

SGP

Systemes d'exploitation - Gestion de processus

Master S.T.S. mention informatique, première année

Isabelle Puaut

Septembre 2011

Table des matières

| | | |
|-----------|--|-----------|
| I | Introduction aux systèmes d'exploitation | 1 |
| 1 | Architecture d'un ordinateur | 2 |
| 2 | Rôle d'un système d'exploitation | 2 |
| 2.1 | Définition d'un système d'exploitation | 2 |
| 2.2 | Structuration d'un système d'exploitation | 4 |
| 2.3 | Un exemple simple : micro-ordinateur individuel | 4 |
| 3 | Introduction aux systèmes multi-tâches | 5 |
| 3.1 | Exemple 1 : synchronisation entre processeur et périphérique | 6 |
| 3.2 | Exemple 2 : utilisation du parallélisme entre E/S et calcul | 8 |
| 3.3 | Quelques problèmes posés par le multi-tâche | 10 |
| 4 | Un peu d'histoire des systèmes d'exploitation | 11 |
| 5 | Bibliographie | 16 |
| | | |
| II | Processus et synchronisation | 17 |
| 1 | Exécution d'un programme | 18 |
| 1.1 | Processeur et trace d'exécution | 18 |
| 1.2 | Trace d'exécution dans le cas d'activités parallèles | 19 |
| 2 | Notion de processus | 20 |
| 2.1 | Définitions | 20 |
| 2.2 | Modes d'exécution des processus : niveaux de parallélisme | 20 |
| 2.3 | Relations temporelles : temps logique et temps physique | 21 |
| 2.4 | Processeur virtuel | 21 |
| 3 | Section critique - exclusion mutuelle | 22 |
| 3.1 | Exemples | 22 |
| 3.2 | Section critique et exclusion mutuelle | 24 |
| 4 | Un mécanisme de synchronisation : le sémaphore | 25 |
| 4.1 | Définition des sémaphores | 25 |
| 4.2 | Propriétés des sémaphores | 26 |
| 4.3 | Exemples de synchronisation | 26 |
| 5 | Interface d'un système d'exploitation | 28 |
| 5.1 | Compléments de terminologie : processus vs thread | 28 |
| 5.2 | L'interface UNIX (très partielle) | 28 |

| | | |
|------------|--|-----------|
| III | Eléments de mise en œuvre d'un noyau de synchronisation | 30 |
| 1 | Mécanismes de contrôle de l'exécution | 31 |
| 1.1 | Mécanismes de protection | 31 |
| 1.2 | Commutation de contexte | 31 |
| 1.3 | Mécanismes provoquant une commutation de contexte | 32 |
| 2 | Mise en œuvre du noyau de synchronisation | 33 |
| 2.1 | Représentation des processus | 33 |
| 2.2 | Mise en œuvre de l'exclusion mutuelle | 36 |
| 2.3 | Mise en œuvre des sémaphores | 39 |
| 2.4 | Architecture du noyau | 41 |
| 3 | Allocation de l'Unité Centrale (ordonnancement) | 42 |
| 3.1 | Préemption | 42 |
| 3.2 | Politiques d'allocation | 43 |
| IV | Synchronisation entre processus : compléments | 45 |
| 1 | Schémas classiques de synchronisation | 46 |
| 1.1 | Producteur/consommateur | 46 |
| 1.2 | Client/serveur | 48 |
| 1.3 | Sémaphores privés | 50 |
| 2 | Autres outils de synchronisation | 51 |
| 2.1 | Messages | 51 |
| 2.2 | Variations autour des sémaphores | 53 |
| 2.3 | Moniteurs | 55 |
| 2.4 | Transactions | 57 |
| V | Gestion d'entrées/sorties physiques | 60 |
| 1 | Le matériel | 61 |
| 1.1 | Périphérique (device) | 61 |
| 1.2 | Interface de contrôle | 62 |
| 2 | Le logiciel : problèmes de synchronisation | 65 |
| 2.1 | Le processeur à l'initiative du transfert | 65 |
| 2.2 | Le processeur n'a pas l'initiative du transfert | 68 |
| 3 | Le logiciel : problèmes de structuration | 70 |
| 3.1 | Interface de la machine virtuelle | 70 |
| 3.2 | Architecture fonctionnelle | 70 |
| 3.3 | Organisation du contrôle de l'exécution | 71 |

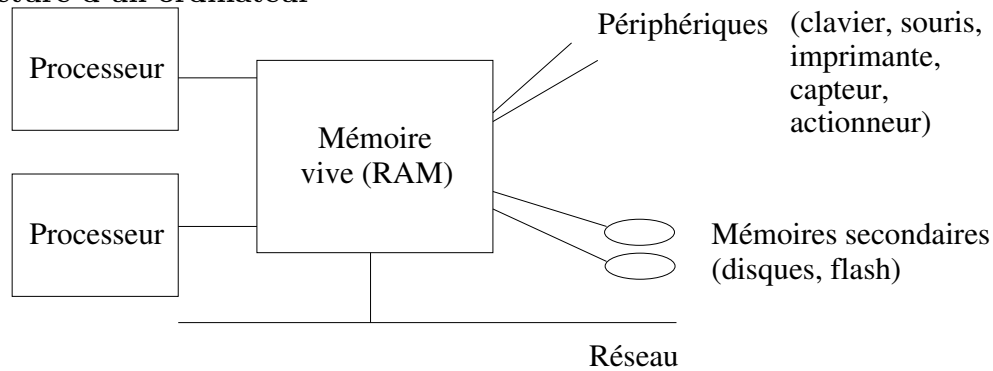
| | |
|--|-----------|
| 4 Exemple : gestion des disques | 74 |
| 4.1 Le matériel | 74 |
| 4.2 Exemple de contrôle des E/S | 75 |
| 4.3 Amélioration du temps de réponse | 77 |
| 4.4 Fiabilité | 81 |
| | |
| VI Allocation de ressources et interblocage | 82 |
| | |
| 1 Interblocage | 83 |
| 1.1 Définition et caractérisation | 83 |
| 1.2 État sain | 84 |
| 1.3 Types de solutions | 86 |
| | |
| 2 Prévention de l'interblocage | 87 |
| 2.1 Prévention statique | 87 |
| 2.2 Eviter l'interblocage : l'algorithme du banquier | 88 |
| | |
| 3 Détection et guérison de l'interblocage | 90 |
| 3.1 Détection de l'interblocage | 90 |
| 3.2 Détection de l'interblocage dans Linux et Windows | 91 |
| 3.3 Guérison | 91 |
| | |
| VII Problèmes d'architecture, exemples | 93 |
| | |
| 1 Modèle d'exécution offert à l'utilisateur | 94 |
| 1.1 Un espace virtuel par processus | 94 |
| 1.2 Plusieurs processus se partagent un espace virtuel | 96 |
| 1.3 Un espace virtuel unique | 97 |
| | |
| 2 Contrôle de l'exécution | 97 |
| 2.1 Contrôle par procédure : exemple d'UNIX | 97 |
| 2.2 Contrôle par processus : exemple de MINIX | 98 |
| | |
| 3 Architecture du système | 99 |
| 3.1 Système monolithique : UNIX | 99 |
| 3.2 Micro-noyau et serveurs : MACH | 100 |

Première partie

Introduction aux systèmes d'exploitation

1 Architecture d'un ordinateur

Architecture d'un ordinateur



Unité centrale (processeur)

cycle (pour chaque instruction)

<charger une instruction en mémoire>

<l'exécuter>

$PC = PC + \text{taille instruction}$ (ou saut)

si demande d'interruption, se brancher à la routine associée

fincycle

- Remarques 1.**
- *Exécution d'instruction atomique (indivisible) : pas d'état intermédiaire visible*
 - *Le fonctionnement du processeur présenté ici est quelque peu idéalisé ...*

Coupleurs (contrôleurs de périphériques)

- Programmation via des *registres*
- Signalement de fin d'entrées sorties
 - Par lecture registre et/ou
 - Par demande d'interruptions au processeur
- *Parallélisme* entre le fonctionnement du processeur et des périphériques

Comment utiliser un ordinateur ?

- Sans logiciel, un ordinateur est inutilisable
- Quel logiciel utiliser ?
 - *Dépend de l'utilisation* du système informatique
- ⇒ Pour tout domaine d'utilisation, *ensemble commun de fonctions* utiles à toutes les applications
- ⇒ Regroupées dans ce qu'on appelle *système d'exploitation*

2 Rôle d'un système d'exploitation

2.1 Définition d'un système d'exploitation

Système d'exploitation

- Definition 2** (Système d'exploitation). – Un système d'exploitation est un ensemble de fonctions, mises en œuvre *par logiciel*, permettant d'offrir à ses utilisateurs une version plus abstraite de la machine (physique) sous-jacente (*machine virtuelle*).
- La machine virtuelle est définie par une *interface*, composée d'un ensemble d'*appels système*.

Interface d'un système d'exploitation

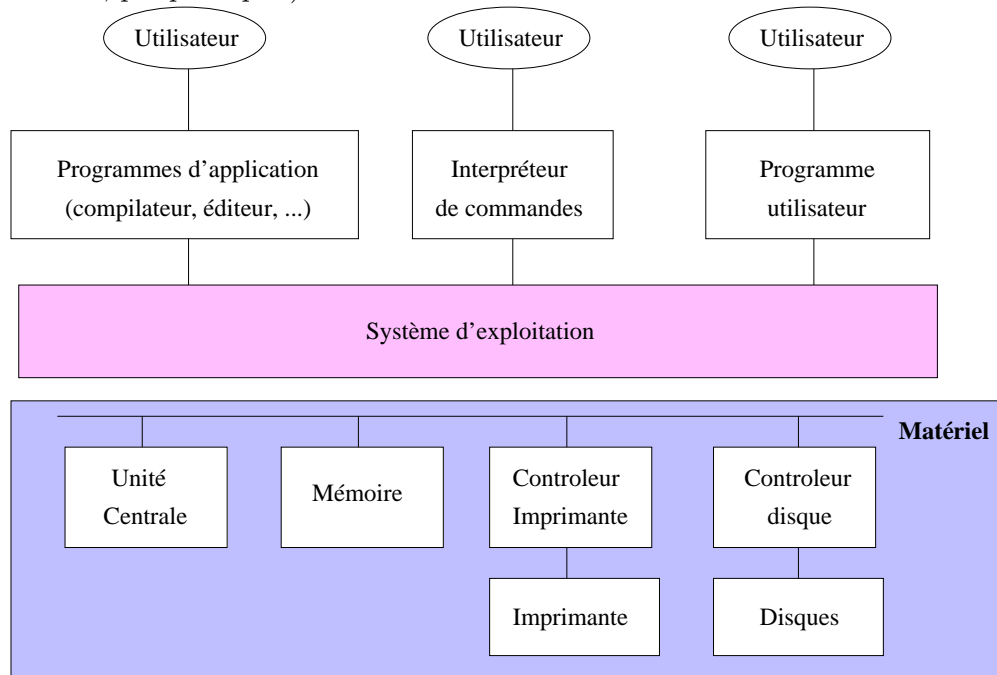
Exemple 3. API (Application Programming Interface) POSIX (normalisation de l'interface des systèmes UNIX) :

- Exécution des programmes en multi-tâches (partage du processeur)
- Stockage d'informations (fichiers, répertoires)
- Entrées/sorties
- Communication et synchronisation entre programmes (messages, sémaphores, tubes/pipes)
- Gestion du temps (alarmes, accès au temps système)

Rôles d'un système d'exploitation

Aspects complémentaires du rôle d'un système d'exploitation :

- *Définir la machine virtuelle* (interface) la plus adaptée aux utilisateurs. Intérêt : augmenter la *portabilité* des programmes d'une architecture matérielle à une autre
- *Mettre en œuvre* cette machine virtuelle, en gérant "au mieux" les ressources de l'architecture (processeur, périphériques)



Critères d'appréciation d'un système d'exploitation

- Pour l'utilisateur : *puissance* de la machine virtuelle, confort d'utilisation (ergonomie), fiabilité
- Pour le responsable de l'installation : efficacité de la gestion globale des ressources :
 - *Performance* : temps d'exécution des programmes
 - *Équité* de l'avancement de l'exécution des différents programmes

- Bonne utilisation des périphériques et ressources (mémoire, disque, périphériques)

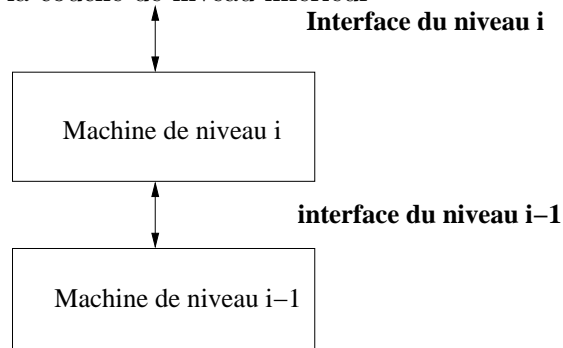
2.2 Structuration d'un système d'exploitation

- *Ecart important* entre machine virtuelle et machine réelle \implies passage de l'une à l'autre est *complexe*
- Possibilité : définition de machines virtuelles intermédiaires (*structuration en couches*)

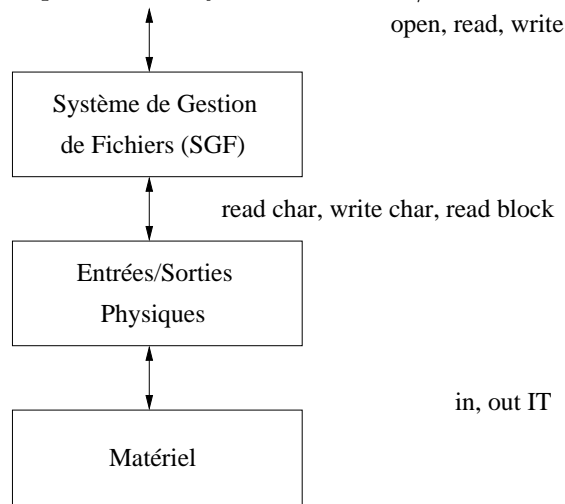
Structuration en couches

Chaque couche (niveau) :

- Exporte une interface aux couches supérieures
- Utilise l'interface de la couche de niveau inférieur



Exemple 4. Exemple : décomposition du système d'entrées/sorties



Intérêts :

- Interface : *abstraction* des détails de mise en œuvre en ne gardant que les concepts essentiels
- Possibilité de modification d'un niveau (optimisation, adaptation) *indépendamment des autres*
- On peut rendre une bonne partie du système *indépendant du matériel* \implies évolutivité

2.3 Un exemple simple : micro-ordinateur individuel

Système mono-utilisateur

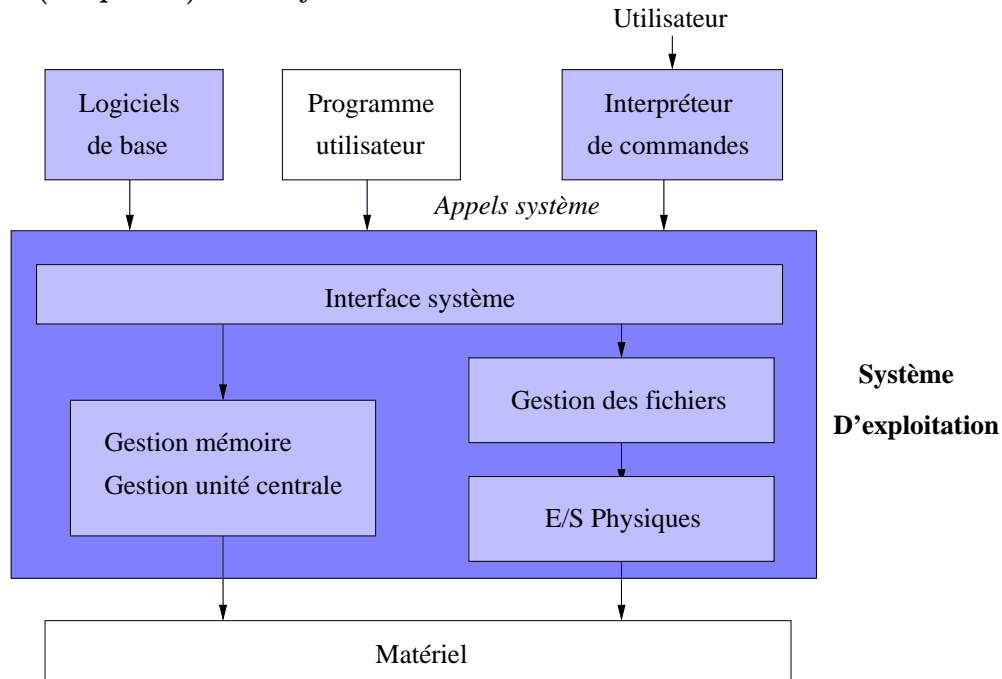
- Matériel : processeur + mémoire + clavier/écran + lecteurs de disques et contrôleurs adaptés

- *Un seul utilisateur* sur la machine à la fois
- *Un seul programme* en cours d'exécution à la fois
- Exécution de logiciels interactifs (éditeur de texte, tableur, etc)
- Besoin important de stockage d'informations

Interface d'un système mono-utilisateur

- Manipulation de fichiers (création, destruction, copie, gestion des noms)
- Entrées/sorties
- Chargement et exécution de programmes, un seul étant exécuté à la fois (système *mono-programmé, mono-tâche*)
- Gestion de la mémoire (allocation, libération)
- Configuration du système (reconnaissance de nouveaux périphériques)
- Divers (utilisation de l'horloge)

Structure (simplifiée) d'un système mono-utilisateur



3 Introduction aux systèmes multi-tâches

Systèmes multi-tâches

Definition 5 (Système multi-tâches). Un système d'exploitation est dit *multi-tâche* (*multiprogrammé*) lorsque deux programmes peuvent être en cours d'exécution en même temps

Remarque 6.

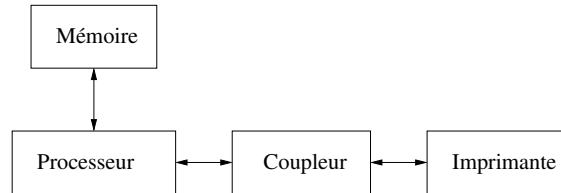
- *En cours d'exécution* = ayant commencé à s'exécuter et n'ayant pas encore terminé
- Deux programmes peuvent s'exécuter réellement en même temps uniquement si l'on dispose de plusieurs processeurs

- Quel est l'intérêt des systèmes multi-tâches ?
 - ⇒ Montré sur deux exemples
- Quels problèmes le multi-tâche pose au système d'exploitation ?
 - ⇒ Enoncés de ces problèmes
 - ⇒ Solutions à ces problèmes font l'objet de la suite du cours

3.1 Exemple 1 : synchronisation entre processeur et périphérique

Exemple 1 : matériel

Contrôle par le processeur d'une imprimante via un coupleur capable d'échanger des caractères :

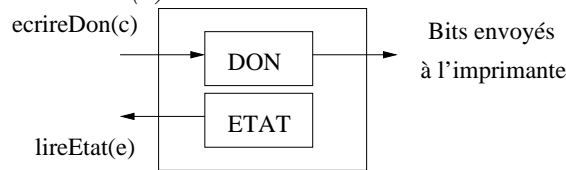


Fonctionnement du coupleur en sortie

Registres du coupleur :

- DON : registre de donnée. Stockage d'un octet ⇒ lancement de l'E/S
- ETAT : situation du coupleur (libre/occupé)

Instructions *lireEtat(e)* et *ecrireDon(c)*



⇒ Besoin d'ordonner dans le temps les actions du processeur et du coupleur (*synchronisation*)

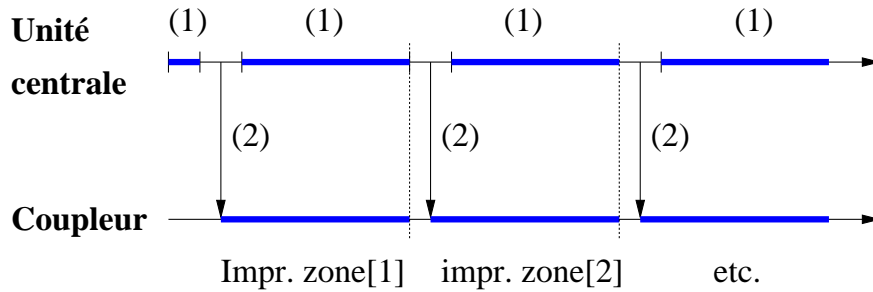
Synchronisation par attente active

- Le processeur observe en continu l'état du coupleur, dès que celui-ci passe à *libre*, le processeur peut lancer une nouvelle sortie de caractère
- Exemple : impression du contenu d'une zone de 100 caractères

```

char zone[100];
for (i=0; i<100; i++) { // pour chaque caractère de zone
    lireEtat(&e);
    while (e==occupe) lireEtat(&e);           (1)
    ecrireDon(zone[i]);                       (2)
}

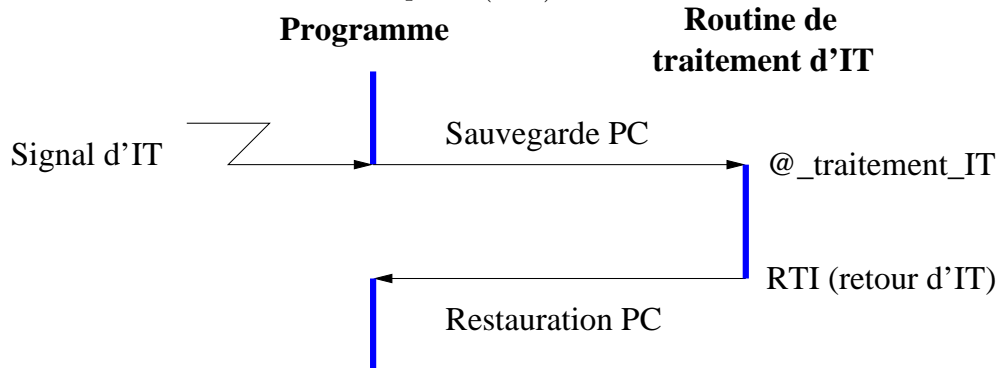
```



Mécanisme d'interruption

Lorsqu'un signal d'interruption est émis (*demande d'interruption*) :

- A la fin de l'instruction en cours d'exécution
 - *Sauvegarde* de la valeur courante du compteur ordinal (PC)
 - *Branchement* à une adresse fixe AD_TRAIT_IT
- Une instruction de retour d'interruption (RTI) affecte PC à la dernière valeur sauvegardée



Masquage vs suppression d'interruption

- *Suppression d'interruption* : configuration du périphérique pour qu'il ne positionne pas de signal d'interruption
- *Masquage d'interruption* : configuration du processeur pour qu'il n'effectue pas de déroutement en cas de signal d'interruption (apparaît dans le mot d'état du processeur) - Instruction x86 sti/cli

Synchronisation par interruption

- Programmation du coupleur pour qu'il émette un signal d'interruption à la fin de l'impression de tout caractère
- Programme et routine de traitement d'IT associés :

Programme P

```
// On suppose qu'il n'y a pas d'E/S en cours
// Initialisation de l'impression
i=0; ecrireDon(zone[i]);
... (*) ...
```

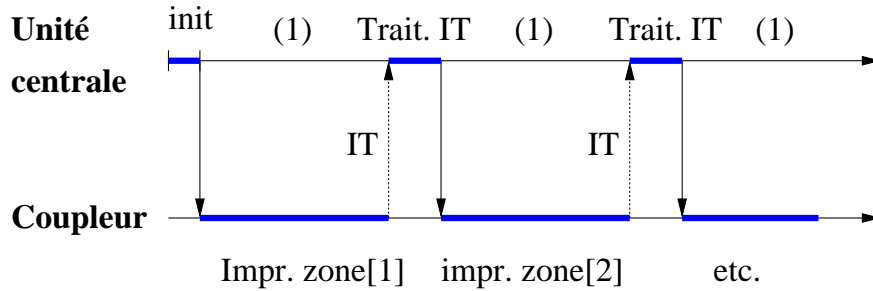
Traitement d'IT

```
// On est sûr qu'il n'y a pas d'E/S en cours
i = i+1;
```

```

if (i ≤ 100) ecrireDon(zone[i]);
Retour_IT

```



Quoi faire dans les zones marquées (1) ?

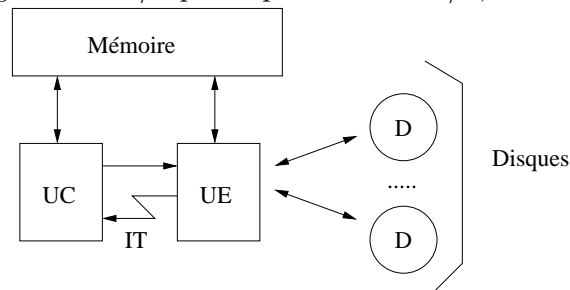
- Continuer l'exécution du programme P : *parallélisme entre E/S et calcul*
- Exécuter un autre programme P' (par exemple quand P doit attendre la fin de l'impression avant de faire autre chose) : on est alors dans un cas de *multi-tâche*, car on fait progresser deux activités en même temps

⇒ Intérêt du multi-tâche : mieux utiliser le processeur en exploitant le parallélisme entre E/S et calcul

3.2 Exemple 2 : utilisation du parallélisme entre E/S et calcul

Exemple 2 : matériel

- Deux dispositifs ayant un accès direct à la mémoire (processeur, processeur d'E/S)
- Prise en charge intégrale de l'E/S par le processeur d'E/S, interruption en fin d'E/S



Logiciel

- Procédures système (exécutées par le processeur)
 - *lancerLectBloc(d, adm)* : lancement de la lecture du bloc suivant sur le disque d , vers la zone d'adresse adm (durée lecture 100 ms)
 - *attendreFinLect(d)* : attendre la fin de la lecture en cours sur le disque d
- Application P1 : traitement des 1000 blocs disque d'un fichier situé sur le disque 1 (20 ms de traitement par bloc)

Solution séquentielle

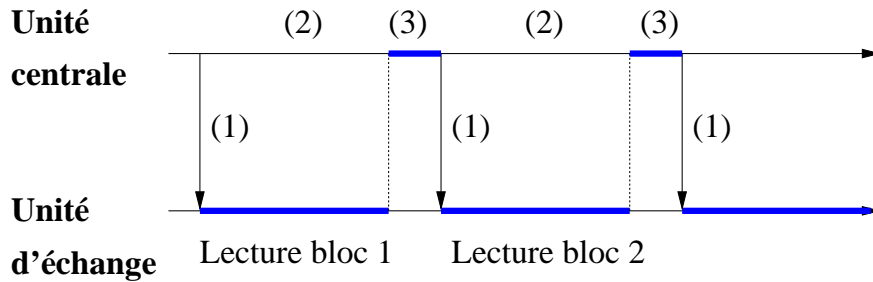
- Une seule zone mémoire de la taille d'un bloc ⇒ attente de la fin de la lecture avant de pouvoir le traiter

– Programme :

```

char t[SZ];
for (i=0; i<1000; i++) {
    lancerLectBloc(1,t);           (1)
    attendreFinLect(1);           (2)
    traiterBloc(t);               (3)
}

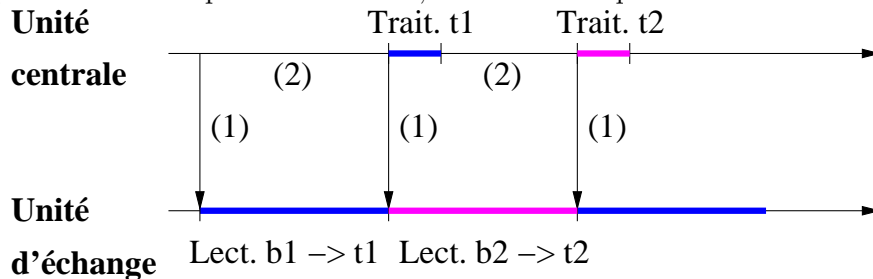
```



- Durée du programme = 120 s
- Occupation processeur = $20 / 120 = 16,6\%$
- ⇒ On veut exploiter le parallélisme entre calcul et entrée/sortie

Gestion du parallélisme par le programme

Deux tampons : un dans lequel on lit un bloc, l'autre dans lequel on traite le bloc lu précédemment



- Durée du programme : 100 s
- Occupation processeur : 20%

```

char t1[SZ],t2[SZ];
char *adLect,*adTrait;
char *suivant(char *adt); //rend l'adresse de l'autre tampon

```

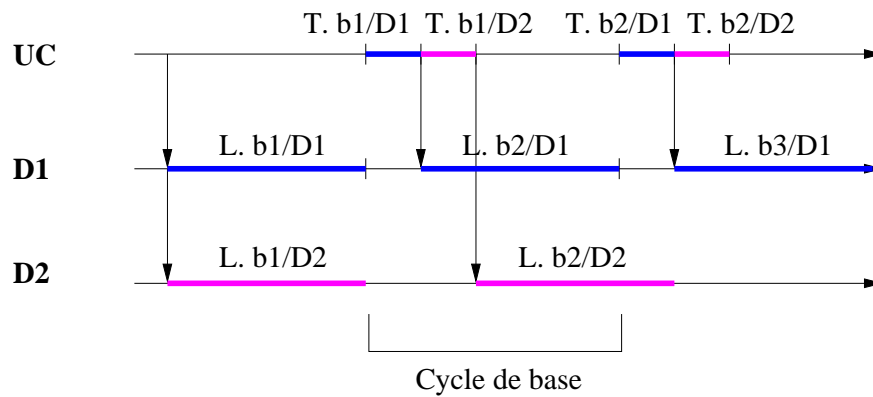
```

adLect = t1; lancerLectBloc(1,adLect);
attendreFinLect(1);
for (i=0; i< 999; i++) {
    adTrait = adLect;
    adLect = suivant(adLect);
    lancerLectBloc(1,adLect);           (1)
    traiterBloc(adTrait);               (2)
    attendreFinLect(1);
}

```

Gestion du parallélisme par le système

Deux programmes P1 et P2 lisent et traitent les blocs de deux fichiers situés sur des disques différents, 1 tampon par programme



- Durée d'exécution de P1+P2 : environ 120 s
- Taux d'occupation du processeur : 33,3 %

3.3 Quelques problèmes posés par le multi-tâche

Problèmes posés par le multi-tâche

- *Partage de ressources* par plusieurs programmes
- Types de ressources :
 - Processeur
 - Mémoire
 - Périphériques

Partage du processeur

Un seul processeur, plusieurs programmes

- Arrêt et reprise d'un programme :
 - Sauvegarde et restauration du *contexte d'exécution* du programme
 - *Synchronisation* : mécanismes pour suspendre un programme et le réveiller ensuite
 - *Allocation du processeur (ordonnancement)* : choix du programme à exécuter à un instant donné
- ⇒ Notion de *processus* (chapitre suivant)

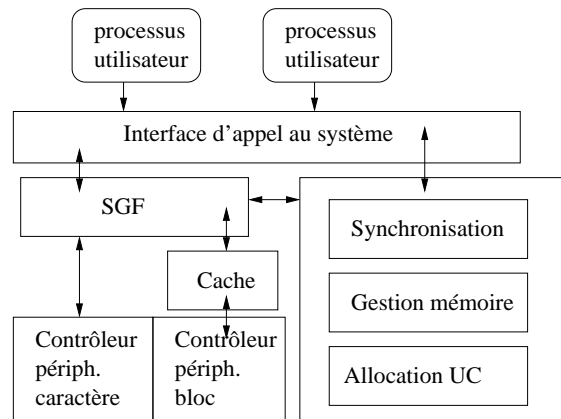
Partage de la mémoire

- *Allocation de la mémoire*
- *Synchronisation* des accès à la mémoire.
- *Cohabitation avec l'ordonnancement*
- *Protection*

Partage des périphériques

- *Synchronisation* des accès aux périphériques

Structure d'un système multi-tâche



4 Un peu d'histoire des systèmes d'exploitation

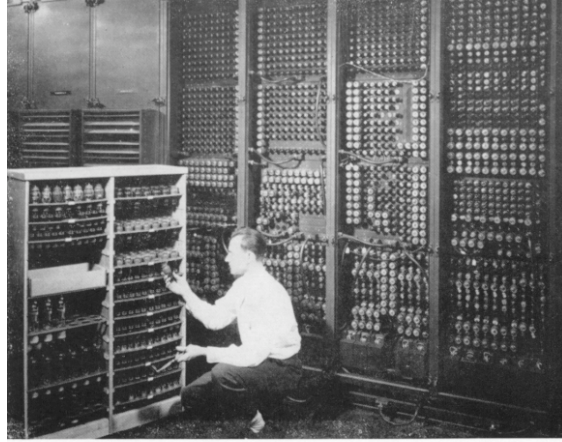
Un peu d'histoire ...

