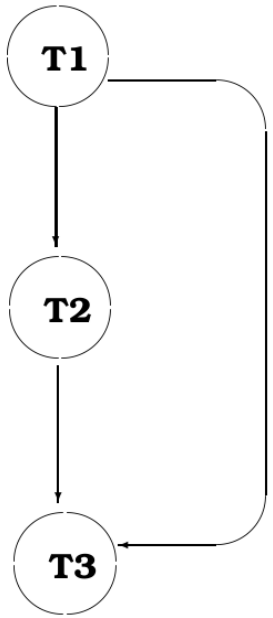
• Si on n'a ni $T_i < T_j$ ni $T_j < T_i$, l'ordre d'exécution de T_i et de T_j n'a pas d'importance (peuvent s'exécuter en parallèle).

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

Processus* (suite)

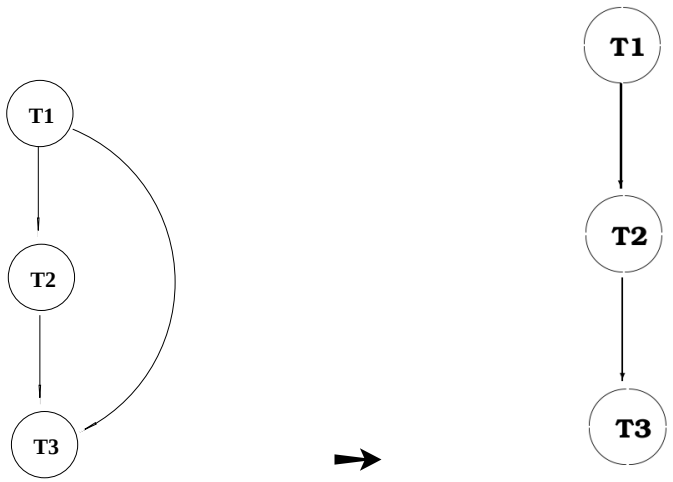
Exemple (graphe non compacte) :

- Il existe un arc entre T_i et T_j ssi $T_i < T_j$.



Processus* (suite)

- Le graphe *compacte* (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 :
    $$T_i \rightarrow T_j \text{ SSI } T_i < T_j \text{ et } \forall\, T_k, \text{ on n'a pas } T_i < T_k < T_j.$$
- Pour l'exemple précédent, le graphe compacte est :



Processus* (suite)

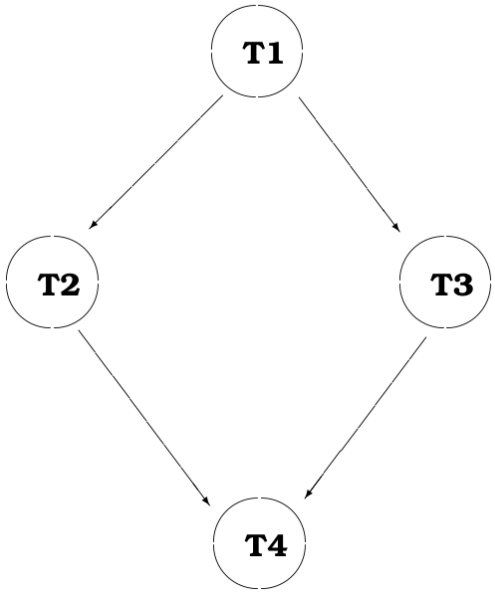
- T_j est *successeur* de T_i ssi $T_i \rightarrow T_j$.
- Une tâche sans *prédécesseur* (*successeur* d'aucune tâche) est dite *initiale*.
- Une tâche sans successeur est dite *terminale*.
- T_i et T_j sans aucune relation (exécutables en parallèle) sont dites *indépendantes*.

Langage des tâches :

- mots de tâches construits sur l'alphabet $A = \{d_1, f_1, d_2, f_2, ..., d_n, f_n\}$
- Pour $T = T_1 T_2 T_3$, $A = \{d_1, f_1, d_2, f_2, d_3, f_3\}$, on peut construire $6\,!$ mots.
- Certains mots seront éliminés selon les contraintes de précédences.

Processus* (suite)

Exemple de graphe et de langage des tâches :



- Dans cet exemple, les mots suivants sont acceptables :

$d1f1d2d3f3f2d4f4 \quad | \quad d1f1d3d2f2f3d4f4 \quad | \quad d1f1d3f3d2f2d4f4$

  - Ces mots satisfont la relation de précédence exprimée par le graphe.

Processus* (suite)

- 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 :*

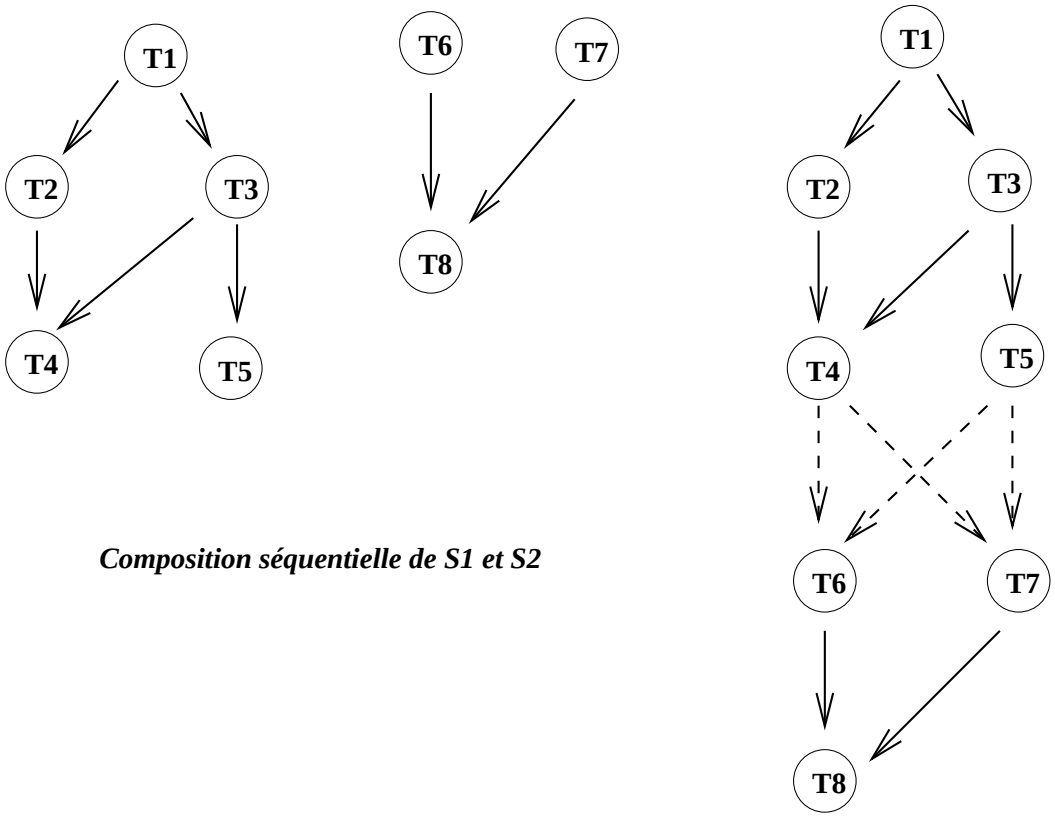
  - $\forall\, T_i \in E$, *ω contient exactement une occurrence de di et une de fi ;*
  - $\forall\, T_i \in E$, *di est avant fi (di figure à gauche de fi)*

- ⏵ Chaque mot décrit un **comportement** possible de S.

Processus* (suite)

Composition Séquentielle de systèmes de tâches :

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



Composition séquentielle de S1 et S2

Processus* (suite)

Composition Parallèle de systèmes de tâches :
• 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 :

$$\boxed{L = \{\omega = \omega1\omega2, \omega1 \in L1, \omega2 \in L2\}}$$

• 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.

Spécification dans les langages évolués

Spécification de systèmes de tâches dans les langages évolués :

- Séquence : séparé par ' ;' dans un bloc *begin* − *end* :

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

  - En notation textuelle, on écrira également $(T1\ T2)$

- Parallèle : séparé par ' ;' dans un bloc *parbegin* − *parend* :

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

  - En notation textuelle, on écrira également $(T1||T2)$

Spécification dans les langages évolués (suite)

Exemple

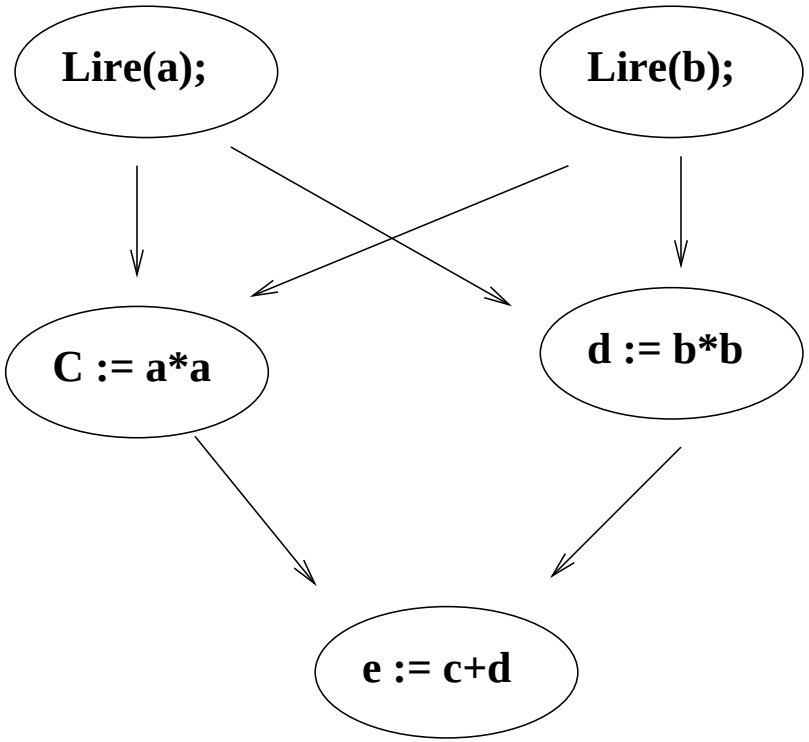
```
begin
    parbegin
        Lire(a) ; Lire(b) ;
    parend

    parbegin
        C := a*a ; d := b*b ;
    parend
    e := c+d ;
end
```

⇒ Une notation textuelle :

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

Spécification dans les langages évolués (suite)

- Cette expression correspond au graphe :



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

Spécification dans les langages évolués (suite)

Exemple 2 :

$$\boxed{\textbf{T1} : N := 0; \ \textbf{T2} : N := N + 1; \ \textbf{T3} : N := N + 1; \ \textbf{T4} : \mathit{affiche}(N);}$$

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

Exemple

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 peut montrer ce problème en examinant les détails du code ../..

