Systèmes d'exploitation

Philippe Laroque

Licence d'informatique

septembre 2010

Plan I

1. Introduction
2. Gestion des processus
   - Introduction
   - Fonctions utiles sous UNIX
   - Ordonnancement
3. Communication entre processus
   - Communication par signaux
   - Communication par tubes
   - Synchronisation
   - Les sémaphores
   - Les sémaphores sous UNIX
   - Communication par mémoire partagée
   - Communication par files de messages
4. Gestion des entrées-sorties
   - Introduction

Plan II

- Méthodes d'accès
- Méthodes d'allocation
- Fonctions d'E/S sous UNIX

5. Gestion de la mémoire
   - Introduction - espaces d'adressage
   - Techniques d'allocation
   - Mémoire virtuelle

6. Interblocages

Situation de l'OS

Modèle en couches. De haut en bas:

1. Les programmes applicatifs
2. Les programmes utilitaires (compilateurs, éditeurs, shells,...)
3. Les fonctions de bibliothèques
4. Les appels système
5. Le noyau
6. Les pilotes de périphériques
7. Le matériel (CPU, contrôleurs, ...)

L'OS est formé des couches 3 à 6.

Rôles d'un OS

- Plusieurs composants:
  - Gestion des processus
  - Gestion mémoire (principale et secondaire)
  - Gestion des E/S (fichiers et réseau)
- Deux dimensions:
  - Etendre la machine (ajout de fonctionnalités)
  - Abstraire la machine (indépendance / matériel)

Gestion des processus

- Création et suppression
- Ordonnancement (allocation du processeur à un processus)
- Synchronisation (accès aux données/ressources partagées)
- Communication (échange d'informations entre processus)
- Prévention, détection et résolution des interblocages

Gestion de la mémoire principale

- Maintien d'une carte des zones occupées par processus
- Allocation/libération mémoire
- Stratégie d'allocation (pagination, segmentation, ...)

Gestion de la mémoire secondaire

- 2 tâches: mémoire virtuelle et stockage des fichiers
- Gestion des blocs disponibles
- Ordonnancement des requêtes au disque

Attention: grande différence de performances (accès mémoire $10^{-9}s$, accès fichier $10^{-3}s$)!

Gestion des E/S

L'OS comprend

- une interface standard pour les pilotes de périphériques (haut niveau);
- des pilotes liés à un matériel donné (bas niveau);
- des programmes de gestion des interruptions;
- des procédures de gestion des erreurs.

Une opération d'E/S

1. Le programme utilisateur demande une lecture (par exemple sur CD-ROM)
2. L'interface transmet la demande au pilote de CD
3. Le pilote place le code de l'opération (ici, lecture) dans les registres du contrôleur
4. Le contrôleur effectue l'opération
5. Si OK, le contrôleur envoie une interruption au pilote. Sinon,
    1. le contrôleur tente de résoudre le problème
    2. en cas d'échec, il demande au pilote de le faire (ex: relecture)
    3. en cas d'échec, le pilote renvoie une erreur

Gestion des fichiers / du réseau

Fichiers:

- Vision uniforme et organisée des données (arborescence)
- Masquage des détails de l'organisation physique

Réseau:

- Connexion à distance
- Partage de fichiers / périphériques
- Envoi / réception de messages

Modes d'exécution

Mode *utilisateur*

- restrictions d'accès aux ressources,
- jeu d'instructions processeur réduit

Mode noyau (ou superviseur)

- accès libre aux ressources
- jeu d'instructions complet

Noyau préemptible (un processus en mode noyau peut être interrompu par un autre) ou non

Typologie des OS

- Mono(multi)-tâche, mono(multi-)utilisateur, IHM, ...

Outline

1. Introduction
2. Gestion des processus
   - Introduction
   - Fonctions utiles sous UNIX
   - Ordonnancement
3. Communication entre processus
   - Communication par signaux
   - Communication par tubes
   - Synchronisation
   - Les sémaphores
   - Les sémaphores sous UNIX
   - Communication par mémoire partagée
   - Communication par files de messages
4. Gestion des entrées-sorties
   - Introduction
   - Méthodes d'accès
   - Méthodes d'allocation

Les processus

Definition

Processus = programme en cours d'exécution

Tout processus

- est représenté dans le système par un PCB (Process Control Block)
- est dans un état qui lui permet ou lui interdit certaines opérations
- peut communiquer des informations avec d'autres processus
- peut concourir avec d'autres processus pour l'obtention d'une ressource (CPU, périphérique, ...)

Structure partielle du PCB

1. Gestion du processus: registres, compteur ordinal, pointeur de pile, état, priorité, PID, PPID, groupe, signaux, heures, ...
2. Gestion de la mémoire: adresse du segment de texte, du segment de données, du segment de pile, ...
3. Gestion de fichiers: répertoires racine et courant, descripteurs de fichiers, UID, GID,...

Ces PCB sont rangés dans un tableau appelé Table des processus

Etats d'un processus

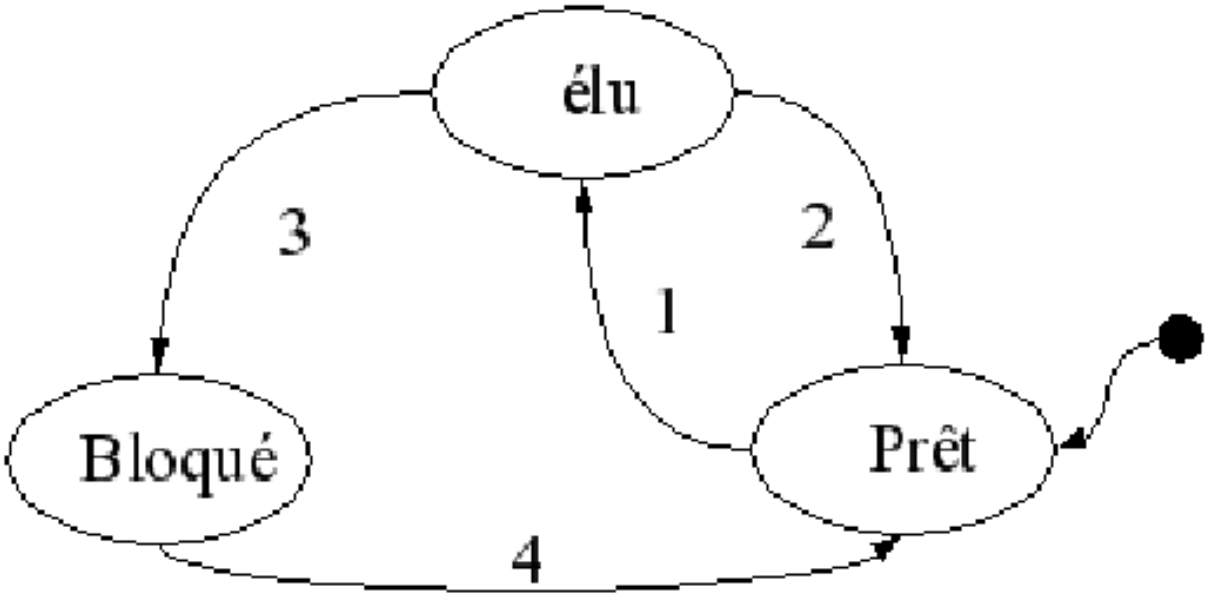


Figure: Les états d'un processus

1. L'ordonnanceur a désigné le processus
2. L'ordonnanceur a provoqué un changement de contexte
3. Le processus attend une information ou un événement
4. L'information est disponible ou l'événement survient

Le mode utilisateur

- Le mode par défaut
- Indiqué dans le PSW (Process Status Word) du processus
- Les instructions manipulant les interruptions sont inaccessibles
- Lors de l'invocation d'un *appel système*, le processus bascule en mode noyau

Le mode noyau

- Noyau non préemptible (ex. Linux): le processus libère de lui-même le processeur
- Quand le processus est en mode noyau, il exécute le code de l'OS et pas celui du programme → DEUX piles d'exécution

Commutation de contexte

Le passage du mode *user* au mode *kernel* constitue une commutation de contexte:

- sauvegarde du contexte utilisateur (principalement valeur des registres CO, PSW) dans la pile noyau
- chargement d'un contexte noyau avec dans le CO l'adresse de la fonction à exécuter
- déroulement de la fonction
- autre commutation de contexte pour retourner au mode utilisateur

Causes d'une commutation de contexte

3 causes majeures possibles:

1. le processus invoque une fonction système (explicite);
2. le processus exécute une opération illicite (division par 0, violation mémoire,...): provoque une exception, ou trappe (arrêt du processus);
3. le système reçoit une interruption matérielle (IRQ, Interrupt ReQuest): provoque l'exécution de la routine d'interruption associée

Gestion des interruptions Ex. Linux: chaque interruption (matérielle (IRQ) ou logicielle (trappe)) est repérée par un entier sur 8 bits, le *vecteur d'interruption*.

- 0 $\rightarrow$ 31: interruptions non masquebles, exceptions;
- 32 $\rightarrow$ 47: interruptions masquables levées par les périphériques (IRQ)
- 128: appels système
- autres: utilisables pour les trappes autres que celles émises par l'OS

Une table des vecteurs d'interruption est placée en mémoire centrale et associe à chaque valeur l'adresse d'une routine de gestion.

La CPU vérifie à chaque instruction si une interruption a été reçue.

Outline

1. Introduction
2. Gestion des processus
   - Introduction
   - Fonctions utiles sous UNIX
   - Ordonnancement
3. Communication entre processus
   - Communication par signaux
   - Communication par tubes
   - Synchronisation
   - Les sémaphores
   - Les sémaphores sous UNIX
   - Communication par mémoire partagée
   - Communication par files de messages
4. Gestion des entrées-sorties
   - Introduction
   - Méthodes d'accès
   - Méthodes d'allocation

Création - terminaison de processus

2 techniques principales de création:

1. Création "ex nihilo" (ex: `CreateProcess`, Windows)
2. Clonage (ex: `fork`, UNIX)

Dans le second cas,

- on aboutit à une arborescence de processus (1: `init`)
- les deux processus ne diffèrent que par leurs PID / PPID

Création de processus sous UNIX

Sous UNIX:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork();
```

Valeur de retour:

- -1 en cas d'échec du clonage
- 0 dans le fils
- le PID du fils dans le père

Terminaison sous UNIX

```
#include <stdlib.h>
void exit(int status);
```

- Par convention, un `status` de 0 indique une fin normale. Tout retour non nul doit être documenté.
- Les processus fils (*zombis*) sont "repris" par `init`
- Le processus père reçoit `SIGCHLD`

Statut des fonctions

- Cas général: fonctions entières, retournent 0 en cas de succès et -1 en cas d'échec
- Variable errno pour détails sur l'échec:

```
#include <stdio.h>
void perror(const char *s);
#include <errno.h>
const char *sys_errlist[]; // messages d'erreur std
int sys_nerr;  // taille du tableau
int errno;  // indice erreur courante
```

Accès aux PID

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid();
pid_t getppid();
```

L'accès au PID du fils se fait par retour de `fork()`

Attente (sleep)

Permet d'attendre pendant une durée maximale fixée

```
#include <unistd.h>
unsigned int sleep(unsigned int nbSec);
```

- renvoie 0 à expiration du délai;
- renvoie le nombre de secondes restantes si le processus reçoit un signal

nanosleep

```
#include <time.h>
int nanosleep (const struct timespec *req,
                     struct timespec *rem); // = 0
 struct timespec [
   time_t tv_sec;
   long tv_nsec;
}
```

- renvoie 0 à expiration du délai
- renvoie -1 si interrompue (`rem` contient le temps restant)

Gestion du temps

```
#include <sys/time.h>
#include <time.h>
int gettimeofday(struct timeval *tv, struct timezone *tz);
int settimeofday(const struct timeval *tv , const struct ti

struct timeval {
  time_t      tv_sec;     /* seconds */
  suseconds_t tv_usec;    /* microseconds */
};
```

Retournent 0 en cas de succès et -1 en cas d'échec (utiliser errno)

Attente (wait)

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);  // = 0
pid_t waitpid(pid_t pid, int *status, int options);  // = 0
```

- pid: 0 pour un processus fils quelconque, > 0 pour un fils particulier
- Retour (sur fin de fils ou réception de signal):
  - le PID du fils terminé
  - -1 si erreur (ou pour waitpid si interrompue):

```
while (waitpid(0,0,0)<0);
```

Substitution d'image (exec)

La famille des fonctions `exec...` permet de remplacer l'image du processus courant par une autre
Pas de retour en cas de succès

```
#include <unistd.h>
int execlp(const char* path, const char* arg0, ..., 0);
```

Le `path` peut être incomplet si la commande est dans le `PATH`:

```
execlp("ls", "ls", "-l", 0);
```

Outline

1. Introduction
2. Gestion des processus
   - Introduction
   - Fonctions utiles sous UNIX
   - Ordonnancement
3. Communication entre processus
   - Communication par signaux
   - Communication par tubes
   - Synchronisation
   - Les sémaphores
   - Les sémaphores sous UNIX
   - Communication par mémoire partagée
   - Communication par files de messages
4. Gestion des entrées-sorties
   - Introduction
   - Méthodes d'accès
   - Méthodes d'allocation

Ordonnancement

1. Long terme: décider du moment où les programmes doivent être chargés en mémoire
2. Moyen terme: décider de la suspension / reprise de certains processus (swapping)
3. *Court terme*: choisir le prochain processus à exécuter parmi les procesus prêts.

Propriétés:

- doit éviter la famine (un processus n'obtient jamais la ressource)
- doit optimiser le taux d'utilisation de la ressource, le débit (nb de processus/s), le temps réel d'exécution, le temps de réponse,... (critères presque toujours exclusifs: choisir!)

Ordonnancement (2)

Quelques grandeurs pertinentes:

1. Le *débit*: nombre de processus par unité de temps (OS)
2. Le *taux d'occupation* de la ressource (OS)
3. Le *temps de restitution* (durée entre la soumission et la fin du processus) (user)
4. Le *temps de réponse* (durée entre la soumission et la production des premiers résultats) (user)
5. Le *temps d'attente* (temps passé dans la file des processus prêts) (user)

On considère deux types de cycles (UC et E/S)

Ordonnancement (3)

Ordonnancement non préemptif: un processus élu le reste tant qu'il n'est pas bloqué ou terminé

- FIFO,
- SJF,
- priorités simples

Ordonnancement préemptif: possibilité d'interruption (durée maximale d'exécution sans interruption atteinte, arrivée de processus éligibles plus prioritaires,...) pour un processus élu.

- SRJF
- Round-robin
- multi-niveaux

FIFO

Les processus prêts sont dans une FIFO

Avantage:

- simple
- pas de famine

Inconvénients:

- temps d'attente moyen important
- peu adapté au temps partagé

Jeux de test

2 jeux de test (on mesure le temps moyen d'attente et on ne considère qu'un cycle d'UC – voir les résultats un peu plus loin):

- arrivées distinctes:

proc.	date	durée	FIFO
P1	0	7	0
P2	1	4	6
P3	2	7	9
P4	3	5	15

$\bar{t_a} = 7,5$

- arrivées communes:

proc.	durée
P1	30
P2	3
P3	3

$\bar{t_a} = 21$

SJF

Shortest Job First: le processus élu a le plus court cycle d'UC.
Avantages:

- Le meilleur pour le temps moyen d'attente dans les non-préemptifs

Inconvénients:

- Risque de famine
- Peu adapté au temps partagé (risque de monopolisation de la CPU)
- Nécessite de connaître à l'avance le temps de cycle! (adapté au traitement par lot où l'on demande une estimation à l'utilisateur)

Algorithme à priorités simples

Catégorisation des processus par une priorité entière (en général, petite valeur = grande priorité)
A chaque valeur de priorité correspond une FIFO
La recherche du processus élu commence dans la file la plus prioritaire
Avantages:

- Simple (FIFO)
- Générique (FIFO et SJF en sont des cas particuliers: $t_{restant} \cong \frac{1}{prio}$)

Inconvénients:

- Risque de famine (solution: augmenter la priorité des processus qui attendent depuis longtemps)

SRJF

Shortest Remaining Job First, préemptif
L'algorithme choisit le processus ayant le plus court temps de cycle restant
Si les processus démarrent en même temps, équivalent à SJF
Avantages:

- Efficace (mieux que SJF)

Inconvénients:

- Risque de famine
- Il faut connaître à l'avance la durée du cycle

Round-Robin (tourniquet)

Le processus alloue un *quantum* de temps avant un changement de contexte.
Les processus prêts sont dans une FIFO
Avantages:

- Pas de monopolisation de l'UC, équitable
- Pas de famine
- Bon temps de réponse

Inconvénients:

- Temps d'attente moyen en général important
- Influence de la valeur du quantum difficile à cerner (grand → FIFO, petit → perte de temps dans les changements de contexte)

Multi-niveaux

Catégorisation des processus, mais chaque catégorie a sa propre politique d'ordonnancement (ex. FIFO pour les batch, RR pour les interactifs)
Chaque file est prioritaire par rapport aux files inférieures (interruption du processus en cours lors de l'arrivée d'un processus plus prioritaire)
Avantages:

- Permet une catégorisation des tâches accomplies par le système

Inconvénients:

- Risque de famine
- Priorités statiques (dépendent de la nature du processus)

Exemple de Multi-niveaux

- La priorité d'un processus évolue au cours du temps.
- N files à priorité décroissante, le quantum dans la file i vaut q^i.
- A l'expiration du quantum, le processus descend dans la file inférieure
- La dernière file est régie par un FIFO non préemptif

Avantages:

- Flexibilité (nb files, algo pour chaque file, quantum, certains types de tâche peuvent commencer dans une file peu prioritaire)

Inconvénients:

- Famine
- Ajustement délicat des paramètres
- Nombreux changements de contexte

Comparaison

proc.	date	durée
P1	0	7
P2	1	4
P3	2	7
P4	3	5

FIFO	SJF	SRJF	RR(2)	RR(5)	RR(10)
0	0	9	14	14	0
6	6	0	7	4	6
9	14	14	14	14	9
15	8	2	14	11	15
7,5	7	6,25	12,25	10,75	7,5

proc.	durée
P1	30
P2	3
P3	3

FIFO	SJF	SRJF	RR(2)	RR(5)	RR(10)	RR(30)
0	6	6	6	6	6	0
30	0	0	6	5	10	30
33	3	3	7	8	13	33
21	3	3	6,33	6,33	9,66	21

Avec cycles d'E/S

- Un processus peut être bloqué en attente d'une E/S: il n'est plus éligible
- Simulation = alternance de cycles UC / E/S
- Hypothèses simplificatrices:
  - temps de commutation négligeable
  - E/S parallélisables (pas de FIFO des processus bloqués)
- Exemple simple avec 2 processus:

proc	t_{arr}	CPU_1	IO_1	CPU_2	IO_2	CPU_3
P_1	0	3	4	2	3	3
P_2	1	1	2	4	1	4

Exemple

Comparaison FIFO, SRJF et RR(2):

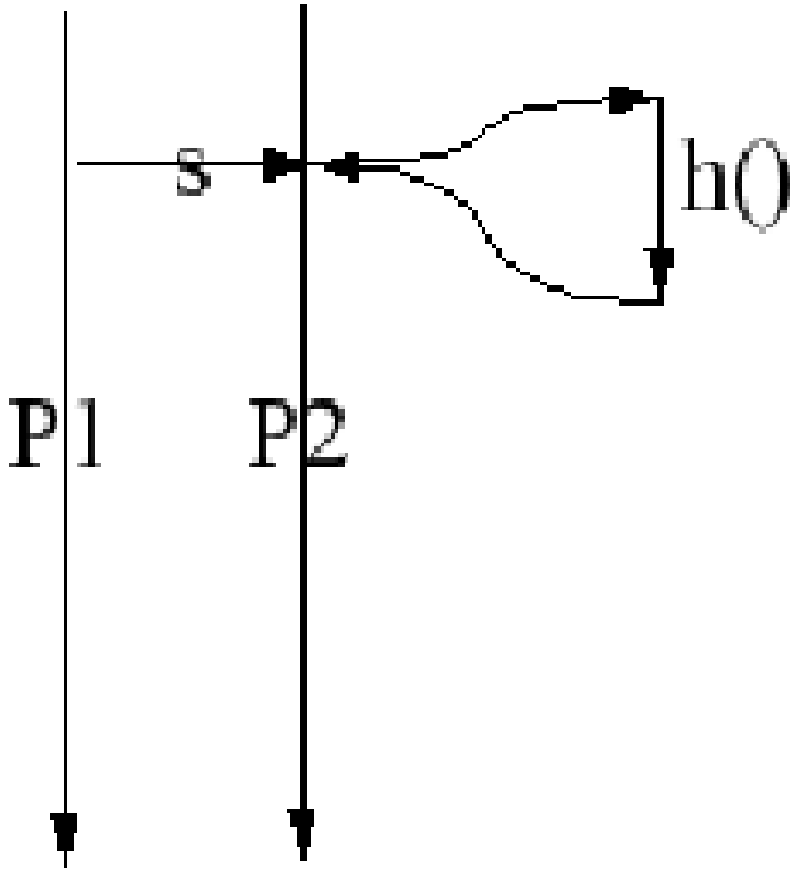
t	FIFO		SRJF		RR(2)	
	p1	p2	p1	p2	p1	p2
1	CPU		CPU		CPU	
2		W	W	CPU		W
3			CPU	IO	W	CPU
4	IO	CPU			CPU	IO
5		IO	IO		IO	
6				CPU		CPU
7		CPU				
8	W					
9			CPU	IO	CPU	W
10				W		
11	CPU	IO	IO		IO	CPU
12		W		CPU		IO
13	IO					CPU
14		CPU	W		W	
15			CPU		CPU	W
16	W					
17	CPU				W	CPU
18						
19					CPU	
20						
tAtt	3.5		1.5		4.5	
tRest	17		15		18	
tRep	1		0		0.5	

Outline

1. Introduction
2. Gestion des processus
   - Introduction
   - Fonctions utiles sous UNIX
   - Ordonnancement
3. Communication entre processus
   - Communication par signaux
   - Communication par tubes
   - Synchronisation
   - Les sémaphores
   - Les sémaphores sous UNIX
   - Communication par mémoire partagée
   - Communication par files de messages
4. Gestion des entrées-sorties
   - Introduction
   - Méthodes d'accès
   - Méthodes d'allocation

Communication par signaux

Communication par signaux: mécanisme simple, performant, asynchrone
Principe: similaire à une interruption logicielle



Signaux

- Le processus émetteur connaît le destinataire
- Le processus destinataire *ne connaît pas* l'émetteur
- Seule information: le numéro de signal (`kill -l`):
  - 1 (HUP), 2 (INT - ^C), 15 (TERM, défaut): exit proc
  - 3 (QUIT): core dumped
  - 9 (KILL): exit immédiat
- Les signaux SIGKILL et SIGSTOP (linux: 19) ne sont pas gérables

Etapes

1. Le processus destinataire déclare vouloir gérer un signal
2. Il associe au signal concerné une fonction de traitement (souvent à définir; profil: `void handle(int sigNum);`) grâce à la fonction `sigaction`
3. Le processus émetteur envoie ce signal au destinataire grâce à la fonction `kill`
4. Le code en cours est interrompu, dérouté sur la fonction de traitement
5. Le code en cours reprend après la fin de la fonction de traitement

Rappel sur les pointeurs de fonction en C

