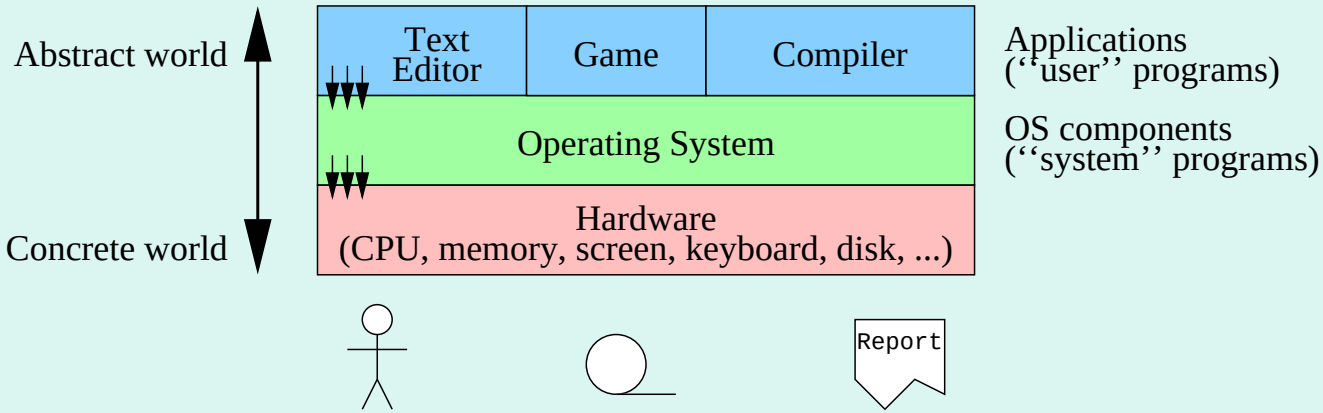
IFT2245
Systèmes d'exploitation

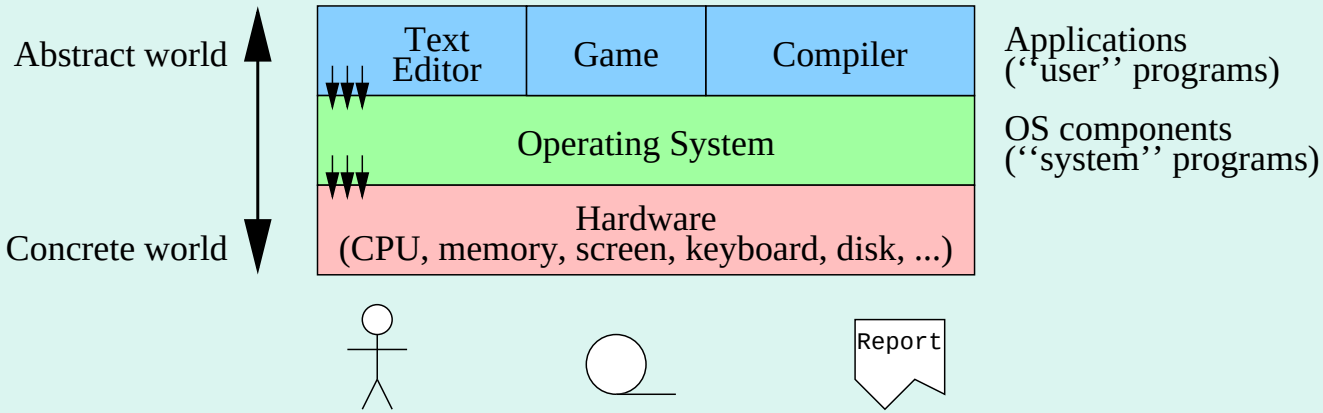
http://www.iro.umontreal.ca/~feeley/cours/ift2245/

Rôle des SE



- SE = ensemble des composantes logicielles qui font le pont entre les **applications** et le **matériel**
- Offre des **services abstraits** aux applications (p.e. ouvrir le fichier "abc") et demande des **services concrets** du matériel (p.e. lire le secteur N du disque)
- Responsable de l’exécution des applications et la gestion des ressources (physiques et logiques)
- Classification floue (compilateur = application ou composante du SE? ça dépend du point de vue)

Attraits des SE



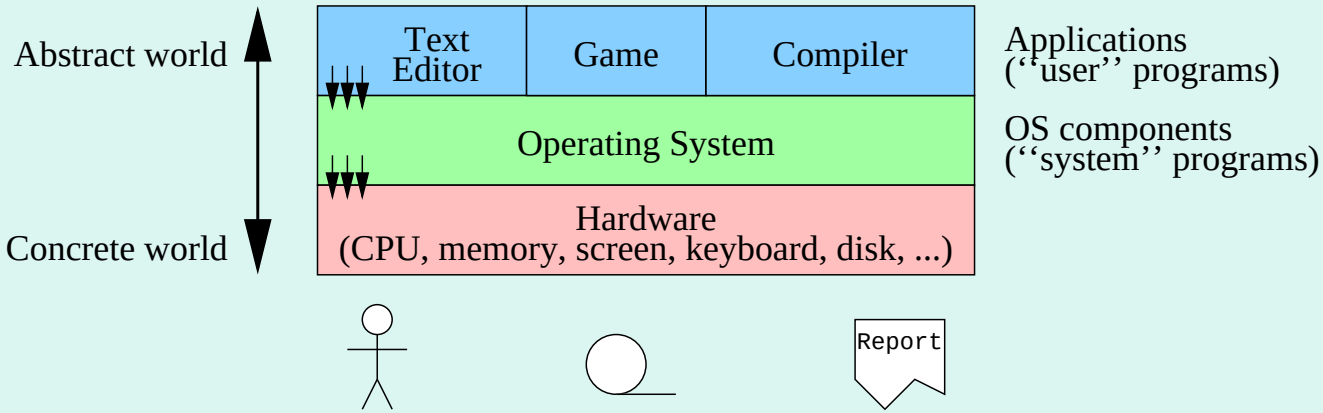
1) **Facilite la réalisation** des applications car il fournit plusieurs services prédéfinis:

   ⇒ Gestion des fichiers, processus, réseau, etc

2) Applications plus portables car il offre une abstraction du matériel (architecture matérielle PC/Mac/..., composantes matérielles, etc)

   ⇒ l’API du SE définit une **machine virtuelle**

Attraits des SE



3) Uniformise l’interface usager des applications (qu’il soit graphique, textuel, ou autre) ce qui contribue à la **convivialité**

4) Centralise les politiques de gestion des ressources ce qui permet d’en améliorer l’**efficacité** et de **protéger** l’accès aux ressources privées ou critiques

Objectifs des SE

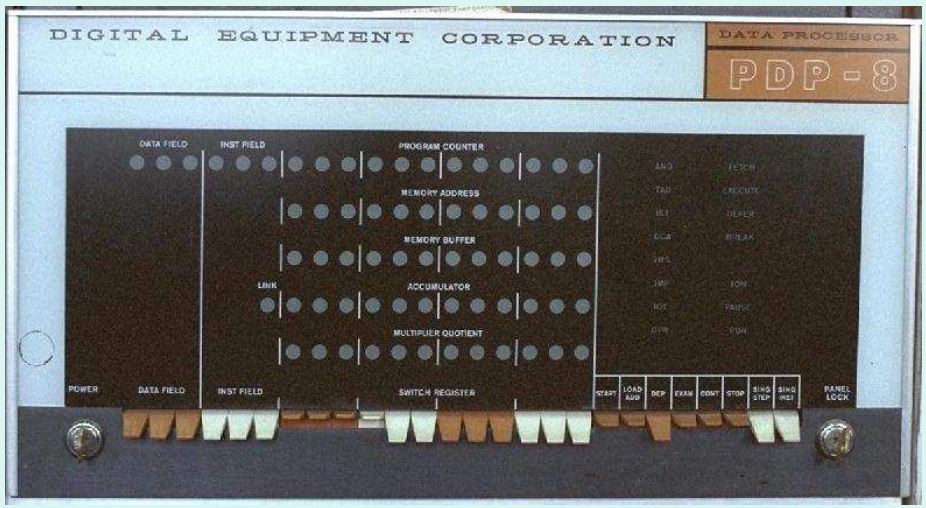
- Rendre l'ordinateur **plus convivial** (plus facile à utiliser)
- Accroître l'**efficacité** de l'ordinateur (meilleure exploitation des ressources)
- Ce sont des **objectifs contradictoires** qui mènent à des compromis dans le SE (p.e.: interfaces usagers graphiques, multi-usagers)

Composantes Fondamentales d'un SE

- Le logiciel central du SE est le noyau ("**kernel**"); celui-ci est normalement toujours chargé en mémoire
- Le "**bootstrap loader**" s'occupe de charger le kernel en mémoire lorsque l'ordinateur est démarré (à la mise sous tension ou "reset")
- Pour chaque type de périphérique (disque dur, carte réseau, carte graphique, carte de son, etc) il y a une composante qui est responsable de son contrôle; les **pilotes** ("**device drivers**")
- Pour chaque type de ressource (CPU, mémoire, imprimantes, lien réseau, etc) il y a une composante qui est responsable de sa gestion; les **allocateurs de ressources**

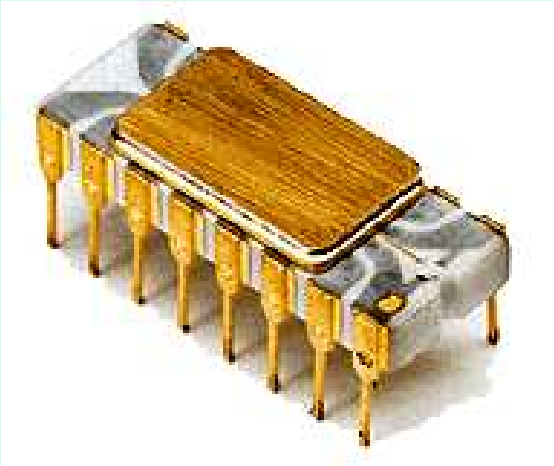
Les “Mainframes” et Minis

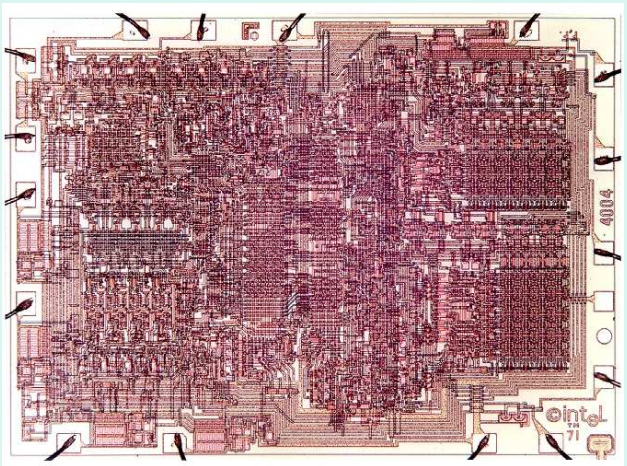
- 1954 – IBM 704 = premier ordinateur à succès commercial, tubes, 36/15 bits (donnée/adresse), première mémoire RAM, 40K additions par seconde, “GM/NAA-I/O” = premier SE “batch” (écrit par les usagers)
- 1965 – développement du SE “MULTICS” (Multiplexed Information and Computing System) pour le mainframe GE-645: multi-usagers, mémoire virtuelle, système de fichier, multiprocesseur, écrit en PL/I; DEC PDP-8 = premier mini-ordinateur, 12/12 bits (donnée/adresse), 4K mots RAM, transistors, $19K



- 1969 – développement du SE “UNICS” à Bell Labs par Ken Thomson et Dennis Ritchie (pour jouer à “Space Travel” sur un PDP-7!), écrit en assembleur; inspiré de Multics; porté au PDP-11 en 1970; conception de C en 1971; kernel récrit en C en 1972; System V UNIX sort en 1973

Les Microprocesseurs





- 1971 – Intel 4004 = premier microprocesseur, inspiré du PDP-8, conçu pour Busicom pour le marché des calculatrices, remplace une douzaine de circuits logiques, 4/12 bits (donnée/adresse), 2300 trans., ˜100KHz, $100, base du premier micro-ordinateur (“Intellec-4”)
- 1972 – Intel 8008 = extension 8 bit du 4004, 3300 trans., ˜200KHz
- 1973 – Gary Kildahll de Digital Research écrit en PL/M un SE pour 8008 qui se nomme CP/M (Control Program/Monitor)
- 1974 – Intel 8080: 8/16 bits (donnée/adresse), 6000 trans., ˜2MHz, $360; la série Intel (8086, 80386, 80486, Pentium) reste compatible avec le 8080; Xerox fabrique l’Alto (premier ordinateur avec écran bitmap, souris, Ethernet, Smalltalk)

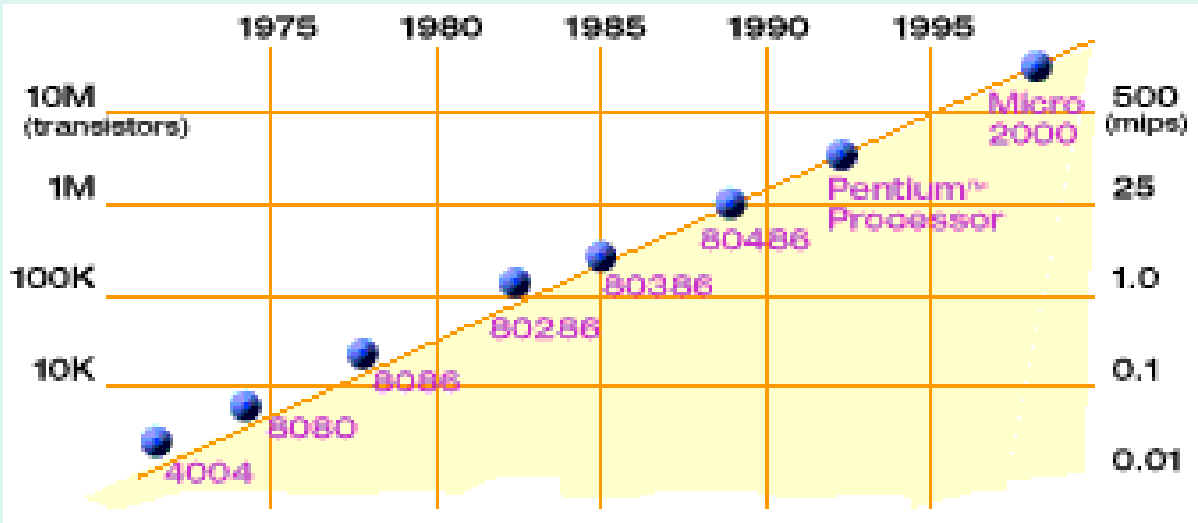
Les Micro-Ordinateurs

- 1975 – MITS (Micro Instrumentation Telemetry Systems) annonce le kit “Altair 8800” basé sur le 8080, 256 byte RAM, bus S-100, $395; Bill Gates écrit le premier interprète BASIC pour MITS; la compagnie “Traf-O-Data” qu’il a fondée avec Paul Allen est renommée “Micro-Soft”

- 1976-1979 – Gates est renvoyé de Harvard; kit Apple I; TRS-80 vendu par Radio Shack; Apple II; le tableur “VisiCalc” sur Apple II devient la première “killer application”
- 1980 – IBM cherchant un OS pour l’IBM-PC contacte Bill Gates qui leur suggère d’acheter CP/M, IBM ne le fait pas et Gates achète plutôt QDOS (Quick and Dirty Operating System, à peu près 4000 lignes d’assembleur, écrit en 6 semaines, inspiré de CP/M) de Seattle Computer Products pour $50K pour l’étendre et en vendre une license non-exclusive à IBM qui le revend sous le nom de DOS; MS/DOS est la base des autres SE Microsoft jusqu’à Windows-98
- 1981 – Lancement de l’IBM-PC 5150, basé sur le 8088, 4.77MHz, 16Kbyte RAM, clavier, écran, un floppy, $3000; l’architecture est copiée par les fabriquants de clones, Microsoft recoit une redevance pour chaque machine vendue avec MS/DOS

Loi de Moore

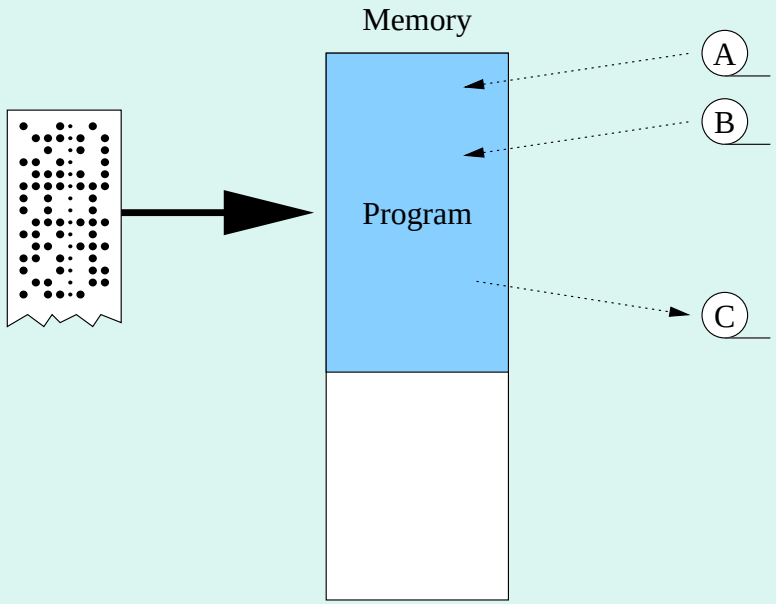
- En 1965 Gordon Moore prédit que la densité des puces va doubler à chaque 12 mois
- En 1975 cette loi est révisée: la “puissance” double à chaque 18 mois



- Grâce à cet accroissement, de plus en plus de fonctionalités sont offertes par les SE (interfaces usagers graphiques, multimédia, composantes objet, etc)

Au Tout Début

- Les premiers ordinateurs n'avaient **pas de SE résidant**
- Le programme était **chargé en mémoire** (d'un ruban perforé, cartes, ou autre) à un emplacement préétabli et puis son exécution lancée à son **point d'entrée**



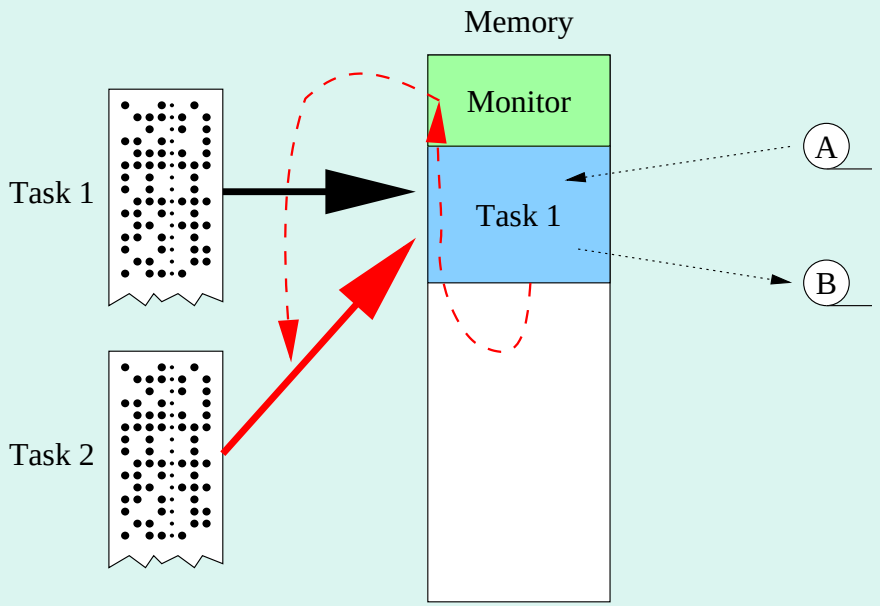
- Des **bibliothèques** de routines (principalement pour diverses opérations d'E/S) ont vu le jour pour éviter à les écrire à nouveau pour chaque nouveau programme

Traitement en Lot

- Plus tard l'utilisation d'ordinateurs pour traiter des lots de tâches ("**batch**" processing) devint la norme
- Déf: une **tâche** est un ensemble de calculs à effectuer; une **sous-tâche** c'est une partie d'une tâche; la **taille** d'une tâche c'est la quantité d'opérations à effectuer
- Typiquement, chaque tâche correspondait à la lecture de données d'un ou plusieurs rubans magnétique et l'écriture de données sur ruban après un traitement relativement simple
- Pour **minimiser le temps d'inactivité de l'ordinateur** lorsque l'opérateur humain montait les rubans requis par le programme, il était profitable de grouper dans un même lot des tâches qui utilisaient les mêmes rubans
- De nos jours **les E/S sont toujours relativement lentes** et demandent une considération spéciale

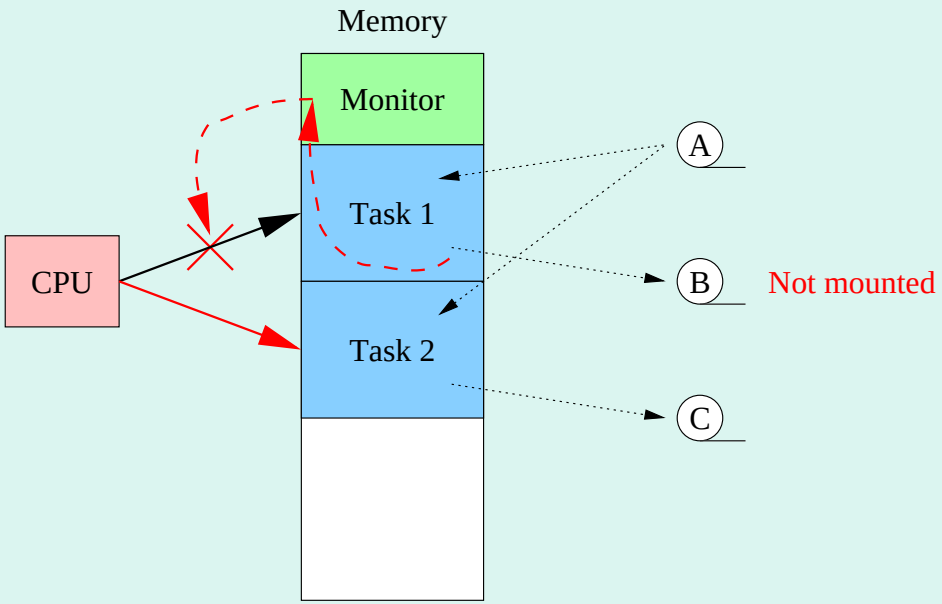
Moniteur

- Les premiers SE (“**moniteur**” résidant) automatisaient cette procédure en enchaînant simplement l’exécution de chaque tâche
  - le moniteur charge la première tâche puis y transfère l’exécution
  - à la fin de cette tâche le contrôle est redonné au moniteur
  - le moniteur charge la deuxième tâche puis y transfère l’exécution, etc



Multiprogrammation

- Le prochain pas fut la **multiprogrammation**: plusieurs tâches sont chargées en mémoire; le CPU s'occupe de seulement une tâche à la fois; lors d'une opération d'E/S qui ne peut être complétée immédiatement le moniteur fait passer le CPU à la prochaine tâche (**multiplexage du CPU**):



- Ceci permet au CPU de **faire des calculs utiles** plutôt que d'être inactif pendant les opérations d'E/S

Multi-tâche (“Multitasking”)

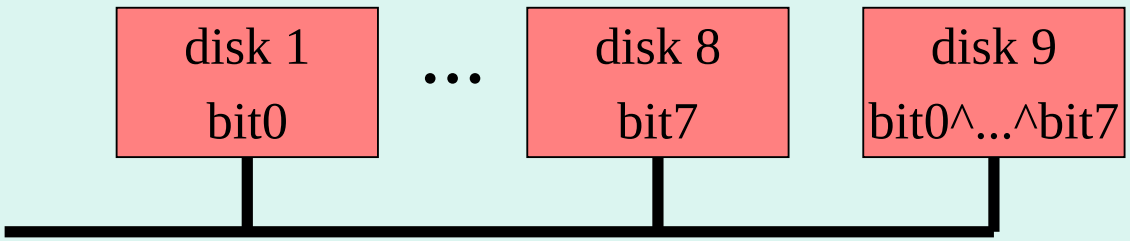
- Le **multi-tâche** (“multitasking” ou “time sharing”) est une extension de la **multiprogrammation** pour permettre à **plusieurs usagers** d’utiliser simultanément un même ordinateur (au moyen d’un terminal)
- Le CPU passe rapidement d’une tâche à la suivante à intervalles réguliers (“**quantum**”)
- Chaque usager reçoit une fraction de la puissance de l’ordinateur et a l’illusion d’être le seul usager
- Le SE peut permettre un nombre de tâche qui **dépasse la quantité de mémoire physique**
- L’idée de la **mémoire virtuelle** c’est que seules les données qui sont présentement traitées ont besoin d’être en mémoire physique (pour un accès rapide)
- Le SE présente à chaque tâche l’illusion d’une mémoire qui lui est propre (**mémoire logique**)

SE Temps-Réel

- Dans un **SE temps-réel** le SE place des garanties sur le temps de réponse (le temps entre l'occurrence d'un événement et la réaction par le système)
  - **"Hard real-time"**: temps de réponse maximal garanti
  - **"Soft real-time"**: le temps de réponse est court statistiquement
- Domaines: contrôle (robotique, freins ABS, machinerie), jeux vidéo, multimédia, télécommunication, serveurs
- Pour atteindre des contraintes de temps serrées, il faut normalement abandonner les aspects plus avancés (système de fichier, mémoire virtuelle, multi-tâche)

SE Tolérant aux Pannes

- Dans un **SE tolérant aux pannes** le SE peut s'adapter à des pannes pour maintenir le service offert par les applications (en utilisant la redondance)
- Exemple: système de disque **RAID**



bit0^bit1^bit2^bit3^bit4^bit5^bit6^bit7^bit8=0

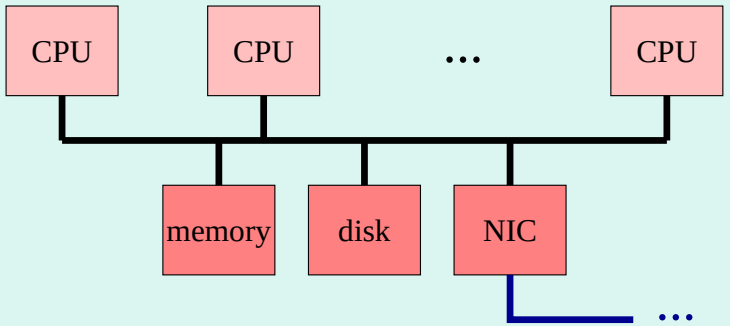
- Domaines: télécommunication (systèmes "five-nine" = en fonction 99.999% du temps = hors fonction 5 minutes/année), services essentiels, bases de données

SE Parallèle (1)

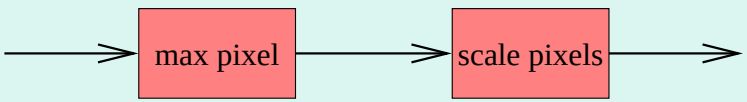
- Les **SE parallèles** gèrent les ressources partagées d'un multiprocesseur (CPUs, mémoire, périphériques)
- Avantages des multiprocesseurs:
  - **débit plus élevé**: en théorie il est possible d'accélérer le traitement jusqu'à un facteur de N avec N processeurs (cela dépend du **degré de parallélisme** de la tâche)
  - **économies d'échelle**: un CPU à très haute vitesse est plus cher que deux de moitié vitesse
  - **tolérance aux pannes**: en théorie la panne d'un CPU ne fait que réduire légèrement (facteur de $(N-1)/N$) le débit si le SE peut s'adapter à la panne
- **Bientôt tous les ordinateurs seront parallèles** (la loi de Moore se heurtera aux limites physiques)

SE Parallèle (2)

- Multiprocesseur **symétrique** (“SMP”): chaque processeur contient une copie du SE et agit comme les autres (mémoire partagée, migration de tâches et de données)

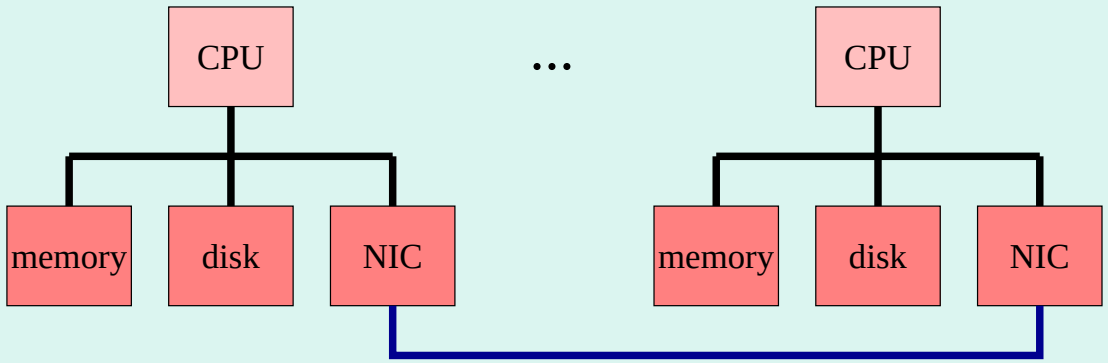


- Multiprocesseur **asymétrique**: des tâches spécifiques sont assignées à chaque processeur
  - Exemple: traitement de signal (video, audio, etc) par un pipeline de processeurs



SE Distribué

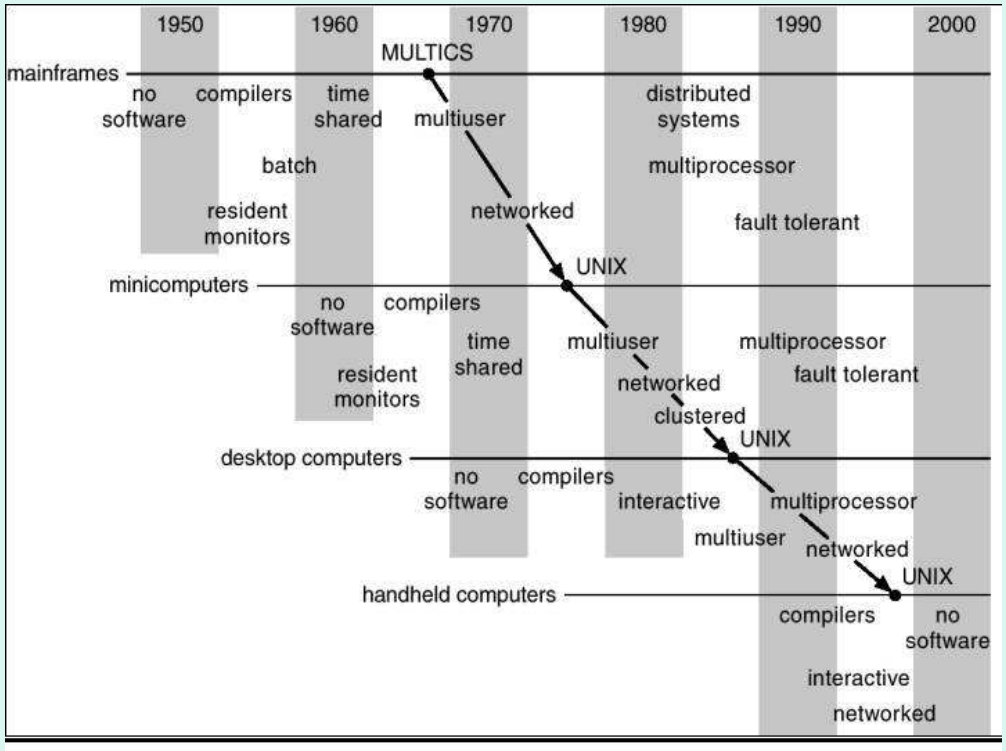
- Dans un **SE distribué** plusieurs processeurs sont interconnectés en réseau (de type LAN ou WAN) et la gestion des ressources se fait par **échange de messages**



- La **sécurité** et **efficacité** sont des considérations importantes dans les SE distribués

Migration des Fonctionalités

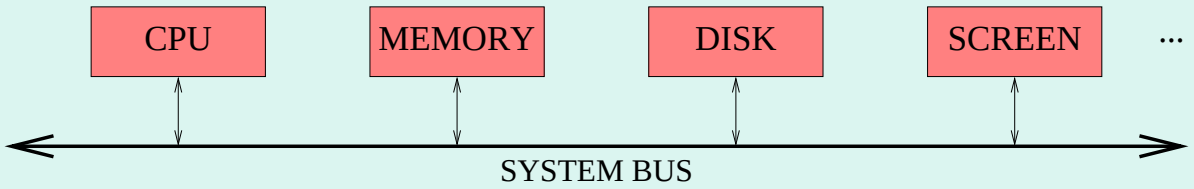
- Les fonctionalités des SE pour “mainframe” se font rapidement intégrer aux SE pour micro-ordinateurs



- Grâce aux avancement technologiques, il y a de moins en moins de différences entre un “mainframe” et un micro-ordinateur en terme de puissance

Le Matériel

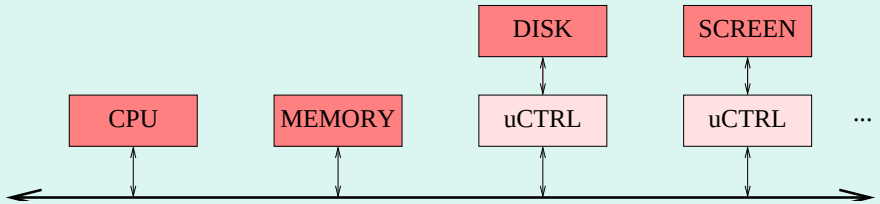
- Tout ordinateur a besoin de **périphériques** pour faire **l'entrée et la sortie** des données
- Architecture simplifiée:



- Il existe plusieurs types de périphériques
  - **Interface homme/machine**: clavier, souris, écran, carte de son, imprimante, caméra, scanner, joystick, écran tactile
  - **Stockage**: disque dur, floppy, ruban magnétique, CD-ROM, graveur de CD, CompactFlash
  - **Communication**: lien sériel, modem, carte réseau, infra-rouge, USB, FireWire
  - **Autre**: horloge, GPS, thermomètre, ventilateur

Les Périphériques (1)

- **Le CPU coordonne** les actions des périphériques
- Les périphériques plus complexes (disque, écran, son, réseau) sont en fait eux-même **composés de petits ordinateurs** (micro-contrôleurs) qui coordonnent les actions détaillées du périphérique

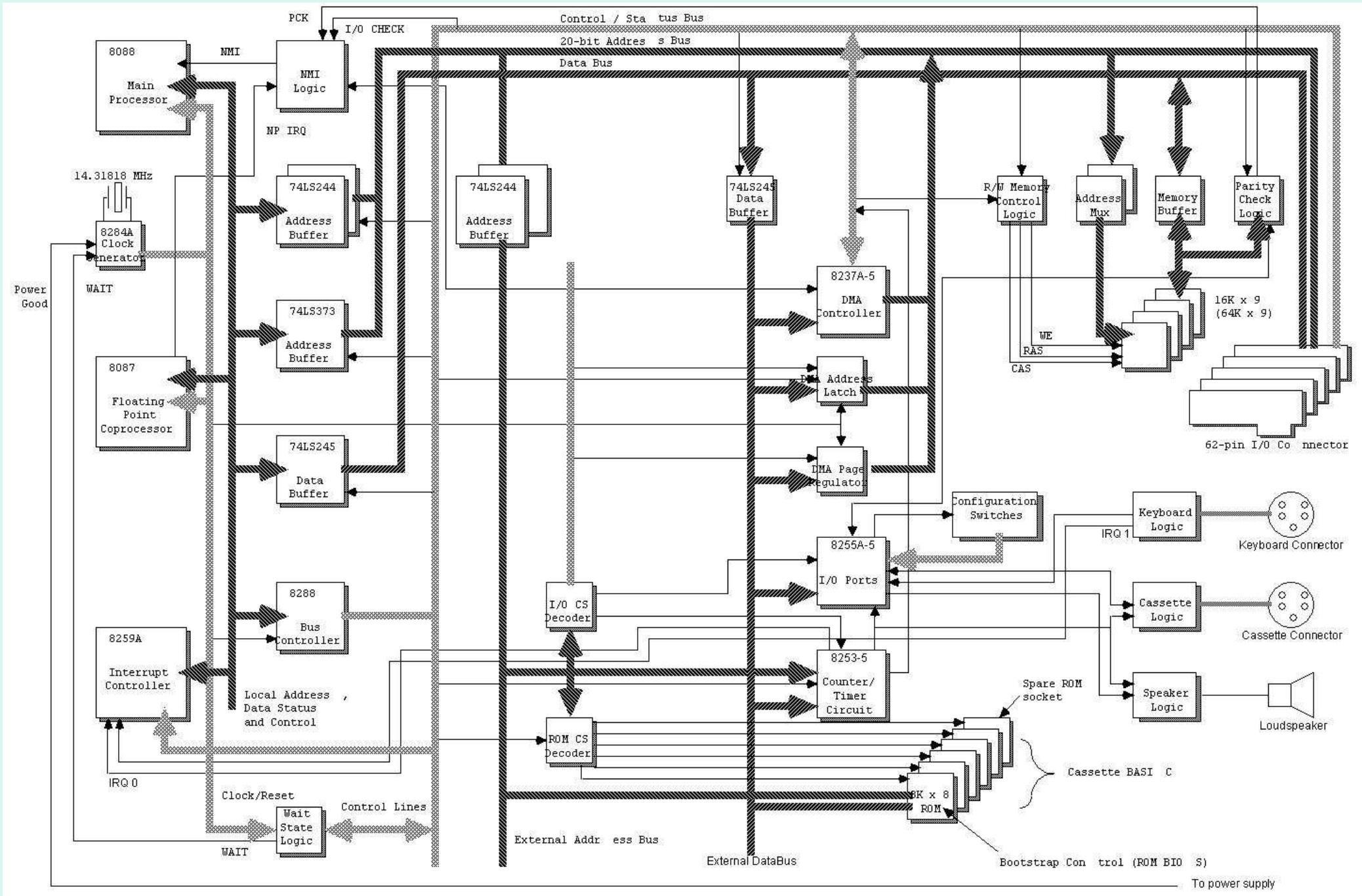


- Par exemple, un disque peut recevoir du CPU une commande de lecture du secteur N et le micro-contrôleur du disque fera le **positionnement de la tête de lecture**, **la synchronisation avec la rotation du disque** et le **transfert des informations** dans une mémoire tampon qui servira à satisfaire les lectures octet par octet du CPU

Les Périphériques (2)

- Puisque les micro-contrôleurs ont une certaine autonomie et indépendance, l'architecture d'un ordinateur typique est en fait celle d'un **multiprocesseur asymétrique**
- Exemple: les composantes pourraient simultanément
  - **CPU**: faire un calcul numérique
  - **CLAVIER**: accepter une entrée de l'usager
  - **DISQUE 1**: lire le secteur N
  - **DISQUE 2**: écrire le secteur M
- Un SE aura avantage à **exploiter ce parallélisme** pour améliorer la performance du système (il faut éviter les moments ou une ressource n'est pas utilisée)
- **Un SE moderne est un programme parallèle, qu'il soit pour un ordinateur à un ou plusieurs CPU**

Architecture de l'IBM-PC 5150

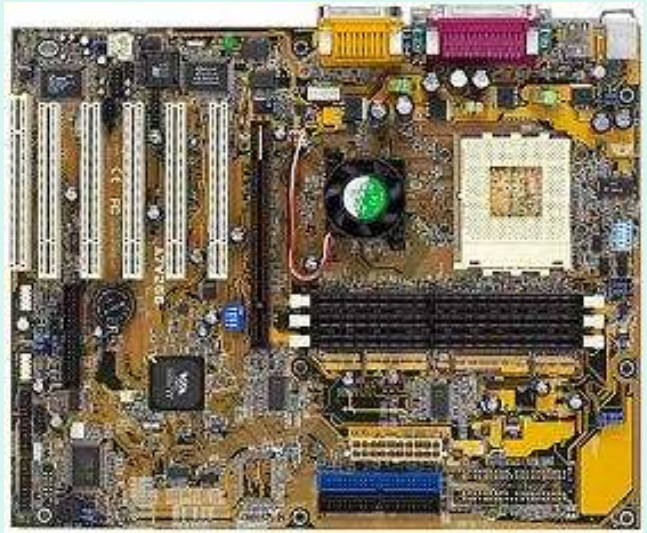


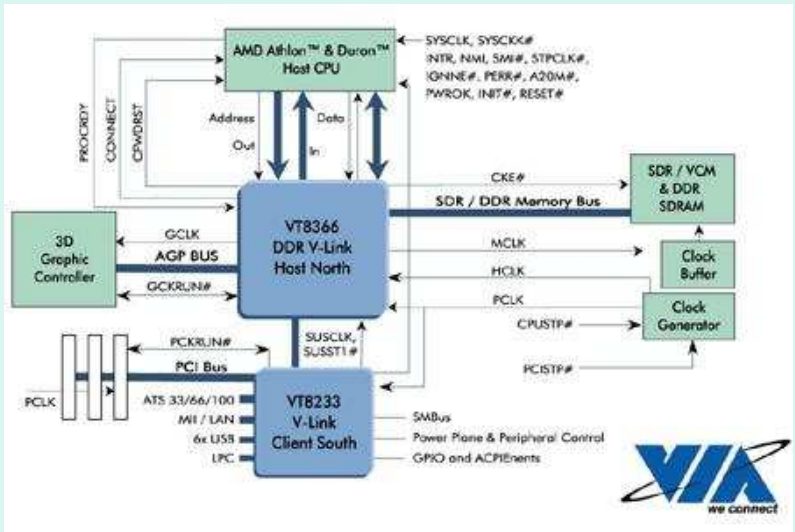
Périphériques “Classiques” de l’IBM-PC

- 2× **i8259A PIC** (Programmable Interrupt Controller): Gère les signaux d’interruption envoyés au CPU
- **MC146818 RTC** (Real-Time Clock): Horloge contenant l’heure présente et mémoire permanente de ˜100 octets
- **i8254 PIT** (Programmable Interval Timer): Génère des interruptions à intervalles réguliers ou après un certain laps de temps
- 2× **i8237A DMA** (Direct Memory Access): Transfère des blocs de données sans intervention du CPU
- **i8042** (Keyboard Controller): Clavier et souris
- **CRTC 6845** (Cathode Ray Tube Controller): Écran VGA
- **i8250 UART** (Universal Asynchronous Receiver Transmitter): Ports sériels (COM1/COM2)
- **Port Parallèle**: Imprimante
- **i8272A FDC** (Floppy Disc Controller): “floppy drive”
- **Contrôleur IDE** (Intelligent Drive Electronics): Disque dur

Architecture d'un PC Moderne

- Le "**chipset**" intègre les composantes principales





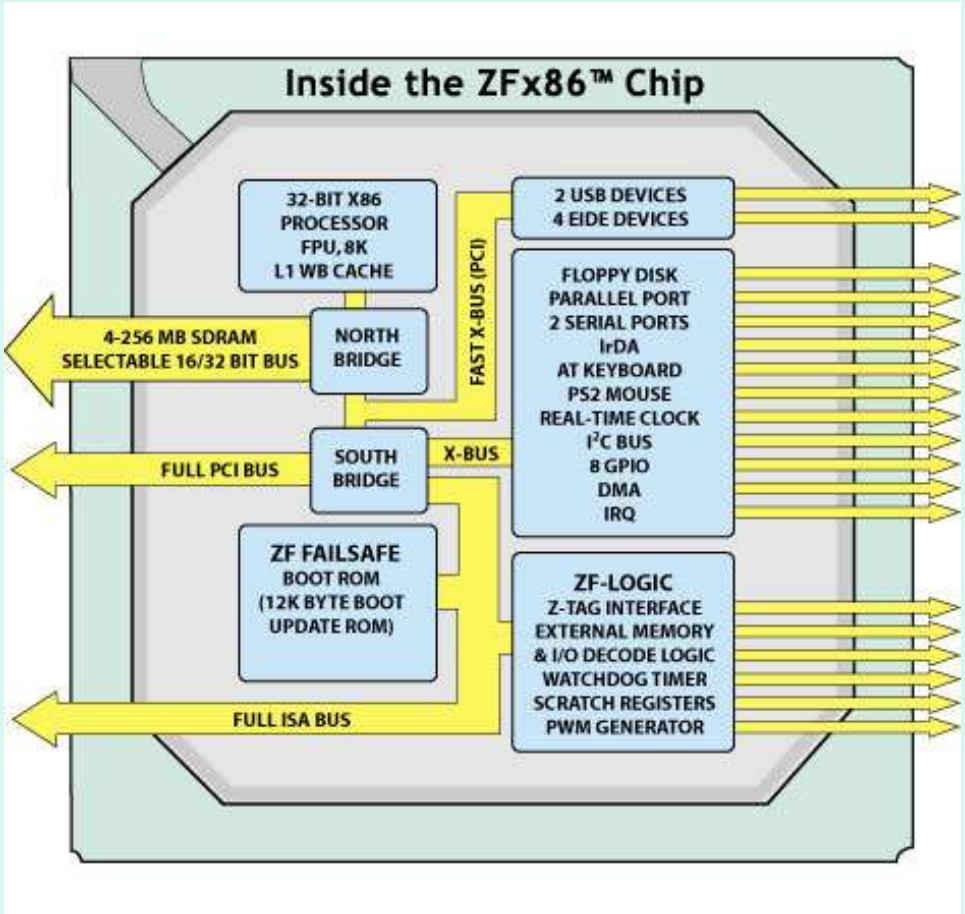
- Le CPU se branche sur le "**northbridge**" qui contient la logique de contrôle de la mémoire RAM et port AGP
- Le "**southbridge**" contient les circuits de contrôle des périphériques (horloge, clavier, disque, etc) de façon **compatible avec l'architecture des premiers PC** (afin de permettre aux vieux logiciels et SE de marcher)

Systèmes Embarqués (1)

- Un **système embarqué** (“embedded system”) est un ordinateur intégré à un appareil (électroménagé, machine industrielle, sonde, robot, automobile, avion, etc) qui sert à le contrôler ou receuillir des informations
- Dans le passé, de tels systèmes avaient **peu de mémoire et périphériques**, le CPU était **peu puissant**, les SE étaient **minimaux** et la programmation se fesait souvent en **assembleur**
- L’évolution technologique fait qu’on peut maintenant intégrer plusieurs fonctionalités sur une même puce et de la mémoire abondante ce qui permet d’utiliser des **SE et langages de programmation avancés** (comme Linux et Windows)

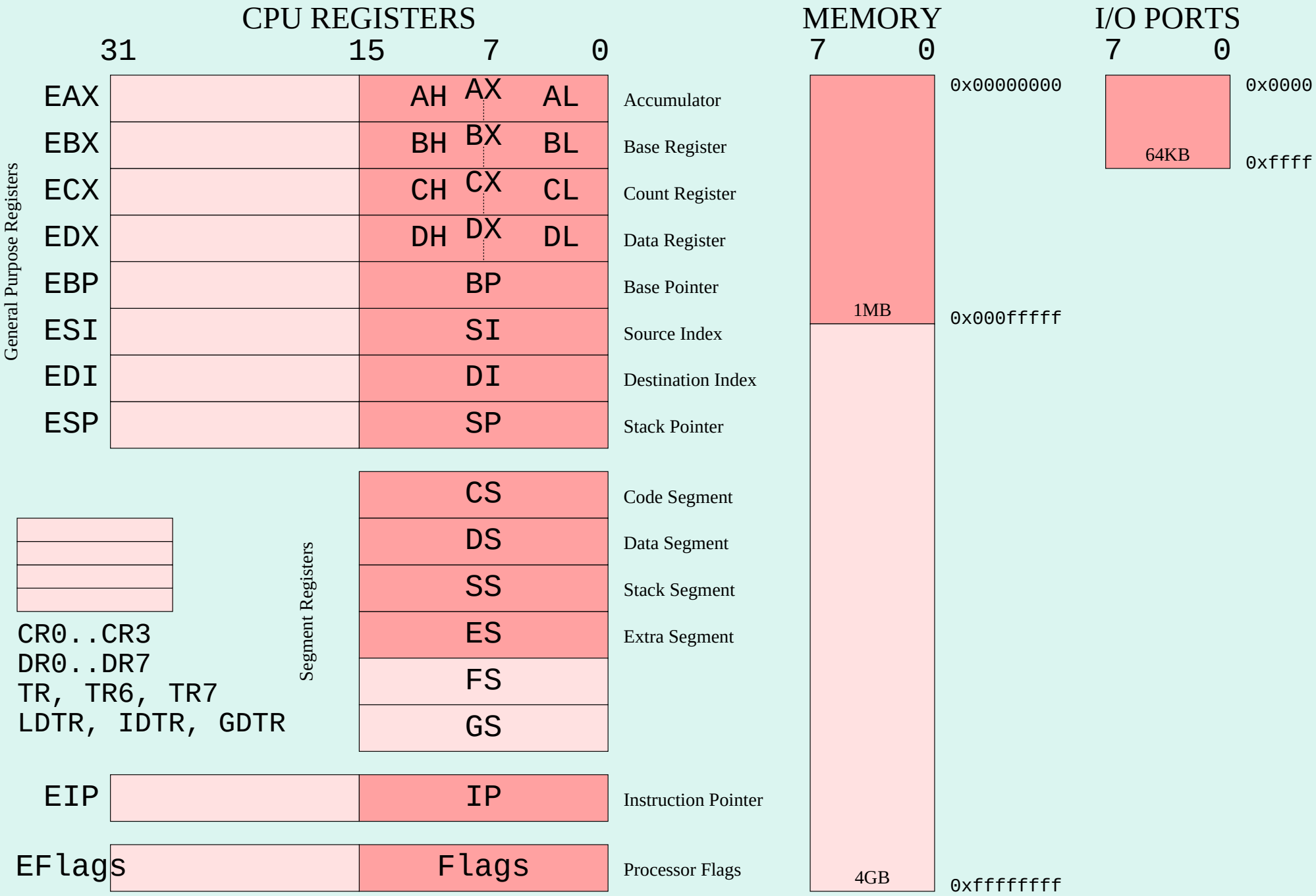
Systèmes Embarqués (2)

- Exemple: le ZFx86 “Embedded PC-on-a-chip”