Spécification dans les langages évolués (suite)

- 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
```

⇒ Pour la partie $(T2||T3)$ du système , 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'instructions de base) sont possibles.

Spécification dans les langages évolués (suite)

Une conclusion :

- Utiliser *parbegin...parend* avec précaution.
- Utiliser les mécanismes de **protection et de partage**.

Autres formalismes et modèles :

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

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)
    ➥ Avec mémoire commune : protection et partage de zones
  - Les messages sont aussi utilisés dans un système centralisé

Interaction de processus* (suite)

• Interactions mal contrôlées ⇒ Problèmes de blocage, etc.

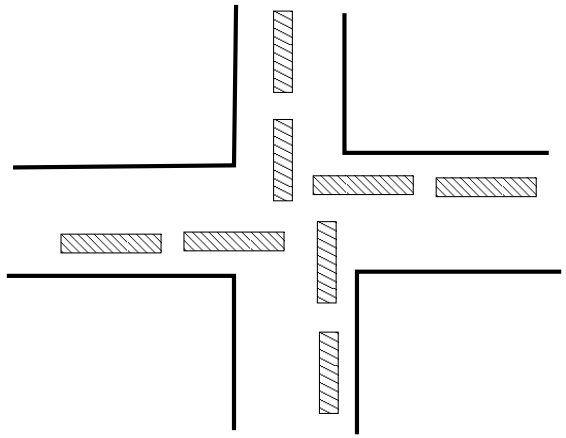
• **Problème-1** : accès concurrent aux ressources.
Vu dans l'exemple précédent (des tâches et accès concurrent à la mémoire).

- 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.

Interaction de processus* (suite)

- **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

Interaction de processus* (suite)

• La modélisation par système de tâches ne fournit pas assez d'informations 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 $\omega \in L(S) = \{a_1, ..a_l\}, a_i \in \{di, fi\}$
- une séquence SE d'états associée à ω : $SE = s_0, ..s_l$ 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).

Interaction de processus* (suite)

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 :**
$T11 : Lire(X)$	$T21 : Lire(Z)$
$T12 : X := X + Z$	$T22 : Y := X + Z$
$T13 : affiche(Y)$	

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

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

Interaction de processus* (suite)

Exemple (Rappel)

processus P1 :	**processus P2 :**
$T11 : Lire(X)$	$T21 : Lire(Z)$
$T12 : X := X + Z$	$T22 : Y := X + Z$
$T13 : affiche(Y)$	

• Pour le système précédent, plusieurs comportements possibles avec des résultats différents (système **indéterminé**)

- Le seul comportement (déterminé) : $(T_{11}||T_{21})T_{12}T_{22}T_{13}$

Propriété non-interférence $\leftrightarrow$ Caractère déterminé

Interaction de processus* (suite)

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**

Interaction de processus* (suite)

Dans l'exemple précédent :

processus P1 :	**processus P2 :**
T11 : $Lire(X)$	T21 : $Lire(Z)$
T12 : $X := X + Z$	T22 : $Y := X + Z$
T13 : $affiche(Y)$	

$\Rightarrow$ On a : - $L_{22} = \{X, Z\}$ et $E_{12} = \{X\}$

- T12 et T22 sont **interférantes** car T12 n'est pas prédécesseur / successeur de T22 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**.

Interaction de processus* (suite)

• Pour vérifier que 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.

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

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 .

Blocage

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...

Synchronisation de processus

- Section critique et problèmes de synchronisation et d'exclusion mutuelle
- Solutions logicielles : algorithmes
- Solutions matérielles : Tas, interruptions, etc.
- Sémaphoes
- Moniteurs
- ...

Synchronisation de processus (suite)

• Outil : la relation de précédence sur un ensemble de tâches dans un système S pour établir un ordre d'accès aux ressources.

⏵ Mais cette approche est insuffisante / indésirable pour certains problèmes, e.g. gestion d'événements **asynchrones**.

• **Exemple** : une ressource, 2 processus :

Exemple

Deux gardiens placés aux deux entrées d'un magasin ;
Ils comptent les clients qui entrent dans dans le magasine par une variable unique (la ressource partagée) N.

Un autre exemple : le cas des tickets au rayon poissonnerie d'un magasin avec des vendeurs différents qui incrémentent un même et unique compteur.

Synchronisation de processus (suite)

Algorithme :

```
Tantque "le magasin est ouvert" faire
        Si entrée alors N := N+1 ; Finsi ;
FinTq ;
```

Le **programme parallèle** de cette activité :

```
Début
        N := 0 ;
        Parbegin Compteur1 ; Compteur2 ; Parend
Fin
```

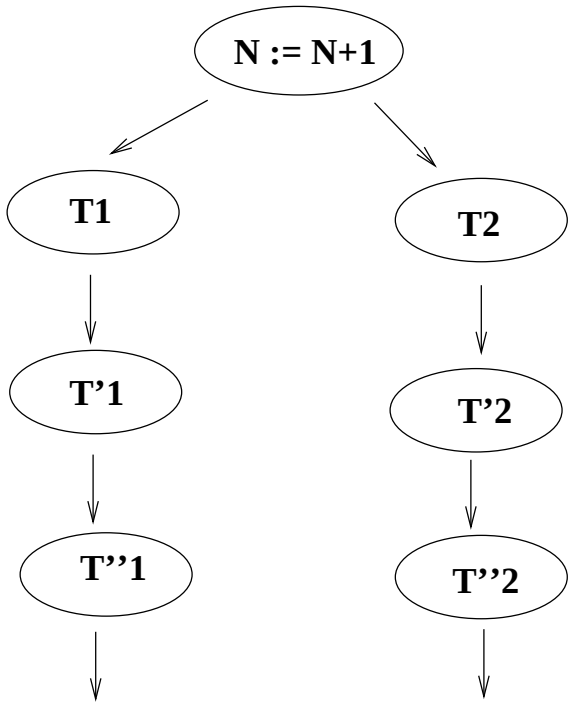
N.B : le comptage doit avoir lieu en parallèle car l'ordre des entrées des clients est imprévisible.

• On montre le problème d'accès à N par le système de tâches :

⏵ Soit $T1 = d1f1$; $T'1 = d'1f'1$..... la représentation des incrémentations successives (la boucle *Tanque*) pour *compteur1*. Idem pour *Compteur2*.

Synchronisation de processus (suite)

Un graphe de précédence :



T1 : Compteur1 (N=N+1)
T2 : Compteur2 (N=N+1)

Synchronisation de processus (suite)

Rappel :

T1 : Compteur1 (N=N+1) : $d_1\ f_1$

T2 : Compteur2 (N=N+1) : $d_2\ f_2$

- La condition de **BERNSTEIN** n'est pas réalisée.
- On peut avoir des comportements *correctes* :

e.g. $\boxed{d1\ f1\ d2\ f2\ ,\ d1\ f2\ d1\ f1}$

mais aussi $\boxed{d1\ d2\ f1\ f2\ ,\ d1d2f2f1\ ,\ d2d1f1f2}$, etc.

- Solution : T1 et T2 doivent être **indivisibles** (*atomiques*).

➥ Il faut avoir ($T1\ T2$) ou ($T2\ T1$) sans rien imposer aux gardiens.

- Les solutions transforment un graphe G pour obtenir un graphe équivalent en ajoutent des tâches à G pour rendre certaines tâches atomiques.

➧ Problème de **section critique** avec des solutions logicielles et matérielles.

➧ Primitives de synchronisation pour rendre certaines exécutions indivisibles (en les enfermant dans des sections protégées).

Problème de section critique

• N processus parallèles : $P1$ || $P2$ || $P3$ || .... avec pour chacun la structure :

```
Répéter
        < Section restante >        % les 2 sections exécutées
        < Section critique >        % en séquence
Jusqu'à faux
```

Problème : pour cetrains comportements du système, plusieurs processus peuvent être à un instant donnée dans la section critique.

➥ **Une solution** : Transformer les graphes (processus) pour qu'à tout instant, seul un processus soit dans une section critique.

On dit : l'entrée et l'exécution du code de la section critique (SC) s'effectue en **exclusion mutuelle**

• Technique : réaliser l'exclusion mutuelle sur les ressources lorsque plusieurs processus demandent l'accès à une ressource *non partageable*.

➥ Dans l'exemple des gardiens, la variable N est cette ressource (de même pour une imprimante partagée où on risque d'imprimer des fichiers entrelacés !).

Problème de section critique (suite)

Principe : on transforme le code de chaque processus en :

Répéter % code exécuté en séquence
< Section restante (SR) >
< Section **d'entrée** >
< SC >
< Section **de sortie** >
Jusqu'à faux

Remarques importantes :

• Toutes les solutions proposées concernent les sections **entrée** (SE) et **sortie** (SS).

• Rien ne permet de dire que les sections SE et SS sont exécutées de manière indivisible (2 processus peuvent être en même temps dans leur SE / SS).

Problème de section critique (suite)

Un autre exemple :
- imprimante et *race condition* (problème d'accès concurrent) :
- Pour imprimer, un processus place le nom d'un fichier dans un répertoire spécial (SPOOL).
- Un autre processus (démon d'impression) vérifie périodiquement ce répertoire et imprime les fichiers.
- Le répertoire Spool a un certain nombre de places (même infini) numérotés de 1 à n.
- Il y a 2 variables : *in* pour désigner la première place libre et *out* pour désigner le prochain fichier à imprimer.
✓Ces deux variables sont accessibles à tout processus.

Loi de *Murphy* et problème d'accès concurrent aux variables *in* et *out* :
- P1 lit *in*, il est coupé, *p*2 lit la même valeur et les deux processus placent leur fichier au même endroit.
- L'un des deux fichiers ne sera jamais imprimé .

Solutions au problème de SC : solutions logicielles et matérielles.

Solutions logicielles*

- Une première solution naïve.
- Deux processus $P1$ et $P2$ (pour plus de processus, solution plus compliquée).
  - P1 et P2 se partagent une variable $Tour \in [1,2]$.
  - Chaque processus consulte $Tour$; Pi entre dans SC si $Tour = i$; en sortant, il donne le tour à l'autre processus.

Programme principal :

```
Tour := 1
Parbegin      P1 ; P2      Parend
```

Code de $P1$ (symétrique au code de $P2$) :