```
typedef void (*fpType)(int);
void f(int n) { ...}
...
fpType ptr = f;
ptr(3); /* équivalent à f(3); */
```

Envoi de signal (kill)

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

Envoie le signal sig au processus de PID pid.
Retour:

- 0 si OK
- -1 en cas d'erreur

Choix d'une fonction de traitement (sigaction)

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact); /* =0 */
struct sigaction {
  void (*sa_handler) (int);
/* SIG_DFL, SIG_IGN, fonction */
  ...
}
```

Associe la fonction pointée par act->sa_handler au signal signum.
Retour:

- 0 si OK
- -1 en cas d'erreur

Outline

1. Introduction
2. Gestion des processus
   - Introduction
   - Fonctions utiles sous UNIX
   - Ordonnancement
3. Communication entre processus
   - Communication par signaux
   - Communication par tubes
   - Synchronisation
   - Les sémaphores
   - Les sémaphores sous UNIX
   - Communication par mémoire partagée
   - Communication par files de messages
4. Gestion des entrées-sorties
   - Introduction
   - Méthodes d'accès
   - Méthodes d'allocation

Communication par tubes

Rappel: tube = connexion entre la sortie d'un processus et l'entrée d'un autre

Fonctions principales

- création d'un tube (`pipe()`)
- lecture / écriture (`read()` / `write()`)
- fermeture du tube (`close()`)

Création d'un tube

```
#include <unistd.h>
int pipe(int filedes[2]);
```

- `filedes[0]` est utilisé pour la lecture
- `filedes[1]` est utilisé pour l'écriture
- En général, un processus lit, l'autre écrit

Retourne 0 en cas de succès, -1 en cas d'échec

Lecture / écriture dans un tube

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

- buf désigne l'emplacement mémoire contenant les données à lire ou écrire
- count représente le nombre d'octets à transférer

Retourne le nombre d'octets effectivement transférés (-1 si erreur)

Fermeture du tube

```
#include <unistd.h>
int close(int fd);
```

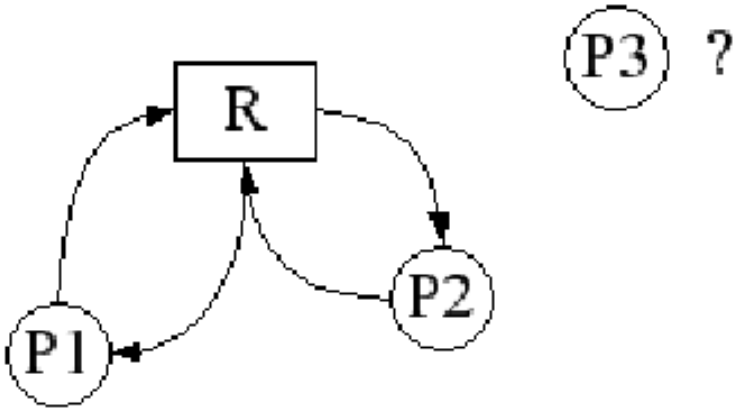
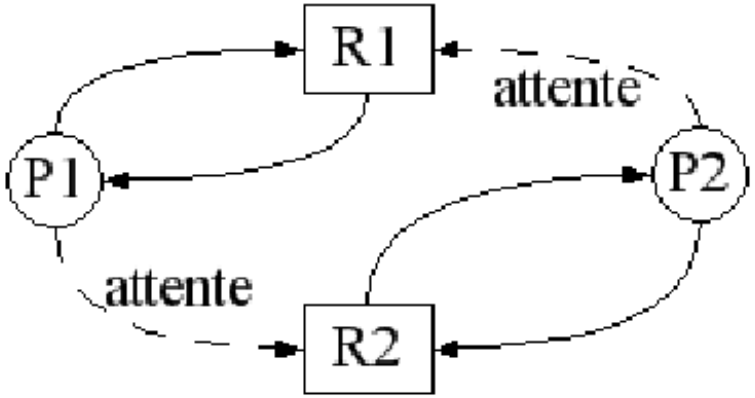
Retourne 0 en cas de succès, -1 en cas d'échec

Outline

1. Introduction
2. Gestion des processus
   - Introduction
   - Fonctions utiles sous UNIX
   - Ordonnancement
3. Communication entre processus
   - Communication par signaux
   - Communication par tubes
   - Synchronisation
   - Les sémaphores
   - Les sémaphores sous UNIX
   - Communication par mémoire partagée
   - Communication par files de messages
4. Gestion des entrées-sorties
   - Introduction
   - Méthodes d'accès
   - Méthodes d'allocation

Synchronisation

L'interaction entre processus peut être volontaire (collaboration) ou involontaire (partage de ressources)
Problèmes courants: famine, interblocage, gestion des accès concurrents

Famine	Interblocage
	

Accès concurrents

Ex1: deux guichets de réservation

Guichet 1	Guichet 2
`Si place 10 libre alors`	
	`Si place 10 libre alors`
	`  réserver place 10`
`  réserver place 10`	

Accès concurrents (2)

Ex2: magasin à deux entrées, mécanisme centralisé de comptage des clients (N)
Chaque processus effectue

```
tant que magasin ouvert
  si nouveau client
  alors N = N + 1
```

A bas niveau, l'addition est décomposée en opérations plus simples (assembleur):

```
MOV reg, N
INC reg
MOV N, reg
```

Accès concurrents (3)

Entrée 1	Entrée 2
MOV reg,N	
INC reg	
	MOV reg, N
	INC reg
	MOV N, reg
MOV N, reg	

N n'a augmenté que de 1!

Notion de section critique

L1, L2: accès en lecture à N
E1, E2: accès en écriture à N
Séquences possibles:

1. L1, L2, E2, E1 (ci-dessus): incorrect
2. L1, L2, E1, E2: incorrect
3. L2, L1, E1, E2: incorrect
4. L2, L1, E2, E1: incorrect
5. L1, E1, L2, E2: correct
6. L2, E2, L1, E1: correct

Le couple (L_i, E_i) est "insécable": *section critique*

Section critique et Exclusion mutuelle

Definition

Section critique: ensemble d'instructions exécutées soit totalement, soit pas du tout (non interruptible)

Definition

Exclusion mutuelle: à tout instant, on a au plus un processus dans une section critique donnée

Propriétés à assurer:

- Progression: un processus suspendu ne doit pas bloquer la progression des autres
- Attente bornée: aucun processus ne doit attendre indéfiniment une entrée en SC: pas d'interblocage, pas de famine.

Solutions possibles

- Au niveau logiciel applicatif
- Au niveau matériel
- Au niveau de l'OS

Solutions logicielles

On suppose que les processus suivent le schéma

```
while true
   [SC]
   [SNC]
```

On va ajouter une section d'entrée en SC et une de sortie de SC:

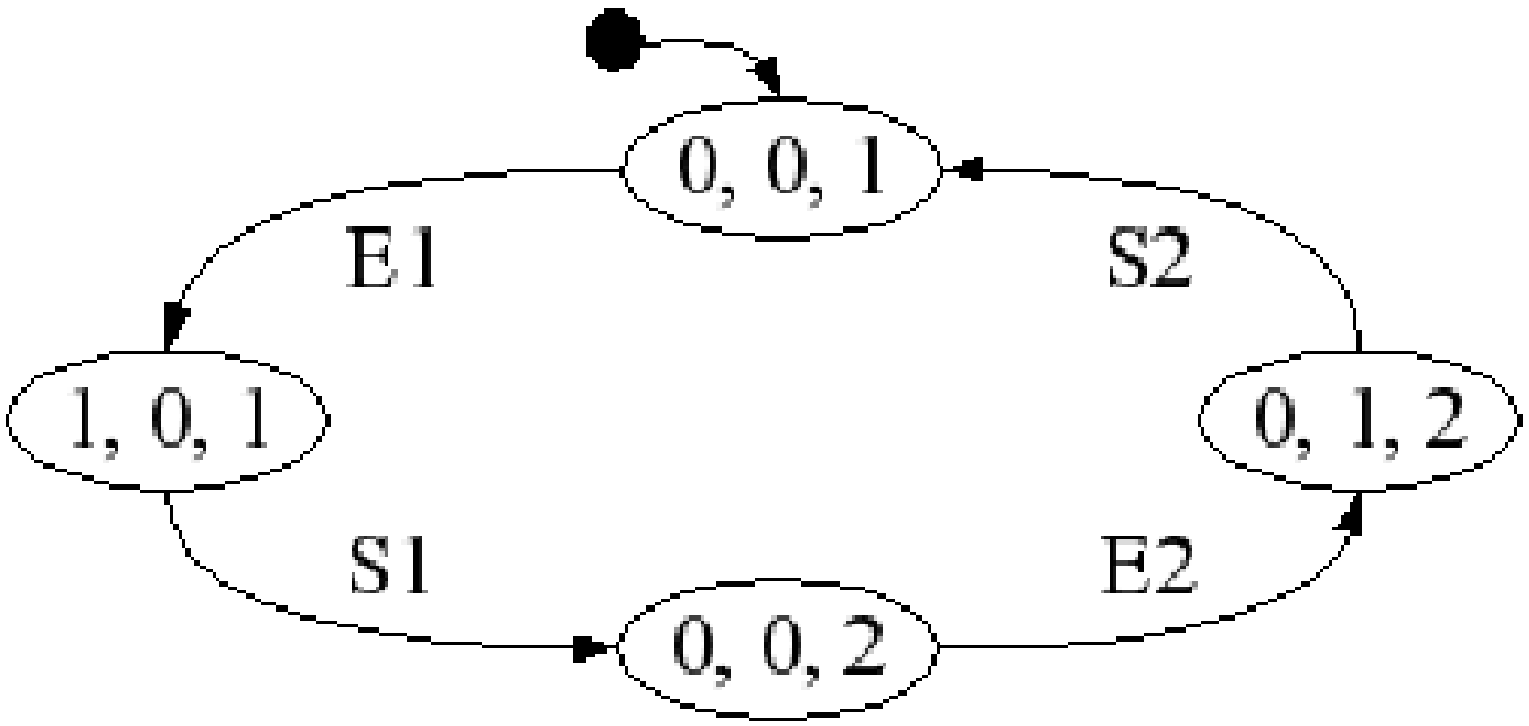
```
while true
   [E]
   [SC]
   [S]
   [SNC]
```

Première idée

Une variable entière (tour), partagée, indiquant le processus autorisé:

P1	P2
while true E1: tant que tour !=1; SC1:... S1: tour = 2; SNC1: ...	while true E2: tant que tour != 2; SC2: ... S2: tour = 1; SNC2: ...

Etats possibles



Triplets: (P1 en SC, P2 en SC, tour)

- Exclusion mutuelle: oui (pas d'état $(1, 1, i)$)
- Progression: non (si P1 termine, P2 est bloqué indéfiniment)

deuxième idée

Un tableau partagé de booléens:

P1	P2
while true tour[0] = 1; E1: tant que tour[1]; SC1:... S1: tour[0] = 0; SNC1: ...	while true tour[1] = 1; E2: tant que tour[0]; SC2: ... S2: tour[1] = 0; SNC2: ...

- Exclusion mutuelle: non (interblocage si changement de contexte juste après la première instruction).

troisième idée

"Echange de politesses"

P1	P2

while true
tour[0] = 1;
E1: tant que tour[1]
tour[0] = 0;
attendre
tour[0] = 1;
SC1:...
S1: tour[0] = 0;
SNC1: ...
	while true
	tour[1] = 1;
	E2: tant que tour[0]
	tour[1] = 0;
	attendre
	tour[1] = 0;
	SC2: ...
	S2: tour[1] = 0;
	SNC2: ...

Pb: si attente simultanée, les processus ne progressent pas

Algorithme de Peterson

1981. Mélange des idées un et deux:

- `actif` représente le numéro du proc. qui a "le droit d'insister"
- on conserve le tableau de booléens `tour` de la seconde idée

P1	P2

`while true`
tour[0] = 1; actif = 2;
`E1: tant que tour[1]`
`et actif == 2;`
`SC1:...`
`S1: tour[0] = 0;`
`SNC1: ...`
	`while true`
	tour[1] = 1; actif = 1;
	`E2: tant que tour[0]`
	`et actif == 1;`
	`SC2: ...`
	`S2: tour[1] = 0;`
	`SNC2: ...`

Algorithme de Peterson (2)

Exclusion mutuelle:

- P1 en SC: tour[0] et (!tour[1] ou actif == 1)
- P2 en SC: tour[1] et (!tour[0] ou actif == 2)
- P1 et P2 en SC: tour[0] et tour[1] et actif == 1 et actif == 2: impossible

Progression:

- supposons P1 en SNC: on a !tour[0]. Si P2 ne peut entrer en SC on a tour[0] et actif==1: impossible

Attente bornée:

- si P1 et P2 en entrée, P1 a fait au moins une SC. Alors on a tour[0] et actif == 2: P2 peut entrer en SC.

Algorithme de Lamport "Boulangerie"

Généralisation à $N > 2$ processus

- La ressource unique (la boulangère) sert un seul client à la fois
- Chaque client prend un ticket en entrant
- Le client servi est celui de plus petit numéro
- Quand le client sort, il jette son ticket

Variables partagées:

```
bool choix[N](= false): proc prêts à prendre un ticket
int num[N](= 0): les numéros des tickets tirés
```

Variables locales:

```
int j;
```

Algorithme de Lamport (2)

Pour le processus P_i:

```
while true
  choix[i] = true
  num[i] = 1 + max_j (num[j])
  choix[i] = false
  pour j de 1 à N
    si j != i
      while choix[j];
      while num[j] != 0 et
              (num[j] < num[i] ou
                   (num[j] == num[i] et j <i));
  SC_i
  num[i] = 0
  SNC_i
```

Algorithme de Lamport (3)

Exclusion mutuelle:

- vient de l'ordre FIFO sur le service des clients

Progression:

- si P_i est en SNC, on a num[i] == 0
- donc le "et" de la condition d'attente pour l'entrée de P_j en SC est faux: P_j ne peut pas être bloqué (du moins pas par P_i).

Attente bornée:

- P_i attend au pire $N-1$ tours (puisque quand P_j est passé en SC, on a $num[j] > num(i])$

Solutions matérielles

Problème des solutions logicielles:

- Complexes à mettre en œuvre
- Peu généralisables
- *Attente active*: les "`tant que ...;`"

Solutions matérielles:

1. Désarmement des interruptions
2. Existence d'instructions atomiques

Désarmement des interruptions

```
while true
   [desarmer]
   SC
   [armer]
   SNC
```

Problèmes:

- dangereux
- uniquement valable en environnement mono-processeur

Instructions atomiques

Principe: consultation et modification de variable insécables (Test&set):

`int TS(int *i) {`
`  if (!*i) {`
`  *i = 1;`
`  return 1;`
`  }`
`return 0;`
`}`
	`while true`
	`  while !TS(verrou);`
	`  SC`$_i$
	`  verrou = 0`
	`  SNC`$_i$

Problèmes:

- Attente active
- Pas d'attente bornée: risque de famine et d'interblocage

Solutions fournies par l'OS

- Sémaphores:
  - blocage de processus,
  - exclusion mutuelle,
  - partage de ressources
- Moniteurs:
  - plus haut niveau,
  - fonctions de suspension/reprise de procesus sous condition

Outline

1. Introduction
2. Gestion des processus
   - Introduction
   - Fonctions utiles sous UNIX
   - Ordonnancement
3. Communication entre processus
   - Communication par signaux
   - Communication par tubes
   - Synchronisation
   - Les sémaphores
   - Les sémaphores sous UNIX
   - Communication par mémoire partagée
   - Communication par files de messages
4. Gestion des entrées-sorties
   - Introduction
   - Méthodes d'accès
   - Méthodes d'allocation

Sémaphores

Definition

Un sémaphore est une varibale entière accessible par seulement deux opérations atomiques, P et V.
P: décrémente si valeur strictement positive
V: incrémente

```
void P(int * s) {
  while (*s <= 0);
  *s = *s - 1;
}
```

```
void V(int * s) {
  *s = *s + 1;
}
```

La solution concrète évite l'attente active par une FIFO des processus ayant exécuté l'opération P.