- Sans prétention d'exhaustivité
- Uniquement pour montrer que les problèmes posés *ne sont pas nouveaux* (de même que leurs solutions)
- Historique commence vers le milieu des années 40

Les “premiers” systèmes (années 40 à 50)

- Matériel : processeurs peu puissant et très chère, peu de mémoire, périphériques lents (lecteurs de cartes, imprimantes)
- Logiciels
 - Traducteurs de langages simples (assembleur, Fortran)
 - Bibliothèque de programmes d'entrées/sorties
- Accès à la machine en “porte ouverte” : réservation de la machine par tranche horaire, contrôle direct par l'utilisateur via le tableau de commande (mono-usager, mono-tâche)
- Fonctionnement *très interactif*, mais *très mauvaise* utilisation des ressources
- Exemples : ENIAC(1946), EDVAC (1944-46) - technologie de tubes à vide. Premier ordinateur commercial : UNIVAC I (1951)

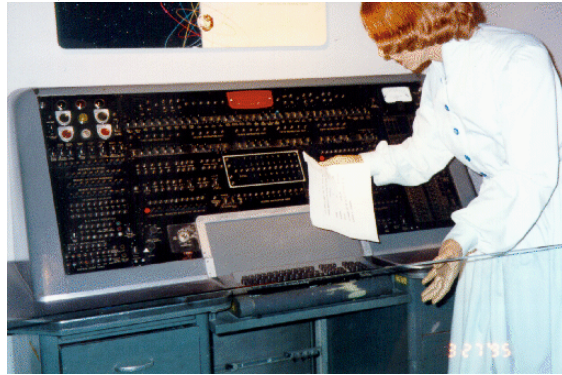
L'ENIAC I (1946)



Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

L'UNIVAC I (1951-1957)

- 5000 tubes à vide, 7 tonnes, 1000 calculs par seconde
- Vendu à 46 exemplaires (agences gouvernementales)



Systemes à enchainement de travaux ("batch") (fin 50 à début 60)

- Matériel : évolution sur la période
 - Apparition des lecteurs de bandes magnétiques, plus rapides que les lecteurs de carte
 - Apparition des transistors
- Logiciels : vers une rationalisation de l'utilisation du processeur
 - Plus d'accès direct à la machine. Soumission de travaux ou "jobs" (paquets de cartes) récupération de résultats (listing). Un *opérateur* (humain) lance les travaux
 - Améliorations par apparition de *moniteurs d'enchânement des travaux* qui enchainent les jobs automatiquement (disparition de l'opérateur)
 - Les systèmes restent mono-usager

Systemes à enchainement de travaux ("batch") (fin 50 à début 60)

Moniteurs d'enchânement des travaux

- Plan mémoire

| |
|--|
| Moniteur (résident en permanence) |
| Espace utilisateur (compilateur, programme, données, etc) |

- Programme source donné au moniteur
 - \$JOB user_spec ; identify the user for accounting purposes
 - \$FORTRAN ; load the FORTRAN compiler
 - source program cards
 - \$LOAD ; load the compiled program
 - \$RUN ; run the program
 - data cards
 - \$EOJ ; end of job
 - \$JOB user_spec ; identify a new user
 - \$LOAD application
 - \$RUN
 - data
 - \$EOJ

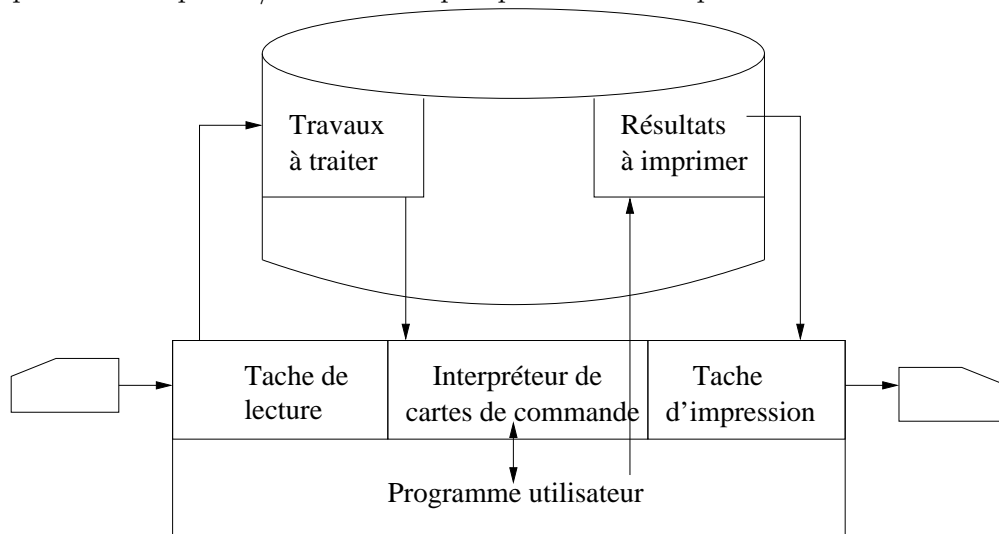
Evolutions dans les années 60

Evolutions matérielles

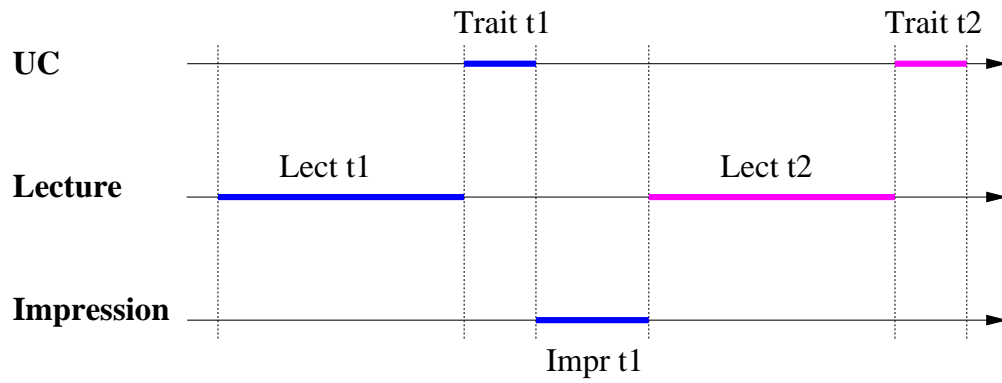
- processeurs plus puissants et moins chers
- Mémoires plus grandes
- Apparition des disques
- Contrôleurs d'E/S autonomes

Evolutions dans les années 60 - tamponnage des E/S (spooling)

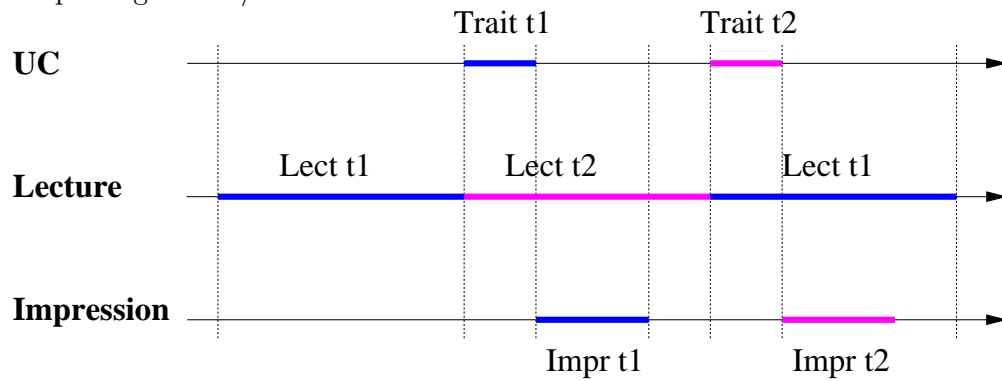
- Toujours en mode *mono-utilisateur, mono-tâche*, non interactif, avec traitement par lots (batch)
- *Tamponnage des Entrées/Sorties* (Spooling - Simultaneous Peripheral Operation On-Line) : récupérer les temps d'E/S sur un tampon pour faire des opérations sur un autre



Sans tamponnage des E/S



Avec tamponnage des E/S



Remarque 7. *Ce type de système posait déjà certains des problèmes liés au multi-tâche :*

- *Réalisation des E/S en utilisant des interruptions*
- *Partage du processeur entre les tâches système et le programme utilisateur*
- *Protection du système résident en mémoire vis à vis du programme utilisateur*

IBM 360



Systemes à temps partagé (années 70)

- Evolutions matérielles : apparition des terminaux (clavier + papier, puis clavier + écran)

- Evolutions des systèmes d'exploitation : retour à un fonctionnement plus *interactif*. Chaque utilisateur est doté d'un terminal et peut intervenir à tout instant sur le déroulement de son programme
- *Multi-tâche*, processeur partagé par *tranche de temps* de durée fixée, pour équité entre usagers
- Exemple : Multics
- ⇒ Tous les mécanismes de base des systèmes multi-programmés conçus à cette époque

PDP-11



Mini et micro-ordinateurs (années 70 - 80)

- Evolutions matérielles : apparition des microprocesseurs (Intel 4004 en 1971, 8008 en 1972) permet de répandre l'utilisation des ordinateurs
- On revient à "une machine - un utilisateur" (ère de l'*ordinateur personnel*). Systèmes CP/M, MS/DOS
- Evolutions avec l'augmentation des performances des processeurs et des écrans :
 - Interfaces plus conviviales (fenêtres, souris)
 - Multifenêtrage ⇒ retour au *systèmes multi-tâches* (même si toujours mono-usager)
- Actuellement, peu de différences entre les systèmes d'exploitation pour ordinateur personnel et serveur (mêmes concepts et mécanismes)

Réseaux et systèmes distribués (années 80)

- Réseaux *longue distance* (WAN) pour relier les ordinateurs centraux (Transpac)
- Réseaux *locaux* (LAN) pour relier micro-ordinateurs d'une même structure géographique (⇒ partage de ressources, disques, imprimantes, etc)
- ⇒ *Systèmes distribués* pour faire abstraction de la localisation physique des ressources partagées

Informatique embarquée (années 90)

- Evolutions matérielles : baisse des coûts et augmentation des performances des processeurs

- Conséquence : on embarque des processeurs partout (lave-vaisselle, téléphones portables, automobile, plus de 80 processeurs dans une Laguna). Une Mégane de 2002 embarque plus de puissance de calcul qu'un Airbus de 1970
- Impact sur les systèmes d'exploitation : *contraintes différentes* (énergie, temps-réel, pas toujours de disque, coût, ressources limitées)
- Et maintenant ? System on a Chip, Network on a Chip, multi-cœurs

5 Bibliographie

Bibliographie

- Volontairement limitée à un petit nombre d'ouvrages accessibles et disponibles à la bibliothèque
- Bases de travail : [1], [2], [3] (version ré-éditée de [2]), [4]
- Partie sur les systèmes distribués dans [5] et [4]

Deuxième partie

Processus et synchronisation

1 Exécution d'un programme

Programme

Definition 8 (Programme). Un *programme* est un texte décrivant à la fois des objets (données) et les actions à effectuer sur ces objets (instructions)

1.1 Processeur et trace d'exécution

Fonctionnement (simplifié) d'un ordinateur

- *Mémoire centrale* (RAM) : contient une représentation des instructions et des données d'un programme
- *Processeur* : mécanisme d'exécution, l'état d'avancement de l'exécution étant contenu dans un registre PC. Autres *registres* servant comme éléments de mémorisation
 - cycle** (pour chaque instruction)
 - <charger une instruction en mémoire>
 - <l'exécuter>
 - PC = PC + taille instruction (ou saut)
 - si demande d'interruption, se brancher à la routine associée
 - fincycle**
- Exécution *séquentielle* des instructions, exécution *atomique* (*indivisible*) (pas d'état intermédiaire visible)

Trace d'exécution

- Definition 9.**
- *Etat* de la machine : état du processeur et état de la mémoire
 - *Point d'observation* : instant où on peut observer l'état de la machine
 - *Etat observable* : état de la machine en un point d'observation
 - *Trace d'exécution* : suite, ordonnée dans le temps, d'états observables
 - (Notion équivalente, état initial de la machine et suite des instructions exécutées)

Exemple 10. Programme

```
MOV AX,2    (1)
CMP AX,0    (2)
JNE E1      (3)
MOV BX,0    (4)
JMP E2      (5)
E1 : MOV BX,1 (6)
E2 :
```

Trace d'exécution (ici, une)

- (1)
- (2)
- (3)
- (6)

1.2 Trace d'exécution dans le cas d'activités parallèles

Trace d'exécution et activités parallèles

Programmes considérés

- Programme P du premier exemple de l'introduction

0 i = 0; (p1)

4 ecrireDon(zone[i]); (p2)

8 AttendreFinImpression;

Traitement d'IT :

1000 i :=i+1 (i1)

1004 **if** (i>100) **goto** FIN (i2)

1008 ecrireDon(zone[i]); (i3)

100C FIN : RéveillerPgmImpression ; RTI; (i4)

- Programme P' indépendant

100 k :=0; (a1)

104 k :=k+1; (a2)

108 k :=77; (a3)

- P implanté à l'adresse 0
- P' implanté à l'adresse 100
- Routine de traitement d'interruptions implantée à l'adresse 1000
- Instructions sur 4 octets
- Etat de la machine (zone intéressantes seulement)
 - Etat de la mémoire pour les variables *i* et *k*
 - Compteur ordinal (PC)
 - registre de données du coupleur (DON)
 - Emplacement de sauvegarde de l'adresse de retour en cas d'IT (SAUV)

| Instr | i | k | PC | DON | SAUV |
|---------|---|----|------|-----|------|
| | - | - | 0 | - | - |
| p1 | 0 | - | 4 | - | - |
| p2 | 0 | - | 100 | a | - |
| a1 | 0 | 0 | 104 | a | - |
| IT → i1 | 0 | 1 | 1004 | a | 104 |
| i2 | 1 | 1 | 1008 | a | 104 |
| i3 | 1 | 1 | 100C | a | 104 |
| i4 | 1 | 1 | 104 | b | 104 |
| a2 | 1 | 1 | 108 | b | 104 |
| a3 | 1 | 77 | 10C | b | 104 |

- Remarque 11.** – *La trace d'exécution au niveau processeur entrelace les traces des différentes activités. L'entrelacement peut être quelconque (instant de demande de l'interruption imprévisible)*
- *L'entrelacement rend difficile l'examen de la trace. Il serait plus intéressant pour l'analyse de séparer les deux activités*
 - *Entrelacement entre P et P' soumis à des contraintes (attendreFinImpression). On exprime ici une contrainte de synchronisation*

2 Notion de processus

2.1 Définitions

Definition 12 (Processus). Un *processus* (ou *processus séquentiel*) est l'exécution d'un texte de programme séquentiel.

C'est la suite, ordonnée dans le temps, d'actions atomiques effectuées lors de l'exécution d'un texte de programme séquentiel

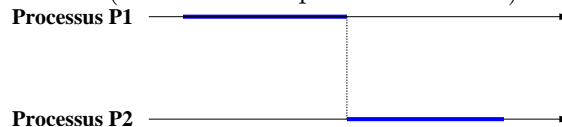
Definition 13 (Contexte d'un processus). Le *contexte d'un processus* est constitué de l'ensemble des informations que les actions du processus peuvent consulter ou modifier

- Un processus est *séquentiel*, c'est à dire que si dans un processus on a la suite d'actions ..., a_i , a_{i+1} , ... alors la fin de a_i a lieu avant le début de a_{i+1}
- Intérêt de la notion de processus : raisonner au niveau de l'*activité séquentielle* en faisant abstraction des problèmes d'entrelacement entre activités

2.2 Modes d'exécution des processus : niveaux de parallélisme

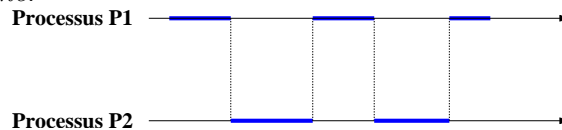
Exécution séquentielle

- Un seul processeur
- Exécution de P1 puis de P2 (début de P2 après la fin de P1)



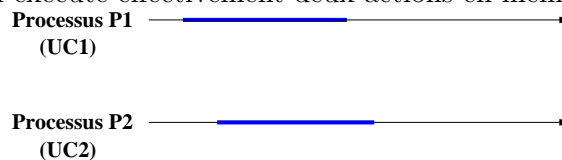
Exécution pseudo parallèle

- Toujours un seul processeur
- On peut exécuter une partie de P1, puis une partie de P2, etc, jusqu'à la fin de P1 et P2
- *Parallélisme* : le début de P2 est antérieur à la fin de P1 et le début de P1 est antérieur à la fin de P2 (il y a des instants où P1 et P2 sont tous les deux en cours d'exécution).
- Mais, comme on a un seul processeur, une seule action est faite à un instant donné, on parle de *pseudo-parallélisme*.



Parallélisme réel

- Si on dispose de deux processeurs, on peut exécuter P1 sur l'un et P2 sur l'autre
- *Parallélisme* : le début de P2 est antérieur à la fin de P1 et le début de P1 est antérieur à la fin de P2
- *Parallélisme réel*, on exécute effectivement deux actions en même temps

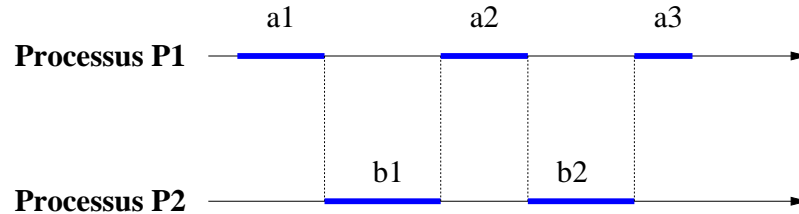


2.3 Relations temporelles : temps logique et temps physique

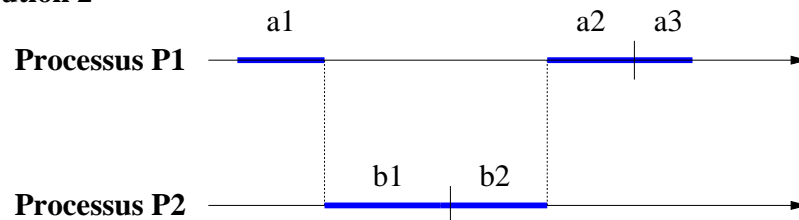
Exemple

- P1 et P2 processus indépendants, a_i b_i actions atomiques
- P1 : a_1 a_2 a_3
- P2 : b_1 b_2
- Exécutions pseudo parallèles possibles :

Execution 1



Execution 2

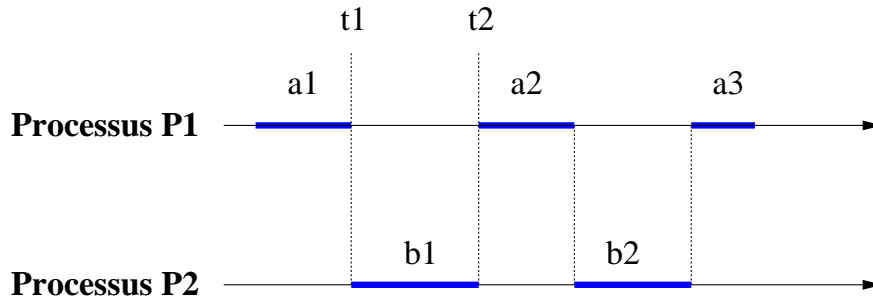


Relations temporelles dans un processus

- Actions atomiques à l'intérieur d'un processus sont *ordonnées*
- Valeurs absolue de l'intervalle de temps *réel* qui sépare a_1 de a_2 peut varier
- ⇒ Raisonnement dans le temps *logique* du processus, qui n'est pas le temps *réel* du processeur
- On ne peut rien dire a priori sur l'ordre dans lequel sont exécutées deux actions de deux processus distincts
- Ordre dépend de la manière dont le système d'exploitation choisit de faire progresser les processus (ordonnancement)
- Cette propriété importante peut être formulée de 2 manières :
 - En l'absence de synchronisation explicite, il n'y a pas d'ordre imposé sur l'exécution de deux actions atomiques de deux processus différents
 - On ne peut pas faire d'hypothèses a priori sur les vitesses d'exécution de deux processus
- Moyen d'imposer des contraintes sur l'ordre d'exécution : *synchronisations explicites*

2.4 Processeur virtuel

- Ordonnancement dans le temps de P1 et P2 ci-dessous ne dépend pas de P1 et P2 (indépendants, pas de synchronisation explicite)



- Résulte de la manière dont le processeur est partagé entre les processus
- Pour P1, son t_2 est le même qu'en t_1 . De son point de vue, rien ne s'est passé entre t_1 et t_2 .
- ⇒ Notion de *processeur virtuel*

Definition 14 (Processeur virtuel). On considère que chaque processus est doté d'un *processeur virtuel* qui lui est propre (mécanisme d'exécution, PC, registres,...).

Pour un processus P, tout se passe comme si on disposait d'un processeur dédié à l'exécution de P, dont la vitesse d'exécution est indéterminée.

Definition 15 (Processus). Un *processus*, c'est la suite d'actions atomiques engendrée par l'exécution d'un texte de programme sur un processeur (virtuel) qui lui est entièrement dédié

3 Section critique - exclusion mutuelle

Introduction au problème d'exclusion mutuelle

- Coopération entre processus ⇒ communication entre processus, via la mémoire, qui devient une *ressource partagée*
 - Malheureusement, les opérations élémentaires (ex : affectation) peuvent conduire à des *incohérences*
 - ⇒ De manière générale l'accès à des ressources partagées entre plusieurs processus nécessite des *synchronisations explicites* pour être correct
1. Mise en évidence ici de ces problèmes d'incohérence (sur des exemples)
 2. Définir la notion de section critique (mise en œuvre sera vue dans le chapitre suivant)

3.1 Exemples

Exemple 1 : ressource matérielle partagée

- Soient deux programmes P1 et P2 désirant imprimer sur la même imprimante (mode caractère)
- La fonction *ecrire* est supposée être une opération *atomique* d'impression d'un caractère

| | |
|---|---|
| P1 : char t1[4]='abcd'; for (i=0; i<4;i++) { ecrire(t1[i]); } | P2 : char t2[4]='ABCD'; for (i=0; i<4;i++) { ecrire(t2[i]); } |
|---|---|

Comportement en l'absence de synchronisation explicite

- On peut voir imprimer n'importe quelle séquence de caractère respectant l'ordre d'exécution de chaque programme ('abcdABCD' 'aAbBcCdD', 'abcABCdD', etc.)
- On voudrait voir imprimée la séquence 'abcdABCD'

Exemple 2 : ressource logicielle (variable) partagée

- Soient deux processus *Retrait* et *Dépot* correspondant à deux types d'opérations sur votre compte en banque
- Variable *solde* partagée entre les deux processus

int solde = 400;

Retrait (ent somme) :

solde = solde-somme;

Retrait :

MOV AX,solde (R1)

SUB AX,somme (R2)

MOV solde,AX (R3)

Dépot (ent somme) :

solde = solde+somme;

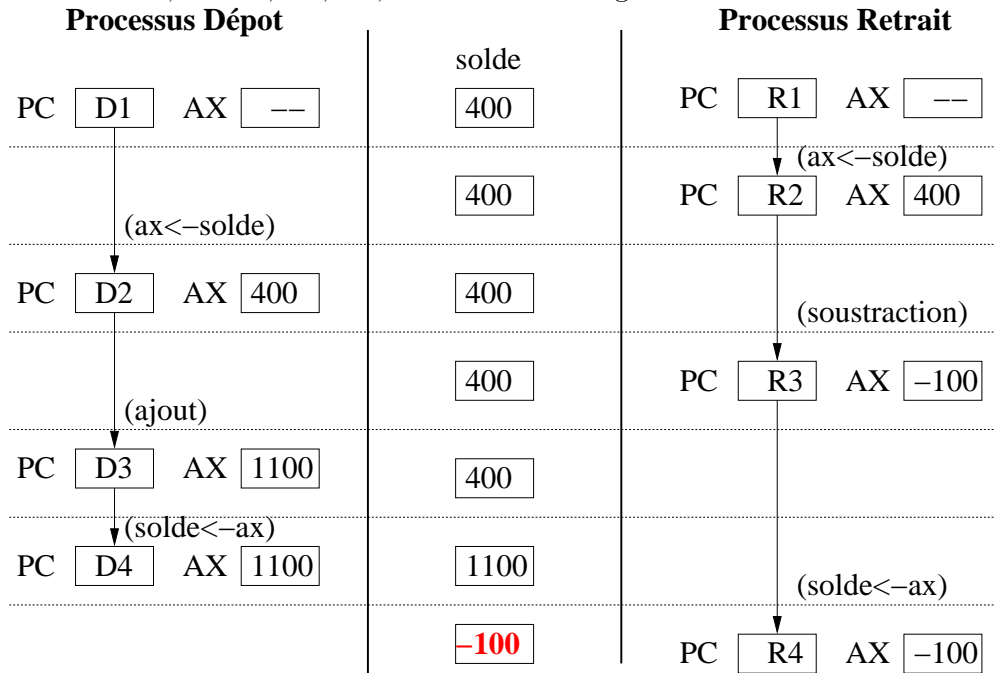
Dépot :

MOV AX,solde (D1)

ADD AX,somme (D2)

MOV solde,AX (D3)

- Exécution séquentielle de *Retrait(500)* puis *Dépot(700)* aboutit à un solde de 600 (même chose si l'on inverse l'ordre)
- Exécution de R1, D1 R2, D2, D3, R3 \implies solde négatif de -100



\implies On voudrait voir ici le retrait puis le dépôt, ou l'inverse, mais en aucun cas le résultat obtenu ici. On veut rendre chacune des opérations (Retrait, Dépot) *indivisibles*.

Exemple 3 : ressource logicielle (variable) partagée

- Variable partagée *v* entre deux processus P1 et P2
- Le code traduit le fait que *v* ne doit jamais devenir négatif (*v* n'est pas modifiée par ailleurs)

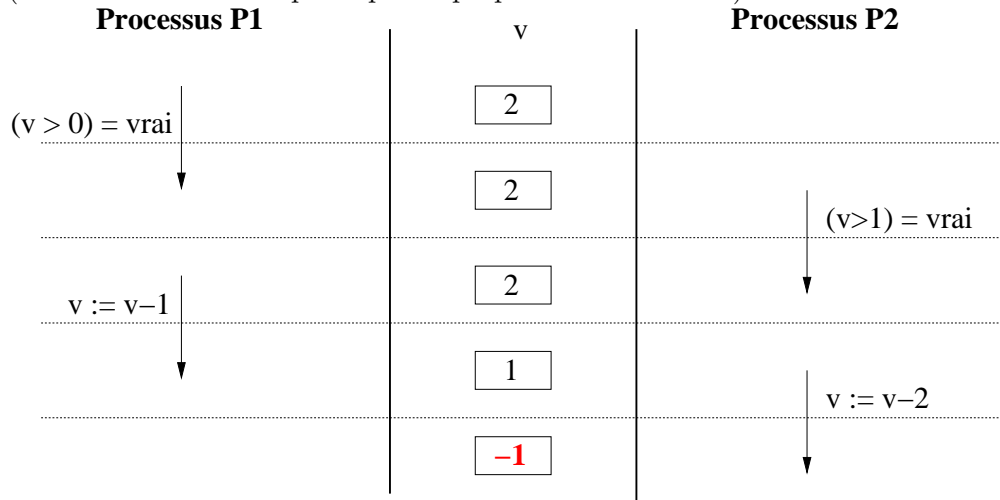
```

int v = 2;
processus P1;
...
if (v > 0) v = v - 1;
...

processus P2;
...
if (v > 1) v = v - 2;
...

```

- Exécutions séquentielles possibles donnent les résultats suivants, conformes à la propriété voulue sur v :
 - $\{v = 2\}$ P1 $\{v = 1\}$ P2 $\{v = 1\}$
 - $\{v = 2\}$ P2 $\{v = 0\}$ P1 $\{v = 0\}$
- Une exécution parallèle de P1 et P2 sans synchronisation explicite peut laisser v à 0, 1, ou -1 (dernière valeur ne respecte pas la propriété voulue sur v)



- Dans ce cas, on peut rendre *indivisibles* chacune des deux séquences de code manipulant v

3.2 Section critique et exclusion mutuelle

Section critique et exclusion mutuelle : définitions

Definition 16 (Exécution acceptable). Une exécution parallèle est *acceptable* si elle produit le même résultat qu'une exécution séquentielle possible (il peut en exister plusieurs, comme le montre l'exemple 3)

Definition 17 (Section critique). La partie du programme qui peut rendre inacceptable le résultat de l'exécution parallèle est une *section critique*.

Une section critique est une portion de code que l'on veut rendre indivisible (atomique).

Règle d'or

Toute ressource partagée (matérielle, variable) doit être manipulée en section critique.

Definition 18 (Ressource critique). Une *ressource critique* est une ressource dont la manipulation doit être effectuée dans une section critique

Definition 19 (Exclusion mutuelle). Quand deux processus manipulent une ressource dans une section critique, on dit que la ressource est manipulée en *exclusion mutuelle* (i.e. les deux processus n'y touchent pas en même temps)

Propriétés

Propriétés des algorithmes mettant en œuvre l'exclusion mutuelle :

1. *Sûreté* : deux processus ne doivent pas être en même temps dans leur section critique.
2. *Vivacité* (absence de blocage inutile) si aucun processus n'est en section critique, aucun processus ne doit être bloqué par le mécanisme de contrôle de la section critique.
3. *Absence de privation* : aucun processus ne doit être obligé d'attendre indéfiniment avant de pouvoir entrer en section critique.
4. *Généralité* : on ne doit pas faire d'hypothèses sur la vitesse relative des processus.