```
Répéter
        < SR1 >
        Tantque Tour=2 faire rien finTq
        < SC1 >
        Tour := 2 ;
Jusqu'à faux ;
```

Solutions logicielles* (suite)

• Les problèmes de cet algorithme :
- P1 dans son SC, Tour=1, P2 ne peut pas entrer dans SC ($\Rightarrow$ exclusion mutuelle souhaitée).
- Mais, cette solution impose une **alternance** ; e.g. l'algorithme permet le comportement partiel

$$SE1\ SC1\ SS1\ SR1\ SE2\ SC2\ SS2\ SR2$$

- De plus, si l'un des processus s'arrête, l'autre processus ne peut s'exécuter au qu'une fois .

Solutions logicielles* (suite)

Conditions de comportement correct

1. Deux processus ne peuvent pas être en même temps dans leur SC (**mutex**)
2. Aucune hypothèse sur la vitesse relatives et sur le nombre de processus
3. Aucun processus suspendu à l'extérieur de sa SC ne doit bloquer les autres ou les empêcher d'entrer dans leur SC ($\Rightarrow$ absence de **blocage** $\equiv$ condition de **progression**)
4. Aucun processus ne doit attendre longtemps avant d'entrer en SC (attente **bornée**)

$\Longrightarrow$ Dans l'exemple ci-dessus, la condition de **progression** n'est pas respectée.

• **Une deuxième solution**

- $P1$ et $P2$ se partagent deux variables booléennes $D1$ et $D2$ initialisées à faux.
- Elles correspondent aux demandes de P1 et P2 pour entrer en SC.

• Code de $P1$ (symétrique au code de $P2$) :

Solutions logicielles* (suite)

```
Répéter
        < SR1 >
        D1 := vrai
        Tantque D2 faire rien finTq
        < SC1 >
        D1 := faux ;
Jusqu'à faux ;
```

• Le problème de cet algorithme : possibilité de blocage (Morphy) :
si les deux processus ont fait $Di := vrai$, chaque processus attendra dans sa SR.
• Une troisième solution avec attente bornée (équité) :
équité : si un processus est en attente de SC, borner le nombre de fois où les autres entrent dans la SC.
⇒ Algorithme de **Peterson**
Cet algorithme garantie les 4 conditions : mutex, absence de blocage, progression, attente bornée.

Solutions logicielles* (suite)

• *P*1 et *P*2 se partagent une variable *Tour* (init 1) et deux variables booléennes *D*1 et *D*2 initialisées à faux.
• Code de *P*1 (symétrique au code de *P*2) :

```
Répéter
        < SR1 >
        D1 := vrai
        Tour := 2
        Tantque D2 && Tour=2 faire rien finTq
        < SC1 >
        D1 := faux ;
Jusqu'à faux ;
```

⇒ Le seul cas où *P*1 ne peut pas entrer en SC1 se produit lorsque (idem pour *P*2) :
- P2 a demandé à entrer dans SC2 (D2=vrai)
- C'est le tour de P2 (Tour=2)
- Si cette écriture vient de P1, c'est P2 qui entre sinon P2 sera bloqué et P1 entre.

Rappel : les variables auront la valeur de la dernière écriture.

Solutions logicielles* (suite)

• Chaque processus qui demande l'entrée en SC attendra au plus un tour.
• On peut établir la table de vérité de cet algorithme et constater son fonctionnement.
• La solution logicielle est utilisée dans les multi processeurs (un peu plus compliqué). Son adaptation est plus compliquée dans les systèmes répartis (en général sans mémoire commune mais avec des messages) .
• Autres solutions logicielles : algorithmes de "Decker" et de "Eisenberg", "Dijkstra", etc.
• Le problème **d'attente active** de tous ces algorithmes : voir plus loin.

Solutions matérielles

• Mieux adaptées aux systèmes mono-processeurs.

1- Masquage/désarmément d'Interruptions :
- Le moyen le plus simple :
on masque les interruptions pendant la SC (y compris l'interruption Horloge).
- Inconvénient : cette solution n'est pas utilisable dans les multi processeurs.
➙ Une solution est alors de spécialiser un des processeurs.

• Problèmes de masquage :
Danger de laisser un utilisateur masquer tout. S'il oublie de démasquer ?
➥ Si un processus prioritaire doit passer ?

• Remarque : le principe de *masquage total* est utilisé par le *noyau* pour e.g. mettre à jour la table des processus (PCB) et pour la gestion des "Prêts".

• ... Les autres solutions matérielles :
2- TAS .
3- Verrouillage et réquisition du bus.

Solutions matérielles (suite)

TAS : Test and Set

- Avec verrouillage du bus possible .
- Tas : instruction élémentaire (insécable, atomique, exécutée en 1 cycle) pour lire et écrire un mot mémoire.
- Deux opérandes : un registre *R* et un mot mémoire *B* .
- On copie *B* dans *R* et on place 1 dans *B*.

```
Procédure TAS (var a,b : entier)
        Début
                a := b ; b :=1 ;
        Fin TAS ;
```

TAS résout le problème de SC de la manière suivante :

- Les processus partagent une variable *Verrou*.
- Chaque processus *Pi* possède une variable locale *Testi*.

Solutions matérielles (suite)

• Code de *Pi* :

```
Testi : entier ;
Répéter
        <SR>
        TAS(testi, Verrou) ;
        Tantque Testi=1 faire
                TAS(testi, Verrou) ;
        finTq
        < SC >
        Verrou := 0 ;
Jusqu'à Faux ;
```

• Le premier processus qui exécute TAS trouve $Verrou = 0$ et entre en SC. Les autres trouveront $Verrou = 1$ et attendent.
• Cette solution garantie *l'exclusion mutuelle* mais pas**l'attente bornée** (assurée par d'autres moyens).
• IBM360 était la première machine à proposer TAS (TST).

Solutions matérielles (suite)

• Par la suite, Motorola a implanté TAS puis Intel iAPX86 a proposé XCHG qui échange le contenu d'un registre avec un mot mémoire.
• Code de XCHG :

$$Reg = 1; \quad XCHG \;\; Reg, Verrou$$

➥ $Reg := Verrou$ et $Verrou := Reg = 1 \quad \simeq$ TAS.

• On peut se servir de TAS dans les multiprocesseurs mais *Verrou* est en mémoire

➧ différents processus mettent Verrou à 1 mais d'autres (sortie de SC) le mettent à 0.

➥ Il faut une gestion de la variable Verrou (cf. Motorola 680xx).
➥ Du fait d'affecter 0 ou 1, 68xxx garanti une excéution en un seul cycl.

Solutions matérielles (suite)

Les inconvénients de TAS

• **Problème1** : Attente active (on tourne en rond) et on utilise l'UC.

• **Problème2** : Inversion des priorités (les deux problèmes peuvent être liés) :

⏵ Un processus $P1$ de faible priorité prend l'accès ; $P2$ avec une forte priorité prend l'UC $\Rightarrow$ on ne s'en sort plus ! !

⏵ Blocage d'un type particulier difficile à détecter/éviter (garder une trace de toute instruction coûte cher).

• Pour le Problème-1 (pour Problème2, voir plus loin) :

- Remplacer l'attente active par une attente passive.
- Le Verrou sera libéré et on réveille un processus en attente

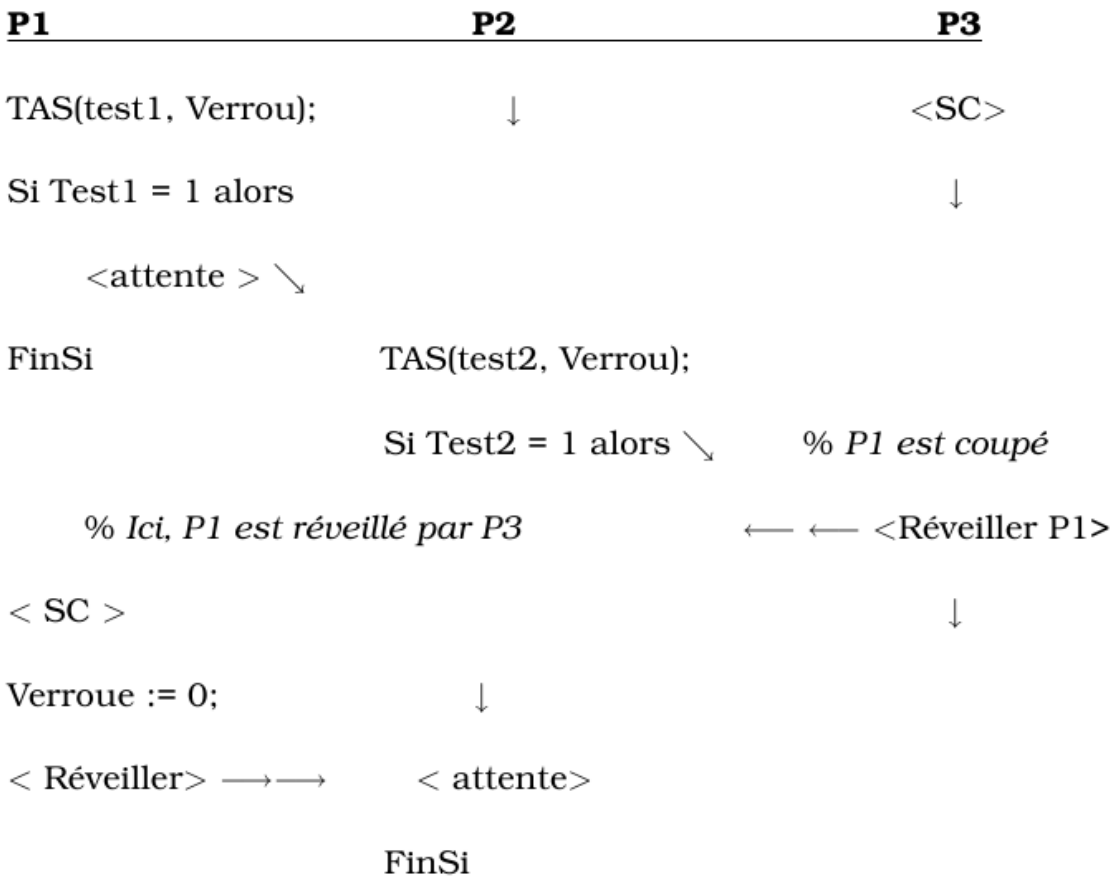
Solutions matérielles (suite)

Suite : libérer le Verrou et réveiller un processus en attente :