Applications des sémaphores

- Initialisé à 0: sémaphore bloquant

prof	etud
afficher(question) V(sem)	P(sem) répondre

- Initialisé à 1: assure l'exclusion mutuelle (mutex):

```
while true
  P(mutex)
  SC
  V(mutex)
  SNC
```

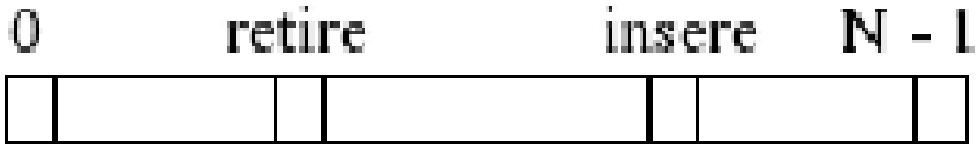
Applications (2)

- Initialisation à $N > 1$: partage de ressource (ex: prêt de N livres identiques)

Prêt	Retour
P(dispo)	rendre
prêter	V(dispo)
[lire]	

Exemple-type: Producteur/consommateur

On a une file “ciculaire” de N ressources disponibles:



(le producteur insère les nouvelles ressources par la droite, le consommateur les retire par la gauche!)
variables partagées:

- compteur: nombre de cases occupées (=0)
- tampon: le tableau représentant la file circulaire

Producteur	Consommateur

`Element o;`
`int insere = 0;`
`while (true) {`
`  o = produire()`
`  while (compteur == N);`
`  tampon[insere++] = 0;`
`  insere %= N;`
`  compteur++;`
`}`
	`Element o;`
	`int retire = 0;`
	`while (true) {`
	`  while (compteur == 0);`
	`  o = tampon[retire++];`
	`  retire %= N;`
	`  compteur--;`
	`  consommer(o);`
	`}`

Problème: les opérations sur compteur ne sont pas insécables!

Supposons que compteur == 5:

Producteur	reg	compteur	Consommateur
compteur++;		5	compteur--;
MOV reg, compteur	5	5	
INC reg	6	5	
	5	5	MOV reg, compteur
	4	5	DEC reg
MOV compteur, reg	4	4	
	4	4	MOV compteur, reg

Solution avec sémaphores

On utilise un `mutex` et deux sémaphores, `nonPlein` (= N) et `nonVide` (= 0):

Producteur	Consommateur

`while (true) {`
`  o = produire();`
`  P(nonPlein);`
`  P(mutex);`
`  insert(o);`
`  V(mutex);`
`  V(nonVide);`
`}`
	`while (true) {`
	`  P(nonVide);`
	`  P(mutex);`
	`  o = extract();`
	`  V(mutex);`
	`  V(nonPlein);`
	`  consommer(o)`
	`}`

Le dîner des philosophes

- N philosophes, N baguettes
- Un philosophe peut, soit penser, soit manger (s'il a deux baguettes)
- But: trouver une solution optimale sans famine ni interblocage

Solution 1

ressources = baguettes (N mutex: `tabMut`)
Code du processus P_i:

```
while true
  penser();
  P(tabMut[i]); P(tabMut[(i+1) % N]);
  manger();
  V(tabMut[(i+1) % N];
  V(tabMut[i];
```

Problème: interblocage (par exemple si chaque φ_i a pris sa baguette gauche)

Solution 2

ajout d'un mutex global
Code du processus P_i:

```
while true
  penser();
  P(mutex);
  P(tabMut[i]); P(tabMut[(i+1) % N]);
  manger();
  V(tabMut[(i+1) % N]; V(tabMut[i];
  V(mutex);
```

Problème: à tout instant, un seul philosophe peut manger!

Solution 3 (Dijkstra)

- Si un philosophe a faim *et* si ses deux baguettes sont libres, il mange; sinon il attend.
- Quand un philosophe a fini de manger il regarde si ses voisins veulent manger (test); si oui, il active leurs sémaphores respectifs

```
typedef enum { PENSE, FAIM, MANGE } Etat;
int gauche(int n) { return (n-1) % N; }
int droite(int n) { return (n+1) % N; }
Etat etats[N] = { PENSE, ...};
Semaphore autorise[N] = { 0, ...};
Semaphore mutex;
```

Processus φ_i:

```
while true
  penser();
  prendreBaguettes(i);
  manger();
  poserBaguettes(i);
```

```
void prendreBaguettes(int i) {
  P(mutex);
  etats[i] = FAIM;
  test(i);
  V(mutex);
  P(autorise[i]);
}
```

```
void poserBaguettes(int i) {
  P(mutex);
  etats[i] = PENSE;
  test(gauche(i));
  test(droite(i));
  V(mutex);
```

```
void test(int i) {
  if(etats[i] == FAIM &&
  etats[gauche(i)] != MANGE &&
  etats[droite(i)] != MANGE) {
    etats[i] = MANGE;
    V(autorise[i]);
  }
}
```

Conclusion: sémaphores complexes, risque d'erreurs important!

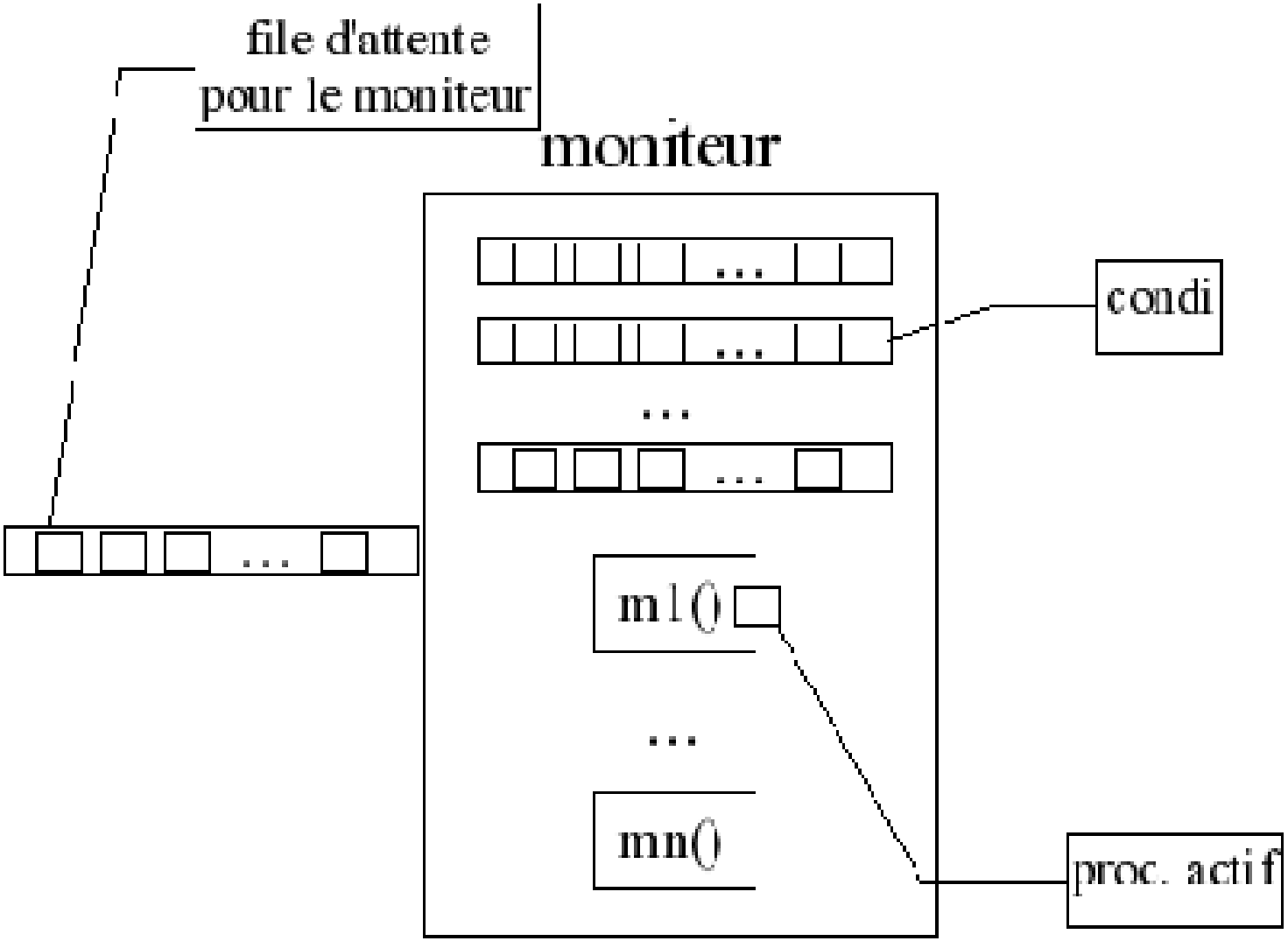
Moniteurs

Ce sont des objets qui

- garantissent 1 seul processus actif par moniteur: assurent l'exclusion mutuelle dans leurs méthodes
- fournissent des fonctions de suspension/reprise conditionnelle:

```
Cwait(condition); // wait until condition
Csignal(condition);
// réveille le premier en attente
```

Structure des moniteurs



Implémentation des moniteurs

A partir d'une classe standard, en ajoutant un mutex:

```
class Transformee {
  private Mutex mutex;
  ... // méthodes
  mi() { mutex.P(); <code>; mutex.V(); }
}
```

Les conditions

```
class Condition {
  File<PCB> file;
  Condition() { file = new File(); }
  Cwait() {
    <suspendre proc en cours>
    mutex.V(); // réveil du premier
    <insérer proc supendu dans la file>
    mutex.P(); }// pour l'unicité du proc actif
  Csignal() {
    if (!file.isEmpty()) {
      <retirer premier proc de la file>
      <reveiller proc> }}}
```

Exemple: producteur - consommateur

```
class ProdCons extends Moniteur {
  int cpt = 0, insere = 0, retire = 0;
  Element buffer[N];
  Condition nonPlein, nonVide;
  void inserer (Element o) {
    if (cpt == N) nonPlein.Cwait();
    buffer[insere++] = o;
    insere %= N;
    if (++cpt == 1) nonVide.Csignal();
  }
  Element retirer() {
    Element o;
    if (cpt == 0) nonVide.Cwait();
    o = buffer[retire++];
    retire %= N;
    if (--cpt == N-1) nonPlein.Csignal();
```

Programme test

```
ProdCons proc;
void producteur() {
  Element o;
  while(true) {
    o = produire();
    proc.inserer(o);
  }
}
```

```
void consommateur() {
  Element o;
  while (true) {
    o = proc.retirer();
    consommer(o);
  }
}
```

```
int main() {
  producteur();
  consommateur();
  return 0;
}
```

Les moniteurs existent de manière native dans certains langages (Java)

Moniteurs et sémaphores utilisent des zones de mémoire partagée: pas possible dans les systèmes distribués: on utilise alors des *envois de messages*

Outline

1. Introduction
2. Gestion des processus
   - Introduction
   - Fonctions utiles sous UNIX
   - Ordonnancement
3. Communication entre processus
   - Communication par signaux
   - Communication par tubes
   - Synchronisation
   - Les sémaphores
   - Les sémaphores sous UNIX
   - Communication par mémoire partagée
   - Communication par files de messages
4. Gestion des entrées-sorties
   - Introduction
   - Méthodes d'accès
   - Méthodes d'allocation

Les sémaphores sous UNIX

1. Associer une clé unique (généralisation de la notion d'adresse) à un *ensemble* de sémaphores
2. Créer l'ensemble de N sémaphores souhaité
3. Utiliser les sémaphores
4. Détruire les sémaphores grâce à la clé

Les clés

```
# include <sys/types.h>
# include <sys/ipc.h>
key_t ftok(const char *pathname, int proj_id);
```

Renvoie la clé unique en cas de succès, -1 si échec

Création de sémaphore(s)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

Appel typique (pour un seul sémaphore):

```
semid = semget(key,1,IPC_CREAT|IPC_EXCL|0600);
```

Retourne un identifiant (semid) unique pour l'ensemble créé, -1 en cas d'échec.

Initialisation d'un sémaphore

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```

Appel typique:

```
semctl(semid,0,SETVAL,valInit);
```

Initialise le premier sémaphore du tableau à la valeur vallnit.
Retourne -1 en cas d'échec, sinon le retour dépend de la commande (ex GETVAL)

Opérations sur sémaphore(s)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops);
struct sembuf {
  unsigned short sem_num;  /* semaphore number */
  short sem_op;  /* semaphore operation */
  short sem_flg;  /* operation flags */
};
```

Renvoie 0 (succès) ou -1 (échec)

- nsops: le nombre d'opérations à effectuer (taille du tableau sops)
- sem_op $>$ 0: valeur ajoutée à celle du sémaphore
- sem_op $=$ 0: wait for zero
- sem_op $<$ 0: décrémentation si valeur finale positive ou nulle, wait sinon

Exemple typique: un P() sur le premier sémaphore: le sembuf vaut { 0, -1, 0 }

Destruction de sémaphore(s)

On utilise la fonction `semctl` avec la commande `IPC_RMID`:

```
semctl(semid,0,IPC_RMID,0);
```

Supprime l'ensemble des sémaphores du tableau (`semnum` ignoré).
Retourne 0 en cas de succès, -1 en cas d'échec

ipcs et ipcrm

Deux commandes UNIX très utiles:

1. `ipcs` liste les ressources (sémaphores (-s), files de messages (-q), mémoire partagée (-m)) actuellement allouées:
   `ipcs [ -asmq ] [ -i id ]`
2. `ipcrm` libère les ressources spécifiées:
   `ipcrm [ -M key | -m id | -Q key | -q id | -S key | -s id ] ...`

Outline

1. Introduction
2. Gestion des processus
   - Introduction
   - Fonctions utiles sous UNIX
   - Ordonnancement
3. Communication entre processus
   - Communication par signaux
   - Communication par tubes
   - Synchronisation
   - Les sémaphores
   - Les sémaphores sous UNIX
   - Communication par mémoire partagée
   - Communication par files de messages
4. Gestion des entrées-sorties
   - Introduction
   - Méthodes d'accès
   - Méthodes d'allocation

Communication par mémoire partagée

Principe: un processus réalise effectivement l'allocation, les autres attachent juste la zone à leur espace d'adressage.
La référence à une zone de SM se fait soit à l'aide d'une clé, soit à l'aide d'un id (comme pour les sémaphores). Principales étapes:

1. Obtention d'une clé (`ftok`)
2. Obtention d'un id (`shmget`)
3. Attachement de la zone à une adresse locale valide (`shmat`)
4. Utilisation...
5. Détachement / destruction (`shmdt`, `shmctl`)

shmget

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

- retrouve ou crée une zone de SM associée à la clé
- retourne un id ou -1 si erreur
- flags: `IPC_CREAT|IPC_EXCL|0600` si création, juste les droits sinon

shmat, shmdt

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
```

- attache (détache) la zone de SM à (de) l'espace d'adressage du proc courant
- adresse et flags: 0 en général
- retournent l'adresse locale (shmat) et 0 (shmdt) si OK, -1 en cas d'erreur

shmctl

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- permet de retrouver des infos sur la zone de SM, ou de la détruire (`cmd = IPC_RMID`)
- retourne 0 (succès) ou -1 (erreur)

Outline

1. Introduction
2. Gestion des processus
   - Introduction
   - Fonctions utiles sous UNIX
   - Ordonnancement
3. Communication entre processus
   - Communication par signaux
   - Communication par tubes
   - Synchronisation
   - Les sémaphores
   - Les sémaphores sous UNIX
   - Communication par mémoire partagée
   - Communication par files de messages
4. Gestion des entrées-sorties
   - Introduction
   - Méthodes d'accès
   - Méthodes d'allocation

Communication par files de messages

Utilisation de mémoire partagée impossible dans les systèmes distribués: mécanisme d'envoi / réception de messages

1. liaison physique et logique émetteur / récepteur
2. send(dest, msg) et receive(src, msg)

La communication peut être directe (chat) ou indirecte (mail) par des BAL. Chaque BAL a une clé

- Envoi bloquant: E attend la réception du msg (AR) pour continuer
- Réception bloquante: R attend la fin de l'envoi pour continuer

Synchronisation

Exclusion mutuelle:

- les N processus partagent la même BAL (clé commune)
- EnB, RB (mécanisme de serveur)
- Au départ, la BAL contient 1 msg (NULL)

```
(Pi): while true
        receive(cle, msg)
        SCi
        send(cle, msg)
        SNCi
```

Producteur-consommateur

2 BAL distinctes, de taille N

- prod: autorisation d'ajout: N messages (NULL)
- cons: autorisation de retrait: 0 messages

Avantages / inconvénients:
(+) souplesse, valable pour N quelconque, généralité (env. distribué)
(-) sensible aux perturbations (réseau: blocage possible), performances