- Supporte une variante temps-réel de Linux (LynxOS) et d’autres SE

Architecture Intel 8086/80386



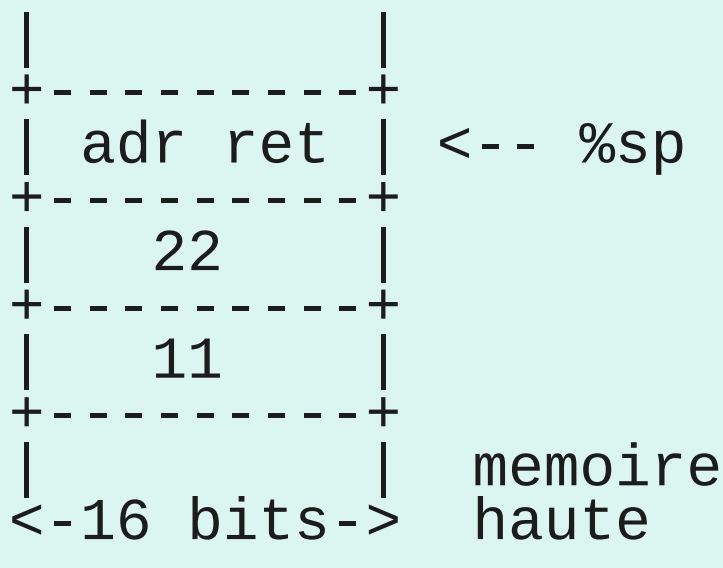
Assembleur 8086

- Exemple 1: somme des nombres de 0 à 10

```
1.  [b90a00]    mov   $10,%cx
2.  [b80000]    mov   $0,%ax
3.  [      ]  loop:
4.  [01c8  ]    add   %cx,%ax
5.  [49    ]    dec   %cx
6.  [75fb  ]    jnz   loop
```

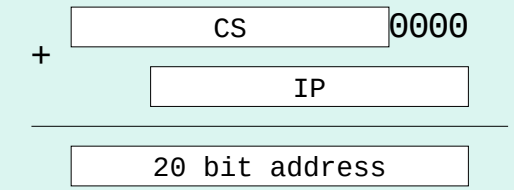
- Exemple 2: fonction “max” et appel

```
1.  [      ]  max:
2.  [89e5  ]    mov   %sp,%bp
3.  [8b4604]    mov   4(%bp),%ax
4.  [8b5e02]    mov   2(%bp),%bx
5.  [39d8  ]    cmp   %bx,%ax
6.  [7702  ]    ja    ax_is_max
7.  [89d8  ]    mov   %bx,%ax
8.  [      ]  ax_is_max:
9.  [c3    ]    ret
10.
11. [6a0b  ]    pushw $11
12. [6a16  ]    pushw $22
13. [e8xxxx]    call  max
14. [83c404]    add   $4,%sp
```

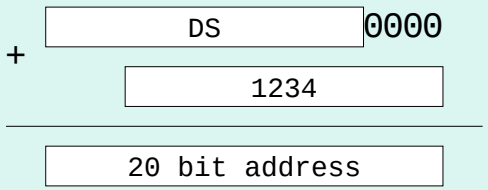


Le Mode “Réel”

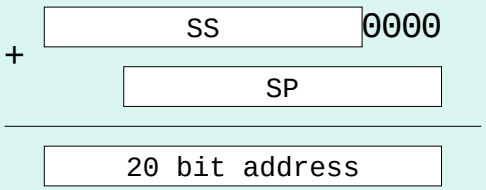
- L’adressage de la mémoire (données et programme) dans le 8086 se fait à l’aide de **segments**
- Les adresses de 20 bits (1MB) sont formées par la combinaison d’un **registre de segment** 16 bits et d’un **déplacement** 16 bit
- **adresse** = reg. segment $\times$ 16 + déplacement
- Le registre de segment utilisé depend de l’opération



`fetch instruction`



`mov 1234,%ax`



`push %ax`

- Les segments permettent d’avoir accès à plus de 64KB mais il est laborieux de manipuler des structures ou tableaux plus gros

Le Mode “Protégé” (1)

- À partir du 80386 l’adressage peut également se faire en mode “**protégé**” (permettant un accès à 4GB de mémoire)
- Au démarrage (et “reset”) le processeur se met en mode réel, pour des raisons de **compatibilité avec les processeurs précédents**
- Il faut mettre à 1 le bit 0 du registre `CR0` pour que le processeur se mette en mode protégé
- Dans ce mode il n’y a plus de segments (et de frontières de segments) ce qui facilite la programmation (un registre 32 bits peut contenir l’adresse de n’importe quel octet en mémoire)

Le Mode “Protégé” (2)

- Le processeur **interprète le code machine différemment** dans les deux modes!
- Il faut utiliser les directives “`.code16`” et “`.code32`” pour que l’assembleur sache quel encodage utiliser

```
1. [          ]  .code16
2. [b90a00    ]  mov    $10,%cx
3. [66b90a00  ]  mov    $10,%ecx  # invalide pour 8086!
4.
5. [          ]  .code32
6. [66b90a00  ]  mov    $10,%cx
7. [b90a00    ]  mov    $10,%ecx
```

- Le préfixe `0x66` sélectionne le mode inverse
- Il faut évidemment s’assurer (dans la logique du programme) que le code machine 16 bits est seulement exécuté en mode réel, et que le code machine 32 bits est seulement exécuté en mode protégé

Accès aux Périphériques de l'IBM-PC

- Le transfert d'information et de commandes entre le CPU et les périphériques se fait au moyen des **ports d'E/S** (particuliers à la famille Intel) et des zones de mémoire ("**memory mapped I/O**")
- Le CPU communique avec les périphériques au moyen des ports d'E/S (instructions 8086 "`out`" et "`in`")
- Une seule instruction `out` est requise si la commande est simple, sinon il en faut plusieurs (par exemple pour écrire sur un disque il faut envoyer le code de commande, le nombre de secteurs, les numéros de cylindre, tête, secteur et disque, en plus du contenu des secteurs à écrire)

Ports d'E/S Standard sur IBM-PC

- 0x000-0x00F: i8237 Direct Memory Access Controller
- 0x020-0x021: i8259A Programmable Interrupt Controller (MAÎTRE))
- 0x040-0x043: i8254 Programmable Interval Timer
- 0x060-0x064: i8042 Keyboard Controller
- 0x070-0x071: MC146818 RTC et contrôle du NMI (0x70=addr, 0x71=data)
- 0x0A0-0x0A1: i8259A Programmable Interrupt Controller (ESCLAVE)
- 0x1F0-0x1F7: Contôleur du disque dur (1)
- 0x2F8-0x2FF: i8250 Univ. Asynchronous Receiver Transmitter (COM2)
- 0x3F6-0x3F7: Contôleur du disque dur (2)
- 0x378-0x37F: Port Parallel
- 0x3B0-0x3DF: CRTC 6845 Video Graphics Adapter
- 0x3F0-0x3F7: i8272A Floppy Disc Controller
- 0x3F8-0x3FF: i8250 Univ. Asynchronous Receiver Transmitter (COM1)

Exemple : le port parallèle (1)

- Le port parallèle est branché à un cable parallèle (connecteur DB25) :

broche DB25	dir	signal			fonction
1	out	-Strobe	C0-		impulsion 0 = envoyer donnée
2	out	Data 0	D0		donnée 8 bit (bit 0)
3	out	Data 1	D1		" " (bit 1)
4	out	Data 2	D2		" " (bit 2)
5	out	Data 3	D3		" " (bit 3)
6	out	Data 4	D4		" " (bit 4)
7	out	Data 5	D5		" " (bit 5)
8	out	Data 6	D6		" " (bit 6)
9	out	Data 7	D7		" " (bit 7)
10	in	-Ack	S6+	IRQ	impulsion 0 = réception OK
11	in	+Busy	S7-		1 = occupée/erreur
12	in	+PaperEnd	S5+		1 = papier épuisé
13	in	+SelectIn	S4+		1 = imprimante sélectionnée
14	out	-AutoFd	C1-		0 = ‘‘autofeed’’
15	in	-Error	S3+		0 = erreur
16	out	-Init	C2+		impulsion 0 = mise-à-zéro
17	out	-Select	C3-		0 = selectionner

Exemple : le port parallèle (2)

- La valeur du port de sortie 0x378 est envoyée sur les signaux D0..D7
- La valeur du port d'entrée 0x379 vient des signaux S3..S7
- La valeur du port de sortie 0x37A est envoyée sur les signaux C0..C5
- Le port parallèle est utilisé principalement pour contrôler une **imprimante**

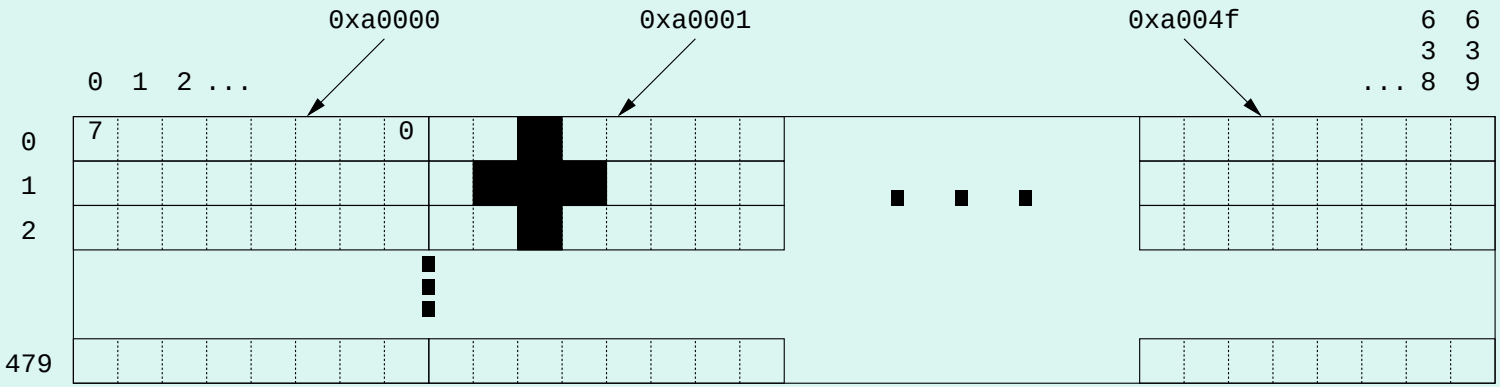
Exemple : le port parallèle (3)

- Pour envoyer un octet à l'imprimante il faut (1) placer l'octet sur les **8 lignes de donnée** du cable parallèle et (2) envoyer une courte **impulsion** sur la ligne "strobe"
- L'étape 1 se fait en envoyant l'octet au **port d'E/S 0x378**, l'étape 2 se fait en changeant le bit 0 du **port d'E/S 0x37a** à 1 puis à 0

```
1.    mov    $0x378,%dx
2.    out    %al,%dx      # suppose octet dans AL
3.
4.    mov    $0x37a,%dx
5.    in     %dx,%al
6.    or     $0x01,%al
7.    out    %al,%dx
8.
9.    nop # attente... (une boucle serait mieux)
10.   nop
11.
12.   and    $0xfe,%al
13.   out    %al,%dx
```

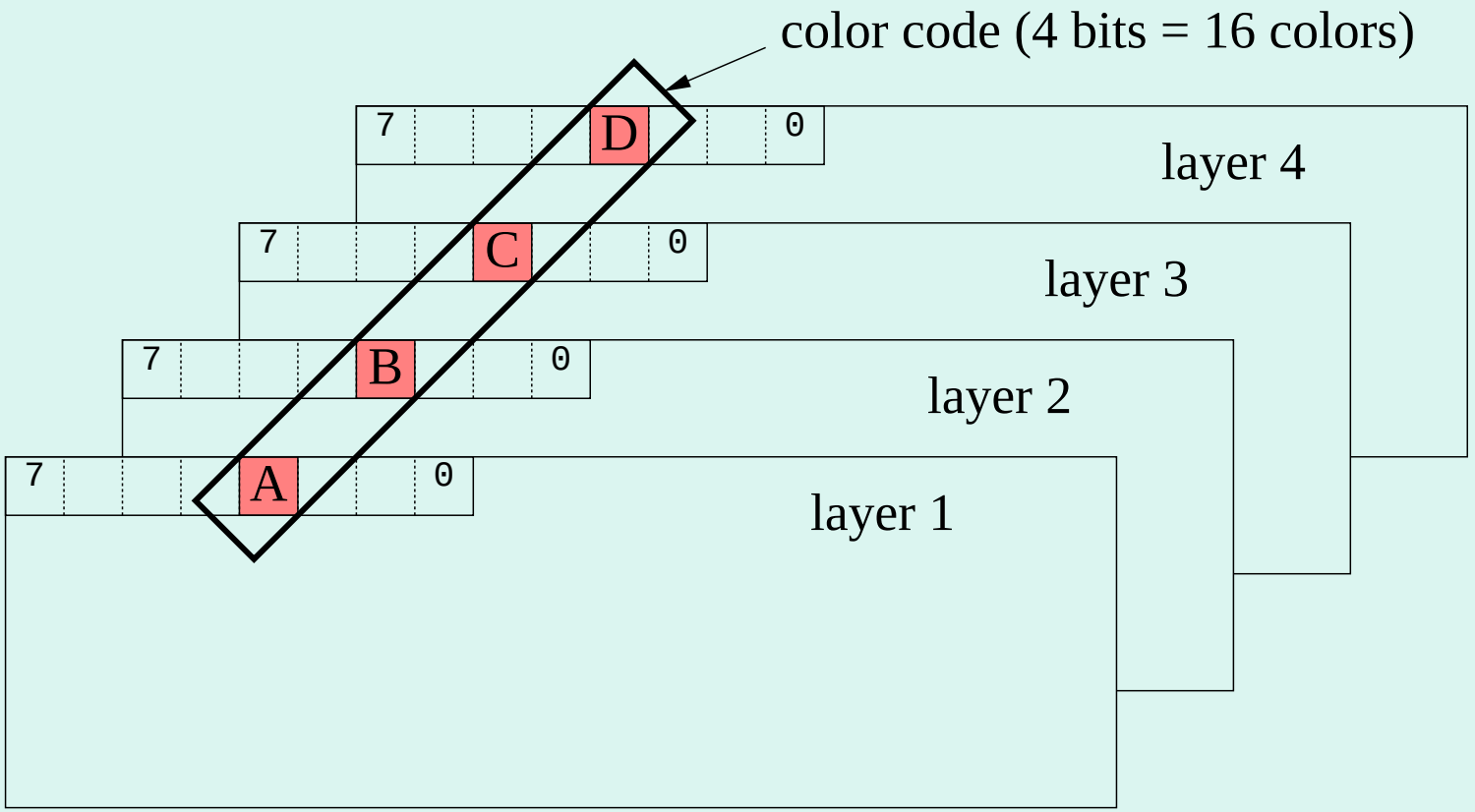
Exemple : accès à l'écran (1)

- À l'aide des ports d'E/S (0x3B0-0x3DF) l'adapteur video peut se configurer dans **plusieurs modes** (résolutions variées, noir-blanc/couleur, texte/graphique)
- En écrivant à des adresses mémoire de 0xa0000 à 0xbffff, c'est la **mémoire vidéo de l'adapteur** qui est modifiée ("**memory-mapped I/O**")
- Exemple : le mode graphique 17 (noir-blanc, 640 par 480) utilise la zone mémoire 0xa0000 à 0xa95ff où chaque bit contrôle un pixel



Exemple : accès à l'écran (2)

- Autre exemple: le mode graphique 18 (16 couleurs, 640 par 480) utilise aussi la zone mémoire 0xa0000 à 0xa95ff, mais il existe 4 couches qu'on doit selectionner une à la fois en envoyant des commandes aux ports d'E/S de l'adapteur



Exemple : accès à l'écran (3)

- Certains modes graphiques couleur associent un octet par pixel pour un jeu de 256 couleurs
- Les modes texte, utilisent deux octets par caractère: un pour le code du caractère et l'autre pour les attributs (couleur, luminosité)
- Les cartes video avancées permettent un grand nombre de couleurs et résolutions, et contiennent des processeurs puissants pour accélérer les opérations graphiques (2D et 3D)

Exemple : “Real-Time Clock” (1)

- “Spec sheet” du DS12887 (compatible au MC146818) :

Rev 0: 6/05

DALLAS SEMICONDUCTOR MAXIM

Real-Time Clock

DS12885/DS12887/DS12887A/DS12C887/DS12C887A

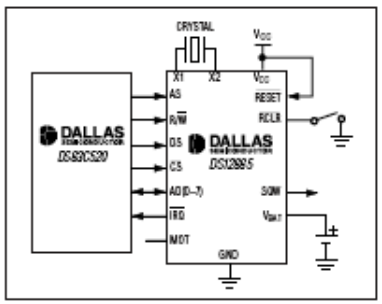
General Description

The DS12885, DS12887, and DS12C887 real-time clocks (RTCs) are designed to be direct replacements for the DS1285 and DS1287. The devices provide a real-time clock/calendar, one time-of-day alarm, three maskable interrupts with a common interrupt output, a programmable square wave, and 114 bytes of battery-backed static RAM (113 bytes in the DS12C887 and DS12C887A). The DS12887 integrates a quartz crystal and lithium energy source into a 24-pin encapsulated DIP package. The DS12C887 adds a century byte at address 32h. For all devices, the date at the end of the month is automatically adjusted for months with fewer than 31 days, including correction for leap years. The devices also operate in either 24-hour or 12-hour format with an AM/PM indicator. A precision temperature-compensated circuit monitors the status of V_{CC}. If a primary power failure is detected, the device automatically switches to a backup supply. A lithium coin-cell battery can be connected to the V_{BAT} input pin on the DS12885 to maintain time and date operation when primary power is absent. The device is accessed through a multiplexed byte-wide interface, which supports both Intel and Motorola modes.

Applications

Embedded Systems
Utility Meters
Security Systems
Network Hubs, Bridges, and Routers

Typical Operating Circuit



Features

- Drop-In Replacement for IBM AT Computer Clock/Calendar
- RTC Counts Seconds, Minutes, Hours, Day, Date, Month, and Year with Leap Year Compensation Through 2099
- Binary or BCD Time Representation
- 12-Hour or 24-Hour Clock with AM and PM in 12-Hour Mode
- Daylight Saving Time Option
- Selectable Intel or Motorola Bus Timing
- Interfaced with Software as 128 RAM Locations
- 14 Bytes of Clock and Control Registers
- 114 Bytes of General-Purpose, Battery-Backed RAM (113 Bytes in the DS12C887 and DS12C887A)
- RAM Clear Function (DS12885, DS12887A, and DS12C887A)
- Interrupt Output with Three Independently Maskable Interrupt Flags
- Time-of-Day Alarm Once Per Second to Once Per Day
- Periodic Rates from 122µs to 500ms
- End-of-Clock Update Cycle Flag
- Programmable Square-Wave Output
- Automatic Power-Fail Detect and Switch Circuitry
- Optional 28-Pin PLCC Surface Mount Package or 32-Pin TQFP (DS12885)
- Optional Encapsulated DIP (EDIP) Package with Integrated Crystal and Battery (DS12887, DS12887A, DS12C887, DS12C887A)
- Optional Industrial Temperature Range Available
- Underwriters Laboratory (UL) Recognized

Pin Configurations and Ordering Information appear at end of data sheet.

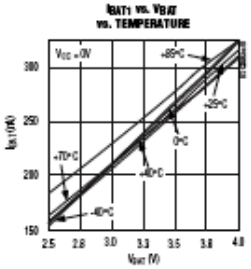
For pricing, delivery, and ordering information, please contact Maxim/Dallas Direct! at 1-888-629-4642, or visit Maxim's website at www.maxim-ic.com.

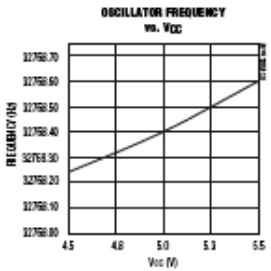
Real-Time Clock

DS12885/DS12887/DS12887A/DS12C887/DS12C887A

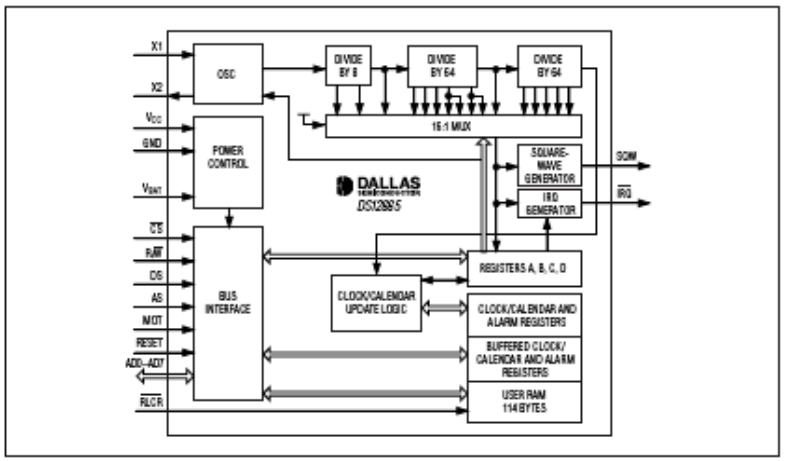
Typical Operating Characteristics

(V_{CC} = +5.0V, T_A = +25°C, unless otherwise noted.)



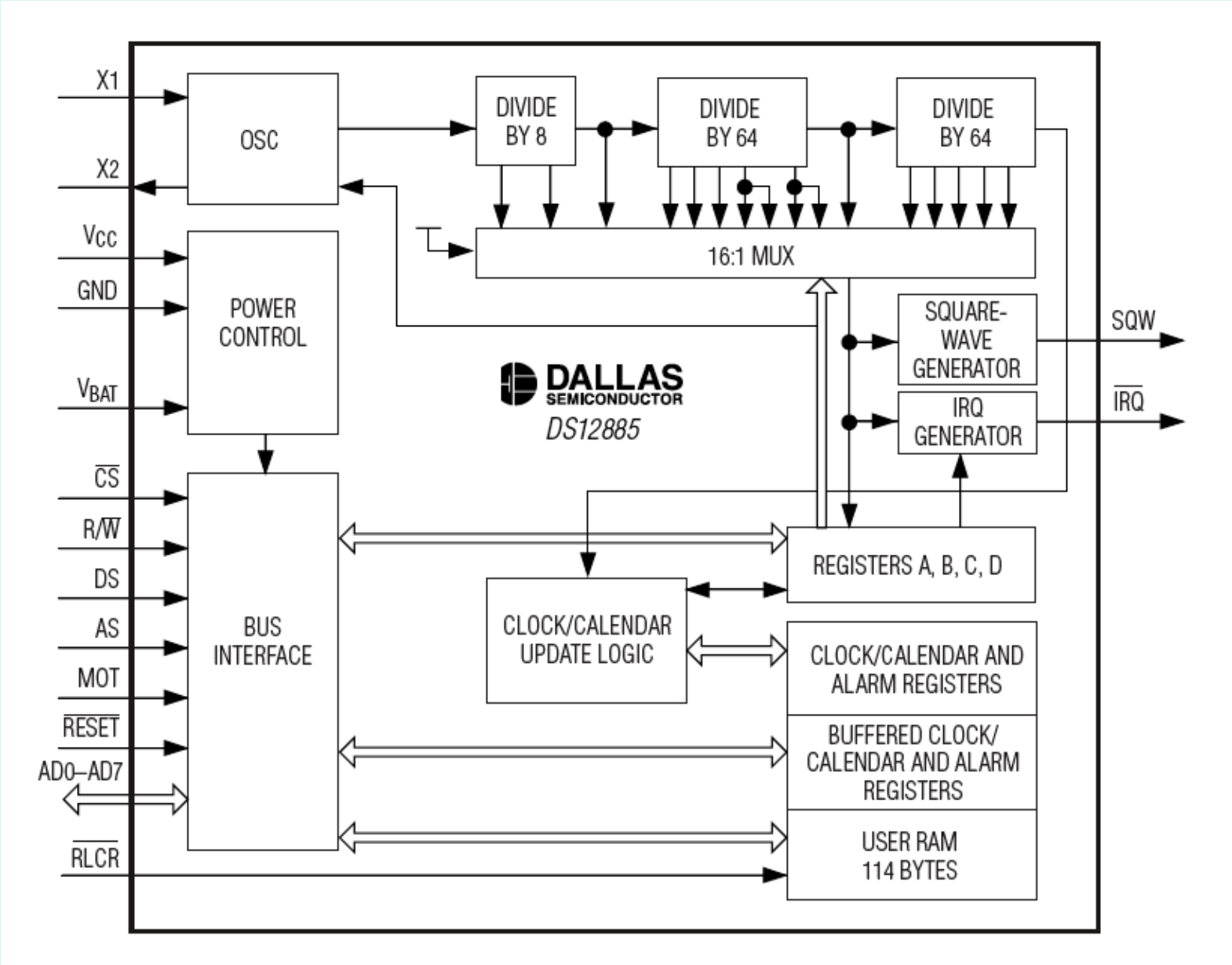


Functional Diagram



Exemple : “Real-Time Clock” (2)

- Le schéma fonctionnel du RTC :



Exemple : “Real-Time Clock” (3)

- Les registres du RTC :

Table 2B. Time, Calendar, and Alarm Data Modes—Binary Mode (DM = 1)

ADDRESS	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	FUNCTION	RANGE
00H	0	0	Seconds						Seconds	00–3B
01H	0	0	Seconds						Seconds Alarm	00–3B
02H	0	0	Minutes						Minutes	00–3B
03H	0	0	Minutes						Minutes Alarm	00–3B
04H	AM/PM	0	0	0	Hours				Hours	01–0C +AM/PM 00–17
	0			Hours						
05H	AM/PM	0	0	0	Hours				Hours Alarm	01–0C +AM/PM 00–17
	0			Hours						
06H	0	0	0	0	0		Day		Day	01–07
07H	0	0	0	Date					Date	01–1F
08H	0	0	0	0	Month				Month	01–0C
09H	0	Year							Year	00–63
0AH	UIP	DV2	DV1	DV0	RS3	RS2	RS1	RS0	Control	—
0BH	SET	PIE	AIE	UIE	SQWE	DM	24/12	DSE	Control	—
0CH	IRQF	PF	AF	UF	0	0	0	0	Control	—
0DH	VRT	0	0	0	0	0	0	0	Control	—
0EH-31H	X	X	X	X	X	X	X	X	RAM	—
32H	N/A				N/A				Century*	—
33H-7FH	X	X	X	X	X	X	X	X	RAM	—

X = Read/Write Bit.

Exemple : “Real-Time Clock” (4)

- Le registre de contrôle A :

Control Register A

MSB **LSB**

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
UIP	DV2	DV1	DV0	RS3	RS2	RS1	RS0

Bit 7: Update-In-Progress (UIP). This bit is a status flag that can be monitored. When the UIP bit is a 1, the update transfer occurs soon. When UIP is a 0, the update transfer does not occur for at least 244µs. The time, calendar, and alarm information in RAM is fully available for access when the UIP bit is 0. The UIP bit is read-only and is not affected by $\overline{\text{RESET}}$. Writing the SET bit in Register B to a 1 inhibits any update transfer and clears the UIP status bit.

Bits 6, 5, and 4: DV2, DV1, DV0. These three bits are used to turn the oscillator on or off and to reset the countdown chain. A pattern of 010 is the only combination of bits that turn the oscillator on and allow the RTC to keep time. A pattern of 11x enables the oscillator but holds the countdown chain in reset. The next update occurs at 500ms after a pattern of 010 is written to DV0, DV1, and DV2.

Bits 3 to 0: Rate Selector (RS3, RS2, RS1, RS0). These four rate-selection bits select one of the 13 taps on the 15-stage divider or disable the divider output. The tap selected can be used to generate an output square wave (SQW pin) and/or a periodic interrupt. The user can do one of the following:

1) Enable the interrupt with the PIE bit;
2) Enable the SQW output pin with the SQWE bit;
3) Enable both at the same time and the same rate; or
4) Enable neither.

Table 3 lists the periodic interrupt rates and the square-wave frequencies that can be chosen with the RS bits. These four read/write bits are not affected by $\overline{\text{RESET}}$.

- La lecture de l’heure/date doit débuter lorsque UIP=0
- RS3..RS0=0000 ⇒ SQW=rien, RS3..RS0=0011 ⇒ SQW=8192Hz, RS3..RS0=1111 ⇒ SQW=2Hz

Real-Time Clock : “mc146818.h” de FreeBSD (1)


```
1.  /* The registers, and the bits within each register. */
2.
3.  #define MC_SEC           0x0     /* Time of year: seconds (0-59) */
4.  #define MC_ASEC          0x1     /* Alarm: seconds */
5.  #define MC_MIN           0x2     /* Time of year: minutes (0-59) */
6.  #define MC_AMIN          0x3     /* Alarm: minutes */
7.  #define MC_HOUR          0x4     /* Time of year: hour (see above) */
8.  #define MC_AHOUR         0x5     /* Alarm: hour */
9.  #define MC_DOW           0x6     /* Time of year: day of week (1-7) */
10. #define MC_DOM           0x7     /* Time of year: day of month (1-31) */
11. #define MC_MONTH         0x8     /* Time of year: month (1-12) */
12. #define MC_YEAR          0x9     /* Time of year: year in century (0-99) */
13.
14. #define MC_REGA          0xa     /* Control register A */
15.
16. #define  MC_REGA_RSMASK 0x0f     /* Interrupt rate select mask (see below) */
17. #define  MC_REGA_DVMASK 0x70     /* Divisor select mask (see below) */
18. #define  MC_REGA_UIP    0x80     /* Update in progress; read only. */
19.
20. #define MC_REGB          0xb     /* Control register B */
21.
22. #define  MC_REGB_DSE    0x01     /* Daylight Savings Enable */
23. #define  MC_REGB_24HR   0x02     /* 24-hour mode (AM/PM mode when clear) */
24. #define  MC_REGB_BINARY 0x04     /* Binary mode (BCD mode when clear) */
25. #define  MC_REGB_SQWE   0x08     /* Square Wave Enable */
26. #define  MC_REGB_UIE    0x10     /* Update End interrupt enable */
27. #define  MC_REGB_AIE    0x20     /* Alarm interrupt enable */
28. #define  MC_REGB_PIE    0x40     /* Periodic interrupt enable */
29. #define  MC_REGB_SET    0x80     /* Allow time to be set; stops updates */
30.
31. #define MC_REGC          0xc     /* Control register C */
```

Real-Time Clock : “mc146818.c” de FreeBSD (1)


```
1.  /* Get time of day and convert it to a struct timespec.
2.   * Return 0 on success, an error number otherwise.
3.   */
4.
5.  int mc146818_gettime(device_t dev, struct timespec *ts)
6.  { struct mc146818_softc *sc = device_get_softc(dev);
7.    struct clocktime ct;
8.    int timeout, cent, year;
9.
10.   timeout = 1000000;   /* XXX how long should we wait? */
11.
12.   /* If MC_REGA_UIP is 0 we have at least 244us before
13.    * the next update. If it's 1 an update is imminent.
14.    */
15.   for (;;) {
16.     mtx_lock_spin(&sc->sc_mtx);
17.     if (!((*sc->sc_mcread)(dev, MC_REGA) & MC_REGA_UIP))
18.       break;
19.     mtx_unlock_spin(&sc->sc_mtx);
20.     if (--timeout < 0) {
21.       device_printf(dev, "%s: timeout\n", __func__);
22.       return (EBUSY);
23.     }
24.   }
```

Real-Time Clock : “mc146818.c” de FreeBSD (2)


```
25. #define FROMREG(x) \
26.   ((sc->sc_flag & MC146818_BCD) ? FROMBCD(x) : (x))
27.
28.   ct.nsec = 0;
29.   ct.sec = FROMREG((*sc->sc_mcread)(dev, MC_SEC));
30.   ct.min = FROMREG((*sc->sc_mcread)(dev, MC_MIN));
31.   ct.hour = FROMREG((*sc->sc_mcread)(dev, MC_HOUR));
32.   /* Map dow from 1 - 7 to 0 - 6. */
33.   ct.dow = FROMREG((*sc->sc_mcread)(dev, MC_DOW)) - 1;
34.   ct.day = FROMREG((*sc->sc_mcread)(dev, MC_DOM));
35.   ct.mon = FROMREG((*sc->sc_mcread)(dev, MC_MONTH));
36.   year = FROMREG((*sc->sc_mcread)(dev, MC_YEAR));
37.   year += sc->sc_year0;
38.   if (sc->sc_flag & MC146818_NO_CENT_ADJUST) {
39.       cent = (*sc->sc_getcent)(dev);
40.       year += cent * 100;
41.   } else if (year < POSIX_BASE_YEAR)
42.       year += 100;
43.   mtx_unlock_spin(&sc->sc_mtx);
44.
45.   ct.year = year;
46.
47.   return (clock_ct_to_ts(&ct, ts));
48. }
```

Langages d'Implantation (1)

- L'utilisation de **langages de haut-niveau** facilite l'écriture de tout programme complexe
- Un SE a cependant besoin d'**accéder directement le matériel** de la machine (registres spéciaux du CPU, instructions spéciales, etc) ce qui n'est normalement possible qu'en assembleur
- Certaines parties du SE sont donc normalement écrites en **assembleur** ("boot loader", changement en mode protégé, configuration de la mémoire virtuelle, etc)

Langages d'Implantation (2)

- Pour le reste, un **langage de niveau intermédiaire** (comme C) est approprié puisqu'il permet d'exprimer des algorithmes complexe et aussi d'accéder à certaines ressources directement (comme la mémoire)
- En C, les pointeurs sont très utiles:

```
1. char* ecran = (char*) 0xa0000;
2.
3. for (i=0; i<640*480/8; i++)
4.   ecran[i] = 0;
```

Langages d'Implantation (3)

- Le code C peut facilement être combiné avec du code en assembleur, pour avoir les **avantages des deux langages**

```
1. // fichier: "floppy.cpp"
2.
3. extern "C" void outb (int val, int port);
4.
5. void floppy_off ()
6. {
7.   outb (0, 0x3f2);
8. }
```

```
1. # fichier: "io.s"
2.
3.  .globl outb
4. outb:
5.  movl 4(%esp),%eax
6.  movl 8(%esp),%edx
7.  outb %al,%dx
8.  ret
```

Langages d'Implantation (4)

- L'**éditeur de liens** combine les divers fichiers objets (".o") produits pour obtenir un programme (ou "kernel") exécutable:

```
% g++ -c floppy.cpp   <-- génère "floppy.o"
% as -o io.o io.s     <-- génère "io.o"
% ld floppy.o io.o -o kernel -oformat binary  <-- génère
                                                  "kernel"
```

Langages d'Implantation (5)

- Plusieurs compilateurs C, comme `gcc`, supportent des **extensions au langage C** qui facilitent l'inclusion de code assembleur dans le code C

- Par exemple l'instruction "`asm`" de `gcc`:

```
void floppy_off ()
{
  asm ("movb $0,%al;outb %al,$0x3f2");
}
```

Langages d'Implantation (6)

- L'instruction "`asm`" peut prendre des paramètres:

```
1. #define outb(val,port)                                  \
2. asm ("outb %b0,%w1" : : "a" (val), "d" (port))
3.
4. #define inb(port)                                       \
5. ({                                                      \
6.    char val;                                            \
7.    asm ("inb %w1,%b0" : "=a" (val) : "d" (port)); \
8.    val;                                                 \
9. })
```

Note: "=" ⟶ sortie, "a" ⟶ EAX, "d" ⟶ EDX

- La macro "`inb`" exploite une autre extension de `gcc`, l'**expression énoncé**: "`({ ... })`"

```
z = 2 * ({ int a = x, b = y;
           if (a < b) a = b;
           a;
         }) + 1;
```

Approches d'E/S (1)

- Une E/S sur un périphérique demande une **interaction entre le CPU et le contrôleur** de périphérique:

Entrée

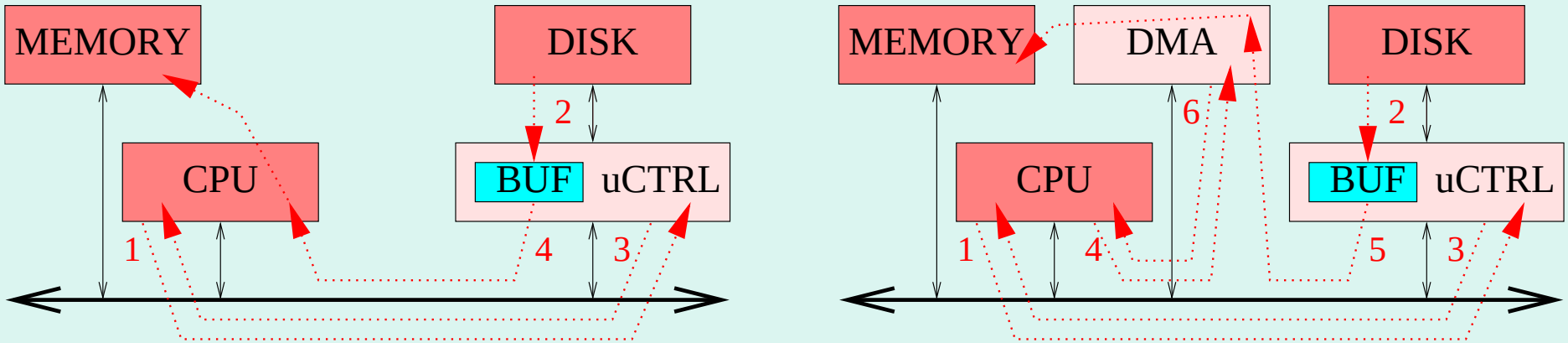
1. le CPU envoit la **commande** de lecture au contrôleur
2. le contrôleur **opère** le périphérique
3. le contrôleur **informe** le CPU que l'opération est complétée
4. le CPU récupère le **code d'erreur** du contrôleur (si une erreur est possible) ainsi que les **données** du **tampon interne**

Sortie

1. le CPU envoit la **commande** d'écriture au contrôleur ainsi que les **données** qui sont placées dans un **tampon interne**
2. le contrôleur **opère** le périphérique
3. le contrôleur **informe** le CPU que l'opération est complétée
4. le CPU récupère le **code d'erreur** du contrôleur (si une erreur est possible)

Approches d'E/S (2)

- Schématiquement, pour une entrée:



- Les étapes 2 et 5 peuvent prendre un **temps important**
  - **borné**: E/S disque, sortie vers lien sériel, opération graphique
  - **indéterminé**: entrée d'un lien sériel/clavier/souris, sortie vers imprimante, E/S vers réseau
- Il faut **éviter de faire attendre le CPU** (une option: faire le transfert de donnée par DMA pour **alléger la charge de travail du CPU**)

E/S Synchrone et Asynchrone

- Une E/S est **synchrone** si le programme qui demande l'opération d'E/S est **inactif** pendant l'opération d'E/S (en d'autres termes, le programme ne fera du travail "utile" qu'une fois que l'opération d'E/S est complétée)
- Une E/S est **asynchrone** si le programme qui demande l'opération d'E/S a la possibilité de faire du travail utile pendant l'opération d'E/S (le contrôle revient au programme avant que l'E/S complète ou même commence); éventuellement le programme est **informé** que l'E/S est complétée
- L'utilisation d'E/S **synchrone** dans un programme mène à des programmes **simples** et **compréhensibles**, mais s'il y a un seul programme il n'y aura **pas de parallélisme**

Polling et Interruption

- Il y a deux approches pour que le CPU soit **informé d'un événement** par un contrôleur de périphérique (p.e. opération d'E/S devenue possible ou complétée)
  - "**Polling**" (approche active): le CPU examine le statut du contrôleur **répétitivement**, jusqu'à ce que le statut indique l'occurrence de l'événement

    Analogie: **lecture de son courriel**
  - **Interruption** (approche passive): le CPU reçoit une interruption du contrôleur à l'occurrence de l'événement; la routine d'interruption s'occupe du traitement de l'événement

    Analogie: **réception d'un appel téléphonique**

Exemples d'E/S par "Polling"

- Lecture **synchrone** du clavier avec **attente active** (port 0x60 = tampon, port 0x64 = statut)

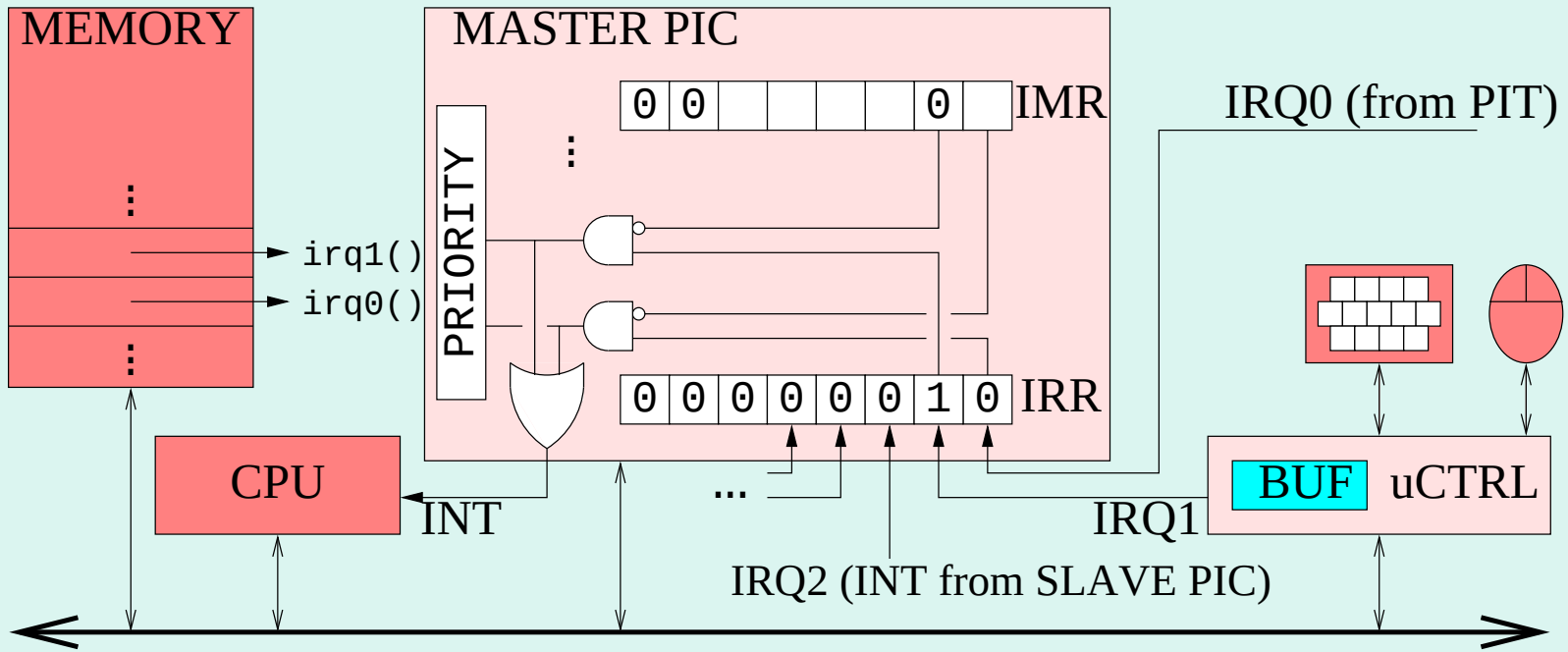
```
1. for (;;)
2.   {
3.     for (;;)
4.       if (inb (0x64) & 1) break;
5.     cout << "keycode=" << inb (0x60);
6.   }
```

- Lecture **asynchrone** du clavier

```
1. x = 0;
2. for (i=0; i<10000000; i++)
3.   {
4.     x += i; // travail utile
5.     if (inb (0x64) & 1)
6.       cout << "keycode=" << inb (0x60);
7.   }
8. cout << x;
```

Le contrôleur d'interruption programmable (PIC)

- Le "**Programmable Interrupt Controller**" gère les interruptions par priorité



- Lorsque le CPU est interrompu par le PIC, il demande le niveau de priorité au PIC et s'en sert pour choisir la routine de traitement d'interruption dans la **table de vecteurs d'interruption**

IRQ et vecteurs d'interruption (1)

- PIC "MASTER" :

IRQ	Vecteur	Périphérique
IRQ0	0x08	"Programmable Interval Timer" (i8254)
IRQ1	0x09	Contrôleur de clavier et souris (i8042)
IRQ2	0x0A	Interruption du SLAVE PIC
IRQ3	0x0B	Ports sériels COM2 et COM4
IRQ4	0x0C	Ports sériels COM1 et COM3
IRQ5	0x0D	Port parallèle 2
IRQ6	0x0E	Contrôleur de floppy
IRQ7	0x0F	Port parallèle 1

IRQ et vecteurs d'interruption (2)

- PIC "SLAVE" :

IRQ	Vecteur	Périphérique
IRQ8	0x70	"Real-Time Clock" (MC146818)
IRQ9	0x71	libre
IRQ10	0x72	libre
IRQ11	0x73	libre
IRQ12	0x74	Souris PS/2
IRQ13	0x75	Coprocesseur numérique
IRQ14	0x76	Contrôleur IDE primaire
IRQ15	0x77	Contrôleur IDE secondaire

Exemple d'E/S par Interruption (1)

- Lecture **asynchrone** du clavier par **interruption** (si on suppose que la routine d'interruption pour IRQ1 se nomme "`irq1`")

```
1.  int main ()
2.  {
3.    // mise-à-zero du bit 1 du IMR
4.    outb (inb (0x21) & 0xfd, 0x21);
5.    ...
6.  }
7.
8.  extern "C" void irq1 ()
9.  {
10.   cout << "keycode=" << inb (0x60);
11.
12.   // mise-à-zero du bit 1 du IRR
13.   outb (0x61, 0x20);
14. }
```

Exemple d’E/S par Interruption (2)

- Lecture **synchrone** du clavier par **interruption**

```
1. int main ()
2. {
3.   // mise-à-zero du bit 1 du IMR
4.   outb (inb (0x21) & 0xfd, 0x21);
5.
6.   // attente ‘‘faible courant’’ d’une interruption
7.   asm ("hlt");
8.   ...
9. }
```

Processus

- Un **programme** c'est une description d'un calcul à faire (c'est-à-dire un algorithme)
- Un **processeur** c'est une machine qui exécute un programme
- Un **processeur réel** c'est une machine réelle (avec un CPU, mémoire, etc)
- Un **processus** c'est un **processeur virtuel** qui est simulé par un autre processeur (réel ou virtuel)
- Par **multiplexage du CPU** il est donc possible d'avoir plus d'un processus qui évolue sur un même ordinateur
- **Chaque processus a l'illusion d'être sur un processeur qui lui est dédié**

Processus: à Quoi Bon? (1)

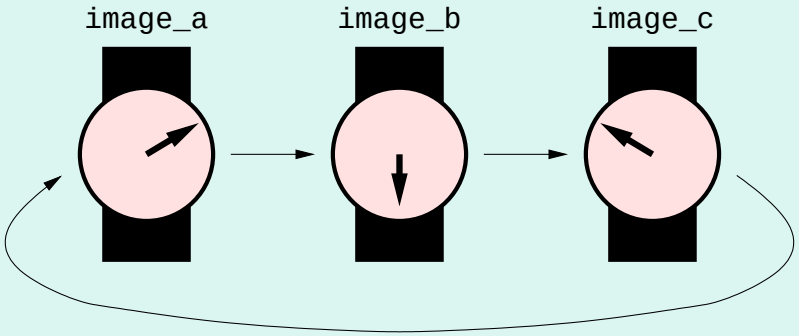
- Tout calcul qui peut se faire en utilisant des processus **pourrait aussi se faire sans processus** (un seul programme sur un processeur réel)
- La raison d'être des processus c'est **que ça facilite la conception de certains systèmes complexes**
- Si on décompose un système en processus celui-ci devient plus **modulaire**
  - chaque processus vise une tâche précise (**spécification simple**, **maintenable**, **réutilisable**)
  - l'intégrité des processus peut être garanti (principes de **découplage du contrôle**, d'**encapsulation** et de **protection**)
  - exécution parallèle (si on a un multiprocesseur et le programme a du parallélisme)

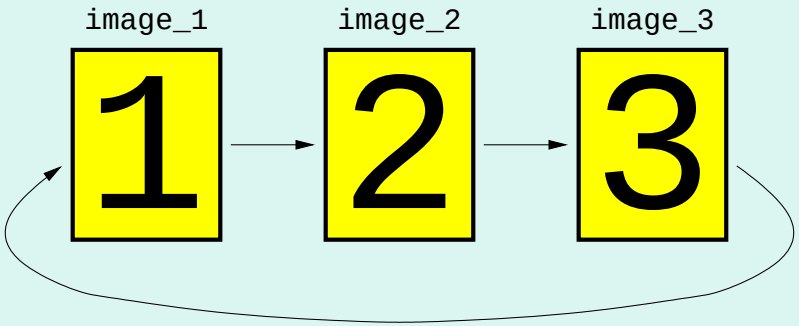
Processus: à Quoi Bon? (2)

- Les processus sont particulièrement bien adaptés lorsqu'il y a plus d'un calcul à effectuer, **et chaque calcul évolue à une vitesse différente**
- Exemples:
  - des processus qui interagissent avec des usagers
  - simulations (processus "modélisation", processus "affichage")
  - serveur web (différentes requêtes)

Exemple d'Animation (1)

- Pour motiver l'utilisation des processus, essayons d'écrire un programme qui anime deux images à l'écran (une montre et un compteur)





- On suppose l'existence d'une fonction "`draw`" qui trace une image au bon endroit sur l'écran

Exemple d'Animation (2)


```
for (;;)
  {
    draw (image_a);
    draw (image_1);

    draw (image_b);
    draw (image_2);

    draw (image_c);
    draw (image_3);
  }
```

- Le programme ci-dessus s'occupe **à la fois d'animer la montre et le compteur**
- Ça fonctionne mais **ce n'est pas modulaire**; les deux animations sont **entremêlées**
  - Que faire si l'animation du compteur n'a que 2 images?
  - Comment faire pour ajouter une troisième animation?