```
Testi : entier ;
Répéter
        <SR>
        TAS(testi, Verrou) ;
        Si Testi=1
        alors <se mettre en attente>
        FinSi ;
        < SC >
        S'il existe : < Réveiller un Processus >
        Sinon Verrou := 0 ;
        Finsi
Jusqu'à Faux ;
```

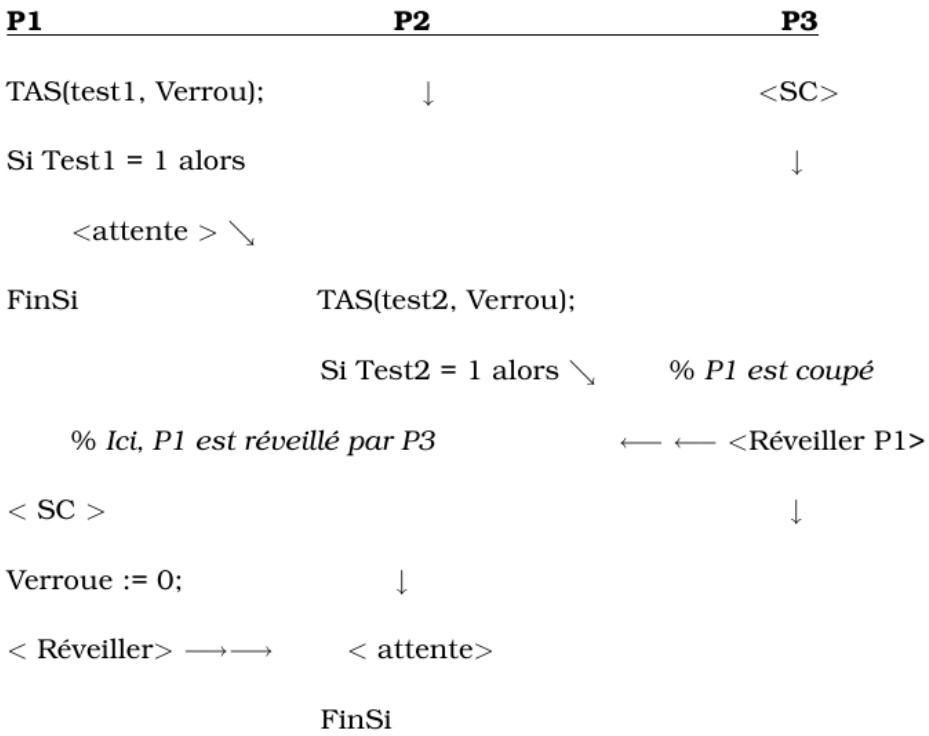
⇒ Mais cette solution peut poser problème :

Solutions matérielles (suite)



Solutions matérielles (suite)

- SC libérée mais un processus (P2) attend
- Attente perpétuelle pour *P*2 si P1 reste dans sa SR et P3 ne demande pas SC ! !
- Le problème vient du fait que <Réveiller> s'exécute trop tôt.



Solutions matérielles (suite)

- Pour régler ces problème, il faut garder la trace de plusieurs réveils.
- C'est l'objet d'autres mécanismes (e.g. *Sémaphores*).
- D'autres mécanismes :
  - SLEEP wait, attendre et WAKEUP (signaler, réveiller).

- SLEEP : suspendre l'appelant (par un SVC) en attendant un réveil par un autre.

- Voir des exemples plus loin.

Sémaphores

• Un sémaphore S est un entier auquel on n'accède que par deux primitives P et V.
Version 1 : attente active sans file d'attente :

P(S) : Tanque S $\preceq$ 0 faire **rien** FinTq ;
S := S -1 ;

V(S) : S := S +1 ;

N.B. : parfois, il y a une opération *init(S,val)* où val=nombre de ressources (=1 pour SC dans les problèmes mutex simples).

Important : dans les opération P et V, le test et la modification de S sont exécutés de manière indivisible (à l'aide d'un TAS par exemple).
⏵ Mais les procédures P et V n'ont pas besoin d'être exécutées en mutex (c'est pourtant souvent le cas car plus simple à traiter).

Sémaphores (suite)

• Pour implanter P et V dans un mono processeur, on peut désarmer les interruptions.

⏵ Ce qui ne suffit pas dans un multi processeur / réparti sauf si les Interruptions sont gérées par l'un d'entre eux.

• Dans un multi processeur avec la mémoire commune, on utilise un TAS, ou bien on verrouille le bus pendant l'accès (ou même pendant P et V dont les codes sont courts).

• Pour assurer la mutex pour N processus, on utilise un mutex init 1.

Exemple (code de *Pi*)

Répéter
    <SR>
    P(mutex)
    <SC>
    V(mutex)
Jusqu'à faux

Réalisation des Sémaphores

- Explication et fonctionnement.
- La logique des Sémaphores (le principe) est implantée de diverses manières.
- Problème d'attente active réglée avec wait / signal.

Exemple

Type Sémaphore : enregistrement
val : entier ;
File-d-attente : File de PCB
Fin Enregistrement

- Initialisations :

Exemple

Sémaphore S ;
S.File-d-attent := vide ;
S.val := ...

N.B. : dans une approche objet, les initialisations sont mieux cachées.

Réalisation des Sémaphores (suite)

Codes de P et V adapté à la déclarartion :

Exemple

P(S) : Si S.val $\preceq$ 0 alors
<Ajouter PCB de l'appelant à S.File-d-attente>
<mettre l'appelant en attente>
Sinon S.val := S.val-1 ;
Finsi

Exemple

V(S) : Si S.File-d-attente $\neq$ vide alors
<Choisir et enlever un PCB de S.File-d-attente>
< mettre ce processus dans Prêts>
Sinon S.val := S.val+1 ;
Finsi

- Le choix du PCB dans la File d'attente

Réalisation des Sémaphores (suite)

→ problème de famine ⇒ gestion Fifo.

N.B. : Certains OS proposent des sémaphores variés :

- OS2 possède deux types de sémaphores : binaires et système avec des files séparées
- On peut attendre sur plusieurs sémaphores.
- Unix a également plusieurs types de sémaphores

Autres réalisations de Sémaphore

Avec masquage / démasquage (attente active) :

Exemple (P)

P(S) : *masquer*
Tantque S.val ⩽ 0 faire
démasquer ; masquer ;
FinTq
S.val := S.val -1 ;
démasquer

Exemple (V)

V(S) : *masquer*
S.val := S.val + 1 ;
démasquer

- *De facto*, P et V sont ici insécables.

Autres réalisations de Sémaphore (suite)

Avec TAS (attente passive) :

Exemple (P et V)

P(S) : % accès à S.val en mutex à l'aide du TAS
TAS(Test, Verrou) ;
Tantque Test = 0 faire TAS(Test, Verrou) ; FinTq ;
Si S.val ⩽ 0 alors Verrou=0 ;
<Attendre dans S.File-d-attente >
Sinon S.val := S.val -1 ; Verrou=0 ;
FinSi

V(S) : % Même code pour accéder à S.val par un TAS
Si S.File-d-attent ≠ vide alors < Réveiller un processus, PCB → Prêts >
Sinon S.val := S.val + 1 ;
Finsi
Verrou=0 ;

⇒ Une attente active pour avoir le TAS.

Autres réalisations de Sémaphore (suite)

Une autre version simplifiée :
Il nous faut accéder à P et V en mutex.

```
P(S) :   S.val := S.val -1 ;
         Si S.val ≺ 0 alors
                        <Attendre dans S.File-d-attente >
         FinSi
```

```
V(S) :   S.val := S.val +1 ;
         Si S.val ⪯ 0 alors % ou si File ≠ vide
                        < Réveiller un processus, PCB → Prêts >
         Finsi
```

• Pour garantir l'indivisibilité, on peut placer *solo()* et *Tuti()* au début et à la fin de ces codes.
- **solo()** : seul : masquage ou prise de Verrou ;
- **tuti()** : tous : l'inverse

Utilisation des sémaphores

- Pour les problèmes de synchronisation
- Pour les problèmes d'exclusion mutuelle

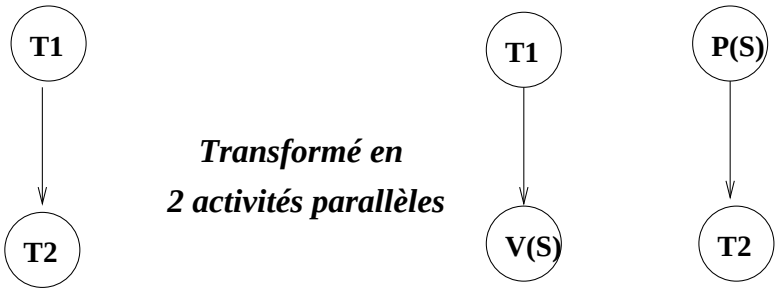
Problèmes de synchronisation

- synchronisation et contraintes de précédence

Exemple 1

- Soient deux tâches $T1$ et $T2$ avec la contrainte $T1 < T2$.
- Elles peuvent se présenter dans n'importe quel ordre.

```
S : Sémaphore init 0 ;
parbegin
        begin T1 ; V(S) ; end ;
        begin P(S) ; T2 ; end ;
parend
```

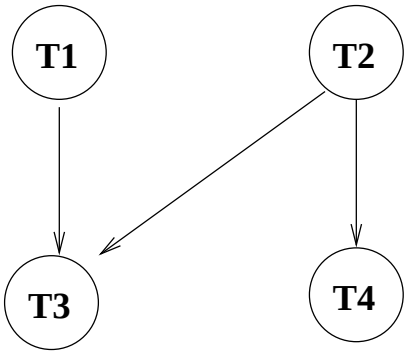


$\Rightarrow$ Le graphe de précédences des tâches est remplacé par un graphe équivalent.

Utilisation des sémaphores (suite)

Exemple 2

- Soient le graphe de précédences suivant :



- Une solution qui minimise le nombre de sémaphores :

```
S : Sémaphore init 0 ;
parbegin
                begin T1 ; P(S) ; T3 ; end ;
                begin T2 ; V(S) ; T4 ; end ;
parend
```

Utilisation des sémaphores (suite)

• Les sémaphores servent à traiter des problèmes plus complexes que la contrainte de précédence.

Problèmes d'exclusion mutuelle

• Exemple de distribution de ressources :

- Soit K exemplaires d'une ressource ($K \geqslant 2$)
- Pour un (parmi N) processus, l'accès à X exemplaires de la resource est une SC.
- Pour X=1, on peut avoir au plus K processus en SC.
- Une solution (le code de chaque processus) :