Fonctions de gestion des messages

1. Création/obtention d'une MQ: `msgget`
2. Envoi/réception de messages: `msgsnd/msgrcv`
3. Suppression d'une MQ: `msgctl`

msgget

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

- flags=0: obtention, IPC_CREAT | IPC_EXCL pour la création
- retourne un mqid ou -1 si erreur

msgsnd / msgrcv

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, struct msgbuf *msgp,
     size_t msgsz, int msgflg);
ssize_t msgrcv(int msqid, struct msgbuf *msgp,
     size_t msgsz, long msg-typ, int msgflg);
struct msgbuf {
   long mtype;      /* message type, must be > 0 */
   char mtext[1];   /* message data */
};
```

- le buffer doit être alloué à l'avance
- mtype(> 0) est local à l'application et identifie un traitement à effectuer
- retournent -1 en cas d'erreur. Sinon, `msgsnd` retourne 0 et `msgrcv` le nombre d'octets copiés dans le buffer.

msgctl

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

- cmd=IPC_RMID pour la destruction
- flags à 0
- retourne 0 en cas de succès, -1 en cas d'erreur

Outline

1. Introduction
2. Gestion des processus
   - Introduction
   - Fonctions utiles sous UNIX
   - Ordonnancement
3. Communication entre processus
   - Communication par signaux
   - Communication par tubes
   - Synchronisation
   - Les sémaphores
   - Les sémaphores sous UNIX
   - Communication par mémoire partagée
   - Communication par files de messages
4. Gestion des entrées-sorties
   - Introduction
   - Méthodes d'accès
   - Méthodes d'allocation

Gestion des entrées-sorties

- Rôles du SGF
- Concept de fichier
- Opérations sur les fichiers
- Méthodes d'accès
- Méthodes d'allocation
- Structure d'un disque
- Scheduling

Rôles du SGF

- Fournir un mécanisme d'accès et de stockage des données et programmes
- Gérer les accès concurrents aux fichiers (partage)
- Gérer les droits d'accès aux fichiers
- Optimiser les requêtes

Concept de fichier

Definition

Fichier = séquence de bits dont la sémantique est définie par l'applicationVue logique, indépendante du support physique (RAM, disque, bande,...)

- Plus petit niveau de granularité du SGF (insécable)
- Deux grands types: les fichiers "texte" et les fichiers "binaires" (*exemple*).
- Attributs classiques: nom, emplacement, taille, droits, propriétaire, timestamps...
- Les *répertoires* structurent l'espace mémoire de manière hiérarchique

Opérations sur les fichiers

- Création:
  - trouver de la place dans le SGF
  - créer une entrée de répertoire
- Lecture / écriture:
  - maintien par l'OS d'un pointeur de lecture / écriture sur le fichier (parfois séparés)
- Déplacement: juste une MAJ du pointeur: *aucune* E/S
  - troncage (RAZ): mêmes attributs sauf données effacées (taille = 0)

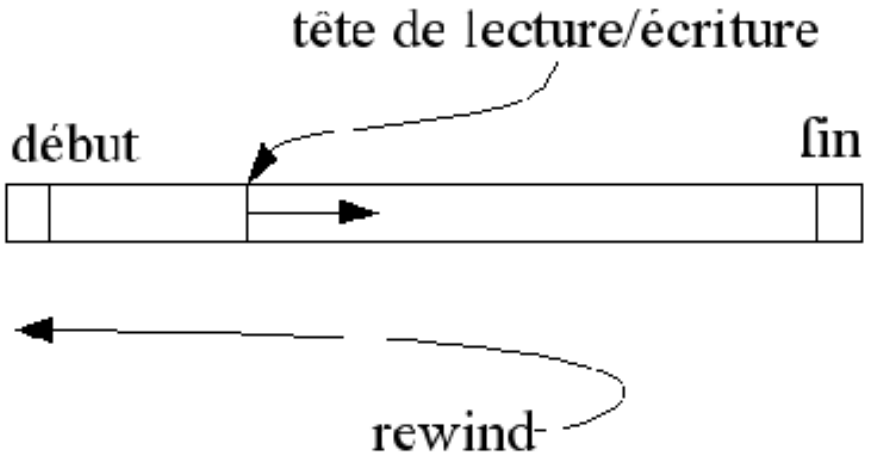
Toute autre opération s'exprime à partir de ces primitives

Outline

1. Introduction
2. Gestion des processus
   - Introduction
   - Fonctions utiles sous UNIX
   - Ordonnancement
3. Communication entre processus
   - Communication par signaux
   - Communication par tubes
   - Synchronisation
   - Les sémaphores
   - Les sémaphores sous UNIX
   - Communication par mémoire partagée
   - Communication par files de messages
4. Gestion des entrées-sorties
   - Introduction
   - Méthodes d'accès
   - Méthodes d'allocation

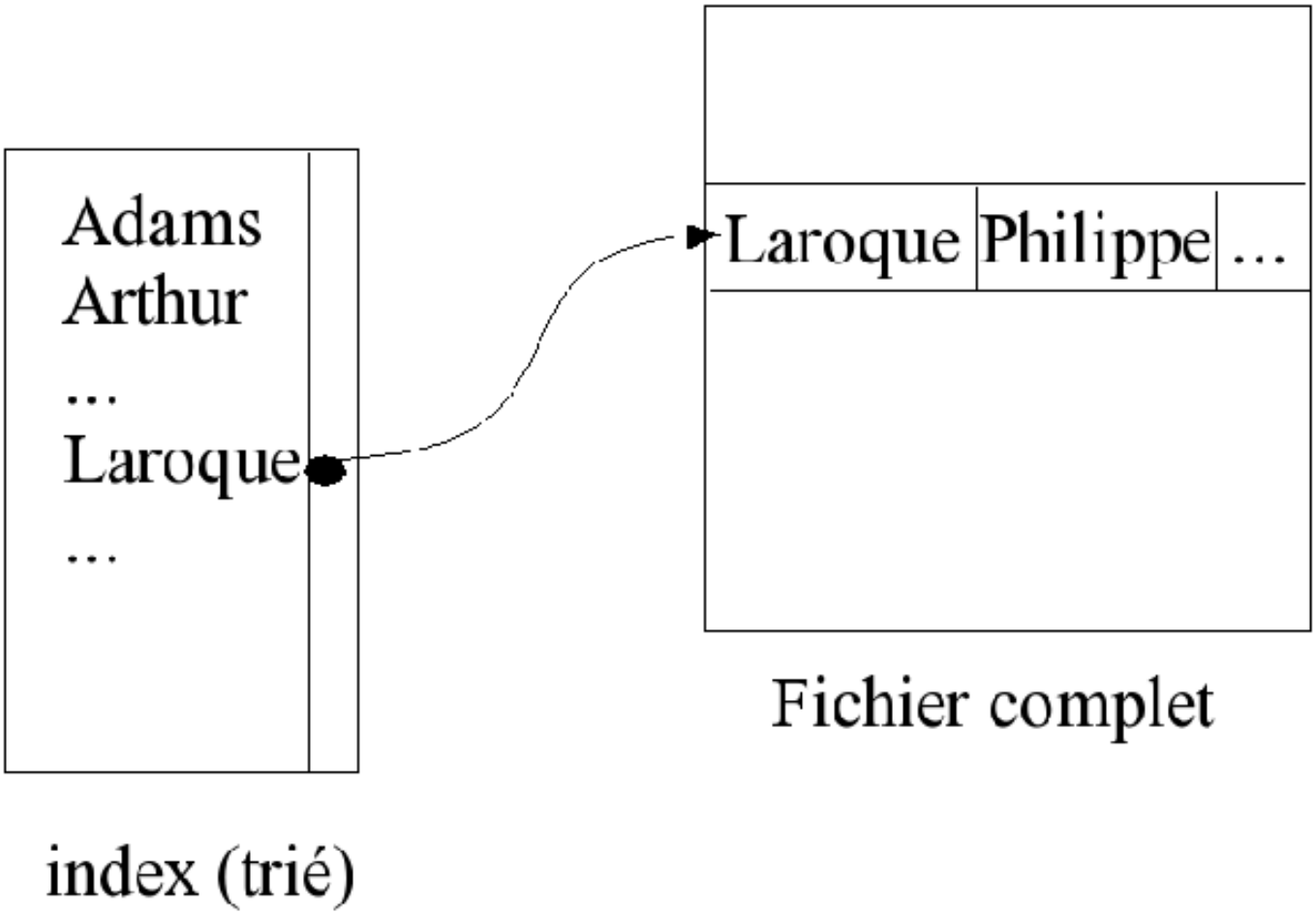
Méthodes d'accès

- Séquentiel:



- Direct: suppose une connaissance de la taille (fixe) des enregistrements du fichier
- Indexé: suppose l'existence d'un fichier d'index. Forme simple: accès au numéro d'enregistrement (maintenu en RAM si possible): permet la dichotomie et le direct)

Exemple de séquentiel indexé



ISAM: index principal en RAM, blocs d'index secondaires, fichier complet

Les répertoires

- Hiérarchie: disque / partition / répertoire / fichier
- Opérations supportées: recherche, création, suppression, renommage de ficbier; liste des fichiers maintenus
- Structure la plus fréquente: l'arborescence (Windows, UNIX, VMS, ...); notion de chemins relatif et absolu
- Cas d'UNIX: liens (hard et soft), montage

ATTENTION aux accès concurrents: le dernier qui parle a souvent raison (ex. UNIX)!!

Outline

1. Introduction
2. Gestion des processus
   - Introduction
   - Fonctions utiles sous UNIX
   - Ordonnancement
3. Communication entre processus
   - Communication par signaux
   - Communication par tubes
   - Synchronisation
   - Les sémaphores
   - Les sémaphores sous UNIX
   - Communication par mémoire partagée
   - Communication par files de messages
4. Gestion des entrées-sorties
   - Introduction
   - Méthodes d'accès
   - Méthodes d'allocation

Méthodes d'allocation

Partition divisée en *blocs* de taille fixe
Contiguë:

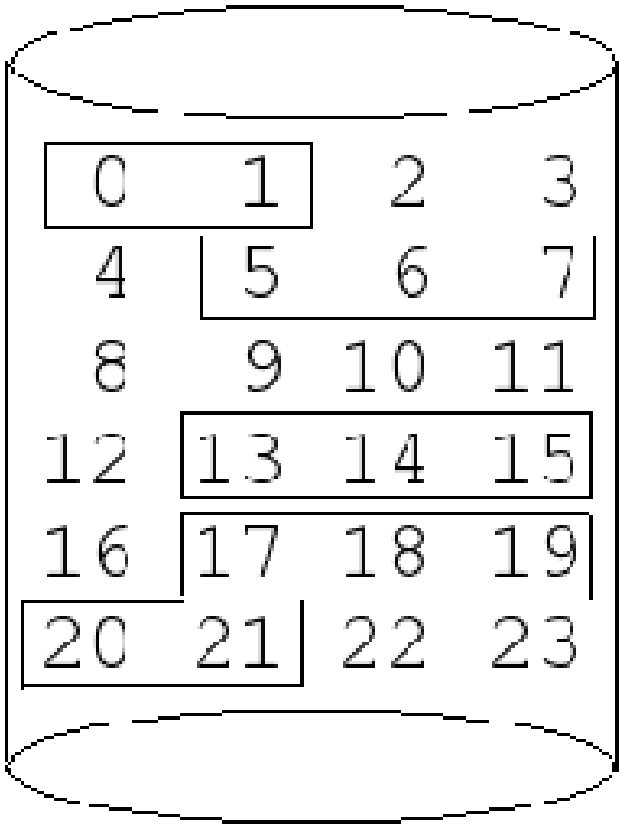
- nombre minimal de déplacements
- temps moyen de positionnement minimal

Mais:

- difficulté de gestion de l'espace disque (fragmentation)
- taille de fichier à estimer *a priori*: difficile

Allocation contiguë

ex: VM/CMS (IBM)



répertoire

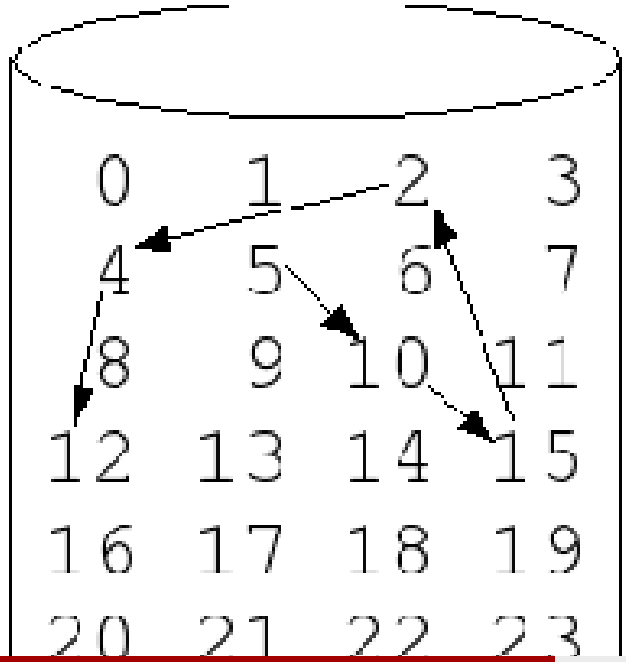
fichier	début	long
f1	0	2
titi	5	3
rominet	13	3
gus	17	5

Allocation chaînée

Résout les problèmes de l'allocation contiguë (pas de fragmentation, fichier de taille initiale nulle)

Mais:

- seul l'accès séquentiel est possible
- problème de fiabilité en cas de perte d'un pointeur
- perte de place pour le chaînage

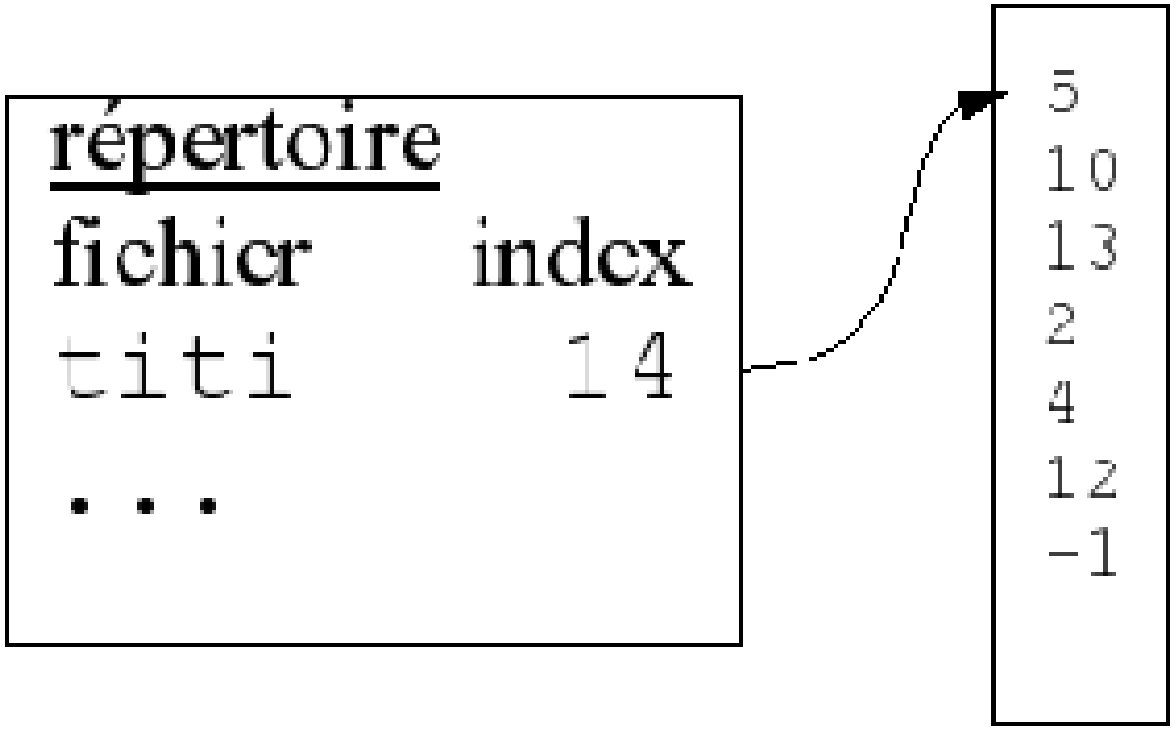


répertoire

fichier	début	fin
titi	5	12
...		

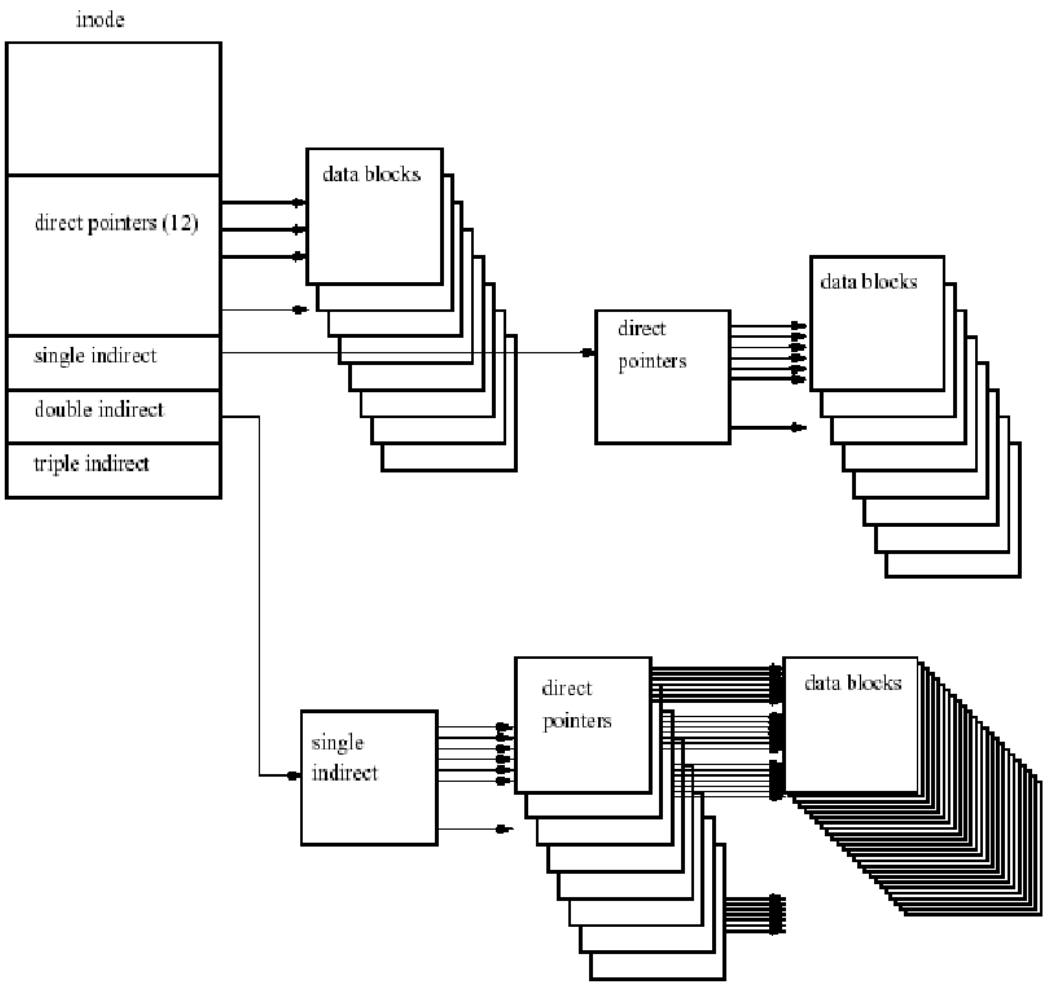
Allocation indexée

Résout le problème de l'accès direct en rangeant les pointeurs dans un bloc spécial, tableau de blocs disques (pb: perte de place disque plus importante)



En cas de gros fichiers, l'indexation doit être arborescente (ex: UNIX)

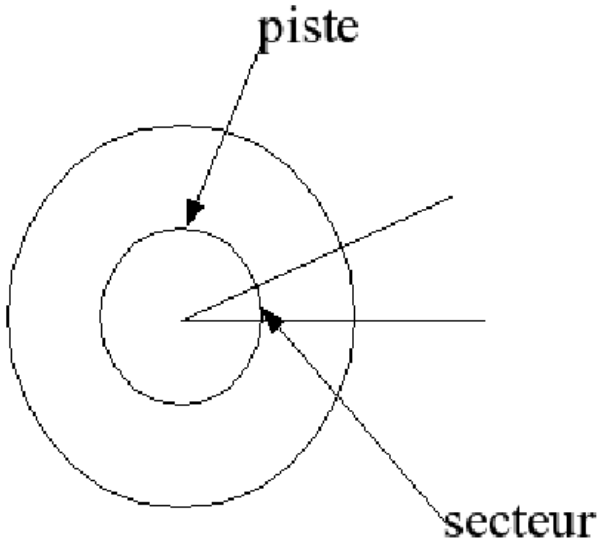
Exemple d'UNIX



Avec des blocs de 4 Ko, on obtient

$48 + 1024 * 4 + 1024^2 * 4 + 1024^3 * 4 \simeq 4\text{Go}$

Structure des disques



adresse en plusieurs parties: disque/face/piste/secteur
cylindre = { pistes accessibles sans déplacer la tête }
En général, secteurs de 32 à 4096 octets (512), 4 à 32 secteurs/piste, 20 à 1500 pistes/face

Performances des disques

1. Déplacement de la tête → piste cherchée (positionnement)
2. Attente → secteur cherché sous la tête (latence)
3. transfert par blocs de n secteurs (transfert)

Pb: minimiser les déplacements de têtes: scheduling des opérations d'E/S

Opération d'E/S

Si s secteurs par piste, p pistes par cylindre, un accès au cylindre i, face j, secteur k correspond au bloc b:
$b = k + s(j + ip)$
Chaque opération d'E/S correspond à un appel système avec

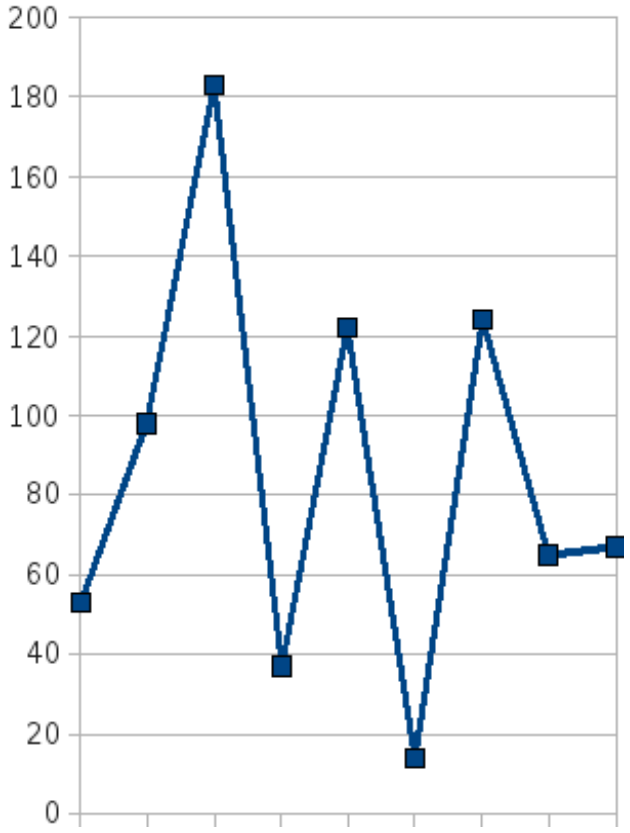
- Type de l'opération (lecture/écriture
- @ disque (numéro de bloc transformé comme indiqué ci-dessus)
- @mémoire pour le transfert
- taille à transférer

Plusieurs politiques possibles pour l'ordonnancement des requêtes d'E/S

L'algorithme FCFS

First come, first served

- + Facile à mettre en œuvre
- + "Juste"
- - peu efficace: 98, 183, 37, 122, 14, 124, 65,67 (tête sur 53 au départ)

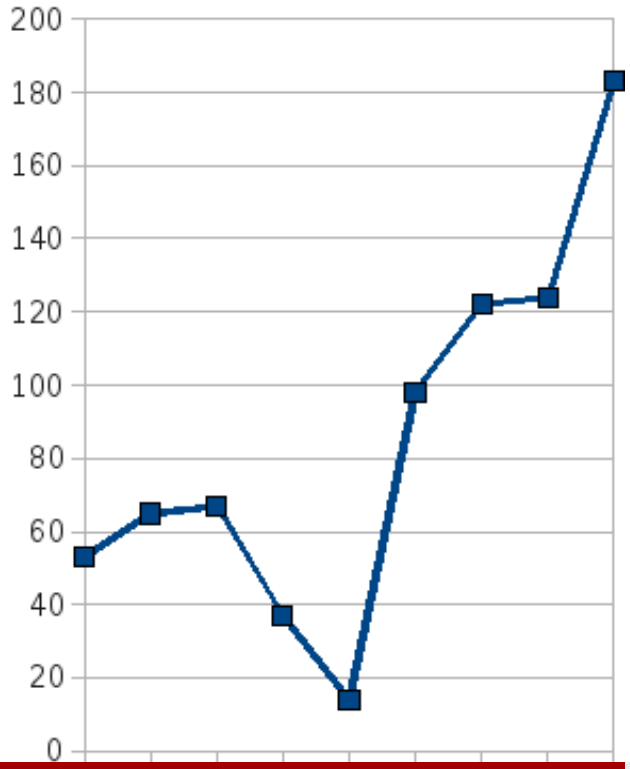


position	distance
98	45
183	85
37	146
122	85
14	108
124	110
65	59
67	2
TOTAL	640

L'algorithme SSTF

Shortest Seek Time First, très fréquent: on sert la requête la plus proche de la position courante

- + performant
- - risque de famine (cf SJF)
- - pas optimal (53 - 37 - 14 - 65 donne 208)

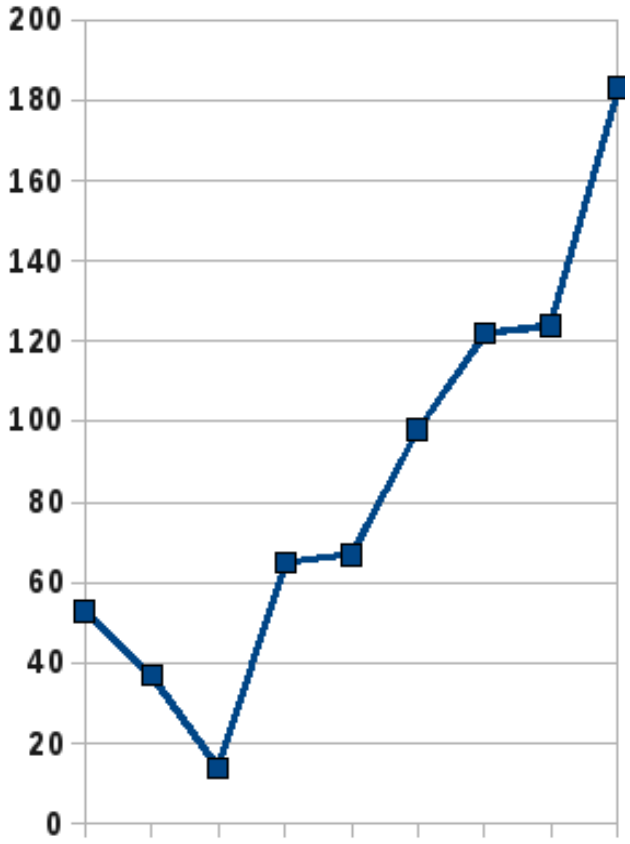


Position	Distance
65	12
67	2
37	30
14	23
98	84
122	24
124	2
183	59
TOTAL	236

L'algorithme SCAN

Balayage, ou "ascenseur": la tête va d'une extrémité à l'autre et sert les requêtes quand elle arrive sur la piste concernée. On suppose ici un déplacement vers la gauche.

- Variante: C-SCAN (circular): en fin de disque retour à 0 sans servir les requêtes
- LOOK (et C-LOOK): on repart dans l'autre sens dès la dernière requête servie



Position	Distance
37	16
14	23
65	51
67	2
98	31
122	24
124	2
183	59
TOTAL	208

Outline

1. Introduction
2. Gestion des processus
   - Introduction
   - Fonctions utiles sous UNIX
   - Ordonnancement
3. Communication entre processus
   - Communication par signaux
   - Communication par tubes
   - Synchronisation
   - Les sémaphores
   - Les sémaphores sous UNIX
   - Communication par mémoire partagée
   - Communication par files de messages
4. Gestion des entrées-sorties
   - Introduction
   - Méthodes d'accès
   - Méthodes d'allocation

Entrées-sorties sous UNIX (fonctions)

- Ouverture/création: `open` et `creat`
- Fermeture: `close`
- Lecture/écriture: `read`, `write`
- Déplacement: `lseek`
- Gestion des liens: `link`, `unlink`, `symlink`
- Création, suppression d'un répertoire: `mkdir`, `rmdir`
- Ouverture d'un répertoire: `opendir`
- Parcours d'un répertoire: `readdir`
- Fermeture d'un répertoire: `closedir`
- Autres fonctions sur les répertoires: `chdir`, `getcwd`
- Status d'un fichier: `stat`, `fstat`, `lstat`

open et creat

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

flags ∈ { `O_RDONLY`, `O_WRONLY`, `O_RDWR` } | O_CREAT, O_APPEND, O_TRUNC...

creat ⇔ `O_CREAT | O_WRONLY | O_TRUNC`

retourne -1 en cas d'échec

close