Exemple d'Animation (3)

process 1

```
for (;;)
  {
    draw (image_a);
    draw (image_1);

    draw (image_b);
    draw (image_2);

    draw (image_c);
    draw (image_3);
  }
```

process 2

- En séparant les deux animations on s'apperçoit que 2 processus sont impliqués
- Chaque processus s'occupe d'**une seule animation**

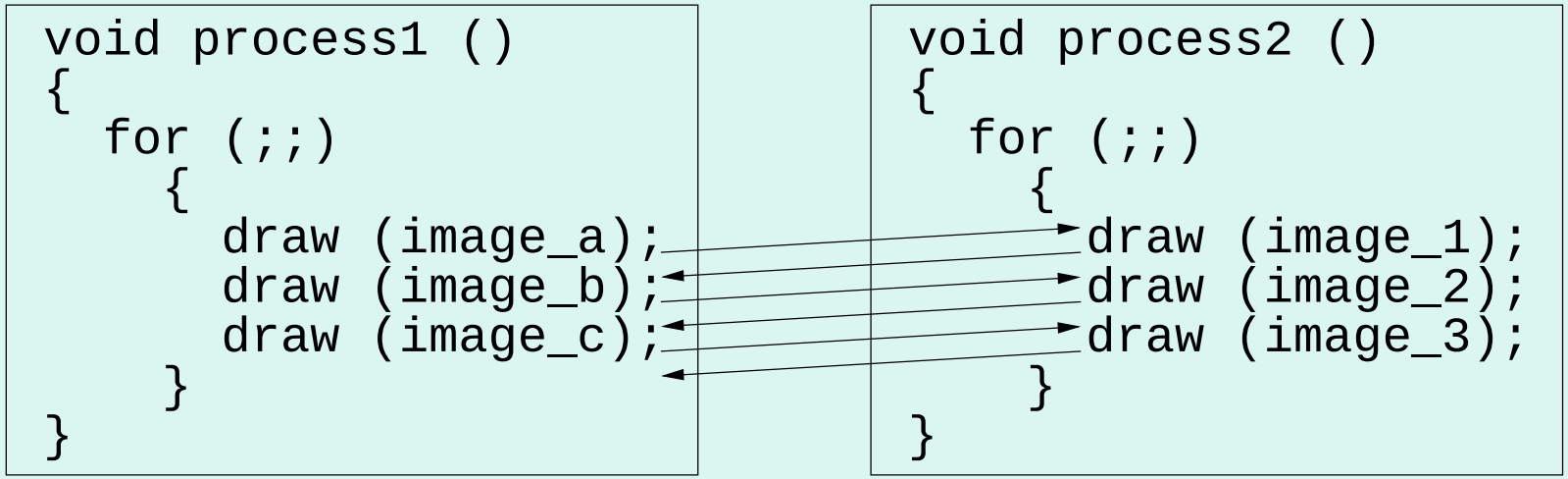
Exemple d'Animation (4)


```
void process1 ()
{
  for (;;)
    {
      draw (image_a);
      draw (image_b);
      draw (image_c);
    }
}
```

```
void process2 ()
{
  for (;;)
    {
      draw (image_1);
      draw (image_2);
      draw (image_3);
    }
}
```

- Ci-dessus on voit une organisation plus modulaire basée sur le concept de processus
- Il faut s'assurer que ces processus vont **progresser en unisson**

Exemple d'Animation (5)



- Sur un monoprocesseur, cela peut se faire en échangeant le contrôle d'un processus à l'autre après chaque opération "`draw`"
- Une **coroutine** c'est un processus qui décide à quel moment et à quelle autre coroutine céder le processeur

Implantation de Coroutines en C

- Le concept de coroutine peut être appliqué à n'**importe quel langage**
- À chaque processus est associé une structure dans laquelle est stocké l'état du processus (le "**Process Control Block**")
  - Ceci peut être difficile si la pile d'exécution fait partie de l'état (dans notre exemple ce n'est pas le cas)
- Il faut aussi un mécanisme pour **céder le processeur** à un autre processus: "`yield_to(process)`"
- Idéalement on écrirait:

```
1. void process1 ()
2. { for (;;)
3.     { draw (image_a); yield_to (process2);
4.       draw (image_b); yield_to (process2);
5.       draw (image_c); yield_to (process2);
6.     }
```

Implantation de Coroutines en C (code)

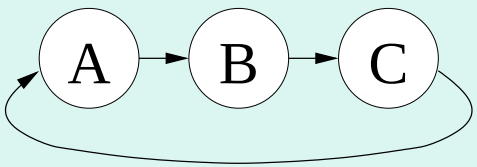

```
1.  struct pcb { void (*fn) (); int pc; };
2.  struct pcb p1, p2, *self; // process control blocks
3.
4.  #define yield_to(x) do { self = &x; return; } while (0)
5.
6.  void process1 () {
7.    for (;;) switch (self->pc++) {
8.                case 1: draw (image_a); yield_to (p2);
9.                case 2: draw (image_b); yield_to (p2);
10.               case 3: draw (image_c); yield_to (p2);
11.               case 4: self->pc = 1;
12. }}
13.
14. void process2 () {
15.   for (;;) switch (self->pc++) {
16.               case 1: draw (image_1); yield_to (p1);
17.               case 2: draw (image_2); yield_to (p1);
18.               case 3: self->pc = 1;
19. }}
20.
21. main () {
22.   p1.fn = process1;  p1.pc = 1;
23.   p2.fn = process2;  p2.pc = 1;
24.   self = &p1;
25.   for (;;) self->fn ();
```

Processus = Automate (1)

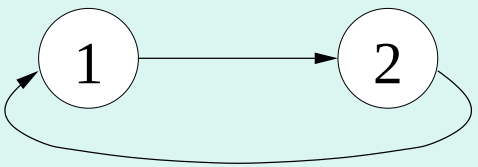
- Pour représenter un processus, on peut se servir d'un **automate** (fini ou infini) qui contient tous les **états** possible du processus
- Chaque état correspond à **une configuration des données** dans la mémoire du processus (précisément cela correspond aux variables globales, pile, et registres incluant le compteur ordinal)
- L'utilisation d'automates permet de mieux **comprendre** et **raisonner** sur le fonctionnement du processus
- Le nombre d'états de l'automate dépend du **degré de détail** dans une configuration (par exemple si on considère que le compteur ordinal passe par toutes les instructions machine, il y aura un grand nombre d'états même pour un programme simple)

Processus = Automate (2)

- Pour simplifier l'automate on fait abstraction des détails pour **faire ressortir l'essentiel**
- Exemple : les automates des processus de l'exemple d'animation ont 3 et 2 états respectivement si on considère les opérations "`draw`" **atomiques** (indivisibles)



process 1



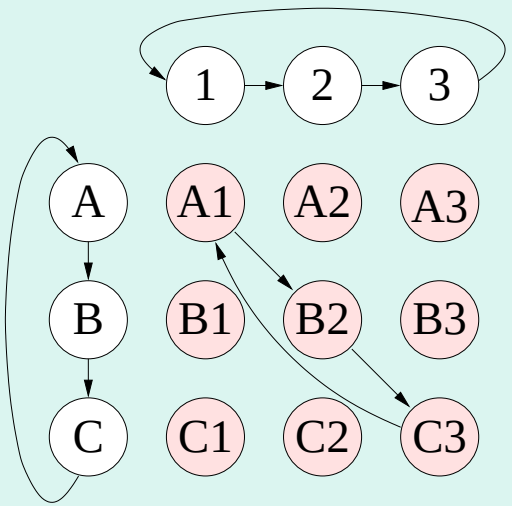
process 2

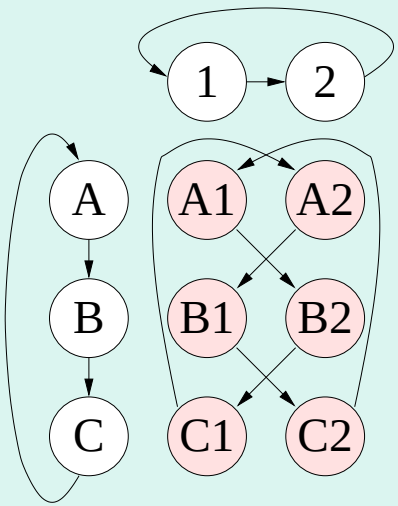
```
void process1 ()
{
  for (;;)
    {
     Ⓐdraw (image_a);
     Ⓑdraw (image_b);
     Ⓒdraw (image_c);
    }
}
```

```
void process2 ()
{
  for (;;)
    {
     ①draw (image_1);
     ②draw (image_2);
    }
  }
```

Processus = Automate (3)

- Il est toujours possible de combiner **un groupe de processus** en **un seul processus** en effectuant le **produit cartésien** des automates
- L'automate résultant a $N \times M$ états si on combine un processus à N états avec un processus à M états (il se peut que certains de ces états soient inaccessibles, et donc inutiles)
- Exemples:



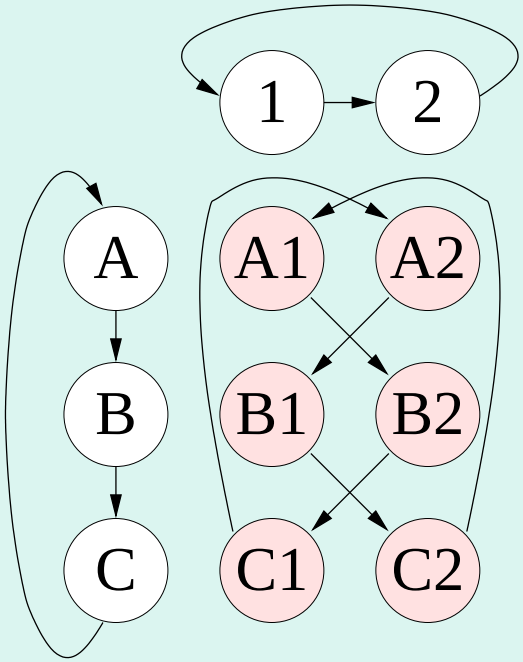


Processus = Automate (4)

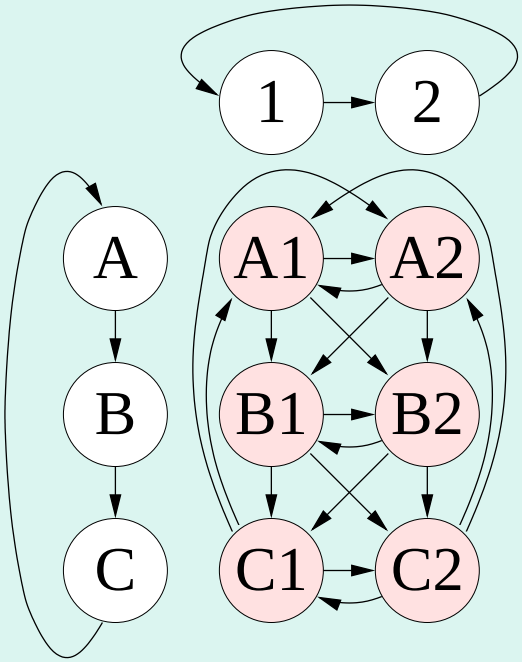
- Ce modèle basé sur les automates est utile pour **raisonner** sur le fonctionnement d'un groupe de processus (pour vérifier son bon fonctionnement) en particulier lorsque ces processus **coopèrent** et il faut **synchroniser** leurs actions

- Les transitions entre états dépendent des **contraintes de synchronisation** des processus
  - synchrone (chacun avance en même temps)
  - multiprocesseur (indépendance complète)
  - multiplexage du CPU (un seul à la fois)
  - coroutines (exécution alternée)

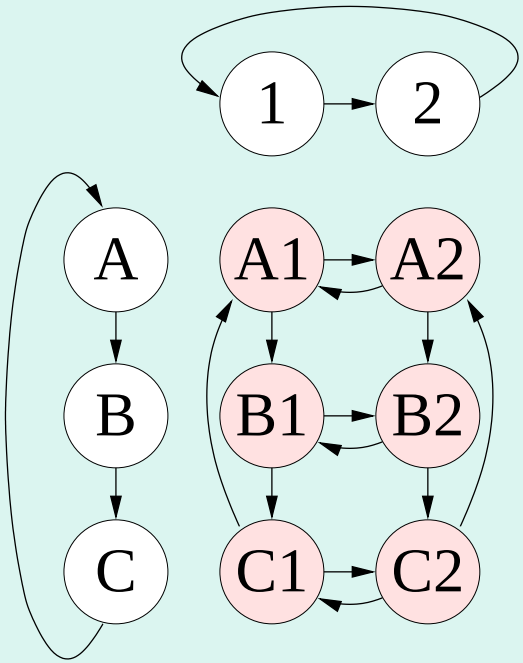
Processus = Automate (5)



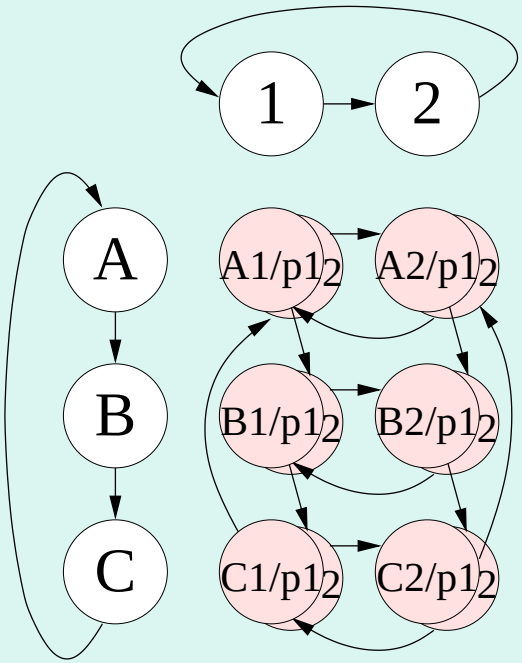
synchrone



multiprocesseur



multiplexage
CPU



coroutines

Le Modèle de Processus UNIX

- Les processus UNIX sont souvent appelés **processus lourds** (“**process**”) pour les distinguer des **processus légers** (“**threads**”) que nous verrons plus tard
- Les processus (lourds) sont **encapsulés**, c’est-à-dire que la mémoire principale accessible à un processus est **seulement accessible de ce processus**
  - Cela assure une certaine “**protection**” aux données du processus (il est impossible qu’un autre processus les corrompe)
  - Grâce à la mémoire virtuelle, chaque processus a l’illusion d’avoir accès à un **espace d’adressage logique privé** complet (2^{32} octets); deux processus peuvent avoir des données à la même adresse logique sans que cela cause un conflit

Échange d'Informations

- Puisque les processus ne peuvent pas changer ou examiner la mémoire principale des autres, il faut avoir d'autres façons pour échanger des informations entre processus

- Les méthodes de base sont
  - le système de fichier
  - les variables d'environnement
  - les arguments de ligne de commande
  - le statut de terminaison du processus

Shells UNIX (1)

- Les “shells” permettent à l’usager d’**exécuter des commandes** interactivement; fondamentalement ce sont des interprètes et ils acceptent un certain langage
- Chaque shell a ses particularités, mais en gros on retrouve deux grandes familles de shells avec des syntaxes et fonctionnalités similaires:
  - Famille “Bourne shell”: **sh**, **ksh**, **bash**, **zsh**
  - Famille “C shell”: **csh**, **tcsh**
- Exemple de différences syntaxiques

sh

```
% export x=DIRO
% for n in 5 abc $x; do
>   echo $x $n
> done
DIRO 5
DIRO abc
DIRO DIRO
```

csh

```
% setenv x DIRO
% foreach n (5 abc $x)
foreach? echo $x $n
foreach? end
DIRO 5
DIRO abc
DIRO DIRO
```

Shells UNIX (2)

- Certaines commandes sont “**builtin**”, c’est-à-dire que le shell les exécutent sans démarrer un autre programme (par exemple “`cd /`”, “`echo xxx`”, “`logout`”, “`pwd`”, “`for i in *.txt; do echo $i; done`”)

- Sinon le shell va démarrer l’exécution du programme **en tant que nouveau processus** (par exemple “`ls`”, “`gcc test.c`”, “`lpr rapport.txt`”, “`man ls`”)

```
% type cd
cd is a shell builtin
% type ls
ls is /usr/local/bin/ls
% /usr/local/bin/ls *.c*
tp1.c  tp2.c  tp3.c
% for i in *.c; do mv $i ${i/%.c/.cpp}; done
```

Variables d'Environnement (1)

- Chaque shell maintient un **environnement** qui contient un ensemble de **noms de variables** et les **valeurs associées** (chaîne de caractères)

- L'usager peut **consulter l'environnement**
  - commande "`env`"
  - notation "$*var*"

- L'usager peut **changer l'environnement**
  - commande "`export` *var*`=`*val*" (sh)
  - commande "`setenv` *var* *val*" (csh)
  - commande "`env` *var*`=`*val* *cmd*"

Variables d'Environnement (2)

- Manipulation de l'environnement à partir du shell:

```
% env
HOME=/u/feeley
PATH=/usr/local/bin:/usr/bin:/usr/bin/X11:/bin
PRINTER=hp3252
% echo ma maison est $HOME
ma maison est /u/feeley
% lpq
Printer: hp3252@papyrus  'HP 8000N publique local 325
 Queue: no printable jobs in queue
% export PRINTER=q3185   # ou "setenv PRINTER q3185"
% env
HOME=/u/feeley
PATH=/usr/local/bin:/usr/bin:/usr/bin/X11:/bin
PRINTER=q3185
% lpr rapport.txt
% env PRINTER=hp3252 lpr rapport.txt
% lpr -Php3252 rapport.txt
```

- Il est aussi possible de manipuler l'environnement à partir d'un programme avec `getenv`/`putenv`/etc.

Exemple (1)

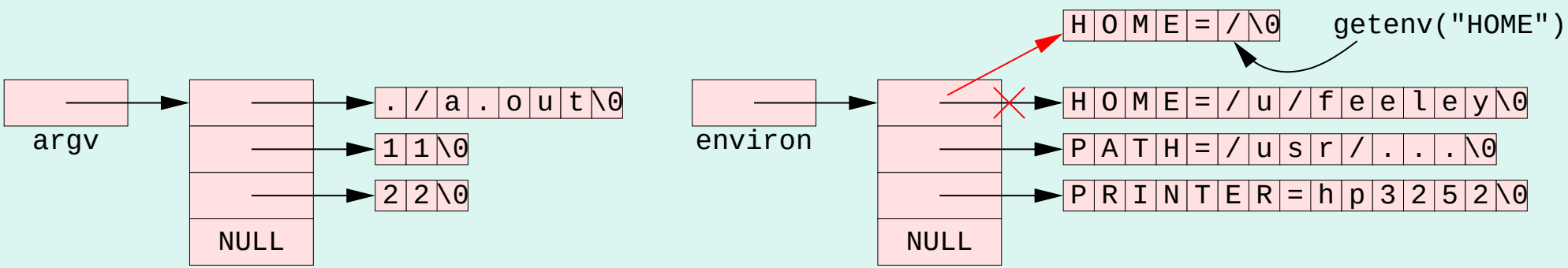

```
1.  #include <iostream>
2.  #include <stdlib.h> // pour getenv, putenv et atoi
3.
4.  extern char** environ;
5.
6.  int main (int argc, char* argv[])
7.  {
8.    cout << "argc = " << argc << "\n";
9.
10.   for (int i = 0; argv[i] != NULL; i++)
11.     cout << "argv[" << i << "] = " << argv[i] << "\n";
12.
13.   for (int i = 0; environ[i] != NULL; i++)
14.     cout << "environ[" << i << "] = " << environ[i] << "\
15.
16.   int somme = 0;
17.   for (int i = 1; i<argc; i++) somme += atoi (argv[i]);
18.   cout << "somme = " << somme << "\n";
19.
20.   putenv ("HOME=/");
21.
22.   cout << "HOME = " << getenv ("HOME") << "\n";
23.
24.   return 123; // statut de terminaison != 0  ==>  erreur
25. }
```

Exemple (2)

- Exécution:

```
1. % g++ test.cpp
2. % ./a.out 11 22
3. argc = 3
4. argv[0] = ./a.out
5. argv[1] = 11
6. argv[2] = 22
7. environ[0] = HOME=/u/feeley
8. environ[1] = PATH=/usr/local/bin:/usr/bin:/usr/bin/X11
9. environ[2] = PRINTER=hp3252
10. somme = 33
11. HOME = /
12. % echo $?
13. 123
```

- Structure de `argv` et `environ`:



Organisation en Mémoire (1)

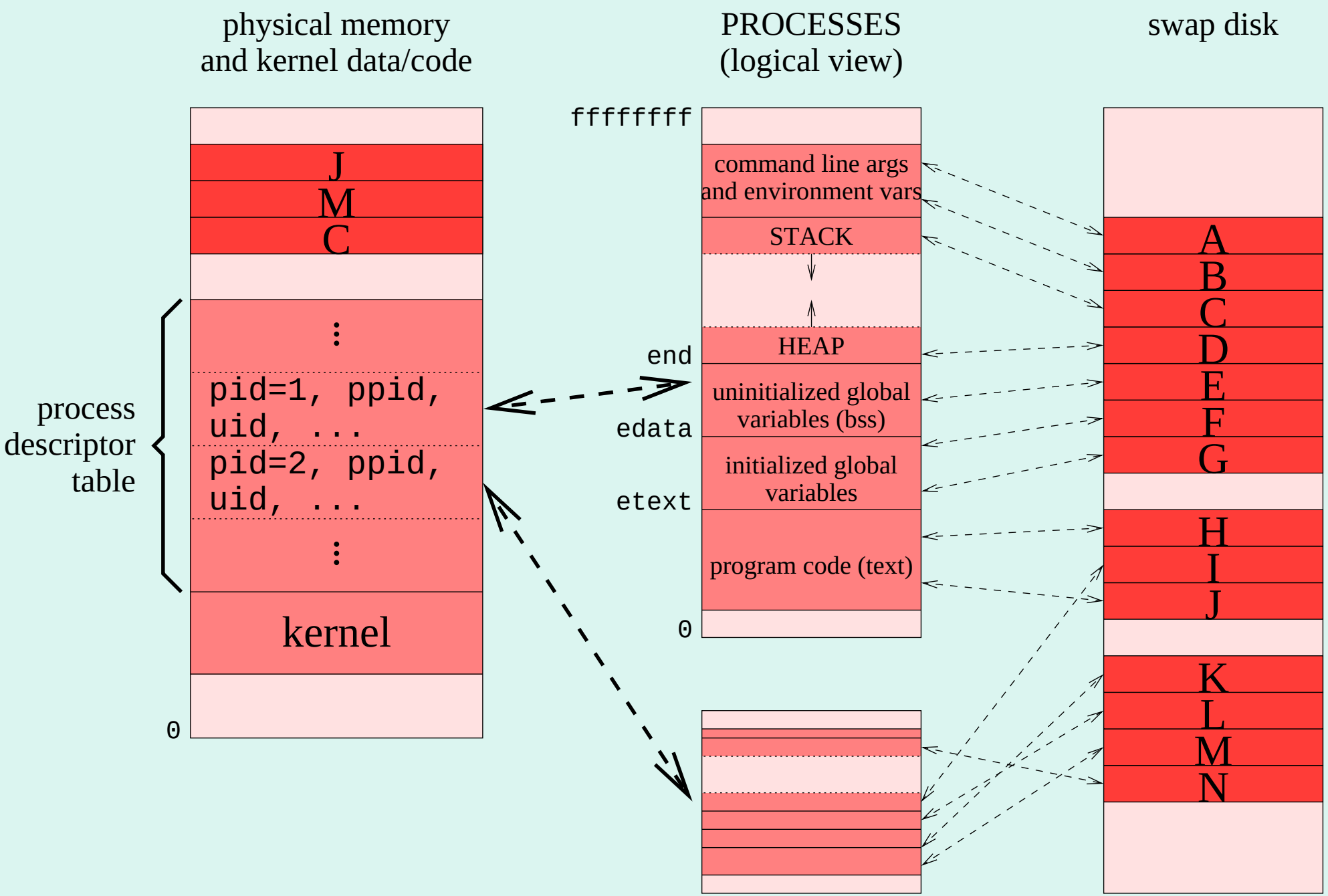
- Le kernel maintient une table (“**process descriptor table**”) qui contient pour chaque processus les attributs attachés à ce processus

- Attributs d’identification:
  - le `PID` (“**Process IDentifier**”): un entier qui identifie le processus (normalement entre 0 et 32767)
  - le `PPID` (“**Parent PID**”): le PID du processus qui a créé ce processus

Organisation en Mémoire (2)

- L'espace mémoire accessible au processus contient
  - le code machine exécutable ("**text**")
  - les variables globales initialisées (p.e. `int n = 5;`)
  - les variables globales non-initialisées (p.e. `int x;`) que le kernel initialise à zéro au chargement du programme ("**bss**" = "Block Started by Symbol")
  - la pile et le tas (pour les allocations dynamiques)
  - les variables d'environnement et les arguments de ligne de commande

Organisation en Mémoire (3)



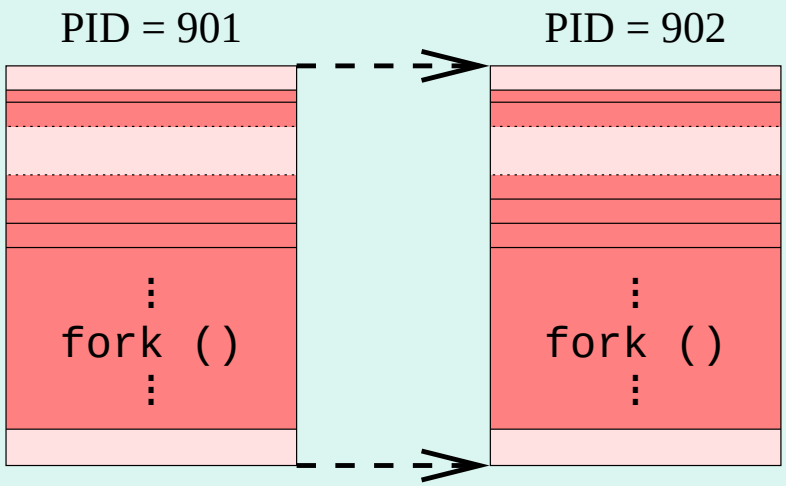
Création de Processus (1)

- L'unique façon de créer un nouveau processus c'est de faire appel à la fonction "`fork`" qui crée une **copie du processus présent** (l'**enfant**) et retourne un **PID**
- La copie est conforme, incluant les descripteurs de fichier et le contenu de la pile (donc l'enfant et le parent retournent tous deux de la fonction `fork`), sauf que
  - la fonction `fork` **retourne le PID de l'enfant** dans le parent et **retourne 0** dans l'enfant
  - les PIDs et PPIDs sont différents: PPID(enfant) = PID(parent)
  - les temps d'exécution (retournés par `times`) de l'enfant sont initialisés à 0
  - le **traitement des signaux** et les **verrous de fichiers** sont différents

Création de Processus (2)


```
1. #include <iostream>
2. #include <sys/types.h> // pour pid_t
3. #include <unistd.h> // pour getpid, getppid, fork et sleep
4.
5. int main (int argc, char* argv[])
6. { cout << "PID=" << getpid () << " PPID=" << getppid () << "\n";
7.
8.   pid_t p = fork ();
9.
10.  cout << "PID=" << getpid () << " PPID=" << getppid ()
11.       << " p=" << p << "\n";
12.
13.  sleep (1);
14.  return 0;
15. }
```

```
1. % ps
2.   PID TTY          TIME CMD
3.   817 pts/0    00:00:00 bash
4.   900 pts/0    00:00:00 ps
5. % echo $$
6. 817
7. % ./a.out
8. PID=901 PPID=817
9. PID=901 PPID=817 p=902
10. PID=902 PPID=901 p=0
```



- Pourquoi “`sleep (1)`”? Évite que p soit orphelin...

Exécution de Programmes (1)

- Pour démarrer l'exécution d'un programme il faut utiliser une des fonctions de la famille "`exec`", par exemple

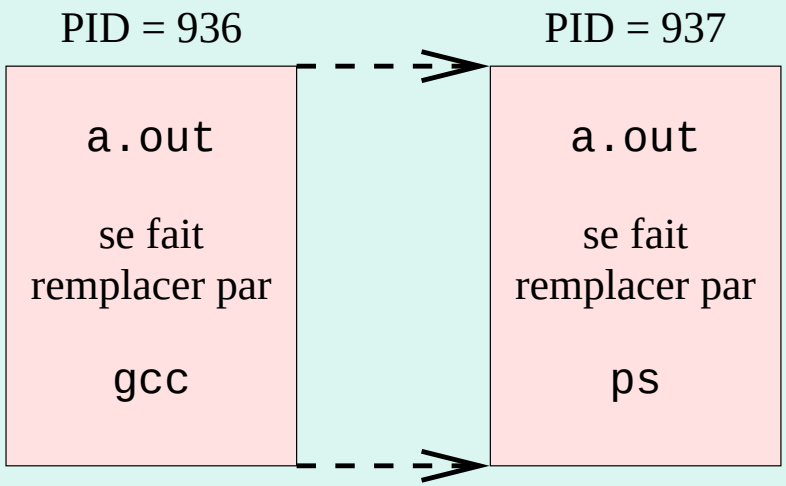
  `int execlp (char*`$filename$`, char*`arg_0`, char*`arg_1`, ...)`

- Les fonctions "`exec`" **ne créent pas un nouveau processus**; le processus qui exécute la fonction **remplace le contenu du processus par l'exécutable provenant du système de fichier**
- C'est le même concept que l'**appel terminal**
- On utilise souvent `exec` avec `fork`
- La fonction `CreateProcess` de Windows combine les opérations de création de processus et chargement du programme exécutable

Exécution de Programmes (2)


```
1. #include <unistd.h> // pour fork et execlp
2.
3. int main (int argc, char* argv[])
4. { cout << "PID=" << getpid () << "\n";
5.
6.   pid_t p = fork ();
7.
8.   if (p == 0) // enfant?
9.     { execlp ("ps", "ps", NULL);
10.      return 1;
11.    }
12.
13.  // parent
14.  execlp ("gcc", "gcc", "test.c", NULL);
15.  return 1;
16. }
```

```
1. % ./a.out
2. PID=936
3.   PID TTY          TIME CMD
4.   817 pts/0    00:00:00 bash
5.   936 pts/0    00:00:00 gcc
6.   937 pts/0    00:00:00 ps
7. % echo $?
8. 0
```



Exécution de Programmes (3)

- Famille des fonctions “exec”

```
int execlp (char*filename, char*arg0, ...)
int execl (char*pathname, char*arg0, ...)
int execle (char*pathname, char*arg0, ..., char**environ)
int execvp (char*filename, char**argv)
int execv (char*pathname, char**argv)
int execve (char*pathname, char**argv, char**environ)
```

“l/v” = liste/vecteur d’arguments
“p” = si aucun “/”, recherche dans PATH
“e” = valeur de l’environnement

- Valeur de retour = -1 si erreur

Exécution de Programmes (4)


```
1. #include <unistd.h> // pour execlp
2. #include <errno.h> // pour errno
3.
4. int main (int argc, char* argv[])
5. {
6.   int x;
7.
8.   x = execlp ("inconnu", "inconnu", NULL);
9.   cout << "x = " << x << "\n";
10.  cout << "errno = " << errno << "\n";
11.  cout << "strerror (errno) = " << strerror (errno) << "\n";
12.
13.  x = execlp ("pasexec", "pasexec", NULL);
14.  cout << "x = " << x << "\n";
15.  cout << "errno = " << errno << "\n";
16.  cout << "strerror (errno) = " << strerror (errno) << "\n";
17.
18.  return 1;
19. }
```

```
1. % ls -l pasexec
2. -rw-r--r--  1 feeley  feeley  4 14 Jan 12:25 pasexec
3. % ./a.out
4. x = -1
5. errno = 2
6. strerror (errno) = No such file or directory
7. x = -1
8. errno = 13
9. strerror (errno) = Permission denied
```

Exécution de Programmes (5)

- Fichier “`errno.h`”

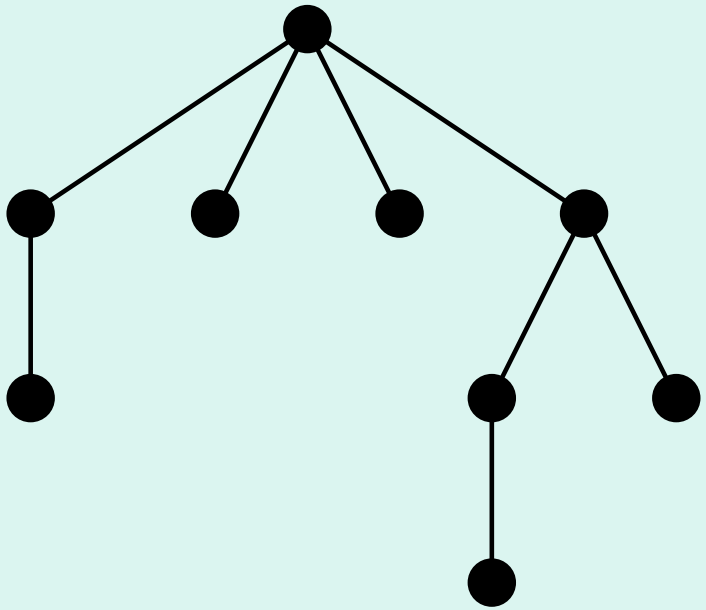
```
#define EPERM    1  /* Operation not permitted   */
#define ENOENT   2  /* No such file or directory */
#define ESRCH    3  /* No such process           */
#define EINTR    4  /* Interrupted system call   */
#define EIO      5  /* I/O error                 */
#define ENXIO    6  /* No such device or address */
#define E2BIG    7  /* Arg list too long         */
#define ENOEXEC  8  /* Exec format error         */
#define EBADF    9  /* Bad file number           */
#define ECHILD  10  /* No child processes        */
#define EAGAIN  11  /* Try again                 */
#define ENOMEM  12  /* Out of memory             */
#define EACCES  13  /* Permission denied         */
#define EFAULT  14  /* Bad address               */
...
```

- Exemple d’utilisation

```
execlp ("xxx", "xxx", NULL);
if (errno == ENOENT)
  cout << "programme xxx pas trouve\n";
else
  cout << "ca va vraiment mal!\n";
```

Exécution de Programmes (6)

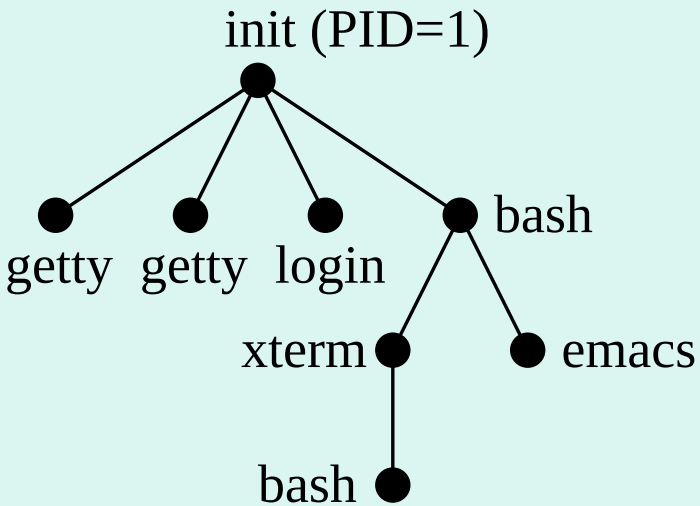
- La relation parent-enfant qui lie les processus donne une structure **hiérarchique** à l'ensemble des processus



- Tous les processus ont un parent, sauf celui à la racine
- Quel processus est à la racine?

Exécution de Programmes (7)

- La racine est le processus “`init`” (PID=1) qui est créé spécialement par le kernel au moment du boot



- `init` se base sur un fichier de configuration pour démarrer les processus de service (`/etc/inittab`)
- En particulier, des processus exécutant le programme `getty` et (par un `exec`) `login` qui permet à un usager de se brancher
- Si le mot de passe fourni par l’usager est conforme à la base de donnée `/etc/passwd`, `login` change l’environnement, répertoire, UID, etc. du processus et fait un `exec` du shell de l’usager

Fichier /etc/inittab typique


```
1. # Default runlevel. The runlevel is changed with "telinit N".
2. #   0 = halt,  1 = single user,  2 = multiuser (no NFS),
3. #   3 = full multiuser,  4 = unused,  5 = X11,  6 = reboot
4. id:5:initdefault:
5.
6. si::sysinit:/etc/rc.d/rc.sysinit     # System initialization.
7.
8. l0:0:wait:/etc/rc.d/rc 0
9. l1:1:wait:/etc/rc.d/rc 1
10. l2:2:wait:/etc/rc.d/rc 2
11. l3:3:wait:/etc/rc.d/rc 3
12. l4:4:wait:/etc/rc.d/rc 4
13. l5:5:wait:/etc/rc.d/rc 5
14. l6:6:wait:/etc/rc.d/rc 6
15.
16. ud::once:/sbin/update      # Things to run in every runlevel.
17.
18. 1:2345:respawn:/sbin/mingetty tty1       # Run getty.
19. 2:2345:respawn:/sbin/mingetty tty2
20. 3:2345:respawn:/sbin/mingetty tty3
21. 4:2345:respawn:/sbin/mingetty tty4
22. 5:2345:respawn:/sbin/mingetty tty5
23. 6:2345:respawn:/sbin/mingetty tty6
24.
25. x:5:respawn:/etc/X11/prefdm -nodaemon    # Run xdm.
```

Fichier `/etc/passwd` typique


```
1. root:PbIcvDfTbu90k:0:0:root:/root:/bin/bash
2. bin:xjXPZoh34RPms:1:1:bin:/bin:
3. daemon:4Pi9UcjqGq9NQ:2:2:daemon:/sbin:
4. feeley:oGLlRk0xGcFIA:353:472:Marc Feeley:/u/feeley:/bin/bash
```

- Le champ “mot de passe” est le mot de passe de l’usager, encrypté avec la fonction “`crypt`” (25 $\times$ DES)
- Cette pratique est insécure, car le fichier `/etc/passwd` est lisible par tous (pour des programmes comme `finger`)
- Si on connait le mot de passe encrypté M il est possible de faire une recherche “force brute” du mot de passe non-encrypté N pour lequel `crypt(`N`)=`M (il y a 2^{56} mots de passe encryptés possibles) Note: des approches plus subtiles existent!
- Les versions plus récentes de UNIX conservent le mot de passe dans le fichier `/etc/shadow` qui est lisible seulement par le “superusager” (`root`) et on trouve un marqueur spécial dans le champ “mot de passe” de `/etc/passwd`

Processus Orphelins

- Que se passe-t'il si le **parent d'un processus P meurt avant P**?
- Le processus P devient **orphelin**
- Le processus `init` (PID=1) adopte automatiquement les processus orphelins, de sorte que le parent de tout processus soit toujours un processus vivant
- Le processus `init` ne meurt jamais

Statut de Sortie et Fonction `exit`

- L'appel `exit` (x) termine un processus abruptement avec le statut de sortie x
- Un statut différent de 0 indique une **erreur**
- Pour permettre de faire un **nettoyage quelconque** avant de terminer le processus (p.e. eliminer des fichiers temporaires) il est possible d'ajouter des fonctions de nettoyage avec la fonction `atexit`

```
1. #include <stdlib.h> // pour atexit
2.
3. void f1 () { cout << "f1\n"; }
4. void f2 () { cout << "f2\n"; }
5.
6. int main (int argc, char* argv[])
7. { atexit (f1);
8.   atexit (f2);
9.   return 0; // aucune erreur, affiche f2 puis f1
10. }
```

- L'appel `_exit` (x) ignore les fonctions de nettoyage

Attente d'un Processus (1)

- Les fonctions `wait`, `waitpid` et `wait4` permettent à un processus d'**attendre la terminaison** d'un processus et d'**obtenir le statut de terminaison**
- Attendre un processus enfant (i.e. PPID(P) = moi) :

```
pid_t wait (int* status);
```

- Attendre un processus spécifique, ou un processus dans un certain groupe de processus :

```
pid_t waitpid (pid_t pid, int* status, int options);
pid_t wait4 (pid_t pid, int* status, int options, struct
rusage* rusage);
```

$pid > 0$: le processus pid
$pid = 0$: un processus enfant du même groupe
$pid = -1$: un processus enfant
$pid < 0$: un processus du groupe $-pid$

Attente d'un Processus (2)

- Cela est une forme simple de synchronisation
- Synchronisation de type "**fork-join**": on crée un sous-processus, on fait un travail concurrent, puis on attends que le sous-processus ait terminé

```
1.  pid_t p = fork ();
2.
3.  if (p == 0) // enfant?
4.    {
5.      f1 ();
6.      _exit (0);
7.    }
8.
9.  f2 ();
10.
11. int statut;
12. waitpid (p, &statut, 0);
```

Attente d'un Processus (3)

- Le "statut" est accédé avec ces macros de `<sys/wait.h>`:
  - `WIFEXITED(statut)` : vrai si terminé avec un appel à `exit`
  - `WIFSIGNALED(statut)` : vrai si terminé avec un signal
  - `WIFSTOPPED(statut)` : vrai si suspendu (ctrl-Z)
  - `WEXITSTATUS(statut)` : code de terminaison passé à `exit`
  - `WTERMSIG(statut)` : numéro du signal qui a causé la terminaison

Processus Zombie

- Que se passe-t'il si un processus P meurt mais qu'**aucun processus n'a encore obtenu son statut de terminaison** avec un appel à `wait` ou `waitpid`?
- Le processus P est un **zombie**
- Le problème c'est que le kernel ne peut savoir si plus tard un processus voudra obtenir le statut de P
- Le descripteur de processus de P **doit être préservé** par le kernel jusqu'à ce que son statut soit consulté avec un appel à `wait` ou `waitpid`
- Un zombie qui devient orphelin (i.e. son parent meurt) est éliminé de la table de descripteurs du kernel, car `init` exécute `wait` dans une boucle sans fin

Processus Légers

- L'encapsulation offerte par les processus lourds est gênante lorsque les processus doivent **partager** et **échanger** des données
- Avec les processus légers ("threads"), **la mémoire est commune** à tous les processus légers
- Un processus peut écrire des données en mémoire et un autre processus peut immédiatement les lire
- Normalement, à chaque processus lourd est attaché un ensemble de processus légers; ceux-ci doivent respecter l'encapsulation du processus lourd
- À la création d'un processus lourd, il y a un processus léger qui est créé pour exécuter le code de ce processus lourd (thread "**primordial**")
- Les risques de corruption sont mineurs

POSIX Threads (1)

- Les processus (légers) peuvent être **intégrés au langage de programmation** ou bien être **une bibliothèque**
- Sous UNIX il y a plusieurs bibliothèques de threads, “**POSIX Threads**” est une des plus portables

POSIX Threads (2)

- Pour **créer et démarrer un nouveau thread** on utilise

```
int pthread_create (pthread_t* id,
                    pthread_attr_t* attr,
                    void* (*fn) (void*),
                    void* arg)
```

- *id* = descripteur de thread
- *attr* = attributs de création, p.e. priorité (`NULL` = défauts)
- *fn* = fonction que le thread exécutera
- *arg* = paramètre qui sera passé à *fn*
- résultat = code d'erreur (0 = aucune erreur)

POSIX Threads (3)

- Pour **attendre la terminaison d'un thread** on utilise

  `int pthread_join (pthread_t` *id*`, void**` *résultat*`)`

- *résultat* = valeur retournée par la fonction d'exécution du thread (ou bien celle passée à `pthread_exit`)

  `void pthread_exit (void*` *résultat*`)`

Exemple 1


```
1.  #include <pthread.h>
2.
3.  void* processus (void* param)
4.  {
5.    int i = (int)param;
6.    for (int j = 0; j<10000000; j++) ;
7.    cout << i << "\n";
8.    return (void*)(i*i);
9.  }
10.
11. int main ()
12. { pthread_t tid[5];
13.   void* resultats[5];
14.
15.   for (int i = 0; i<5; i++)
16.     pthread_create (&tid[i], NULL, processus, (void*)i);
17.
18.   for (int i = 0; i<5; i++)
19.     pthread_join (tid[i], &resultats[i]);
20.
21.   for (int i = 0; i<5; i++)
22.     cout << "resultat du processus " << i << " = "
23.          << (int)resultats[i] << "\n";
24.
25.   return 0;
```

Exemple 1 (cont.)

- Compilation et exécution du programme

```
% g++ exemple.cpp -lpthread
% ./a.out
1
2
0
4
3
resultat du processus 0 = 0
resultat du processus 1 = 1
resultat du processus 2 = 4
resultat du processus 3 = 9
resultat du processus 4 = 16
```

- La sortie en désordre est dû au **phénomène de “course” entre processus**
- Des processus sont en **condition de course** (“race condition”) s’ils tentent d’utiliser une ressource partagée (la console dans l’exemple) **sans synchroniser** l’accès à cette ressource avec les autres processus

Exemple 2


```
1.  #include <pthread.h>
2.
3.  int compteur = 0;
4.
5.  void* processus (void* param)
6.  {
7.    for (int i = 0; i<10000000; i++)
8.      compteur++;
9.    return NULL;
10. }
11.
12. int main ()
13. { pthread_t tid[5];
14.   void* resultat;
15.
16.   for (int i = 0; i<5; i++)
17.     pthread_create (&tid[i], NULL, processus, NULL);
18.
19.   for (int i = 0; i<5; i++)
20.     pthread_join (tid[i], &resultat);
21.
22.   cout << "compteur = " << compteur << "\n";
23.
24.   return 0;
25. }
```

Exemple 2 (cont.)

- Compilation et exécution du programme

```
% g++ exemple.cpp -lpthread
% ./a.out
compteur = 31074212
% ./a.out
compteur = 34410197
% ./a.out
compteur = 15885877
% ./a.out
compteur = 22522139
```

- Comment expliquer que le résultat n’est pas 50000000?
- Le problème c’est la course pour l’incrémentation du compteur

Modèle de concurrence (1)

- Le modèle suivant d'exécution concurrente est utile:
  1. L'exécution de chaque processus est décomposé en **opérations élémentaires** indivisibles
  2. On suppose que deux opérations O_1 et O_2 de deux processus s'exécutent dans l'ordre O_1-O_2 ou O_2-O_1 (**pas en même temps**)
  3. L'exécution globale du programme est une **permutation de toutes les opérations** qui respecte l'ordre séquentiel imposé par chaque processus
- P_1=AB et P_2=xyz, 10 exécutions possibles:

```
ABxyz   AxByz   AxyBz   AxyzB   xAByz
xAyBz   xAyzB   xyABz   xyAzB   xyzAB
```

- Pour 2 processus avec n et m opérations, il y a un nombre total d'exécutions possibles égal à

$$(n+m)!/(n! * m!)$$

Modèle de concurrence (2)

- L'énoncé "`compteur++`" n'est **pas atomique**; il se décompose en 3 opérations élémentaires

  1. `temp = compteur` (Lecture)
  2. `temp = temp+1` (Addition)
  3. `compteur = temp` (Écriture)

- Si P_1=`LAE` et P_2=`lae` il y a 20 exécutions possibles:

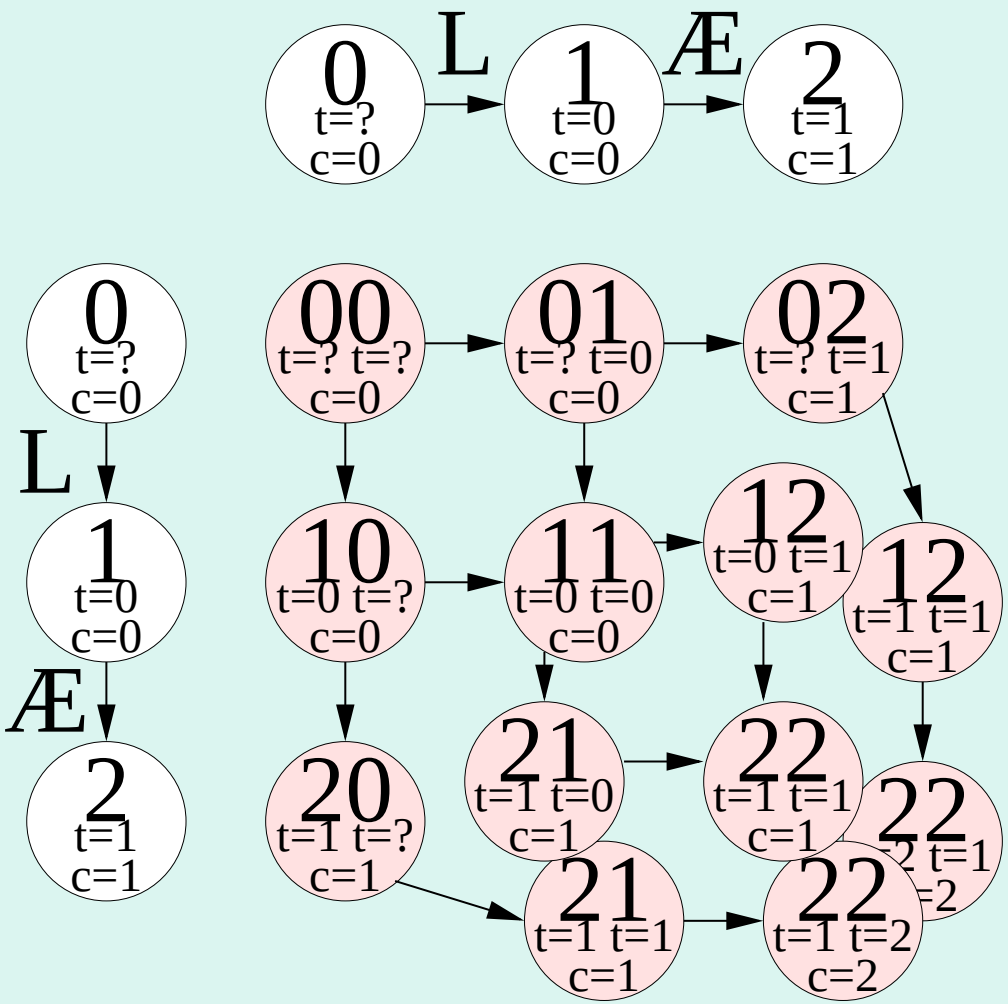
```
LAElae   compteur=2
laeLAE   compteur=2
LAlEae   compteur=1
LAlaeE   compteur=1 (idem pour autres exécutions)
```

- Pour simplifier l'analyse de l'interaction de processus, **seules les opérations qui changent l'état observable des autres processus (ou qui observent un tel état) ont besoin d'être considérées**

- `temp = temp+1` pas observable d'un autre processus

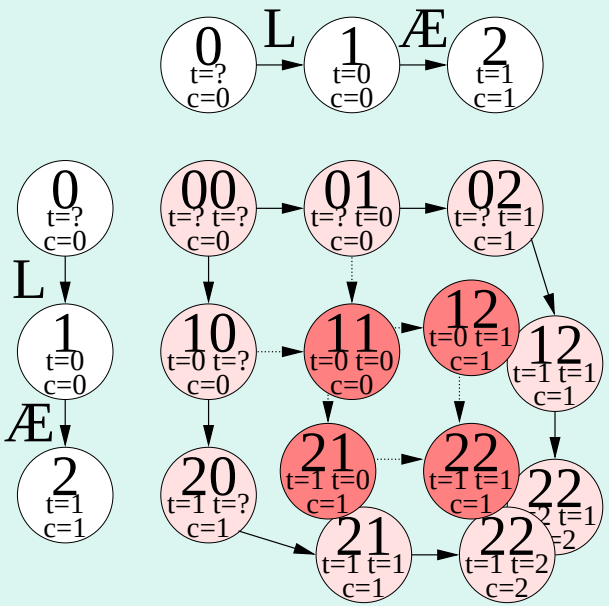
Modèle de concurrence (3)

- Si P_1=LÆ et P_2=læ il y a seulement 6 cas possibles:
  LÆlæ læLÆ LlÆæ llæÆ llÆæ LlæÆ
- Une analyse détaillée est possible avec les automates de processus



Modèle de concurrence (4)

- Pour que `compteur` soit mis-à-jour correctement, il faut s'assurer que l'exécution des opérations LAE par un processus soit **indivisible du point de vue des autres processus**



- Une **section critique** c'est une section de code qui ne peut être en exécution que par un nombre de processus limité (normalement un seul, i.e. **exclusion mutuelle**)

Synchronisation (1)

- Sur un **monoprocesseur** l'entrelacement des opérations de divers processus provient du multiplexage du CPU qui est contrôlé par une **interruption périodique** (p.e. "timer" 100 Hz)
- Donc, pour implanter une **exclusion mutuelle** il suffit de s'assurer que le processeur **ne sera pas interrompu pendant la section critique**

```
void incrémenter ()
{ disable_interrupts (); // instruction ‘‘cli’’
  compteur++;            // section critique
  enable_interrupts ();  // instruction ‘‘sti’’
}
```

- Cela n'est pas conseillé pour les sections critiques de **longue durée** car cela bloque l'exécution de **tous les autre processus** (c'est-à-dire qu'une **dépendance artificielle** est introduite)

Synchronisation (2)

- Il faut faire attention car certaines opérations qui semblent atomiques au niveau du langage de programmation ne le sont peut être **pas au niveau machine**

- Exemple 1:

```
long long n; // entier de 64 bits

void process1 () { n = 1; }  // ‘‘n.lo32 = 1;
                             //   n.hi32 = 0;’’

void process2 () { n = -1; } // ‘‘n.lo32 = -1;
                             //   n.hi32 = -1;’’
```

  4 résultats possibles!