```
S : Sémaphore init K ;
Répéter
                <SR>
                P(x) ;
                <SC>
                V(x) ;
jusqu'à faux ;
```

Utilisation des sémaphores (suite)

• Sémaphores **privés** : les P et V ne sont forcément faits par le même processus (cf. les exemples de synchronisations précédents)

• Inconvénient des sémaphores :
l'utilisateur doit bien placer P et V.
→ Risque de blocage en particulier lorsque le code est compliqué avec des sémaphores privés.

➥ Outil de plus haut niveau : **Moniteurs.**

Exercices Sémaphores

1- producteur consommateur
    1-1 : producteur consommateur simple synchronoisés
    1-2 : tampon borné
    1-3 : tampon non borné
    1-4 : N producteur / M consommateurs
    1-5 : contraintes de consommation : tout doit être consommé (par tous)
2- Lecteur Rédacteur
    2-1 : Un / N lecteurs , un /M rédateurs
    2-2 : Priorité aux lecteurs / rédacteurs
3- Les 5 philosophes
4- La disctibution de ressources (billes)
5- Coiffeur
6- etc.

Suite IPC : Moniteurs

Motivation : palier les problèmes des Sémaphores et dispenser l'utilisateur des détails de synchronisation et d'exclusion mutuelle (e.g. P et V des sémaphores).
• Mécanisme proposé par Brinvh-Hansen, Dijkstra et Hoare

Un Moniteur : une strcuture contenant des variables (d'état) partagées et du code de manipulation de ces variables.
$\Rightarrow$ existe dans certains langages comme Concurrent Pascal sous forme d'une strcture de données.

La strcuture Moniteur contient :
- des variables partagées uniquement accessibles dans le Moniteur
- des procédures / fonctions internes seules à pouvoir manipuler les variables
- ces fonction et leur paramètre = l'interface avec l'extérieur.
- une partie initialisation des variables

Suite IPC : Moniteurs (suite)

- Exemple des gardiens :

```
Type nb_clients = Moniteur
var N : entier ;
Proc incrémenter
begin
                    N++ ;
end ;
begin % initialisation
                    N := 0 ;
end ;
```

⇒ Les processus *compteur*1 et *compteur*2 (les gardiens) :

```
Tantque <le magasin est ouvert> faire
        Si <entrée> alors nb_clients.incrémenter ; Finsi ;
FinTq ;
```

Suite IPC : Moniteurs (suite)

Avantages des Moniteurs :

- La SC est mise dans un Moniteur
- Les SC sont transformées en procédures / fonctions de Moniteur au lieu d'être dispersées
- La gestion de la SC n'est plus à la charge de l'utilisateur
- Le Moniteur garantie qu'au plus un processus à la fois peut accéder à cette structure.
- Un sémaphore binaire (mutex) suffira pour garantir cette exclusion mutuelle.
- Le Moniteur tout entier est implanté comme une SC.

Suite IPC : Moniteurs (suite)

Fonctionnement :
- Un processus appelle une routine d'un Moniteur
- Il y entre si le Moniteur est libre
- Sinon, son PCB est mis en file d'attente associée à ce Moniteur (e.g. file du sémaphore).
- Quand le Moniteur deviendra libre, un processus est sorti de cette file en entre dans le Moniteur.

- Ce fonctionnement assure une exclusion mutuelle.
- Pour le synchronisation, les Moniteurs utilisent des **conditions**

Suite IPC : Moniteurs (suite)

Variables de type condition :
- Donnent aux Moniteurs la capacité de synchronisation des sémaphores.
- Déclararion dans le Moniteur sous forme *X : condition*

- X désigne une file d'attente (dans le Moniteur) manipulée par *wait* et *signal*.
  - une condition n'est pas un compteur (comme les sémaphores) :
  - on ne mémorise pas les signaux pour un traitement ultérieur.

- un processus (dans le Moniteur) exécute *wait* sur une condition, se met en attente de la réalisation de la condition → le Moniteur devient libre.

- l'exécution d'un *signal* sur une condition par un processus déclenche le réveil d'un processus (s'il y en a) en attente dans la file de cette condition .

Suite IPC : Moniteurs (suite)

L'implantation de *signal* doit régler deux problèmes :

1- le choix du processus à réveiller et à sortir de l'attente

2- le choix du processus qui doit continuer dans le Moniteur (le signaluer ou le réveillé ?)

➞ risque d'avoir deux processus dans le Moniteur !

➞ - Il y a aussi ceux qui demandent le Moniteur ! (3 processus)

• Plusieurs solutions (les files gérées en FIFO) :

- **Hoare** : priorité au signaleur : on attend qu'il quitte le Moniteur (ou qu'il se mette en attente sur une autre condition) pour passer la main au réveillé.
- **Concurrent Pascal , Hansen** : le signaleur quitte immédiatement le Moniteur (le réveille continue).

➞ dans le code, il ne faut rien mettre après *signal*.

Suite IPC : Moniteurs (suite)

Exemple : N processus, une ressource

```
Type allocateur = Moniteur
var occupé : bool ;
var libre : condition ;
Proc acquisition
begin
                    Si occupé alors libre.wait Finsi ;
                    Occupé := true ;
end ;
Proc libération
begin
                    Occupé := true ;
                    libre.signal ; % on peut sortir de Moniteur.
end ;
Initialisations
begin          Occupé := false ;      end ;
```

$\Rightarrow$ On remarque l'analogie entre les deux procédures et P et V d'un sémaphore.

Suite IPC : Moniteurs (suite)

Exemple : lecteurs rédacteur :
- N processus : 1 rédacteur, N-1 lecteurs (critiques).
- une masion d'édition : le Moniteur
- Les lecteurs attendent le livre.
- Il faut garantir qu'ils auront tous le livre.

```
Type Edition = Moniteur
var livre_dispo : bool
var livre : string ;
var dépot_fait : condition ;
Proc Dépot(l : string)
begin
        livre_dispo := true ; livre := l ; % dépôt physique du livre
        depot_fait.signal ;
end ;
fonction Retrait () → string
begin
        Si not livre_dispo alors
                depot_fait.attendre ;
                depot_fait.signaler ; % signal en chaîne
        Finsi
        return livre ;
end ;
begin           livre_dispo := false ;          end ;
```

Suite IPC : Moniteurs (suite)

Code de l'écrivain :	**Code d'un lecteur :**
livre : string ;	livre : string ;
<élaborer le livre>	livre := Edition.Retrait() ;
Edition.Depot(livre) ;	<Critiquer le livre>

$\Rightarrow$ discuter des détails.

Suite IPC : Moniteurs (suite)

Implantation des Moniteurs

- Avec des sémaphores (voir BEs SE)
- etc.

Inconvénients des Moniteurs

- peu de langages de support
- utiles dans les systèmes à mémoire partagée (pas distribués)

➥ d'où la solution : passage de messages.

Messages

- Utiles dans les systèmes distribués pour régler les problèmes de synchronisation et d'exclusion mutuelle.
- Existent aussi dans les systèmes centralisés.
- Primitives (appels SVC) :

  - **send(dest, message)**
    bloquant / non bloquant
  - **receive(source, message)**
    Si source=Any,on reçoit de tous
    bloquant / non bloquant

Propriétés des systèmes d'échange de messages :

- aussi intéressant que les mécanismes Sémaphore / Moniteur
- les messages peuvent se perdre !
  $\Rightarrow$ acquittement dans un délai fixe, sinon, on recommence.
- si l'acquittement se perd ! on risque d'emettre 2 fois
  $\Rightarrow$ numéroter les messages

Messages (suite)

- savoir nommer les processus dans *send* et *receive* ?
  $\Rightarrow$ format : **processus@machine.domaine**
- Authentification : à qui a-t-on réellement affaire :
  - comment authentifier le (vrai) serveur de fichiers
  - comment le serveur sait que c'est un (vrai) client
  $\Rightarrow$ codage des messages avec clefs (connues des utilisateurs). $\rightarrow$ cf. Cryptographie
- problème de performances : c'est plus lent qu'un Sémaphore / Moniteur
  -dépend de la vitesse de traitement et de transmission
  -on utilise des registres pour traiter des messages courts

../..

Messages (suite)

• Un exemple :
Producteur - consommateur avec une Boîte de taille N Send non bloquant, receive bloquant

Producteur :

```
M : message
while (true)
        <produire DATA>
        receive(Consommateur, message vide M)
        <créer message M avec DATA >
        send(Consommateur, M)
end while
```

Consommateur :