Solution triviale

```
char occup = 0; // indique si un processus est en SC
while (occup == 1) {; // rien };
occup = 1;
<section critique>;
occup = 0;
```

- La solution triviale *ne fonctionne pas* (si deux processus essaient d'entrer simultanément en section critique, les deux voient *occup* à 0 et entrent en section critique)
- Le problème vient de l'absence d'atomicité de la séquence test-modification de *occup*
⇒ Solutions au problème présentées dans le chapitre suivant

4 Un mécanisme de synchronisation : le sémaphore

4.1 Définition des sémaphores

Sémaphore

- Introduit par Edsger Dijkstra en 1965
- Sémaphore dit à *compteur*
- Un outil parmi d'autres ...
- Accent mis sur la *définition* et les *propriétés* des sémaphores (*mise en œuvre* examinée dans le chapitre suivant)

Un sémaphore *s* est un objet logique ayant :

- Un attribut (le *compteur*) : un entier noté *s.e* (initialisé à une valeur $s.e_0 \geq 0$) dont la valeur ne peut pas devenir négative
- Deux *primitives d'accès atomiques* : P (ou Wait) et V (ou Signal).

Description des primitives d'accès P et V :

```
typedef struct {int e} sémaphore;
```

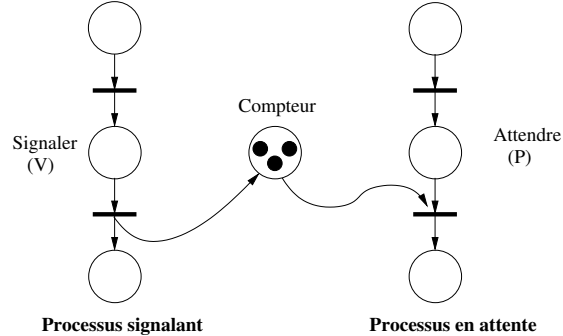
```
void P (sémaphore *s) {
    when (s->e > 0)
        s->e = s->e-1; //Tant que pas vrai, on attend
}
```

```
void V (sémaphore *s) {
    s->e = s->e+1;
```

}

Sémaphore

Représentation graphique (réseaux de Petri)



4.2 Propriétés des sémaphores

Propriétés des sémaphores

Notations (pour un sémaphore s)

- $np(s)$: nombre total d'appels de P (nombre d'entrées dans P)
- $nv(s)$: nombre total d'appels de V (nombre d'entrées dans V)
- $nf(s)$: nombre total de "franchissements" de P (nombre de sorties de P)
- $s.e_0$: valeur initiale du compteur (≥ 0)

Propriété 20 (1). $s.e = s.e_0 - nf(s) + nv(s)$

En effet, $s.e$ vaut initialement $s.e_0$, chaque exécution complète de P le décrémente de 1, chaque exécution de V l'incrémente de 1, et ces opérations se font en exclusion mutuelle

Propriété 21 (2). $s.e$ représente le nombre de processus pouvant "franchir" un $P(s)$ sans blocage

Cette propriété est déduite directement du code de P

Propriété 22 (3). $nf(s) = \min(np(s), s.e_0 + nv(s))$

On a : $nf(s) \leq np(s)$ (on ne peut "franchir" un P sans l'appeler) $nf(s) \leq e_0 + nv(s)$ (initialement e_0 processus peuvent franchir le sémaphore, chaque V autorise un franchissement de plus) donc $nf(s) \leq \min(np(s), s.e_0 + nv(s))$ **Cas 1** : si $nf(s) < np(s)$ il y a au moins un processus bloqué et $s.e = 0$ De (1) on déduit alors que $nf(s) = s.e_0 + nv(s)$ **Cas 2** : si $nf(s) = np(s)$ alors on a : soit $s.e > 0$ et de (1) on déduit $nf(s) < s.e_0 + nv(s)$ soit $s.e = 0$ et de (1) on déduit $nf(s) = s.e_0 + nv(s)$ On a donc exactement $nf(s) = \min(np(s), s.e_0 + nv(s))$

4.3 Exemples de synchronisation

Exclusion mutuelle

- Propriété à assurer : *au plus* un processus en section critique à la fois
- Entrée en section critique peut conduire à un blocage \implies utilisation d'un P
- Valeur initiale du sémaphore = 1 (initialement un processus et un seul peut faire P sans se bloquer)
- Code :

```

sema mutex(1); // valeur initiale du sémaphore mutex est de 1
P(mutex);
<section critique>;
V(mutex);

```

Propriétés de la solution

(On nomme nc le nombre de processus en section critique)

1. *Sûreté* (exclusion) D'après la structure du code, on a $nc = nf(mutex) - nv(mutex)$ D'après (3), on a $nf(mutex) = \min(np(mutex), 1 + nv(mutex))$ ou en reformulant $nf(mutex) \leq 1 + nv(mutex)$ et donc $nc \leq 1 + nv(mutex) - nv(mutex)$, soit $nc \leq 1$
2. *Vivacité* (absence de blocage inutile) Supposons qu'aucun processus ne soit en section critique D'après la structure du code, on a $nc = 0$, ou encore $nf(mutex) = nv(mutex)$ Ceci peut se ré-écrire $nf(mutex) < nv(mutex) + 1$ D'après (3), on a $nf(mutex) = np(mutex)$, donc on n'a pas de processus bloqué
3. *Absence de privation* : non vérifié en l'absence de règle sur le processus libéré par un V (entre autres)

Exclusion mutuelle et interblocage

- L'imbrication de sections critiques peut conduire à une situation où tous les processus sont bloqués indéfiniment, parce qu'aucun processus ne peut effectuer le réveil nécessaire (*interblocage*)

```

sema mutex1(1); {contrôle de la section critique SC1}
sema mutex2(1); {contrôle de la section critique SC2}
processus P1                processus P2
...
P(mutex1); (1)              P(mutex2) (3)
P(mutex2); (2)              P(mutex1) (4)
V(mutex2)                   V(mutex1)
V(mutex1)                   V(mutex2)
...

```

- Si on exécute, dans l'ordre (1), (3), (2), (4) les deux processus sont bloqués définitivement

Exclusion mutuelle et interblocage

- Situation d'interblocage peut se rencontrer dans de nombreux cas d'allocation de ressources
- Situation typique d'interblocage : P en section critique
- L'emboîtement des sections critiques peut être "cachée" par des appels de procédure
- Solutions aux problème d'interblocage vues plus tard

Attente d'événement

- Un processus doit *attendre* qu'un événement extérieur se produise avant de poursuivre son exécution au-delà d'un certain point (exemple : attente de fin d'E/S)
- L'événement attendu peut être arrivé *avant* que le processus ne se mette en attente \implies dans le cas l'arrivée de l'événement doit être *mémorisée*
- Code :

| | |
|--|---------------------------------------|
| <code>sema atEvt(0);</code> | |
| Attente de l'événement | Arrivée de l'événement |
| ... | ... |
| <code>P(atEvt);</code> | <code>{l'événement se produit}</code> |
| <code>{l'événement s'est produit}</code> | <code>V(atEvt);</code> |
| ... | ... |

5 Interface d'un système d'exploitation

5.1 Compléments de terminologie : processus vs thread

Lien entre gestion du processus et gestion de la mémoire \implies deux notions de "processus" :

- *Thread* (processus *léger*)
 - Simple fil de contrôle (suite d'exécution d'instructions)
 - Partage de l'espace mémoire utilisateur (espace d'adressage) par un ensemble de threads (voire tous)
 - Pas de protection entre threads associés au même espace d'adressage
 - Processus *lourd*
 - Entité associant un fil de contrôle et diverses ressources (mémoire, fichiers)
 - Protection entre processus
 - Exemple : processus UNIX
- \implies Clarification des différences en SGM

5.2 L'interface UNIX (très partielle)

Processus lourds : gestion de processus

- `pid_t fork(void)`; duplication du processus. Retourne 0 dans le fils, l'identificateur du fils dans le père
- `int execl(char *path, char *arg, ..., 0)`. Exécute un nouveau programme (écrase code et données du programme en cours)
- `pid_t wait(int *status)`; Attente de la terminaison d'un processus fils
- `void exit(int status)`; Terminaison du processus (0=terminaison normale).

Processus lourds : sémaphores

- `int semget(key_t key, int nsems, int flag)`; Création d'un sémaphore
- `int semop(int semid, struct sembuf *array, size_t nops)`; Opérations sur groupe de sémaphores (P/V). La valeur à ajouter/retirer au compteur et le sémaphore concerné dans le groupe sont des arguments de l'appel système (array)
- `int semctl(int semid, int semnum, int cmd, ...)`; Opérations de contrôle sur le sémaphore (récupération compteur, destruction)

Processus légers (threads) : gestion de processus

- `int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)`; création d'un nouveau thread
- `int pthread_detach(pthread_t thread)`; marque le thread comme terminé (destruction à la terminaison seulement)
- `void pthread_exit(void *value_ptr)`; terminaison du thread courant

- int pthread_join(pthread_t thread, void **value_ptr) : attente de la terminaison d'un thread spécifique
- int pthread_cancel(pthread_t thread) : annulation d'un thread
- Et beaucoup d'autres ... (modification attributs, ordonnancement, etc.)

Processus légers (pthreads) : synchronisation

- Sémaphores d'exclusion mutuelle
- Sémaphores lecteurs/rédacteurs
- Variables de condition

Troisième partie

Eléments de mise en œuvre d'un noyau de synchronisation

1 Mécanismes de contrôle de l'exécution

Mécanismes de contrôle de l'exécution

- Mécanismes *minimaux* pour mettre en œuvre un système d'exploitation
 - *Mécanismes de protection*
 - *Commutation du contexte d'exécution* : passage d'un processus à l'autre
- Les processeurs peuvent être dotés de mécanismes *plus sophistiqués*

1.1 Mécanismes de protection

- Système multi-usager \implies on ne peut pas laisser un usager utiliser toutes les possibilités de la machine
- Mécanismes de protection reposent au final sur des *mécanismes matériels*
 \implies Mécanismes de protection portant sur :
 - Le contrôle de l'accès à la mémoire (voir deuxième semestre)
 - Le contrôle de l'utilisation des instructions

Mode d'exécution utilisateur et système

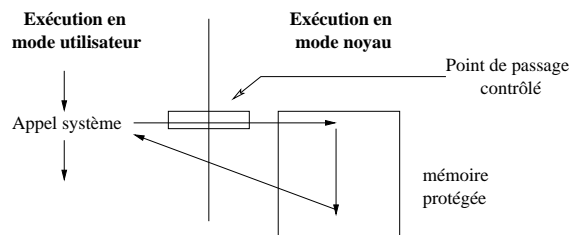
Mécanisme de protection le plus simple pour contrôler l'utilisation des instructions : deux *modes d'exécution* pour le processeur :

- Le mode *noyau* (ou "maître", "superviseur"...) dans lequel l'exécution de toutes les instructions est possible
- Le mode *utilisateur* (ou "asservi"...) dans lequel l'exécution de certaines instructions (et l'accès à certaines zones mémoire et registres d'accès aux contrôleurs d'entrées/sorties) est interdit.
- Mode d'exécution courant stocké dans le *mot d'état* du processeur, ainsi que d'autres informations décrivant l'état du processeur (masque d'IT, code condition, ...)

Changement de mode d'exécution : appels système

- Réalisation de certaines opérations (E/S, etc) nécessite de passer en mode noyau
- Passage en mode noyau doit être contrôlé
 \implies Instruction spécialisée de passage en mode noyau, *appel système* (*int* en x86, *syscall* en MIPS, *trap* en 68k)

1. Passage en mode noyau
2. Branchement à une adresse fixée



1.2 Commutation de contexte

Contexte d'exécution du processeur

- Informations accessibles par le processeur à un instant donné, stockées dans ses *registres*

- Mot d'état (SR)
- Compteur ordinal (PC)
- Sommet de pile (SP), pointeur de frame (FP),
- Registres généraux (entiers, flottants, etc).

Pourquoi la commutation de contexte ?

- Plusieurs processus pour un seul processeur, un seul jeu de registres pour l'unique processeur
- Quand on suspend un processus pour le reprendre par la suite, on veut retrouver le *même contexte d'exécution* du processeur qu'à sa suspension
- ⇒ Opération de *commutation de contexte*
 1. *Sauvegarde* du contexte d'exécution courant
 2. *Mise en place* d'un nouveau contexte d'exécution
- Remarque : la mémoire n'est pas sauvegardée, a priori non partagée

Où sauvegarder le contexte d'exécution ?

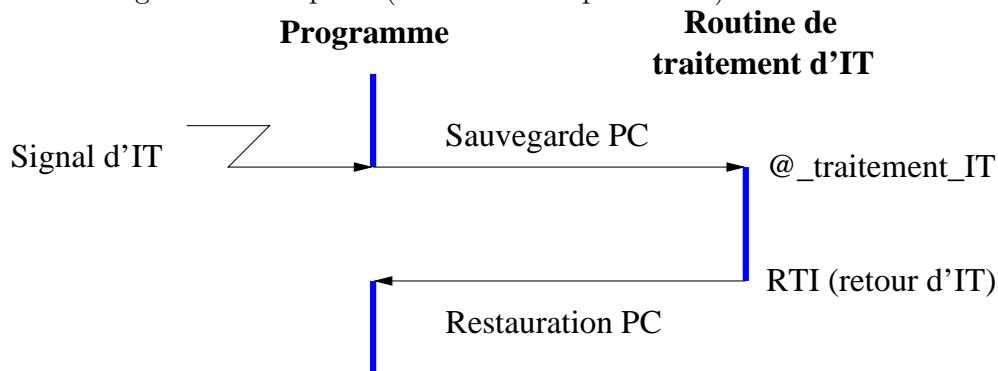
- Sauvegarde en mémoire
- On doit être capable de retrouver le contexte au redémarrage du processus
 - Pile d'exécution du processus (+ lien pour retrouver le sommet de pile)
 - Descripteur du processus

1.3 Mécanismes provoquant une commutation de contexte

Definition 23 (Exception). On nomme *exception (trappe)* tout mécanisme provoquant une commutation de contexte (occasionnant l'exécution d'une séquence de code extérieure au programme courant avec retour possible ultérieur)

Types d'exceptions : interruptions

- Cause *externe* au programme, *non demandée* explicitement
- Un niveau d'interruption (IT) peut être :
 - *Armé/désarmé (activé/désactivé)* : suppression de la source de l'interruption (signal d'interruption)
 - *Masqué/démasqué* : configuration du processeur pour qu'il n'effectue pas de déroutement en cas de signal d'interruption (mot d'état du processeur)



- Contexte minimal sauvegardé en cas d'interruption (PC, SR).

⇒ Le reste du contexte doit être sauvegardé par logiciel (si nécessaire)

Types d'exceptions : exceptions matérielles

- Cause *interne* au programme, *non demandée* explicitement
- Détection d'une situation d'exécution anormale
- Exemples
 - Division par zéro, débordement
 - Accès à une zone mémoire invalide
 - Exécution d'une instruction interdite
- Routine de traitement associée : remédie à l'anomalie ou le signale à l'utilisateur

Types d'exceptions : appels système

- Cause *interne* au programme, *demande explicite*
- Passage contrôlé en mode système, schéma d'exécution associé
 1. Sauvegarde du contexte
 2. Exécution de l'appel système
 3. Restauration du contexte

Remarques 24. - *Quel contexte est restauré en (3) ? Dépend de l'appel système réalisé (bloquant, non bloquant)*

- *Contexte restauré au retour d'une interruption ? Processus interrompu ou autre processus ? Dépendant de la mise en œuvre du noyau*

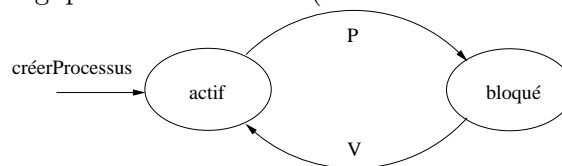
2 Mise en œuvre du noyau de synchronisation

2.1 Représentation des processus

Etats d'un processus

Etats logiques d'un processus en faisant abstraction du partage du processeur entre processus :

- *Actif* : peut logiquement s'exécuter et s'exécute car il dispose d'un processeur dédié pour s'exécuter
- *Bloqué* : ne peut pas logiquement s'exécuter (attente liée à une synchronisation)



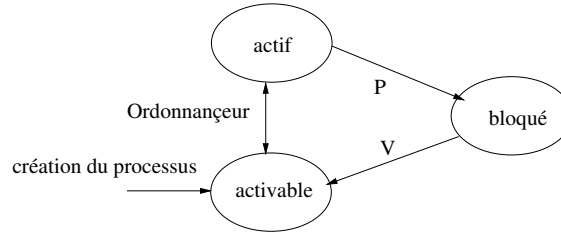
Etats d'un processus en supposant *un seul* processeur réel :

⇒ Un seul processus peut être actif à un instant donné ⇒ découpage de l'état actif en deux sous-états

- *Actif* : peut logiquement s'exécuter et s'exécute car il dispose à cet instant du processeur réel
- *Activable* (éligible) : peut logiquement s'exécuter mais ne s'exécute pas car ils ne dispose pas à cet instant du processeur réel
- *Bloqué* : ne peut pas logiquement s'exécuter

⇒ Problème de déterminer à qui attribuer le processeur (ordonnanceur du système d'exploitation)

Etats d'un processus en supposant *un seul* processeur réel :



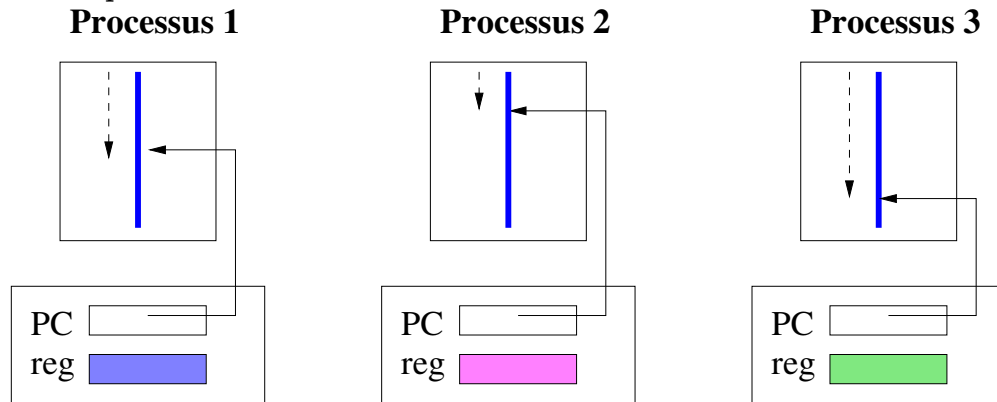
(tels qu'affichés par la commande *ps* sous Mac OS X)

- I (Idle) : endormi depuis plus de 20 secondes
- R (Ready) : actif
- Re (Runnable) : activable
- S (Sleeping) : endormi depuis moins de 20 secondes
- U : dans une attente non interrompible
- Z (Zombie) : terminé alors que son père ne l'est pas (ne consomme pas de ressource, juste un descripteur)

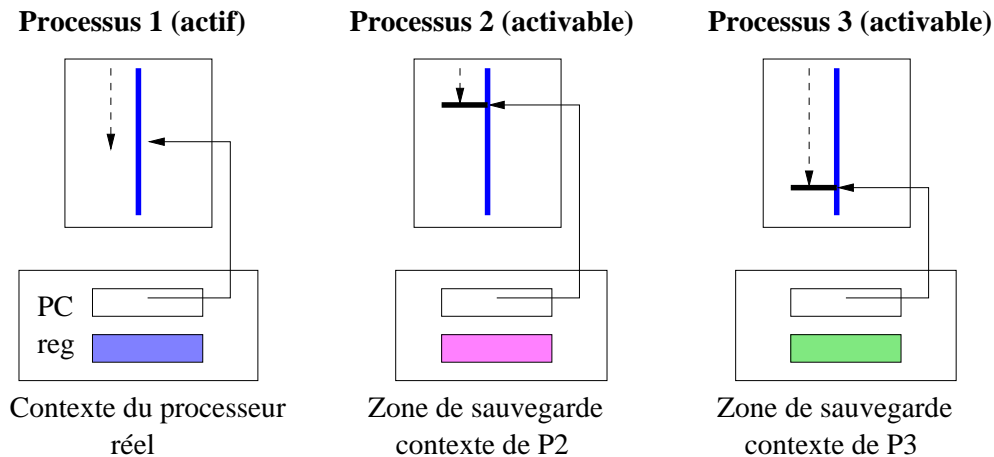
Contexte d'un processus

- Vision abstraite : le contexte d'exécution d'un processus est constitué des registres du processeur (virtuel) l'exécutant
- Vision concrète (un seul processeur réel) :
 - Le contexte d'exécution du processus actif est installé dans les registres du processeur réel. C'est alors le processeur réel lui-même qui le fait évoluer.
 - Le contexte d'exécution d'un processus activable ou bloqué n'évolue pas (le processus ne s'exécute pas). Mais si on veut pouvoir reprendre ultérieurement son exécution il faut avoir conservé quelque part une image de ce contexte.

Contexte d'un processus : vision abstraite



Contexte d'un processus : vision concrète

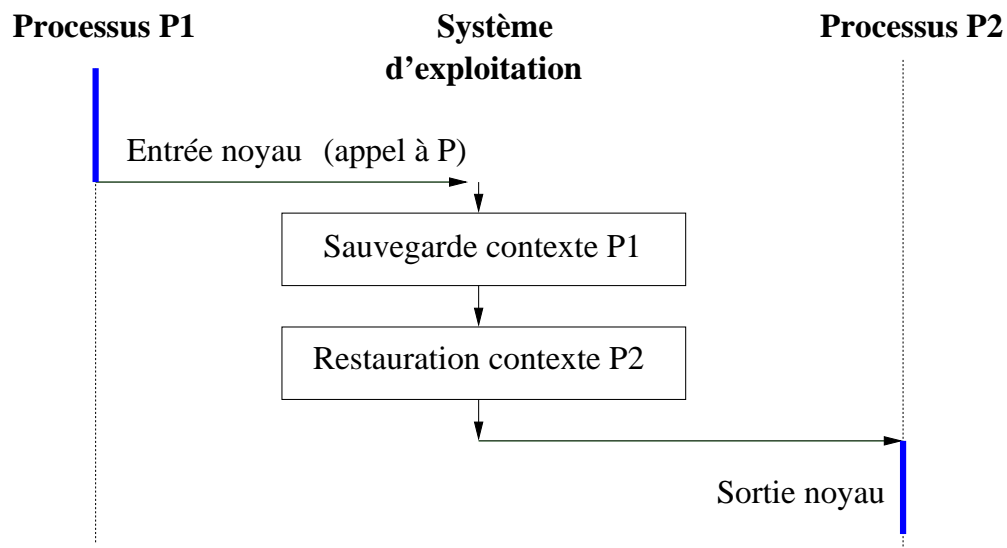


Commutation de contexte

Association allocation/désallocation du processeur et sauvegarde/restauration du contexte d'un processus :

- Perte du processeur par un processus \Rightarrow sauvegarde de son contexte d'exécution (registres du processeur \rightarrow zone de sauvegarde du contexte)
- Allocation du processeur \Rightarrow installation de son contexte dans les registres du processeur réel (zone de sauvegarde du contexte \rightarrow registres du processeur)

Exemple : commutation de contexte lors d'un P bloquant



Exemple 25.

Descripteur de processus

- Un *descripteur* par processus (PCB, Process Control Block), contenant :
 - les caractéristiques propres du processus (type, priorité...)
 - des caractéristiques décrivant son état d'exécution (état, temps CPU utilisé...)
 - la zone de sauvegarde de son contexte (ou son adresse)

- Descripteurs *chaînés* entre eux pour regrouper les processus de caractéristiques identiques (file des processus activables, des processus bloqués sur un sémaphore, etc)
- La forme de la structure de données dépendra de l'ordonnancement mis en œuvre (par priorité, FIFO, etc.)

2.2 Mise en œuvre de l'exclusion mutuelle

- Variables partagées entre plusieurs processus \implies manipulation en *exclusion mutuelle*
- Ce problème existe au sein du noyau et non pas seulement au niveau de programmes utilisateur (files de descripteurs de processus, etc)
- Besoin d'un mécanisme de *bas niveau* pour l'exclusion mutuelle en complément des sémaphores
 - Problème de *récurtivité* : on ne peut pas rendre indivisible les opérations P et V sur les sémaphores en utilisant les sémaphores eux-mêmes
 - Mécanisme *trop lourd* pour être utilisé dans un noyau, ou les sections critiques sont souvent nombreuses et de courte durée

a. Masquage des interruptions

- Permet au processeur d'ignorer (temporairement) les interruptions pendant une période
- Pas d'interruption \implies pas de déroutement de l'exécution sur le processeur \implies le processus garde le processeur pour lui seul
- Code

```

...
masquer_IT;                (1)
section critique;          (2)
démasquer_IT;              (3)
...                          (4)

```

- Solution très *rapide* \implies intéressante à un niveau très bas dans le système
- Non généralisable :
 - Aux sections critiques *longues* : risque de *perte d'interruptions*
 - Aux processus utilisateur : problème de *sécurité* (oubli de démasquer ou démasquage tardif \implies risque de compromettre des E/S en cours pour *tout* le système)
 - Aux systèmes *multi-processeurs* : le masquage des IT ne concerne que le processeur sur lequel l'instruction de masquage est exécutée

b. Solutions algorithmiques

- Utilisation de *variables partagées* entre processus
- Basées sur la propriété *d'exclusivité d'accès* à un emplacement mémoire
- Séquence d'entrée en section critique : *attente active* (attente des conditions lui permettant d'entrer en section critique, tout en conservant le processeur)

b. Solution algorithmique triviale

```

char occup=0; { indique si un processus est en SC}
while (occup);
occup=1;

```

<section critique>;

occup=0;

- *Ne fonctionne pas* Si deux processus essaient d'entrer simultanément en section critique, les deux voient *occup* à *faux* et entrent en section critique
- Le problème vient de l'absence d'atomicité de la séquence test-modification de *occup*

b. Algorithme de Peterson

- Solution *simple* fonctionnant dans le cas de *deux* processus
- Correction basée sur la propriété *d'exclusivité d'accès* à un emplacement mémoire

```
int turn;
```

```
char interested[2] := {0, 0};
```

```
void entrer_sc (int p); // p = n° processus demandeur
```

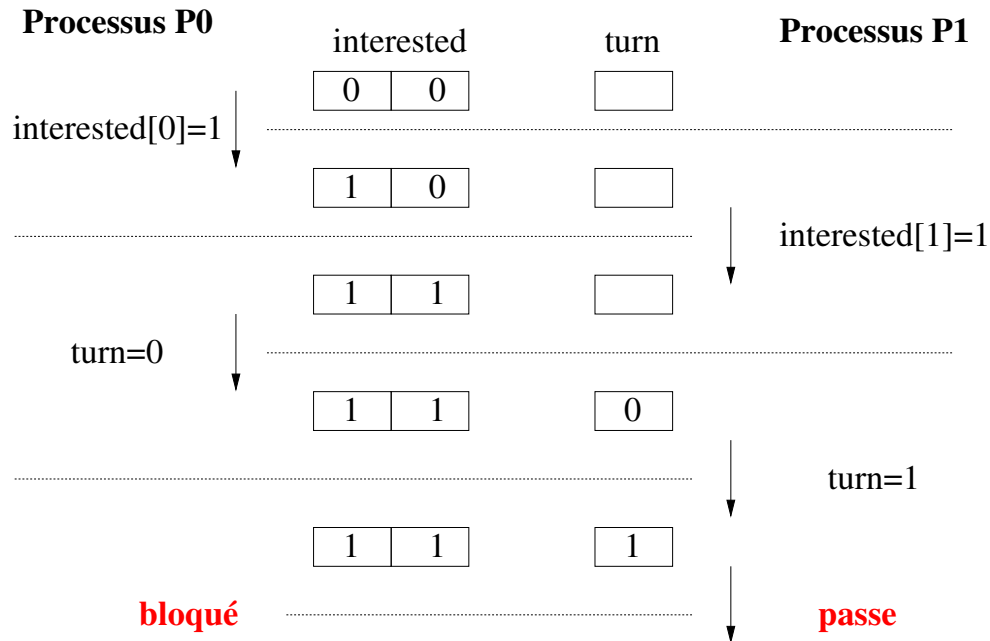
```
{  
    interested[p]=1;           (1)  
    turn=p;                   (2)  
    while ((turn==1-p) && (interested[1-p])); (3)  
}
```

```
void sortie_sc (int p); //p = n° processus demandeur
```

```
{  
    interested[p]=0;  
}
```

- La variable *interested*
 - Sert à enregistrer toutes les demandes d'accès (*interested[i]* uniquement manipulée par le processus *i*)
 - Indique que le processus *i* est en section critique
 - Un élément *interested[i]* par processus \implies pas de risque d'écrasement de la demande par l'autre processus
- La variable *turn* sert à gérer l'entrée simultanée en section critique des deux processus (départager les ex-aequo)

Exécution avec entrée quasi-simultanée en section critique



b. Solutions algorithmiques

- Généralisables à N processus, si N connu
- Difficiles à utiliser à un niveau bas du système (N connu, consommation mémoire)
- A un niveau plus élevé, c'est l'attente active qui n'est plus acceptable (monopolisation du processeur)

c. Solutions basées sur un support matériel

- Solution algorithmique triviale


```
char occup=0; // indique si un processus est en SC
while (occup);
occup=1;
<section critique>;
occup=0;
```
- N'est pas correcte parce que la séquence (consultation - modification du booléen occup) *n'est pas indivisible*

⇒ *Instructions spéciales* assurant de manière *indivisible*

- Une consultation (test)
- Une modification (set) d'un emplacement mémoire
- Un accès exclusif au bus
- Exemples : TAS (68k), XCHG (x86) (TAS = Test And Set)