- Exemple 2:

```
void process1 () { cout << 123; }
void process2 () { cout << 456; }
```

Synchronisation (3)

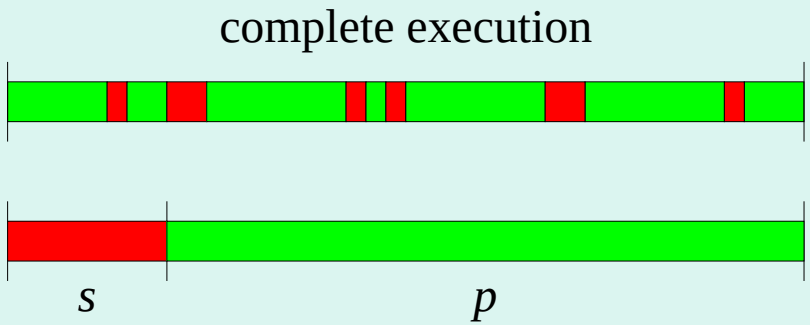
- Toute section critique a un **impact négatif sur le degré de parallélisme** du système car les processus impliqués doivent exécuter la section critique **séquentiellement**

- La section critique devient un **goulôt d'étranglement** ("bottleneck") qui limite le parallélisme du programme

Synchronisation (4)

- Soit un programme dont l'exécution est composé de parties strictement **séquentielles** et le reste est infiniment **parallèle**

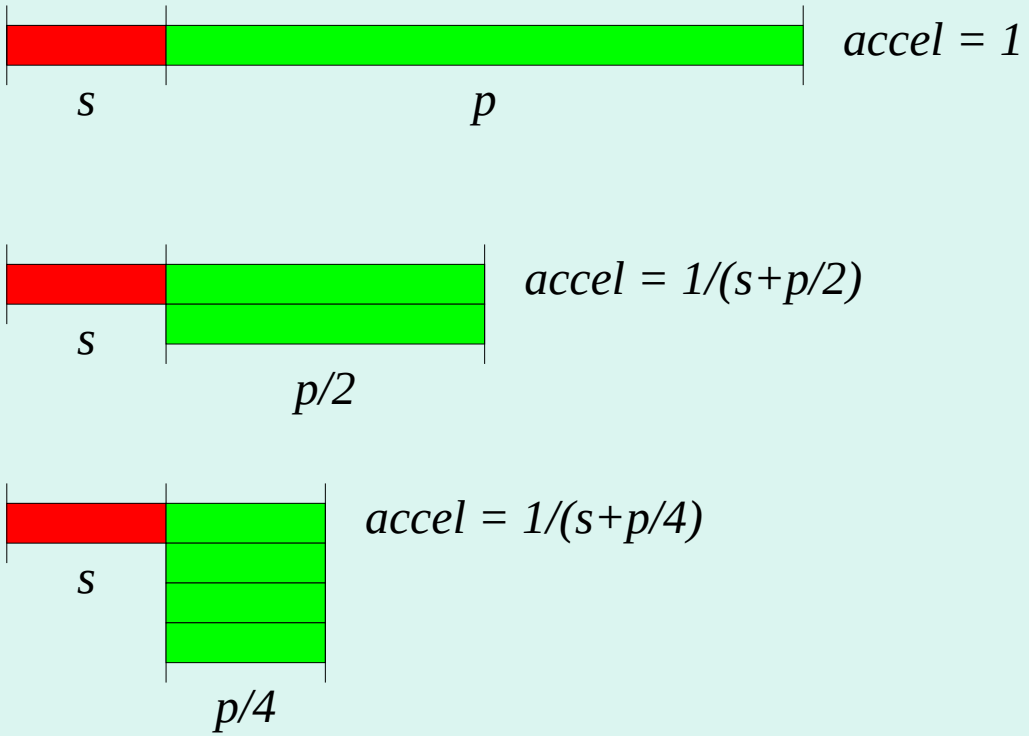
- Soit s la proportion du travail total qui est séquentiel et p la proportion du travail total qui est parallèle (donc $s + p = 1$)



s = sequential
p = parallel

Synchronisation (5)

- Déf: l'accélération c'est le facteur d'amélioration de la vitesse: $accel = \frac{T_{\text{orig}}}{T_{\text{nouveau}}}$
- **Loi de Amdhal**: l'accélération dû au parallélisme est bornée par $1/s$



Synchronisation (6)

- La loi de Amdhal s'applique aussi dans la **programmation système**
  - s = routines du kernel et bibliothèques (que l'usager ne peut modifier)
  - p = programme usager (que l'usager peut améliorer)

- Les routines du kernel placent une borne supérieure sur l'amélioration de performance qu'on peut atteindre en modifiant le programme

Synchronisation (7)

- **Sémaphore**: mécanisme général qui permet d'implanter des sections critiques

- Une sémaphore S c'est une variable entière qui supporte **2 opérations indivisibles**

  1. `acquérir(S)`: attendre S> 0 puis décrémenter S
  2. `céder(S)`: incrémenter S

- Pendant l'attente, d'autre processus peuvent exécuter

Synchronisation (8)

- On associe une sémaphore à la section critique et on l'initialise au nombre maximal de processus permis dans la section critique (donc normalement S=1, **sémaphore binaire ou "mutex"**)

```
S = 1;  // initialisation globale

void incrémenter ()
{ acquérir (S);
  compteur++;  // section critique
  céder (S);
}
```

Synchronisation (9)

- Cette approche n'est pas équivalente (même si on suppose que les lectures et écritures sont atomiques):

```
S = 1;  // initialisation globale

void incrémenter ()
{
  while (S == 0) ;
  S = 0;
  compteur++;  // section critique
  S = 1;
}
```

- Il se peut que plus d'un processus entre dans la section critique
- Le problème c'est que le test "`S==0`" et l'affectation "`S=0`" ne sont pas **atomiques** (il y a un risque qu'entre le test et l'affectation l'autre processus execute son propre test "`S==0`")

Synchronisation (10)

- Sur un monoprocesseur ceci implante correctement l'exclusion mutuelle:

```
S = 1;  // initialisation globale

void incrémenter ()
{ disable_interrupts ();
  while (S == 0)
    { enable_interrupts ();
      disable_interrupts ();
    }
  S = 0;
  enable_interrupts ();

  compteur++;  // section critique

  S = 1;  // supposé atomique
}
```

- Un "**spinlock**" est une sémaphore binaire implantée avec une telle **attente active**
- C'est acceptable si l'**attente est courte**

Synchronisation (11)

- Certains CPUs offrent des opérations atomiques (sur mono ou multiprocesseur) qui facilitent l'implantation des sections critiques, par exemple

```
int test_and_set (int& x); // retourne ancienne
                           // valeur de x

void swap (int& x, int& y); // échange x et y
```

- Avec `test_and_set` et une affectation atomique:

```
S = 0;  // initialisation globale, note: inversion

void incrémenter ()
{
  while (test_and_set (S) == 1) ;

  compteur++;  // section critique

  S = 0;
}
```

- Cette implantation comporte toujours une attente active

Synchronisation (12)

- Pour éviter l'attente active, le kernel peut **suspendre l'exécution du processus** (c'est-à-dire le retirer de la liste des processus exécutables) et réactiver ce processus lorsque la sémaphore se libère

- Chaque sémaphore contient la **liste des processus en attente** sur cette sémaphore

```
typedef struct
  {
    int compte;
    liste_de_processus en_attente;
  } sémaphore;
```

Synchronisation (13)


```
1.  void acquérir (sémaphore& S)
2.  {
3.    disable_interrupts ();
4.    S.compte--;
5.    if (S.compte < 0)
6.      suspendre_processus_présent_sur (S.en_attente);
7.    enable_interrupts ();
8.  }
9.
10. void céder (sémaphore& S)
11. {
12.   disable_interrupts ();
13.   S.compte++;
14.   if (S.compte <= 0)
15.     {
16.       processus p = retirer_processus (S.en_attente);
17.       réactiver (p); // ajoute ‘‘p’’ à la liste
18.     }                // des processus exécutables
19.   enable_interrupts ();
20. }
```

- Si `S.compte`< 0, `-S.compte` est le nombre de processus en attente

Synchronisation (14)

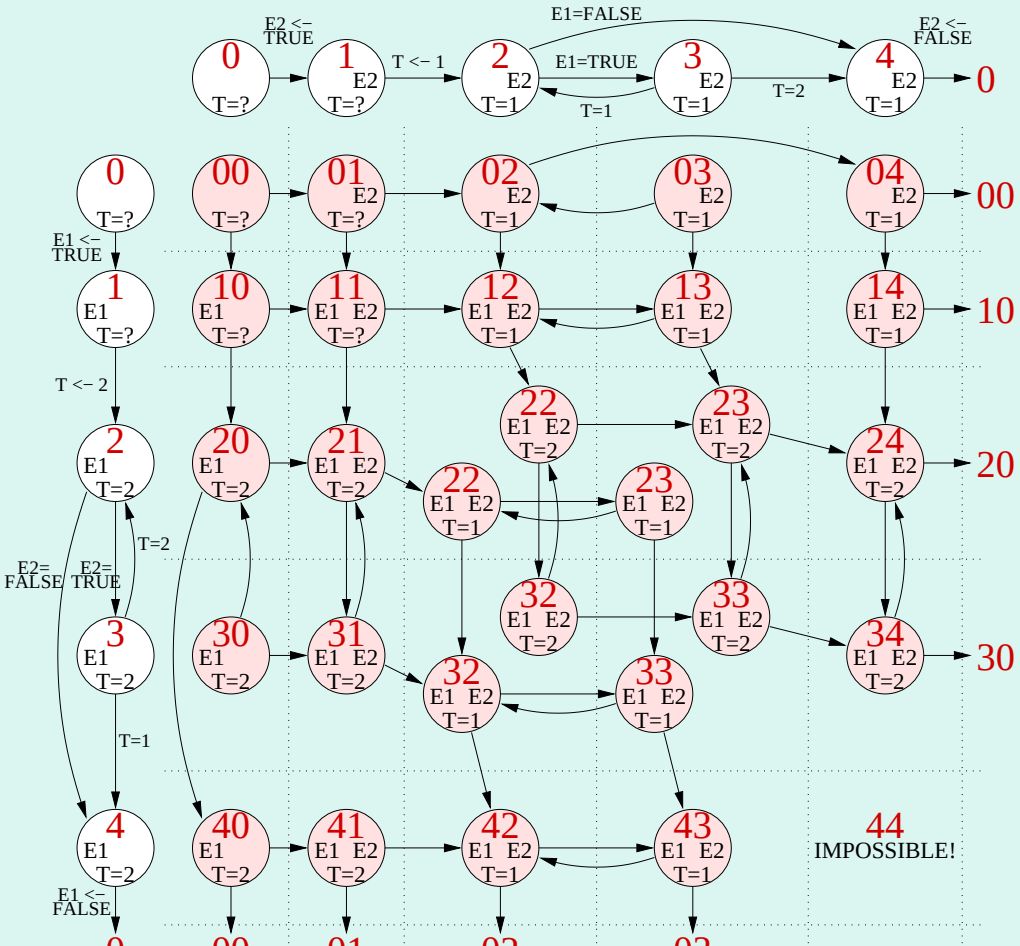
- Est-ce possible d'implanter une exclusion mutuelle entre 2 processus (P1 et P2) sur une machine où **seules les lectures et écritures sont des opérations atomiques**?
- Oui! Il existe plusieurs algorithmes: Dijkstra, Dekker, Peterson, ...
- Algorithme de Peterson (1981): 3 variables partagées

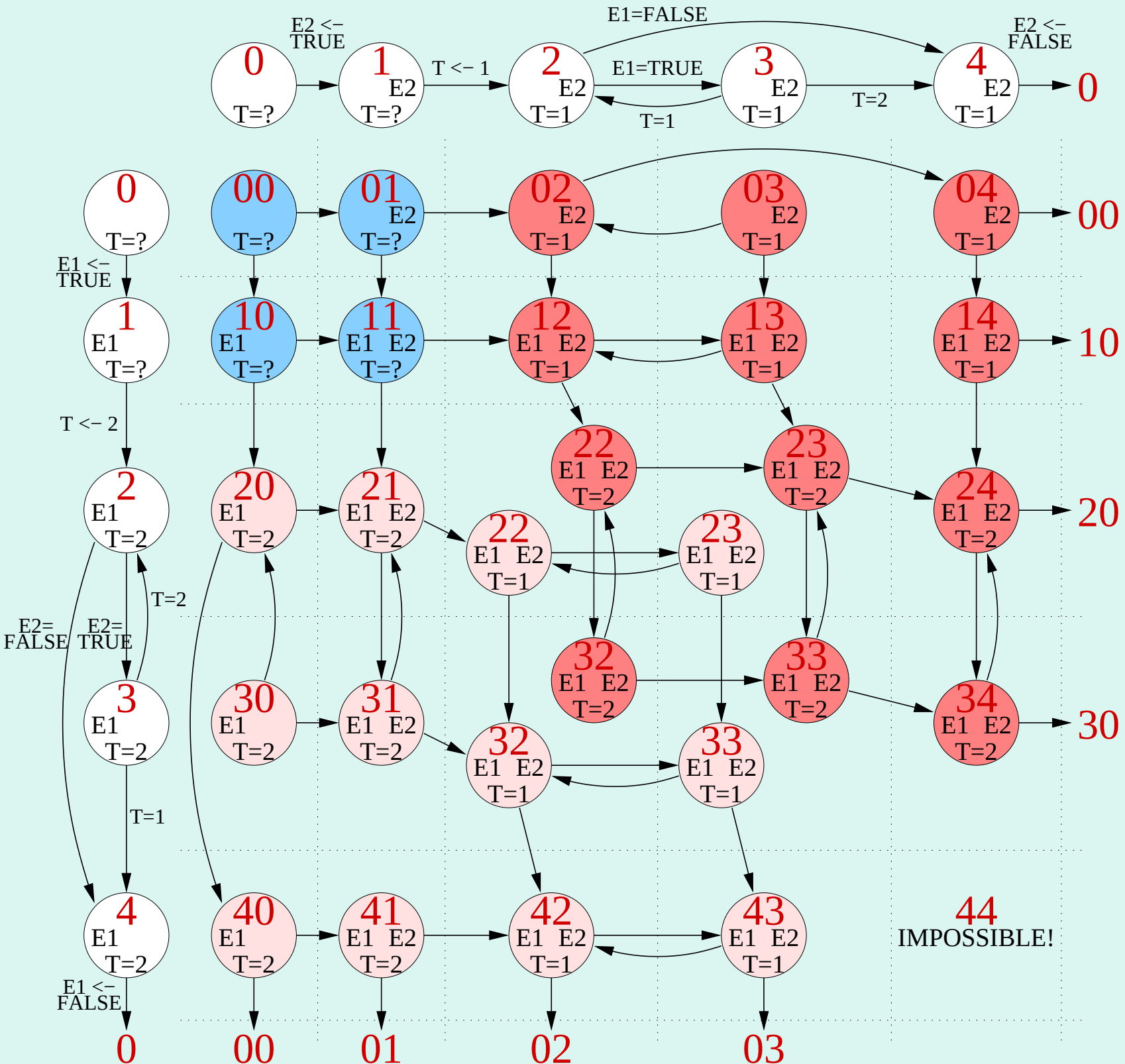
```
E1 : booléen   indique que P1 désire entrer
E2 : booléen   indique que P2 désire entrer
T  : 1 ou 2    indique quel processus a la priorité
```

```
   processus P1                    processus P2

E1 <- VRAI                      E2 <- VRAI
T <- 2                          T <- 1
attendre tant que E2 et T=2     attendre tant que E1 et T=1
<<<SECTION CRITIQUE>>>          <<<SECTION CRITIQUE>>>
E1 <- FAUX                      E2 <- FAUX
```

Synchronisation (15)

- Pour analyser correctement il faut considérer que les processus tentent d'**exécuter la section critique répététivement**
- La combinaison des processus donne l'automate:




0
T=?
E2 <–
TRUE
1
E2
T=?
T <– 1
2
E2
T=1
E1=FALSE
E1=TRUE
3
E2
T=1
T=1
T=2
4
E2
T=1
E2 <–
FALSE
0
0
T=?
E1 <–
TRUE
1
E1
T=?
T <– 2
2
E1
T=2
E2=
FALSE
E2=
TRUE
T=2
3
E1
T=2
T=1
4
E1
T=2
E1 <–
FALSE
0
00
T=?
01
E2
T=?
02
E2
T=1
03
E2
T=1
04
E2
T=1
00
10
E1
T=?
11
E1 E2
T=?
12
E1 E2
T=1
13
E1 E2
T=1
14
E1 E2
T=1
10
20
E1
T=2
21
E1 E2
T=2
22
E1 E2
T=2
23
E1 E2
T=2
24
E1 E2
T=2
20
22
E1 E2
T=1
23
E1 E2
T=1
30
E1
T=2
31
E1 E2
T=2
32
E1 E2
T=2
33
E1 E2
T=2
34
E1 E2
T=2
30
32
E1 E2
T=1
33
E1 E2
T=1
40
E1
T=2
41
E1 E2
T=2
42
E1 E2
T=1
43
E1 E2
T=1
44
IMPOSSIBLE!
00
01
02
03

Synchronisation (17)

- Cet algorithme **possède trois qualités** importantes (désirables de tout mécanisme de synchronisation):

1. **Exclusion mutuelle**: il ne peut pas y avoir plus d'un processus qui exécute dans la section critique (Algo de Peterson: absence de l'état 44)
2. **Progrès global**: parmis les processus qui désirent entrer dans la section critique, au moins un y entrera (Algo de Peterson: absence de cycle qui ne passe pas par l'état 4, en laissant la chance aux 2 processus d'exécuter)
3. **Attente bornée / Équité**: si un processus désire entrer dans la section critique son temps d'attente est borné (Algo de Peterson: alternance de l'état 4 par les deux processus une fois rendu à l'état 22)

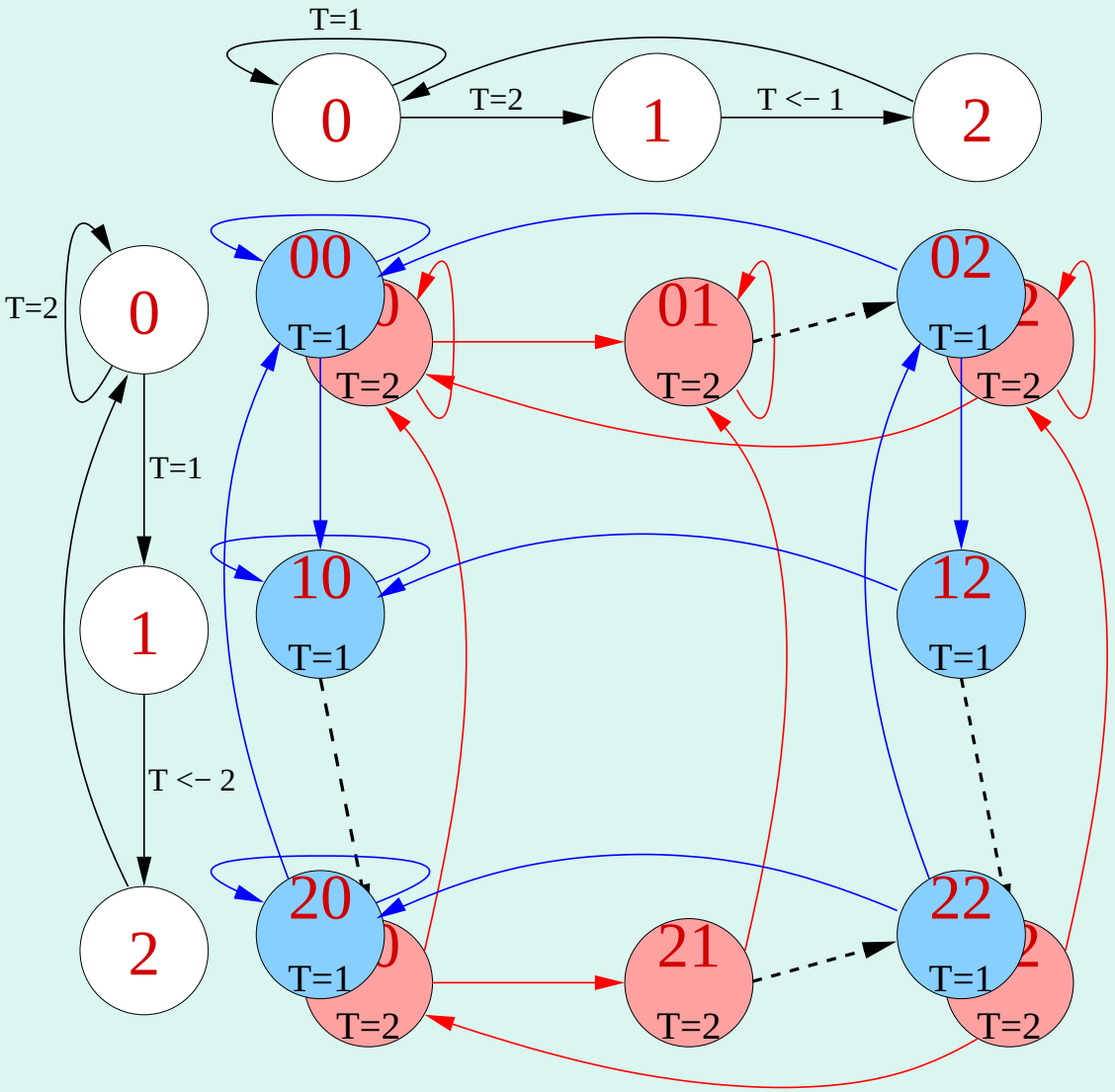
Synchronisation (18)

- Analysons un algorithme simplifié:

```
  T  : 1 ou 2    indique si c'est le tour de P1 ou P2

     processus P1                      processus P2

attendre tant que T=2          attendre tant que T=1
<<<SECTION CRITIQUE>>>         <<<SECTION CRITIQUE>>>
T <- 2                         T <- 1
```

- Est-ce que cet algorithme nous donne
  1. **Exclusion mutuelle?**
  2. **Progrès global?**
  3. **Attente bornée?**

Synchronisation (19)



- Donc: **exclusion mutuelle** et **progrès global**
- **Attente non-bornée** (si l'autre P reste dans l'état 2)

Synchronisation (20)

- Analysons un autre algorithme simplifié:

```
E1 : booléen   indique que P1 désire entrer
E2 : booléen   indique que P2 désire entrer

 processus P1                    processus P2

E1 <- VRAI                      E2 <- VRAI
attendre tant que E2            attendre tant que E1
<<<SECTION CRITIQUE>>>          <<<SECTION CRITIQUE>>>
E1 <- FAUX                      E2 <- FAUX
```

- Est-ce que cet algorithme nous donne

1. **Exclusion mutuelle? Oui**
2. **Progrès global? Non**

Variables de Condition (1)

- Les “**variables de condition**” permettent des synchronisations plus générales que les sémaphores
- Une variable de condition est un objet qui correspond à un ensemble de processus en attente qu’une **certaine condition (définie par le programmeur) soit vraie de l’état du programme** (existence d’un objet particulier dans une structure de donnée, occurrence d’un événnement particulier, etc)
- C’est utile pour qu’un processus puisse **être informé par d’autres processus d’un changement d’état** qui l’intéresse **sans avoir à faire une attente active**

Variables de Condition (2)

- Exemple : supposons une méthode “`dépot(`n`)`” qui ajoute n au solde d’un compte banquaire et une méthode “`retrait(`n`)`” qui est seulement exécutable si n n’est pas plus grand que le solde du compte (sinon il faut attendre que le solde soit suffisant)

```
dépot(n):    solde <- solde + n

retrait(n):  attendre jusqu'à ce que (solde >= n)
             solde <- solde - n
```

- Cela ne fonctionne pas tel quel car le test et la modification de `solde` ne sont pas atomiques
- Ici la condition est “`solde>=n`”
- On aimerait suspendre l’exécution du processus lors d’un retrait où cette condition est fausse et **réveiller le processus lorsque cette condition devient vraie** pour compléter le retrait

Variables de Condition (3)

- Pour obtenir l'exclusion mutuelle on peut utiliser une sémaphore binaire S:

```
dépot(n):
  acquérir(S)
  solde <- solde + n
  céder(S)

retrait(n):
  boucle:
    acquérir(S)
    si (solde >= n)
      solde <- solde - n
      céder(S)
    sinon
      céder(S)
      goto boucle
```

- Malheureusement il y a une **attente active** qui gaspille du temps CPU

Variables de Condition (4)

- La suspension des processus se fait avec une variable de condition “C” attachée à la condition `solde>=n`
- Tout processus qui **change possiblement la valeur de la condition** doit réveiller les processus en attente, pour qu’ils puissent vérifier la condition

Variables de Condition (5)

- L'opération `wait(C,S)` suspends le processus présent sur la variable de condition `C` et cède la sémaphore `S` **atomiquement**; au réveil du processus celui-ci fera automatiquement l'acquisition de `S`
- L'opération `signal(C)` réveille un des processus qui est en attente dans la variable de condition `C`
- L'opération `broadcast(C)` réveille tous les processus en attente dans la variable de condition `C`
- On doit toujours utiliser une variable de condition conjointement avec une sémaphore, de façon à **garantir que la condition ne peut pas devenir vraie entre le test et la suspension du processus** (car à ce moment l'effet attendu du `signal(C)` serait perdu)

Variables de Condition (6)

- Version avec broadcast :

```
dépot(n):
  acquérir(S)
  solde <- solde + n
  céder(S)
  broadcast(C)                          ***nouveau***

retrait(n):
  acquérir(S)                           ***nouveau***
  boucle:
    si solde >= n
      solde <- solde - n
      céder(S)
    sinon
      wait(C,S)                         ***nouveau***
      goto boucle
```

Variables de Condition (7)

- Version avec signal :

```
dépot(n):
  acquérir(S)
  solde <- solde + n
  céder(S)
  signal(C)                              ***nouveau***

retrait(n):
  acquérir(S)                            ***nouveau***
  boucle:
    si solde >= n
      solde <- solde - n
      céder(S)
      signal(C)                          ***nouveau***
    sinon
      wait(C,S)                          ***nouveau***
      goto boucle
```

Variables de Condition (8)

- La version avec `signal(C)` et `broadcast(C)` sont correctes lorsque les processus font des **retraits de même valeur**
- Seulement la version avec `broadcast(C)` fonctionne correctement lorsque les retraits sont de valeur quelconque
- Voici un exemple où la version avec `signal(C)` ne fonctionne pas correctement

```
processus P1          processus P2          processus P3

dépot(2)              retrait(5)
                        => wait(C,S)
                                            retrait(3)
                                              => wait(C,S)
dépot(2)
  => signal(C)        si solde>=5...
                        => wait(C,S)        ***P3 jamais
                                             réveillé***
```

Variables de Condition (9)

- Version avec signal corrigée :

```
dépot(n):
  acquérir(S)
  solde <- solde + n
  céder(S)
  signal(C)                              ***nouveau***

retrait(n):
  acquérir(S)                            ***nouveau***
  boucle:
    si solde >= n
      solde <- solde - n
      céder(S)
      signal(C)                          ***nouveau***
    sinon
      signal(C)                          ***nouveau***
      wait(C,S)                          ***nouveau***
      goto boucle
```

Variables de Condition (10)

- Avec 3 processus ou plus cette solution donne une **attente active**!
- Ce n’est donc pas une solution acceptable

Synchronisation sous POSIX threads

- La bibliothèque “POSIX threads” offre les types et routines:

```
// types: pthread_mutex_t et pthread_mutexattr_t

int pthread_mutex_init (pthread_mutex_t* mutex,
                        const pthread_mutexattr_t* mutexatt
int pthread_mutex_lock (pthread_mutex_t* mutex);
int pthread_mutex_trylock (pthread_mutex_t* mutex);
int pthread_mutex_unlock (pthread_mutex_t* mutex);
int pthread_mutex_destroy (pthread_mutex_t* mutex);

// types: pthread_cond_t et pthread_condattr_t

int pthread_cond_init (pthread_cond_t* cond,
                       pthread_condattr_t* cond_attr);
int pthread_cond_signal (pthread_cond_t* cond);
int pthread_cond_broadcast (pthread_cond_t* cond);
int pthread_cond_wait (pthread_cond_t* cond,
                       pthread_mutex_t *mutex);
int pthread_cond_timedwait (pthread_cond_t* cond,
                            pthread_mutex_t* mutex,
                            const struct timespec* time);
int pthread_cond_destroy (pthread_cond_t* cond);
```

Exemple 1: Compte Banquaire


```
1.  #include <iostream>
2.  #include <pthread.h>
3.
4.  int solde = 0;
5.  pthread_mutex_t mut;
6.  pthread_cond_t cv;
7.
8.  void depot (int n)
9.  {
10.   pthread_mutex_lock (&mut);
11.   solde += n;
12.   pthread_mutex_unlock (&mut);
13.   pthread_cond_broadcast (&cv);
14. }
15.
16. void retrait (int n)
17. {
18.   pthread_mutex_lock (&mut);
19.   while (solde < n)
20.     pthread_cond_wait (&cv, &mut);
21.   solde -= n;
22.   pthread_mutex_unlock (&mut);
23. }
```

```
24. void* processus (void* param)
25. {
26.   depot (1);
27.   retrait (5);
28.   depot (10);
29.   return NULL;
30. }
31.
32. int main ()
33. {
34.   pthread_t tid[5];
35.   void* resultat;
36.
37.   pthread_mutex_init (&mut, NULL);
38.   pthread_cond_init (&cv, NULL);
39.
40.   for (int i = 0; i<5; i++)
41.     pthread_create (&tid[i], NULL, processus, NULL);
42.
43.   for (int i = 0; i<5; i++)
44.     pthread_join (tid[i], &resultat);
45.
46.   pthread_mutex_destroy (&mut);
47.   pthread_cond_destroy (&cv);
48.
49.   cout << "solde = " << solde << "\n";
50.
51.   return 0;
52. }
```

Exemple 2: FIFO

- Un FIFO est utile lorsque deux processus veulent s'échanger une information (message) **sans que les processus ne soient couplés fortement**
- Le processus **envoyeur** dépose le message dans le FIFO (au moment qu'il choisit), et le processus **receveur** attends la présence d'un message dans le FIFO (au moment qu'il choisit)
- Un FIFO peut avoir une capacité limitée (tamponage de N messages au maximum) ou une capacité illimitée
- Dans le cas spécial $N = 1$, l'envoyeur doit attendre que le FIFO soit vide avant d'envoyer un message, et le receveur doit attendre que le FIFO soit non-vide avant de retirer un message du FIFO
- Ça se fait facilement avec une variable de condition

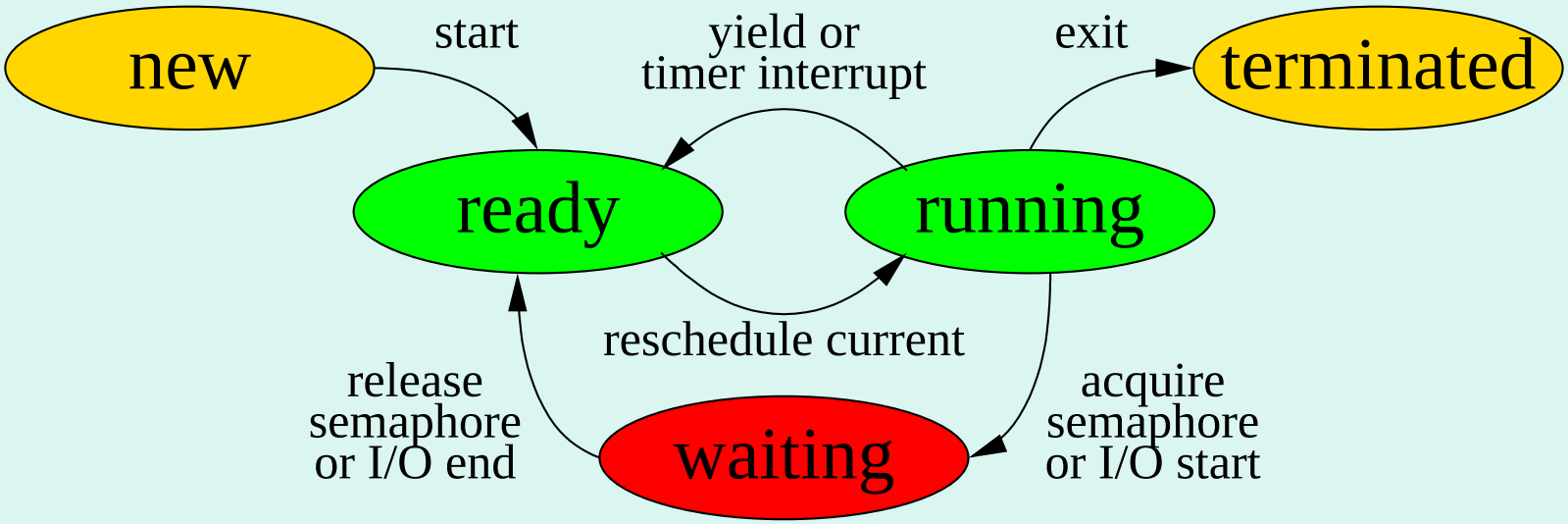
```
1.  char* message;
2.  bool empty = TRUE;
3.  pthread_mutex_t mut;
4.  pthread_cond_t send_cv; // envoyeurs suspendus
5.  pthread_cond_t recv_cv; // receveurs suspendus
6.
7.  void send (char* m)
8.  {
9.    pthread_mutex_lock (&mut);
10.   while (!empty)
11.     pthread_cond_wait (&send_cv, &mut);
12.   message = m;
13.   empty = FALSE;
14.   pthread_mutex_unlock (&mut);
15.   pthread_cond_signal (&recv_cv);
16. }
17.
18. char* recv ()
19. {
20.   pthread_mutex_lock (&mut);
21.   while (empty)
22.     pthread_cond_wait (&recv_cv, &mut);
23.   char* temp = message;
24.   empty = TRUE;
25.   pthread_mutex_unlock (&mut);
26.   pthread_cond_signal (&send_cv);
27.   return temp;
28. }
```

```
29. void* processus (void* param)
30. {
31.   send ("hello");
32.   send ("world");
33.   return NULL;
34. }
35.
36. int main ()
37. {
38.   pthread_t tid;
39.
40.   pthread_mutex_init (&mut, NULL);
41.   pthread_cond_init (&send_cv, NULL);
42.   pthread_cond_init (&recv_cv, NULL);
43.
44.   pthread_create (&tid, NULL, processus, NULL);
45.
46.   cout << recv ();
47.   cout << recv ();
48.
49.   return 0;
50. }
```

L'ordonnanceur (1)

- L'**ordonnanceur de processus** ("scheduler") est responsable de la gestion des processus (activation, suspension, ...)

- Chaque changement d'état d'exécution du processus demande une intervention de l'ordonnanceur

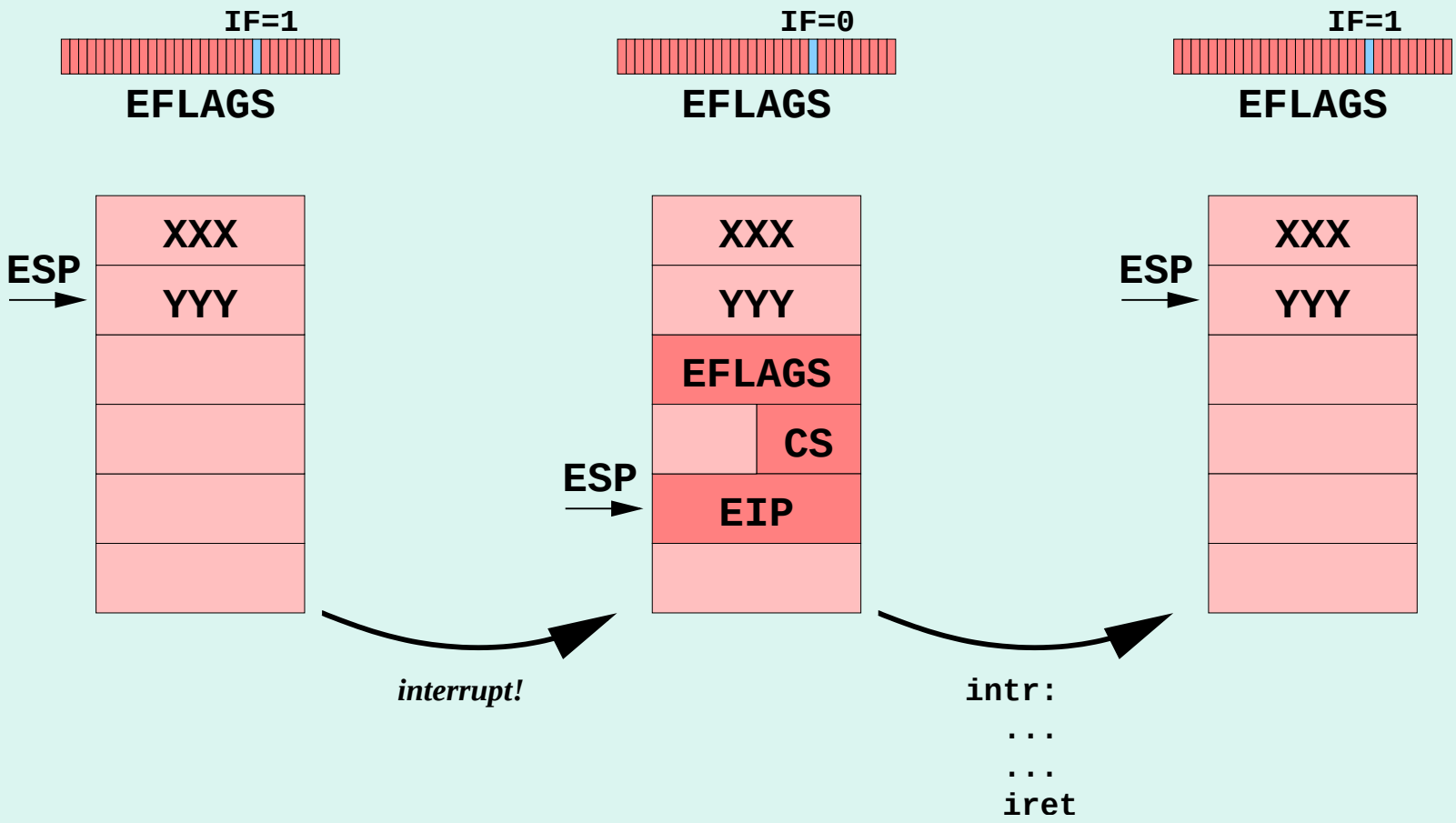


L'ordonnanceur (2)

- Un ordonnanceur simple peut conserver les processus exécutables dans une **file d'attente** ("ready queue")
- Sur un monoprocesseur, le processus à la tête de cette file est le **processus présent** (celui a qui est assigné le CPU)
- Les nouveaux processus s'ajoutent à la queue de la file
- Lorsque le processus présent doit attendre sur une sémaphore (ou une variable de condition), l'ordonnanceur **sauve son contexte** dans le descripteur de processus ou sa pile (les registres y compris `pc`, `sp` et les flags), puis **transfère le processus** de la `readyq` à la file d'attente de la sémaphore, et **réveille le nouveau processus présent** en **rétablissant son contexte**

L'ordonnanceur (3)

- Le mécanisme de **changement de contexte** doit sauvegarder le contexte d'exécution du processus
- Cela est similaire au mécanisme de traitement d'interruption :



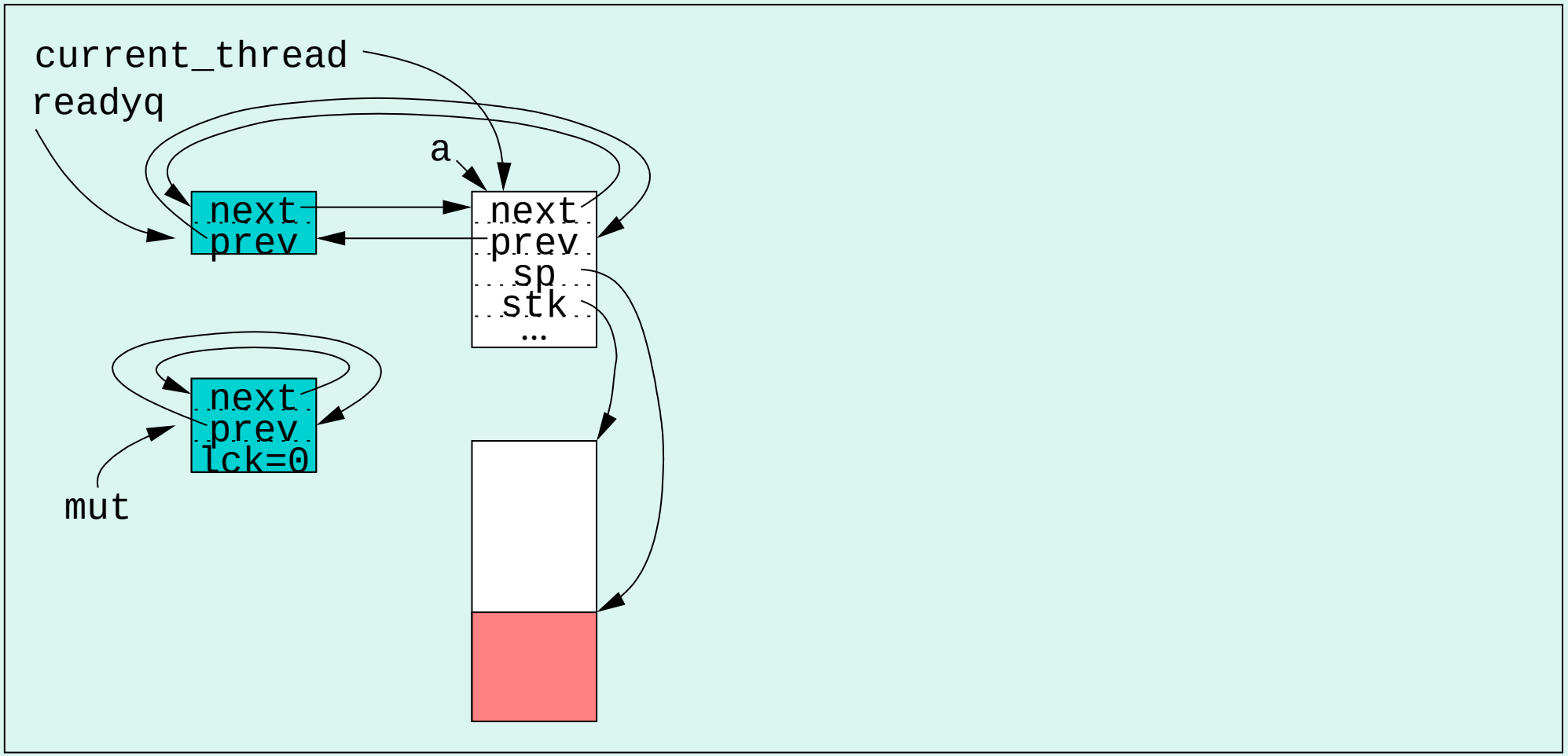
L'ordonnanceur (4)

L'ordonnanceur (5)


```
1.  pthread_t a;
2.  pthread_t b;
3.  pthread_mutex_t mut;
4.
5.  int compteur = 0;
6.
7.  void* processus (void* param)
8.  { pthread_mutex_lock (&mut);                          //6
9.    compteur++;
10.   pthread_mutex_unlock (&mut);
11.   return NULL;
12. }
13.
14. int main ()
15. { a = pthread_self ();                                //1
16.   pthread_mutex_init (&mut, NULL);                    //2
17.   pthread_mutex_lock (&mut);                          //3
18.   pthread_create (&b, NULL, processus, NULL);         //4
19.                                                       //5:int time
20.   compteur++;                                         //7
21.   pthread_mutex_unlock (&mut);                        //8
22.
23.   ...
24.
25.   return 0;
```

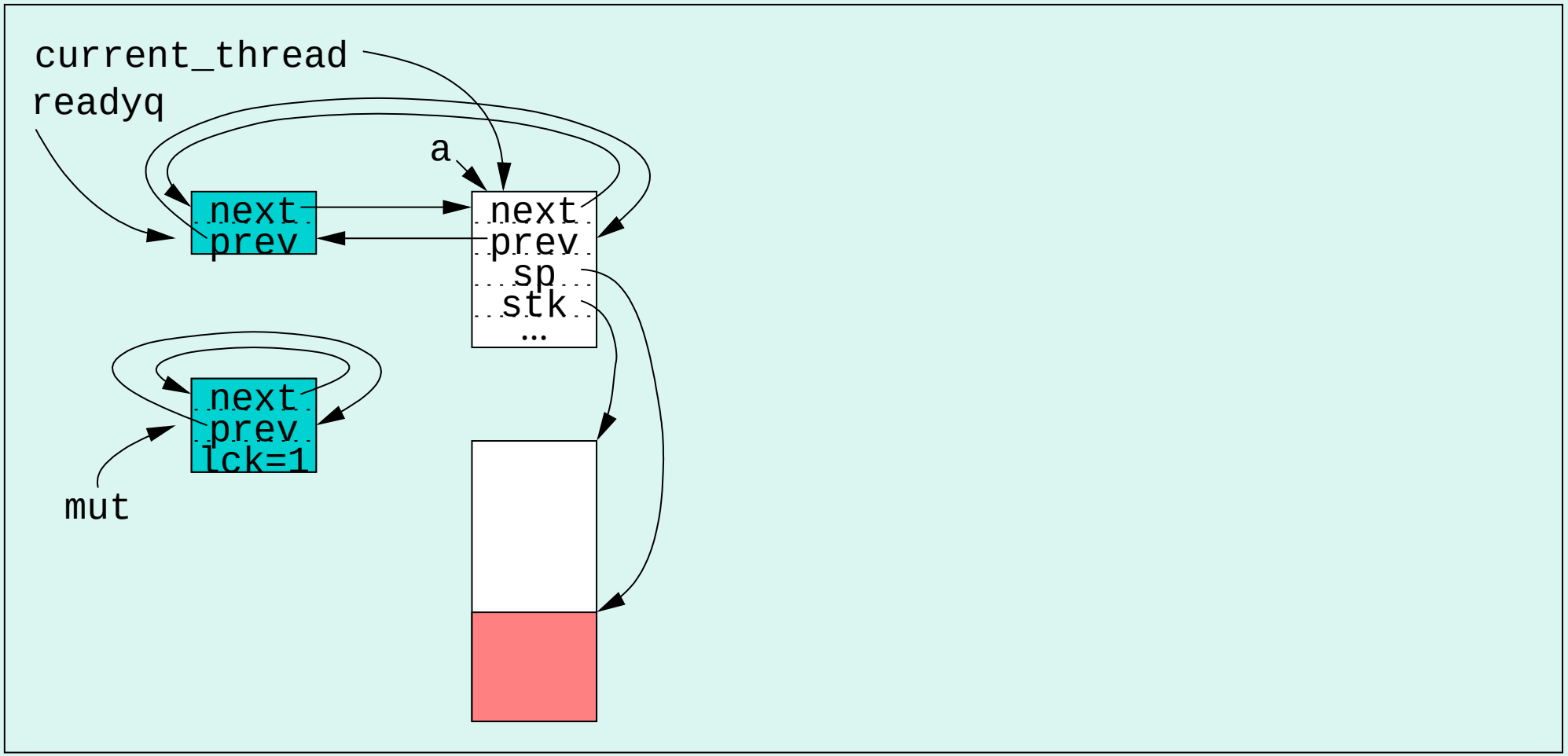
L'ordonnanceur (6)

- Après: `a=pthread_self(); pthread_mutex_init(&mut,...`



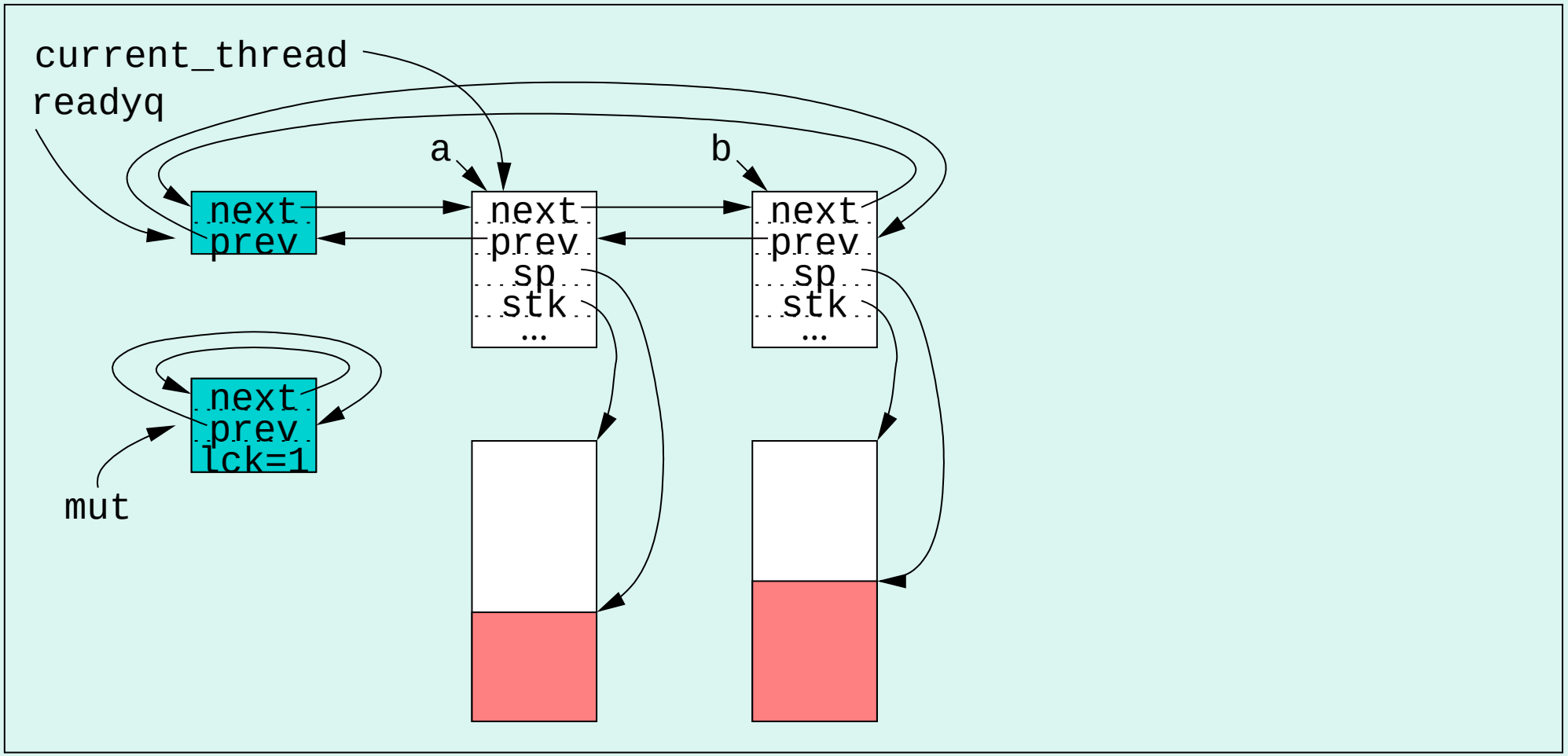
L'ordonnanceur (7)