```
M : message := vide ;
for i : 1..N
        send(Producteur, M)
while (true)
        receive(Producteur, message M)
        <retirer Data de M>
        send(Producteur, message vide M) % débloque le prod
end while
```

Messages (suite)

Remarques :
• La communication inter processus d'Unix peut se faire aussi avec des tubes (pipes) de communication

• La différence avec une boîtes aux lettres (messages) : les messages ne sont pas délimités dans un tube.
⇒ une lecture de tube peut donner K messages à la fois (qu'il faudra traiter)
⇒ on peut lire des messages de taille fixe .

• On peut démonter l'équivalence entre sémaphores, Moniteurs et les messages.
⇒ chacun peut être implanté à l'aide des autres
⇒ Voir aussi BE-SE : boîte aux lettres.

Ordonnancement de l'UC

- But : maintenir un taux d'utilisation élevé de l'UC
- Idée première : recouvrement des E/S
- Quelques mesures :
  - le **taux** d'utilisation de l'UC :
    - théorique : 0 à 100%
    - pratique : 40 à 95%
  - le **débit**
    - nombre moyen de programmes utilisteurs traités par unité de temps
    - le taux augment le débit
    - le débit ne tient pas compte de la taille des programmes
  - Optimisation de ces valeurs revient à optimiser :
    - temps de **traitement moyen** d'un système de tâches :
    $\rightarrow$ la moyenne des intervalles séparant la soumission et la fin de la tâche.
    - temps de **traitement total** d'un ensemble de processus donné
    - temps de **réponse maxi** : soumission - réponse à une requête (important dans les systèmes T.R.).

Ordonnancement de l'UC (suite)

• Pour optimiser ces temps, il ne suffit pas de recouvrir les E/S par des calculs.
⇒ l'odonnanceur doit choisir le processus qui améliore les critères.
• Définitions : pour une tâche

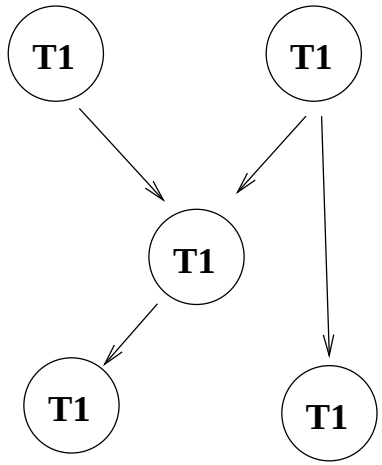
- T_i : la tâche ;
- τi : sa durée d'exécution ;
- ti : sa date d'arrivée dans la file des prêts
- **assignation** : la description de l'exécution des tâches dans le système (mono / multi processeur) construite par l'odonnanceur
  → un *ordonnancement*

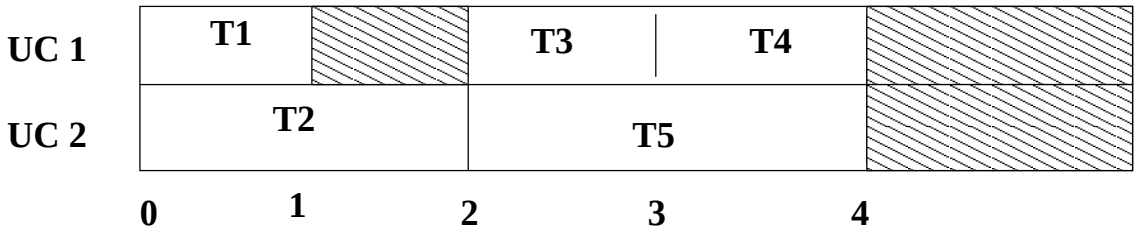
T_i	τi	ti
$T1$	1	0
$T2$	2	0
$T3$	1	0
$T4$	1	0
$T5$	2	0

Ordonnancement de l'UC (suite)

- Exemple : 5 tâches, 2 processeurs

	ti	τi
T1	**1**	**0**
T2	**2**	**0**
T3	**1**	**0**
T1	**1**	**0**
T5	**2**	**0**





Ordonnancement de l'UC (suite)

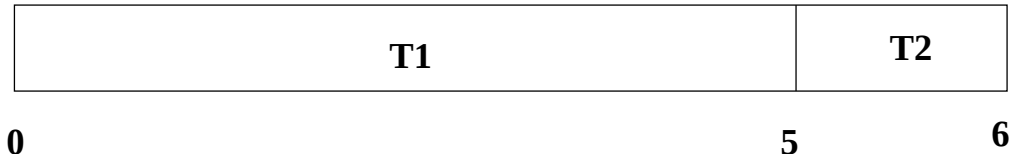
• Diagramme de **Gantt** : une représentation d'une assignation.
Remarques :
- On ne représente pas les temps de commutations (très faible).
- τi peut représenter le temps exact d'exécution d'une tâche si aucune autre n'existe dans le système.
- Ctte valeur est incalculable par avance ; on s'en sert a posteriori pour analyser un système.
- Le τi pris en compte est une borne sup de la durée (utile dans les systèmes TR)
- Le calcul d'une estimation de τi est possible (V. biblio).
- Les algorithmes d'ordonnancement pratiqués sont classés par :
avec (sans) réquisition, avec (sans) priorité, avec (sans) contrainte de précédence (et sans tenir compte des it°).
• Exemple (T1 et T2 indépendantes, sans synchro) :

T_i	τi	ti
$T1$	5	0
$T2$	1	2

Sans réquisition : $tmoyen$: $\frac{5+(6-2)}{2} = 4,5$

Ordonnancement de l'UC (suite)

L'assignation :



Rappel : *tmoyen* = la moyenne des intervalles séparant l'entrée et la fin effective d'une tâche.
- Même exemple avec réquisition : *tmoyen* : $\frac{6+(3-2)}{2} = 3,5$

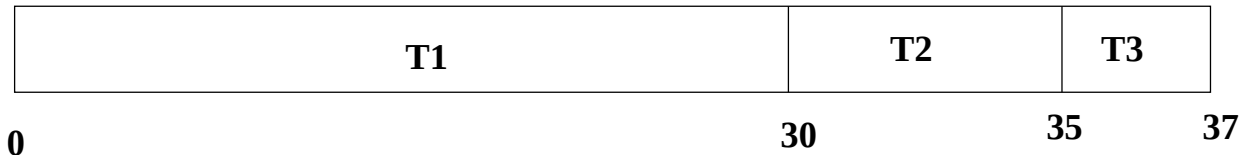
Ordonnancement sans réquisition

Dans l'ordre d'arrivée FIFO :
Méthode souple mais ne minimise pas le temps moyen.

T_i	τi	ti
$T1$	30	0
$T2$	5	ϵ
$T3$	2	2ϵ

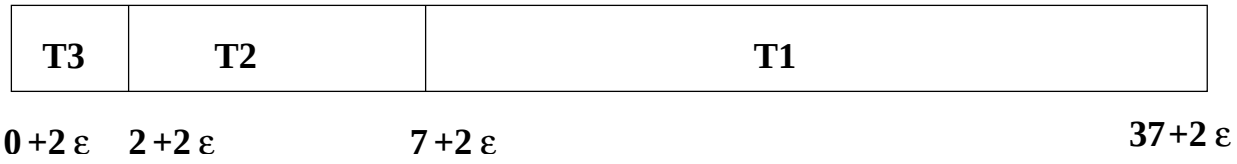
$$tmoyen : \frac{30+(35-\epsilon)+(37-2\epsilon}{3} = 34$$

- L'assignation :



La tâche $T1$ avec un temps long bloque la file car elle est arrivée en avance.
On peut faire mieux avec $tmoyen$: $\frac{2+7+37)}{3} \approx 15,5$

Ordonnancement sans réquisition (suite)

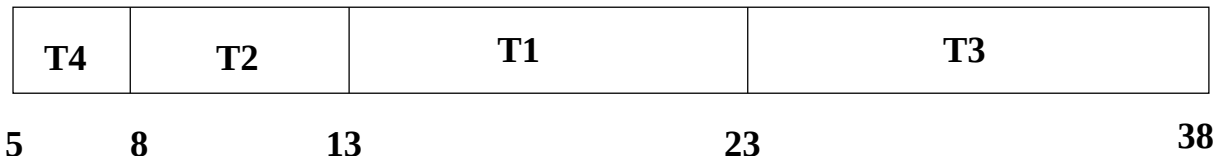


Exexmple : A l'instant **5**, l'état de la file des prêts est :

T_i	τi	ti
$T1$	10	0
$T2$	5	2
$T3$	15	3
$T4$	3	4

$$tmoyen : \frac{(8-4)+(13-2)+(23-0)+(38-3)}{4} = 18,25$$

L'assignation :



Ordonnancement sans réquisition (suite)

On peut démontrer que l'algorithme PCTE **minimise** le temps moyen pour l'ensemble des algorithmes d'ordonnancements **sans réquisition**.
$\Rightarrow$ Cette proposition n'est pas vraie si les tâches entrent au cours d'une assignation (à moins de refaire l'assignation à intervalle X donné).
PCTE est intéressant pour permettre de comparer, après coup, les perfomances des algorithmes réellement implantables, aux valeurs min qu'il est possible d'obtenir (avec des estimations).

Ordonnancement avec réquisition

- Nécessaire dans un système à temps partagé (ou TR)
- Un SE doit fournir une machine virtuelle ;
- Les tâches longues ne doivent pas bloquer les autres ;
- Question d'équité ;

Méthode Tourniquet (Round Robin) :
- On utilise des It°, commutations, Horloge ;
- On choisit un intervalle **Quantum**

> *Exemple : dans Vax/VMS, le quantum est multiple de 10ms (intervalle entre 2 tops d'horloge = 10ms) ; par défaut, quantum = 200ms*

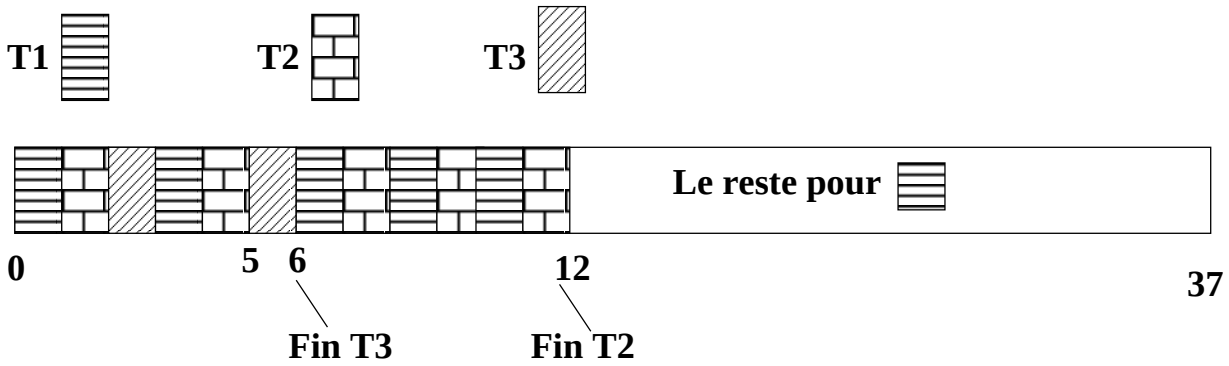
- File d'attente circulaire (ou double file) ;
- L'ordonnanceur choisit une tâche, le distributeur (dispatcheur) lance son exécution ;
- La tâche peut termine avant le quantum ($\rightarrow$ remise à 0) ;
- Sauvegarde du contexte, cumul des temps, choix du prochain, commutation ;

Ordonnancement avec réquisition (suite)

- Exemple :

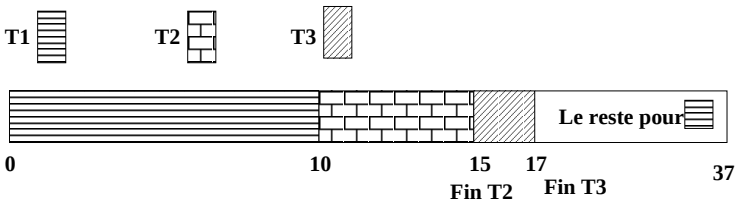
T_i	τi	ti
$T1$	30	0
$T2$	5	ϵ
$T3$	2	2ϵ

L'assignation (Quantum = 1) : tmoyen = $\frac{37+12+6-3\epsilon}{3} = 18,33$



Ordonnancement avec réquisition (suite)

L'assignation (Quantum = 10) : tmoyen = $\frac{37+15+17-3\epsilon}{3} = 23$



• Importance du Quantum :
  - si quantum trop grand : temps de réponse augmente ;
  - si quantum trop petit : trop de commutations !

  $\Rightarrow$ Une idée de CDC6000 : 10 mots d'état en mémoire

• NB : Round Robin avec une double file des prêts.

Plus court temps d'exécution restant : PCTER

• Principe : à chaque réquisition, on calcule le temps d'exécution restant et on choisit la tâche dont le TER est minimum....

Ordonnancement avec priorité

• Principe : égalité pour tous (équité)
Mais pour des raisons d'efficacité, les tâches systèmes sont plus prioritaires ;
→ d'autres devraient fournir des résultats avant certaine date.
⇒ Solution : attribution de priorités ;