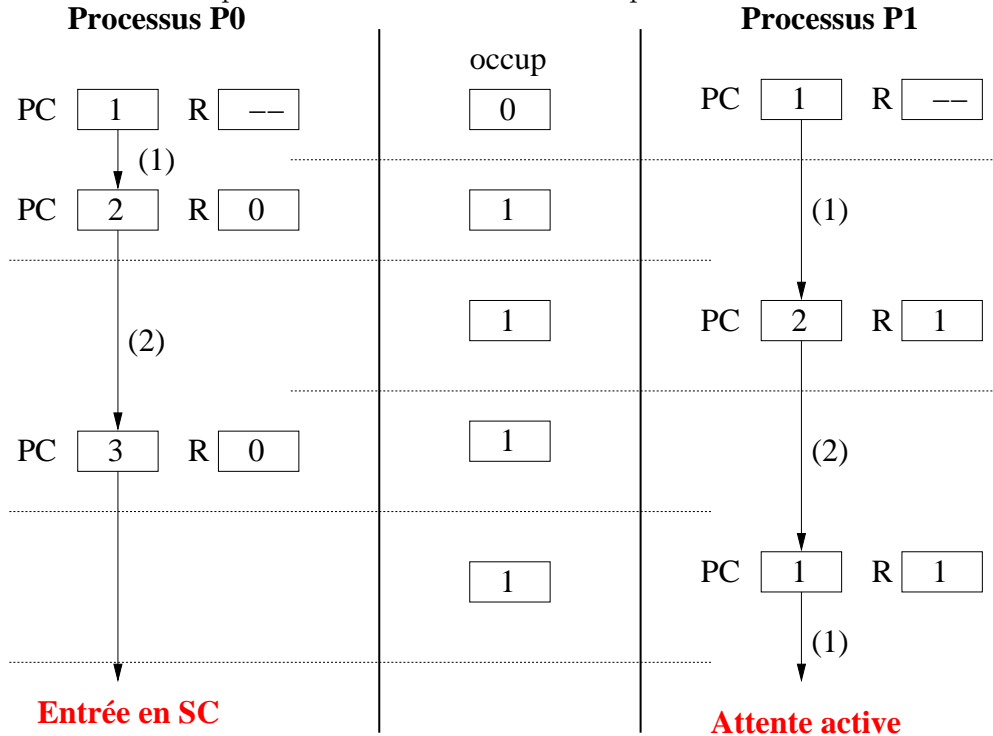
Instruction TAS :

- Syntaxe : TAS R,ad
- Fonctionnement :


```
R := MEM[ad];
MEM[ad] := 1;
```

- Code d'entrée en section critique
 test : tas R,occup; (1)
 if R==1 goto test; (2)
- Code de sortie de section critique
 occup= 0;

Exécution avec entrée quasi-simultanée en section critique



Bilan des solutions de synchronisation utilisées

- Solutions basées sur une *attente active* (solutions algorithmiques ou support matériel)
 - consommation de temps processeur lors de la phase d'attente
 - ⇒ mauvaise utilisation du processeur
 - ⇒ Routines de synchronisation tels que les sémaphores, pour lesquels on relâche le processeur pendant les phases d'attente : *attente passive*
- Solutions de bas niveau (IT, solutions par attente active) utilisées pour assurer la propriété d'atomicité des primitives P et V sur les sémaphores

2.3 Mise en œuvre des sémaphores

- Objectif : ne pas *monopoliser le processeur* par un processus ne pouvant plus progresser (bloqué)
- Principe : chaînage du descripteur de processus dans une *file d'attente* associée au sémaphore (blocage = mise en file)
- Représentation d'un sémaphore :
 - Entier associé au sémaphore (compteur)
 - File d'attente : liste des descripteurs de processus bloqués sur le sémaphore

```

desc_processus *actif; // processus actif
file desc_processus *activables; // processus prêts

```

```

void P (semaphore s)
{
    if (s.e == 0) {
        entrer(actif, s.f); // mise en attente
        actif = nil; // processus bloqué
    }
    // on a forcément s.e > 0
    s.e = s.e-1;
}

```

```

void V (semaphore s)
{
    descr_processus *pr;
    s.e = s.e+1;
    if (s.f non vide) {
        pr=sortir(s.f);
        entrer(pr, activables);
    }
}

```

Mise en œuvre des sémaphores (alternative)

```

void P (sémaphore s)
{
    s.e = s.e-1;
    if (s.e < 0) {
        entrer(actif, s.f);
        actif := nil;
    }
}

```

Signification de l'attribut $s.e$:

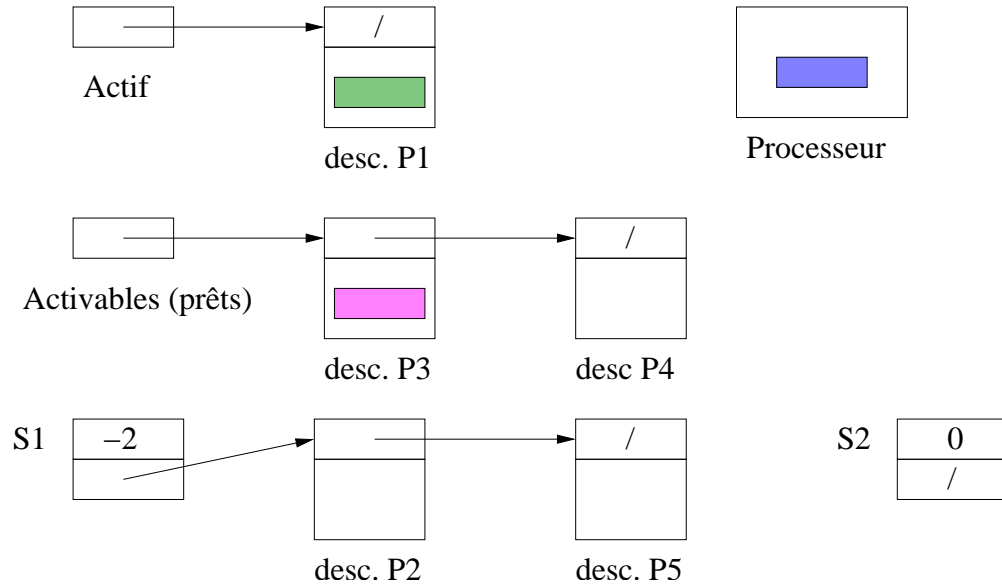
- Si $s.e \geq 0$, $s.e$ égal au nombre de processus pouvant faire un P sans se bloquer
- Si $s.e < 0$, $s.e$ est égal au nombre de processus bloqués
- On a toujours $s.e = s.e_0 - np(s) + nv(s)$
- C'est cette mise en œuvre qui sera *utilisée en TP* (et dans de nombreux systèmes)

Mise en œuvre des sémaphores (alternative)

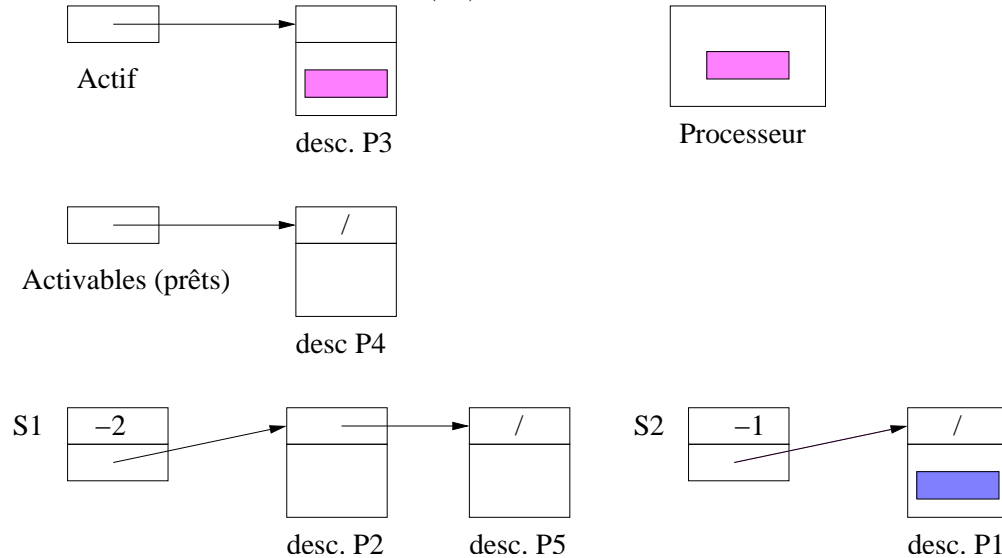
- Exemple 26.* – Cinq processus P1, P2, P3, P4, P5
- P1 est actif
 - P3 et P4 sont activables
 - P2 et P5 sont bloqués sur le sémaphore S1

- Le sémaphore S2 est initialisé à 0

Exemple 27. Etat initial :



Exemple 28. Après exécution par P1 de P(S2) et allocation du processeur à P3 :

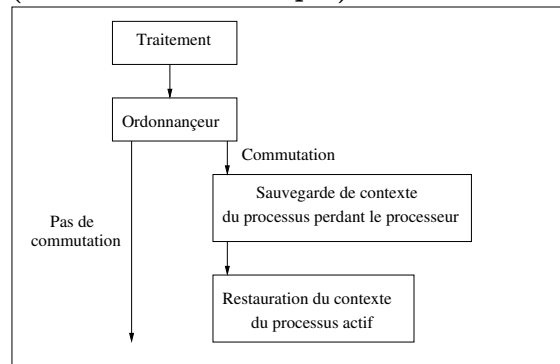


2.4 Architecture du noyau

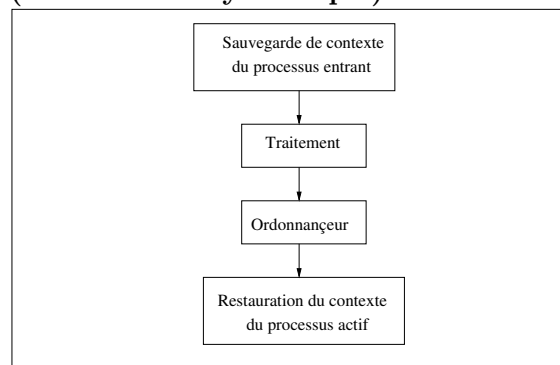
Grandes parties du noyau de gestion de processus :

- *Sauvegarde du contexte* du processus perdant le processeur
- Réalisation proprement dite des *primitives de la machine virtuelle* (P, V, créer_processus, etc)
 - ⇒ manipulation des structures de données (descripteurs de processus, de sémaphores,...)
- Choix du processus à qui on va allouer le processeur (*ordonnançeur*)
- Mise en place du contexte du processus à qui on a alloué le processeur
 - ⇒ Différentes manières d'articuler ces grandes parties

Architecture du noyau (architecture classique)



Architecture du noyau (architecture symétrique)



3 Allocation de l'Unité Centrale (ordonnancement)

- Rôle : Choisir parmi les processus activables celui à qui attribuer le processeur
- Critères possibles de jugement d'une stratégie d'allocation :
 - *Équité* : les processus se partagent "à part égale" le temps processeur
 - *Efficacité* : le processeur est utilement actif le plus souvent possible
 - *Temps de réponse* : on veut minimiser le temps d'attente de l'utilisateur
 - *Débit* : on veut maximiser le nombre de travaux traités par le système pendant un temps donné

3.1 Prémption

Definition 29 (Prémption). – Un ordonnancement est dit *préemptif* quand un processus peut perdre *involontairement* le processeur sans exécuter aucune primitive de synchronisation bloquante

- Au contraire, un ordonnancement est *non préemptif* quand un processus ne perd le processeur que quand il exécute *volontairement* une primitive de synchronisation bloquante

Mécanismes impliquant de la prémption

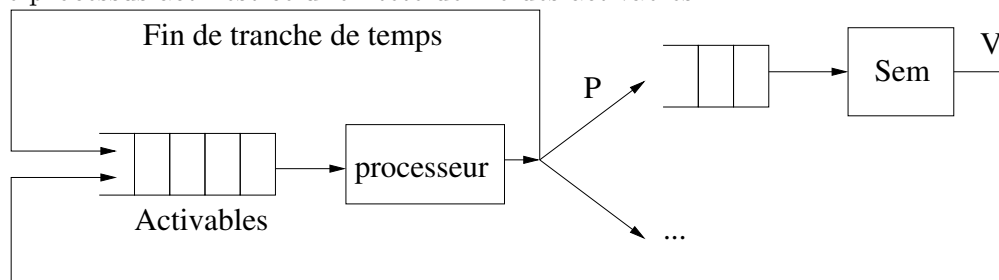
- *Partage de temps*

- on alloue le processeur à un processus pour un quantum de temps q , si au bout de q unités de temps, ce processus ne s'est pas bloqué, on lui enlève le processeur
- Peut être facilement mis en œuvre grâce à l'horloge de la machine
- Allocation sur la base de *priorités* : on attribue à chaque processus une priorité et c'est le processus le plus prioritaire qui devient actif

3.2 Politiques d'allocation

a. Stratégie du tourniquet (round-robin)

- Partage de temps, sans priorité
- File des activables gérée en FIFO (First-In First-Out) :
 - Un processus qui termine sa tranche de temps ou qui est débloqué est mis à la fin de la file des activables
 - Le processus actif est celui en tête de file des activables



b. Stratégies à base de priorité

Motivations à l'utilisation des priorités :

- Certains *processus système* doivent impérativement s'exécuter rapidement, sous peine d'inefficacité (ex : on n'utilise pas au mieux un périphérique)
- L'exécution de certains processus n'est pas pénalisante pour le temps de réponse des autres utilisateurs (processus *faisant beaucoup d'E/S*)
- Un utilisateur donnant des commandes *courtes à traiter* supporte moins d'attendre qu'un usager lançant un *gros calcul*

Principe du *mécanisme* d'ordonnancement par priorités :

- Chaque processus a une priorité
- On attribue le processeur au processus activable le plus prioritaire

Classification :

- Priorité *statique* : priorité affectée un fois pour toute au processus
- Priorité *dynamique* : recalculée au cours de la vie du processus en fonction de divers critères

Exemples de *politiques* d'attribution des priorités :

- Priorité fonction du *type* des processus (système/usager...)
- Priorité calculée de manière à pénaliser les processus occupant *beaucoup* le processeur.
- Priorité calculée de manière à *équilibrer* l'utilisation du processeur.
- Priorité calculée en fonction de la *date limite d'exécution* d'un processus (temps-réel)

c. Exemple : ordonnancement UNIX

- Priorités *dynamiques*

- Un processus peut s'exécuter en mode utilisateur ou noyau, préemption en mode utilisateur seulement
- Priorité = entier (petite valeur \implies très prioritaire), une file par priorité
- Compteur d'utilisation du processeur par processus (Cuc).
- Politique d'attribution des priorités :
 - Un processus qui termine sa tranche de temps perd le processeur, et conserve sa priorité;
 - Un processus utilisateur qui se bloque dans le noyau (attente d'E/S...) se voit attribuer une priorité négative (il devient plus prioritaire);
 - A intervalle régulier (1 seconde), les priorités des processus utilisateurs sont recalculées :
 $\text{nouvelle_priorité} = \text{base} + \text{Cuc}/2$. Ce qui pénalise les processus ayant beaucoup utilisé le processeur.

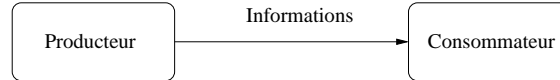
Quatrième partie

Synchronisation entre processus : compléments

1 Schémas classiques de synchronisation

1.1 Producteur/consommateur

- Un processus, le *producteur*, calcule des informations, qui sont utilisées par l'autre processus, le *consommateur*



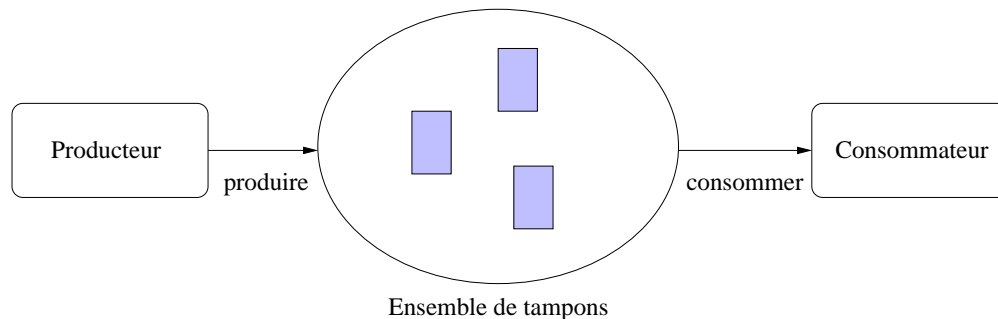
Contraintes de synchronisation

Adaptation vitesse du consommateur à celle du producteur : le consommateur ne peut pas aller plus vite que le producteur

Contrainte 30 (1). Le consommateur ne peut s'exécuter que s'il y a des informations produites et non consommées

- Le producteur peut aller plus vite que le consommateur \implies pour ne pas le freiner, on mémorise l'information dans un *ensemble de tampons* (de nombre borné)
- Un tampon est *plein* s'il contient une information produite et pas encore consommée, sinon il est *vide*

Contrainte 31 (2). Le producteur ne peut s'exécuter que s'il a au moins un tampon vide



Procédures d'accès aux tampons :

- *produire(d)* permet de recopier une information *d* dans un tampon vide et de passer au tampon suivant, le tampon passe alors dans l'état plein (résultat indéterminé s'il n'existe pas de tampon vide) ;
- *consommer* permet de rendre en résultat le contenu d'un tampon plein et de passer au suivant, ce qui a pour effet de "vider" le tampon
- Ces deux procédures ne comportent *pas de synchronisation*, elles permettent juste de masquer la gestion des tampons (listes, tableaux).

processus producteur

```
{  
  info d;  
  while (1) {  
    d = calcul;  
    attendre "tampon vide";  
    produire(d);  
  }  
}
```

processus consommateur

```
{  
  info d;  
  while (1) {  
    attendre "tampon plein";  
    d = consommer;  
    calcul(d);  
  }  
}
```

} }

Expression des contraintes de synchronisation en utilisant les sémaphores :

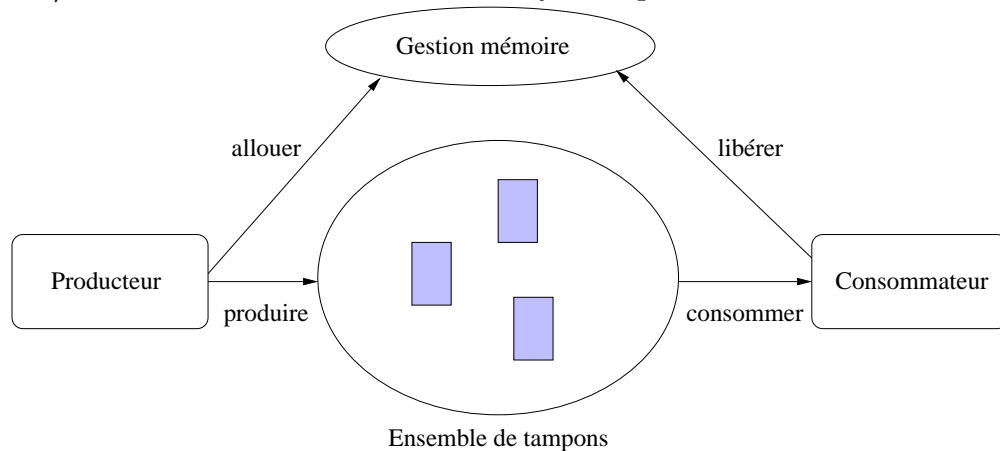
- Condition pour produire : $nb\ tampon\ vide > 0$
 - Or, $nb\ tampon\ vide = nb\ tampon - nb\ production + nb\ consommation$
 - Formule analogue à celle donnant la valeur du compteur d'un sémaphore s : $s.e = s.e0 - nf(s) + nv(s)$
 - \Rightarrow Un sémaphore $nbvide$ compte le nombre de tampons vides
 - \Rightarrow On effectue un P sur ce sémaphore avant chaque production
 - \Rightarrow On effectue un V sur ce sémaphore après chaque consommation
- Condition pour consommer : $nb\ tampon\ plein > 0$
 - Un raisonnement similaire amène à utiliser un sémaphore $nbplein$ qui compte le nombre de tampons pleins
 - \Rightarrow P sur ce sémaphore avant chaque consommation
 - \Rightarrow V sur ce sémaphore après chaque production

```
sema nbplein(0);
sema nbvide(nb_tampons);
```

```
processus producteur
{
  info d;
  while (1) {
    d = calcul;
    P(nbvide);
    produire(d);
    V(nbplein);
  }
}

processus consommateur
{
  info d;
  while (1) {
    P(nbplein);
    d = consommer;
    V(nbvide);
    calcul(d);
  }
}
```

Producteur/consommateur avec allocation dynamique



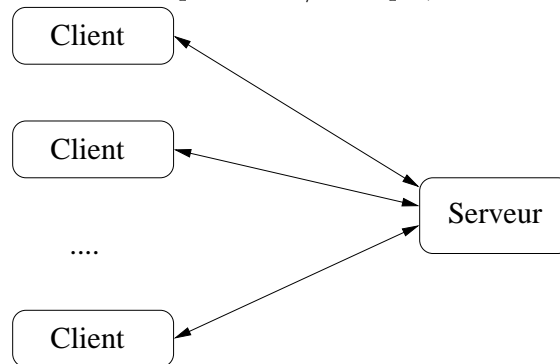
```
sema nbplein (0);
```

```
processus producteur
{
  info d;
  while (1) {
    d = calcul;
    allouer;
    produire(d);
    V(nbplein);
  }
}
```

```
processus consommateur
{
  info d;
  while (1) {
    P(nbplein);
    d = consommer;
    calcul(d);
    libérer;
  }
}
```

1.2 Client/serveur

- Ensemble de processus (*clients*) qui demandent un travail à un processus spécialisé (*serveur*)
- Type de travail : traitement de requêtes d'E/S disque, allocation de ressource, etc.



- Demande de service = *requête*
- En général, implique une *réponse* du serveur
 - Résultat du service (exemple bloc disque lu en cas de requête de lecture disque)
 - Simple acquittement indiquant que le service a été accompli

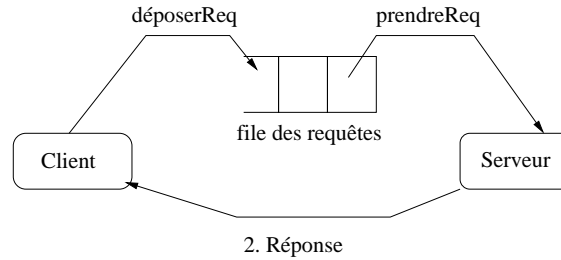
1. Requête



2. Réponse

- *Bloc de requête* contenant à la fois
 - La description du travail demandé (requête)
 - La réponse du serveur (vide au départ)
- *File des requêtes* : mémorisation des demandes de services demandées et pas encore traitées
 - Procédures d'accès *déposerReq* et *prendreReq*
 - Ces procédures assurent que la file est accédée en *exclusion mutuelle*

- Pas de capacité de stockage maximale dans la file de requête
- Clients et serveur se partagent le même espace mémoire



Contraintes de synchronisation

Contrainte 32 (1). Le serveur ne peut s'exécuter s'il n'y a pas de requête à traiter

Contrainte 33 (2). Le client ne peut pas utiliser la réponse tant que le serveur ne la lui a pas communiquée

- Contrainte (1)
 - Sémaphore de comptage des requêtes, *nbreq*
 - P lors du retrait par le serveur
 - V lors du dépôt par le client
- Contrainte (2)
 - Il faut réveiller *le* client qui a demandé le service
 - On utilise donc un sémaphore *par client*
 - Valeur initiale du sémaphore = 0 (sémaphore *bloquant* : initialement, le serveur n'a pas encore rendu le service)
 - Le serveur doit avoir accès au sémaphore d'attente de tous les clients. Une solution consiste (sans fixer le nombre de clients) à transmettre le sémaphore dans le bloc de requête

```
typedef { demande d, sema attRep(0), reponse r } blocReq;
sema nbreq (0); sema mutexFreq (1); file blocReq fileReq;
void deposerReq (pointeur blocReq ptb) {
    P(mutexFreq);
    mettre le bloc de requête repéré par ptb dans fileReq;
    V(mutexFreq);
}
blocReq *prendreReq (void) {
    P(mutexFreq);
    enlever un bloc de fileReq et rendre son adresse;
    V(mutexReq);
}
```

```
processus client {
    blocReq rq;
    ...
    rq.d = initialisation de la requête;
    deposerReq(&rq);
```

```

V(nbreq);
... (1)
P(rq.attRep);
exploiter la réponse(rq.r);
...
}

processus serveur {
  blocReq *ptb;
  while (1) {
    P(nbreq);
    ptb :=prendreReq;
    traitement requête(ptb → d);
    ptb → r = reponse;
    V(ptb → attRep);
  }
}

```

Version “synchrone” du client

```

procedure faireTraiterReq (blocReq *ptb) {
  deposerReq(ptb);
  V(nbreq);
  P(ptb → attRep);
}
processus client {
  blocReq rq;
  ...
  rq.d = initialisation de la requête;
  faireTraiterReq(&rq);
  exploiter la réponse(rq.r);
  ...
}

```

1.3 Sémaphores privés

Definition 34 (Sémaphore privé). Un *sémaphore privé* est un sémaphore sur lequel un seul processus (le propriétaire) peut faire un P.

- Intérêt : quand on fait un V on sait exactement quel processus on réveille (le processus propriétaire du sémaphore)
- ⇒ Il est possible d’exprimer des contraintes de synchronisation complexes, délicates à écrire directement avec des sémaphores “standard”

2 Autres outils de synchronisation

2.1 Messages

Communication par messages

- Associe *synchronisation* et *transfert d'information*
- Interface typique :
 - **void** envoyer (**identité** destinataire, **message** m) provoque l'émission du message *m* vers l'entité désignée par *destinataire*, elle est *non bloquante* (\implies stockage des messages dans une *file*)
 - **void** recevoir (**var identité** source, **message*** m) rend un message et l'identité de l'entité émettrice, *bloque* le processus demandeur s'il n'a pas de messages émis

a. Exemples d'utilisation des envois de messages

Client/Serveur :

```
processus client {
    requête rq;
    réponse r;
    identité id;
    ...
    rq =init requête;
    envoyer(serveur, rq);
    recevoir(id, &r);
    exploiter la réponse(r)
    ...
}

processus serveur {
    requête rq;
    réponse r;
    identité id;
    while (1) {
        recevoir(id, &rq);
        traiter requête(rq);
        r = calcul réponse;
        envoyer(id, r);
    }
}
```

Producteur/consommateur

```
processus producteur {
    info m;
    while (1) {
        m =calcul;
        envoyer(consommateur,m);
    }
}

processus consommateur {
    info m;
    while (1) {
        recevoir(producteur,&m);
        calcul(m);
    }
}
```

b. Remarques sur la communication par envoi de messages

- *Boîte à lettres* vs *Port*
 - *Boîte à lettres* : file par processus. Dans ce cas, *source* et *destinataire* désignent le processus
 - *Port* : file par port, ports créés selon les besoins. Permet de distinguer les sources de messages, et éventuellement d'attendre un message d'une source particulière. Dans ce cas, *source* et *destinataire* désignent le port

- *Gestion mémoire* : recopies de messages lors de l'envoi d'un message. Une mise en œuvre plus efficace consiste à transmettre l'adresse du message plutôt que de recopier leur contenu. Mais risque de problème car l'envoi n'est pas bloquant. Copy-on-write (voir second semestre)

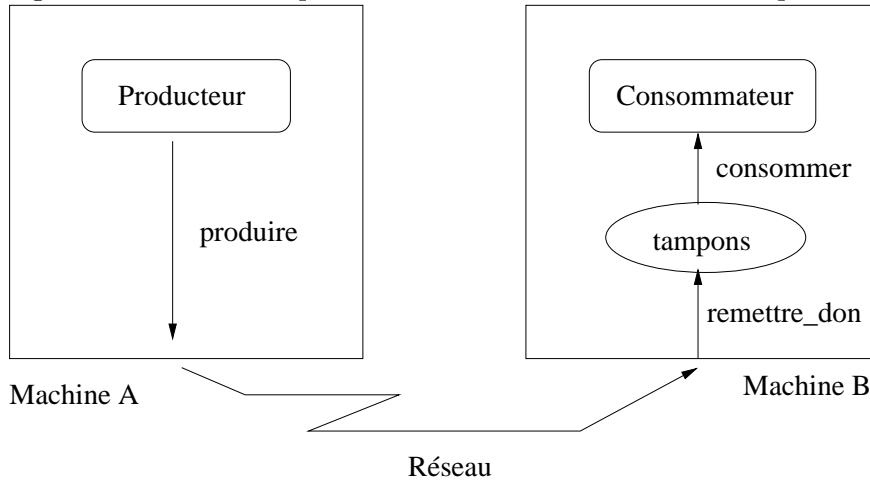
b. Interface POSIX. Message queues

- `int msgget(key_t key, int msgflg)`; Création d'un file de messages
- `int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg)`; Envoi d'un message sur une file
- `ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg)`; Réception d'un message dans une file
- `int msgctl(int msqid, int cmd, struct msqid_ds *buf)`; Opération de contrôle sur une file de message (destruction, consultation taille de file, etc)

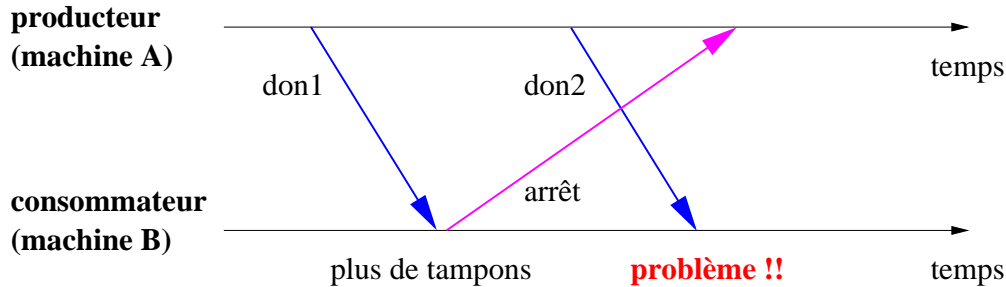
c. Extension aux systèmes distribués

- S'étend naturellement très bien à des processus situés sur des processeurs ne se partageant pas de mémoire (*système distribué*)
- Dans ce cas, *envoyer* et *recevoir* ne sont pas seulement des recopies de mémoire, mais représentent des communications effectives à travers un *réseau*
- ⇒ Peut poser des problèmes délicats de gestion de la mémoire (voir exemple ci-après) à cause notamment des délais de communication variables

Problème de gestion mémoire : risque de *saturation* de l'ensemble de tampons du consommateur



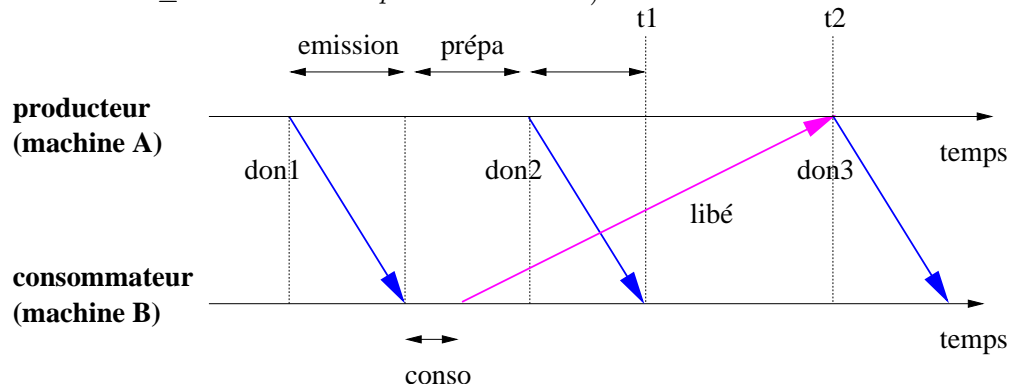
- Solution 1 : le consommateur envoie un message d'arrêt de production quand tous ses tampons sont pleins