- Après: `pthread_mutex_lock (&mut);`



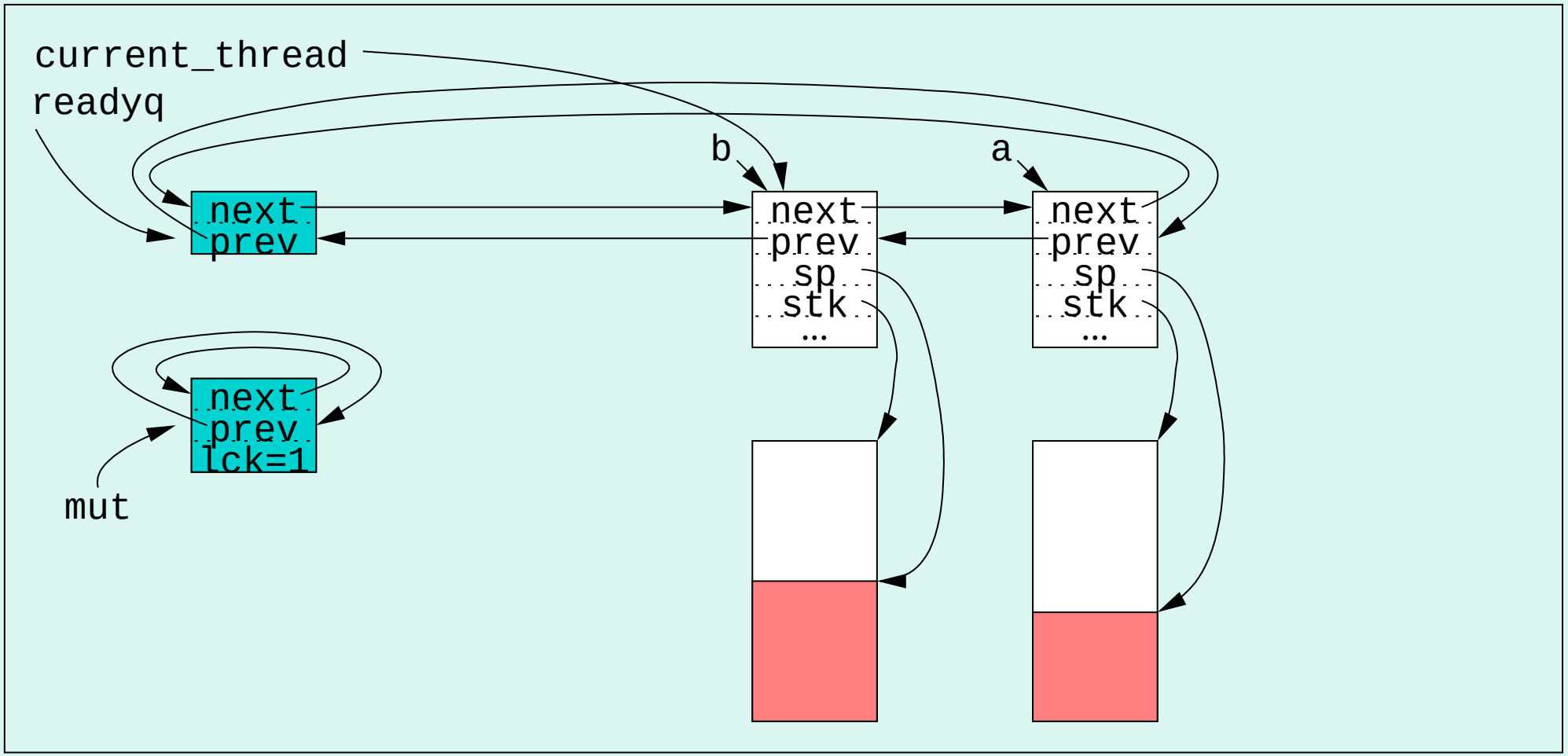
L'ordonnanceur (8)

- Après: `pthread_create (&b, NULL, processus, NULL);`



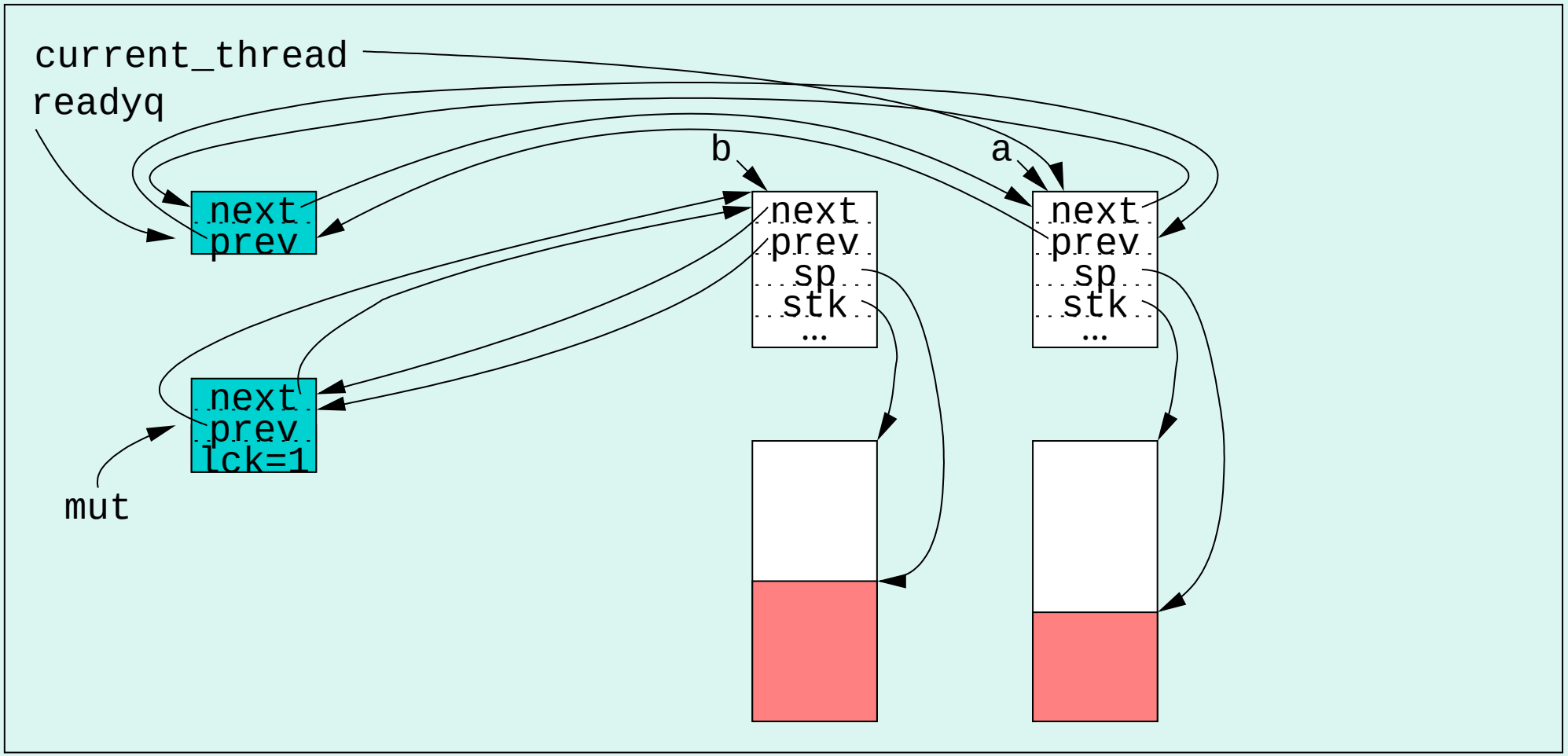
L'ordonnanceur (9)

- Après: expiration du "quantum" (interruption du timer)



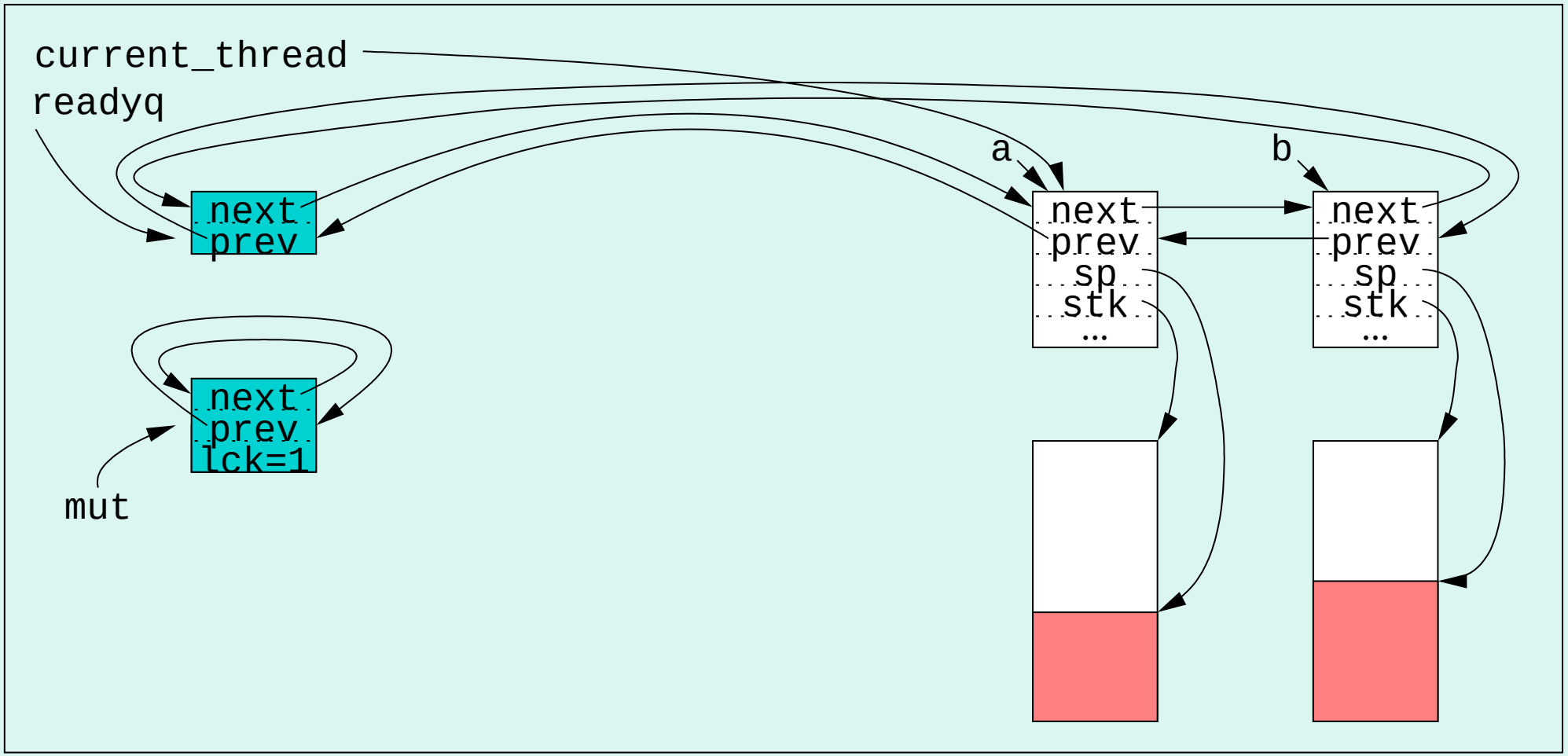
L'ordonnanceur (10)

- Après: `pthread_mutex_lock (&mut);`



L'ordonnanceur (11)

- Après: `compteur++;  pthread_mutex_unlock (&mut);`



Synchronisation Structurée

- Les sémaphores et variables de condition sont des mécanismes de relativement **bas niveau** (non-structurés) pour synchroniser des processus (il est facile d'obtenir des bugs de logique et il faut parfois plusieurs lignes de code pour implanter un certain type de synchronisation)

- Plusieurs bibliothèques et langages concurrents possèdent des **mécanismes de synchronisation structurés**

Barrière

- Dans une synchronisation de type **barrière**, un sous-ensemble bien défini (ou tous) les processus participent à la synchronisation
- Les processus qui arrivent à la barrière attendent que **tous les autres processus du sous-ensemble** soient arrivés avant de quitter la barrière
- Cela est utile pour diviser un calcul parallèle en phases séquentielles
- Exemple : animation video, P1 dessine le haut de l'écran et P2 dessine le bas

```
        processus P1

for (;;)
  { dessiner_haut ();
    barriere ();
    ajuster_le_modele ();
    barriere ();
  }
```

```
        processus P2

for (;;)
  { dessiner_bas ();
    barriere ();

    barriere ();
  }
```

Moniteur (1)

- Un **moniteur** c'est un objet (au sens OO) avec la contrainte qu'**au maximum un processus à la fois peut exécuter une de ses méthodes** (toutes ses méthodes sont donc mutuellement exclusives)

- Le compilateur implante les moniteurs à l'aide de sémaphores (une sémaphore par objet), mais c'est invisible au programmeur

- Les moniteurs sont utilisés par Java, mais les méthodes mutuellement exclusives sont **indiquées explicitement** (avec le mot clé `synchronized`)

Moniteur (1)

- Exemple : gestion d'un compte banquaire

```
1.  class Compte
2.   {
3.    private int solde;
4.
5.    Compte() { solde = 0; }
6.
7.    public synchronized int lire ()
8.      {
9.        return solde;
10.     }
11.
12.   public synchronized void depot (int n)
13.     {
14.       solde += n;
15.     }
16.
17.   public synchronized void retrait (int n)
18.     {
19.       solde -= n;
20.     }
21.  }
```

```
class Proc extends Thread
 {
  private Compte c;

  Proc(Compte compte) { c = compte; }

  public void run()
    { for (int i=0; i<100000; i++) c.depot(1); }
 }

class Banque
 {
  public static void main (String[] args)
     throws InterruptedException
   {
    Compte compte = new Compte();

    Proc p1 = new Proc(compte); // créer processus 1
    Proc p2 = new Proc(compte); // créer processus 2

    p1.start(); // démarrer processus 1
    p2.start(); // démarrer processus 2

    p1.join();  // attendre fin du processus 1
    p2.join();  // attendre fin du processus 2

    System.out.println (compte.lire());
  }
 }
```

Rendez-vous (1)

- Un **rendez-vous** c'est une partie de programme ou **2 processus se fusionnent en 1** (temporairement pour cette partie)
- C'est en quelque sorte l'inverse d'une exclusion mutuelle car le rendez-vous est seulement exécuté **si les deux processus sont arrivés au rendez-vous** (le premier arrivé va attendre l'autre)
- En plus de servir de point de synchronisation, le rendez-vous permet d'**échanger des données** entre les 2 processus
- En Ada: le processus P1 déclare qu'il est **prêt à accepter un rendez-vous** et le processus P2 fait **un appel de ce rendez-vous** (possiblement avec des paramètres) pour déclencher le rendez-vous

Rendez-vous (2)

- Syntaxe (déclaration d'un rendez-vous):

```
accept ⟨nom⟩( ⟨paramètres_formels⟩ ) do
   ⟨énoncés⟩
end;
```

- Syntaxe (appel d'un rendez-vous):

```
⟨processus⟩.⟨nom⟩( ⟨paramètres_actuels⟩ );
```

- Lorsqu'un processus *P* exécute une **déclaration** d'un rendez-vous portant un certain *nom*, c'est seulement parmis les processus qui font **l'appel** *P.nom(...)* que sera choisi le processus *P'* avec lequel se fera le rendez-vous
- Les processus qui font plutôt l'appel *P.autre_nom(...)* **attendront** tout simplement que *P* atteigne une déclaration portant ce nom

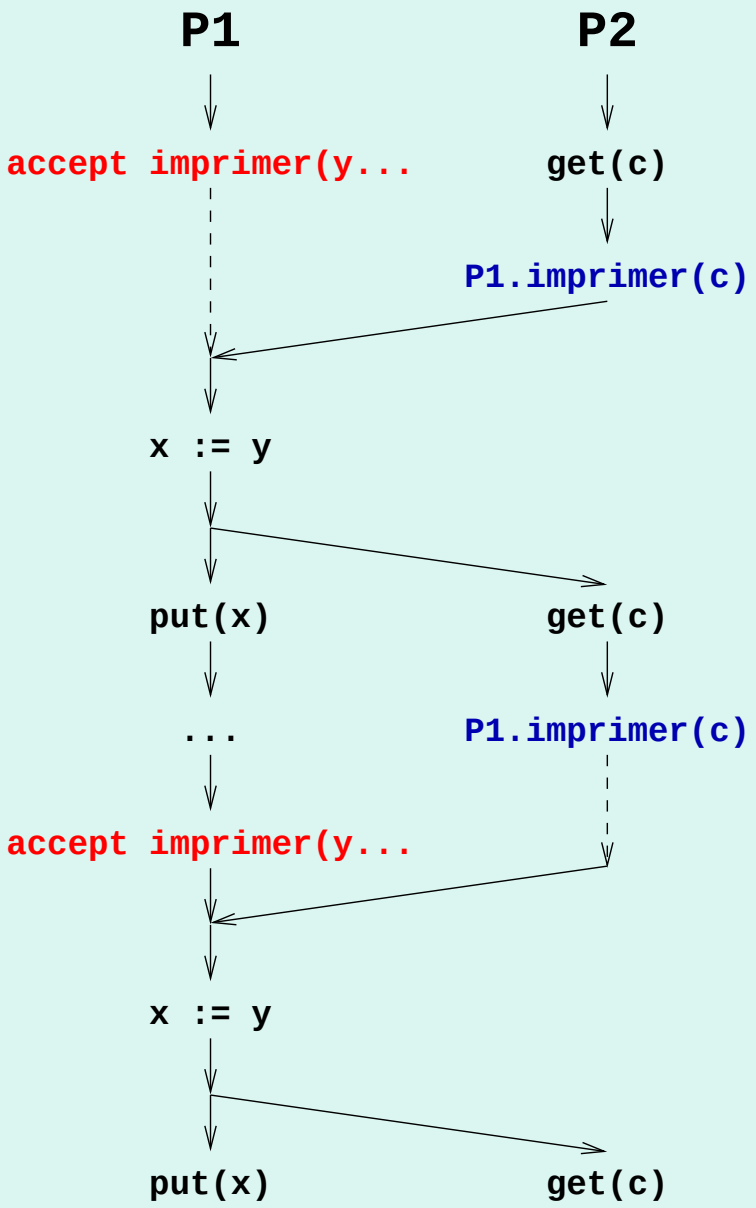
Rendez-vous (3)

- Exemple : producteur/consommateur (FIFO de profondeur 0)

```
1.  task body P1 is  -- le consommateur
2.    x : character;
3.  begin
4.    loop
5.      accept imprimer( y : in character ) do
6.        x := y;
7.      end;
8.      put(x);  -- écrire à l'écran
9.      ...;     -- autre traitement
10.   end loop;
11. end P1;
12.
13. task body P2 is  -- le producteur
14.   c : character;
15. begin
16.   loop
17.     get(c);           -- lire caractère du clavier
18.     P1.imprimer(c);   -- se présenter au rendez-vous
19.   end loop;
20. end P2;
```

Rendez-vous (4)

- Exécution de l'exemple:



Rendez-vous (5)

- Le mécanisme de rendez-vous est capable de simuler les sémaphores
- Idée: chaque sémaphore est un processus qui gère l'état de la sémaphore

```
1.  task body S is  -- le processus ‘‘sémaphore’’
2.  begin
3.    loop
4.      accept acquerir do
5.        -- rien
6.      end;
7.      accept ceder do
8.        -- rien
9.      end;
10.   end loop;
11. end S;
12.
13. procedure incrementer_compteur is
14. begin
15.   S.acquerir;
16.   compteur := compteur+1;
17.   S.ceder;
18. end incrementer_compteur;
19.
20. task body P1 is
21. begin
22.   incrementer_compteur;
23. end P1;
24.
25. task body P2 is
26. begin
27.   incrementer_compteur;
28. end P2;
```

Étreinte Fatale (1)

- Une **étreinte fatale** (“deadlock”) survient lorsqu’un ensemble de processus se mettent en attente et l’événement qui pourrait les réveiller **ne se produit jamais**
- Exemple 1:

```
void afficher_message (char* msg)
{
  lock (mut);
  if (msg != NULL) cout << msg;
  else afficher_message ("msg est NULL");
  unlock (mut);
}
```

- Certains systèmes (e.g. POSIX threads) offrent des mutex **“récursifs”** qui peuvent être acquis à nouveau par le même processus (une variable interne au mutex compte le niveau de “récursivité”)

Étreinte Fatale (2)

- Exemple 2: un processus veut imprimer le fichier de log du système, et l'autre processus veut imprimer une page de test et ajouter un message dans le fichier de log

```
    processus P1                          processus P2

lock (mut_fichier_log);          |   lock (mut_imprimante);
lock (mut_imprimante);           |   lock (mut_fichier_log);
                                 |
SECTION CRITIQUE:                |   SECTION CRITIQUE:
  envoyer fichier à              |     initialiser imprimante
  l'imprimante                   |     et ajouter message dans
                                 |     le fichier de log
                                 |
unlock (mut_imprimante);         |   unlock (mut_fichier_log);
unlock (mut_fichier_log);        |   unlock (mut_imprimante);
```

Étreinte Fatale (3)

- Supposons que les synchronisations du programme sont nécessaires pour contrôler l'accès à des **ressources partagées** (mémoire, CPU, fichiers, imprimantes, etc) et se font par un processus suivant le protocole:

  1. **requête**: acquisition de la ressource (possiblement après une attente); se traduit par l'acquisition d'une sémaphore
  2. **utilisation**: la ressource obtenue est utilisée exclusivement par le processus
  3. **libération**: la ressource est libérée par le processus qui a fait la requête

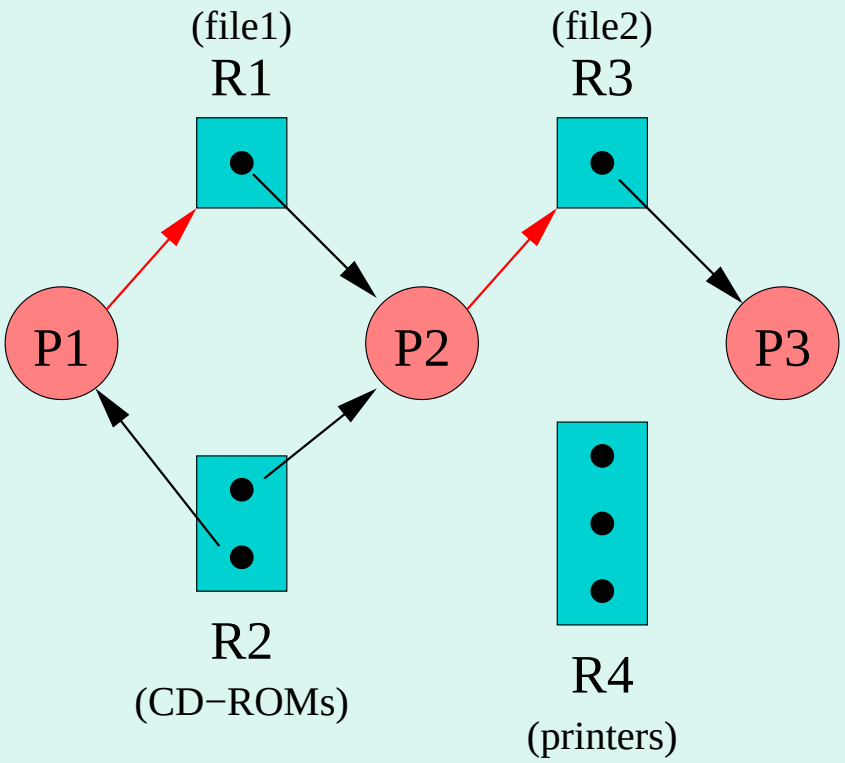
- Exemples: `malloc`/`free`, `open`/`close`, `connect`/`disconnect`

Graphe d'Allocation de Ressources

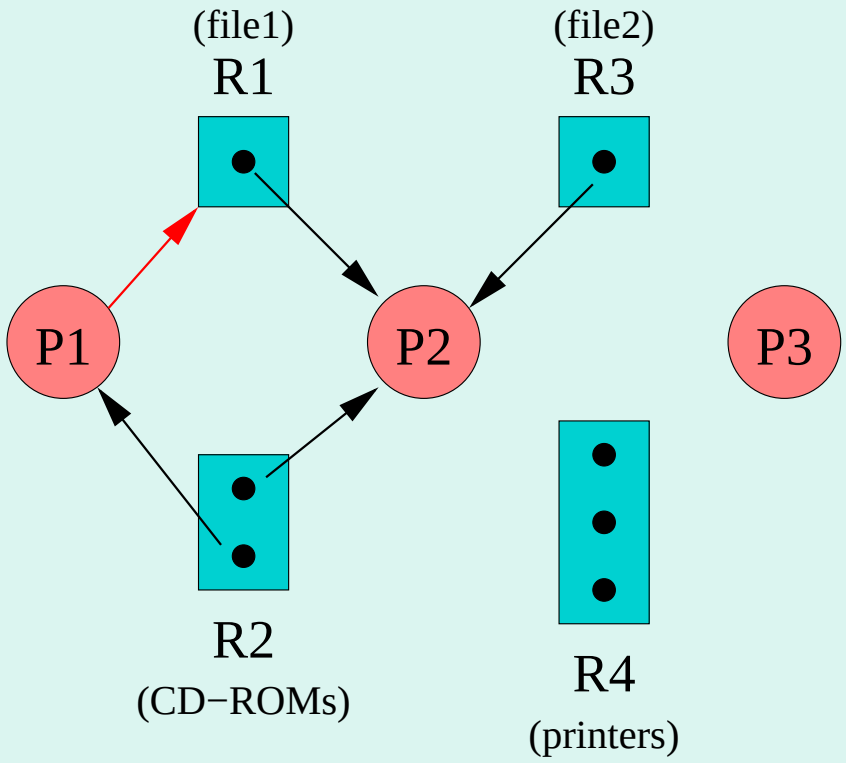
- Soit $P = \{P_1, \ldots, P_n\}$ l'ensemble des **processus** dans le système et $R = \{R_1, \ldots, R_m\}$ l'ensemble des **types de ressources** (un type peut avoir plusieurs instances mais elles doivent être "équivalentes")
- Le **graphe d'allocation de ressources** contient les noeuds $P \cup R$
  - il existe un arc $P_i \rightarrow R_j$ si le processus P_i est en attente d'une instance du type de ressource R_j (**arc de requête**)
  - il existe un arc $R_j \rightarrow P_i$ si une instance de la ressource R_j a été allouée au processus P_i (**arc d'assignation**)
- Lorsqu'une requête est satisfaite, un arc de requête est renversé pour le transformer en arc d'assignation

Exemple (1)

- Graphe à un certain point du calcul:

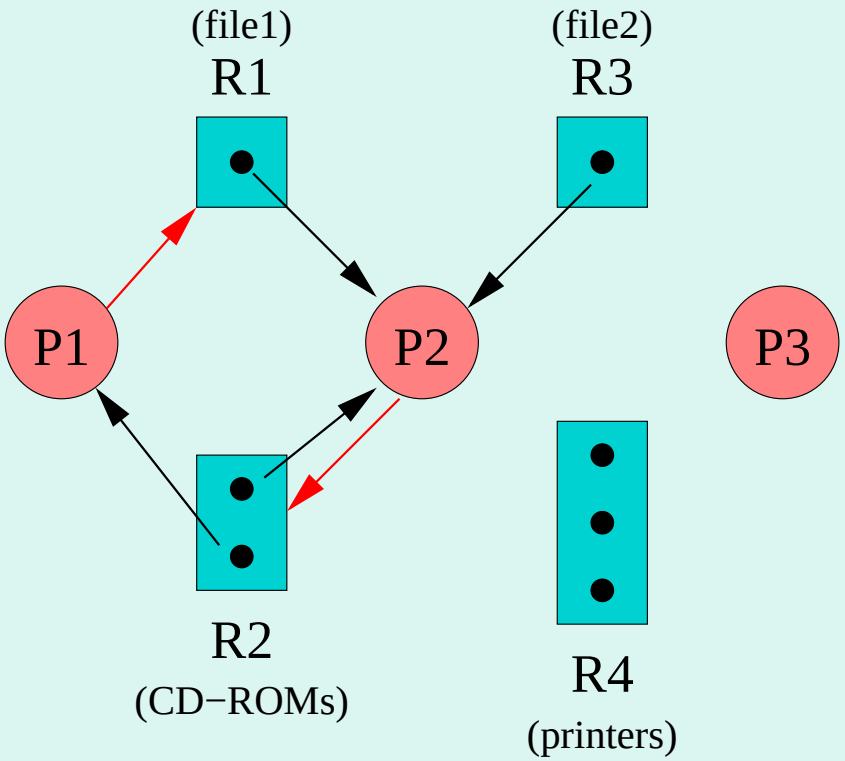


- Après la libération de R_2 par P_3:

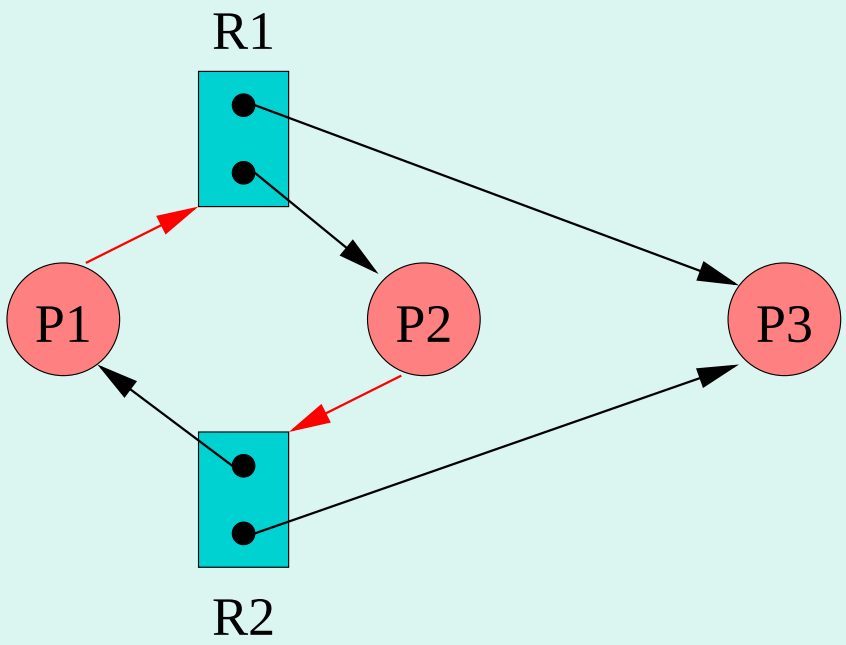


Exemple (2)

- Un cycle qui mêne à un deadlock:



- Un cycle qui ne mêne pas à un deadlock:



Détection de Deadlock (1)

- Si toutes les ressources n'ont qu'une seule instance (utilisation de "mutex"), alors il suffit de **détecter la présence d'un cycle**
- L'ensemble des **processus sur un cycle** sont en deadlock
- Algorithme $O(n)$:

1. $\forall$ noeud N, colorer N en blanc
2. $\forall$ noeud N, faire une fouille en profondeur à partir du noeud N, colorant les noeuds blancs en gris, puis en noir lorsque tous les enfants de N sont noirs
3. cycle ssi noeud gris est rencontré

Détection de Deadlock (2)

- Si des ressources peuvent avoir plus d'une instance, alors utiliser cet algorithme :

1. Prendre une copie du graphe
2. Identifier un processus P dont toutes les requêtes peuvent être satisfaites (ou qui n'a pas de requête) et retirer P et ses assignations
3. Répéter l'étape 2 tant que des processus peuvent être retirés
4. Les **processus restants** (s'il y en a) sont en deadlock

Détection de Deadlock (3)

- Les algorithmes de détection de deadlock sont lents; il faut **limiter leur usage**
  - Seulement nécessaire lorsqu'une requête doit attendre
  - Employer lorsque l'utilisation du CPU est faible ($<$40%)

- Et si un deadlock est détecté?
  - Rien faire (à chaque mois on "reboot")
  - Tuer tous les processus en deadlock, ou un seul
  - Retour d'un processus à un état précédent ("Rollback") pour libérer des ressources

Prévention de Deadlocks (1)

- Une approche pour éviter les attentes circulaires (et donc les deadlocks) c'est d'**imposer un ordre total** d'acquisition des ressources

- À chaque ressource est attribué une étiquette entière unique

- Un processus qui a fait l'acquisition de certaines ressources peut seulement placer une requête pour une ressource dont l'étiquette est **plus grande** que toutes celles qu'il détient présentement

- Un processus qui désire acquérir les trois ressources A, B, et C pour faire un certain traitement doit s'arranger pour les acquérir dans le bon ordre (donc le processus doit être conscient de l'ordre à respecter)

Prévention de Deadlocks (2)

- L'absence de deadlock se prouve facilement par contradiction
- Supposons qu'il y a un deadlock et que les processus $\{P_1, P_2, \ldots, P_n\}$ sont sur le cycle et que $\forall i$: P_i attends la ressource R_i et possède la ressource $R_{(i \bmod n)+1}$
- Pour avoir atteint cet état, il faut que $\forall i$: $etiquette(R_i) > etiquette(R_{(i \bmod n)+1})$
- Donc: $etiquette(R_1) > etiquette(R_2) > \ldots > etiquette(R_n) > etiquette(R_1)$
- Impossible!

Ordonnancement

- Les objectifs de l'ordonnanceur:

1. **Utilisation du CPU et des périphériques** s'approchant de 100%
2. **Débit élevé**, par exemple nombre de processus complétés par unité de temps (cela suppose que tous les processus ont la même "importance")
3. **Temps bout-à-bout faible**: le temps entre le démarrage d'un processus et sa complétion
4. **Temps d'attente dans l'état exécutable faible**
5. **Temps de réponse faible**: le temps entre l'arrivée d'une donnée au processus et une réponse de celui-ci (particulièrement recherché pour les processus interactifs, p.e. editeur de texte)

Algorithmes d'Ordonnancement

1. **Premier arrivé premier servi** ("FCFS"): les processus sont exécutés dans l'ordre où ils deviennent exécutables
2. **Plus petite tâche en premier** ("Shortest-Job-First"): parmis les processus exécutables, celui qui va terminer le plus vite est exécuté en premier (suppose qu'on peut estimer le temps d'exécution à l'avance)
3. **Ordonnancement "round-robin" (cyclique)**: utilisation d'un timer pour multiplexer le CPU; chaque processus obtient à tour de rôle un **quantum** ou **"time slice"** pour avancer son exécution
4. **Ordonnancement par priorité**: les processus exécutables les plus prioritaires exécutent en premier (FCFS pour un même niveau de priorité)

Analyse

- Soit 3 processus qui ont été soumis pour exécution dans l'ordre P1, P2, P3 et les temps d'exécution respectifs sont 24, 3, 3 (et qu'il n'y a pas d'E/S)
- Ordonnancement avec chaque approche

	Average wait time
FCFS	(0+24+27)/3=17
SJF	(0+3+6)/3=3
RR	(6+5.5+6)/3=5.833

- Pour minimiser le temps d'attente dans l'état exécutable, l'ordonnancement **SJF est optimal** mais RR s'en approche et a l'avantage de ne pas supposer les temps d'exécution connus à l'avance

Système de Priorité (1)

- Un système de priorité permet d'attribuer un **degré d'"urgence"** à chaque processus:
  - Il est critique qu'un processus de haute priorité **complète son travail rapidement**, même si c'est au dépend d'un processus de plus faible priorité
- Par exemple un processus qui doit **répondre rapidement à un événement** (panne de courant, arrivée d'un paquet du réseau) a une priorité élevée (autrement des données pourraient être perdues)
- Il peut y avoir un nombre fixe de priorités (petit: < 10, moyen: 30, ou grand: > 100) ou même infini
- La priorité d'un processus peut **être fixée à sa création** ou **varier pendant son exécution**

Système de Priorité (2)

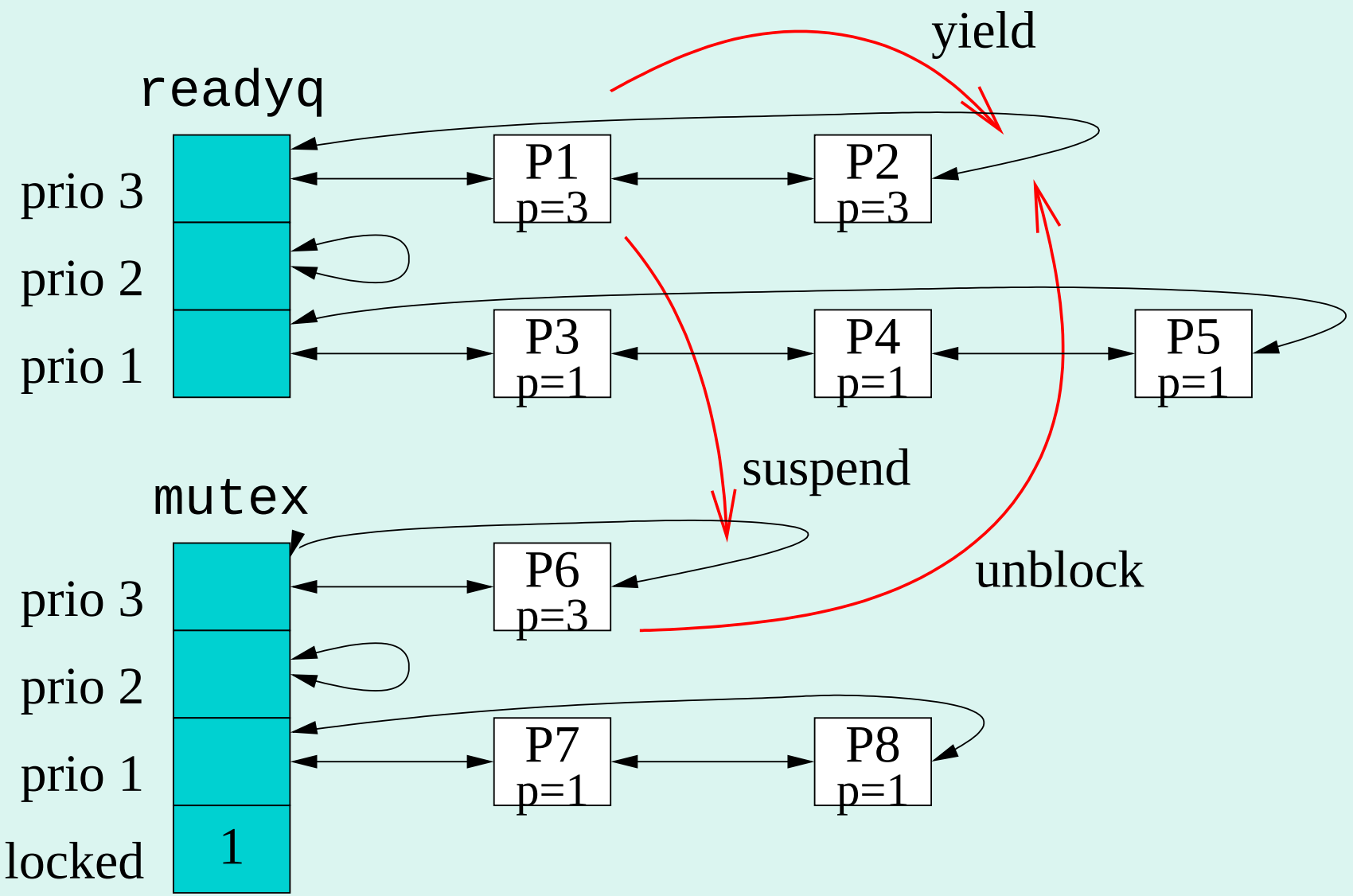
- Les garanties de l'ordonnanceur vis-à-vis la priorité des processus varie d'un S.E. à un autre, mais en général:
  - $\forall$ processus en exécution P et $\forall$ processus exécutable P': $prio(P) \geq prio(P')$
  - Sur monoprocesseur, le processus en exécution a la **plus haute priorité des processus exécutables** (les autres processus doivent attendre qu'il n'existe plus de processus exécutable de plus haute priorité)
  - Soit p la **priorité la plus élevée** parmis l'ensemble de processus en attente sur un objet de synchronisation; le **premier processus réveillé** sera choisi parmis les processus de priorité p
  - S'il y a un choix de processus (à exécuter, à réveiller) c'est **premier arrivé premier servi** ("FCFS")

Système de Priorité (3)

- L'ordonnanceur utilise des files d'attentes implantées avec des **queues de priorité** (pour le "ready queue" et objets de synchronisation)
- Plusieurs choix d'implantation sont possibles:
  - **liste ordonnée**: insertion = $O(n)$, retrait = $O(1)$
  - **tableau de p listes de processus**: insertion = $O(1)$, retrait = $O(p)$
  - **monceau ("heap"), arbres rouge-noir/2-3/"splay"**: insertion = $O(\log n)$, retrait = $O(\log n)$
- Une queue de priorité peut aussi être utilisée pour conserver tous les processus en attente chronométrée (par exemple pour implanter `sleep (n)`); la "priorité" c'est **l'inverse du temps d'expiration de l'attente**

Système de Priorité (4)

- Tableau de 3 listes de processus:



Système de Priorité (5)

- Pour accélérer la recherche du processus le plus prioritaire, on peut conserver (dans une variable entière) un bit par priorité pour signaler les listes non-vides (dans les deux cas: `101`) et aussi la priorité la plus élevée

Priorités sous UNIX (1)

- La priorité d'un processus UNIX **varie au cours de sa vie** en fonction des ressources qu'il utilise et d'un niveau de "nice": -20 (peu gentil envers les autres processus) à 20 (très gentil envers les autres processus)

- L'ordonnanceur tente d'**augmenter la priorité des processus qui n'ont pas obtenu beaucoup de temps CPU** dans un passé récent (quelques secondes)

- Ce genre de processus effectue **surtout des entrées/sorties** et il est important de privilégier leur exécution pour **augmenter le degré d'utilisation des périphériques** (parallélisme des composantes de l'ordinateur) et **réduire le temps de réponse des programmes interactifs**

Priorités sous UNIX (2)

- Les paramètres suivants sont utilisés (par 4.4BSD):
  - **“load”**: la charge est le nombre moyen de processus exécutables durant la dernière minute (échantillonné à chaque seconde)
  - **“nice”**: chaque processus a un niveau de gentillesse (-20 à 20) de partage du CPU avec les autres processus (0 par défaut, négatif réservé à `root`)
  - **“share(x)”**: le nombre de top d’horloge (à 100 Hz) où le processus était en exécution pendant la seconde x dans le passé

Priorités sous UNIX (3)

- Soit $f = \frac{2 \times load}{2 \times load + 1}$
  Note: f est une valeur entre 0 (système peu chargé) et 1 (système très chargé)
- Soit cpu une variable propre au processus qui est mise à jour (+1) à chaque top d'horloge où le processus exécute et à chaque seconde avec

$$cpu \leftarrow f \times cpu + nice$$

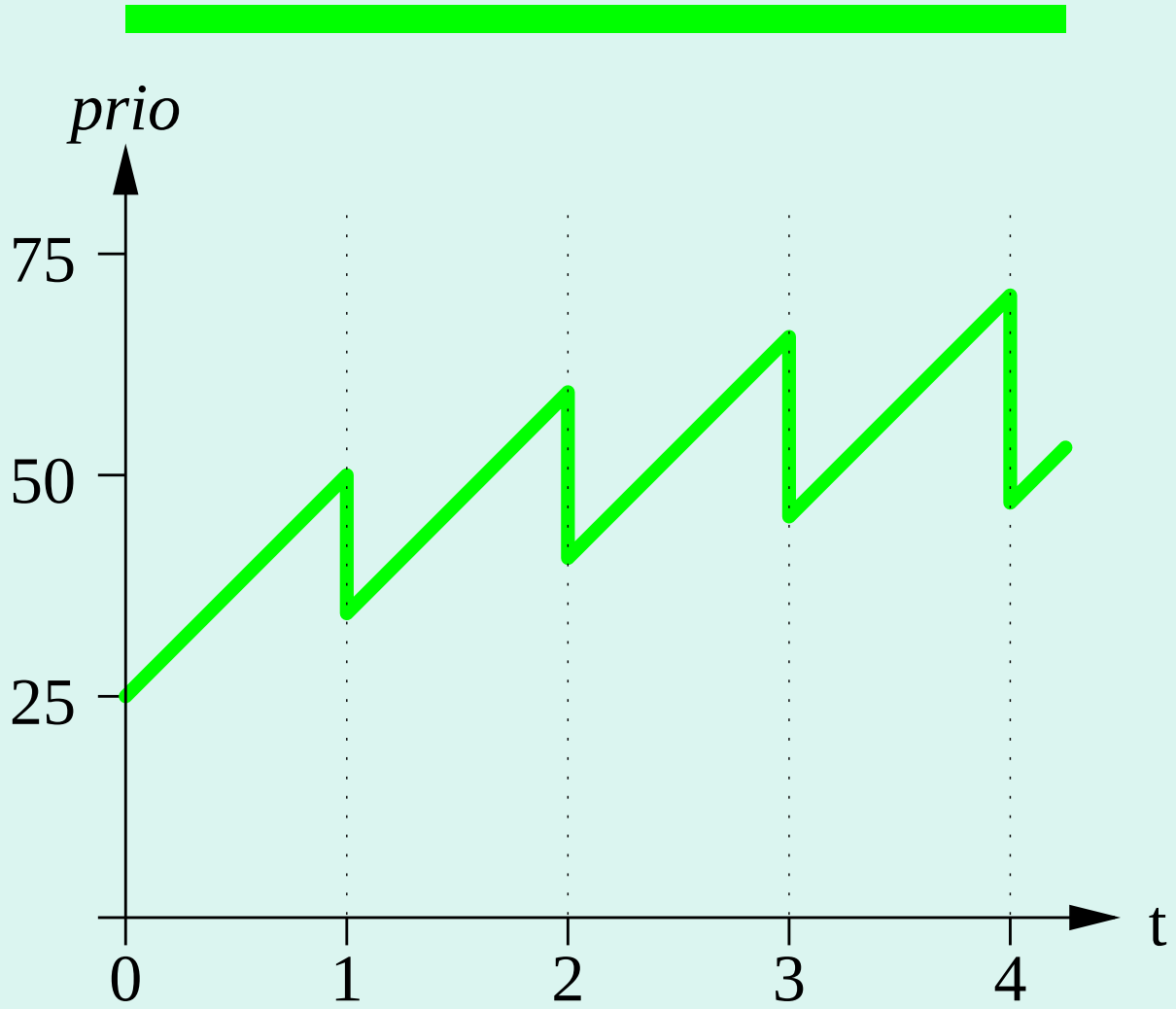
- La priorité du processus est

$$prio = \frac{cpu + 8 \times nice}{4}$$

- La priorité va à l'inverse de la valeur de $prio$ (c'est-à-dire $prio = 0$ est la plus haute priorité)

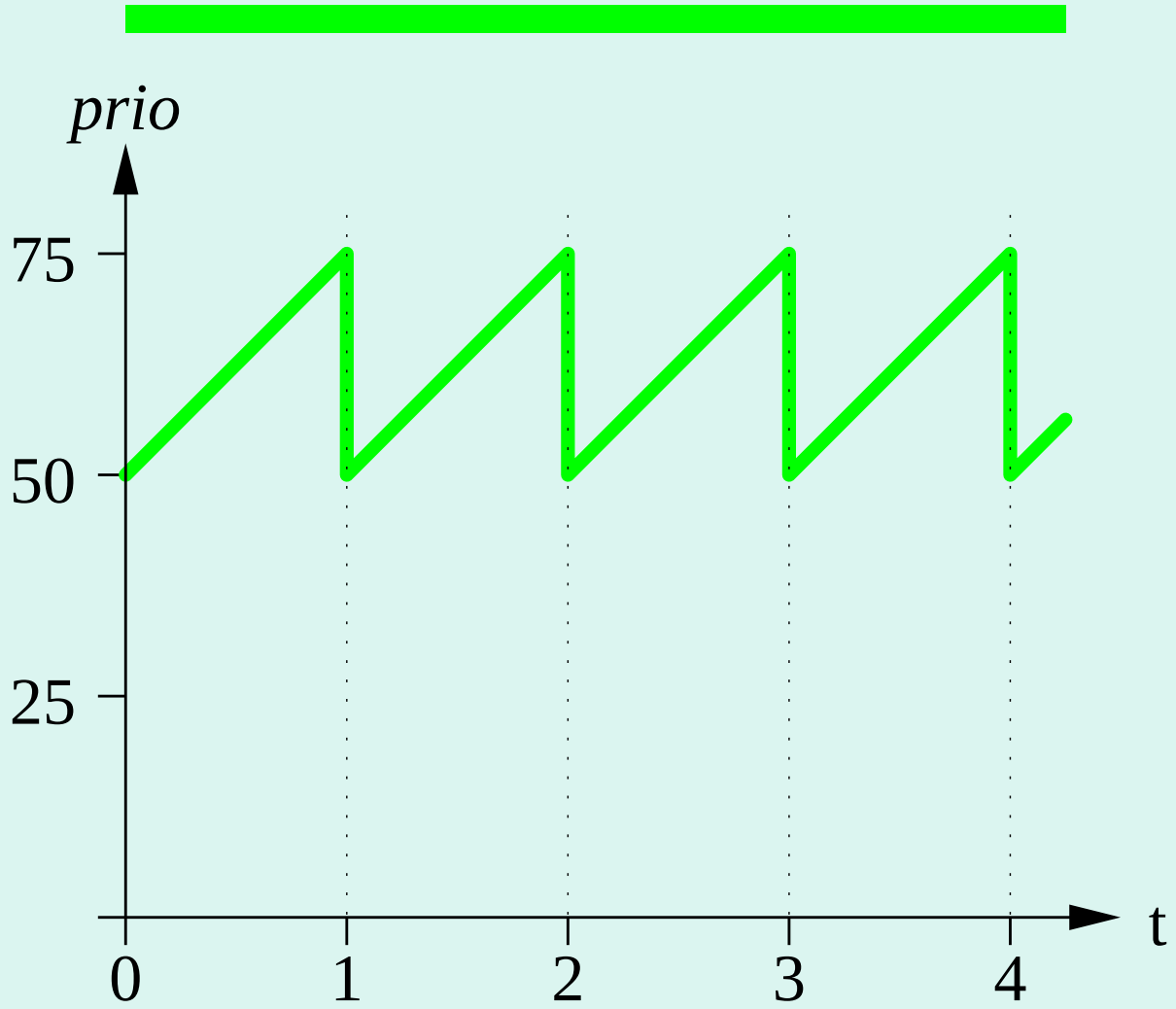
Priorités sous UNIX (4)