```
#include <unistd.h>
int close(int fd);
```

retourne 0 (succès) ou -1 (échec)

read, write

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

retournent le nombre d'octets lus(écrits), ou -1 en cas d'échec

lseek

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

whence $\in$ {SEEK_SET (début), SEEK_CUR(courante), SEEK_END(fin)}
retourne la nouvelle position (en octets) par rapport au début du fichier

link, unlink, symlink

```
#include <unistd.h>
int link(const char *oldpath, const char *newpath);
int symlink(const char *oldpath, const char *newpath);
int unlink(const char *pathname);
```

retournent 0(succès) ou -1(échec)

mkdir, rmdir

```
#include <sys/stat.h>
#include <sys/types.h>
int mkdir(const char *pathname, mode_t mode);
#include <unistd.h>
int rmdir(const char *pathname);
```

- retournent 0(succès) ou -1(échec)
- le répertoire doit être vide pour `rmdir`

opendir

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
```

retourne NULL (0) en cas d'échec.

readdir

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dir);
struct dirent {
  long d_ino;                    /* inode number */
  off_t d_off;                   /* offset to this dirent */
  unsigned short d_reclen;       /* length of this d_name */
  char d_name [NAME_MAX+1];      /* file name (null-terminated
};
```

retourne la prochaine (première) entrée de répertoire, NULL (0) en fin de liste

closedir

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dir);
```

retourne 0(succès) ou -1(échec)

chdir

```
#include <unistd.h>
int chdir(char *path);
int fchdir(int fd);
```

retournent 0 (succès) ou -1 (échec)

getcwd

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

retourne 0 si erreur (errno=ERANGE si `size` trop petit), le chemin (id. `buf`) sinon

stat, fstat, lstat

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *file_name, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
```

- lstat stat le lien et non le fichier pointé en cas de symlink
- fstat repère le fichier par son descripteur et non par son nom
- retournent 0(succès) ou -1(échec)

La structure stat