- Non satisfaisant à cause du délai de transmission

Solution 2

- Le producteur compte le nombre de tampons vides $nbvideB$ du consommateur
- Transmission de messages du consommateur au producteur à chaque libération de tampon ($\implies nbvideB \leq \text{nombre de tampons vides sur } B$)



2.2 Variations autour des sémaphores

a. Sémaphores avec messages

Différences avec un sémaphore à compteur classique :

- Paramètre supplémentaire au V : *message*, qui sera transmis lors du réveil (ou non blocage) d'un processus
- Le P retourne en résultat le message transmis lors du V
- Mise en œuvre :
 - Ajout d'une file de messages dans le descripteur du sémaphore

```
typedef struct { (int e; file descripteur f; file message fm; } sémaphore;
message P (sémaphore s) {
    s.e = s.e-1;
    if (s.e < 0) {
        entrer(aktif, s.f);
        aktif = nil;
    }
    return sortir(s.fm);
}
```

```
void V (sémaphore s, message m) {
    descripteur_processus *pr;
    s.e = s.e+1;
    if (s.f non vide) {
        pr =sortir(s.f);
        message_reçu(pr) = m;
        entrer(pr, activables);
    }
    else entrer(s.fm,m);
}
```

Remarques 35. - Les deux files ne sont jamais pleines simultanément, par contre elle peuvent être toutes les deux vides :

- si $s.e > 0$, alors $s.m$ contient e messages et il n'y a pas de processus en attente
- si $e < 0$, alors $s.f$ contient $-e$ processus en attente, mais il n'y a pas de message

b. Sémaphores d'exclusion mutuelle (verrous, locks)

Différences avec les sémaphores à compteur :

- Pas de valeur initiale en paramètre de la création (implicitement, valeur de 1)
 - Pas de blocage dans le cas d'acquisition de verrou par le même processus que celui qui le possède déjà
 - Test du fait que l'acquisition et la libération du verrou sont réalisés par le même processus
- Implantation (C)

```
typedef struct {
    tid holder;
    file *F;
    int nesting;
} t_mutex;
P (t_mutex *m) {
    if (m->holder != 0 && m->holder != active) {
        active → état = bloqué;
        insérer(active,m->F);
    }
    m->holder = active;
    (m->nesting)++;
}
```

```
V (t_mutex *m) {
    if (active != m->holder) erreur;
    if (nesting == 0) erreur;
    (m->nesting)-;
    if (m->nesting == 0) {
        if (!empty(m->F)) {
            t=retirer(m->F);
            t → état = prêt;
        }
        else m->holder = 0;
    }
}
```

b. Mutexes de la bibliothèque pthread

- `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)` Initialize a mutex with specified attributes.
- `int pthread_mutex_lock(pthread_mutex_t *mutex)` Lock a mutex and block until it becomes available.
- `int pthread_mutex_trylock(pthread_mutex_t *mutex)` Try to lock a mutex, but don't block if the mutex is locked by another thread, including the current thread.
- `int pthread_mutex_unlock(pthread_mutex_t *mutex)` Unlock a mutex.

c. Verrous lecteur-rédacteur

- Schéma de synchronisation très courant lors de l'accès aux données (fichiers, bases de données)
- Distinction de deux types d'acquisition du verrou :
 - *Lecture seule* : plusieurs accès simultanés en lecture sont autorisés
 - *Écriture* : jamais deux écritures en même temps ou une lecture en même temps qu'une écriture

c. Verrous lecteur-rédacteur bibliothèque pthread (extrait)

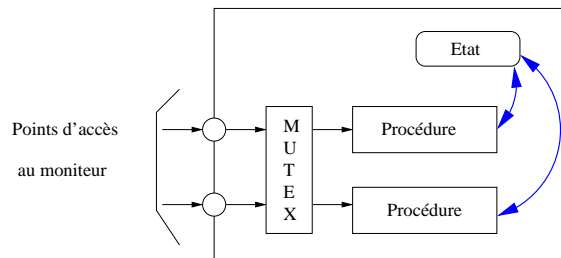
- `int pthread_rwlock_init(pthread_rwlock_t *lock, const pthread_rwlockattr_t *attr)` Initialize a read/write lock object.
- `int pthread_rwlock_rdlock(pthread_rwlock_t *lock)` Lock a read/write lock for reading, blocking until the lock can be acquired.
- `int pthread_rwlock_wrlock(pthread_rwlock_t *lock)` Lock a read/write lock for writing, blocking until the lock can be acquired.
- `int pthread_rwlock_unlock(pthread_rwlock_t *lock)` Unlock a read/write lock.

2.3 Moniteurs

- Notion de moniteur : Hoare 1974, Brinch Hansen 1975
- Outil de plus haut niveau que les sémaphores, destiné à faciliter l'écriture de programmes parallèles corrects
- Concept du *niveau langage*
- Peut être mis en œuvre en utilisant des mécanismes de synchronisation de plus bas niveau, tels que les sémaphores

Definition 36 (Moniteur). Un moniteur est constitué :

- D'un ensemble de *variables d'état*, inaccessibles directement aux utilisateurs du moniteur
- D'un ensemble de *procédures* manipulant ces variables, et accessibles aux utilisateurs du moniteur. Ces procédures s'exécutent en *exclusion mutuelle*



Definition 37 (Conditions). Une *condition* est une variable d'état d'un type particulier. Sur une condition c on peut réaliser les opérations suivantes (p désigne le processus qui s'exécute) :

- $c.attendre$: bloque le processus p , on dit que p est "en attente" de c ;
- $c.vide$: rend vrai si aucun processus n'est "en attente" de c ;
- $c.signaler$: réveille un des processus en attente de c , s'il en existe ($c.vide$ vrai), ne fait rien sinon

Moniteur : exemple

Producteur/consommateur utilisant un moniteur :

```
moniteur gestTampons ;

int nbplein = 0 ;
condition tplein, tvide ;
...déclaration de l'ensemble de tampons ...

// Fonctions internes non exportées
tampon *obtenirTvide (void) ;
    // pré-condition : il existe au moins un tampon vide
    // rend l'adresse d'un tampon vide, et le considère alors plein
tampon *obtenirTplein (void) ;
    // pré-condition : il existe au moins un tampon plein
    // rend l'adresse d'un tampon plein, et le considère alors vide

// procédures exportées du moniteur
void produire (info d) {
    tampon *ptProd ;
    if (nbplein == N) tvide.attendre() ;
    ptProd = obtenirTvide() ;
    *ptProd = d ;
    nbplein = nbplein+1 ;
    tplein.signaler()
};

// procédures exportées du moniteur
void consommer (var info d) {
    tampon *ptCons ;
    if (nbplein == 0) tplein.attendre() ;
    ptCons = obtenirTplein ;
    d = *ptCons ;
    nbplein = nbplein-1 ;
    tvide.signaler() ;
};

processus producteur {
    info d ;
    while (1) {
        d = calcul ;
        gestTamp.produire(d) ;
    }
};

processus consommateur {
    info d ;
    while (1) {
        gestTamp.consommer(d) ;
    }
};
```

```

    calcul(d);
  }
};

```

2.4 Transactions

- Type de problème résolu par les transactions : *partage cohérent d'informations*
- Origine de ce mécanisme : domaine des bases de données

Definition 38 (Transaction). Une *transaction* permet d'ordonner dans le temps les accès à des données, afin de respecter certaines propriétés, nommées *contraintes d'intégrité*.

a. Problème de cohérence des données partagées

- Manipulation d'une variable partagée par plusieurs processus concurrents \implies besoin de *synchronisation*
 - Exclusion mutuelle, ou
 - Lecteur/rédacteur (plus souple)
- Plusieurs variables manipulées \implies exclusion mutuelle (ou lecteur/rédacteur) sur *chaque* variable
 - Ce n'est *pas suffisant* si les données manipulées ne sont pas indépendantes (voir exemple ci-après)

Exemple 39. - Variables partagées manipulées grâce à 2 primitives *lire* et *écrire*

- On suppose qu'il existe une contrainte d'intégrité $C1$ entre les variables partagées A et B : $A = B$

| | |
|--|--|
| Processus P1 { int x,y; x = lire(A); x = x+1; > (P1.1) écrire(x,A); y = lire(B); y = y+1; > (P1.2) écrire(y,B); } | Processus P2 { int x,y; x = lire(A); x = 2*x; > (P2.1) écrire(x,A); y = lire(B); y = 2*y; > (P2.2) écrire(y,B); } |
|--|--|

Exemple 40. Remarques

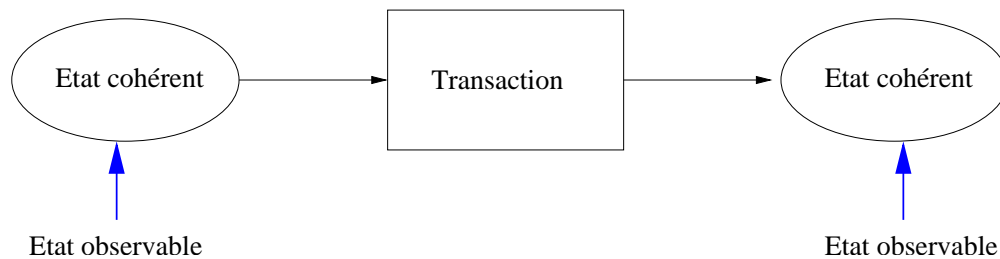
- On considère l'exécution de P1 seul, si $C1$ est vérifiée avant l'exécution, $C1$ est également vérifiée après. De même pour P2
- Toutes les exécutions séquentielles de P1 et P2 ne fournissent pas la même valeur, mais elles laissent $C1$ vérifiée
 - $\{A=B=i\}$ P1 $\{A=B=i+1\}$ P2 $\{A=B=2(i+1)\}$
 - $\{A=B=i\}$ P2 $\{A=B=2i\}$ P1 $\{A=B=2i+1\}$
- Si on exécute P1 et P2 en parallèle, en considérant que l'on entrelace les opérations dans l'ordre P1.1 ; P2.1 ; P2.2 ; P1.2 la contrainte d'intégrité $C1$ est violée (partant de $A = B = i$ on obtient $A = 2(i+1)$ et $B = 2i+1$)

Exemple 41. Solutions ?

- Manipuler A (resp. B) en *exclusion mutuelle* ne *résoud pas le problème*. Source du problème : A et B ne sont pas indépendantes, mais sont liées par la relation C1
 - Englober la *séquence entière* dans une section critique :
 - Résoud le problème, mais
 - Interdit tout parallélisme dans l'utilisation des données partagées (impensable dans une base de données)
- ⇒ Mécanisme de *transaction*, garantissant les contraintes d'intégrité en conservant la possibilité de parallélisme

b. Définition des transactions

Definition 42 (Transaction). Une transaction est une séquence d'opérations qui fait passer le système d'un état cohérent à un autre état cohérent.



- Durant l'exécution de la transaction, certains états intermédiaires *peuvent ne pas être cohérents*, mais ils ne sont *pas visibles* de l'extérieur de la transaction
- Seuls sont observables les états *avant et après* l'exécution complète de la transaction
- Deux transactions sont dites *concurrentes* s'il existe une période de temps pendant laquelle elles sont toutes les deux commencées et non terminées (le début de l'une se situe entre le début et la fin de l'autre)

Propriétés des transactions (ACID) :

- *Atomicité* : une transaction est soit réalisée entièrement, soit pas du tout (*tout ou rien*).
- *Cohérence* : une transaction fait passer le système d'un état cohérent à un autre (maintient des *contraintes d'intégrité*)
- *Isolation* : Les exécutions concurrentes d'une transaction sont isolées les unes des autres (pas d'interférence). En pratique on considère une autre propriété plus forte, la *serialisation* : le résultat de l'exécution concurrente de plusieurs transactions doit être identique à celui d'une exécution séquentielle
- *Durabilité* : une fois exécutée, les effets d'une transaction sont permanents et visibles à l'extérieur

c. Éléments de mise en œuvre

Primitives considérées d'un point de vue système :

- *débutTransaction* et *finTransaction* : définissent la portée d'une transaction
- *abandonTransaction* : arrête la transaction courante en ramenant le système dans l'état antérieur au début de la transactions abandonnée

Definition 43 (Transactions en conflit). Deux transactions sont dites *en conflit* si elles accèdent concurrentement à une même variable partagée A et si l'une des deux transactions la modifie

Contrôle de la concurrence :

- Objectif : permettre des exécutions concurrentes (accepter un certain entrelacement des opérations internes des transactions) en gardant la propriété de sérialisation
- Une solution (la plus simple) : verrouillage à *deux phases* (2PL)

Verrouillage à deux phases :

- Technique de contrôle a priori (pessimiste)
 - Verrous lecteur/rédacteur associé à chaque donnée
 - Verrouillage encadrant chaque opération sur une donnée (notion de transaction *bien formée*)
 - Verrouillage à *deux phases* : pas de verrouillage suivant un déverrouillage \implies deux phases
 - Une phase où le nombre de verrous posés ne fait qu'augmenter ;
 - Une phase où le nombre de verrous posés ne fait que diminuer.
 - Propriété : on peut démontrer qu'une transaction bien formée et à deux phases est sérialisable
- Le mécanisme de verrouillage introduit un risque d'*interblocage* (exemple I1 ; I2 : I3 ; I4)

| | |
|---------------------------|---------------------------|
| transaction T1 { | transaction T2 { |
| verrouiller_echr(A); (I1) | verrouiller_echr(B); (I2) |
| verrouiller_echr(B); (I3) | verrouiller_echr(A); (I4) |
| ... | ... |
| } | } |

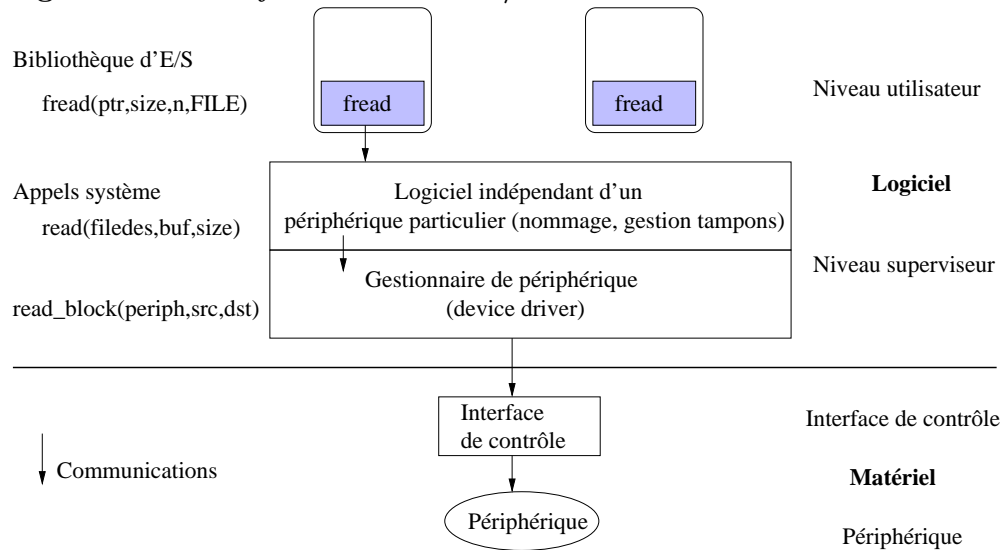
Solution possible ici :

- *Abandon* d'une des transactions en conflit, en "défaisant" évidemment toutes les opérations faites par la transaction abandonnée (propriété d'atomicité)
- Mise en œuvre possible de l'abandon : journal des opérations, avec le risque associé d'annulation en cascade

Cinquième partie

Gestion d'entrées/sorties physiques

Structure générale d'un système d'entrées/sorties



Éléments mis en œuvre

- *Périphérique* : dispositif mécanique, électromagnétique ou électronique assurant physiquement le transfert ou le stockage d'information (disque, clavier, imprimante, etc)
- *Interface de contrôle* : circuits assurant la liaison entre le processeur et le périphérique, et constituent le seul moyen d'accès au périphérique
- *Gestionnaire de périphérique* : gère les entrées/sorties physiques en utilisant l'interface de contrôle. Masquage à l'utilisateur du fonctionnement de l'interface de contrôle et du périphérique
- *Fonctions de bibliothèque* : encapsulent les appels système et mettent en œuvre des fonctions supplémentaires (ex : tampons d'entrée/sortie)

1 Le matériel

1.1 Périphérique (device)

Types de périphériques :

- Type *bloc* : stockage de l'information par blocs de taille fixe, chacun ayant sa propre adresse. Un bloc peut être lu/écrit indépendamment des autres.
Exemple : disques, disquettes, CD-rom, etc.
- Type *caractère* : accepte ou délivre un flot de caractères sans structure de bloc. Il n'est pas adressable et n'a pas d'opérations de positionnement.
Exemple : terminaux, imprimantes, interfaces réseau, souris.

Ordres des grandeurs des vitesses :

- Imprimante : vitesse de transfert de 400 car/s (un octet toutes les 2.5ms)
- Disquette : vitesse de transfert de 200 Ko/s (un octet toutes les 50 μ s)
- Disque : vitesse de transfert de 3Mo/s (un octet toutes les 0,3 μ s)

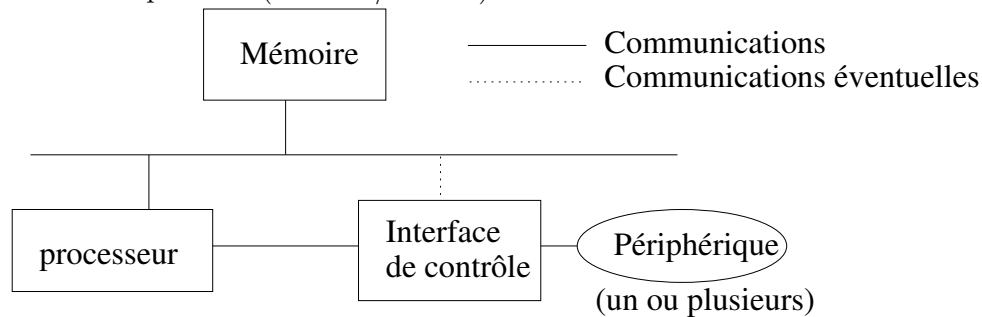
Remarques 44. – *Ordres de grandeurs donnés ici sont très approximatifs*

- *Ordres de grandeurs des vitesses conditionnent la façon dont le processeur gère le périphérique. Le processeur peut contrôler caractère par caractère les échanges avec l'imprimante, mais cela lui est impossible pour un disque.*

1.2 Interface de contrôle

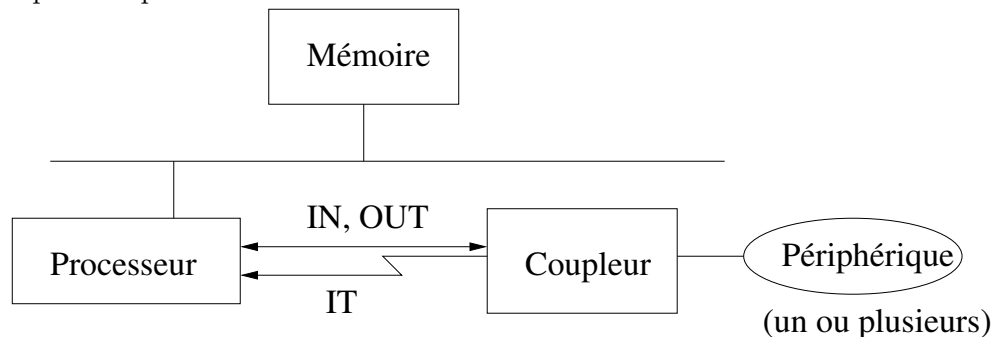
Interface de contrôle (device controller/adapter)

- Carte insérée dans l'ordinateur, servant à commander le périphérique
- Peut ou non selon les cas transférer directement l'information en mémoire
- Plusieurs degrés de complexité, correspondant à trois grandes classes d'interface de contrôle, de plus en plus sophistiquées
 - Coupleur
 - Coupleur + DMA (Direct Memory Access)
 - Processeur spécialisé (sur bus / réseau)



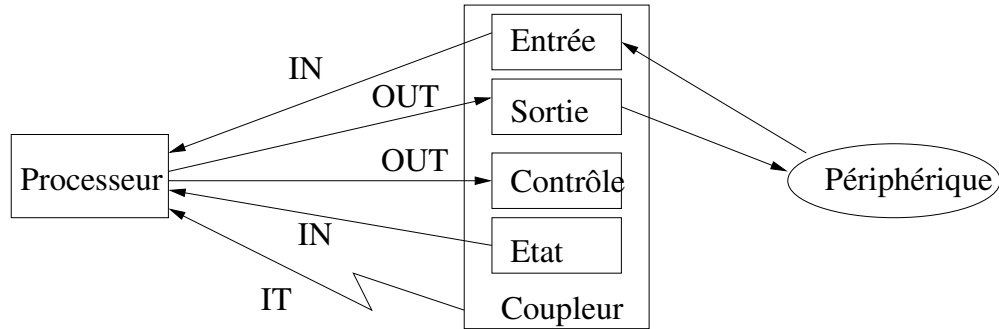
a. Coupleur

- Rôle : transfert d'information (un octet) entre les registres internes du coupleur et le périphérique
- Selon les architectures, registres du coupleur vus :
 - Comme des *adresses banalisées* (\implies manipulées par des instructions de transfert mémoire)
 - Memory Mapped I/O
 - Dans un *espace d'entrées/sorties* séparé (\implies manipulées par des instructions spéciales de type IN ou OUT)
- Le processeur assure *lui-même* les échanges avec la mémoire
- Exemple : coupleur série



Types de registres

- Registres de *données* : destinés à contenir les informations échangées avec le périphérique. Ils peuvent être lus (entrée) ou écrits (sortie)
- Registre de *contrôle* : sert à préciser au coupleur ce qu'il doit faire, et dans quelles conditions (vitesse, format des échanges,...). Ecrit par le processeur
- Registre d'*état* : décrit l'état courant du coupleur (libre, en cours de transfert, erreur détectée,...). Lu par le processeur.

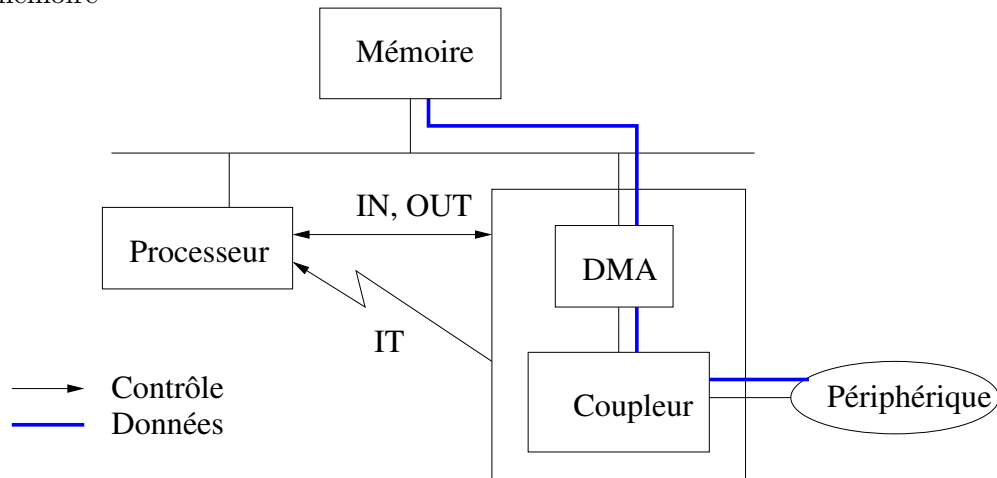


Synchronisation entre le processeur et le coupleur

- Rôle : adapter le comportement du processeur à l'état du coupleur (libre, en cours d'E/S)
- Types de mises en œuvre de la synchronisation
 - Par *attente active* : le processeur lit le registre d'état du coupleur, jusqu'à ce que celui ci soit dans l'état attendu
 - Par *interruption* : le coupleur signale au processeur son changement d'état par une IT

b. DMA (Direct Memory Access)

- Utilisé quand le processeur ne peut pas suivre le débit d'arrivée des caractères pour les stocker en mémoire
- DMA : circuit qui assure le transfert des informations de l'espace mémoire du coupleur vers la mémoire



- Fonctionnement
 - On indique au DMA une adresse mémoire et un compte d'octets à transférer
 - Synchronisation transparente entre le DMA et le périphérique
 - Schéma typique du code

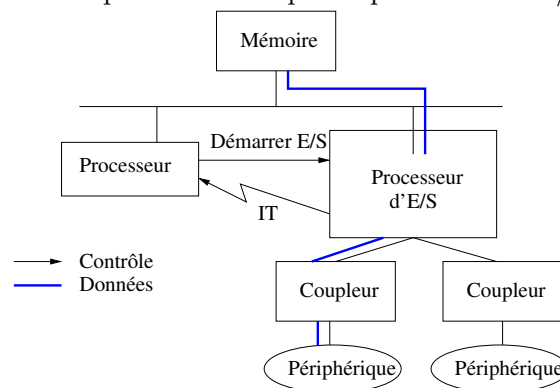

```
DMA.adresse = mémoire ;
DMA.compte = nb.octets ;
// Lancer l'E/S auprès du coupleur
// Fin transfert signalé dans DMA.etat et éventuellement IT
```

Remarques 45. - *Synchro DMA / coupleur transparent à la CPU*

- Le processeur peut continuer à effectuer des actions pendant l'E/S, mais ...
- Bus de la machine est partagé par le DMA et le processeur
- Partage du bus géré par le contrôleur de bus, vol de cycles, ralentissement de l'activité sur le processeur
- Cas du disque : transfert vers la mémoire se fera après avoir transféré les données du périphérique dans la mémoire du coupleur. En effet, à cause du partage du bus entre plusieurs éléments (périphériques/CPU), il serait possible que le DMA ne réussisse pas à suivre le débit d'arrivée des octets sur disque.

c. Processeur d'E/S

- Degré supérieur de déchargement du processeur des activités d'E/S
- Les processeurs peuvent ou non être spécialisés
- Rôle : gérer des E/S complexes de manière totalement autonome par rapport au processeur principal, ils peuvent contrôler plusieurs périphériques
- Où trouve t'on de tels processeurs ?
 - Dans les systèmes à forte demande en E/S (gros serveurs orientés gestion de données), machines réalisant des E/S de manière intensive (Playstation2), interfaces de contrôle élaborées, périphériques sur réseau
- Firmware : logiciel embarqué et exécuté par le processeur d'E/S



- Mécanisme d'exécution pour l'exécution d'un programme. Jeu d'instructions spécialisé ou non selon le processeur. Le programme est stocké en mémoire
- Accès direct à la mémoire (DMA), la mémoire est partagée avec le processeur
- Eventuellement, mémoire locale

Déroulement d'une E/S

1. L'OS, sur le processeur, prépare les paramètres de l'E/S (adresse disque, adresse mémoire, longueur, demandes d'IT...)
2. Le processeur signale au processeur d'E/S la présence d'une nouvelle requête
3. A partir de ce moment l'E/S est entièrement prise en charge par le processeur d'E/S, le processeur est libre
4. Quand le processeur d'E/S a terminé tout ou partie de l'exécution de la requête, il émet un signal d'IT vers le processeur

2 Le logiciel : problèmes de synchronisation

- Synchronisation entre le processeur et des dispositifs matériels *externes* (périphériques, contrôleurs de périphériques)
 - ⇒ On ne peut pas toujours ignorer la vitesse réelle du processeur par rapport au périphérique
- Deux cas sont distingués par la suite :
 - Le processeur a l'initiative du transfert
 - Le processeur n'a pas l'initiative du transfert
- On reste autant que possible indépendant du type d'interface de contrôle utilisée

2.1 Le processeur à l'initiative du transfert

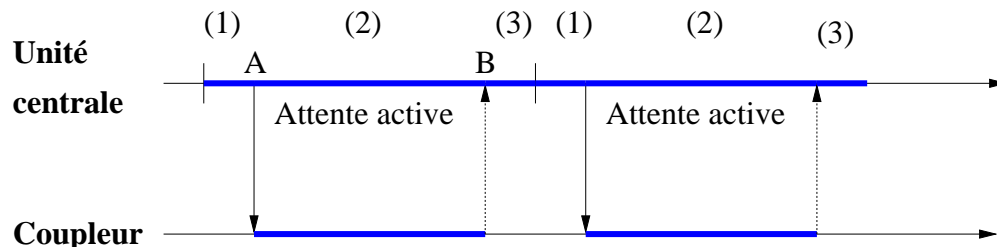
Synchronisation quand le processeur à l'initiative du transfert

- C'est le processeur qui décide quand l'E/S doit avoir lieu (ex : lectures/écritures disque, impression)
- Schéma typique
 1. Le processeur prépare les paramètres et lance l'E/S ;
 2. ... l'E/S se déroule ;
 3. le processeur effectue les traitements en fin d'E/S (résultats, erreurs...).

a. Synchronisation par attente active (test d'état)

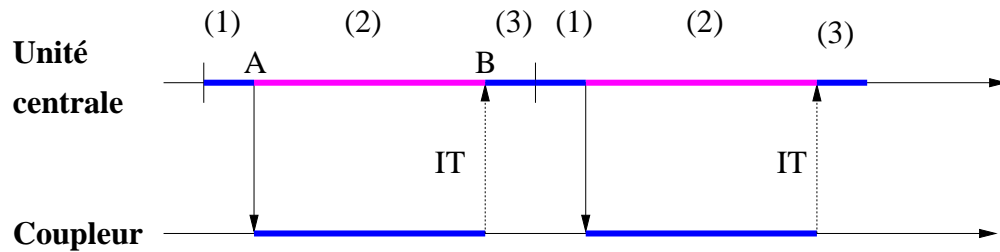
- Schéma typique du code (interface de contrôle=coupleur)

```
for (i = 0 ; i < N ; i++) {  
    coupleur.donnée = data[i] ;  
    while (coupleur.état == occupé) {;} // Attente active  
}
```



b. Synchronisation par interruptions

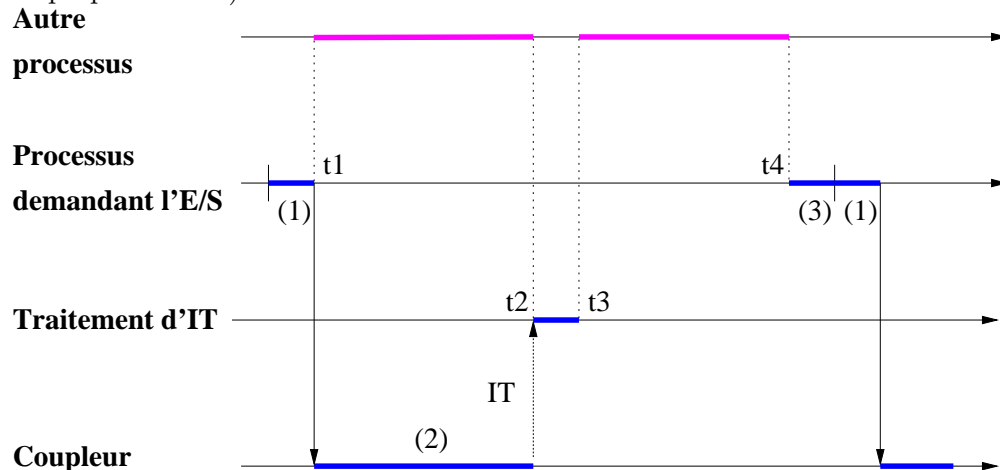
- Pendant la phase 2, le processeur est libre d'effectuer un autre traitement (entre A et B - zone rose)
- Autre dénomination : attente *passive*
- Principe des échanges sous interruptions (exemple précédent, diagramme temporel idéal) :