- Exemple 1: un seul processus exécutable en tout temps ($f = 0.66$) et $nice = 0$



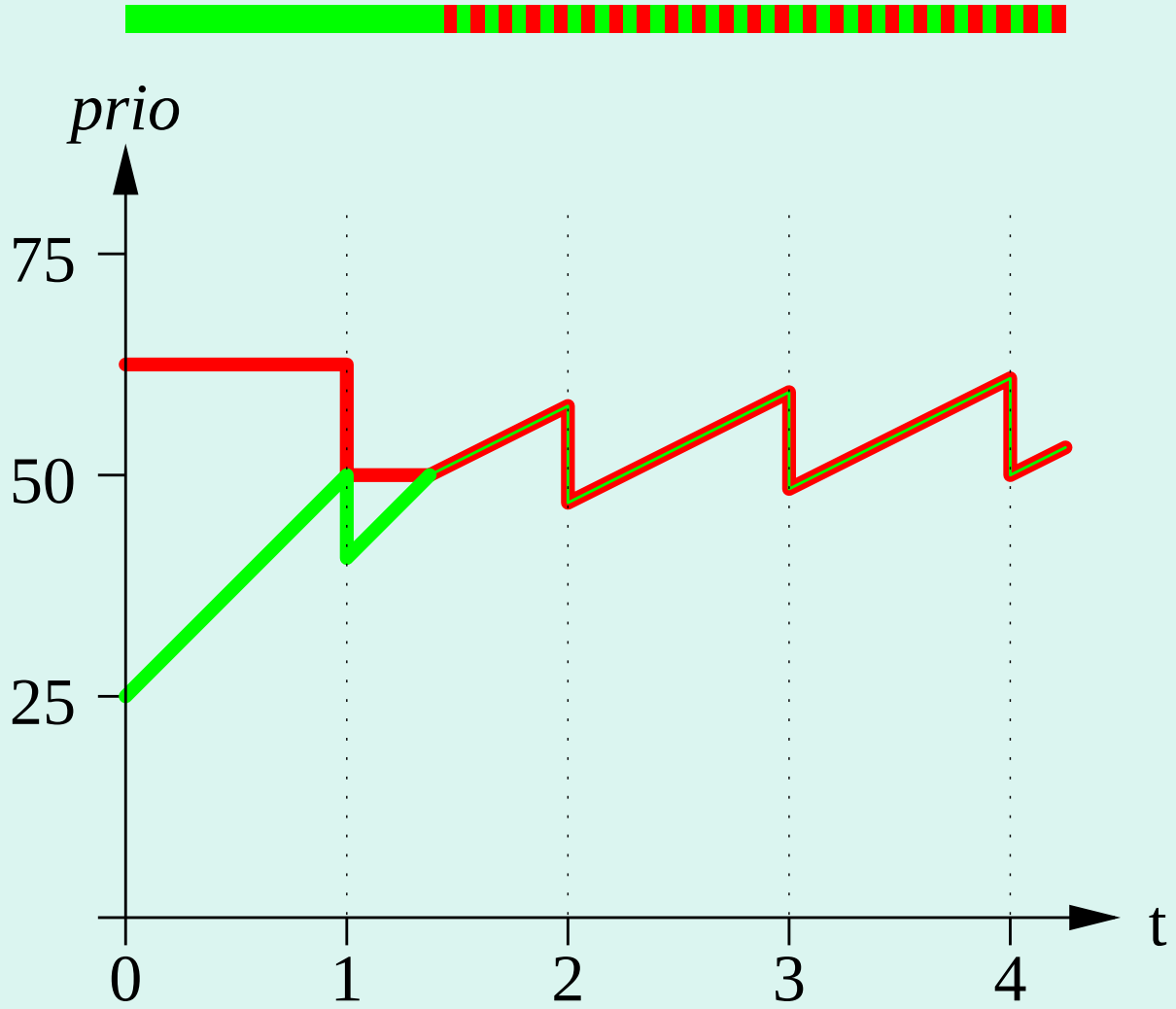
Priorités sous UNIX (5)

- Éventuellement on atteint un **point d'équilibre**



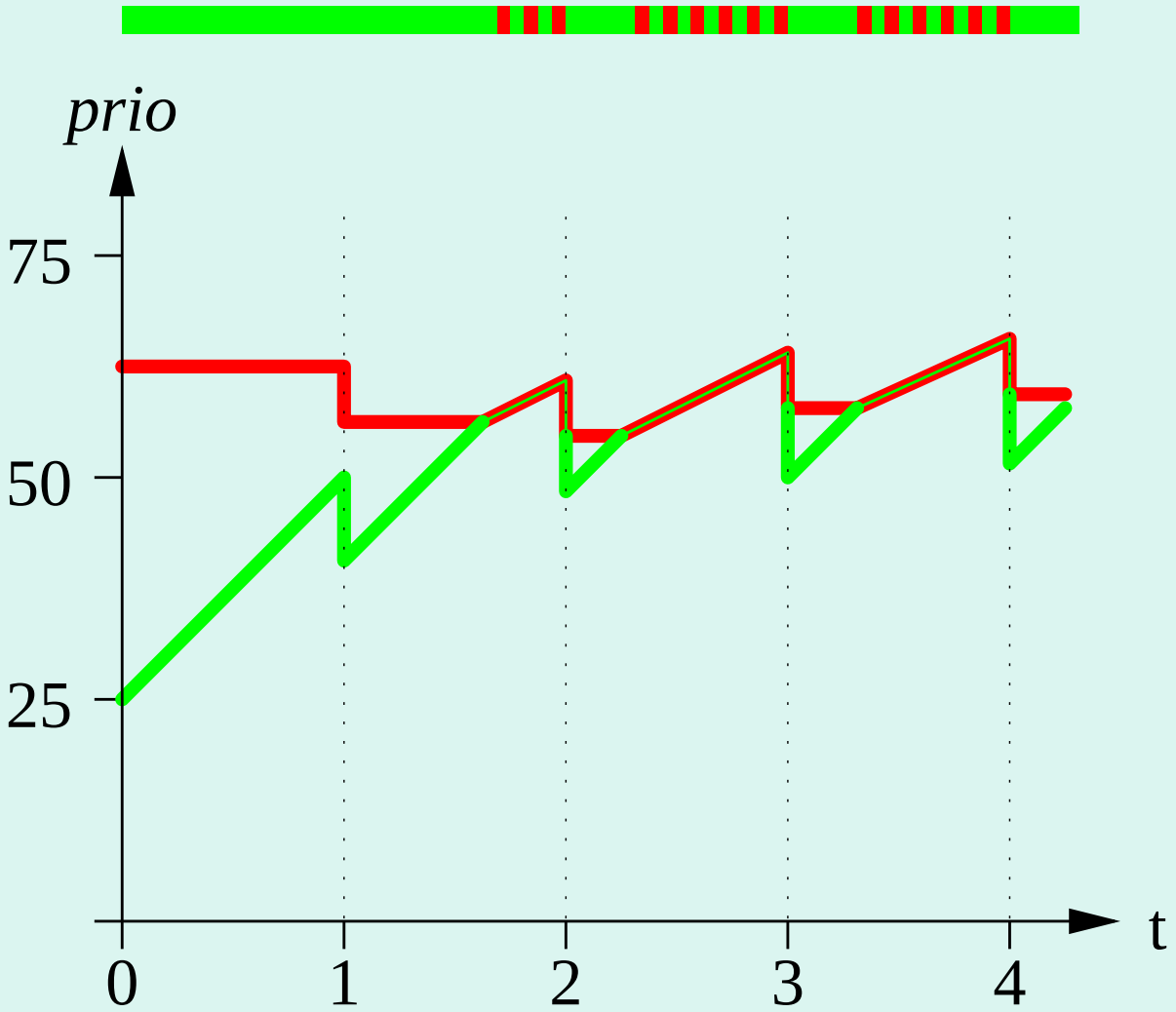
Priorités sous UNIX (6)

- Exemple 2: deux processus exécutables en tout temps ($f = 0.80$) et avec $nice = 0$



Priorités sous UNIX (7)

- Exemple 3: deux processus exécutables en tout temps ($f = 0.80$) et avec $nice = 0$ (vert) et $nice = 6$ (rouge)

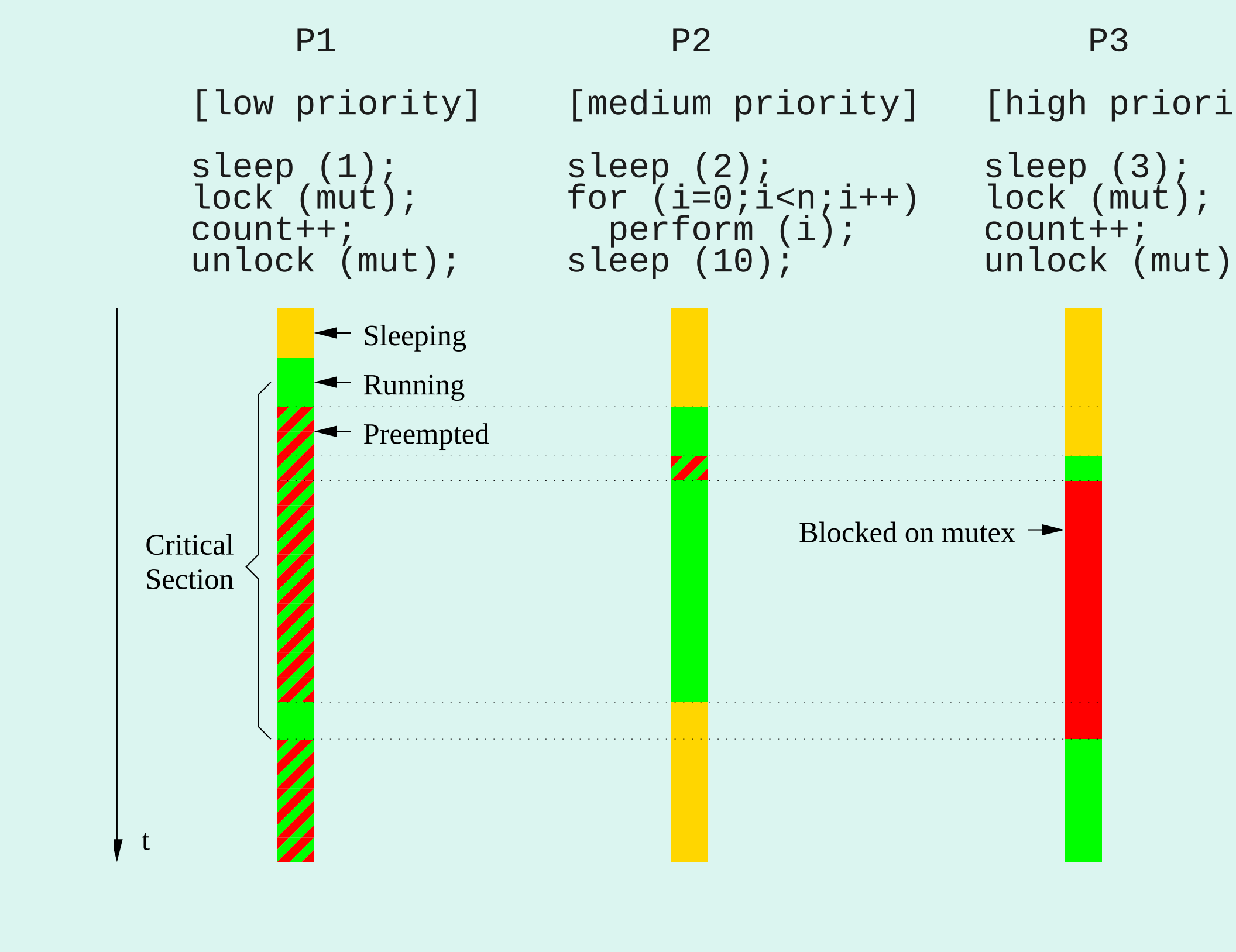


Inversion de Priorité (1)

- Les synchronisations d'un programme ont parfois un effet subtil sur la **priorité observée** des processus
- Pendant qu'un processus P1 de basse priorité exécute une section critique, un processus P3 de plus haute (ou basse) priorité est **bloqué à l'entrée de la section critique**
- C'est nécessaire pour réaliser l'exclusion mutuelle demandée
- P3 doit attendre que P1 sorte de la section critique; si la section critique est courte (comme il se doit) P3 n'attendra pas longtemps

Inversion de Priorité (2)

- Mais si un processus P2 de moyenne priorité se met à exécuter **pendant que P1 est dans la section critique**; P3 doit indirectement attendre que P2 ne soit plus exécutable (ce qui est potentiellement long car P2 n'est pas dans une section critique)


P1
[low priority]
sleep (1);
lock (mut);
count++;
unlock (mut);
P2
[medium priority]
sleep (2);
for (i=0;i<n;i++)
perform (i);
sleep (10);
P3
[high priori
sleep (3);
lock (mut);
count++;
unlock (mut)
Sleeping
Running
Preempted
Critical
Section
Blocked on mutex
t

Inversion de Priorité (4)

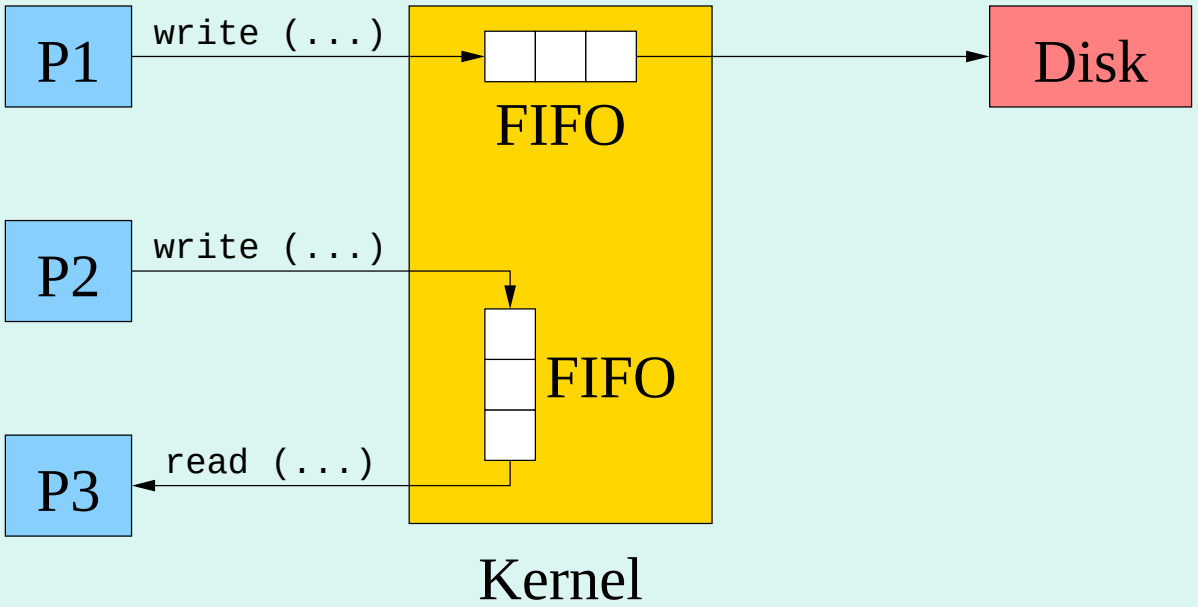
- On dit qu'il y a **inversion de priorité** car un processus de priorité moyenne (P2) empêche un processus de priorité élevée (P3) d'exécuter

- Pour pallier ce problème, l'ordonnanceur peut **temporairement augmenter la priorité du processus** P1 pour que P2 ne puisse pas empêcher son exécution (il suffit de lui donner une **priorité au moins aussi grande que P2**)

- **Héritage de priorité**: définir $prio(P) =$ maximum de la priorité "normale" de P et la priorité de tous les processus qui sont en attente d'une sémaphore présentement acquise par P

Communication avec l'Extérieur

- L'encapsulation des processus demande d'avoir des mécanismes spécifiques pour **échanger des informations** avec d'autres processus: les **fichiers**, **"pipes"**, **"sockets"** et **signaux**
- Cela se fait toujours par l'entremise du kernel



- Le kernel peut valider et contrôler les accès

Signaux (1)

- Le mécanisme des **signaux** est l'analogue de haut-niveau du mécanisme des **interruptions**
- UNIX définit une trentaine de signaux différents
  - **asynchrones** (p.ex. SIGINT = interruption CTRL-C)
  - **synchrones** (p.ex. SIGSEGV = "Segmentation violation")

Signaux (2)

- Les routines système `signal` et `sigaction` permettent d'installer pour chaque signal un **traiteur de signal** ("signal handler") qui se fait appeler lorsque ce signal est reçu par le processus

```
typedef void (*sig_t) (int);
sig_t signal(int sig, sig_t func);
```

`signal` retourne le handler précédent

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t*, void*);
    sigset_t sa_mask; // signaux à masquer
    int sa_flags;
    void (*sa_restorer)(void);
}

int sigaction(int sig,
              struct sigaction* act,
              struct sigaction* oldact);
```

Signaux (3)

- Si un traiteur n'est pas installé, certains signaux sont **ignorés** (SIGWINCH) et d'autres **terminent le processus** (SIGSEGV)

- Tout comme il est possible d'inhiber les interruptions, il est possible d'inhiber les signaux à l'aide des routines système `sigsetmask`, etc.

```
int sigblock (int mask);
int siggetmask (void);
int sigsetmask (int mask);
int sigmask (int signum);
```

- Exemple

```
int oldmask = sigblock(sigmask(SIGINT));
...
sigsetmask(oldmask);
```

Signaux (4)

- La routine système `kill(`pid`,`$signum$`)` envoit le signal $signum$ au processus pid (ou à tous les processus si $pid = -1$)
- La majorité des appels système qui peuvent bloquer (`read`, `write`, `sleep`, `select`) retournent un code d'erreur (EINTR) s'ils se font interrompre par un signal non-inhibé
- Les signaux sont utiles pour

1. forcer un processus à terminer, se suspendre, continuer
2. indiquer une condition urgente à un autre processus
3. indiquer un changement d'état à un processus (pour que par exemple il consulte à nouveau ses fichiers de configuration après les avoir modifiés)

```
#define SIGHUP    1  // Hangup (POSIX)
#define SIGINT    2  // Interrupt (ANSI)
#define SIGQUIT   3  // Quit (POSIX)
#define SIGILL    4  // Illegal instruction (ANSI)
#define SIGTRAP   5  // Trace trap (POSIX)
#define SIGABRT   6  // Abort (ANSI)
#define SIGBUS    7  // BUS error (4.2 BSD)
#define SIGFPE    8  // Floating-point exception (ANSI)
#define SIGKILL   9  // Kill, unblockable (POSIX)
#define SIGUSR1   10 // User-defined signal 1 (POSIX)
#define SIGSEGV   11 // Segmentation violation (ANSI)
#define SIGUSR2   12 // User-defined signal 2 (POSIX)
#define SIGPIPE   13 // Broken pipe (POSIX)
#define SIGALRM   14 // Alarm clock (POSIX)
#define SIGTERM   15 // Termination (ANSI)
#define SIGSTKFLT 16 // Stack fault
#define SIGCHLD   17 // Child status has changed (POSIX)
#define SIGCONT   18 // Continue (POSIX)
#define SIGSTOP   19 // Stop, unblockable (POSIX)
#define SIGTSTP   20 // Keyboard stop (POSIX)
#define SIGTTIN   21 // Background read from tty (POSIX)
#define SIGTTOU   22 // Background write to tty (POSIX)
#define SIGURG    23 // Urgent condition on socket (4.2 BSD
#define SIGVTALRM 26 // Virtual alarm clock (4.2 BSD)
#define SIGPROF   27 // Profiling alarm clock (4.2 BSD)
#define SIGWINCH  28 // Window size change (4.3 BSD, Sun)
#define SIGIO     29 // I/O now possible (4.2 BSD)
#define SIGPWR    30 // Power failure restart (System V)
```

Exemple (1)


```
1.  #include <sys/types.h> // pour pid_t
2.  #include <unistd.h> // pour getpid, getppid, fork, sleep
3.
4.  #include <signal.h> // pour signal, SIGUSR1, SIGINT
5.
6.  void handler1 (int signum)
7.  { cout <<"handler1: pid="<<getpid()<<" signum="<<signum<<
8.
9.  void handler2 (int signum)
10. { cout <<"handler2: pid="<<getpid()<<" signum="<<signum<<
11.
12. int main (int argc, char* argv[])
13. {
14.   signal (SIGUSR1, handler1);
15.   signal (SIGINT,  handler2);
16.
17.   pid_t primordial = getpid ();
18.
19.   cout << "primordial=" << primordial << "\n";
20.
21.   pid_t p = fork ();
22.
23.   if (p != 0)
24.     kill (p, SIGUSR1); // envoyer interruption à l'enfant
```

Exemple (2)


```
25.    int x = sleep (5);
26.    cout << "left=" << x << "\n";
27.
28.    // inhiber les interruptions SIGINT
29.    // pendant le prochain sleep
30.
31.    int old = siggetmask ();
32.    sigblock (sigmask (SIGINT));
33.    sleep (5);
34.    cout << "le sleep est fini\n";
35.    sigsetmask (old);
36.
37.    sleep (5);
38.
39.    return 0;
40. }
```

Exemple (3)


```
% ./a.out
primordial=101
handler1: pid=102 signum=10
<CTRL-C>
handler2: pid=102 signum=2
left=1
handler2: pid=101 signum=2
left=1
<CTRL-C>
<CTRL-C>     (perdu)
<CTRL-C>     (perdu)
le sleep est fini
handler2: pid=101 signum=2
le sleep est fini
handler2: pid=102 signum=2
<CTRL-C>
handler2: pid=102 signum=2
handler2: pid=101 signum=2
```

Les Descripteurs de Fichiers (1)

- UNIX identifie chaque **canal d'E/S** d'un processus (quel que soit son type) par un **descripteur de fichier** ("file descriptor") qui est un petit entier non-négatif

- Un canal peut être **unidirectionnel** ou **bidirectionnel**

- Les routines système `open` et `close` sont normalement utilisées pour gérer l'accès à un fichier (ou autre périphérique):

```
int open (char* pathname, int flags, mode_t mode);
int close (int fd);
```

  Une valeur de retour négative indique une erreur

Les Descripteurs de Fichiers (2)

- $flags$ spécifie des options d'ouverture:
  - `O_RDONLY`, `O_WRONLY`, `O_RDWR` (direction)
  - `O_CREAT` (créer fichier s'il n'existe pas)
  - `O_TRUNC` (vider fichier à son ouverture en écriture)
  - `O_EXCL` (ne pas écraser un fichier existant)
  - `O_APPEND` (chaque écriture se fait à la fin)
  - `O_NONBLOCK` (canal non-bloquant)
- `O_NONBLOCK` : Les lectures/écritures ne font jamais d'attente (utile si un programme doit faire des E/S sur plus d'un canal à la fois)
- `O_APPEND` : Les écritures se font toujours à la fin du fichier (utile pour les fichiers de "log")

Les Descripteurs de Fichiers (3)

- *mode* spécifie les bits de permission:
  - `0400`: lecture par le propriétaire permise
  - `0200`: écriture par le propriétaire permise
  - `0100`: exécution par le propriétaire permise
  - `00X0`: lecture/écriture/exécution par le groupe permises
  - `000X`: lecture/écriture/exécution par les autres permises
- Pour un répertoire la permission d'exécution permet l'accès aux fichiers contenus mais **pas la liste**
- Pour accéder à un fichier il faut la permission de lecture ou d'exécution sur **tous les répertoires englobants**, et la permission d'accès (lecture, écriture ou exécution) sur le fichier lui même
- Exemple : `open ("/u/luc/f", O_RDWR)`

Les Descripteurs de Fichiers (4)

- *outdir* = `O_WRONLY` ou `O_RDWR`
  anydir = `O_WRONLY` ou `O_RDWR` ou `O_RDONLY`
- `open ("f", anydir)`
  ⇒ `"f"` doit exister, sinon `errno = ENOENT`
- `open ("f", anydir | O_CREAT, 0666)`
  ⇒ `"f"` est créé s'il n'existe pas
- `open ("f", anydir | O_CREAT | O_EXCL, 0666)`
  ⇒ `"f"` ne doit pas exister, sinon `errno = EEXIST`
- `open ("f", outdir | O_TRUNC | ..., 0666)`
  ⇒ `"f"` est vidé de son contenu à l'ouverture (à condition qu'aucune des erreurs ci-dessus ne survienne)

Les Descripteurs de Fichiers (5)

- Les routines système `read` et `write` sont normalement utilisées pour recevoir et envoyer des données sur le canal:

  `ssize_t read (int` fd`, void*` buf`, size_t` $count$`)`
  `ssize_t write (int` fd`, void*` buf`, size_t` $count$`)`

  `ssize_t` et `size_t` sont des types entiers
  Une valeur de retour négative indique une erreur

- $count$ indique le nombre d'octets à transférer
- La valeur de retour est le nombre d'octets effectivement transférés
- La valeur de retour peut être $< count$ (**transfert "court"**) si un signal est reçu ou si le canal est non-bloquant, de même qu'une fin de fichier sur un `read`

Exemple 1: Copier un fichier


```
1.  #define SIZE 8192
2.
3.  void copy_file (char* file_in, char* file_out)
4.  { char buf[SIZE];
5.    int in = open (file_in, O_RDONLY);
6.    int out = open (file_out,
7.                    O_WRONLY | O_CREAT | O_TRUNC,
8.                    0644);
9.
10.   for (;;)
11.     {
12.       char* p = buf;
13.       int count = read (in, p, SIZE);
14.       if (count < 0) exit (1);
15.       if (count == 0) break;
16.       while (count > 0)
17.         {
18.           int x = write (out, p, count);
19.           if (x < 0) exit (1);
20.           count -= x;
21.           p += x;
22.         }
23.     }
24.
25.   close (in); close (out);
}
```

Exemple 1: Effet de SIZE

- Plus SIZE est grand, plus le programme est rapide car moins il y aura d'appels système (voir loi de Amdahl)

```
400 MHz PowerPC, LinuxPPC, fichier = 10^6 octets

    SIZE    temps (s)
       1     4.3467
       2     2.1454
       4     1.0726
       8      .5438
      16      .2771
      32      .1451
      64      .0780
     128      .0446
     256      .0243
     512      .0160
    1024      .0118
    2048      .0094
    4096      .0077
    8192      .0074
   16384      .0078
   32768      .0078
   65536      .0078
```

Exemple 2


```
1.  #include <unistd.h> // pour fork, sleep, write, close
2.  #include <sys/types.h> // pour pid_t
3.  #include <sys/stat.h> // pour open
4.  #include <fcntl.h>
5.
6.  int main (int argc, char* argv[])
7.  {
8.    pid_t p = fork (); // créer un deuxième processus
9.    int fd = open ("log", // O_APPEND important
10.                 O_WRONLY | O_APPEND | O_CREAT,
11.                 0644);
12.
13.   for (int i=0; i<5; i++)
14.     {
15.       if (p == 0)
16.         write (fd, "enfant\n", 7);
17.       else
18.         write (fd, "parent\n", 7);
19.       sleep (1);
20.     }
21.
22.   close (fd);
23.
24.   return 0;
25. }
```

Exemple 2 (cont.)


```
% ./a.out
% cat log
parent
enfant
parent
enfant
parent
enfant
parent
enfant
parent
enfant
```

Exemple 3

- Pour traiter correctement les écritures “courtes” si des signaux sont permis, utiliser `write_all` au lieu de `write`:

```
1.  void write_all (int fd, char* buf, int count)
2.  {
3.    while (count > 0)
4.      {
5.        int x = write (fd, buf, count);
6.        if (x < 0)
7.          {
8.            switch (errno)
9.              {
10.             case EINTR: break; // ignorer signaux
11.             default: exit (1); // c'est une erreur
12.             }
13.         }
14.       else
15.         {
16.           count -= x;
17.           buf += x;
18.         }
19.     }
20. }
```

Exemple 3: Cas `EINTR`

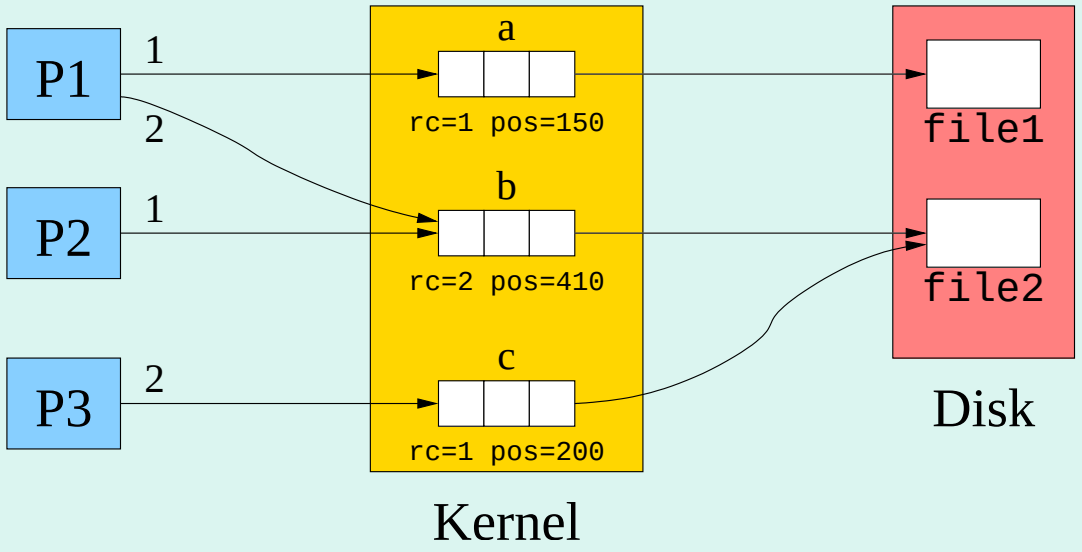
- Comment `write` agit-il en présence d'un signal si N octets ont été transférés et $N > 0$?
- La spécification de POSIX permet à `write` **de retourner une erreur avec** `errno = EINTR` ou bien **de retourner** N
- La première approche risque de causer des problèmes car **le processus ne peut pas savoir combien d'octets ont été écrits**, et donc à quel octet redémarrer l'écriture
- Quel que soit l'approche implantée, le traiteur de signal est avant tout appelé
- Une autre approche (non POSIX): `write` **retourne** N et se "souvient" qu'il y a eu un signal, et le prochain appel à `write` retourne une erreur avec `errno = EINTR` **et sans écrire un seul octet**

Partage de Canaux (1)

- Les canaux d'E/S sont des **objets maintenus par le kernel** qui occupent des ressources dans le kernel (mémoire, tampons, etc)
- Chaque descripteur de fichier est **une référence de canal qui est locale au processus** (donc le descripteur de fichier "1" dans deux processus ne correspond pas nécéssairement au même canal dans le kernel)
- La fonction `fork` crée un processus enfant qui a **les mêmes descripteurs de fichiers associés aux mêmes canaux**

Partage de Canaux (2)

- Rien n'empêche plusieurs processus de **partager un canal (ou fichier) donné**



- La **position de lecture/écriture** est maintenue dans la mémoire du kernel avec le tampon d'E/S, flags, etc
- Le kernel utilise le **"comptage de références"** (compteur décrémenté à chaque `close`) pour savoir quand libérer un canal

Partage de Canaux (3)

- Partager un canal et partager un fichier **ce n'est pas la même chose** car chaque canal a son propre état (position de lecture/écriture, mode bloquant ou non-bloquant, flags, tampons, etc)

- 2 processus qui partagent un canal

```
int fd = open ("file2", O_WRONLY | O_CREAT, 0644);
pid_t p = fork ();
```

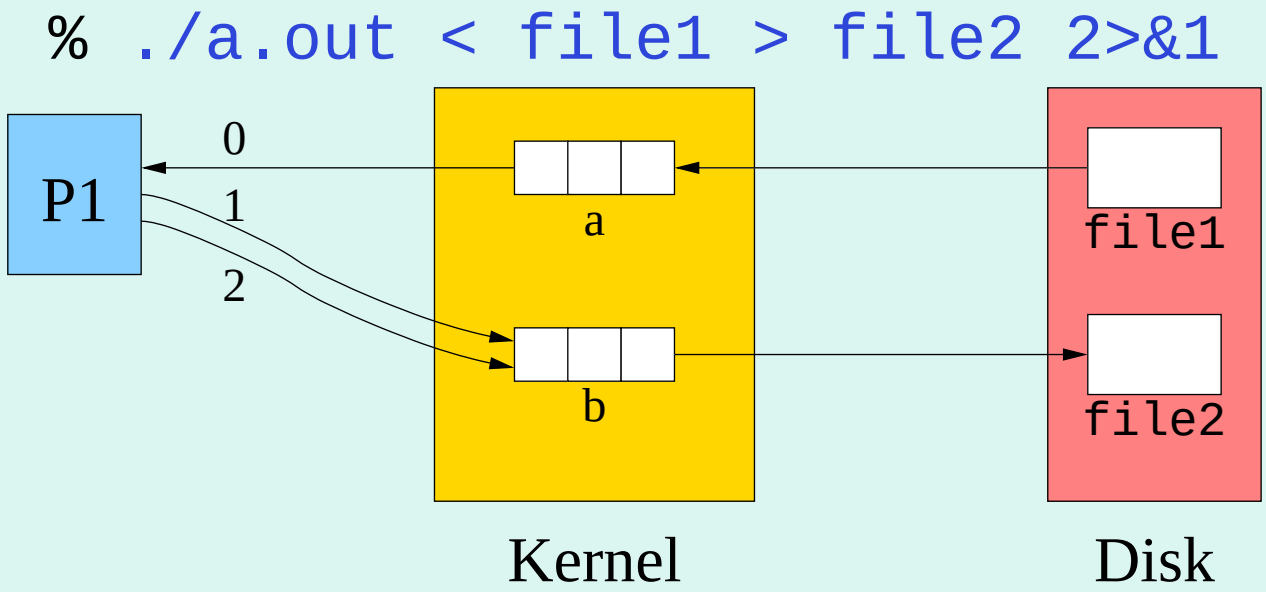
- 2 processus qui partagent un fichier

```
pid_t p = fork ();
int fd = open ("file2", O_WRONLY | O_CREAT, 0644);
```

- Le flag `O_APPEND` est utile pour le partage de fichiers en écriture

Héritage de Canaux

- Les canaux référés par un processus sont (normalement) **hérités par les processus enfants** créés par `fork` sous les **mêmes descripteurs de fichier** (voir exemple précédent)
- Les canaux pour `stdin`, `stdout` et `stderr` sont créés par le shell (par convention ce sont les descripteurs de fichier 0, 1 et 2)



Copie de Descripteurs de Fichier

- Les appels systèmes `dup` et `dup2` permettent de **copier les descripteurs de fichier** (c'est-à-dire créer une nouvelle référence à un canal existant)

  ```
  int dup (int fd);
  int dup2 (int fd, int fd2);
  ```

  $fd2$ est le descripteur de la copie (il est fermé automatiquement s'il n'est pas libre)

  `dup` en choisit un qui est libre

  le descripteur de la copie est retourné

- Cela est principalement utile avec `fork` et `exec` lorsqu'un nouveau programme doit être démarré avec un `stdin`, `stdout`, ou `stderr` particulier

Exemple de `fork`, `exec` et `dup`

- Compter les lignes du fichier “`in`” qui contiennent un nombre

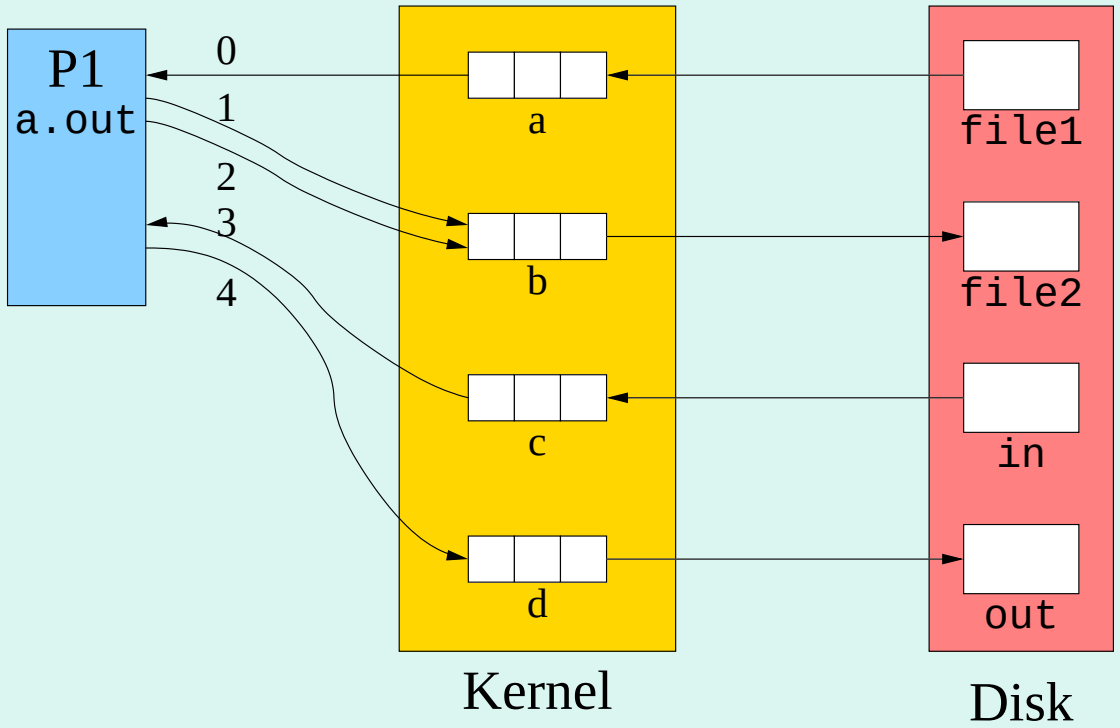
```
% cat in
ceci est 1 petit
fichier de test qui
contient 3 lignes
% cat in | grep [0-9][0-9]*
ceci est 1 petit
contient 3 lignes
% cat in | grep [0-9][0-9]* | wc
       2       7      35
```

Exemple de fork, exec et dup (cont.)


```
1.  int main (int argc, char* argv[])
2.  { int in = open ("in", O_RDONLY);
3.    int out = open ("out",
4.                    O_WRONLY | O_CREAT | O_TRUNC,
5.                    0644);
6.    pid_t p = fork ();
7.
8.    if (p == 0)
9.      { dup2 (in, 0); dup2 (out, 1); dup2 (out, 2);
10.       close (in); close (out);
11.       execlp ("grep", "grep", "[0-9][0-9]*", NULL);
12.       return 1;
13.     }
14.
15.   close (in); close (out);
16.
17.   int status;
18.   waitpid (p, &status, 0);
19.
20.   execlp ("wc", "wc", "out", NULL);
21.   return 0;
22. }
```

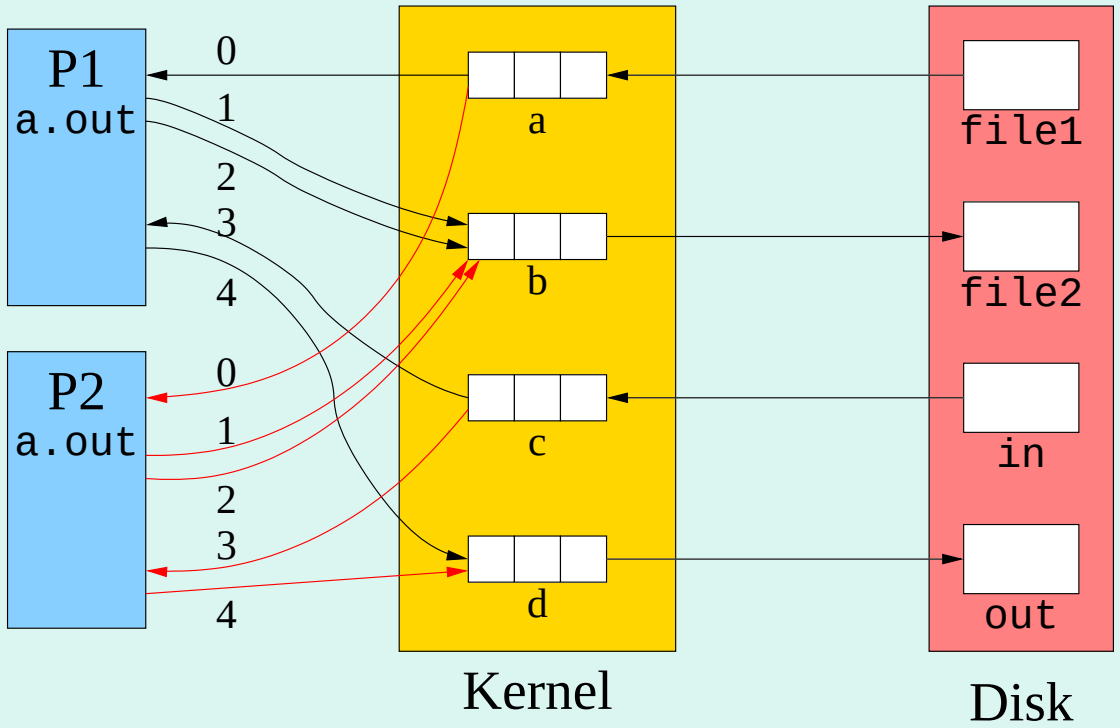
Exécution (1)

- Après les open



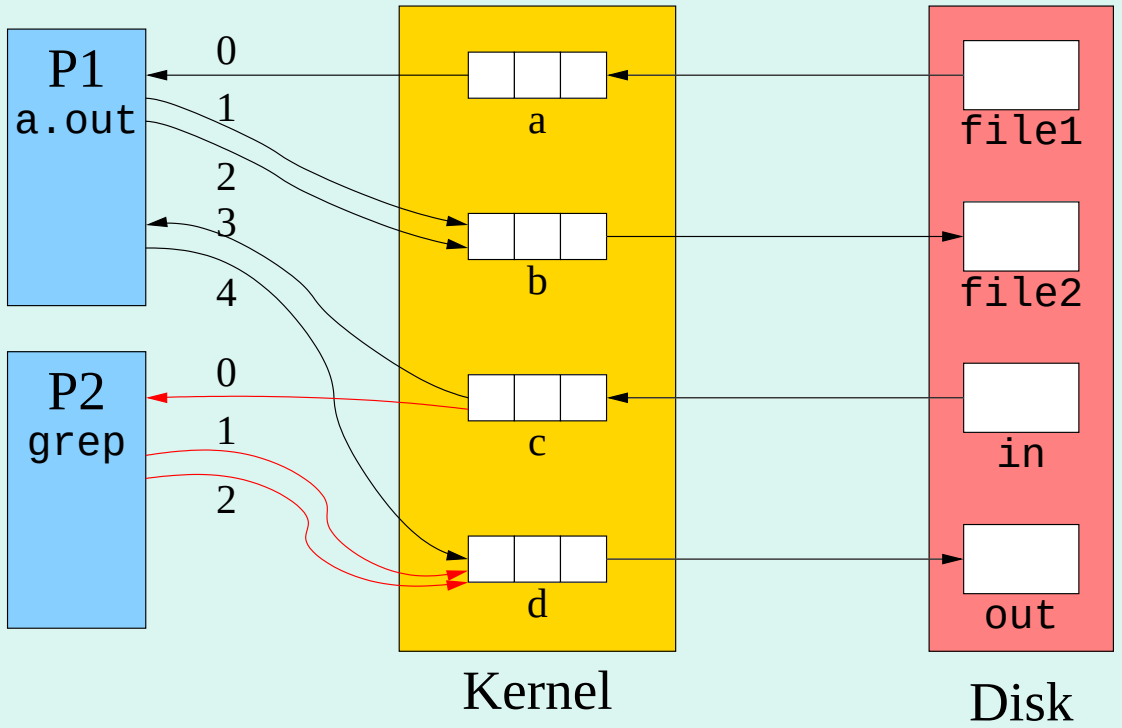
Exécution (2)

- Après le `fork`



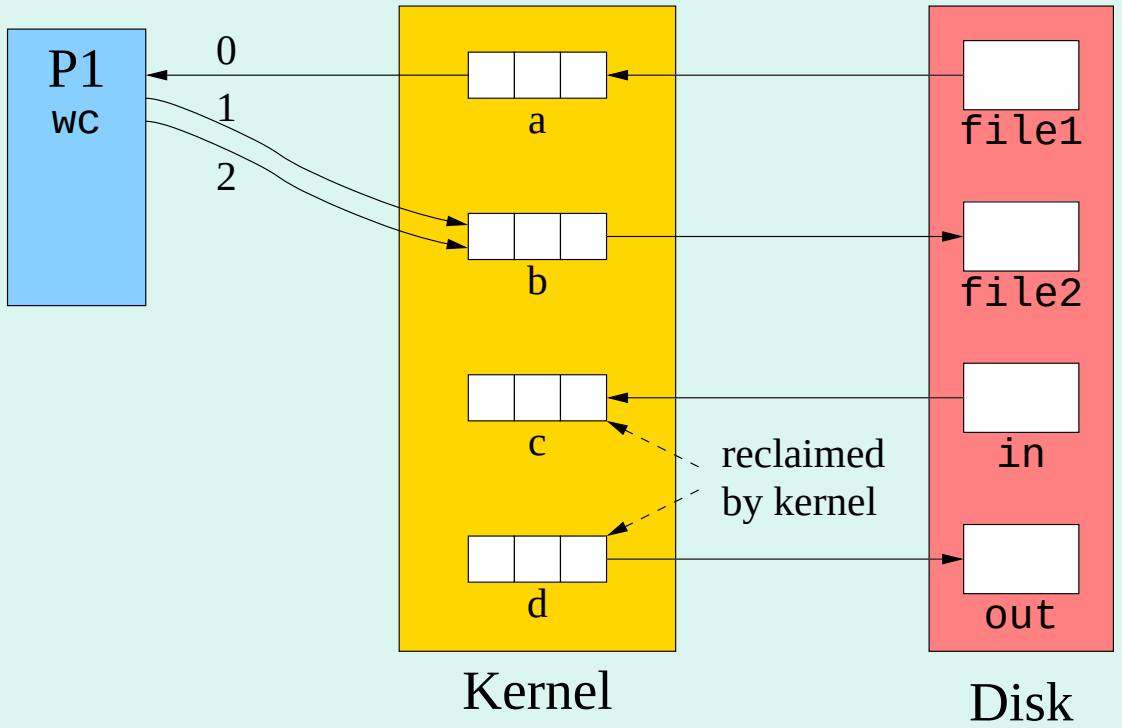
Exécution (3)

- Après les `dup2`, `close` et `execlp ("grep", ...)`



Exécution (4)

- Après les `close` et `execlp ("wc", ...)`



Les “pipes”

- Un **“pipe”** UNIX est un FIFO géré par le kernel et représenté par deux canaux: le **canal d’entrée** et le **canal de sortie**
- Les données qui sont écrites avec `write` sur le canal d’entrée sont immédiatement disponible pour être lues avec un `read` sur le canal de sortie
- L’appel système `pipe` crée un “pipe” et retourne un descripteur de fichier pour chaque canal

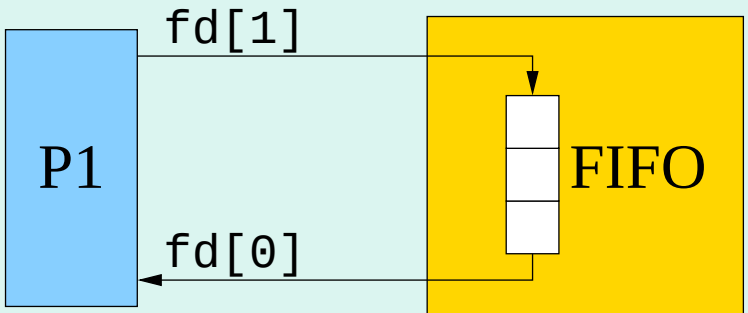
  `int pipe (int fd[2]);`

  `fd[0]` est pour le canal de sortie et `fd[1]` est pour le canal d’entrée

Exemple 1

- Écriture et lecture d'un pipe par le même processus