```
struct stat {
  dev_t st_dev;          /* device */
  ino_t st_ino;          /* inode */
  mode_t st_mode;        /* protection */
  nlink_t st_nlink;      /* number of hard links */
  uid_t st_uid;          /* user ID of owner */
  gid_t st_gid;          /* group ID of owner */
  dev_t st_rdev;         /* device type (if inode device) */
  off_t st_size;         /* total size, in bytes */
  blksize_t st_blksize; /* blocksize for filesystem I/O */
  blkcnt_t st_blocks;    /* number of blocks allocated */
  time_t st_atime;       /* time of last access */
  time_t st_mtime;       /* time of last modification */
  time_t st_ctime;       /* time of last change */
};
```

Le champ mode

				suid	sgid	sb									

- bits 1 à 4: type du fichier
- bits 8 à 16: les trois triplets de droits (u, g, o)
- `S_ISREG(mode)`: regular file
- `S_ISDIR(mode)`: répertoire
- `S_ISCHR(mode)`, `S_ISBLK(mode)`: périphérique en mode caractère/bloc
- `S_ISFIFO(mode)`, `S_ISLNK(mode)`, `S_ISSOCK(mode)`: Fifo, lien et socket

Séquence classique

1. `opendir` → DIR* dir // obtention du répertoire
2. `cur = readdir(dir)` // obtention entrée courante
3. `stat(cur->d_name,&buf)` // obtention infos sur le fichier
4. infos dans buf

Outline

1. Introduction
2. Gestion des processus
   - Introduction
   - Fonctions utiles sous UNIX
   - Ordonnancement
3. Communication entre processus
   - Communication par signaux
   - Communication par tubes
   - Synchronisation
   - Les sémaphores
   - Les sémaphores sous UNIX
   - Communication par mémoire partagée
   - Communication par files de messages
4. Gestion des entrées-sorties
   - Introduction
   - Méthodes d'accès
   - Méthodes d'allocation

Gestion de la mémoire

Plusieurs processus à maintenir en mémoire: ressource partagée

1. Mécanismes de liaison d'adresses
2. Espaces d'adressages logique et physique
3. Swapping
4. Allocation contiguë
5. Pagination
6. Segmentation
7. Mémoire virtuelle

Mécanismes de liaison d'adresses

Un programme traverse plusieurs étapes avant l'exécution:

- la compilation fait passer d'un module source à un module objet
- l'édition de liens transforme les modules objets (+ libs) en un exécutable

Chaque étape à une vision particulière des adresses:

- module source: adresses symboliques (`&x`)
- module objet: adresses translatables ("début module + 104")
- exécutable chargé: adresses absolues (23712)

Liaison d'adresses: conversion d'un espace d'adresses à l'autre

Moment de la liaison

- A la compilation: suppose qu'on connaît l'adresse où sera chargé le programme. En cas de changement, tout recompiler. Ex: les fichiers ".COM" sous DOS
- A l'édition des liens (le plus fréquent): le compilateur gébère du code relogeable (translatable). Si l'adresse change, il suffit de recharger le code utilisateur
- A l'exécution: réclame un matériel spécifique.

Quelques définitions

1. Chargement dynamique: le chargement du code d'une routine est effectué au moment où elle doit être exécutée (⇒ stockage en adresses relogeables)
2. Edition de liens dynamique: extension aux bibliothèques système (+ complexe car pb de partage de code entre processus)
3. Recouvrement: technique pour permettre à un processus plus "grand" que la mémoire physique de s'exécuter, en ne maintenant que les données et instructions nécessaires à chaque instant

Exemple de recouvrement

Un assembleur à deux passes:

- passe 1: génération de la table des symboles
- passe 2: génération du code machine

assembleur = { code passe 1, code passe 2, TS, routines communes } = { 700 KO, 800 KO, 200 KO, 300 KO }. Il faut 2 MO et seul 1,5 MO est disponible

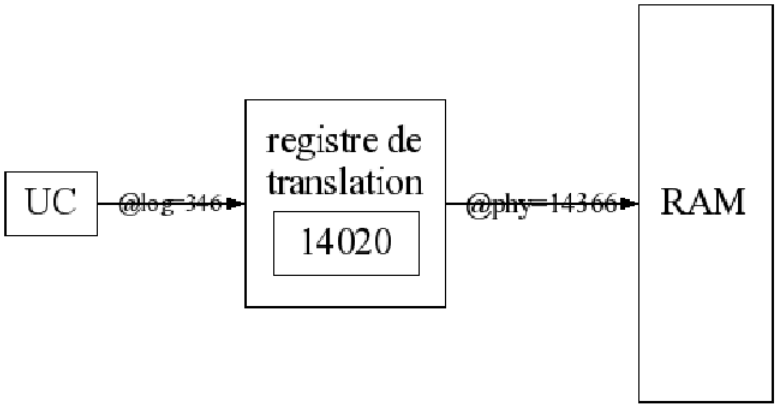
Recouvrement simple:

1. R1 = { CP1, TS, RC, driver } $\simeq$ 1,2 MO
2. R2 = { CP2, TS, RC, driver } $\simeq$ 1,3 MO (code driver recouvrement très petit)

Espaces d'adressages logique et physique

- Adresses logiques: générées par l'UC
- Adresses physiques: vues par l'unité de gestion de la mémoire (MMU, Memory Management Unit)
- Espace d'adressage logique = { @ logiques générées par un programme }
- Espace d'adressage physique = { @ physiques correspondantes }

La conversion entre les deux est prise en charge par la MMU



NB: les programmes utilisateurs ne traitent *QUE* des adresses logiques, ils ne voient aucune adresse physique

Swapping

Un processus peut être transféré temporairement RAM →disque, puis ramené pour poursuivre son exécution

- RAM → disque: swap-out
- disque → RAM: swap-in

Normalement le programme est replacé à la même adresse (superflu si liaison à l'exécution).
Le FS doit pouvoir

- contenir les images mémoire de tous les processus utilisateurs
- permettre l'accès direct à une image

Outline

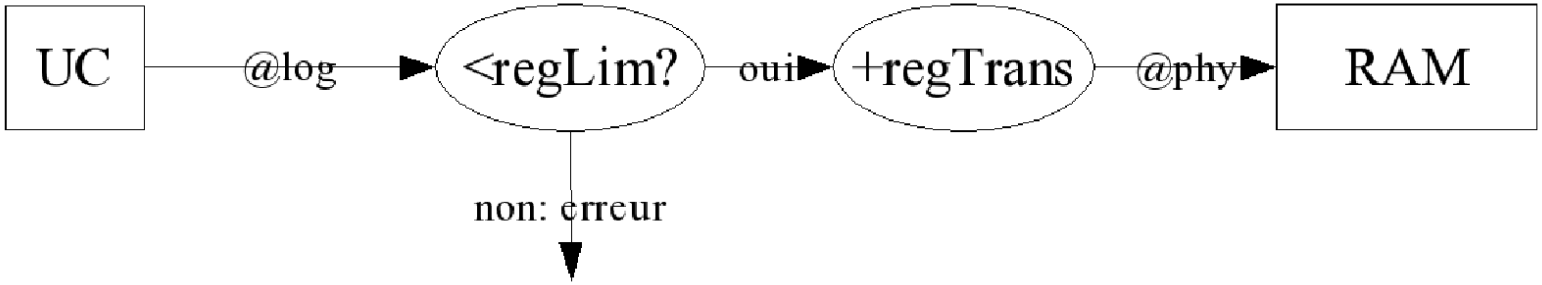
1. Introduction
2. Gestion des processus
   - Introduction
   - Fonctions utiles sous UNIX
   - Ordonnancement
3. Communication entre processus
   - Communication par signaux
   - Communication par tubes
   - Synchronisation
   - Les sémaphores
   - Les sémaphores sous UNIX
   - Communication par mémoire partagée
   - Communication par files de messages
4. Gestion des entrées-sorties
   - Introduction
   - Méthodes d'accès
   - Méthodes d'allocation

Allocation contiguë

En général, on a deux partitions (une pour l'OS, une pour les processus utilisateurs)
Protection nécessaire

- de la partition OS / procs utilisateurs
- de l'espace alloué à P_i / P_j

On utilise un registre de translation et un registre limite:



Pb: plusieurs processus concurrents, chacun avec ses propres besoins mémoire

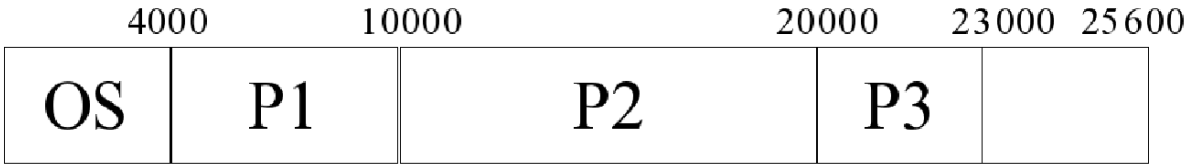
1. N partitions de même taille, 1 proc/partition (OS/360)
2. maintien d'une table des partitions libres

Exemple

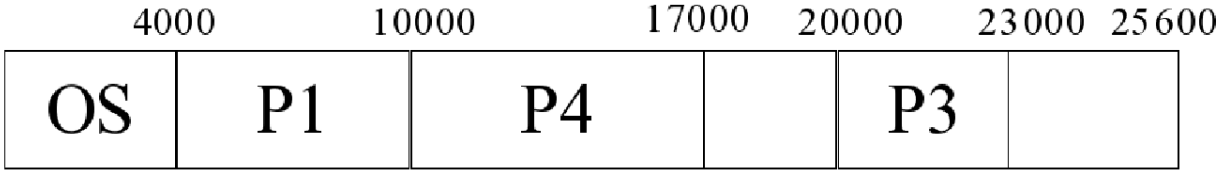
RAM=25600 KO
OS = 4000 KO

processus	*taille*	*temps*
P_1	6000	10
P_2	10000	5
P_3	3000	20
P_4	7000	8
P_5	5000	15

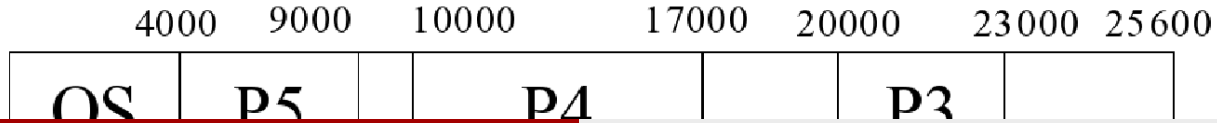
à t=0, on charge OS+P1+P2+P3:



Avec un RR de q=1, P2 se termine à t=14, on peut charger P4:



à t=28, fin de P1, on peut charger P5:



Stratégies possibles

- first-fit: on alloue le premier "trou" suffisamment grand
- best-fit: on alloue le plus petit suffisamment grand
- worst-fit: on alloue le plus grand (permet de garder de "grands" trous)

Pb de fragmentation externe: il peut rester assez de place, mais pas en contigu.
Exemple: avec un trou au lieu de 2, on aurait pu exécuter P5 en même temps que P1!
Règle du 50%: en first-fit, pour N blocs alloués, N/2 blocs perdus à cause de la fragmentation, soit 1/3 de la RAM!!!

Solution 1: compactage

- Possible uniquement si la translation d'adresses est dynamique

4000	9000	10000	16000	19000	25600
OS	P5	P4	P3		

- Plusieurs possiblités, de coût variable, par exemple

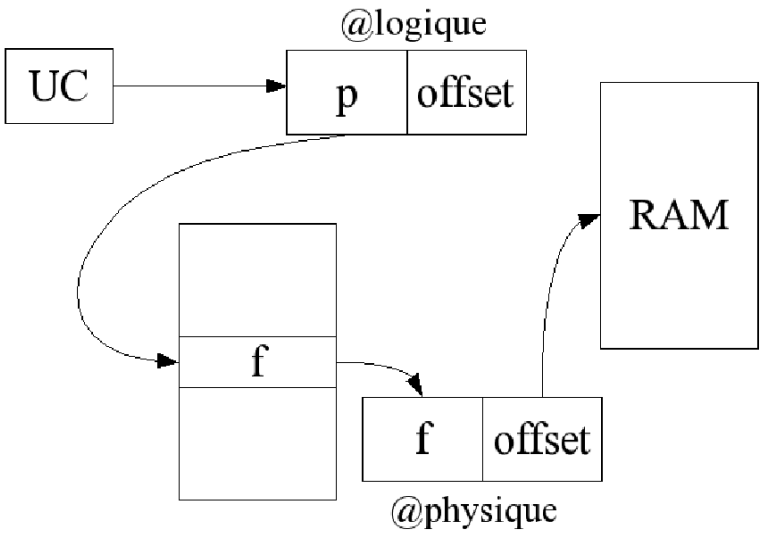
OS(0-3)	P1(3-5)	P2(5-6)	(6-10)	P3(10-12)	(12-15)	P4(15-19)	(19-21)	dépl
OS(0-3)	P1(3-5)	P2(5-6)	P3(6-8)	P4(8-12)	(12-21)			6
OS(0-3)	P1(3(5)	P2(5-6)	P4(6-10)	P3(10-12)	(12-21)			4
OS(0-3)	P1(3(5)	P2(5-6)	(6-15)			P4(15-19)	P3(19-21)	2

Pagination

Autre solution au problème de la fragmentation: permettre le chargement d'un programme dans une zone non contiguë de mémoire
Méthode de base: la mémoire φ est divisée en *cadres de page*, la mémoire logique en *pages*, la mémoire de swap en *blocs* (bloc $\neq$ page)
Chaque @ logique a deux parties:

1. le numéro de page (indice dans une table de pages)
2. l'offset dans la page

Taille d'une page: 2^N(facilité de décodage des sous-parties de l'@ si RAM=2^M)



@= | n° page (m-n bits) | offset (n bits) |

Exemple simple

$N = 2, M = 5$ (donc $2^3 = 8$ pages)

Page de 4 octets $\Rightarrow$ offset sur 2 bits

@φ=n°page φ * taille page + offset

RAM logique

@log	contenu
0	a,b,c,d
4	e,f,g,h
8	i,j,k,l
12	m,n,o,p
16	
20	
24	
28	

table pages

log	φ
0	5
1	6
2	1
3	2

RAM φ

@φ	contenu
0	
4	i,j,k,l
8	m,n,o,p
12	
16	
20	a,b,c,d
24	e,f,g,h
28	

ex: $(n)13 =$ | 0 | 1 | 1 | 0 | 1 | numéro page logique = 3, offset = 1

donc numéro page $\varphi = 2 \Rightarrow$ @$\varphi = 2 \times 4 + 1 = 9(n)$

exo: trouver 5

Remarques

- Avantages:
  - Supprime la fragmentation *externe*
  - Si la taille du processus est indépendante de celle de la page, on a en moyenne 1/2 page par processus de fragmentation *interne* (autant de chance pour qu'un processus se termine en début de page qu'en fin de page): pages de petite taille $\rightarrow$ peu de fragmentation interne, pages de grande taille $\rightarrow$ moins de décalages à calculer. Actuellement, les pages font 2 à 4 KO
- Inconvénients: sauvegarde de la TP avec le compteur d'instruction et les registres $\rightarrow$ augmentation du temps de commutation de contexte
- Une entrée de TP contient

n° cadre	bit de validité	permissions

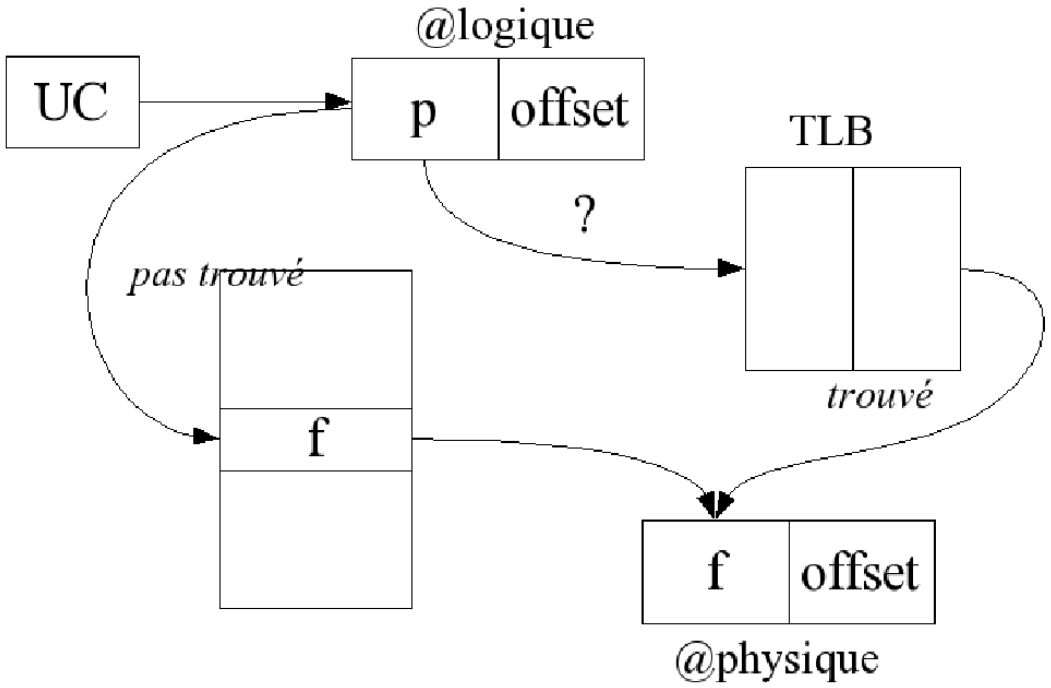
- Bit de validité: la page est-elle en RAM φ ?
- Contrôle effectué par la MMU

Implémentation de la table de pages

- Une table par processus, @ dans un registre de base (valeur dans le PCB)
- Pb: temps d'accès: pour accéder à l'@ i
  - recherche du numéro de page dans la table
  - accès à la page φ

2 accès mémoire$\longrightarrow$ perte de performances d'un facteur 2!
Solution standard: petite mémoire cache TLB (Translation Look-aside Buffer), table de hachage (clé, valeur). Prix important: peu d'entrées ($\leq$ 2048)

Fonctionnement du TLB



- Chaque page consultée est stockée dans le TLB (FIFO)
- Vidage du TLB à chaque context switch

Quelques définitions

Definition

Taux de présence (τ): fraction des numéros de page trouvés dans le TLB.

Ex: si TLB 5 fois plus rapide que RAM (2 ns contre 10 ns) et $\tau = 0,8$, temps effectif d'accès mémoire:

$$\begin{aligned} t_{eff} &= \tau\left(t_r + t_T\right) + \left(1-\tau\right)\left(2t_R + t_T\right) \\ &= 0,8\left(12\right) + 0,2\left(22\right) \\ &= 14 \end{aligned}$$

Soit 40% de ralentissement. Si $\tau = 0,98$, on obtient

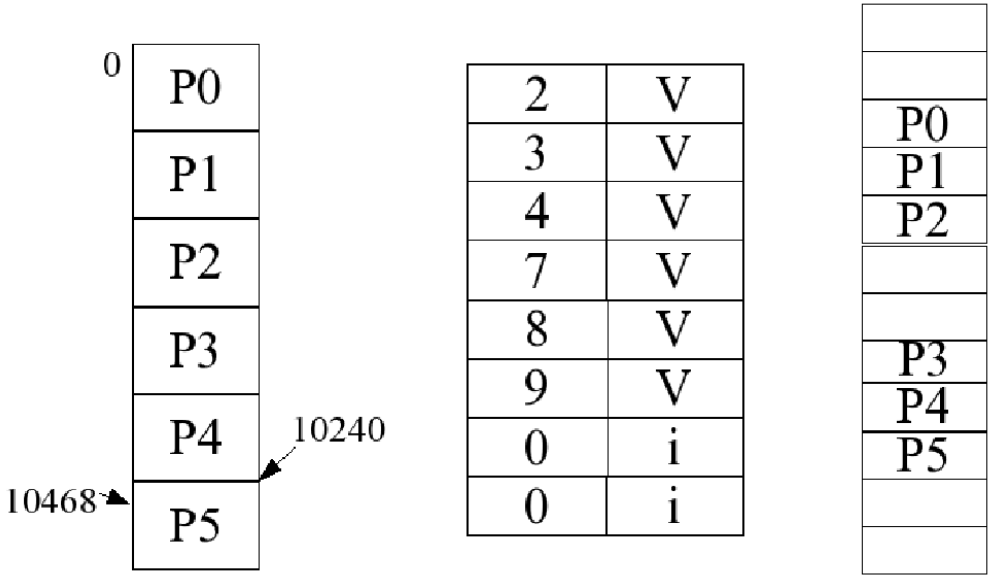
$$\begin{aligned} t_{eff} &= 0,98\left(12\right) + 0,02\left(22\right) \\ &= 12,2 \end{aligned}$$

Soit 22% de ralentissement

- Motorola 68030: TLB à 22 entrées

Bit de validité

- Autorise / interdit au processus l'accès à la page
- Indispensable car toutes les pages ne sont pas systématiquement en RAM
- Ex: @ logiques sur 14 bits (0 → 16383), taille de page 2 K0, 1 proc de taille 10468



Pagination multi-niveaux

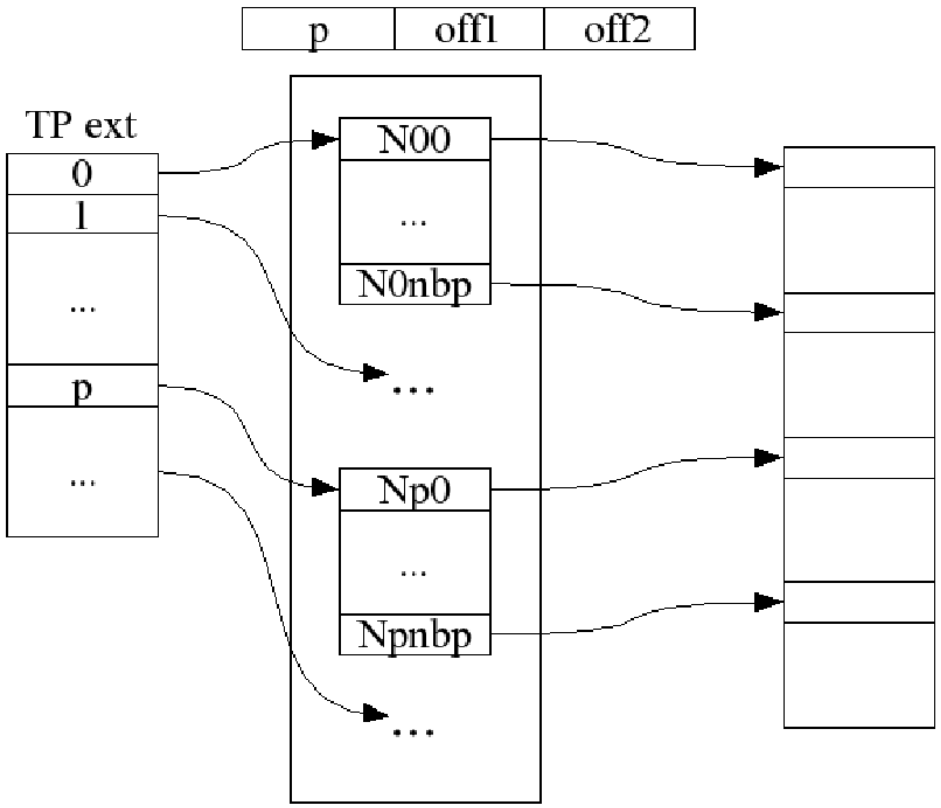
Grand espace d'adressage logique ($2^{32} \rightarrow 2^{64}$) $\rightarrow$ grande TP
Ex: 32 bits, pages de 4 KO(2^{12})$\rightarrow$ 2^{20}pages, soit 10^6entrées (plusieurs MO par processus!!)
Solution: partitionner la table en sous-tables

- Pagination à deux niveaux: la TP est elle-même paginée
- Pour un adressage sur 64 bits, ce n'est pas encore suffisant: on pagine en 3 ou 4 niveaux (pb: on peut avoir besoin de 4 accès mémoire avant d'obtenir l'@φ...)

Pagination à deux niveaux

@logique: | n° page (20) | offset (12) |

en fait | n°page dans TP (10) | depl (10) | offset(12) |



Performances

Considérons une pagination à 4 niveaux (soit 5 accès mémoire + 1 accès TLB au pire)
Si $\tau = 0,98$, on obtient $(0,98 \times 12) + (0,02 \times 52) = 12,8$, soit 28% de perte: ça vaut la peine!
Considérons l'exemple de NT (pagination à deux niveaux, pages de 4 KO: 1024 pages adressables par page de TP)
Si taille du processus=14 MO, il faut donc (au moins) 4 entrées dans la TPext, soit 5 pages à charger (la page externe et les 4 pages de second niveau), soit environ 20 KO contre 4 MO en pagination simple!!

Partage de pages

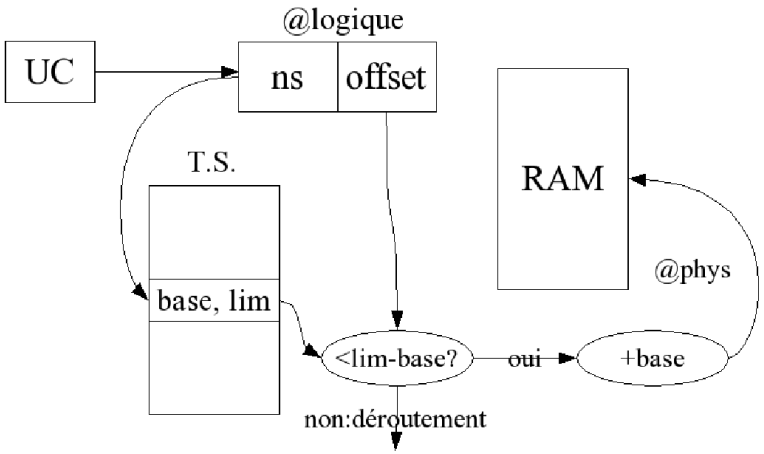
- Souhaitable quand des pages sont en lecture seule (code), ex. un éditeur avec 3 pages de code et une page de données (code *réentrant*):

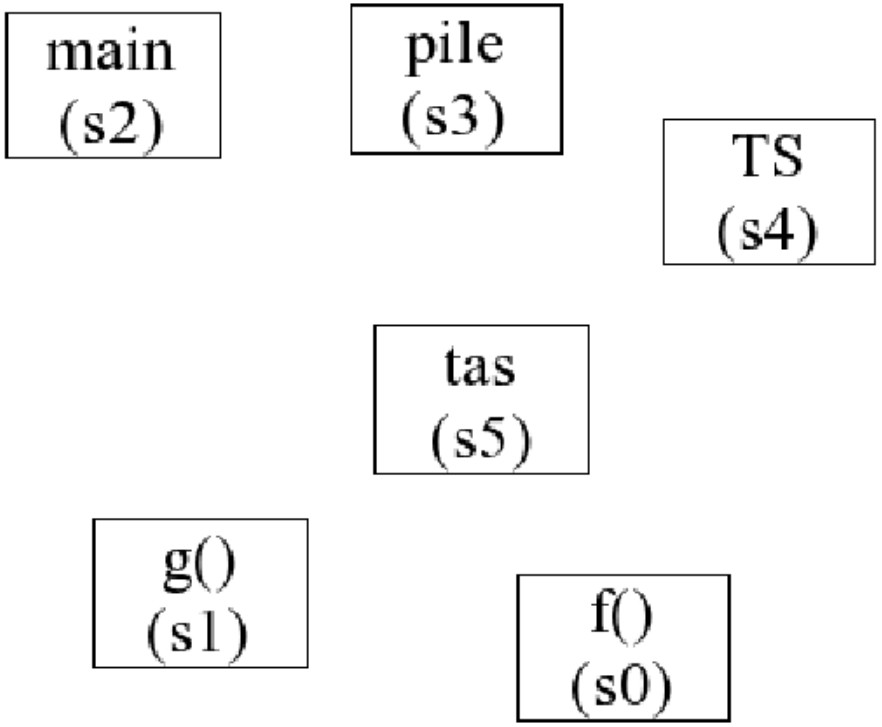
P_1	P_2	P_3
ed1	ed1	ed1
ed2	ed2	ed2
ed3	ed3	ed3
data1	data2	data3
0	0	0
7	7	7
4	4	4
3	9	1

0	ed1
	data3
	data1
	ed3
5	
	ed2
	data2
10	

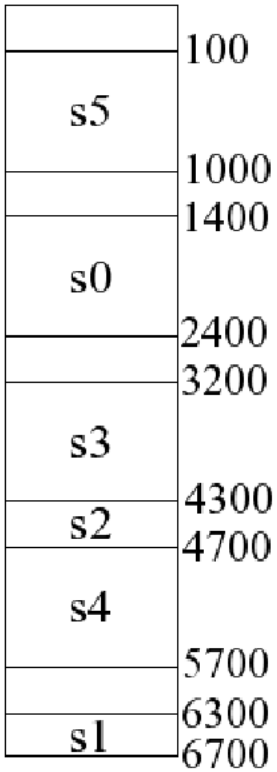
Segmentation

- L'espace d'adressage logique est divisé en blocs de taille variable: les segments
- On identifie chaque segment par un numéro
- Mapping: (segment/déplacement) $\leftrightarrow$ @φ fait par une table des segments TS.
- Chaque entrée de la table contient l'@ de base et l'@ limite du segment





TS	
lim	base
2400	1400
6700	6300
4700	4300
4300	3200
5700	4700
1000	100

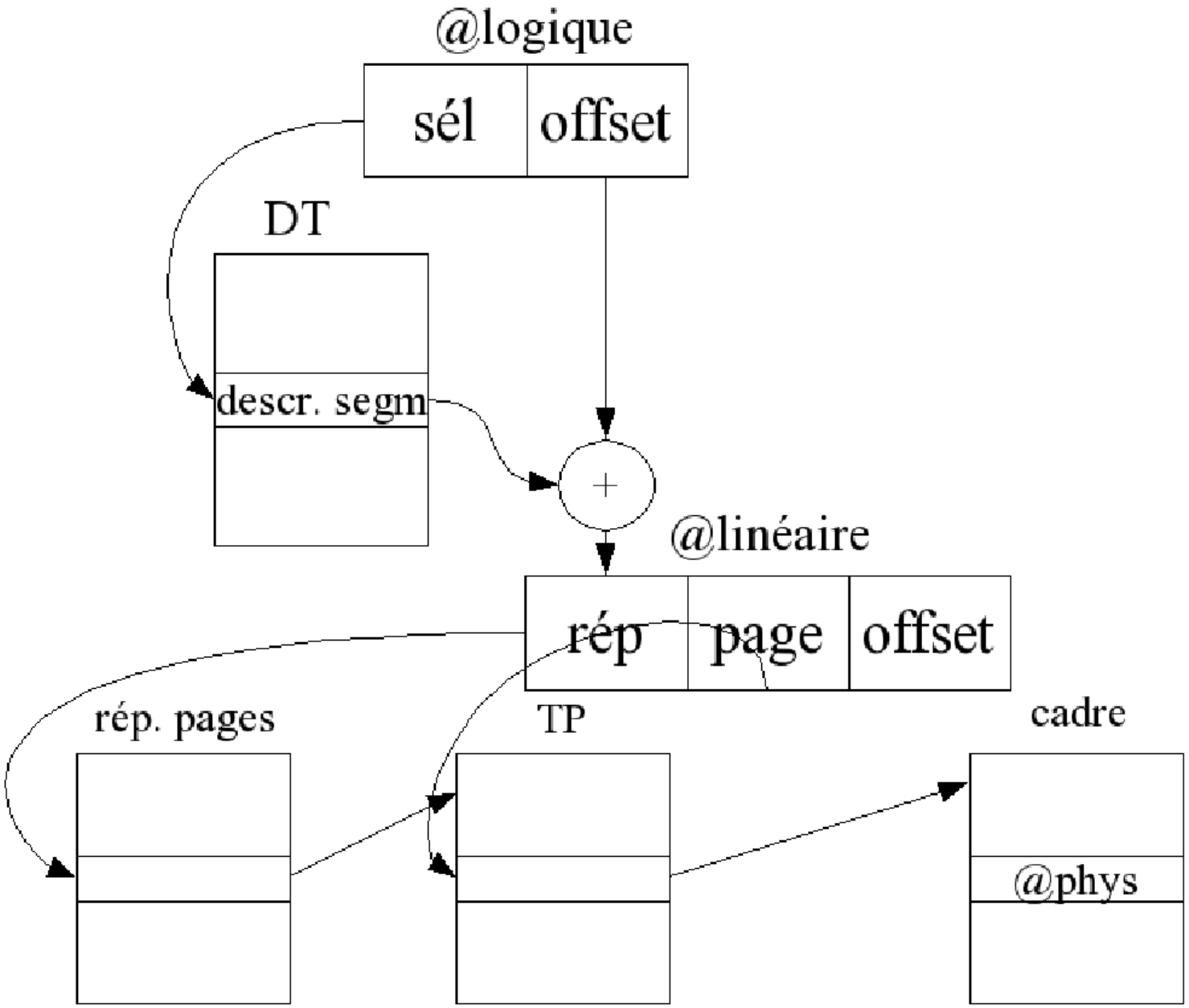


Conclusion sur la segmentation

- Comme pour la pagination, il est possible de partager des segments
- Certaines architectures combinent pagination et segmentation
- Ex: i386
  - 16Ksegments, chaque segment a pour taille max 4 GO; pages de 4 KO
  - l'espace logique est divisé en deux partitions: l'une liée au processus courant (8Ksegments max): LDT, l'autre commune (8Ksegments max): GDT
  - @logique=paire (sélecteur(16), déplacement(32))