`sema atFin (0);`

| | |
|----------------------------|-----------------|
| processus demandeur | traitement d'IT |
| 1. lance l'E/S | V(atFin) |
| P(atFin) | Retour_IT |
| 3. traitement de fin d'E/S | |

- Mise en œuvre réelle d'un échange sous interruptions (retour systématique au processus interrompu par une IT)



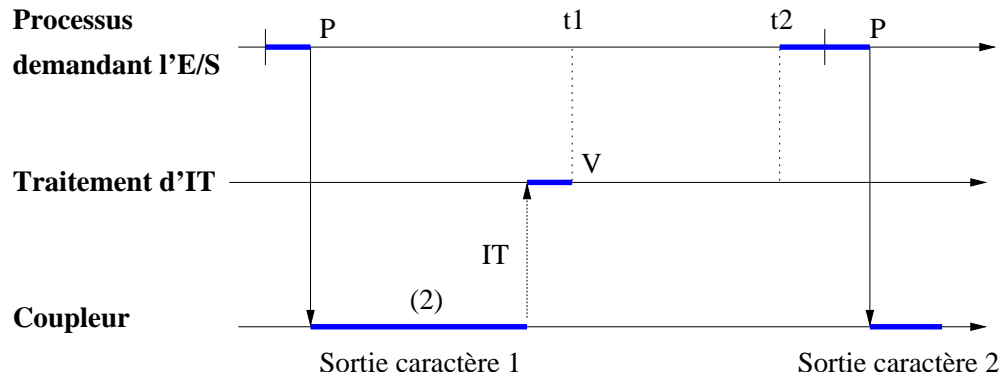
Entretien de l'E/S dans la routine d'IT

- Principe : relance de l'E/S suivante dans la routine d'IT
- Code correspondant à l'exemple de l'introduction (avant modification) :

`sema atFin (0);`

| | |
|---|-------------------|
| processus | traitement d'IT : |
| <code>char zone[100];</code> | V(atFin); |
| <code>for (i=0; i<100; i++) {</code> | Retour_IT |
| <code>ecrireDon(zone[i]);</code> | |
| <code>P(atFin);</code> | |
| <code>}</code> | |

Diagramme temporel (hors processus étrangers à l'E/S) sans entretien de l'E/S dans la routine d'IT



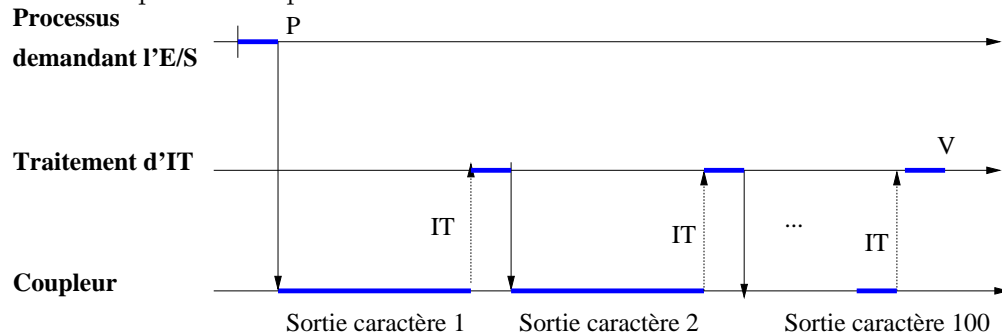
- 100 commutations de processus lors de l'impression de la zone
- Mauvaise utilisation de l'imprimante
- Code avec relance dans la routine d'IT :

```

sema atFin (0);
char limp[100];
int cpt;
processus :
cpt = 0;
ecrireDon(limp[0]);
P(atFin);
        traitement d'IT :
        if (cpt < 100) {
            cpt = cpt+1;
            ecrireDon(limp[cpt]);
        } else V(atFin);
        Retour_IT

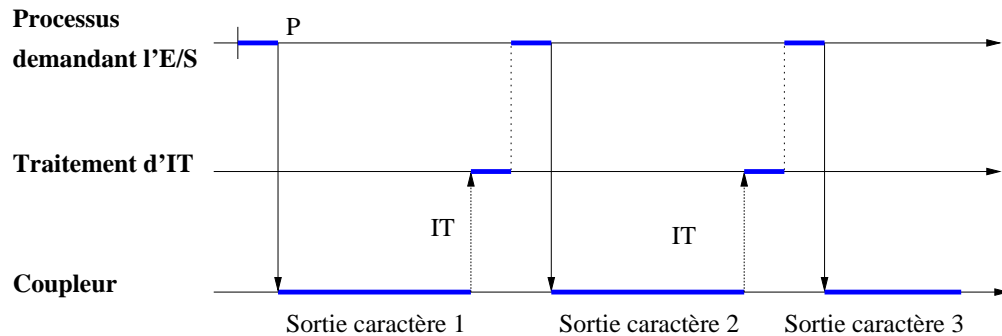
```

Diagramme temporel correspondant :



Appel à l'ordonnanceur en fin de routine d'IT

- Sauvegarde/restauration *complète* du contexte lors d'une routine d'IT
- Appel à l'ordonnanceur en fin de routine d'IT \implies on ne retourne pas nécessairement dans le processus interrompu
- Priorité la plus forte à tout processus bloqué dans le noyau par une entrée sortie
- Diagramme temporel



- Dans les solutions par IT, les traitements d'IT doivent être brefs. En effet, en général, tout ou partie des IT (les moins prioritaires) sont masquées. Avec un traitement d'IT trop long, on risque de ne pas être apte à traiter d'autres ITs en temps utile \implies schéma à privilégier : traitement d'IT :

V(...)
Retour_IT

- Si plusieurs processus utilisent le même périphérique/coupleur/DMA, il va bien entendu falloir assurer l'*exclusion mutuelle* sur ces éléments.

2.2 Le processeur n'a pas l'initiative du transfert

- Arrivée d'information vers le processeur, celui-ci ne maîtrisant pas les instants d'arrivée
- Cas considéré
 - Une donnée arrive toutes les T_a unités de temps ;
 - Il faut T_c unités de temps pour que le processeur traite cette donnée \implies Trois situations possibles selon les valeurs respectives de T_a et T_c

a. Cas $T_c > T_a$

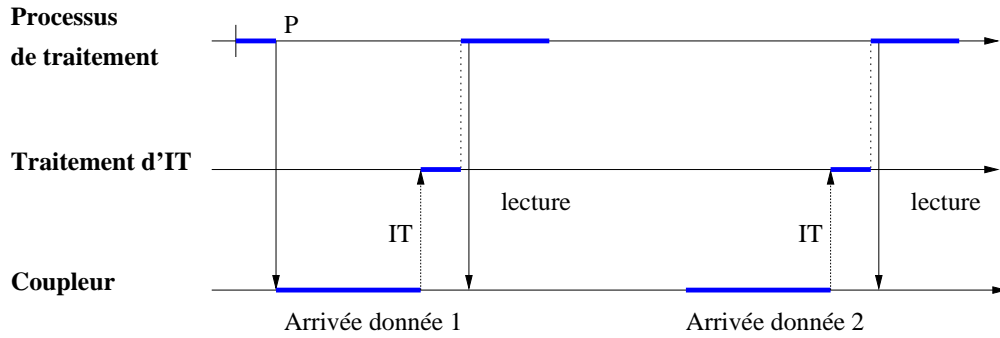
- Situation sans issue, car les données arrivent trop vite pour les traiter
- Contournements possibles du problème
 - Augmenter T_a (exemple : diminuer la fréquence d'échantillonnage)
 - Diminuer T_c en achetant un processeur plus rapide ou en utilisant un circuit spécialisé

b. Cas T_c très légèrement inférieur à T_a

- On peut traiter les données, et le temps restant après traitement est trop petit pour en faire quoi que ce soit d'utile
- Meilleure solution dans ce cas est l'*attente active* (simplicité, rapidité)

c. Cas T_c très inférieur à T_a

- Récupération d'une partie du temps $T_a - T_c$ pour exécuter autre chose \implies utilisation des IT
- Schéma temporel souhaité :



Code :

```
sema atDon init 0;
```

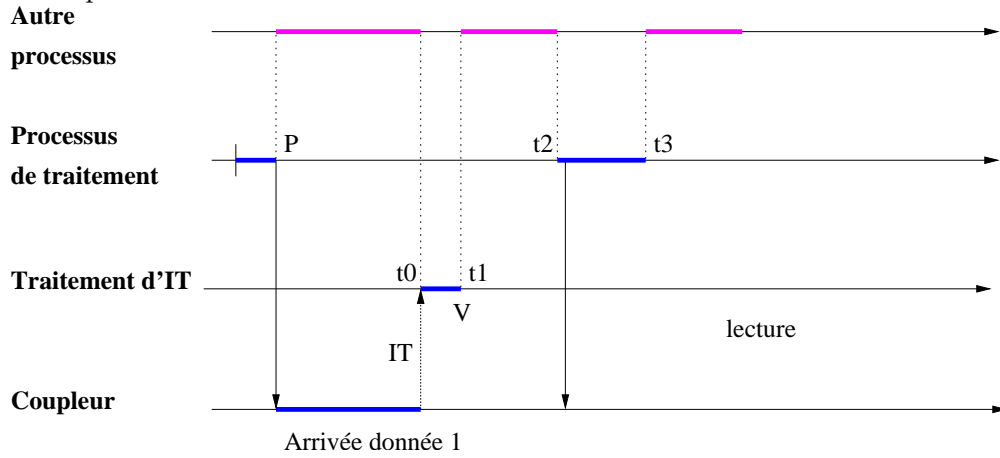
```
processus
```

```
...
while (1) {
  P(atDon)
  d = info arrivée
  traiter(d)
}
```

```
traitement d'IT :
```

```
...
V(atDon)
Retour_IT
```

Schéma temporel :



Tamponnage :

- Sauvegarde des données dans un tampon par la routine d'IT, consommation du tampon (producteur/consommateur un peu particulier, la production étant réalisée par la routine d'IT, ne pouvant pas se bloquer)

Traitement d'IT



Tampon

Processus de calcul

3 Le logiciel : problèmes de structuration

Structuration des systèmes d'exploitation

Examinés ici : intégration des entrées/sorties dans un système d'exploitation

- Interface utilisateur
- Structure interne et modes de communication à l'intérieur du logiciel de gestion des entrées/sorties

3.1 Interface de la machine virtuelle

Entrées/sorties synchrones et asynchrones

Deux manières pour un processus de percevoir le fonctionnement d'un périphérique :

- *E/S asynchrone (non bloquante)* : début et fin d'E/S sont vus comme deux événements distincts. Entre ces deux événements, il peut faire ce qu'il veut
 - ... {avant E/S}
 - lancer E/S
 - ... {pendant E/S}
 - attendre fin d'E/S
 - ... {après E/S}
- *E/S synchrone (bloquante)* : l'opération d'E/S forme un tout indivisible, le processus ne peut rien faire pendant l'E/S.
 - ... {avant E/S}
 - faire E/S
 - ... {après E/S}

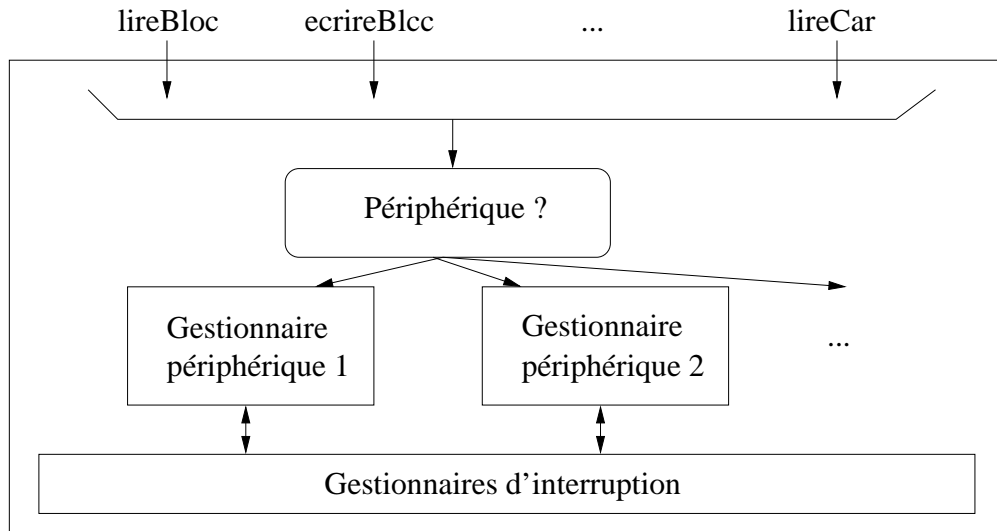
Fonctions d'E/S offertes à l'utilisateur

- Rôle du système de gestion des E/S physiques : fournir aux couches supérieures du système (SGF en particulier) des primitives qui cachent le plus possible les spécificités du matériel et ses contraintes techniques et temporelles.
- Exemple de fonctions d'E/S synchrones
 - périphériques de type bloc (disque par exemple) : lireBloc(ad.périph, ad.disque, ad.mem...), écrireBloc(ad. périph, ad. disque, ad.mem...);
 - périphériques de type caractère (clavier par exemple) : lireCar(ad. périph., ad. mem, nb. car...) et écrireCar(ad. périph, ad. mem, nb. car...)
- Au niveau du système de gestion des E/S physiques, l'exécution de toutes les primitives entraîne effectivement une E/S physique

3.2 Architecture fonctionnelle

Architecture fonctionnelle d'un système d'E/S physiques

- Grands blocs d'un système d'E/S, indépendamment de leur mode d'interaction (vu par la suite)
- Au dessus, entrées sorties de taille variable



3.3 Organisation du contrôle de l'exécution

Deux grandes solutions opposées :

- Système *monolithique*, où tout se passe par appels de *procédures* (système UNIX par exemple)
- Système structuré en *processus* où l'E/S est réalisée par une coopération entre processus, comme par exemple l'envoi de *messages* (systèmes à base de micro-noyaux)

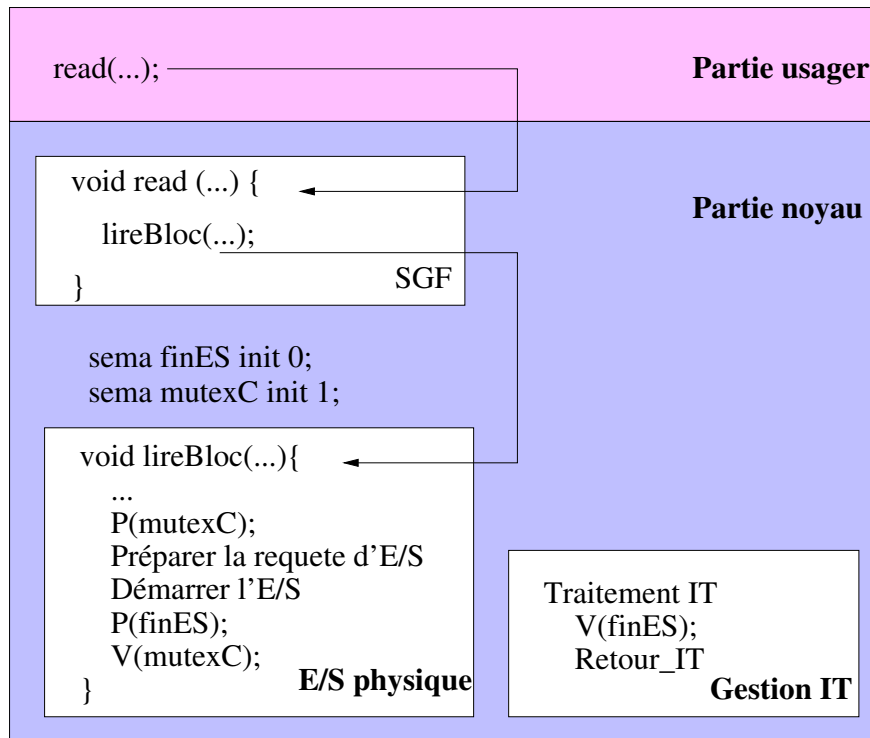
Illustration de ces deux modes de contrôle de l'exécution sur des E/S bloc

Contrôle par procédures

Deux parties dans un processus utilisateur :

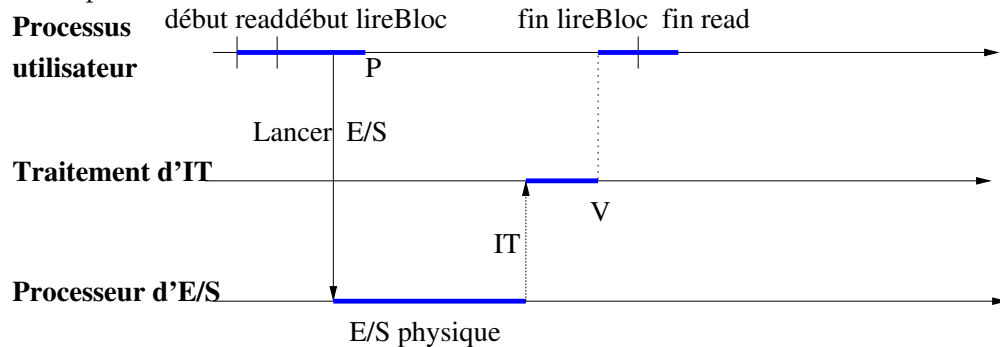
- Partie "usager" (code et données du programme utilisateur)
- Partie "noyau" (code et les données du système)

Un appel à une routine d'E/S conduit à traverser toutes les couches du système par appel de procédures, pour le compte du même processus



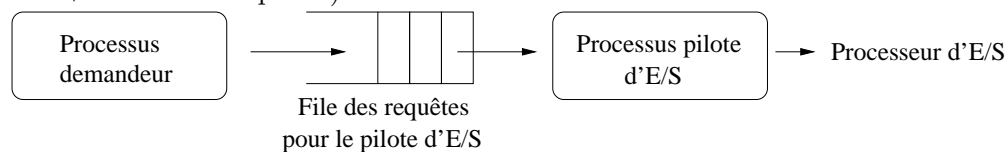
- Le sémaphore mutexC sert à assurer l'exclusivité d'accès au processeur d'E/S ou contrôleur
- Le sémaphore finES sert à synchroniser le processus faisant l'E/S et le disque
- Le pilote disque est représenté par une procédure (+ une routine d'IT)

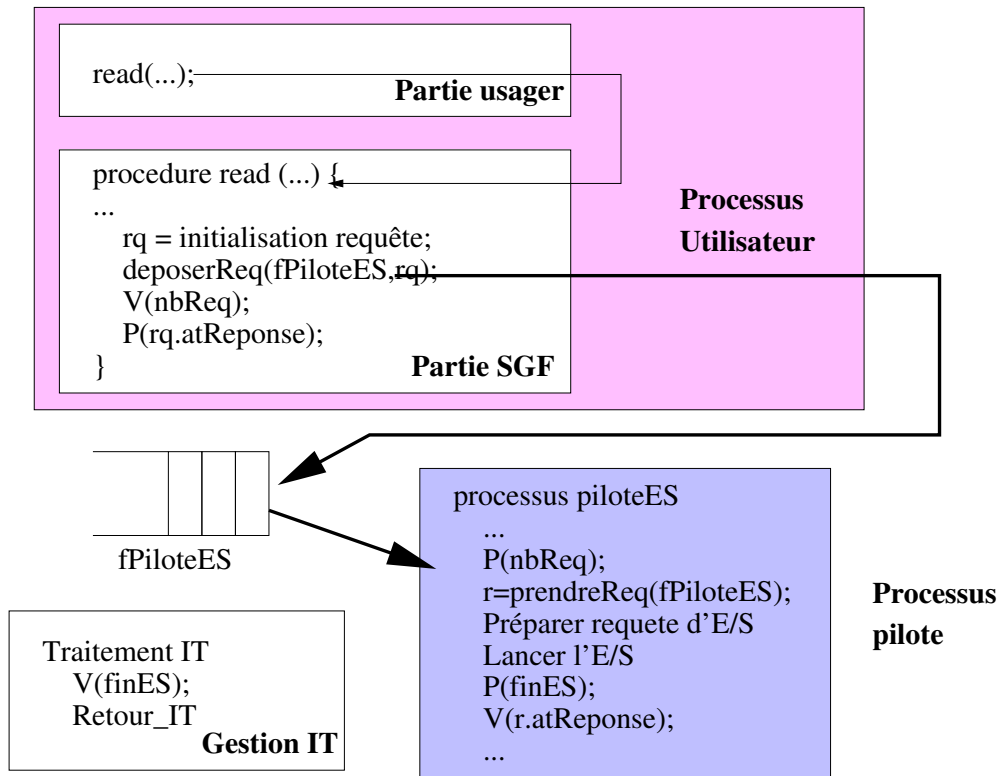
Schéma temporel :



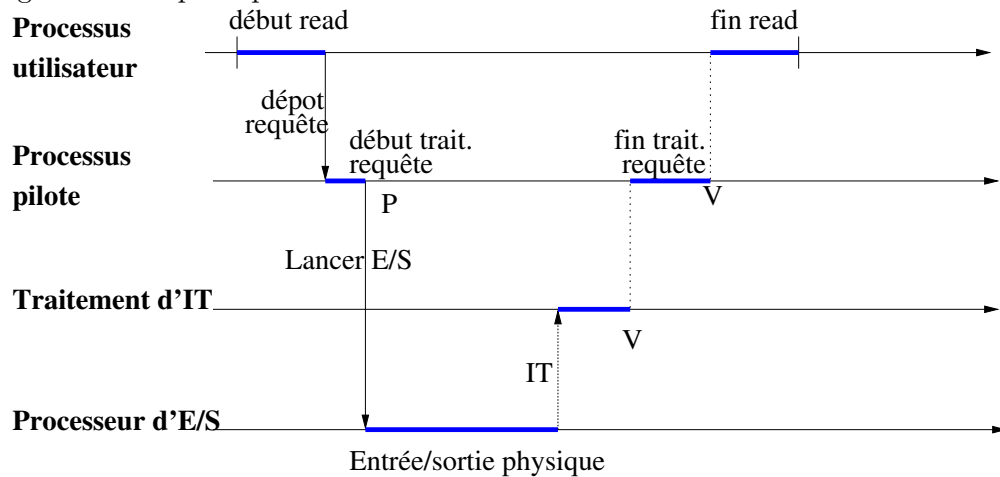
Contrôle par processus

- Le processus utilisateur n'exécute que le code propre à l'utilisateur
- Le système d'E/S est constitué de plusieurs processus de type "serveur" (le pilote d'E/S est un processus séparé)
- Communication entre utilisateur et pilote en utilisant un schéma client/serveur (dépot de requête + attente de réponse)





Un diagramme temporel possible :



Contrôle par processus vs procédure

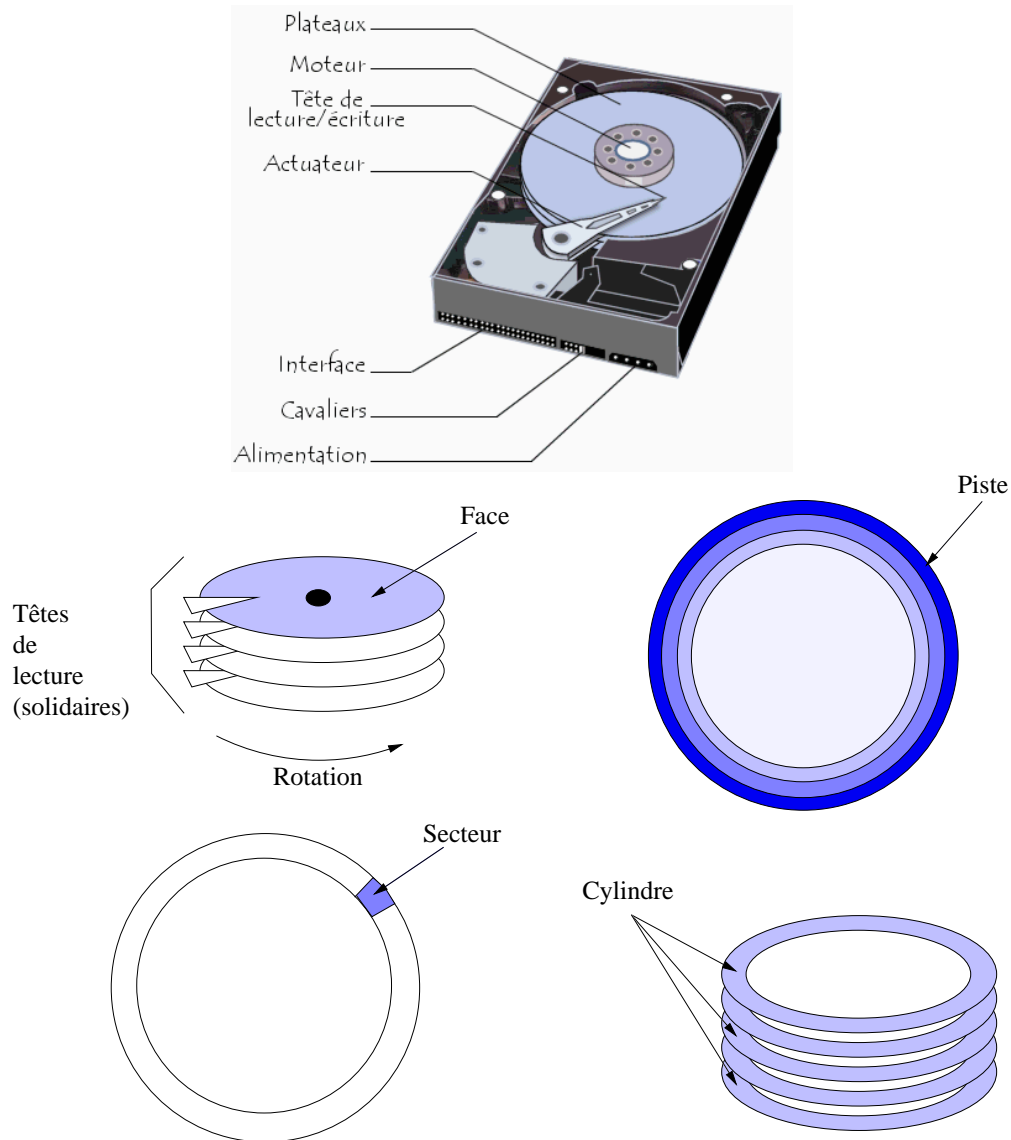
- L'architecture en processus plus claire (parties bien séparées, y compris pendant l'exécution)
- Version processus avec un unique serveur \implies pas de problème de partage de ressources \implies pas besoin d'exclusion mutuelle (si serveur mono-thread), pas de risque d'interblocage
- Commutation de contexte de plus en utilisant un processus serveur \implies besoin d'optimiser la commutation de contexte pour que les performances ne soient pas trop mauvaises
- Version procédurale, le passage en mode noyau est plus cher qu'un simple appel de procédure et se rapprochera de fait d'une commutation de contexte entre processus.