```
1.  int main (int argc, char* argv[])
2.  {
3.    int fd[2];
4.    char c;
5.
6.    pipe (fd);
7.
8.    write (fd[1], "allo\n", 5); // danger de bloquage
9.
10.   read (fd[0], &c, 1); cout << c;
11.   read (fd[0], &c, 1); cout << c;
12.
13.   close (fd[0]);
14.   close (fd[1]);
15.
16.   return 0;
17. }
```

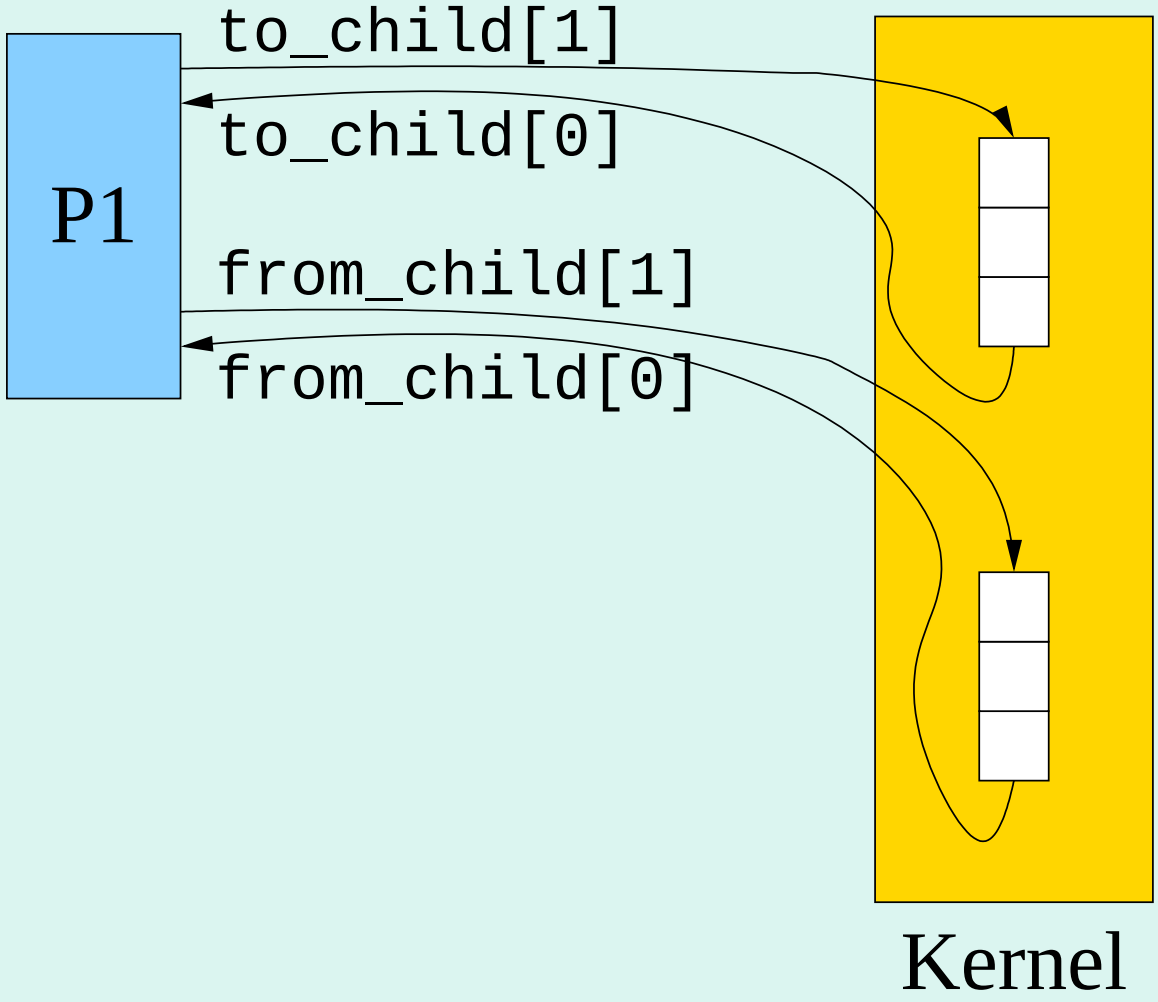


Exemple 2

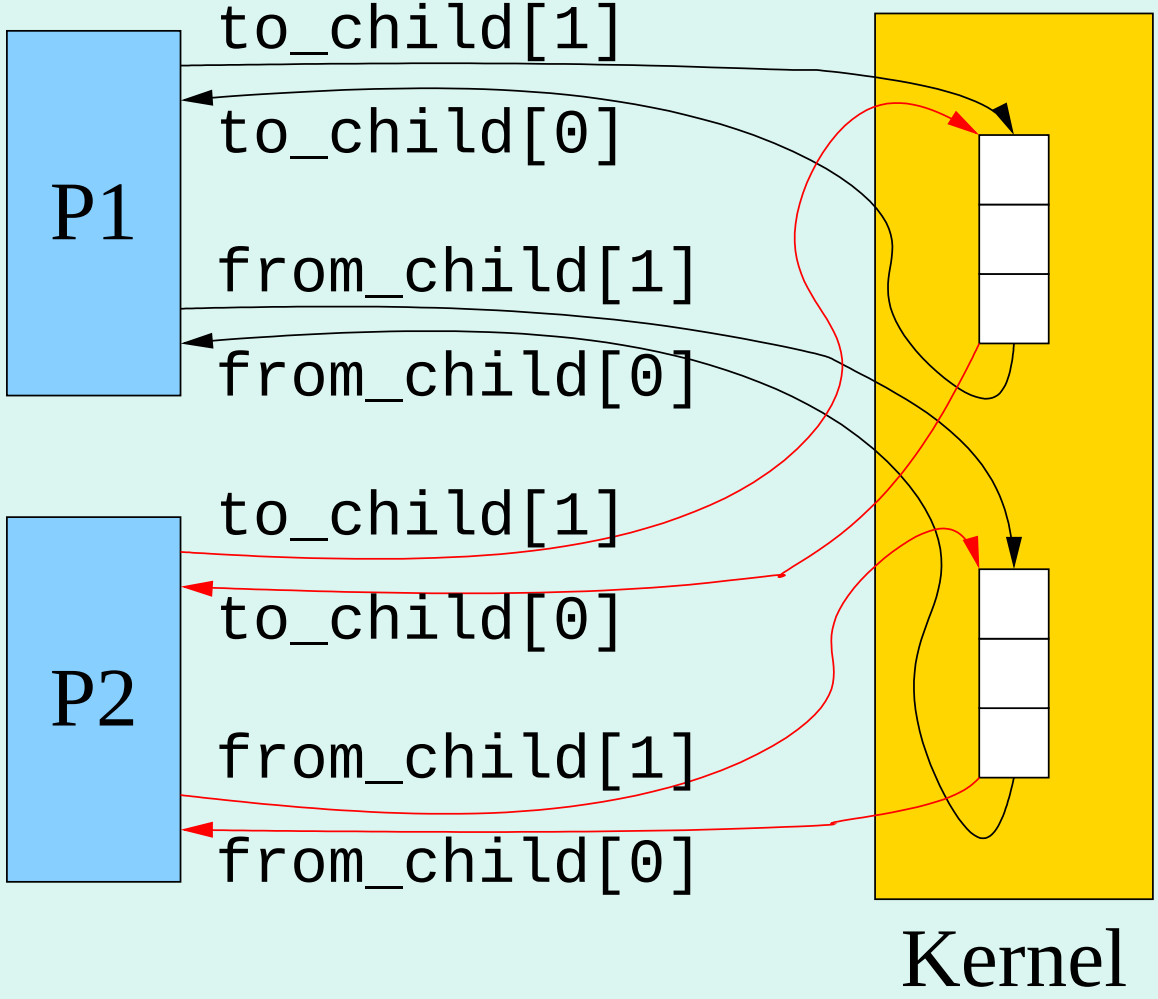
- Interaction avec un sous-processus (calculatrice bc)

```
1.  int to_child[2], from_child[2];
2.  char c;
3.
4.  pipe (to_child); pipe (from_child);
5.
6.  pid_t p = fork ();
7.
8.  if (p == 0)
9.    { dup2 (to_child[0], 0); dup2 (from_child[1], 1);
10.     close (to_child[0]); close (to_child[1]);
11.     close (from_child[0]); close (from_child[1]);
12.     execlp ("bc", "bc", NULL);
13.     return 1;
14.   }
15.
16. close (to_child[0]); close (from_child[1]);
17.
18. write (to_child[1], "2 ^ 1000\n", 9);
19.
20. close (to_child[1]);
21.
22. while (read (from_child[0], &c, 1) == 1) cout << c;
23.
24. close (from_child[0]);
```

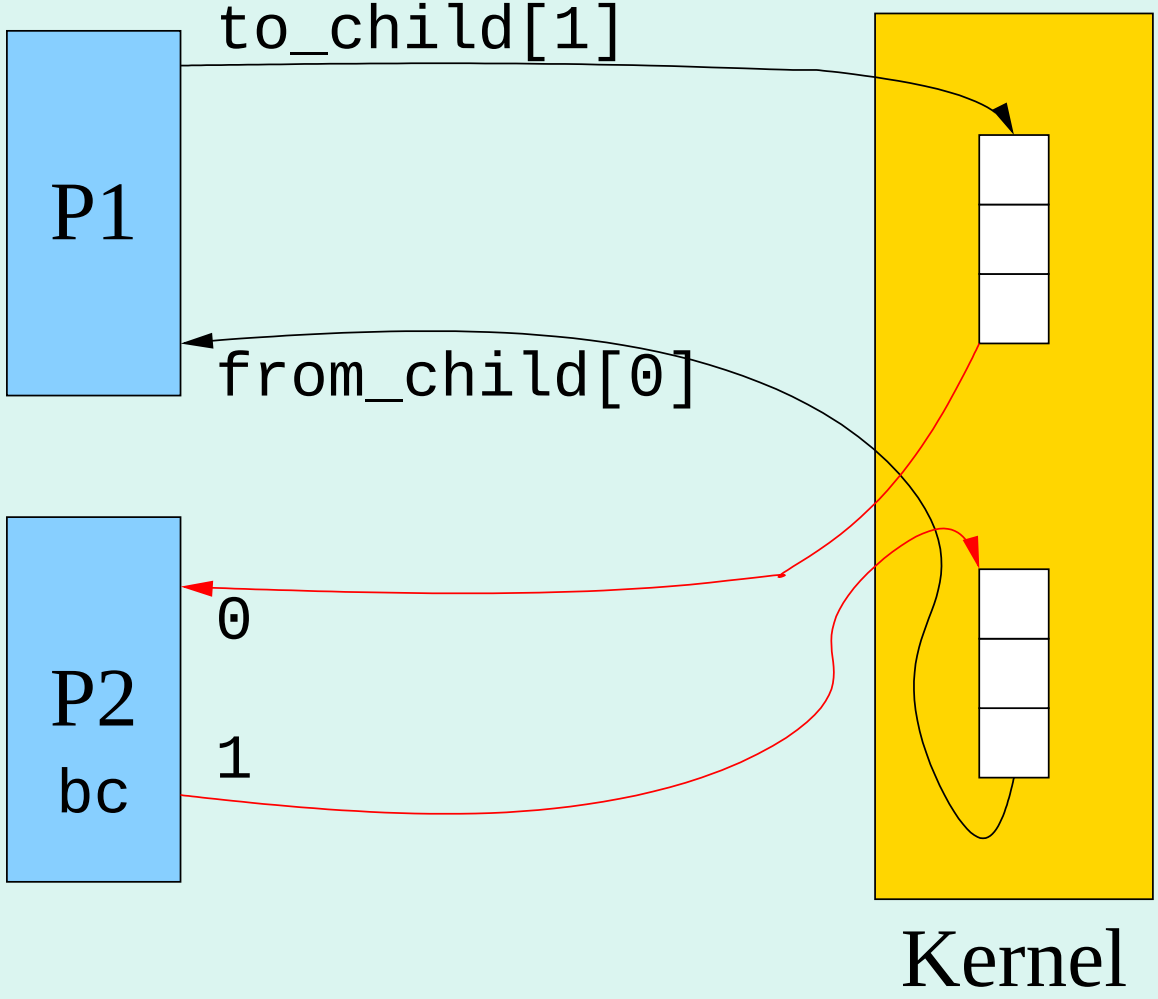
Exécution (1)



Exécution (2)



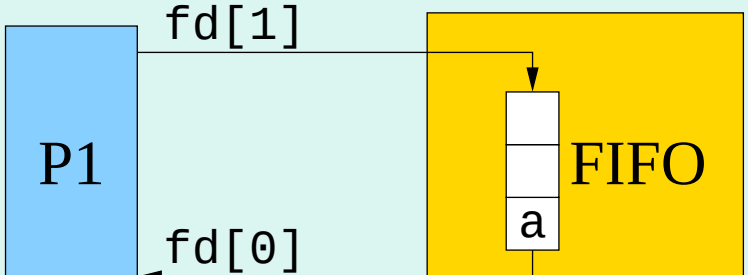
Exécution (3)



Pipes et `close` (1)

- Tant que le canal d'entrée d'un pipe est ouvert, il est possible qu'un processus y envois des données, donc les appels à `read` sur le canal de sortie n'indiquent jamais une "fin de fichier"

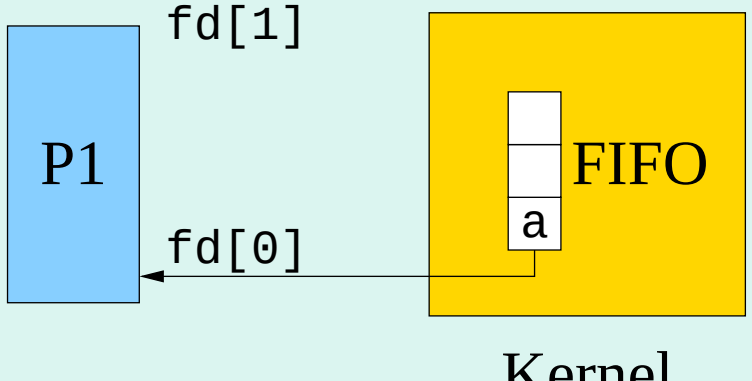
```
1.  int main (int argc, char* argv[])
2.  { int fd[2];
3.    char c;
4.    pipe (fd);
5.    write (fd[1], "a", 1);
6.    if (read (fd[0], &c, 1) == 1) // lit le ''a''
7.      cout << c << "\n";          // affiche ''a''
8.    if (read (fd[0], &c, 1) == 1) // deadlock
9.      cout << c << "\n";
10.   close (fd[0]);
11.   close (fd[1]);
12.   return 0;
13. }
```



Pipes et `close` (2)

- C'est seulement lorsque le canal d'entrée est fermé avec `close` qu'une "fin de fichier" peut être détectée au canal de sortie

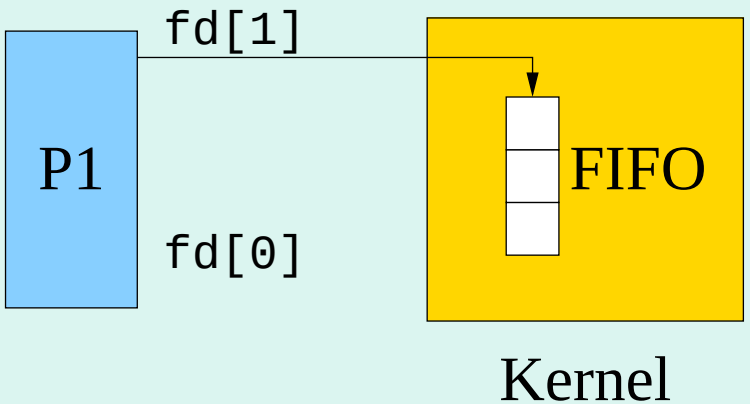
```
1.  int main (int argc, char* argv[])
2.  { int fd[2];
3.    char c;
4.    pipe (fd);
5.    write (fd[1], "a", 1);
6.    close (fd[1]);  // fermer le canal d'entrée
7.    if (read (fd[0], &c, 1) == 1) // lit le ''a''
8.      cout << c << "\n";          // affiche ''a''
9.    if (read (fd[0], &c, 1) == 1) // read retourne 0
10.     cout << c << "\n";
11.   close (fd[0]);
12.   return 0;
13. }
```



Pipes et `close` (3)

- Si le canal de sortie est fermé lorsqu'un processus fait un `write` sur le canal d'entrée, **le signal `SIGPIPE` est reçu** et (si un traiteur de signal se fait appeler) le `write` retourne une erreur avec `errno = EPIPE`

```
1.  void handler (int signum) { cout << "got SIGPIPE\n"; }
2.
3.  int main (int argc, char* argv[])
4.  { int fd[2];
5.    signal (SIGPIPE, handler);
6.    pipe (fd);
7.    close (fd[0]);  // fermer le canal de sortie
8.    if (write (fd[1], "a", 1) < 0) cout << errno << "\n"
9.    close (fd[1]);
10.   return 0;
11. }
```



Pipes et `close` (4)

- **Tous les canaux qu'un processus a d'ouvert sont fermés** par le SE lorsque le processus termine
- Il est donc **impossible d'avoir une "fuite de canal"** (la création d'un canal dont les ressources ne seront jamais récupérées par le SE)
- Le signal `SIGPIPE` et l'erreur `EPIPE` sont utiles pour détecter qu'un processus n'a pas reçu toute les données qu'on cherchait à lui envoyer, tout probablement **parce que ce processus a eu une erreur fatale**

Pipes et `close` (5)

- Cette détection est compliquée par la course qui existe entre la fin d'un `write` par le processus envoyeur et le `close` par le processus receveur
  - `write` après `close`: `SIGPIPE` et `EPIPE`
  - `write` avant `close`: pas d'erreur même si le receveur ne fait pas de `read` pour lire les données

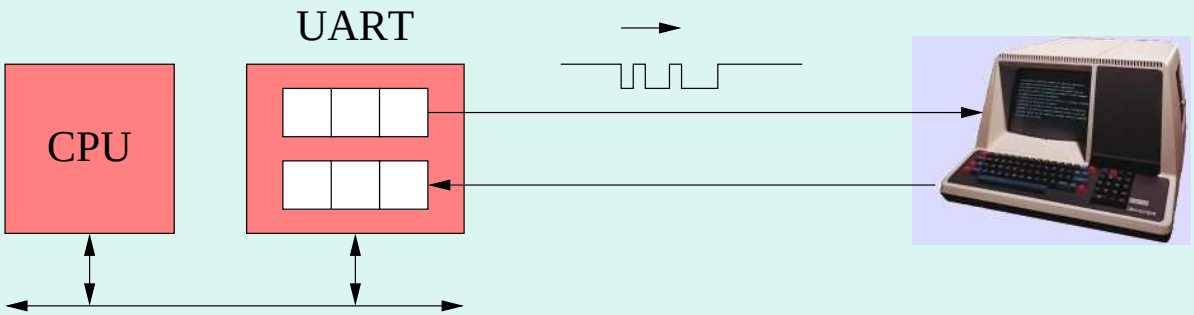
Exemple

- Il n'est pas possible de détecter l'absence de réception dans tous les cas

```
1.  int main (int argc, char* argv[])
2.  { char c;
3.    int fd[2];
4.
5.    pipe (fd);
6.
7.    if (fork () == 0)
8.      { close (fd[1]);
9.        read (fd[0], &c, 1);
10.       sleep (2);
11.       close (fd[0]);
12.     }
13.   else
14.     { close (fd[0]);
15.       write (fd[1], "a", 1);
16.       sleep (1);
17.       write (fd[1], "b", 1); // OK, mais SIGPIPE si
18.       close (fd[1]);         // on avait sleep (3)
19.     }
20.
21.   return 0;
```

Terminaux (1)

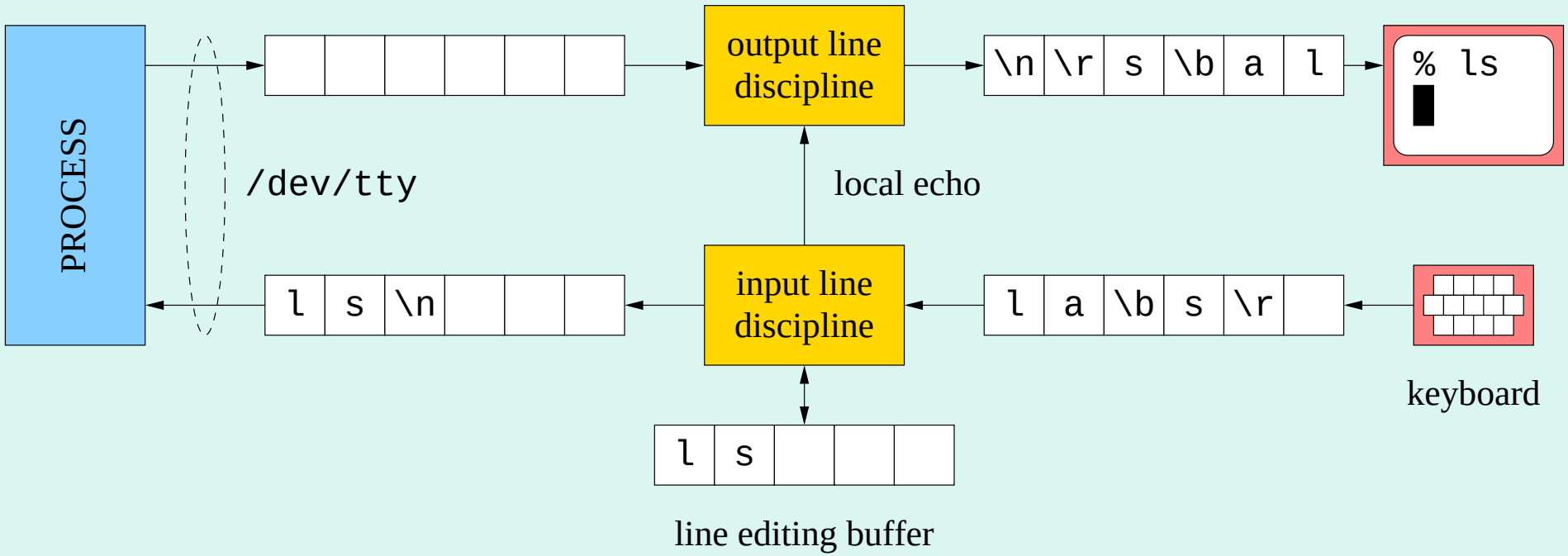
- Aux débuts de UNIX, les usagers interagissaient avec l'ordinateur en utilisant un **terminal** (ou **"tty"**) comportant un clavier et écran textuel, et une ligne sérielle RS-232 de 300 à 38400 baud pour communiquer avec l'ordinateur



- Cette même organisation est utilisée pour communiquer avec les MODEMs et imprimantes
- De nos jours la communication avec l'usager se fait avec des interfaces graphiques, mais le **concept et interface de "terminal" est resté sensiblement le même** pour les émulateurs de terminal (comme `xterm`)

Terminaux (2)

- Un **terminal** UNIX c'est un canal bidirectionnel qui véhicule une séquence d'octets (il n'y a pas de concept de position comme pour les fichiers)
- Par défaut, les E/S aux terminaux subissent un traitement interne pour permettre à l'usager d'éditer les entrées ligne-par-ligne (le **mode canonique**)



Terminaux (3)

- Sous UNIX le **terminal de contrôle** (“CTTY”) d’un processus est accessible avec

```
open ("/dev/tty", O_RDWR)
```

- Le terminal de contrôle est celui qui permet à l’usager de contrôler le groupe de processus du “foreground group” (débranchement → SIGHUP, ctrl-c → SIGINT)

```
% ./a.out < in | grep "[0-9]" | wc
<CTRL-C>
```

- Les ports sériels (typiquement branchés à des MODEMs) sont accessible via `/dev/ttyS0` et `/dev/ttyS1`

Terminaux (4)

- Souvent il est nécessaire de lire chaque caractère tappé par l'usager (incluant backspace, CTRL-C, etc) ou reçu par le MODEM **sans aucun traitement spécial** (mode "raw")

- Applications: éditeurs de texte qui répondent immédiatement qu'une touche est enfoncée, des émulateurs de terminal, des drivers d'imprimante, le protocole PPP, etc

Terminaux (5)

- Le mode du terminal se change avec un appel à `tcsetattr` en spécifiant les options de configuration de terminal
  - `ICANON` (activer mode ligne-par-ligne avec edition)
  - `ECHO` (activer echo des caractères reçus)
  - `CSIZE` et `CS5`,...,`CS8` (nombre de bits par caractère)
  - `ISIG` (caractères `INTR`, `SUSP`, etc génèrent des signaux)
  - `IXON` et `IXOFF` (activer contrôle de flux `CTRL-S`/`CTRL-Q`)
  - `ICRNL` (transformer `\r` → `\n` en entrée)
  - `OPOST` (activer les traitements de sortie, comme `\n` → `\r\n`)
  - `PARENB` et `PARODD` (générer et vérifier la parité)

```
// Attendre que l'usager tappe ''q''

#include <termios.h>

int main (int argc, char* argv[])
{
  int fd = open ("/dev/tty", O_RDWR);

  struct termios t;

  tcgetattr (fd, &t);

  t.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);
  t.c_iflag &= ~(BRKINT|ICRNL|INPCK|ISTRIP|IXON|IXOFF);
  t.c_cflag &= ~(CSIZE | PARENB);
  t.c_cflag |= CS8;
  t.c_oflag &= ~(OPOST);

  tcsetattr (fd, TCSAFLUSH, &t);

  for (;;)
    {
      char c;
      if (read (fd, &c, 1) == 1 && c == 'q')
        break;
    }

  return 0;
}
```

Emulateur de Terminal (1)

- Emulateur: term → modem, modem → term

```
1.  // emulateur de terminal minimal
2.
3.  int term = open ("/dev/tty",
4.                   O_RDWR | O_NONBLOCK);
5.  int modem = open ("/dev/ttyS0",
6.                    O_RDWR | O_NONBLOCK | O_NOCTTY);
7.
8.  struct termios t;
9.
10. tcgetattr (term, &t);
11.
12. t.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);
13. t.c_iflag &= ~(BRKINT|ICRNL|INPCK|ISTRIP|IXON|IXOFF);
14. t.c_cflag &= ~(CBAUD | CSIZE | PARENB);
15. t.c_cflag |= B38400 | CS8 | CRTSCTS | CLOCAL | CREAD;
16. t.c_oflag &= ~(OPOST);
17.
18. tcsetattr (term, TCSAFLUSH, &t);
19.
20. tcgetattr (modem, &t);
21. ...; /* <-- similaire pour le modem */
22. tcsetattr (modem, TCSAFLUSH, &t);
```

Emulateur de Terminal (2)

- Reste du programme (tentative 1)

```
23. char m2t;
24. char t2m;
25.
26. for (;;)
27.   {
28.     if (read (modem, &m2t, 1) == 1)
29.       write (term, &m2t, 1);
30.
31.     if (read (term, &t2m, 1) == 1)
32.       write (modem, &t2m, 1);
33.   }
```

- Attente active et il se peut que les `write` transfèrent 0 octets

Emulateur de Terminal (3)

- Reste du programme (tentative 2)

```
23. char m2t;
24. char t2m;
25. bool m2t_full = FALSE;
26. bool t2m_full = FALSE;
27.
28. for (;;)
29.   {
30.     if (!m2t_full)
31.       m2t_full = (read (modem, &m2t, 1) == 1);
32.
33.     if (m2t_full)
34.       m2t_full = (write (term, &m2t, 1) != 1);
35.
36.     if (!t2m_full)
37.       t2m_full = (read (term, &t2m, 1) == 1);
38.
39.     if (t2m_full)
40.       t2m_full = (write (modem, &t2m, 1) != 1);
41.   }
```

- Aucun octet n'est perdu même si un `write` retourne 0

Utilisation de `select` (1)

- Pour éviter l'attente active il serait utile de **bloquer le processus tant que toutes les opérations d'E/S sont impossibles**
- L'appel système `select` suspend le processus présent jusqu'à ce qu'une E/S soit possible sur un des canaux d'un ensemble de canaux, ou bien un temps donné s'est écoulé

```
int select(int n,
           fd_set* readfds,
           fd_set* writefds,
           fd_set* exceptfds,
           struct timeval* timeout)
```

Utilisation de `select` (2)

- L'ensemble de canaux à vérifier pour
  - la possibilité d'une **lecture** : $readfds$
  - la possibilité d'une **écriture** : $writefds$
  - un **évennement exceptionnel** : $exceptfds$
- $timeout$ est le temps maximal d'attente
- n est le descripteur le plus élevé plus 1
- La valeur de retour est le nombre de canaux sur lesquels une E/S est maintenant possible
- Le type `fd_set` est un ensemble de canaux
  - `void FD_ZERO(fd_set*` fds`);`
  - `void FD_SET(int` fd`, fd_set*` fds`);`
  - `bool FD_ISSET(int` fd`, fd_set*` fds`);`

Utilisation de `select` (3)

- Partie 1

```
23. char m2t;
24. char t2m;
25. bool m2t_full = FALSE;
26. bool t2m_full = FALSE;
27. int n;
28. fd_set readfds, writefds, exceptfds;
29.
30. for (;;)
31.   {
32.     if (!m2t_full)
33.       m2t_full = (read (modem, &m2t, 1) == 1);
34.
35.     if (m2t_full)
36.       m2t_full = (write (term, &m2t, 1) != 1);
37.
38.     if (!t2m_full)
39.       t2m_full = (read (term, &t2m, 1) == 1);
40.
41.     if (t2m_full)
42.       t2m_full = (write (modem, &t2m, 1) != 1);
```

Utilisation de `select` (4)

- Partie 2 (attente avec `select`)

```
43.   FD_ZERO(&readfds);
44.   FD_ZERO(&writefds);
45.   FD_ZERO(&exceptfds);
46.
47.   n = -1;
48.
49.   if (m2t_full)
50.     { FD_SET(term,&writefds); if (term > n) n = term; }
51.   else
52.     { FD_SET(modem,&readfds); if (modem > n) n = modem; }
53.
54.   if (t2m_full)
55.     { FD_SET(modem,&writefds); if (modem > n) n = modem;
56.   else
57.     { FD_SET(term,&readfds); if (term > n) n = term; }
58.
59.   select (n+1, &readfds, &writefds, &exceptfds, NULL);
60.
61.   //if (FD_ISSET(term,&readfds)) cout<<"[term lisible]\n"
62.   //if (FD_ISSET(modem,&readfds)) cout<<"[modem lisible]\
63. }
```

Utilisation de `select` (5)

- C'est bien plus simple si on utilise 2 processus et des canaux bloquants:

```
1.  int term = open ("/dev/tty", O_RDWR);
2.  int modem = open ("/dev/ttyS0", O_RDWR | O_NOCTTY);
3.
4.  ...
5.
6.  pid_t p = fork ();
7.
8.  char m2t;
9.  char t2m;
10.
11. if (p == 0)
12.   for (;;)
13.     { if (read (modem, &m2t, 1) == 1)
14.         while (write (term, &m2t, 1) != 1) ;
15.     }
16. else
17.   for (;;)
18.     { if (read (term, &t2m, 1) == 1)
19.         while (write (modem, &t2m, 1) != 1) ;
20.     }
```

Utilisation de `select` (6)

- `select` retourne lorsque le processus reçoit un signal (mais après l'appel du traiteur de signal)

- Pour faire attendre un processus jusqu'à ce qu'une **lecture soit possible sur le terminal** ou bien l'usager **demande d'interrompre cette lecture avec CTRL-C** le programme suivant pourrait être employé

```
bool stop = FALSE;

void handler (int signum) { stop = TRUE; }

int main (int argc, char* argv[])
{ int term = open ("/dev/tty", O_RDWR | O_NONBLOCK);

  char c;
  fd_set rfds, wfds, efds;

  signal (SIGINT, handler);

  write (term, "entrez votre choix (a, b, c): ", 31);

  for (;;)
    {
      FD_ZERO(&rfds); FD_ZERO(&wfds); FD_ZERO(&efds);
      FD_SET(term,&rfds);
      if (stop
          || select(term+1,&rfds,&wfds,&efds,NULL) == 1)
        break;
    }

  if (stop)
    cout << "usager a interrompu la lecture\n";
  else if (read (term, &c, 1) == 1)
    cout << "choix = " << c << "\n";

  return 0;
}
```

Utilisation de `select` (8)

- Il y a une **condition de course** dans ce programme qui fait qu'un `SIGINT` qui survient entre le test `stop` et l'appel à `select` sera ignoré (et donc l'usager devra tapper CTRL-C à nouveau)

- Pour éviter ce problème certaines implantation de UNIX offrent `pselect` qui change le masque de signaux **atomiquement** avec le début de l'attente

```
int pselect(int n,
            fd_set* readfds,
            fd_set* writefds,
            fd_set* exceptfds,
            struct timespec* timeout,
            sigset_t* sigmask);
```

Utilisation de `select` (9)

- Avec `pselect`:

```
1.  for (;;)
2.   {
3.    sigset_t old = sigblock(sigmask(SIGINT));
4.
5.    FD_ZERO(&rfds); FD_ZERO(&wfds); FD_ZERO(&efds);
6.    FD_SET(term,&rfds);
7.    if (stop
8.        || pselect(term+1,&rfds,&wfds,&efds,NULL,&old)==1)
9.      break;
10.  }
```

Évolution du réseau Internet (1)

- **1962** : U.S. Air Force cherche à s'assurer qu'il pourront maintenir le contrôle de leurs missiles et bombardiers après une attaque nucléaire; la RAND corp. suggère un **réseau commuté** avec un découpage des données en **paquets** contenant l'adresse source et destination; une perte de paquet causera une retransmission
- **1968** : prototype du réseau ARPANET construit par BBN et reliant 4 Universités avec des modems 50kbps
- **1969** : début des RFC ("Network Working Group Request for Comment") qui spécifient les différents protocoles réseau
- **1972-74** : premier courriel "QWERTYUIOP"; début du développement du futur TCP/IP par DARPA

Évolution du réseau Internet (2)

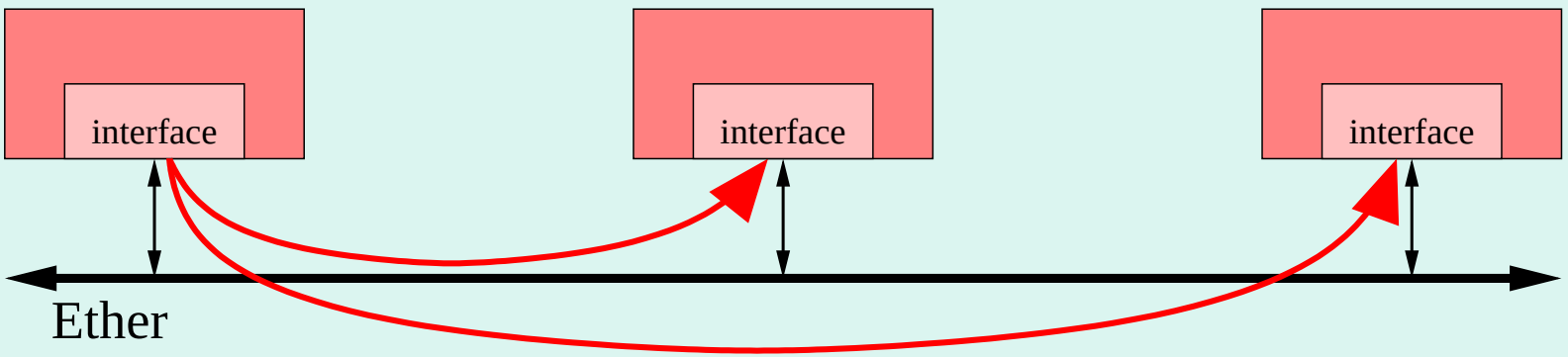
- **1976** : Xerox développe Ethernet et le premier LAN; DoD adopte TCP/IP pour l'ARPANET
- **1983** : création du Domain Name System (DNS); il y a 500 hôtes au total
- **1984** : le débit des liens passe de 56kpbs à 1.5Mbps; il y a 1000 hôtes au total
- **1988** : première attaque importante par un virus
- **1991** : HTTP et HTML; premier serveur et fureteur web
- **1996** : il y a 15,000,000 hôtes au total; début de travaux pour modifier TCP/IP pour éliminer les limites sur le nombre d'hôtes (IPv6 proposé en 1997)
- **2007** : il y a 900,000,000 hôtes au total

Sécurité et protocoles

- Les protocoles réseau ont été développé **sans souci pour la sécurité**
- Cela s'explique par le contexte de développement : il y avait peu de sites branchés au réseau, le matériel était accessible à des chercheurs seulement, et il régnait un climat de **coopération** et **confiance mutuelle**
- L'intégrité et la sécurité du réseau reposent sur **l'obéissance des protocoles**
- Pour la majorité des protocoles, rien n'empêche un fraudeur qui **envois des informations fausses** dans des buts de tromper le système (fausse identité/adresse, fausse réponse à des requêtes, ...)

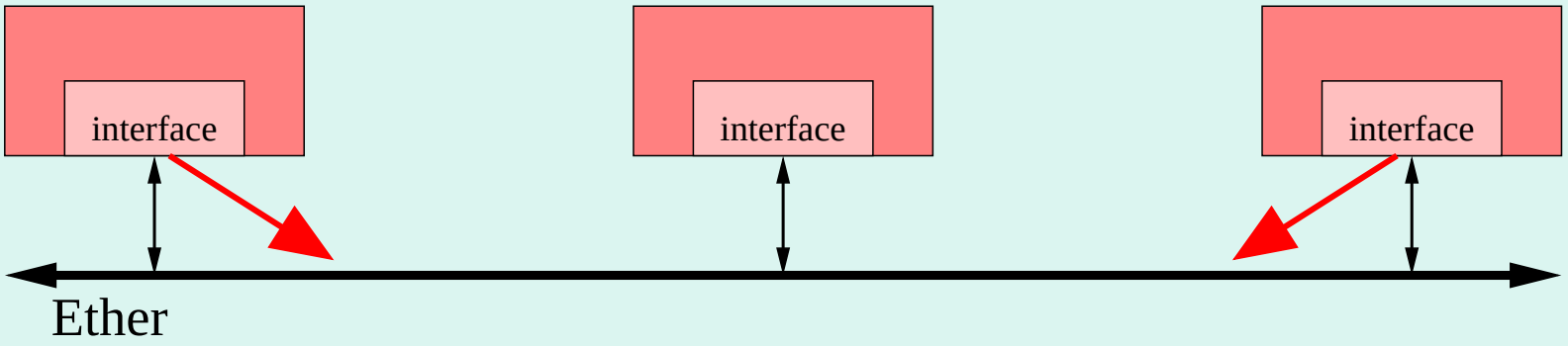
Ethernet (1)

- Ethernet est une technologie très répandue pour réaliser des **réseaux locaux** (LAN = “Local Area Network”)
- Ethernet est essentiellement un **bus sériel** partagé par toutes les **interfaces réseau** du LAN
- Une seule **trame** (“frame”) peut être transmise sur le bus à un moment donné et cette trame est reçue par toutes les interfaces du LAN



Ethernet (2)

- Si deux interfaces envoient des trames sur le réseau en même temps il y a une **collision**

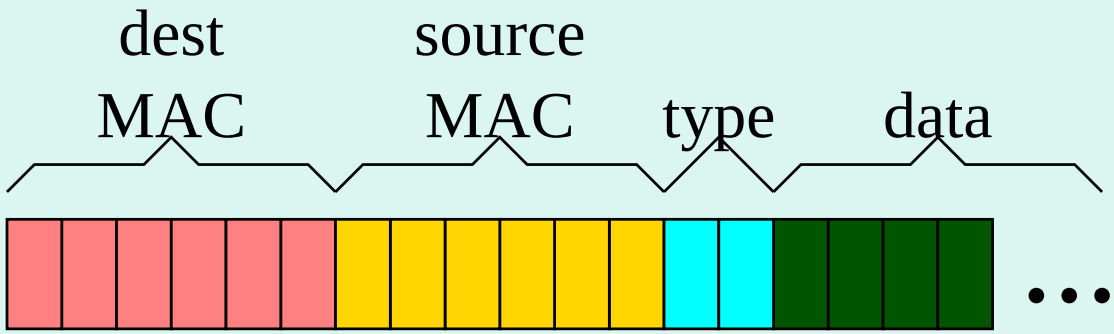


- Grâce au **protocole CSMA/CD** (“Carrier Sense Multiple Access with Collision Detection”) chaque interface détecte la collision (et la renforce avec un “JAM”)

- La transmission est tentée à nouveau après un certain **délai aléatoire** (grandissant à chaque nouvelle collision jusqu’à 16 collisions; “exponential backoff”)

Ethernet (3)

- Chaque interface réseau a une **adresse MAC** (“Media Access Control address”) **globalement unique** de 48 bits qui l’identifie (de nos jours on peut normalement reconfigurer l’adresse MAC des interfaces réseau)
- Chaque trame contient l’**adresse MAC de l’interface réseau destinataire**



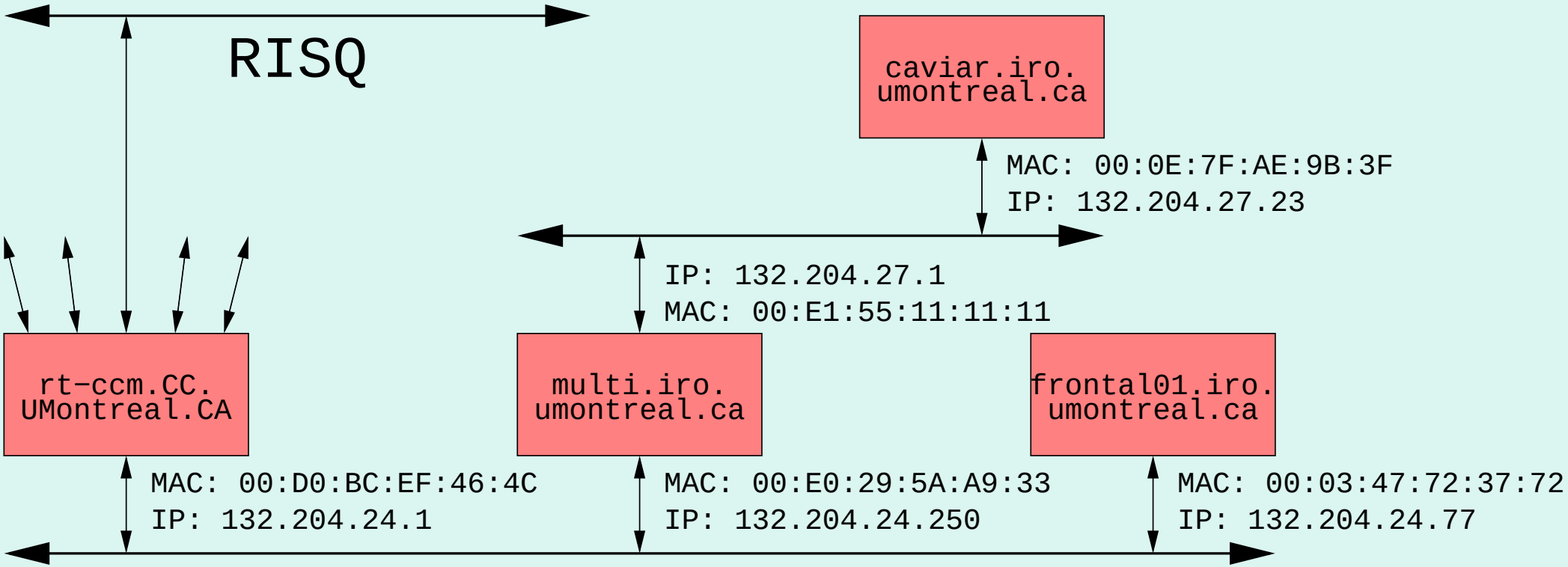
- Ethernet permet uniquement d’échanger des trames directement avec des machines **sur le même LAN**

Ethernet (4)

- Il y a plusieurs modes de transmission des trames :
  - **“unicast”** : 1 destinataire
  - **“multicast”** : plusieurs destinataires
  - **“broadcast”** : toutes les interfaces sur le LAN (adresse de destination = FF:FF:FF:FF:FF:FF)

Internet (1)

- L'Internet est un **réseau de réseaux locaux**
- Ces LANs sont interconnectés par des "passerelles" d'un LAN à un autre, par exemple une machine "multi-home" avec deux interfaces réseau

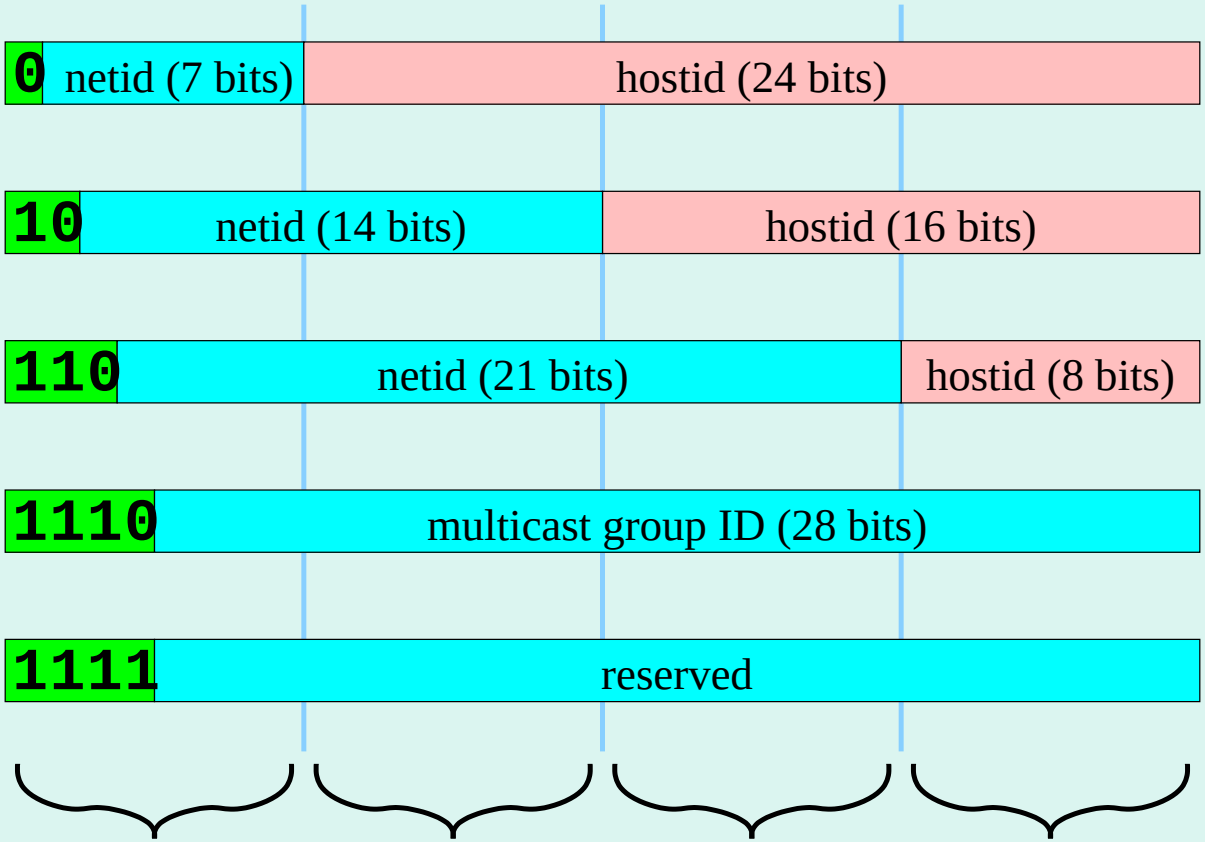


Internet (2)

- Le **protocole IP** (“Internet Protocol”) permet à des machines sur des LANs différents de communiquer
- En plus d’une adresse MAC (pour identifier une interface réseau sur le LAN), **chaque interface a une adresse IP** qui l’identifie
- L’IANA (Internet Assigned Numbers Authority) est l’organisme qui contrôle le système de numérotation de l’Internet (format des adresses, numéro de services standard, etc)
- Les adresses IPv4 ont 32 bits; les adresses IPv6 en ont 128

Adresses IPv4 (1)

- Adresse IP = 32 bits, dans une de 5 classes :



Class A
0.0.0.0 – 127.255.255.255

Class B
128.0.0.0 – 191.255.255.255

Class C
192.0.0.0 – 223.255.255.255

Class D
224.0.0.0 – 239.255.255.255

Class E
240.0.0.0 – 255.255.255.255

132 . 204 . 24 . 77 = frontal01.iro.umontreal.ca

- Cette organisation des adresses permet de faire le routage efficacement

Adresses IPv4 (2)

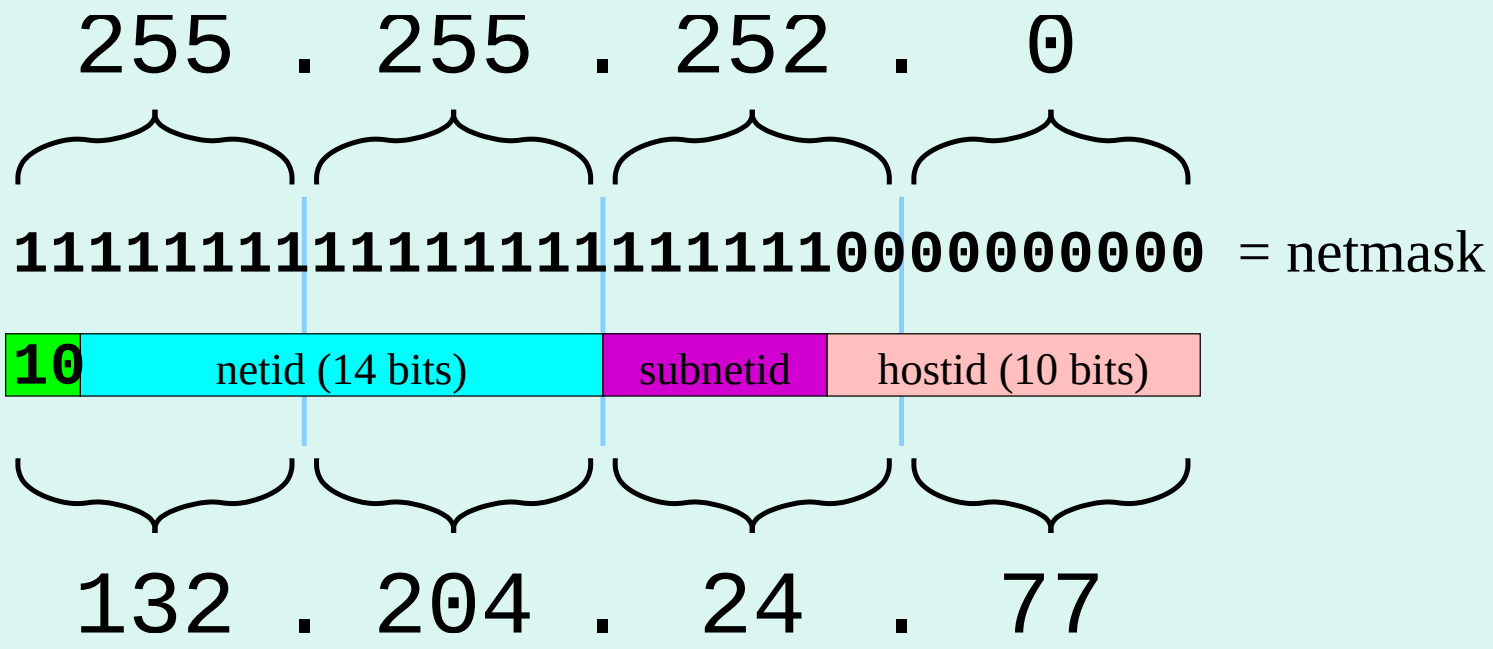
- L'INTERNIC (Internet Network Information Center) assigne les identificateurs réseau (**netid**) aux institutions (universités, banques, compagnies, ...)
- La classe A pour les très grandes institutions, la classe B pour les institutions de taille moyenne, et la classe C pour les petites institutions
- P.ex. l'Université de Montréal a des adresses de classe B (132.204.*.*), ce qui permet d'identifier 65534 hôtes
- Le **netid** est utilisé lors du **routage global** d'un paquet pour l'acheminer à l'institution destinataire (ainsi tout paquet destiné à une adresse 132.204.*.* sera acheminé à l'Université de Montréal)

Adresses IPv4 (3)

- Chaque institution est responsable de l'assignation des identificateurs de ses hôtes (**hostid**)
- Pour permettre aux institutions d'organiser ses hôtes en sous-réseaux locaux (pour des raisons de performance, de sécurité, de facilité de gestion, ...) le champ **hostid** contient le numéro du sous-réseau et le numéro de l'hôte
- C'est le **netmask**, un entier 32 bits, qui précise où est la séparation entre le numéro du sous-réseau et le numéro de l'hôte (les bits 0 en bas du netmask = champ numéro de l'hôte)

Adresses IPv4 (4)

- Deux numéros d'hôte sont réservés : tout à 1 = "all subnets broadcast", tout à 0
- P.ex. le netmask utilisé au DIRO est 255.255.252.0 (i.e. les 10 bits du bas sont 0), donc 1022 hôtes sont possible sur le sous-réseau du DIRO



Adresses IPv4 (5)

- Étant donné une adresse IP X, l'adresse IP locale L et le netmask N il est possible de déterminer facilement la relation entre X et L :

1. X **et** L **sur le même sous-réseau** : $X\&N = L\&N$
2. X **et** L **sur le même réseau** :
   (a) $L \in$ Classe A : X&0xff000000 $= L$&0xff000000
   (b) $L \in$ Classe B : X&0xffff0000 $= L$&0xffff0000
   (c) $L \in$ Classe C : X&0xffffff00 $= L$&0xffffff00
3. X **et** L **sur des réseaux différents** : sinon

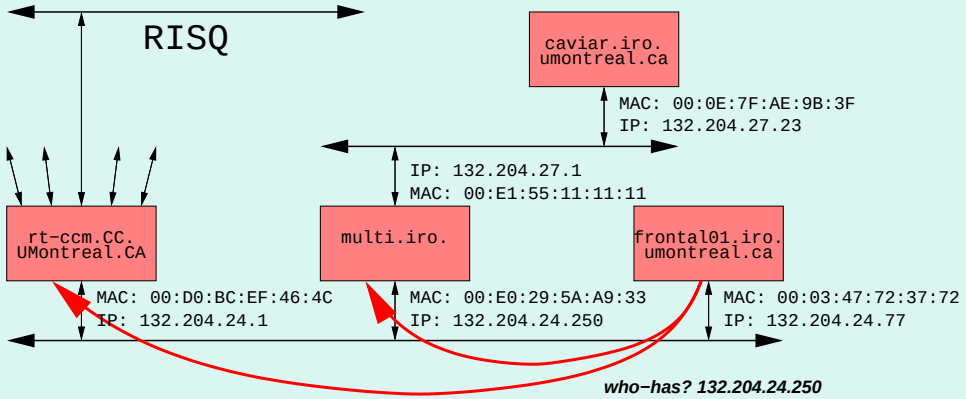
ifconfig

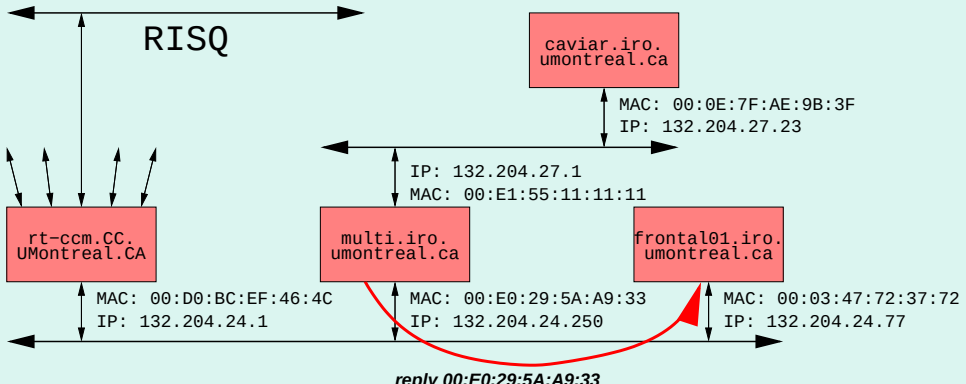
- La configuration d'une interface réseau se fait avec `ifconfig` (UNIX) et `ipconfig` (Windows)