⇒ 2 méthodes :
- Fixe : problème de famine si priorité faible
- Variable : recalcul des priorités selon l'avancement des exécutions.
e.g. : on peut ajouter +1 à la priorité des tâches en attente (et/ou enlever +1 de la priorité de ceux qui s'exécutent).

• Exemple OS2 : proche de VAX/VMS :
- augmentation si phase de calcul courte (avec des E/S).
→ Truc : ajout fictif d'une lecture de temps en temps !

Ordonnancement avec priorité (suite)

• Exemple VAX/VMS :
- Ordonnancement avec priorité
- Tâches normales (user) et tâches systèmes (TR)
- 32 niveaux de priorité : 0 (min) .. 15 : user, 16..32 : TR.
- Priorités modifiables
- UC réquisitionné à l'arrivée d'une tâche plus prioritaires (surtout tâche TR) ;
- Les priorités 0..15 évoluent à l'arrivée de certains événements. Chaque événement apporte une valeur (positive ou négative) de priorité (e.g. fin E/S, libération d'une ressource, fin d'une entrée/sortie depuis/vers un terminal, création d'un processus, etc.)
- L'exécution d'une tâche fait baisser sa priorité (limitée à un seuil) ;
- Round Robin pour la même classe / valeur de priorité ;

Ordonnancement avec priorité (suite)

Ordonnancement avec plusieurs files de priorité :
- Permet de réduire le temps d'attente d'une tâche sans modifier le quantum ;
- Les files ont des N° de priorité fixes ;
- Les tâches vont de file en file ;
- Le temps de l'UC peut être partagé entre les files selon les priorités ; par exemple, la file la plus prioritaire prendra 70% du temps de l'UC.
- Les files peuvent être gérées par des algos différents (adapté au type des tâches de la file) ;

Autre variante :
- Ordonner hiérarchiquement les files
- On n'exécute une file que si les file de priorité supérieure ont été traitées
→ famine sauf si les priorités sont modifiées en permanence et les tâches changent de file ;

Ordonnancement avec priorité (suite)

• Cas d'UNIX : (**préemptif** = temps partagé avec réquisition)
- Un processus - une priorité - une file
- Processus user et système
- Les processus systèmes sont toujours prioritaires
- Dans chaque classe (user, système), différents niveaux de priorité
  ➙ = files différentes
- Les priorités évoluent dynamiquement
- Quand un processus est mis en attente, sa priorité est recalculée en fonction de la raison de l'attente

  → but : diminuer les blocages et favoriser les processus qui au réveil se terminent et libèrent rapidement des ressources ;

  → un processus en attent de la fin d'une E/S avec le disque sera plus prioritaire que celui qui demande une E/S avec le disque ;

  → en plus, le demandeur a besoin d'un tampon que le premier libère ;
- A chaque it° Horloge (1s sous sysV), la priorité d'un processus prêt est augmentée en fonc. du temps passé dans sa file (changement de file poss.) ;

Ordonnancement de Disque

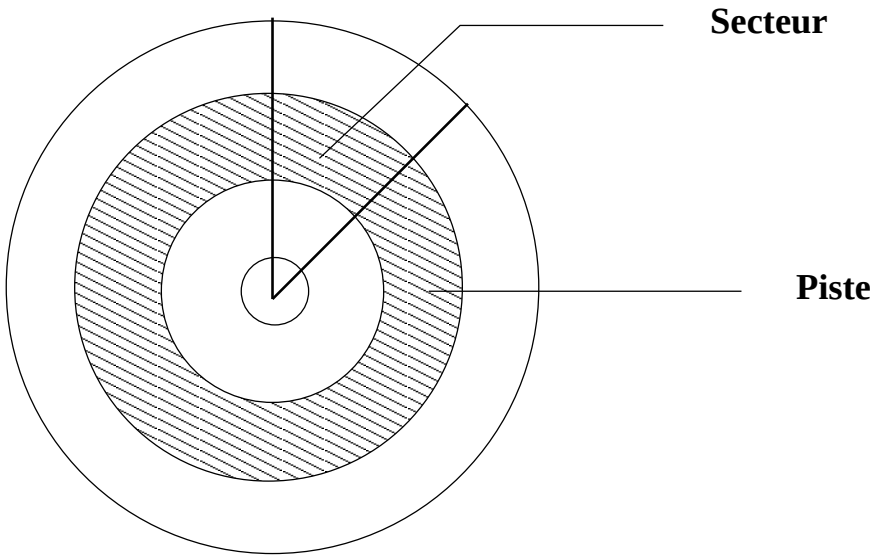
- L'ordonnancement de l'UC est crucial ;
- L'ordonnancement de disque affecte les performances du système
- Les programmes sont chargés de la mémoire Secondaire (MS = Disque) vers la mémoire Centrale (MC).
- Les échanges MS $\leftrightarrow$ MC sont très fréquents ;
- Les supports : bandes (séquentiel), Disques (direct)
- Les ordonnanceurs tiennent compte des caractéristiques ϕ du support

Caractéristiques :
- 2 types de disques : Magnétique et Optique
- Les disques optiques : grande capacité mais plus lents ;
- Disques Magnétiques : plateau circulaire (double face) + enduit magnétique ;
- Capacités : quelques Ko (floppy) aux débuts .. 500 Go et plus now !
- Les disques de stockage : plusieurs plateaux

Ordonnancement de Disque (suite)

- Chaque plateau organisé en :
  - **pistes** *(cercles concentriques)*
  - **quartiers** *d'angle fixe*
  - *un* **secteur** *: l'intersection de piste et de quartier*
  - *un* **cluster** *: un ensemble de secteurs*
  - *parfois (sur PC), 1 bloc (=unité de transfert) correspond à un secteur (angle du quartier adapté)*
  - *Un* **Cylindre** *: ensemble de pistes sur plusieurs faces qui peuvent être atteints à un instant donné ;*

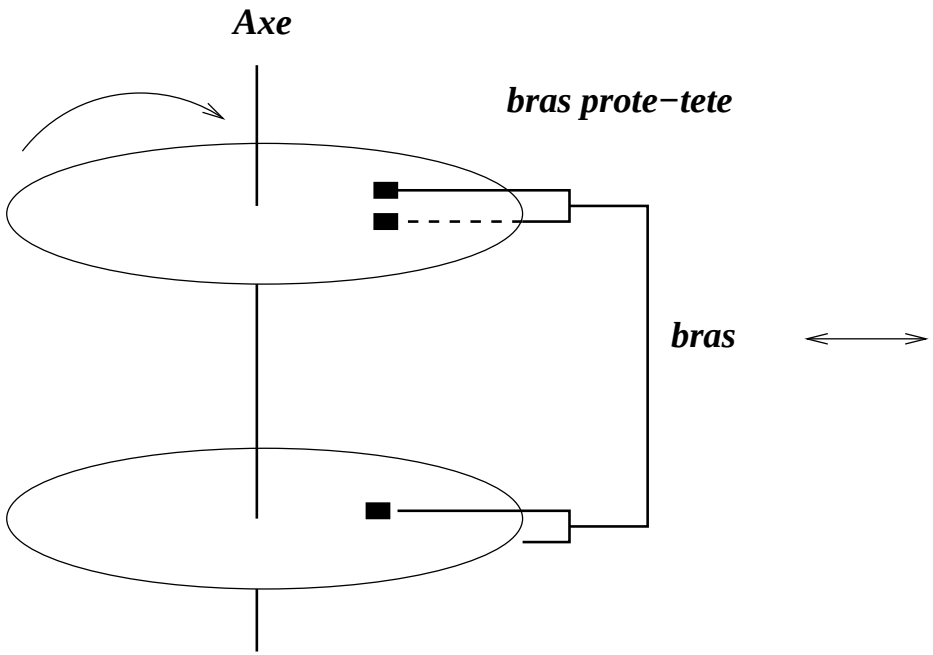
Ordonnancement de Disque (suite)



- Le nombre de pistes dépend de la capacité
- La division en pistes faite lors du formatage ;
- Le formatage définit aussi les *octets utiles* (user) et *octets de contrôle* (Fat, n° de secteur , bit de parité, etc.)
- Un bloc/secteur = unité de transfert MC $\leftrightarrow$ MS

Ordonnancement de Disque (suite)

• **Accès**
- bras, bras porte tête, une tête par face, mouvement horizontal du bras, moteur ; ...



Ordonnancement de Disque (suite)

- Vitesses de rotation actuelles : 3600 à 7200 tours/min (=120 tours/sec !) ;
- Sur un disque souple (floppy), le film magnétique peut supporter le contact de la tête ;
- Sur le disque dur : la tête est à quelques microns de la surface sur un coussin d'air provoqué par la rotation ; contact = catastrophe !
- Matériel associé à un disque :
  - *un* **dispositif de R/W** *pour exécuter les mouvements mécaniques*
  - *un* **contrôleur**
- Le contrôleur (IDE sur PC) peut gérer plusieurs disques ;
- Autre contrôleur : SCSI ;
- L'adresse d'un secteur (4 éléments) :
  - *N° du contrôleur ;*
  - *N° Cylindre (ou piste)*
  - *N° surface*
  - *N° du disque*

Ordonnancement de Disque (suite)

Eléments d'ordonnancements :

- Plusieurs étapes pour faire une E/S :
  1. attente dans la file du dispositif ;
  2. déplacement du bras sur le bon cylindre/piste ;
     → **temps de recherche**.
     Certains algos regroupent les requêtes dans d'autres files (→ attente supplémentaire)
  3. la rotation amène le bon secteur sous la tête
     → **temps de latence**.
  4. transfert depuis/vers MC : → **temps de transfert**

Ordonnancement de Disque (suite)

- Le temps *nominal* (pub !) = la somme de ces 3 temps
$\Rightarrow$ Il ne tient pas compte pas compte des temps d'attente dans les files.
- Le temps de transfert (MC $\leftrightarrow$ MS) est indépendant de l'algorithme d'ordonnancement.
- Le temps de recherche (bras $\rightarrow$ cylindre) est + élevé et les algos d'ordonnancement tentent de les minimiser
- Le temps de recherche est proportionnel à la différence entre 2 pistes $|i - j|$ dans 2 requêtes successives

Algorithmes d'ordonnancement

Algorithmes implantés dans bios/controleur (chipset) ; le driver (pilote) est propre au périphérique

Ordonnancement dans l'ordre d'arrivée (FIFO)

- Simple, équitable mais très peu performant ;
- Exemple :

  - *Un disque avec 20 pistes (0 .. 19)*
  - *La tête sur la piste 14*