- Un bon fonctionnement du schéma avec processus suppose que l'ordonnanceur puisse tenir compte du caractère prioritaire des pilotes (ordonnancement avec avec préemption)
- Ordre de traitement des requêtes plus facile à contrôler avec un schéma à processus

4 Exemple : gestion des disques

4.1 Le matériel

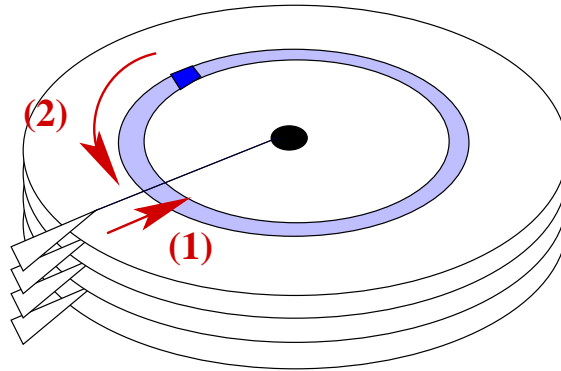
Disques



- Zone Bit Recording : plus de secteurs sur pistes externes (densité)
- Adresse disque = triplet (num. face, num. piste, num. secteur) (traduction numéro secteur → triplet faite par le contrôleur)
- Décomposition d'une E/S disque

1. *Positionnement piste*, on déplace le bras sur la piste indiquée (*seek time*);

2. *Latence*, on attend que le secteur désigné passe sous la tête (statistiquement, 1/2 tour)
3. *Transfert* proprement dit d'un ou 2 secteurs consécutifs



Remarques 46. – *Attente du secteur et le transfert lui-même indissociables*

- *Le contrôleur disque peut éventuellement enchaîner plusieurs commandes de positionnement/transfert.*
- *Le contrôleur disque peut disposer d'un tampon interne pour s'affranchir de certains problèmes temporels et accélérer le traitement des requêtes en évitant des accès disque*
- *Seek time variable selon la distance à parcourir*

4.2 Exemple de contrôle des E/S

- Processeur d'E/S capable d'enchaîner, dans un seul programme, les phases de positionnement et de transfert
- Format d'une commande (instruction)

| | | | | | |
|----|-----|-----|-----|----|-----|
| op | add | adm | nbs | IT | FIN |
|----|-----|-----|-----|----|-----|

 - op = opération à effectuer (pos, lect, ecr)
 - add = adresse disque à utiliser
 - adm = adresse mémoire à utiliser si besoin (pas pour pos)
 - nbs = nombre de secteurs consécutifs à transférer si besoin (pas pour pos)
 - IT = **1** pour demander une IT en fin de commande, **0** sinon
 - FIN = **0** pour enchaîner avec la commande suivante, **1** sinon (fin du programme)
- Lancement du processeur d'E/S
- Mise en œuvre par processus pilote
- Format d'une requête (porte implicitement sur bloc de 2 secteurs)
 - Sens du transfert (lecture ou écriture)
 - Adresse disque
 - Adresse mémoire
 - Sémaphore permettant de bloquer le demandeur (initialisé à 0)
 - Code d'erreur résultat (0 = pas d'erreur).
- Allocation dynamique des blocs de requête

```
typedef struct { int f, p, s; } addisque
typedef struct { sens s, addDisque add, adresse adm, sema atFin init 0, ent er } requete;
typedef struct { operation op, addDisque add, adresse adm, int nbs, int it, fin } commande;
```

```
sema nbReq (0);
```

```

file requete freqDisque;

void lireBloc (adDisque add, adresse adm, int *er) {
    requete *ptreq;
    ptreq = allouerBlocReq;
    ptreq → s = lecture;
    ptreq → add := add;
    ptreq → adm := adm;
    deposerReq(freqDisque, ptreq);
    V(nbreq);
    P(ptreq → atFin);
    *er = ptreq → er;
    libererBlocReq(ptreq)
}

sema finEsPhys (0);
int codeErPhys;
processus piloteDisque {
    commande com1,com2;
    requete *ptreq;
    while (1) {
        P(nbreq);
        ptreq = prendreReq(freqDisque);
        com1 = (pos, ptreq → add, - , - , 0, 0);
        com2 = (ptreq → s, ptreq → add, ptreq → adm, 2, 1, 1);
        Démarrer d'E/S de com1;
        P(finEsPhys);
        ptreq → er = codeErPhys;
        V(ptreq → atFin)
    }
}

```

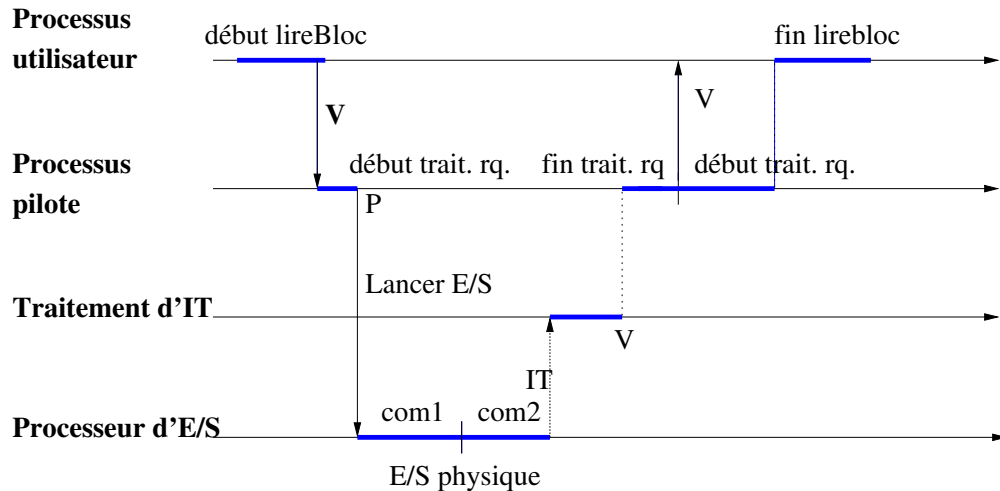
Traitement IT disque :

```

codeErPhys = code erreur fourni par le processeur d'E/S;
V(finEsPhys);
Retour_IT

```

Diagramme temporel :



IDE (Integrated Drive Electronics)

- Disque dur intégrant le contrôleur sur le disque
- ATA : interface d'un disque IDE

SCSI (Small COmputer System Interface)

- Carte d'interface hôte apte à gérer plusieurs périphériques, dont des disques
- Contrôleur associé à chaque disque
- Parallélisme entre contrôleurs (mieux adapté au multi-tâche)

4.3 Amélioration du temps de réponse

Décomposition du temps de réponse d'une requête d'E/S disque :

- Exécution du code du système d'exploitation
- Attente disponibilité du disque/contrôleur (partagés)
 - Système à procédure : blocage à l'entrée de la section critique
 - Système à processus : attente de la requête dans la file du pilote
- Positionnement sur la piste voulue
- Attente du passage du bon secteur sous la tête de lecture/écriture
- Temps de transfert proprement dit

a. Politiques de gestion du bras

- Déplacement du bras : opération longue par rapport au temps de transfert de l'information
- Pas de transfert d'information pendant le déplacement du bras \implies trop de temps passé en déplacement du bras réduit le débit effectif du disque par rapport à son débit maximal
- L'ordre de traitement des requêtes (ordre dans la file des requêtes) a un impact sur le temps total passé en positionnement du bras

Exemple 47. - Disque de 512 pistes par face (numérotées de 0 à 511 à partir de l'extérieur), 100 secteurs par piste

- Temps de passage d'une piste à l'autre de 0,5 ms
- Temps de rotation de 10 ms
- Durée (moyenne) d'une E/S d'un secteur (ms) = $T_{posit} + T_{transfert} = p \cdot 0.5 + 5.1$, avec p le nombre de pistes à traverser
- Requêtes de lecture de secteur pour les pistes 300, 6, 200 (dans l'ordre)

- Initialement, bras positionné sur la piste 0

Exemple 48. Version 1 : traitement des requêtes dans l'ordre d'arrivée

- $T_{ES1} (0 \rightarrow 300) = T_{posit} + T_{transfert} = 300*0.5 + 5.1 = 155.1$ ms
- $T_{ES2} (300 \rightarrow 6) = 294*0.5 + 5.1 = 152.1$ ms ;
- $T_{ES3} (6 \rightarrow 200) = 194*0.5 + 5.1 = 102.1$ ms ;
- Total = 409.3 ms

Version 2 : traitement des requêtes dans 6, 200, 300

- $T_{ES2} (0 \rightarrow 6) = T_{posit} + T_{transfert} = 6*0.5 + 5.1 = 8.1$ ms
- $T_{ES3} (6 \rightarrow 200) = 194*0.5 + 5.1 = 102.1$ ms ;
- $T_{ES1} (200 \rightarrow 300) = 100*0.5 + 5.1 = 55.1$ ms
- Total = 165.3 ms

Exemple 49. Temps de réponse pour les requêtes :

| Requête | Version 1 | Version 2 |
|---------|-----------|-----------|
| 300 | 155.1 | 165.3 |
| 6 | 307.2 | 8.1 |
| 200 | 409.3 | 110.2 |

- Version 2 : le système a mis globalement moins de temps à faire le travail, mais une requête a été servie un peu moins vite que dans la première version
- ⇒ Equilibre à trouver entre deux phénomènes
 - Amélioration du débit global
 - De pas trop défavoriser des requêtes particulières (et éviter la famine)
- Ces politiques n'ont d'intérêt que quand on charge le disque

a. Politiques de gestion du bras FCFS (First Come First Served)

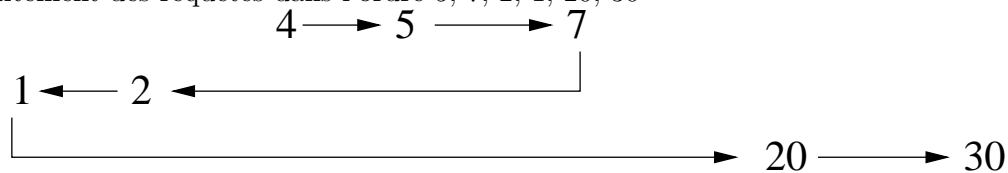
- *Premier arrivé - premier servi* (correspond à la version 1) : requêtes traitées dans l'ordre du dépôt
- Traitement *équitable* des requêtes (aucun risque de famine)
- Mais *beaucoup de déplacements du bras*

a. Politiques de gestion du bras SSTF (Shortest Seek Time First)

- *Plus petit déplacement d'abord* : privilégie la requête qui, à partir de la position courante, nécessite le moins de déplacement du bras

Exemple 50. - Requêtes concernant les pistes 1, 2, 5, 7, 20, 30, départ de la piste 4, pas d'autres requêtes arrivant par la suite

- Traitement des requêtes dans l'ordre 5, 7, 2, 1, 20, 30



a. Politiques de gestion du bras SSTF (Shortest Seek Time First)

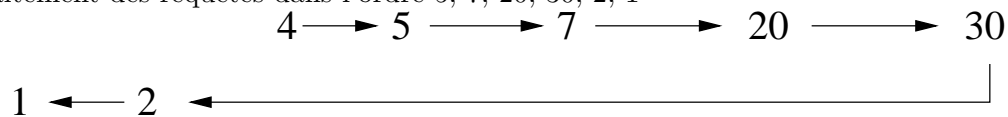
- Cette stratégie *diminue le nombre de déplacements du bras* (tout en n'étant pas optimale)
- Mais stratégie *non équitable*, le temps d'attente d'une requête est non borné (risque de *famine*)
- Exemple : nombreuses requêtes, concernant des pistes toujours proches de la position courante de la tête
- ⇒ le traitement d'une requête concernant une piste éloignée peut être continuellement reporté

a. Politiques de gestion du bras Politiques "de l'ascenseur"

- *Ensemble* de politiques
- Principe commun
- Déplace systématiquement du bras d'un côté à l'autre des faces
- On sert les requêtes concernant une piste quand le bras passe au dessus de cette piste
- Version la plus simple des politiques "de l'ascenseur" (SCAN)
- Déplacement du bras d'un bord à l'autre des faces (jusqu'au bout)
- Transferts dans les *deux sens* de déplacement du bras

Exemple 51 (Politique SCAN). - Requêtes concernant les pistes 1, 2, 5, 7, 20, 30, départ de la piste 4 en direction de la 511, pas d'autres requêtes arrivant par la suite

- Traitement des requêtes dans l'ordre 5, 7, 20, 30, 2, 1



Evaluation de la politique SCAN

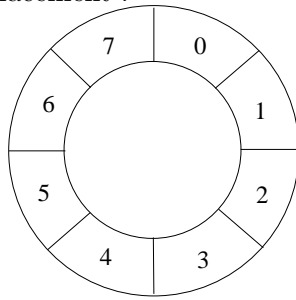
- *Optimise* les déplacements du bras
- Fournit un temps de service *borné* pour une requête
- Pas totalement équitable : les requêtes concernant les pistes du milieu sont avantagées
- ⇒ Variantes :
- C-SCAN : on ne fait des transferts que *dans un sens* de déplacement, retour en un seul coup sans transfert
- LOOK : on *arrête de se déplacer* dans un sens quand il n'y a plus de requêtes concernant les pistes entre la position courante et le bord, dans le sens du déplacement, transferts dans les *deux sens*
- C-LOOK : même chose que LOOK, mais on ne fait les transferts que *dans un sens*

b. Placement des secteurs sur une piste

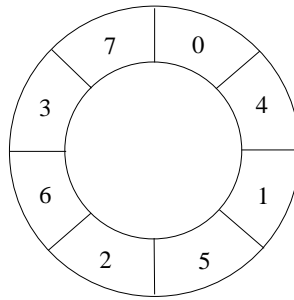
- Une E/S disque nécessite
- Transfert bloc disque → mémoire du contrôleur
- Transfert mémoire du contrôleur → mémoire centrale (via un DMA)
- ⇒ Quand on vient juste de lire un bloc, les contrôleurs simples ne peuvent pas gérer en parallèle
- transfert (mémoire contrôleur → mémoire centrale) et
- transfert (disque → mémoire contrôleur) pour le bloc suivant
- ⇒ Une fois le transfert (mémoire contrôleur → mémoire centrale) terminé, le bloc suivant est tout juste passé sous la tête de lecture

⇒ Espacement des blocs sur disque (*entrelacement/interleaving*) (géré de manière transparente par le contrôleur)

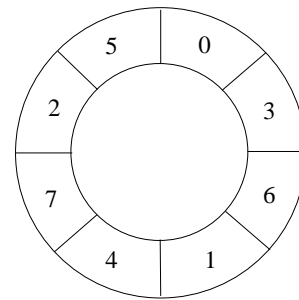
Entrelacement :



(a) Sans entrelacement



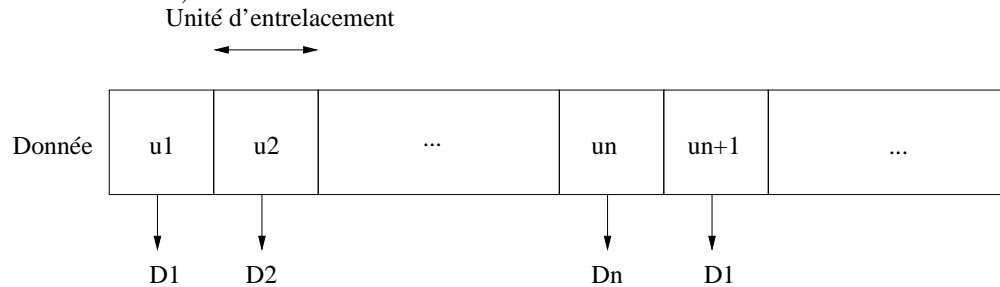
(b) Entrelacement de 1



(c) Entrelacement de 2

c. Tableau de disques

- Tableau de disque ("disk array") : collection de disques fonctionnant en parallèle et vus comme un disque unique
- Idée de base : répartir la donnée à échanger sur les différents disques, on parle d'entrelacement
- Exemple : u_1, u_2 etc. représentent des unités d'entrelacement que l'on répartit sur n disques (sans redondance)



Taille de l'unité d'entrelacement

- *Octet*
 - Augmentation du débit du transfert pour les *grosses requêtes* (traitement en parallèle des transferts, masquage du délai de positionnement si les disques sont synchronisés)
 - Méthode moins intéressante pour les *petites requêtes* qui ne permettent pas de remplir un bloc par disque
- *Bloc*
 - Pas de parallélisme dans le cas de petites requêtes (un seul disque est concerné)
 - Répartition automatique de la charge sur les différents disques

d. Caches disque - préchargement

Observations sur les accès disque :

- Accès fréquent de certains secteurs
- Dans certains cas, possibilité de prédire les accès futurs (accès séquentiels)

⇒ Deux types d'optimisation

- *Caches disque* : on conserve en mémoire une copie de l'ensemble des blocs les plus à même d'être réutilisés dans le futur

- *Préchargement* : on pré-charge en mémoire les blocs qui ont des chances d'être lus dans le futur
- Mémoire utilisée comme cache :
 - *Mémoire centrale* (élimine les temps de positionnement et de transfert) et la synchronisation avec le contrôleur
 - *Mémoire du contrôleur* (n'élimine pas le transfert vers la mémoire centrale)
- Utilisable pour d'autres périphériques que les disques

4.4 Fiabilité

- Avec les tableaux de disque, augmentation du nombre de disques \implies augmentation de la probabilité d'une *défaillance* d'un disque
- Utilisation de *redondance* (duplication de l'information) pour tolérer les pannes de disque (disques RAID - "Redundant Arrays of Independant Disk") à l'origine "Redundant Arrays of Inexpensive Disk"
- Erreurs locales (secteurs défectueux) peuvent être détectés par des codes détecteurs et correcteurs d'erreur sur le secteur

\implies On s'intéresse ici à la défaillance d'un disque complet, qui interdit tout accès à ce disque
Méthodes pour tolérer la défaillance complète d'un disque :

- Méthode simple : deux copies de chaque disque (*disques miroir*)
- Méthode plus économe : un disque de contrôle C pour deux disques de données D1 et D2
 - $C[i] = D1[i] \text{ xor } D2[i]$
 - Défaillance totale du disque D2 \implies on peut reconstituer D2 grâce à D1 et C ($D1[i] \text{ xor } C[i]$)

| D1 | D2 | C=D1 xor D2 | D1 xor C |
|----|----|-------------|----------|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

- Permet de restituer, sans interruption de service, un disque défaillant sur un disque de sauvegarde

Classification des disques RAID

Classement en fonction du niveau de redondance, d'entrelacement et du type de code correcteur utilisé :

- RAID 0 : pas de redondance, entrelacement (avec une grosse unité d'entrelacement on vise à augmenter le débit en octet/s, avec une petite c'est le débit en requête/s que l'on veut augmenter)
- RAID 1 : redondance par disques en miroir
- RAID 2 : accès en parallèle, petite unité d'entrelacement, code de hamming
- RAID 3 : accès en parallèle, petite unité d'entrelacement, bit de parité ;
- RAID 4 : accès indépendants, grosse unité d'entrelacement, parité sur un disque
- RAID 5 : idem RAID 4, mais les bits de parité sont répartis sur tous les disques

Sixième partie

Allocation de ressources et interblocage

Definition 52 (Ressource). Une *ressource* est un objet (matériel, donnée...), géré par le système et dont le système peut concéder temporairement le droit d'utilisation à un, ou plusieurs processus.

Etapas de l'utilisation d'une ressource

- *Demande* de la ressource au système d'exploitation → plusieurs issues possibles
 - Acceptation et allocation immédiate
 - Blocage en attente de disponibilité de la ressource
 - Refus
- *Utilisation* de la ressource allouée
- *Restitution* de la ressource, qui peut prendre deux formes :
 - Libération (abandon volontaire)
 - Réquisition (autoritaire). La réquisition peut poser problème pour certaines ressources

Critères d'un bon allocateur de ressources

Dysfonctionnements à éviter pour un allocateur de ressources :

- *Privation* (ou *famine*) : un processus attend indéfiniment une ressource
- *Congestion* : des allocations mal contrôlées conduisent à une surcharge du système, et à une dégradation de ses performances
- *Interblocage* (*deadlock*) : des processus se trouvent bloqués mutuellement, et indéfiniment
- Ici, étude de l'interblocage

1 Interblocage

1.1 Définition et caractérisation

Interblocage

Definition 53 (Interblocage). On dit qu'un ensemble de processus subit un *interblocage* si chaque processus de l'ensemble est bloqué et attend un événement que seul un autre processus de l'ensemble peut provoquer.

(Ici, l'événement attendu est l'allocation de la ressource).

Exemple 54 (Interblocage : exemple 1). – Système doté d'un disque et d'une imprimante

- Deux processus désirent imprimer un fichier
 - On note $dem(i)$ la demande, éventuellement bloquante, d'allocation du périphérique i
- | | |
|-----------------------|-----------------------|
| processus P1 ; | processus P2 ; |
| dem(imp) ; P11 | dem(disque) ; P21 |
| ... | ... |
| dem(disque) ; P12 | dem(imp) ; P22 |
- Suite d'actions $P_{11}, P_{21}, P_{12}, P_{22}$ conduit à un interblocage
 - Suite d'actions $P_{11}, P_{12} \dots$ ne conduit pas à un interblocage \implies il ne s'agit pas d'une erreur de programmation

Exemple 55 (Interblocage : exemple 2). – Blocage provenant d'une erreur de programmation

| | |
|-----------------------|-----------------------|
| processus P1 ; | processus P2 ; |
| ... | ... |
| P(mutex) ; | P(mutex) ; |

| | |
|----------|----------|
| P(s); | V(s); |
| V(mutex) | V(mutex) |
| ... | ... |

– Ici, le blocage provient du fait que l'on fait un P (bloquant) dans une section critique

Exemple 56 (Interblocage : exemple 3). – Deux disques, un imprimante, trois processus P1, P2, P3

– P1 et P2 veulent imprimer un gros fichier, chacun stocké sur un disque différent, P3 veut copier un fichier d'un disque sur l'autre

| | | |
|-------------------------------|-------------------------------|-------------------------------|
| processus P1; | processus P2; | processus P3; |
| dem(disque1); P ₁₁ | dem(imp); P ₂₁ | dem(disque2); P ₃₁ |
| dem(imp) P ₁₂ ; | dem(disque2); P ₂₂ | dem(disque1) P ₃₂ |
| ... | ... | ... |

– La suite d'actions P₁₁, P₂₁, P₃₁, ... conduit à un interblocage

– Après les trois premières allocations, aucun processus n'est bloqué, mais l'exécution ultérieure conduit *nécessairement* à un interblocage

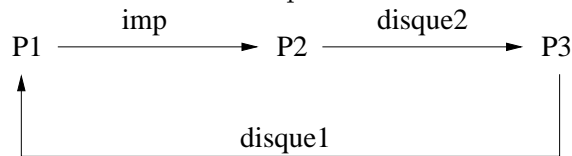
Conditions amenant à l'interblocage

Conditions devant être réalisées pour aboutir à un interblocage :

- *Exclusion mutuelle* : les ressources mises en jeu doivent être partagées et utilisées en exclusion mutuelle
- *Attente en détenant des ressources* : du aux demandes d'allocation en cours d'exécution et au fait qu'une demande d'allocation peut être bloquante
- *Non réquisition* : les processus font explicitement les libérations
- *Attente circulaire* : il existe une suite ordonnée de processus (P₁₁, P₁₂, ..., P_{1n}), telle que P_{ik} attend une ressource de P_{ik+1}, pour k ∈ [1,n-1] et P_{in} attend une ressource de P₁₁.

Graphe des attentes

- Graphe des attentes
 - Nœud du graphe : processus
 - Arc du graphe P_i → P_j si le processus P_i attend une ressource qui est détenue par P_j
- Identification d'une situation d'interblocage : le graphe des attentes possède un cycle
- Exemple : graphe des attentes dans l'exemple 3

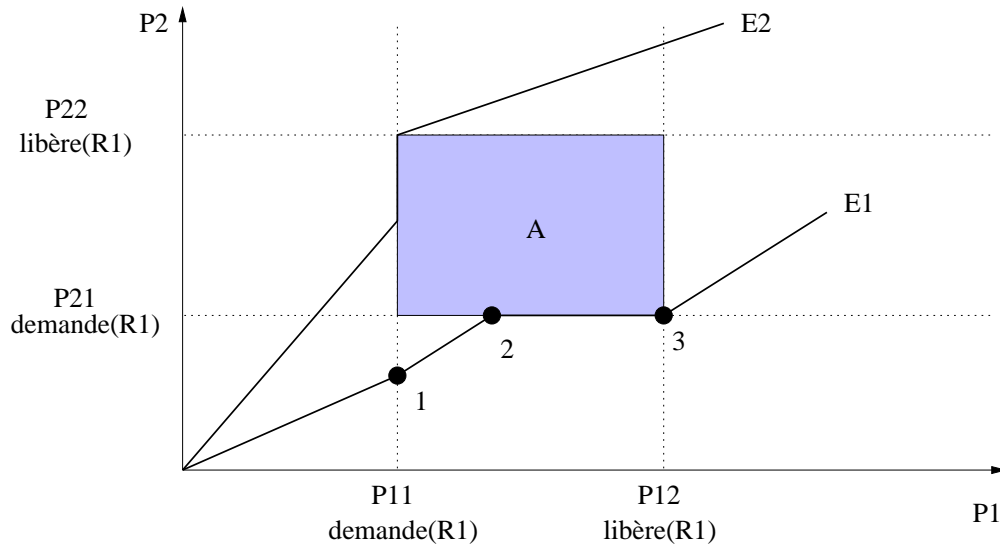


1.2 État sain

- Un interblocage peut devenir inévitable avant que l'on ne constate un blocage effectif de processus (exemple 3)
- ⇒ Modélisation de l'évolution des systèmes pour mettre en évidence ce point
- ⇒ Présentation sur un exemple simple (non sujet à l'interblocage) de deux processus P1 et P2 et une ressource partagée R1

processus P1 ;
 ...
 demande(R1) ; (P11)
 ...
 libère(R1) ; (P12)
 ...

processus P2 ;
 ...
 demande(R1) ; (P21)
 ...
 libère(R1) ; (P22)
 ...

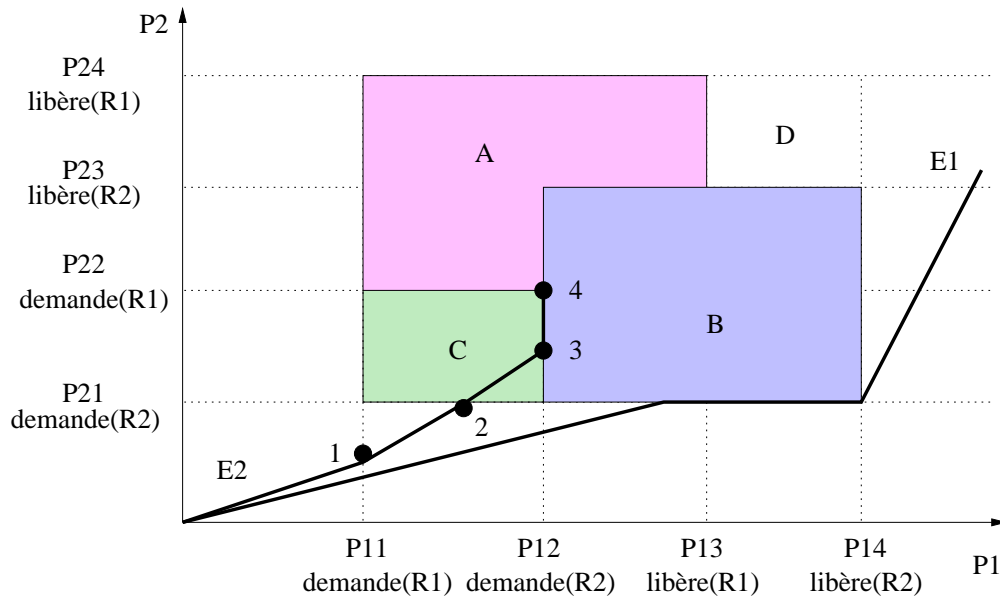


- Zone A interdite (R1 est allouée à P1 et P2)
- Courbe forcément non décroissante en X et Y

Soient deux processus P1 et P2, utilisant deux ressources exclusives R1 et R2

processus P1 ;
 ...
 demande(R1) ; (P11)
 demande(R2) ; (P12)
 ...
 libère(R1) ; (P13)
 libère(R2) ; (P14)
 ...

processus P2 ;
 ...
 demande(R2) ; (P21)
 demande(R1) ; (P22)
 ...
 libère(R2) ; (P23)
 libère(R1) ; (P24)
 ...



- Zones A et B correspondent à des zones d'exclusion mutuelle sur R1 et R2 \implies zones inaccessibles
- E1 est une évolution possible, qui ne conduit pas à l'interblocage
- E2 par contre conduit à un interblocage (à partir du point 2), avec la suite d'événements :
 1. allocation de R1 à P1
 2. allocation de R2 à P2
 3. demande de R2 par P1, qui se bloque
 4. demande de R1 par P2, qui se bloque, il y a interblocage
- En fait tous les points de la zone C conduisent *forcément* à l'interblocage (chacun des processus a obtenu une ressource et va finir par se bloquer en demandant l'autre)

Definition 57 (État sain). On dit qu'un état est *sain*, si à partir de cet état, il existe une stratégie d'allocation qui permet d'éviter l'interblocage

- Sur l'exemple précédent, les zones A et B correspondent à des états non réalisables
- La zone C correspond à des états non sains
- La zone D n'est pas accessible à cause de l'impossibilité pour un processus de revenir en arrière
- Les autres zones correspondent à des états sains

1.3 Types de solutions

- *Prévention* (méthodes pessimistes) : grâce à un contrôle *a priori* on s'assure qu'il n'y aura pas d'interblocage. Deux possibilités :
 - *Prévention statique* : on va rendre l'interblocage impossible. Il s'agit de faire en sorte qu'une des conditions d'interblocage ne soit jamais vérifiée. On limite la liberté des utilisateurs pour rendre l'interblocage impossible
 - *Prévention dynamique* (évitement) : bien que l'interblocage soit potentiellement possible dans le système, un contrôle effectué lors des allocations permet de faire en sorte qu'il ne se produise jamais

- *Détection et guérison* (méthode optimiste) : pas de contrôle *a priori*, mais périodiquement on regarde si le système est interbloqué, si c'est le cas on essaye de briser le blocage avec des méthodes plus ou moins brutales.

2 Prévention de l'interblocage

2.1 Prévention statique

Caractéristiques des méthodes de prévention statiques

- Choix dans la *structure du système* ou de son interface avec l'utilisateur, qui rendent l'interblocage impossible
- Peuvent être utilisées pour un sous-ensemble des ressources uniquement \implies possibilités d'interblocage résiduelles devant être gérées par un autre type de méthode

a. Suppression des ressources partagées

- Elimine la première condition d'interblocage : l'utilisation de ressources en exclusion mutuelle
- Moyen
 - Structuration du système pour qu'un seul processus utilise chaque ressource (structuration en *processus*, modèle client/serveur)
 - Manipulation de la ressource par un client \implies envoi d'une requête au serveur
- Méthode non généralisable à tout type de ressources (accès à des tables partagées) \implies à utiliser en complément d'autres méthodes pour limiter les risques d'interblocage

b. Réquisition

- Elimine la condition d'interblocage *attente en détenant des ressources*
- Fonctionnement : Blocage en détenant des ressources \implies *réquisition* de toutes les ressources détenues par le demandeur avant de le bloquer
- Frein : toutes les ressources ne peuvent être réquisitionnées sans dommage. En pratique, peuvent être réquisitionnées facilement les ressources dont l'état peut être sauvegardé

Remarque 58. *Stratégie utilisée dans les bases de données utilisant les transactions. Quand une ressource ne peut pas être allouée, abandon de la transaction en restituant son état initial*

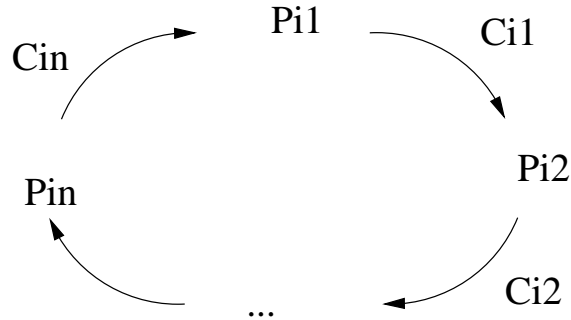
c. Allocation globale

- Elimine également la condition d'interblocage *attente en détenant des ressources*
- Fonctionnement : on impose à chaque processus de réserver globalement, en *une seule fois*, toutes ses ressources
- Impose de grosses contraintes à l'utilisateur qui doit demander le maximum de ressources quelles que soient les circonstances \implies mauvaise utilisation des ressources (allocation inutile ou trop précoce des ressources)

d. Classes ordonnées de ressources

- Elimine la condition d'*attente ciculaire*
- Fonctionnement
 - Les ressources peuvent être demandées en plusieurs fois

- On impose un *ordre* sur les demandes de ressources
- Ressources réparties en classes C_1, C_2, \dots, C_m
- Un processus ne peut demander une ressource de la classe C_i , que s'il a déjà obtenu toutes les ressources nécessaires des classes C_j , pour tout $j < i$
- Autre formulation : si un processus demande une ressource de la classe C_k , il lui est interdit de demander dans le futur une ressource de la classe C_j , avec $j < k$
- Absence d'interblocage (sinon il existerait un cycle dans le graphe d'attente, tel que les ressources étiquetant les arcs du cycle vérifient la propriété $C_{i1} < C_{i2} < \dots < C_{in} < C_{i1}$)



- Démonstration : soit P_i le processus possédant la ressource allouée de rang le plus élevé C_k .
- P_i est sûr de pouvoir terminer son exécution sans se bloquer (toutes les ressources qu'il peut demander sont libres)
- Il libérera alors toutes les ressources qu'il possède \implies répétition du raisonnement sur le processus possédant la ressource de rang l le plus élevé, avec $l < k$.