  - sélecteur:

segment(13)	g/l(1)	prot(2)

  - @φ sur 32 bits, pagination à deux niveaux


@logique
sél
offset
DT
descr. segm
+
@linéaire
rép
page
offset
rép. pages
TP
cadre
@phys

Outline

1. Introduction
2. Gestion des processus
   - Introduction
   - Fonctions utiles sous UNIX
   - Ordonnancement
3. Communication entre processus
   - Communication par signaux
   - Communication par tubes
   - Synchronisation
   - Les sémaphores
   - Les sémaphores sous UNIX
   - Communication par mémoire partagée
   - Communication par files de messages
4. Gestion des entrées-sorties
   - Introduction
   - Méthodes d'accès
   - Méthodes d'allocation

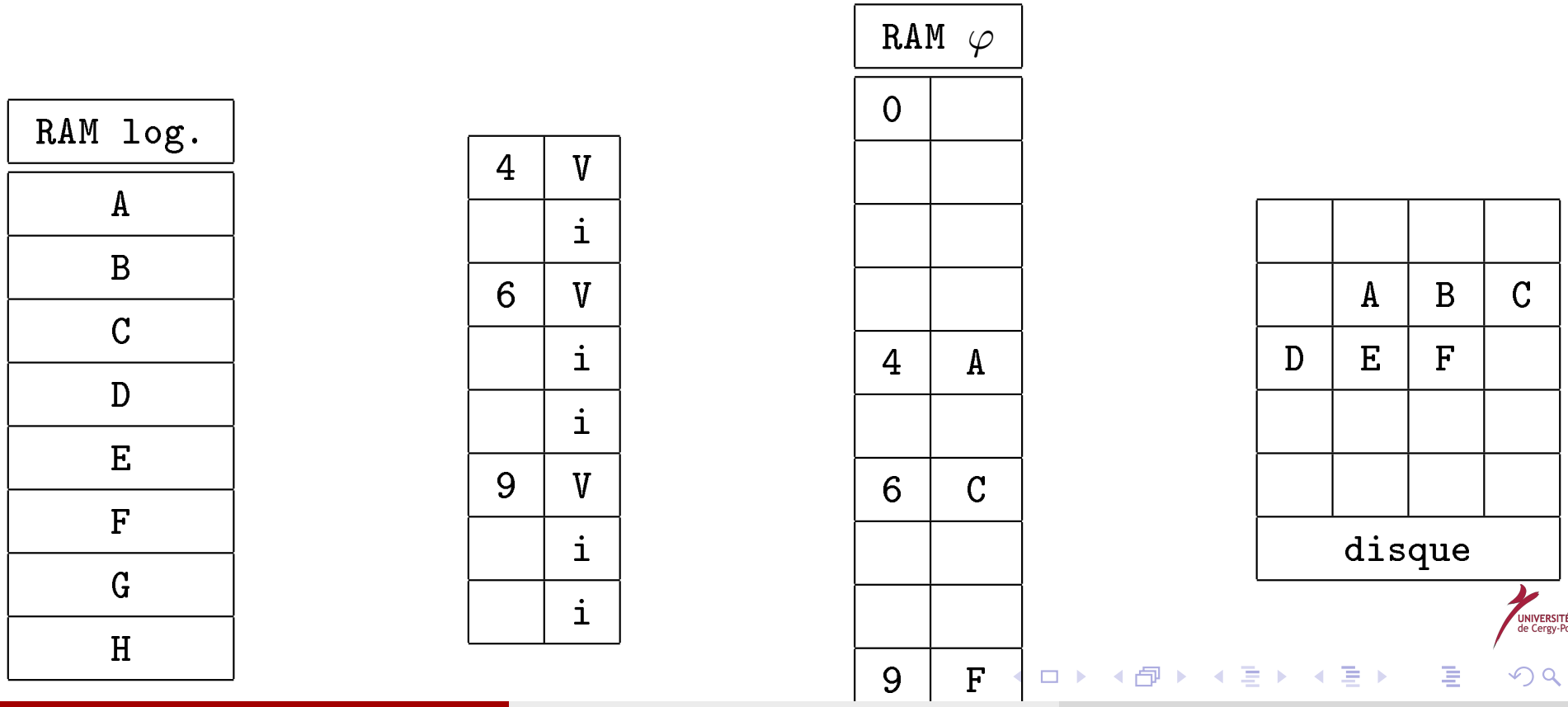
Mémoire virtuelle

Problème: dans les stratégies précédentes, le processus doit être intégralement en mémoire pour s'exécuter
But de la VM: autoriser l'exécution de processus dont une partie est en mémoire secondaire (en particulier, de procs de taille > RAM), en général la partie la moins utilisée (exceptions, zones peu accédées)
Avantages:

- les programmes prennent moins de place → exécution simultanée de plus de pgs
- moins d'E/S aux context switches
- plus besoin de recouvrements

Pagination à la demande

Les processus résident sur disque. Pour exécuter P_i, on ne charge en RAM que les pages nécessaires (lazy paging)
Nécessité d'un dispositif matériel pour indiquer les pages chargées et non-chargées. Par exemple, le bit de validité peut correspondre à "la page est légale et en RAM"



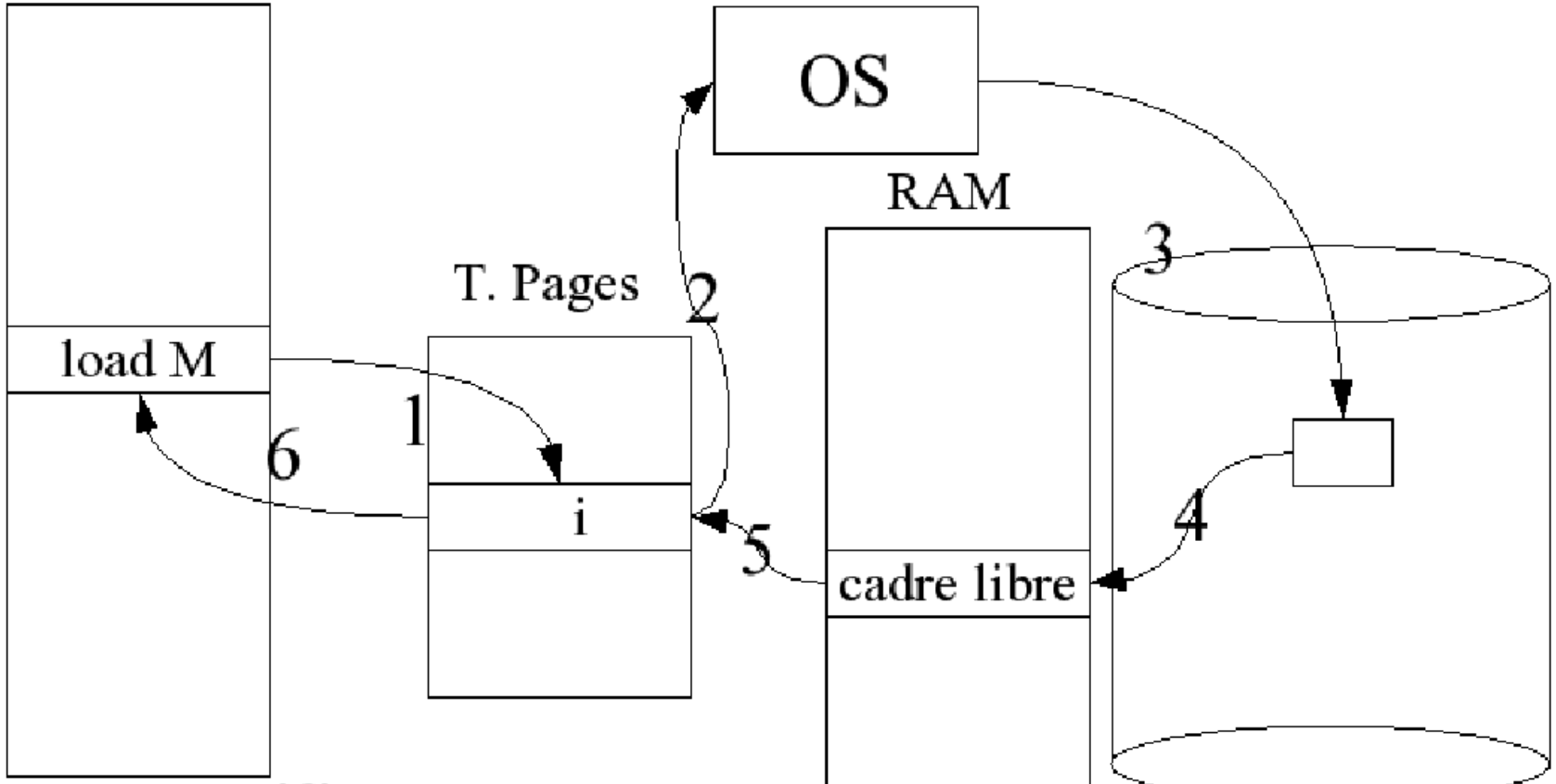
Défaut de page

1. examen de la table des processus (PCB) pour déterminer si l'accès est légal
2. si illégal, fin du processus (memory fault, segmentation fault)
3. recherche d'un cadre de page libre (maintien d'une liste des cadres libres)
4. E/S disque pour charger la page
5. MAJ table des pages et PCB (indiquer que la page est en RAM φ)
6. Redémarrage de l'instruction qui a provoqué le DP: l'interruption est *totalement* transparente

Redémarrage

- Si le DP se produit au chargement de l'instruction: recharger l'instruction
- S'il a lieu pendant l'extraction d'un opérande: recharger l'instruction et les opérandes
- ex: c = a + b

1. extraire et décoder l'instruction (ADD)
2. extraire a
3. extraire b
4. additionner a et b
5. stocker dans c. (si DP ici, refaire 1→ 5)



1- référence
2- déroutement
3- page sur disque
4- chargement page absente
5- MAJ T. Pages (i->V)
6- restart

Performances

- t_m: temps d'accès à la mémoire (de l'ordre de 10 ns)
- t_e: temps effectif en tenant compte du défaut de page
- p: probabilité de défaut de page
- t_d: duré d'un défaut de page

$t_e = (1-p)t_m + p \times t_d$
Grossièrement, 3 phases:

1. traitement de l'interruption due au DP (1 à 10 ns)
2. lecture page (latence 2ms, positionnement 5 ms, transfert 1 à 5 ms)
3. redémarrage processus (1 à 10 ns):
   $p = 10^{-3} \Rightarrow t_e = 0,999 \times 10^{-8} + 10^{-3} \times 10^{-2}$(10$\mu$s:1000 fois plus lent!)

Remplacement de pages

- Il faut impérativement limiter le nombre de DP en augmentant la probabilité pour que la page soit en RAM
- Idée: si tous les cadres sont occupés, remplacer l'un d'eux non utilisé actuellement:

1. trouver la page désirée sur disque
2. trouver un cadre libre. Si tous occupés, utiliser un algo de remplacement de page pour sélectionner le cadre "victime"
   1. écrire la victime sur disque
   2. MAJ les tables
3. charger la page souhaitée dans le cadre libéré. MAJ les tables
4. Redémarrer le processus

Pb: deux transferts de page: t_e est quasiment doublé!

On peut réduire cette surcharge avec un bit de modification dans la page, mis à 1 si écriture: le transfert cadre → disque ne se fait pas si bit à 0
Pbs (très sensibles puisque perfs RAM>>>perfs disque):

- développer un algo d'allocation de cadres
- développer un algo de RP efficace

Algorithmes de Remplacement de Pages (RP)

- On veut un taux de DP minimal: $\tau = \frac{N_{dp}}{N_{refs}}$
- On va évaluer l'algorithme sur une séquence de refs mémoire appelée *chaîne de références*, en mesurant N_{dp}
- 2 hypothèses:
  - localité temporelle: si une page a été récemment référencée, elle sera probablement bientôt à nouveau référencée
  - localité spatiale: si une adresse a été récemment référencée, ses voisines le seront probablement bientôt
- Exemple de trace d'un processus: 100, 432, 101, 612, 102, 103, 104, 101, 611, 102, 103, 104, 101, 610, 102, 103, 104, 101, 609, 102, 105, taille de page=100 octets: la chaîne obtenue est 1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

Nombre de DP

nb cadres	N_{dp}	commentaire
3+	3	1 à chaque première ref. (lazy)
1	11	systématique
2	?	ça dépend

- rempl. 1er:

1	4	1	6	1	6	1	6	1	6	1
1	1		6	1	6	1	6	1	6	1
-	4		4	4	4	4	4	4	4	4

: 10 DP

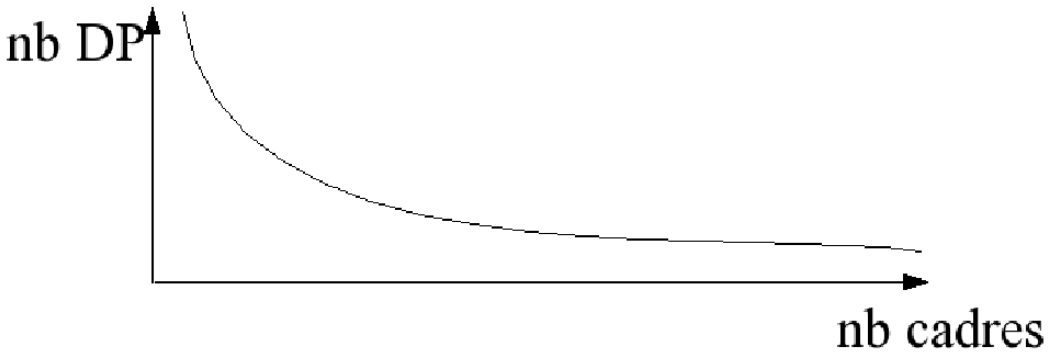
- rempl. dernier:

1	4	1	6	1	6	1	6	1	6	1
1	1		1							
-	4		6							

: 3 DP

Remarques

D'une façon générale, on a



Dans les comparaisons suivantes, on utilisera la chaîne
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1,
avec trois cadres

FIFO

- remplacement de la page la plus ancienne en RAM

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	0		1	2	3	0	4	2			3	0			1	2	7
	0	0	1		2	3	0	4	2	3			0	1			2	7	0
		1	2		3	0	4	2	3	0			1	2			7	0	1

Soit 15 DP

Pb: les performances ne sont pas toujours bonnes

Anomalie de Belady

Contre-exemple: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

N_c	1	2	3	4	5	6	7
N_{dp}	12	12	9	10	5	5	5

La courbe *remonte* entre 3 et 4!!

Cas optimal

- Consisterait à remplacer la page qui mettra le plus de temps à être réutilisée
- Impossible à réaliser: il faut connaître l'avenir
- Il sert juste de base de comparaison pour les algorithmes réels (cf. SJF)

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		2			2			2				7		
	0	0	0		0		4			0			0				0		
		1	1		3		3			3			1				1		

LRU

Least Recently Used. Idée: identifier le passé récent et l'avenir proche

Chaîne S: $LRU(S^{-1}) = LRU(S)$ et $OPT(S^{-1}) = OPT(S)$

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		4	4	4	0			1		1		1		
	0	0	0		0		0	0	3	3			3		0		0		
		1	1		3		3	2	2	2			2		2		7		

12 DP

ni OPT, ni LRU ne souffrent de l'anomalie de Belady

Seconde chance

Approximation logicielle de LRU (qui nécessite un support matériel coûteux): ne nécessite qu'un bit de référence sur chaque page, qui passe de 0 à 1 quand la page est référencée
Principe: presque FIFO. Si la page a le b.r. à 0 elle est remplacée, sinon on le remet à 0 et on passe à la suivante
Algo:

- page référencée: R=1 et placement en queue de file
- DP: on regarde la page en tête de file
  - si R==0, on la retire
  - si R==1, on la place en queue de file avec R=0

Exemple

2	3	2	1	5	2	4	5	3	2	5	2

2	2		2	5	5	5		3		3	
	3		3	3	2	2		2		2	
			1	1	1	4		4		5	

(italique: bit à 0)

2	3	1	→	3	1	*2*	(2° chance pour page 2) →	1	*2*	*3*	→
2	*3*	*1*	→	*3*	*1*	5	(p5) →	*1*	5	2	(p2) →
5	2	4	(p4) →	2	4	*5*	→	4	*5*	*2*	→
5	*2*	*4*	→	*2*	*4*	3	(p3) →	2	*4*	3	(p2) →
4	3	*2*	→	3	*2*	5	(p5) →	3	2	5	(p2)

8 DP: $\tau = \frac{8}{12} = 0,66$

Exercice

Refaire l'algo de seconde chance sur la chaîne de refs exemple:

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	*0*	0	2	2	*0*	*3*	3	*4*	*2*	2	*0*	*3*	1		*2*	*2*	0
	0	0	*1*	1	*0*	0	*3*	4	4	*2*	0	0	*3*	1	2		*0*	0	7
		1	2	2	3	3	4	2	2	0	3	3	1	2	0		7	7	1

14 DP

Allocation de pages physiques

Combien de pages allouer à chaque processus?

- statique: nb pages fixé à la création du processus
- dynamique: nb pages variable
- local: la page choisie est dans l'espace du processus fautif
- global: elle est prise dans l'ensemble des pages chargées

Allocation statique

- Equitable: même nb de cadres à chaque processus. ex, 16 cadres en RAM avec P1 (4 pages), P2 (12 pages) et P3 (3 pages), on alloue 5 (16/3) cadre à chaque processus: gaspillage de 1 cadre.
  Le remplacement ne peut être que local (sinon cela modifie le nb de cadres: dynamique!)
- Proportionnel au nb pages logiques: P1 obtient (4*16)/19=3 cadres, P2 en obtient 10 et P3 2. NB: on en gaspille encore 1
- Proportionnel à la priorité du processus

Allocation dynamique

- L'OS alloue davantage de cadres si le processus provoque beaucoup de DP
- Si le processus a reçu trop de cadres, l'OS en retire
- Surveillance permanente: surcoût

Interblocages

Situations d'interblocage:

- Carrefour avec une voiture dans chaque rue
- Les N philosophes avec chacun une baguette
- ...

Liées à la nécessité pour les processus d'utiliser des ressources

- matérielles (imprimantes, CD, disques, CPU, RAM, etc.)
- logicielles (fichiers, processus, sémaphores, messages, etc.)

Catégories de ressources

1. *partageables*: accessibles simultanément par plusieurs processus. Ex: fichiers en mode "r"
2. *consommables*: utilisables un nombre limité (1) de fois. Ex: messages, interruptions, signaux...
3. *réutilisables*: en exclusion mutuelle mais utilisables par $P_i$dès que libérées par P_j. Ex: CPU, RAM, sémaphores, ...
4. *préemptibles*: peuvent être retirées sans risque au processus qui les utilisaient. Ex: CPU, RAM. C-ex: imprimante

Seules les ressources non partageables et non préemptibles peuvent être cause d'un interblocage.

Séquence typique d'utilisation

1. Demande, attente éventuelle
2. Obtention, utilisation
3. Libération

Ex: fichiers (open/close), RAM (malloc/free), device (request/release)

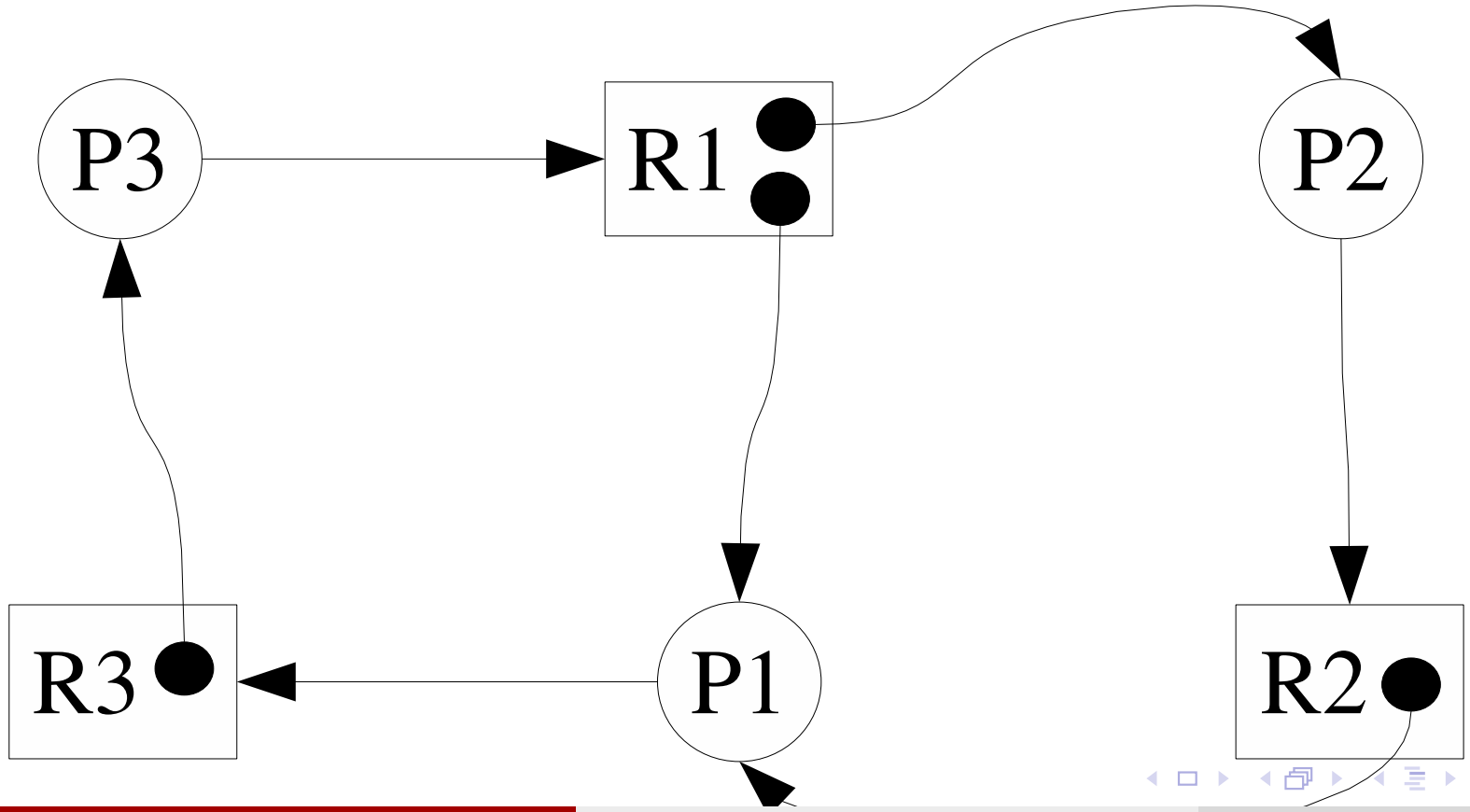
Définition

Interblocage: un ensemble (P_i) de processus est en interblocage si $\forall i, P_i$ attend un événement qui ne peut être généré que par $P_j, j \neq i$
Conditions nécessaires:

1. Ressources en exclusion mutuelle
2. Détention/attente: un des processus détient une ressource et en attend une autre
3. Préemption impossible: l'OS ne peut retirer une ressource à un processus, seul le processus peut la relâcher
4. L'attente est circulaire (cycle de longueur quelconque)

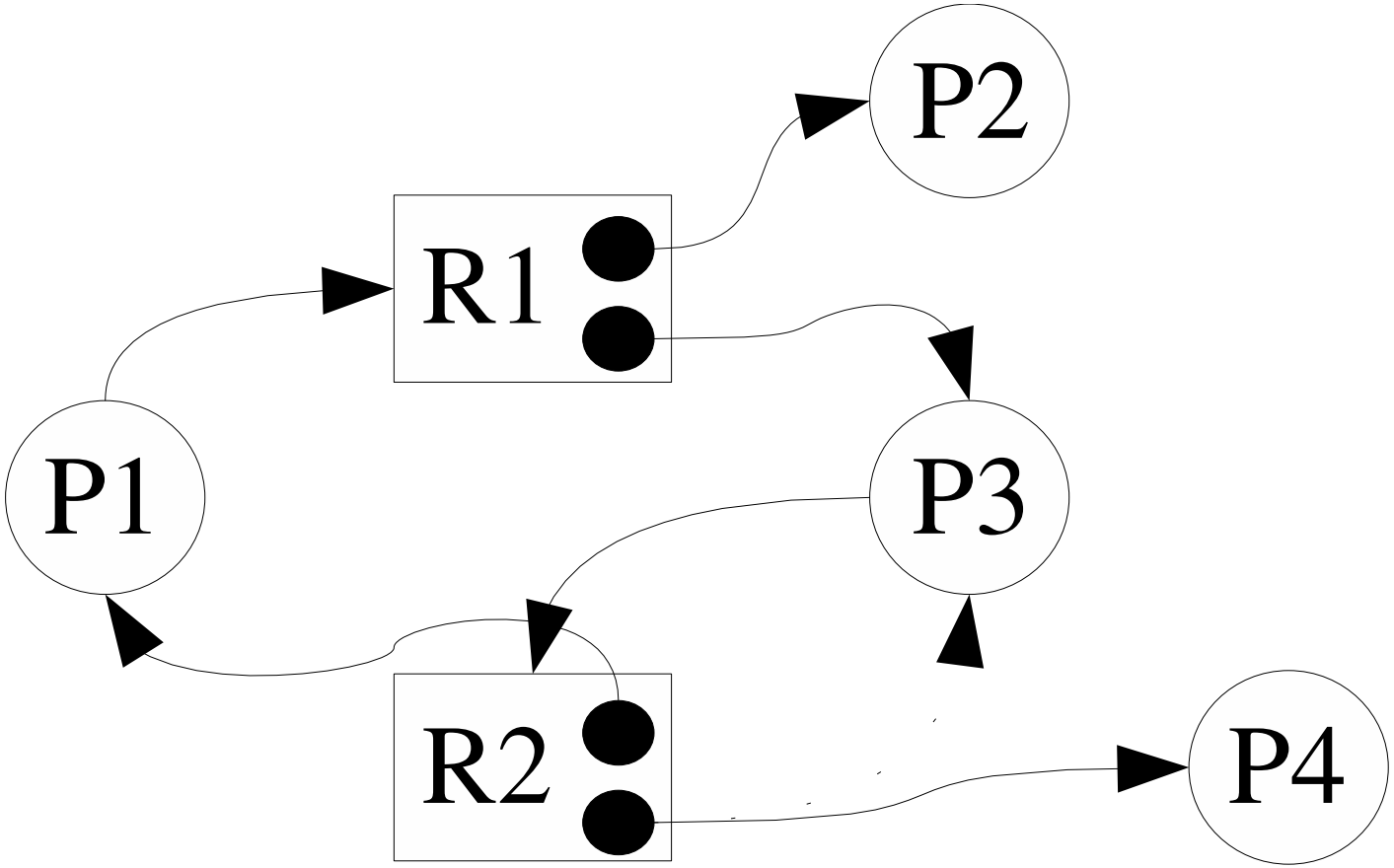
Modélisation de l'allocation de ressources

- Graphe à deux types de nœuds, les processus et les (classes de) ressources
- Ressources pointées (1 point par instance)
- Arcs proc/ressource (attente) et ressource/proc (détention)



Cycles

Condition nécessaire à l'interblocage. Elle est suffisante si chaque classe de ressource n'a qu'une instance.



- Ici, cycle P1/R1/P3/R2/P1 mais pas interblocage (P4 peut libérer R2 qui est allouée à P3)

Traitement des interblocages

1. Politique de l'autruche (pas de précaution): reboot
2. Détection/correction: a posteriori; par exemple suppression d'un processus
3. Evitement (dynamique): à chaque demande d'allocation, vérification de faisabilité avant accord
4. Prévention: l'OS est conçu pour qu'une au moins des conditions d'IB manque tjs (par exemple il peut préempter une ressource après un timeout).

Technique d'évitement I

Il faut disposer d'informations

- sur l'état du système (allocations, nb classes de ressources,...)
- sur le nombre d'instances de chaque classe utilisées par chaque processus

SI on a N processus et M classes de ressources, on définit deux vecteurs

1. $ress = [r_1, ..., r_M]$ le nombre d'instances de $R_i$dans l'OS
2. $disp = [d_1, ..., d_M]$ le nombre d'instances disponibles de R_i

On définit également 2 matrices $N \times M$:

1. annoncées: nb max d'instances nécessaires aux processus
2. allouées: nb max d'instances actuellement allouées aux processus

Conditions:

- toute ressource est soit libre, soit allouée;
- tout processus demande au plus le nombre de ressources détenues dans l'OS (*ress*)
- On n'alloue pas plus que ce qu'un processus a annoncé

Notion d'état "sûr" I

Definition

Un état est dit sûr s'il existe une séquence d'exécution des processus qui leur permet de se terminer tous.

Exemple: annoncées= $\begin{bmatrix} 9 \\ 4 \\ 7 \end{bmatrix}$, allouées= $\begin{bmatrix} 3 \\ 2 \\ 2 \end{bmatrix}$ (on a donc 3 proessus)

10 ressources en tout, donc $10 - (3 + 2 + 2) = 3$ disponibles

exécution de P2: annoncées= $\begin{bmatrix} 9 \\ - \\ 7 \end{bmatrix}$, allouées= $\begin{bmatrix} 3 \\ 0 \\ 2 \end{bmatrix}$, disp=5

exécution de P3: annoncées= $\begin{bmatrix} 9 \\ - \\ - \end{bmatrix}$, allouées= $\begin{bmatrix} 3 \\ 0 \\ 0 \end{bmatrix}$, disp=7

exécution de P1: OK (6 à allouer, 7 dispo): l'état est sûr

Contre-exemple: si le vecteur allouées devient allouées= $\begin{bmatrix} 4 \\ 2 \\ 2 \end{bmatrix}$ (donc disp=2)

exécution de P2: annoncées= $\begin{bmatrix} 9 \\ - \\ 7 \end{bmatrix}$, allouées= $\begin{bmatrix} 4 \\ 0 \\ 2 \end{bmatrix}$, disp=4

- P3 impossible ($7 > 2 + 4$)
- P1 impossible ($9 > 4 + 4$): état non sûr

Algorithme du banquier I

- Le banquier choisit l'ordre des prêts afin de ne pas manquer d'argent
- Les processus déclarent le nombre max. d'instances utilisées
- Lors d'une demande d'allocation, le banquier simule. Si sûr, attribution