  - *La file d'attente selon les N° de pistes contenant 17, 18, 4, 11, 2, 12*
  - *La tête fait : 14 → 17 → 18 ...*
  - *Le* **déplacement** *total (nb. pistes) de la tête :*

  d=(17-14)+(18-17)+(18-4)+(11-4)+(11-2)+(12-2)= 44

  • *Idée : regrouper les requêtes selon les pistes proches → on traite 4, 2, 11, 12, ... → d=30*

  Ou mieux : 2, 4, 11, 12, ...

Algorithmes d'ordonnancement (suite)

Ordonnancement suivant le plus court temps de recherche (PCTR)
- Déplacement bras → cylindre, proportionnel à $|i - j|$
- Une seule file pour les requêtes
- Requêtes regroupées par les pistes proches
- La prochaine requête traitée est celle qui minimise le déplacement de la tête ;
- La file est organisée en permanance

→ problème de **famine** si des requêtes concernant des pistes éloignées restent en attente

Ordonnancement par Balayage :
- Evite le problème de famine précédent
- Adapté aux accès fréquents
- On parcourt les pistes dans une direction données et on traite les requêtes rencontrées (e.g. vers l'intérieur)
- Ensuite, la tête change de direction et balaie les pistes vers l'extérieur
- Les requêtes arrivées pendant attendent le retour de la tête
- La version de base de cet algorithme : **Scan**
- Variante : **Look** : la tête ne va pas au bout si pas de requête (file circulaire)

Algorithmes d'ordonnancement (suite)

- Autres variantes : retour dans une direction en balayant ou pas
- Version **C-Scan** : file circulaire, retour à une extrémité
- Version **C-Look** : file circulaire, retour si pas de requête dans ce sens
- Autres versions : tenir ou pas compte des requêtes arrivées pendant ...

Ordonnancement selon le temps de Latence

- Rappel : secteur sous la tête
- Si les demandes sont trop fréquentes, on peut traiter des requêtes la même piste (cylindre) d'abord.
- Les files ordonnées selon les secteurs pour réduire le temps de latence

Algorithme PCTL : le plus court temps de latence
- Sélectionnér les requêtes concernant les secteurs les plus proches de le tête, compte tenu du sens de la rotation
- On associe **une file** à chaque secteur du même cylindre - La tête sur une pise (cylindre), on traite les requêtes de cette piste dans le sens de la rotation sans tenir compte de l'ordre d'arrivée
$\Rightarrow$ Beaucoup de files : une par secteur du même **cylindre**
- Stratégie facile à implanter ; performances proches de l'optimum ;

Mémoire : généralité

- Différents type = différentes stratégies
- Caractéristiques communes :

    - *Structure hiérarchique*
    - *Gestion par le SE des programmes user en MC*
    - *MC et MS (swap, mémoire virtuelle) : allocation, gestion, libération, ...*

Hiérarchies de mémoires :
- Plusieurs types de mémoire reçoivent des représentations ϕ des programmes ;
- Pour le SE, l'important est la vitesse d'accès et la capacité de stockage ;
- Rapport inverse cout / vitesse / capacité
- 2 types de mémoires : **volatile** et **permanente**

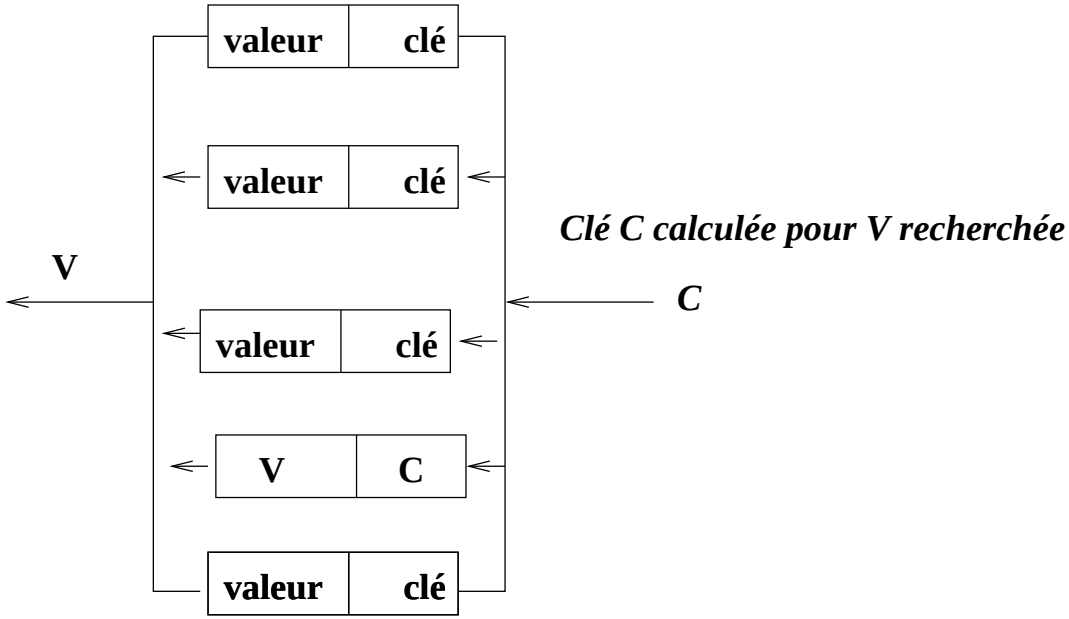
Mémoire : généralité (suite)

Mémoire Volatile
- Circuits intégrés ;
- Accès rapide ;
- Coût élevé ;
- Une ressource critique ;
- 2 types courants : **Registres** et **Mémoire Centrale** (MC) ; Voir aussi la mémoire cache ;

• Un programme est chargé en MC ; l'UC transfère les instructions de la MC vers les registres pour les exécuter ; puis range les résultats dans la MC ; - Registres d'adresse de de données ; nombre limité ; accès rapide ;

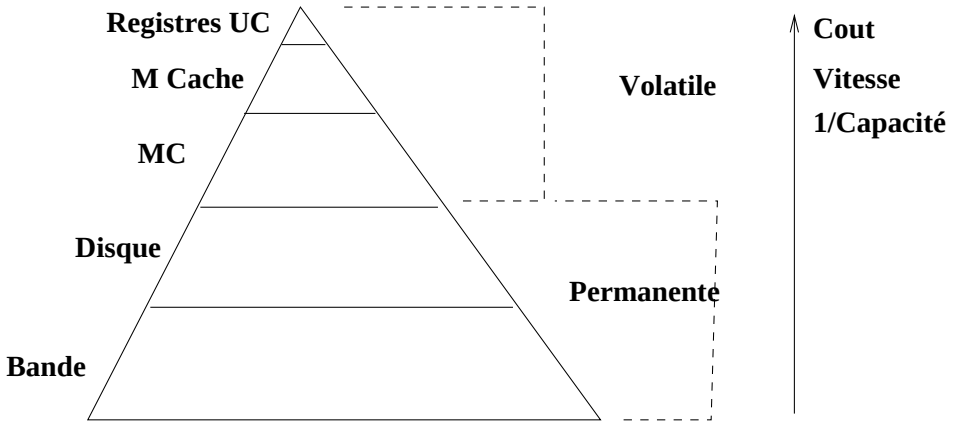
• Mémoire **cache** : volatile, accès rapide ; fonctionnement comme un tampon ;
- Souvent des registres **associatifs** (ou à **contenu adressable**) ;
- A chaque registre est associé une *clé* et une *valeur* ;
- La recherche se fait par une comparaison **simultanée** d'une clé (de la valeur recherchée) avec **toutes** les clés enregistrées (opération faite par le matériel) ;
- En cas de succès, le dispositif emet la valeur associée à la clé retrouvée ;

Mémoire : généralité (suite)



Mémoire : généralité (suite)

Mémoire Permanente (MS, de masse, auxiliaire, ...) :
- Support magnétique (disque bande)
- Mémoire flash récente (une ROM = fausse mémoire !)
- Temps d'accès elevé mais le coût faible ;



type	taille O.	T. accès S.(2002)	coût relatif
Cache	10^3.. 10^6	500 MHz=2.10^{-9}=2 ns.et -	10
MC	10^6 ..10^9	100 MHz=10^{-8}=10 ns.et -	1
Disque	10^9 ..10^{11}	1 à 10 ms. et -	10^{-2} ..10^{-3}
Bande	10^8 ..10^{12}	10 à 100 ms. et -	10^{-4}

Remarque : ces valeurs varient fréquemment.

Mémoire : généralité (suite)

Représentation Logique et ϕ d'un programme :
- Représentation Logique : procédures, modules, packages, librairies
- Représentation Physique : suite binaire
- Remarque : la MS ne fait que stocker les programmes, toute manipulation nécessite son chargement.
- Lors d'une exécution $\rightarrow$ un processus $\rightarrow$ suspension, sauve de contexte,.. : le processus (code + data) devient fichier sur disque (ou en mémoire).
- Lors de sa création puis son exécution, un programme subit plusieurs représentations
- Un processus est tantôt (re)chargé, tantôt stocké sur disque ou en mémoire ;
- A chaque étape, le SE choisit et manipule une représentation donnée ;
- A la fin, le processus est détruit et sa représentation physique disparaît
- Le programmeur a une vision Logique, le SE assure la correspondance transparente ϕ et Logique

Objectifs d'une gestionnaire de mémoire

Organisation matérielle de la mémoire ϕ et sa gestion :
Organisation de la mémoire
- La mémoire : un TDA = un ensemble de mots avec une adresse pour chacun
- LE SE structure la mémoire en Zone de taille N avec des asdresses consécutives ;
- La taille d'une zone peut être fixe ou variable
- Les zones reçoivent une représentation ϕ
- Le SE garantie **l'intégrité** des zones et les **protège** des accès non autorisés ; leur **partage** (sans duplication) - Stratégie d'allocation : attribution de zones ;
- Libération : à la fin/suspension d'un processus (PCB) ;
- La destruction d'un fichier (description en mémoire),...

Objectifs d'une gestionnaire de mémoire (suite)

- Opérateurs du TDA mémoire :

  - Mot : lecture, écriture
  - Zone : allocation, libération, extension de zone, division d'une zone, regroupement de zones, ...

- Le SE doit fournir à l'utilisateur la quantité de mémoire demandée en l'affranchissant des **limites** physiques de la MC (Mémoire virtuelle) ;
- Les techniques de gestions classées en allocation **contiguë** et **non contiguë** ;

• To be continued (from personal notes).