2.2 Eviter l'interblocage : l'algorithme du banquier

- Dû à du à Dijkstra (1965)
- Permet d'éviter l'interblocage (faire en sorte qu'il ne se produise pas dans un système où il est potentiellement possible)
- Pas de contrainte *a priori* sur l'utilisation des ressources, hormis celle de fournir au préalable une *borne supérieure* à ses demandes en ressources
- Principe du contrôle de l'allocation
 - Lors de chaque demande de ressource, on vérifie si l'état obtenu après allocation est sain (i.e. il sera possible d'éviter l'interblocage à partir de cet état). *Blocage* si on détecte que l'état ne sera pas sain.
 - Lors d'une libération de ressources : examen des processus bloqués. Réveil des processus bloqués si l'allocation de la ressource libérée laisse le système dans un état sain
- Ressources regroupées en classes (disques, imprimantes, etc)
- Nombre fixé de ressources de chaque classe
- Structures de données pour l'allocation

// n et m représentent respectivement le nombre de processus et de classes de ressources

```

int Exist[N] // Nombre de ressources existantes
int Dispon[N] // Nombre de ressources disponibles
int Alloc[Processus][Ress] // Ressources allouées
int Max[Processus][Ress] // Demande maximum
  
```

Definition 59 (Etat réalisable). Un état réalisable est tel que :

- $\forall p \in [0, n - 1]$ et $\forall r \in [0, m - 1]$, $O \leq Alloc[p, r] \leq Max[p, r] \leq Exist[r]$
- $\forall r \in [0, m - 1]$, $\sum_{k=0}^{n-1} Alloc[k, r] \leq Exist[r]$
- $\forall r \in [0, m - 1]$, $Dispo[r] = Exist[r] - \sum_{k=0}^{n-1} Alloc[k, r]$

Principe de la détection d'un état réalisable *sain* :

- Vérifier qu'il existe un ordre d'exécution ne conduisant pas à l'interblocage
- Basé sur les demandes *maximales* des processus
- Construction d'une suite ordonnée de processus $P_{i_1}, P_{i_2}, \dots, P_{i_n}$ telle que l'on puisse :
 - satisfaire les demandes maximales de P_{i_1} , avec les ressources disponibles et celles déjà allouées à P_{i_1}
 - satisfaire les demandes maximales de P_{i_2} , avec les ressources disponibles plus celle de P_{i_1} et celles déjà allouées à P_{i_2} , (revient à considérer que P_{i_1} s'est terminé et a libéré toutes ses ressources)
 - ...

Definition 60 (Suite saine). Une suite ordonnée de processus $P_{i_1}, P_{i_2}, \dots, P_{i_k}$ est une *suite saine* si et seulement si

$$\forall r \in [0, m - 1], \forall ik \in i_1, \dots, i_k, Max[ik, r] \leq Dispo[r] + \sum_{ij \leq ik} Alloc[ij, r]$$

Definition 61 (Etat sain). Un état est sain si et seulement si :

- Il est réalisable,
- Il existe une suite saine $P_{i_1}, P_{i_2}, \dots, P_{i_n}$ contenant tous les processus

Exemple 62. – Trois processus P0, P1, P2, trois classes de ressources C0, C1, C2

- Situation initiale

| | |
|-----------------|-----------------|
| Exist = (3,4,2) | Dispo = (1,2,1) |
| Max = 2 2 2 | Alloc = 1 1 0 |
| 1 2 2 | 0 1 1 |
| 2 3 2 | 1 0 0 |

- Besoins maximaux non satisfaits de chaque processus

$$\begin{aligned} P_0 &: (1, 1, 2) \\ P_1 &: (1, 1, 1) \\ P_2 &: (1, 3, 2) \end{aligned}$$

Exemple 63. – Les demandes restantes de P_0 et P_2 ne peuvent être satisfaites avec ce qui est disponible

- Par contre, on peut satisfaire les besoins maximaux de P_1 avec :
 - Ce que possède P_1 : (0, 1, 1)
 - Ce qui est disponible : (1, 2, 1)
 - Total 1, 3, 2 \geq (1, 2, 2)
- On peut ensuite satisfaire les besoins maximaux de P_0 avec :
 - Ce qui est disponible : (1, 2, 1)
 - Ce que possède P_1 : (0, 1, 1)
 - Ce que possède P_0 : (1, 1, 0)
 - Total : (2, 4, 2) \geq (2, 2, 2)

Exemple 64. – On peut ensuite satisfaire les besoins maximaux de P_2 avec :

- Ce qui est disponible : (1, 2, 1)

- Ce que possède P_2 : (1, 0, 0)
- Ce que possède P_1 : (0, 1, 1)
- Ce que possède P_0 : (1, 1, 0)
- Total : (3, 4, 2) \geq (2, 3, 2)
- La suite P_1, P_0, P_2 est saine. L'état est donc sain, le système n'est pas interbloqué

Demande de d_j ressources de la classe j par P_i :

```

if (Alloc[i,j]+dj > Max[i,j]) {
  erreur "demande trop forte"
else if (dj > Dispo[j]) {
  mettre Pi en attente // Pas assez de ressource
else
  Alloc[i,j] = Alloc[i,j]+dj;
  Dispo[j] = Dispo[j]-dj;
  if (nouvel état sain) effectuer l'allocation
  else {
    Alloc[i,j] = Alloc[i,j]-dj;
    Dispo[j] = Dispo[j]+dj;
    mettre Pi en attente // Allocation dangereuse
  }
}
}
}

```

Situations de blocage du processus demandeur :

- Pas assez de ressources disponibles pour satisfaire la demande
- Assez de ressources disponibles mais allocation dangereuse, car elle peut conduire à l'interblocage

Algorithme de test pour savoir si on est dans un état sain

- Recherche d'une permutation de processus parmi toutes les permutations possibles
- Complexité *a priori* : $O(n!)$ opérations. Peut être réduit à $O(n^2)$ en exploitant deux propriétés :
 - Si l'état d'allocation est sain, et si la suite partielle $S=P_{i_1}, \dots, P_{i_k}, k < n$ est saine, alors S est le début d'une suite saine contenant tous les processus
 - \implies Pas besoin de *retour arrière*
 - Si un état sain est transformé par une allocation à un processus P_k , et si on peut construire une suite partielle saine contenant P_k , alors le nouvel état est sain
 - \implies Pas nécessaire de construire une *suite complète* (on s'arrête dès que la suite contient le processus demandant la ressource)

3 Détection et guérison de l'interblocage

3.1 Détection de l'interblocage

- Etat de départ : certaines ressources sont déjà allouées, des processus sont en attente de ressources

- Algorithme utilisable
- Similaire à l'algorithme du banquier, en travaillant sur les *ressources demandées Req* à un instant donné au lieu des *ressources maximales Max*
- Absence d'interblocage quand on est dans un *état sain*, i.e. on est capable d'ordonner tous les processus dans une suite P_{i1}, P_{i2}, \dots de telle façon que :
 - on peut satisfaire entièrement les demandes de P_{i1} avec les ressources disponibles
 - on peut satisfaire les demandes de P_{i2} avec les ressources disponibles, plus celles de P_{i1}
 - ...

```
// n et m représentent respectivement le nombre de processus et de classes de ressources
// Variables décrivant l'état d'allocation
int Dispo[m];           // Nombre de ressources disponibles
int Alloc[n,m];       // Ressources allouées
int Req[n,m];         // Demandes en cours à l'instant considéré
// Variables pour l'algorithme de détection
int Util[m];          // Nombre de ressources utilisables
int Marque[n];       // Tableau de marquage des processus
int interblocage;

for (p = 0; p < n; p++) Marque[p] = 0;
Util = Dispo; interblocage = 0;
while (interblocage == 0 &&  $\exists p \in [0, n-1]$  such that (!Marque[p])) {
  if ( $\exists p$  such that (!Marque[p] &&  $\forall r$  Req[p,r] ≤ Util[r] ) {
    Marque[p] = 1;
    for (j=0; j < m; j++) Util[j] = Util[j]-Alloc[p,j];
  }
  else interblocage = 1;
}
```

⇒ Si on détecte un interblocage, l'ensemble des processus interbloqués est constitué des processus *non marqués* à la fin de l'algorithme

3.2 Détection de l'interblocage dans Linux et Windows

- Linux noyau 2.6.17 : Détection paramétrable
- Windows XP et suite : configurable, associé aux Windows Driver Development Tools

3.3 Guérison

- Méthodes de guérison forcément autoritaires, avec des effets plus ou moins néfastes pour les processus
- Méthodes de guérison :
 - *Destruction* de processus pour récupérer leurs ressources
 - ⇒ Travail fait par les processus détruits est évidemment à refaire
 - ⇒ Peut conduire à des comportements incorrects (mise à jour incomplète de données)
 - *Réquisition*. Beaucoup de ressources ne peuvent pas être réquisitionnées sans remettre en cause l'activité passée ou future du processus

- *Réquisition avec retour en arrière* : On peut périodiquement sauvegarder un état des processus, constituant ainsi un *point de reprise*, ré-installé en cas de réquisition de ressource (retour arrière)

Exemple 65. - Un processus p utilise une table partagée en exclusion mutuelle selon le schéma suivant :

```

... (1)
demande l'accès à la table ;
...
début de modification de la table ;
... (2)
fin de modification de la table ;
libère l'accès à la table

```

- Détection d'interblocage quand p est en (2) + réquisition de la table \implies table dans un état incohérent
- \implies Comportement potentiellement incorrect du processus, voire du système complet

Exemple 66. - Par contre, si on a fait un point de reprise en (1), on peut en (2) :

- Retirer l'accès à la table pour p ;
- Ramener p et la table dans l'état sauvegardé en (1) ;
- Poursuivre l'exécution d'un autre processus utilisant la table.
- Le processus p va être amené à refaire une partie du travail déjà effectué, mais le comportement ultérieur de p et du système sera correct.

Septième partie

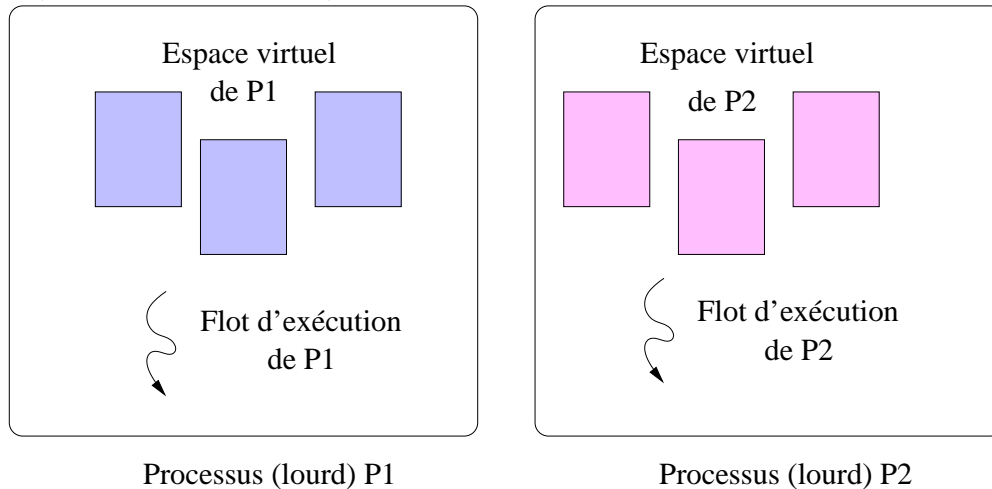
Problèmes d'architecture, exemples

1 Modèle d'exécution offert à l'utilisateur

- *Processus* : entité fondamentale du point de vue du contrôle de l'exécution (abstraction du processeur). Représente un flot d'exécution séquentiel
- *Espace virtuel* : entité fondamentale du point de vue de la mémoire :(abstraction de la mémoire). Ensemble des objets accessibles par le processus
- Une exécution utilise à la fois le processeur et la mémoire \implies il faut *composer* ces deux abstractions pour obtenir un modèle d'exécution complet :
 - Un espace virtuel par processus
 - Des espaces virtuels pouvant être partagés par plusieurs processus
 - Un espace virtuel unique pour tous les processus

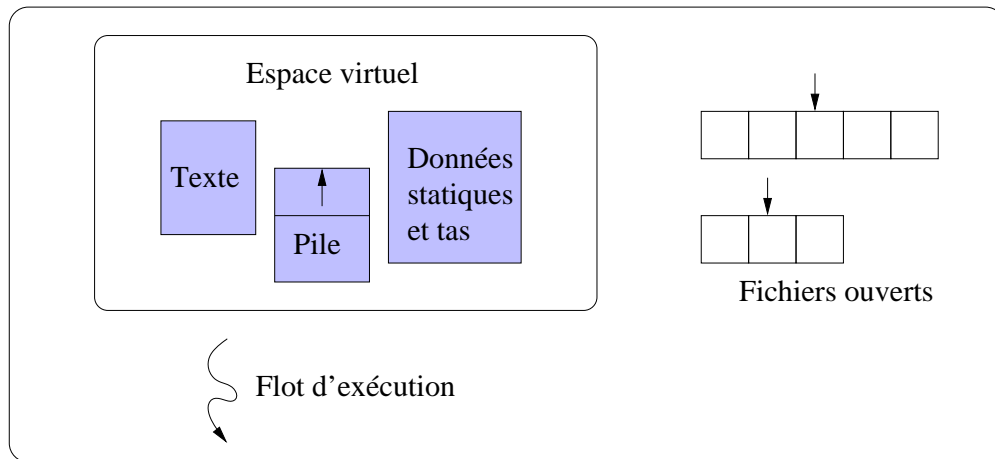
1.1 Un espace virtuel par processus

- Association processus et espace virtuel \implies *processus "lourds"* (modèle Unix)
- Abstraction offerte à l'utilisateur : *machine mono-processeur* (un processeur et une mémoire)
- Réalisation : fait correspondre plusieurs machines virtuelles de ce type sur une unique machine réelle (processeur + mémoire)



Processus Unix

- Correspond à l'exécution d'un programme
- Constitue l'unique entité active fournie par le système
- Constituants utilisateur d'un processus Unix
 - Objets accessibles
 - le texte du programme (code);
 - les données du programme (zone statique, tas, pile);
 - les fichiers en cours d'utilisation, ...
 - Flot de contrôle unique



Processus (lourd) vu par l'utilisateur

Appels système de manipulation des processus :

- *fork* : crée un nouveau processus (fils) identique à son créateur (père), au numéro près
- *exec* : fait exécuter à un processus un nouveau texte de programme (remplace le contenu de l'espace d'adressage)
- *exit* : termine le processus courant
- *wait* : permet au père d'attendre la fin d'un processus fils

Remarque 67. La création de processus par *fork* (duplication de processus) permet de régler le problème de partage des ressources entre processus (ressources partagées entre le père et le fils)

Code typique de création d'un nouveau processus

```

...
pid = fork();
{il y a maintenant deux processus presque identiques, mais
  pour le père pid=numéro du processus fils,
  pour le fils pid=0 }
si pid != 0
alors {code du père}
...
sinon {code du fils}
  exec(NomFichier,...)
fsi

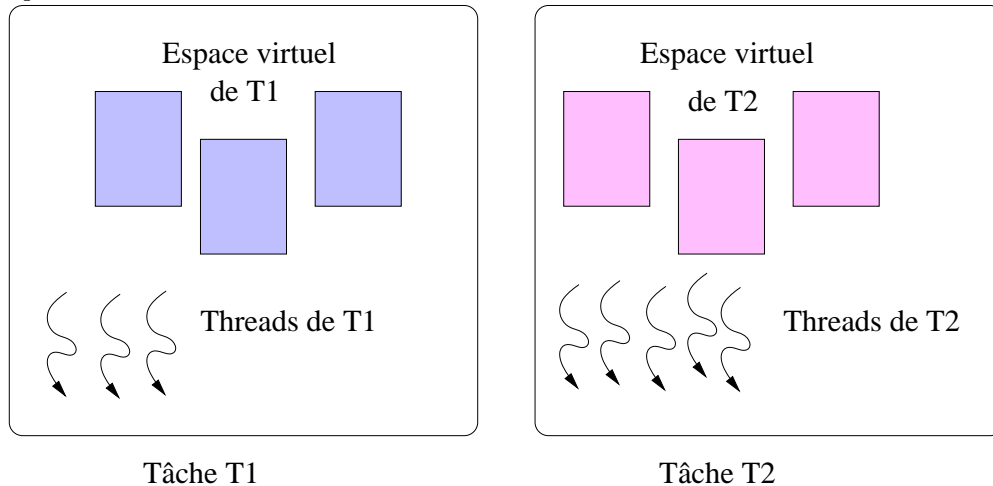
```

Moyens de communication entre processus :

- *Signaux* (événements asynchrones) : un processus peut envoyer un signal (kill), et spécifier une procédure à appeler quand il reçoit un signal (signal)
- *Tubes* (pipes) : représente un flot d'information allant d'un processus à l'autre
- *Fichiers*, qui peuvent être partagés entre un processus et ses fils
- *Zones de mémoire partagée, sémaphores, files de messages*

1.2 Plusieurs processus se partagent un espace virtuel

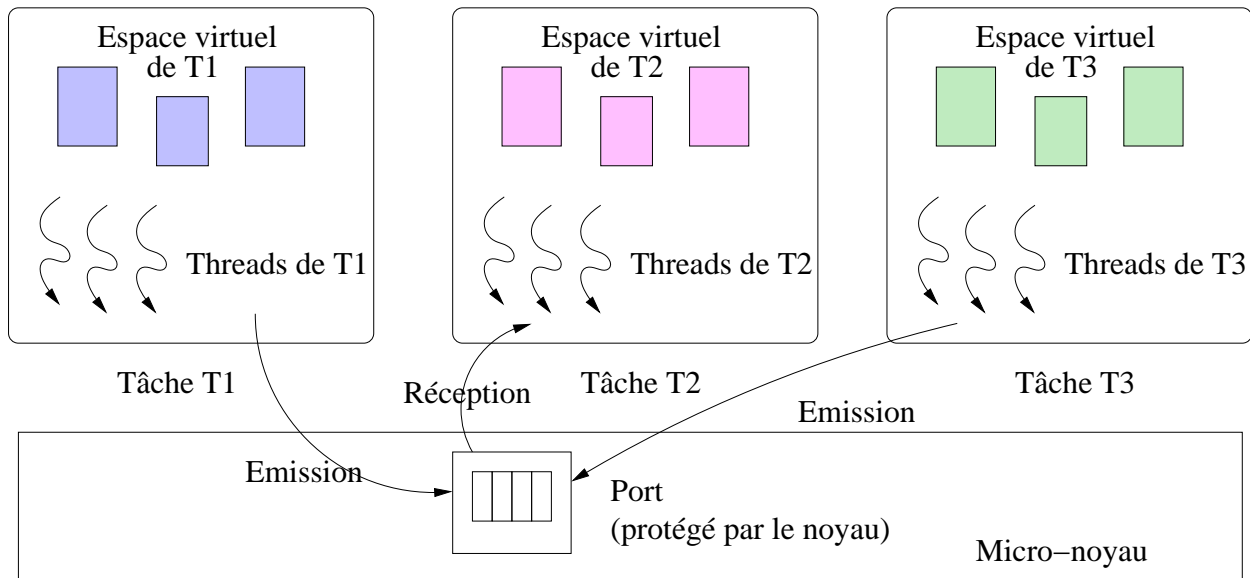
- Schéma précédent lourd si on veut faire coopérer finement plusieurs processus pour remplir une tâche donnée
 - ⇒ Noyaux de systèmes (micro-noyau Mach, Chorus...) introduisent deux notions différentes :
 - *Processus léger* (thread) : flot d'exécution séquentiel élémentaire
 - *Tâche* : ensemble constitué de plusieurs processus légers et d'un espace virtuel partagé entre eux
- Abstraction offerte à l'utilisateur plus complexe que précédemment : *multiprocesseur à mémoire partagée*



Activités et tâches de Mach

Abstractions offertes

- *Activité* (thread) : unité d'exécution séquentielle
Primitives pour créer, détruire, synchroniser les activités
- *Tâche* : unité de structuration, constituée d'un espace virtuel linéaire et d'activités s'exécutant dans cet espace virtuel
Primitives pour créer, terminer, suspendre, reprendre, contrôler l'exécution d'une tâche
- *Ports et messages* : Outils de communication entre tâches. Un port est associé à une tâche, il représente une file de messages, protégée par le noyau.
Primitives pour créer, détruire un port, gérer les capacités d'accès à un port, envoyer et recevoir des messages sur un port.



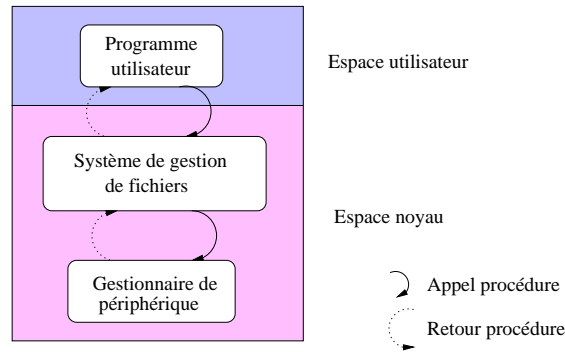
1.3 Un espace virtuel unique

- Cas limite de la solution précédente : un seul espace virtuel dans lequel s'exécutent *tous* les processus
- Problèmes
 - *Taille de l'espace virtuel* : doit pouvoir contenir toutes les informations manipulées par *tous* les processus existant à un instant donné \implies réaliste pour des systèmes spécialisés ou pour des systèmes à très grands espaces virtuels (64 bits)
 - *Protection* des informations sans sacrifier les performances
- Avantage : cadre très favorable pour le partage d'informations, pas besoin de mécanismes de liaison complexes

2 Contrôle de l'exécution

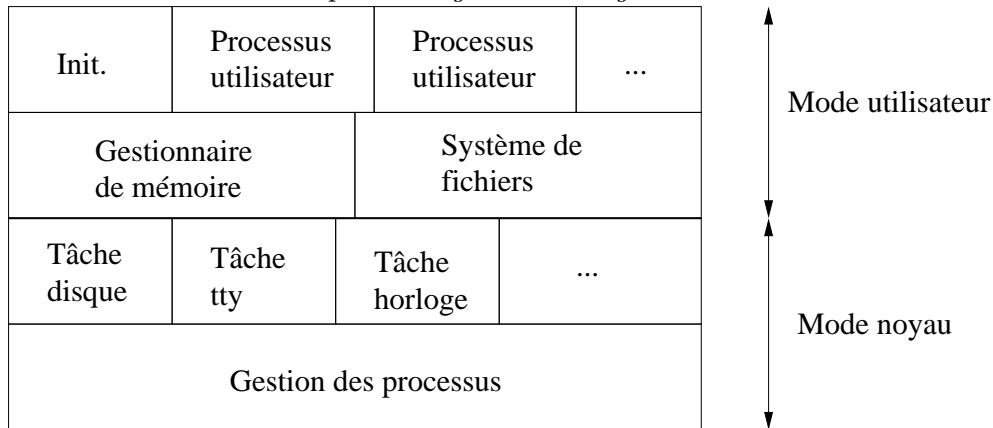
2.1 Contrôle par procédure : exemple d'UNIX

- Processus *utilisateur* composé de deux parties
 - Partie utilisateur : code, les données, pile
 - Partie noyau : code, données, pile noyau. Codes et données sont partagées par tous les processus, pile système privée au processus
- Autres processus existant en plus des processus utilisateur
 - Processus *démons* : s'exécutent en mode utilisateur. Réalisent des tâches d'administration, d'impression différée...
 - Processus *noyau* : s'exécutent entièrement en mode noyau. Très peu nombreux (swappeur, récupérateur de page...)
- Processus utilisateur demandant un service au système continue de s'exécuter en changeant de mode d'exécution (mode "noyau")
- Problème de ce type de structure : allocation des ressources (section critique, risques d'interblocages...)

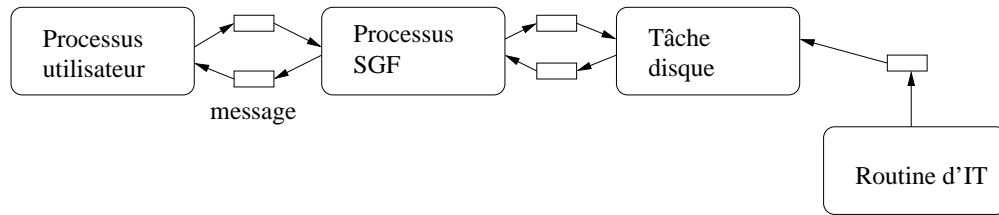


2.2 Contrôle par processus : exemple de MINIX

- Créé à des fins pédagogiques par A.S. Tanenbaum
- Interface compatible avec la version 7 d'UNIX, mais de structure très différente
- Trois types de processus :
 - *Utilisateur*
 - *Serveur* (gestionnaire de mémoire et système de fichier). S'exécutent en mode utilisateur. Traitent respectivement les appels comme fork, exec... pour le gestionnaire de mémoire et read, write... pour le système de fichier.
 - *Tâches d'E/S*. S'exécutant en mode noyau. Chargées de gérer les périphériques (tâche disque, tâche horloge...)
- Communication entre tâches par *échanges de messages* de taille fixe



- Réalisation d'une opération d'E/S (exemple, sur un fichier) : coopération entre le processus utilisateur, le processus système de fichier et la tâche de contrôle du disque, via des *échanges de messages*
- Avantage de cette structuration : moins de problème de partage de ressources (interblocage, section critique)
- Inconvénient : performances, dus aux échanges de messages



3 Architecture du système

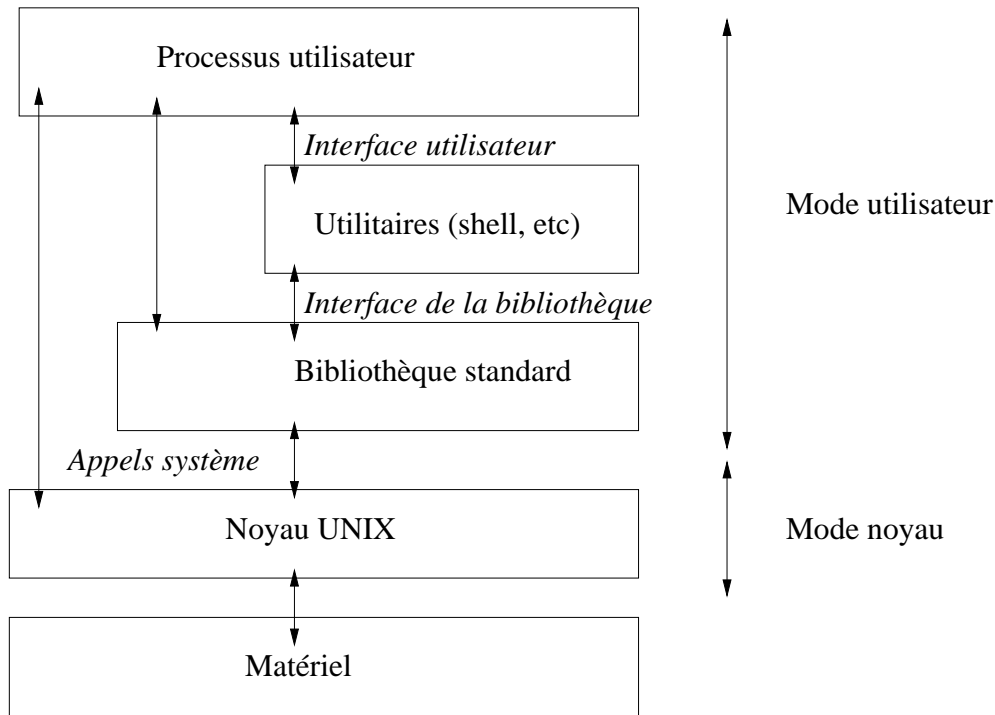
- Architectures *monolithiques* (Unix)
 - Système est vu (pour l'essentiel) comme une collection de procédures appelées par les processus utilisateurs
- Architectures *micro-noyau* (Mach)
 - Plus modulaire
 - Séparation des primitives de bases fournies par le micro-noyau et des services, utilisant le micro-noyau

3.1 Système monolithique : UNIX

- A l'origine, conçu pour être un *petit* système
 - Environnement de travail interactif
 - Petit nombre d'utilisateurs faisant principalement du développement de programmes
 - Gestion mémoire d'origine assez primitive
- Evolution du matériel et les besoins des utilisateurs \implies de plus en plus *gros et complexe*

Services système à plusieurs niveaux :

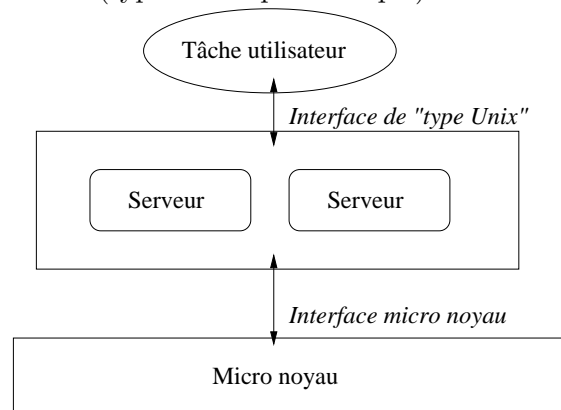
- *Noyau*
 - "Gros" noyau : supporte toutes les abstractions offertes à l'utilisateur
 - Interface du noyau constitue la frontière entre l'environnement système et l'environnement utilisateur (protection du système)
- La *bibliothèque standard* : permet d'encapsuler les appels système, sous forme de procédures conformes au langage de programmation utilisé
- L'interpréteur de commande (*shell*)
- *Utilitaires* de base (éditeur, compilateur,...)



3.2 Micro-noyau et serveurs : MACH

Principe : décomposition du système en deux niveaux

- *Micro-noyau* : offre des concepts de base, de plus bas niveau que ceux offerts à l'utilisateur
- Des *processus serveurs* : s'exécutent au dessus du micro-noyau, offrent à l'utilisateur un interface de plus haut niveau (type UNIX par exemple)



Intérêts

- *Portage facilité* : il suffit de porter le micro-noyau, les serveurs sont indépendants de la machine
- Système *modulaire* : il est possible de n'installer que certains serveurs (exemple : serveur de pagination)
- Stratégies mise en œuvre par les serveurs *personnalisables*

Fournis par le noyau Mach

- Notions de base : tâches, threads, ports, messages
- Gestion de la mémoire physique

Fournis en tant que services :

- Gestion de fichiers
- Serveurs de pagination (envoi de messages à chaque absence de page de la mémoire)

Références

- [1] S. Krakowiak. *Principes de systèmes d'exploitation des ordinateurs*. Dunod, 1987.
- [2] A.S. Tanenbaum. *Les systèmes d'exploitation, conception et mise en œuvre*. Interéditions, 1987.
- [3] A.S. Tanenbaum and A. Woodhull. *Operating systems : design and implementation*. Prentice Hall, 1997.
- [4] A. Silberschatz and P. Galvin. *Principes des systèmes d'exploitation*. Addison-Wesley, 1994.
- [5] A.S. Tanenbaum. *Systèmes d'exploitation : systèmes centralisés - systèmes distribués*. Interéditions, 1994.