```
typedef int Vecteur[M];
typedef struct {
  Vecteur ress, disp, annoncees[N], allouees[N];
} Etat;
Etat E, Enew;
Vecteur demandees[N];
```

```
(pour tout i)
si demandees[i]> E.annoncees[i]-E.allouees[i]
  ...erreur...
si demandees[i]> E.disp
  ... mettre P_i en attente ...
sinon
  Enew=E
  Enew.allouees[i] += demandees[i]
  Enew.disp -= demandees[i]
  si sur(Enew) alors E = Enew
    sinon
      ... mettre P_i en attente ...
```

```
int sur(Etat e)
  Vecteur dispCourant; bool possible, termine[N];
  int i;
  dispCourant = E.disp;
  for(i=0; i<N; i++) termine[i] = false;
  possible = true;
  while possible
    chercher i / !termine[i] && e.annoncees[i]-e.allouees[i]
      < dispCourant
    si i≥0
      dispCourant -= e.allouees[i]
      termine[i] = true
    sinon possible = false
  return (∀i termine[i])
```

Exemple

4 ressources, 5 processus

$$\text{allouées} = \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \text{ annoncées} = \begin{bmatrix} 4 & 1 & 1 & 1 \\ 0 & 2 & 1 & 2 \\ 4 & 2 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 2 & 1 & 1 & 0 \end{bmatrix},$$

$\text{disp} = \begin{bmatrix} 1 & 0 & 2 & 0 \end{bmatrix}$

P5 demande $\begin{bmatrix} 1 & 0 & 1 & 0 \end{bmatrix}$, est-ce sûr?

- besoins = annoncees - allouees (- demandees[5]) = $\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$
- demandees[5]>? besoins[5] non: OK (indices=1..5 pour simplifier)
- demandees[5]>? disp non: OK
- on simule (besoins, allouées, disp): disp=$\begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}$

1. exéc. de P4: $\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 2 \\ 3 & 1 & 0 & 0 \\ - & - & - & - \\ 1 & 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$, disp=$\begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}$
2. exéc. de P1: $\begin{bmatrix} - & - & - & - \\ 0 & 1 & 1 & 2 \\ 3 & 1 & 0 & 0 \\ - & - & - & - \\ 1 & 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$, disp=$\begin{bmatrix} 4 & 1 & 2 & 2 \end{bmatrix}$
3. exéc. de P2: $\begin{bmatrix} - & - & - & - \\ - & - & - & - \\ 3 & 1 & 0 & 0 \\ - & - & - & - \\ 1 & 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$, disp=$\begin{bmatrix} 4 & 2 & 2 & 2 \end{bmatrix}$
4. exéc. de P3: $\begin{bmatrix} - & - & - & - \\ - & - & - & - \\ - & - & - & - \\ - & - & - & - \\ 1 & 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$,

   disp=$\begin{bmatrix} 5 & 3 & 3 & 2 \end{bmatrix}$(OK)

Détection/correction

- Si une seule instance par ressource: détecter les cycles dans le graphe
- sinon, utiliser `sur()` en remplaçant
  `e.annoncees[i]-e.allouees[i]`
  par
  `demandees[i]`

Ex: allouées= $\begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 3 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{bmatrix}$, demandées= $\begin{bmatrix} 0 & 0 & 0 \\ 2 & 0 & 2 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 2 \end{bmatrix}$, disp= $\begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$

pas d'IB: P1/P3/P4/P2/P5 permet de tout terminer

Si on remplace demandees[3,3] par 1 (au lieu de 0):

exéc. de P1: allouées= $\begin{bmatrix} 0 & 0 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 3 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{bmatrix}$, demandées= $\begin{bmatrix} - & - & - \\ 2 & 0 & 2 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 2 \end{bmatrix}$,

disp= $\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$

On ne peut pas exécuter P3: Interblocage