```
% hostname
frontal01.iro.umontreal.ca
% ifconfig eth0 132.204.24.77 netmask 255.255.252.0
% ifconfig
eth0  Link encap:Ethernet  HWaddr 00:03:47:72:37:72
      inet addr:132.204.24.77  Bcast:132.204.27.255  Mask:255.255.252.0
      inet6 addr: fe80::203:47ff:fe72:3772/64 Scope:Link
      UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
      RX packets:201472593 errors:0 dropped:0 overruns:0 frame:0
      TX packets:207917527 errors:0 dropped:0 overruns:0 carrier:201681730
      collisions:0 txqueuelen:1000
      RX bytes:754551201 (719.5 MiB)  TX bytes:309461130 (295.1 MiB)
      Memory:f4020000-f4040000

lo    Link encap:Local Loopback
      inet addr:127.0.0.1  Mask:255.0.0.0
      inet6 addr: ::1/128 Scope:Host
      UP LOOPBACK RUNNING  MTU:16436  Metric:1
      RX packets:4856294 errors:0 dropped:0 overruns:0 frame:0
      TX packets:4856294 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:0
      RX bytes:3080931508 (2.8 GiB)  TX bytes:3080931508 (2.8 GiB)
```

Protocoles ARP (1)

- Le **protocole ARP** (“Address Resolution Protocol”) permet de trouver sur le LAN l’adresse MAC correspondant à une adresse IP: une trame contenant l’adresse IP est **diffusée à toutes les machine sur le LAN**, et la machine qui a cet IP repond





Protocoles ARP (2)

- Si cette adresse IP est sur un autre LAN, aucune réponse n'est reçue et la machine dirigera les datagrammes (paquets IP) destinés à cette adresse IP **vers une passerelle**

- Pour éviter des requêtes ARP trop fréquentes, chaque machine garde une **cache des réponses récentes** (dernières 20 minutes)

- L'utilitaire UNIX `arp` permet de consulter la cache :

```
% arp -an
? (132.204.24.1) at 0:d0:bc:ef:46:4c on eth0 [ethernet]
? (132.204.24.45) at 0:2:b3:a4:22:3f on eth0 [ethernet]
? (132.204.24.77) at 0:3:47:72:37:72 on eth0 [ethernet]
? (132.204.24.179) at 0:e0:18:f3:ad:1f on eth0 [ethernet]
```

Protocoles RARP

- Le **protocole RARP** (“Reverse ARP”) permet de trouver l’adresse IP qui correspond à une certaine adresse MAC (diffusion de l’adresse MAC et la machine serveur RARP qui connait l’adresse IP réponds)

- C’est utile pour trouver sa propre adresse IP pour les machines sans mémoire permanente (“diskless client”)

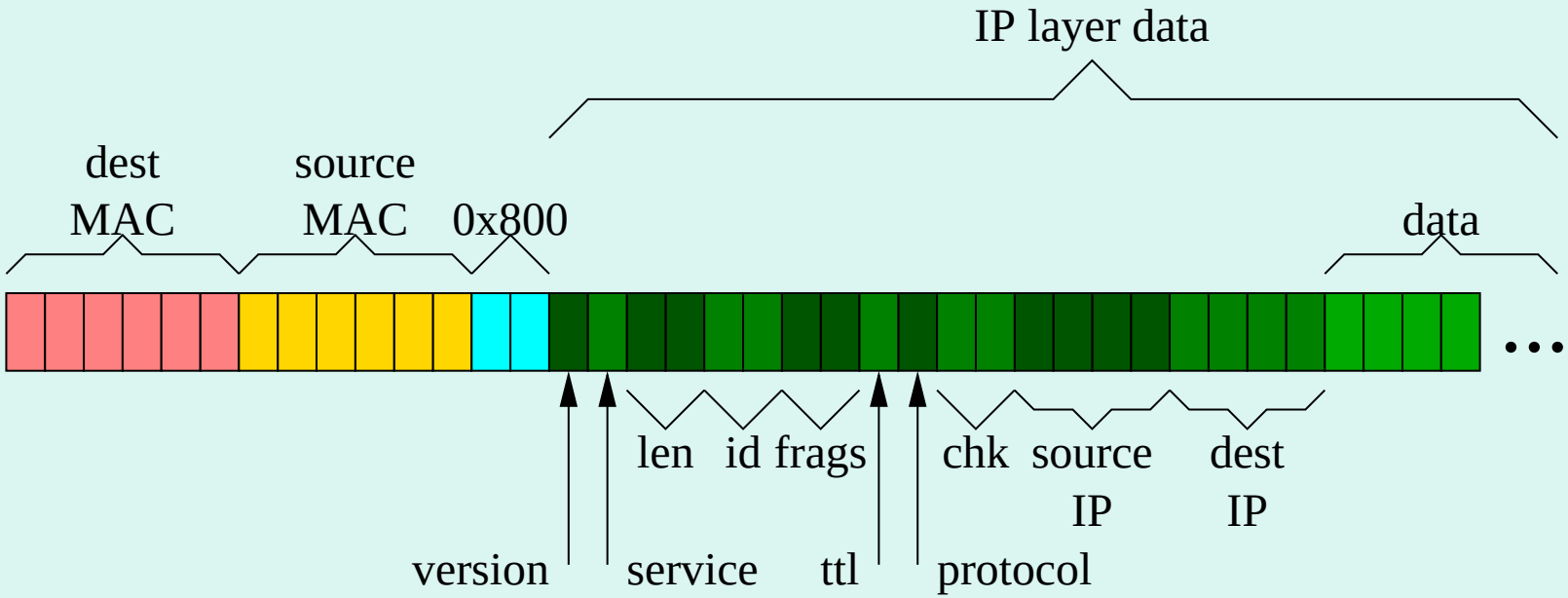
Noms symboliques

- Pour la convivialité, les machines portent des noms symboliques (comme `frontal01.iro.umontreal.ca`) et il faut convertir ces noms en adresse IP (avec une **table statique** locale (`/etc/hosts`) ou bien en contactant un serveur de nom “DNS” (`/etc/resolv.conf`))

```
# /etc/hosts
127.0.0.1         localhost
132.204.24.45     pegase.iro.umontreal.ca pegase
132.204.24.70     odin.iro.umontreal.ca odin loghost
132.204.24.77     frontal01.iro.umontreal.ca frontal01
132.204.24.78     frontal02.iro.umontreal.ca frontal02
132.204.24.79     frontal03.iro.umontreal.ca frontal03
132.204.24.80     frontal.iro.umontreal.ca frontal
132.204.24.179    himalia.iro.umontreal.ca himalia www

# /etc/resolv.conf
search iro.umontreal.ca umontreal.ca
nameserver 132.204.24.45
nameserver 132.204.24.70
```

Protocole IP

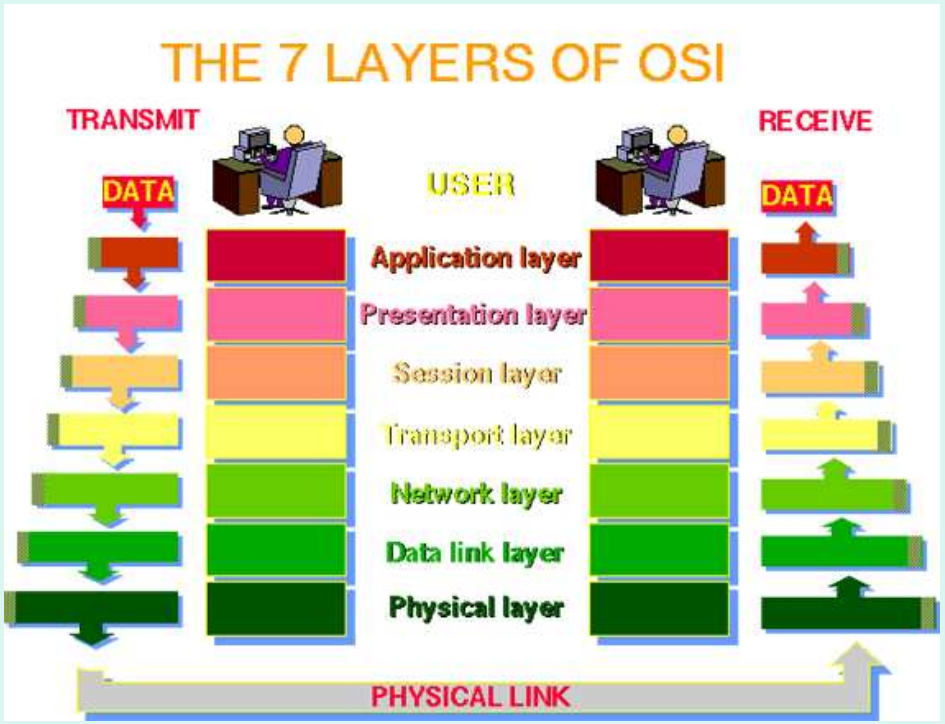
- Format d'un datagramme dans le protocole IP :



- Si la machine A cherche à envoyer un datagramme à B sur un autre LAN, le datagramme est envoyé à l'adresse MAC d'une **passerelle** qui fera le **routage** du datagramme vers l'adresse IP destinataire (l'adresse MAC destinataire change à chaque étape de routage jusqu'à ce que le "Time To Live" (TTL) = 0)

Modèle OSI

- De façon générale chaque protocole de communication est **bâti sur un protocole plus simple** (par exemple, IP repose sur un protocole matériel comme Ethernet)
- Les paquets d'un certain niveau de protocole se font **encapsuler** dans des paquets de niveau inférieur
- Le modèle "Open System Interconnect" a 7 niveaux:



Protocoles principaux

- Les couches de protocoles principales
  - **Ethernet**
    - **ARP** (Address Resolution Protocol)
    - **RARP** (Reverse Address Resolution Protocol)
    - **IP** (Internet Protocol)
      - **ICMP** (Internet Control Message Protocol)
      - **IGMP** (Internet Group Management Protocol)
      - **UDP** (User Datagram Protocol)
      - **TCP** (Transmission Control Protocol)

Protocole ICMP

- Permet de communiquer des messages d'erreur et des information de gestion du réseau
- Utilisé par les utilitaires `ping` et `traceroute`
- `ping` permet de vérifier qu'une interface réseau est fonctionnelle et atteignable

```
% ping 132.204.24.77
PING 132.204.24.77 (132.204.24.77): 56 data bytes
64 bytes from 132.204.24.77: icmp_seq=0 ttl=64 time=0.27
64 bytes from 132.204.24.77: icmp_seq=1 ttl=64 time=0.19
64 bytes from 132.204.24.77: icmp_seq=2 ttl=64 time=0.24
```

- `traceroute` permet de connaître le chemin qu'emprunte un paquet pour se rendre à sa destination

Protocole IGMP

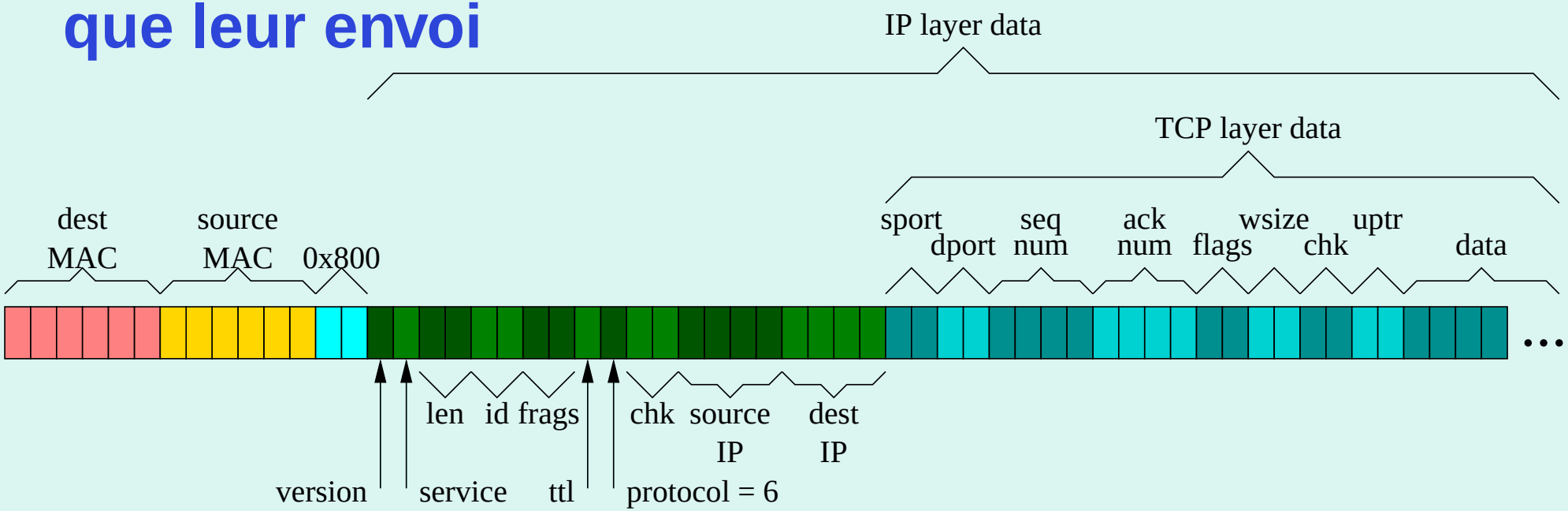
- Le protocole IGMP est utilisé pour la **gestion des groupes de multicast**
- Une machine peut créer une adresse de multicast, et peut s'y joindre
- Tout datagramme envoyé à l'adresse multicast sera envoyée à toutes les machines qui s'y sont joint
- Utile pour les applications multimédia (diffusion de vidéo, audio, etc)

Protocole UDP

- UDP = **User Datagram Protocol**
- Les données sont véhiculées par des datagrammes de **longueur bornée** (64KBytes)
- Les datagrammes envoyés **ne sont pas nécessairement reçus** par le destinataire et UDP ne détecte pas la perte de datagrammes
- Le destinataire peut recevoir les datagrammes **dans un ordre différent de leur envoi**
- Utile lorsque l'information envoyée est courte et devient périmée rapidement

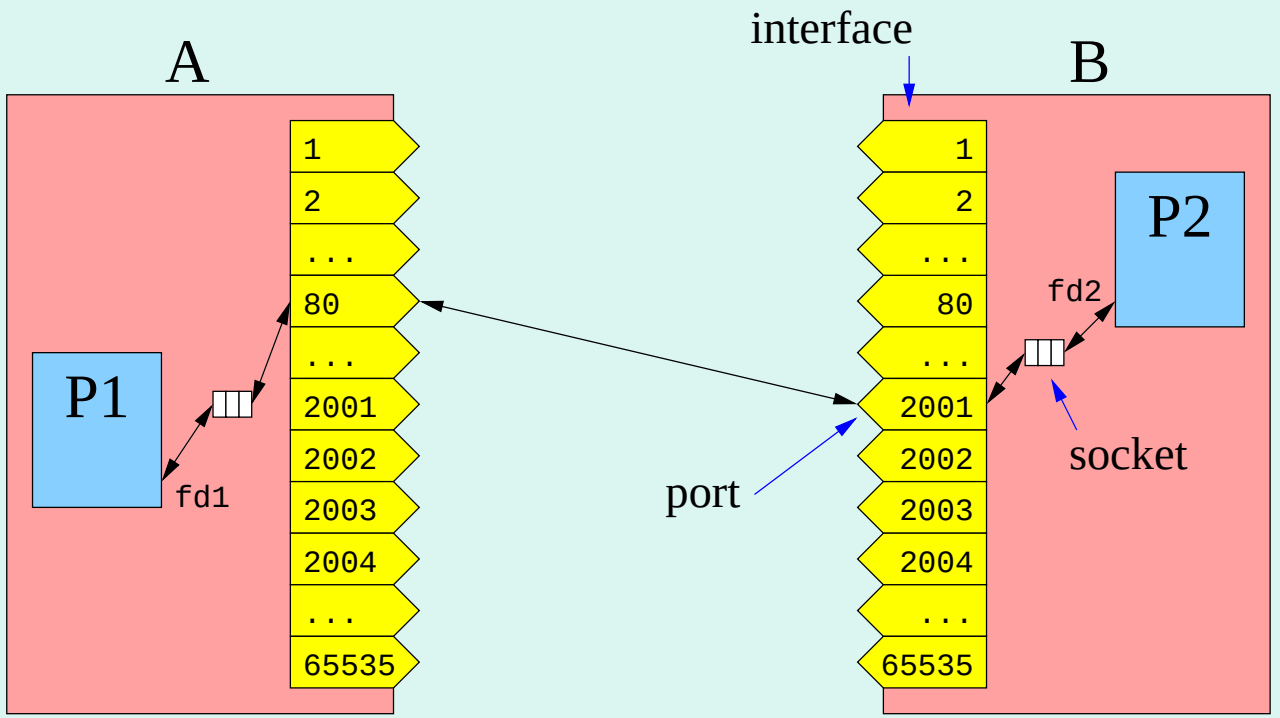
Protocole TCP

- Concept d'établissement et de fermeture de **connexion** (bien adapté au modèle client-serveur)
- Les données sont sous la forme d'une séquence d'octets **de longueur quelconque** (comme les pipes)
- Utilisation d'**accusés de réception** pour retransmettre les paquets (possiblement) perdus
- Le destinataire lit les données **dans le même ordre que leur envoi**



Connexions et Ports (1)

- Avec TCP, chaque connexion est identifiée par les adresses IP des 2 interfaces réseau (sur les machines A et B) et les numéros de **ports** sur ces interfaces
- C'est le **"socket pair"**




```
{ 132.204.24.95/80 , 155.1.1.3/2001 }
```

Connexions et Ports (2)

- La connexion doit être **établie** avant d'échanger des informations
- Normalement le processus serveur **écoute sur un port préétabli** (par exemple 80 pour HTTP, voir `/etc/services`) et le client **demande d'ouvrir une connexion** sur ce port sur cette machine
- À l'établissement de la connexion, un port (éphémère) libre est assigné sur chaque machine et ces ports seront seulement libérés à la fermeture de la connexion
- **1-1023** = "well known" ports (services prédéfinis)
- **1024-5000** = ports éphémères (pour connexions établies)
- **>5000** = ports disponibles aux usagers pour serveurs divers

La Routine `socket`

- Du point de vue d'un processus, une connexion est représentée par un **socket** (identifiée avec un descripteur de fichier) sur lequel les opérations `read` et `write` sont possibles

  `int socket (int` $domain$`, int` $type$`, int` $protocol$`);`

  Pour IP: $domain$ = `PF_INET`
  Pour TCP: $type$ = `SOCK_STREAM` et $protocol$ = 0

- La routine `socket` crée un nouveau socket **qui n'est pas initialement associé à une connexion**

La Routine connect

- Pour établir une connexion, le client utilise connect

```
int connect (int fd,
             struct sockaddr* addr,
             int len);
```

addr = **l'adresse IP et le port** d'écoute du serveur
len = la longueur de la structure *addr*

```
1. struct in_addr { in_addr_t s_addr; }; // 32 bits
2.
3. struct sockaddr_in {
4.  uint8_t         sin_len;    // = 16
5.  sa_family_t     sin_family; // = AF_INET
6.  in_port_t       sin_port;   // 16 bits
7.  struct in_addr  sin_addr;   // 32 bits
8.  char            sin_zero[8];
9. };
```

Les Routines `bind`, `listen`, `accept`

- Le serveur se sert des routines `bind` (réserver un port), `listen` (commencer à écouter pour des connexions) et `accept` (attendre la prochaine connexion et la retourner)

```
int bind (int fd,
          struct sockaddr* addr,
          int len);

int listen (int fd, int backlog);

int accept (int fd,
            struct sockaddr* addr,
            int len);
```

Client HTTP Simple (1)


```
1.  struct sockaddr_in server;
2.  char buf[301];
3.  int n;
4.  int fd = socket (PF_INET, SOCK_STREAM, 0);
5.
6.  server.sin_family = PF_INET;
7.  server.sin_port = htons (80);
8.  server.sin_addr.s_addr =
9.    htonl ((132<<24)+(204<<16)+(24<<8)+179);
10.
11. connect (fd, (struct sockaddr*)&server, sizeof (server));
12.
13. write (fd, "GET /\r\n", 7);
14.
15. if ((n = read (fd, buf, 300)) >= 0)
16.   {
17.     buf[n] = '\0';
18.     cout << buf << "\n";
19.   }
20.
21. close (fd);
```

Client HTTP Simple (2)


```
% ./client
<HTML>
<HEAD>
<TITLE>IRO : Page d'accueil</TITLE>
</HEAD>

<BODY LINK="075E94" VLINK="#032F4A" ALINK="#0EBCFF"
BGCOLOR="#FFFFFF">

<TABLE BORDER=0>
    <TR VALIGN=CENTER>
        <TD><A HREF="http://www.umontreal.ca"><IMG
SRC="/images/logo-UdeM.gif" WIDTH=122 HEIGHT=64
             ALT="Un
%
```

Serveur Concurrent Simple (1)

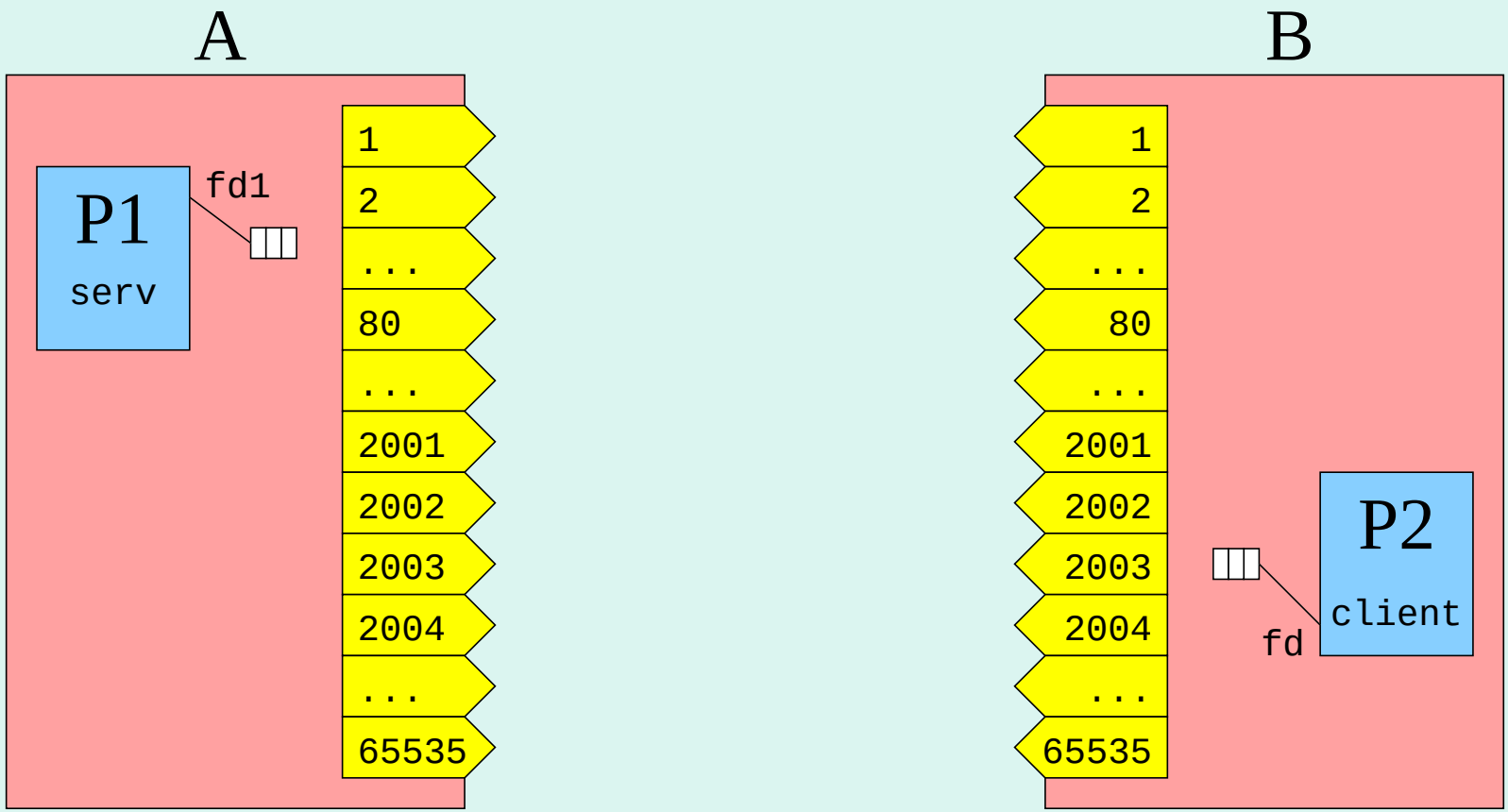

```
1.  struct sockaddr_in addr;
2.  bool x = TRUE;
3.  int fd1 = socket (PF_INET, SOCK_STREAM, 0);
4.
5.  addr.sin_family = PF_INET;
6.  addr.sin_port = htons (80);
7.  addr.sin_addr.s_addr = htonl (INADDR_ANY);
8.
9.  bind (fd1, (struct sockaddr*)&addr, sizeof (addr));
10. listen (fd1, 5);
11.
12. for (;;)
13.   { struct sockaddr client;
14.     int client_len = sizeof (client);
15.     int fd2 = accept (fd1, &client, &client_len);
16.     if (fork () == 0)
17.       { close (fd1);
18.         sleep (5);
19.         write (fd2, x ? "hello\n" : "world\n", 6);
20.         close (fd2);
21.         exit (0);
22.       }
23.     close (fd2);
24.     x = !x;
25.   }
```

Serveur Concurrent Simple (2)


```
% ./serv &
[12] 5880
% telnet localhost 80
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hello
Connection closed by foreign host.
% telnet localhost 80
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
world
Connection closed by foreign host.
% ./client &
[13] 5890
% ./client &
[14] 5892
% hello
world
```

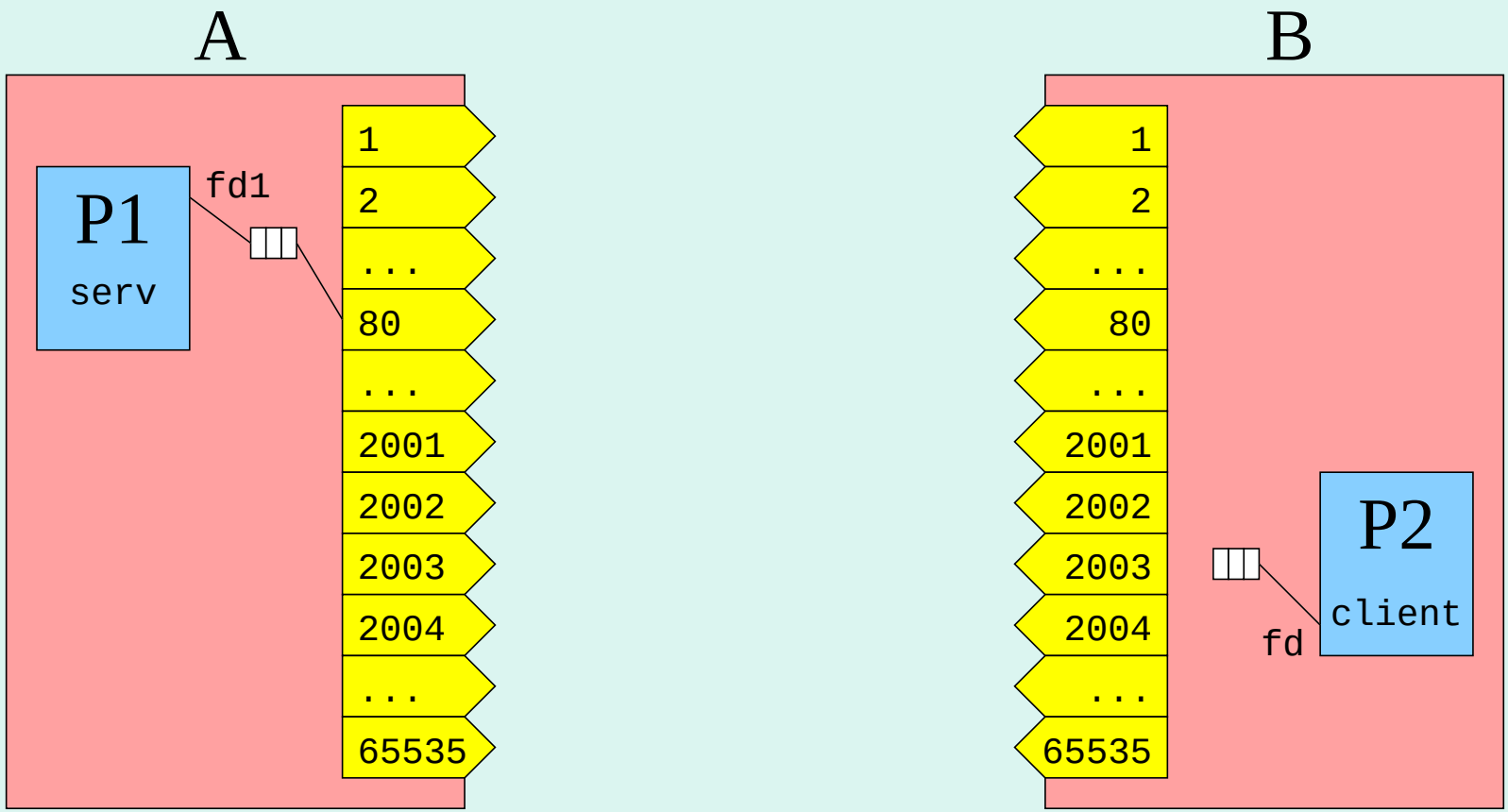
Exécution (1)

- Après `fd1 = socket(...)` et `fd = socket(...)`



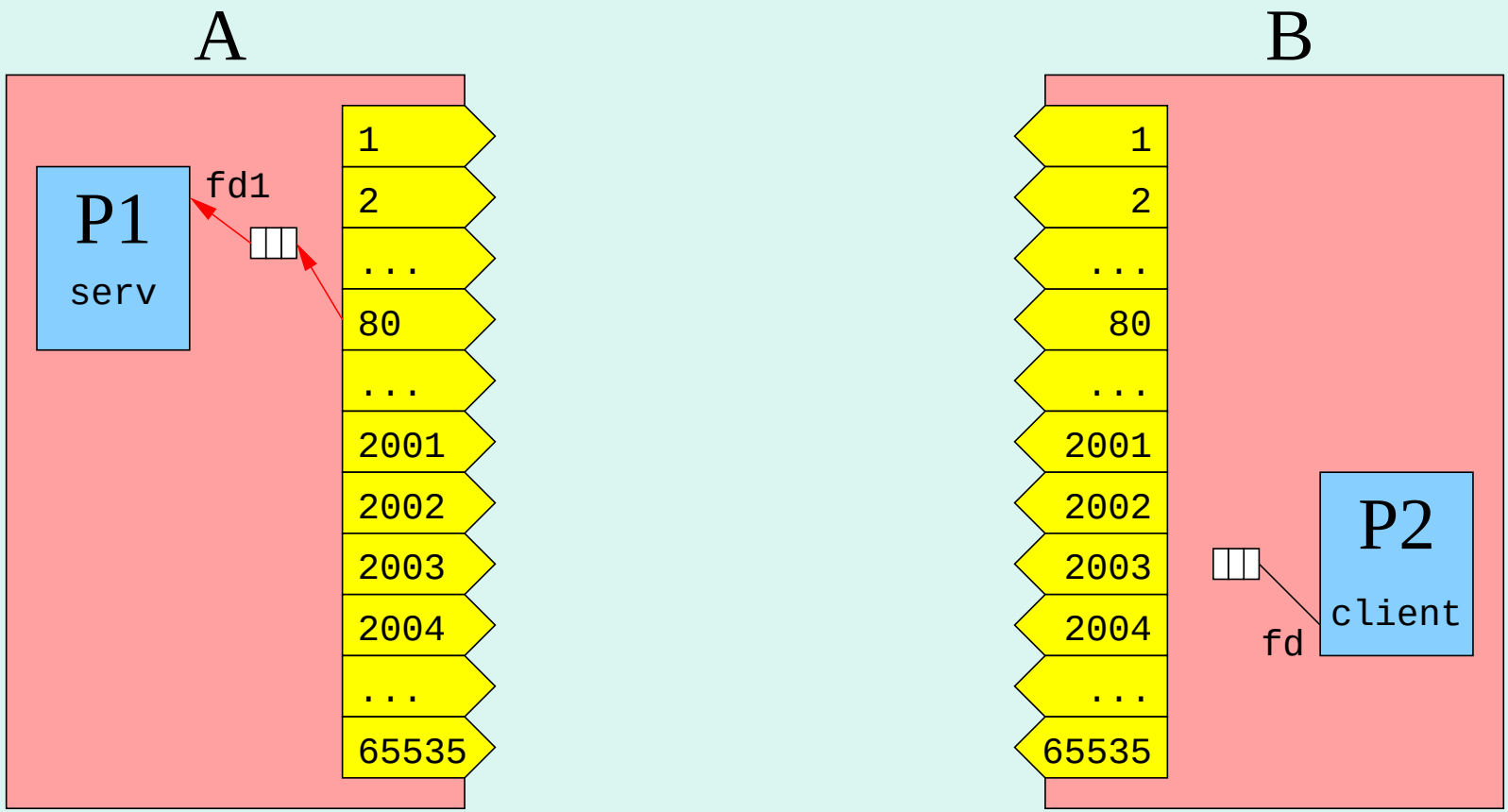
Exécution (2)

- Après `bind (fd1, ...)`



Exécution (3)

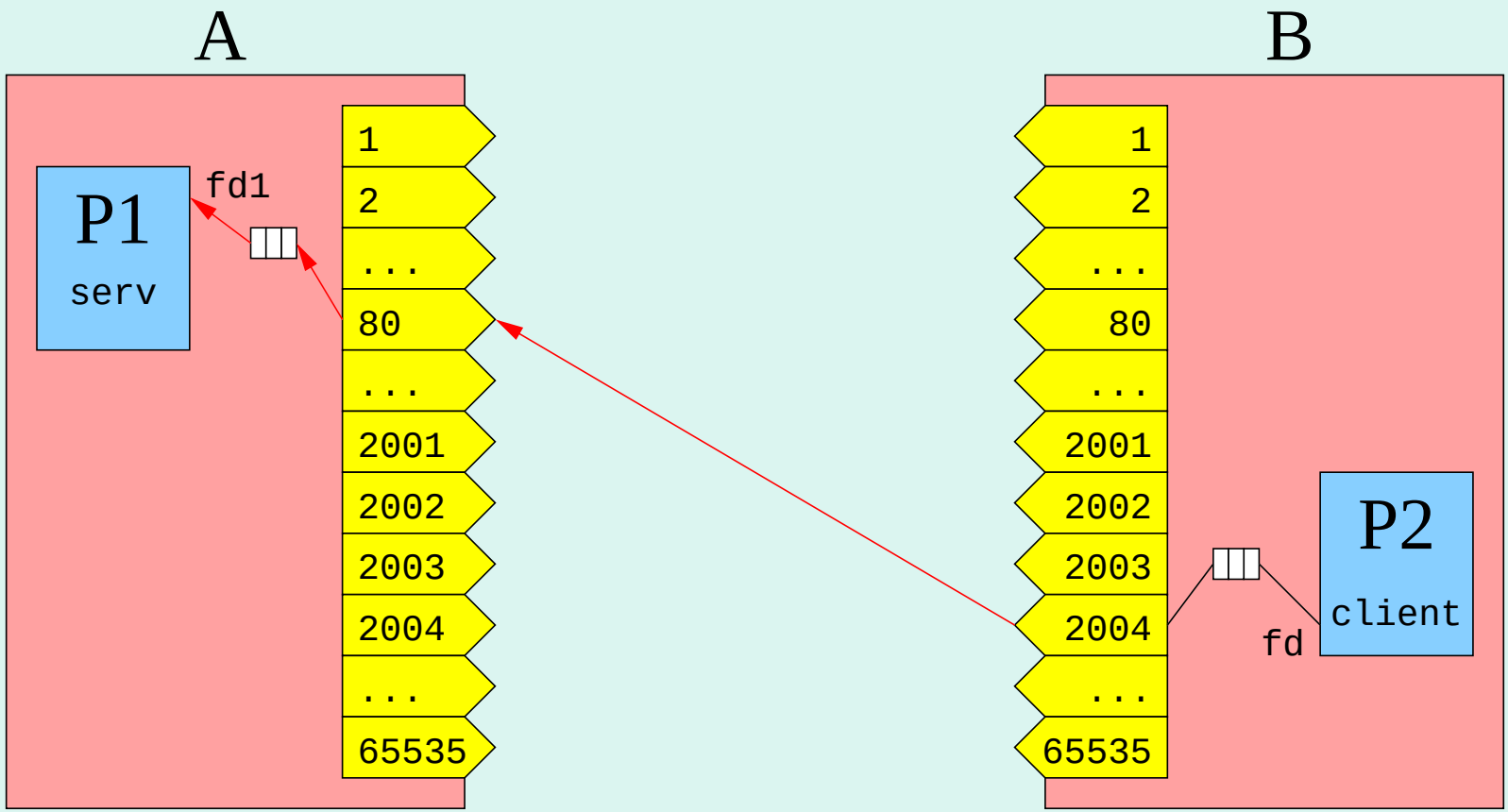
- Après `listen (fd1, 5)`



- Au plus 5 connexions établies mais pas encore `acceptées`

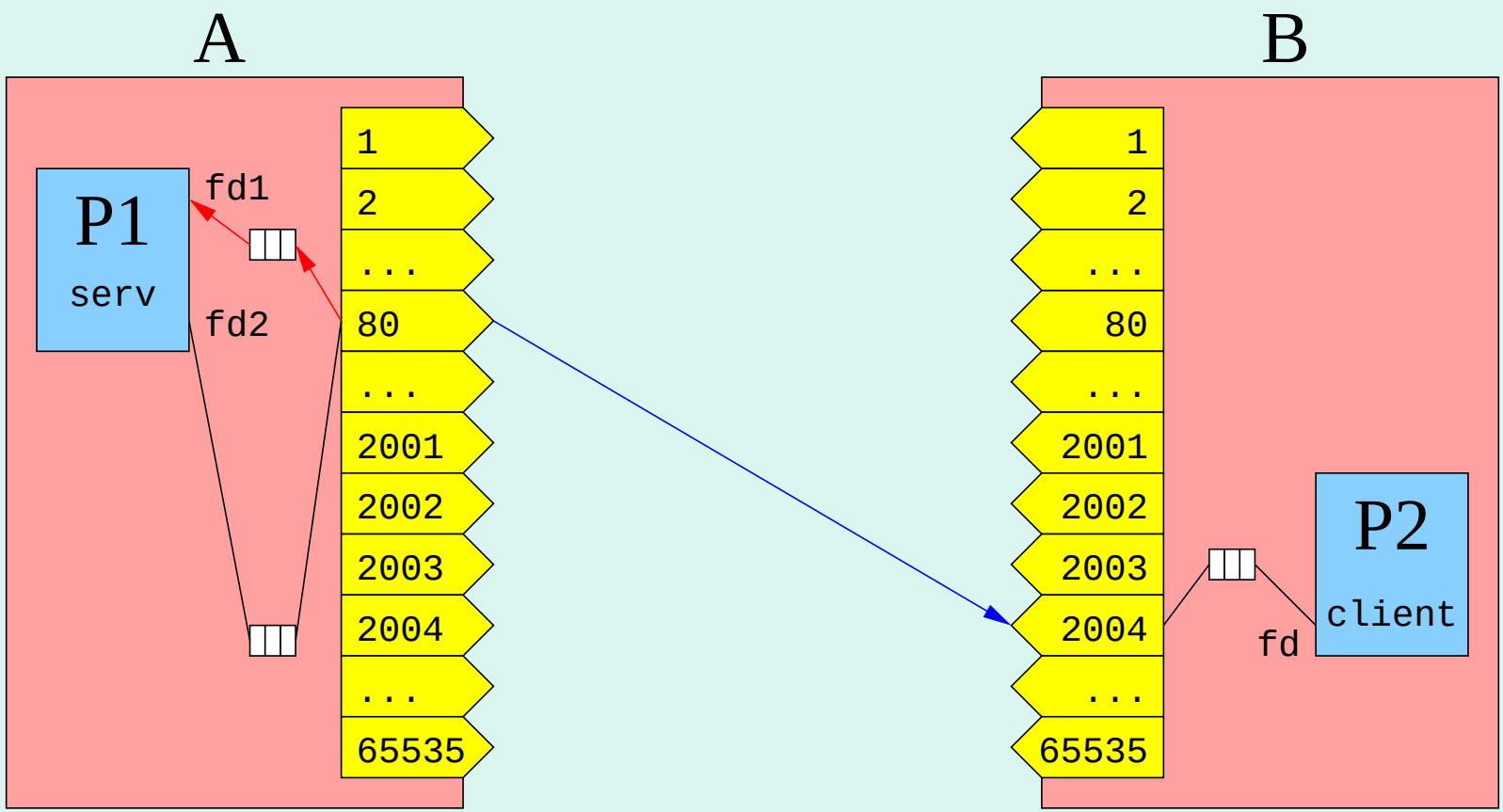
Exécution (4)

- Première phase du `connect (fd, ...)`



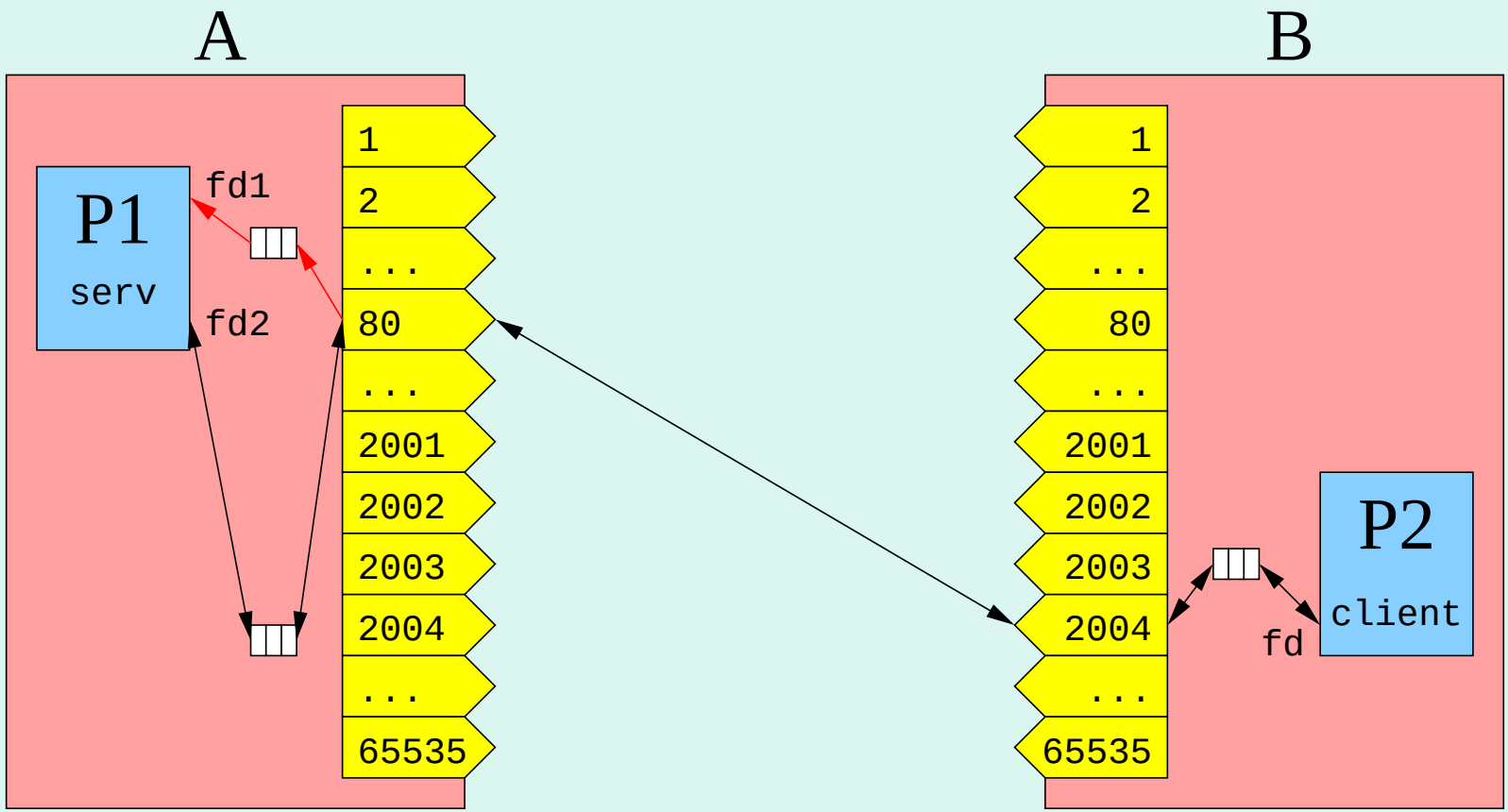
Exécution (5)

- Après `fd2 = accept (fd1, ...)` et deuxième phase du `connect (fd, ...)`



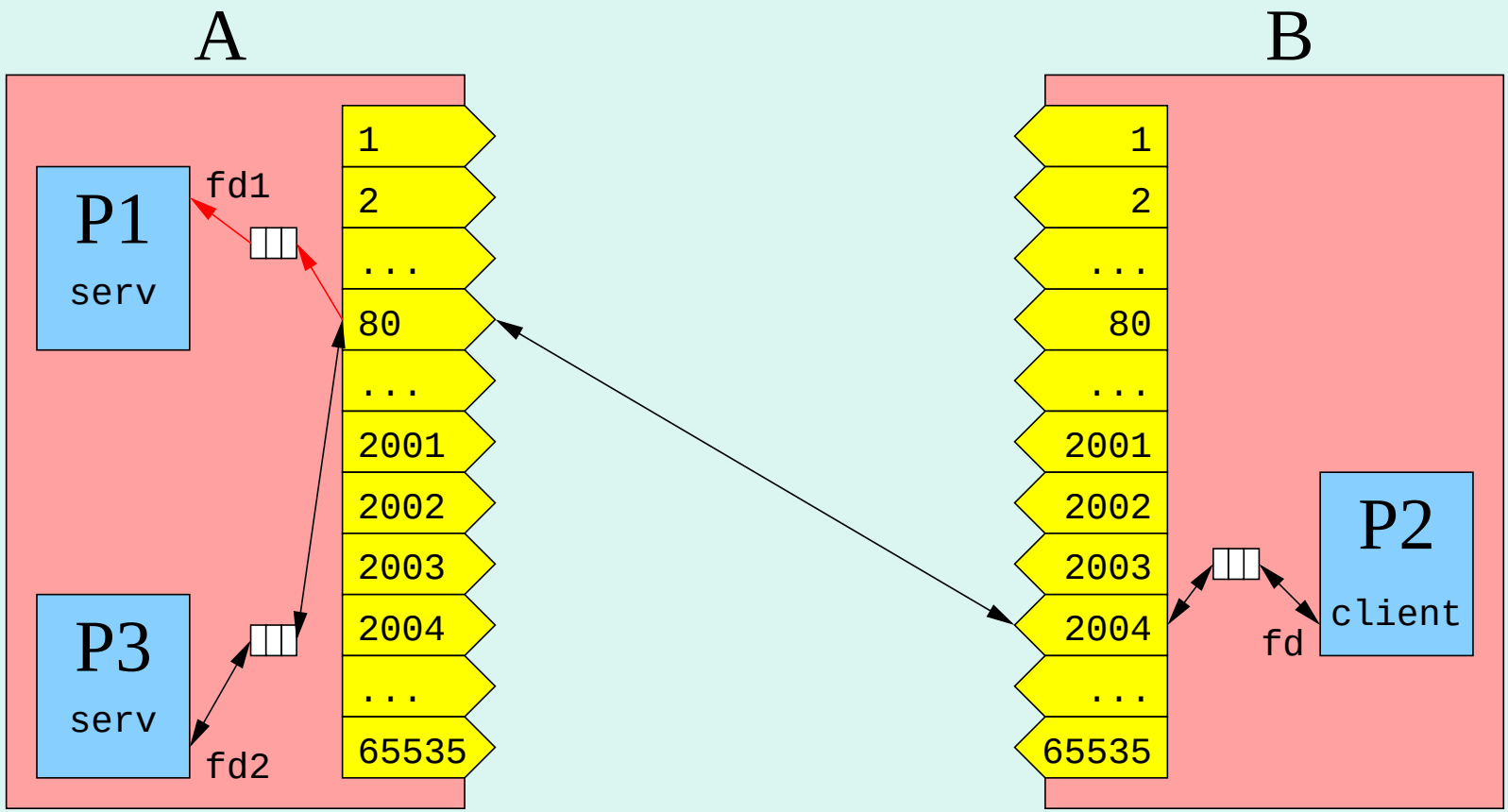
Exécution (6)

- Après `connect (fd, ...)`



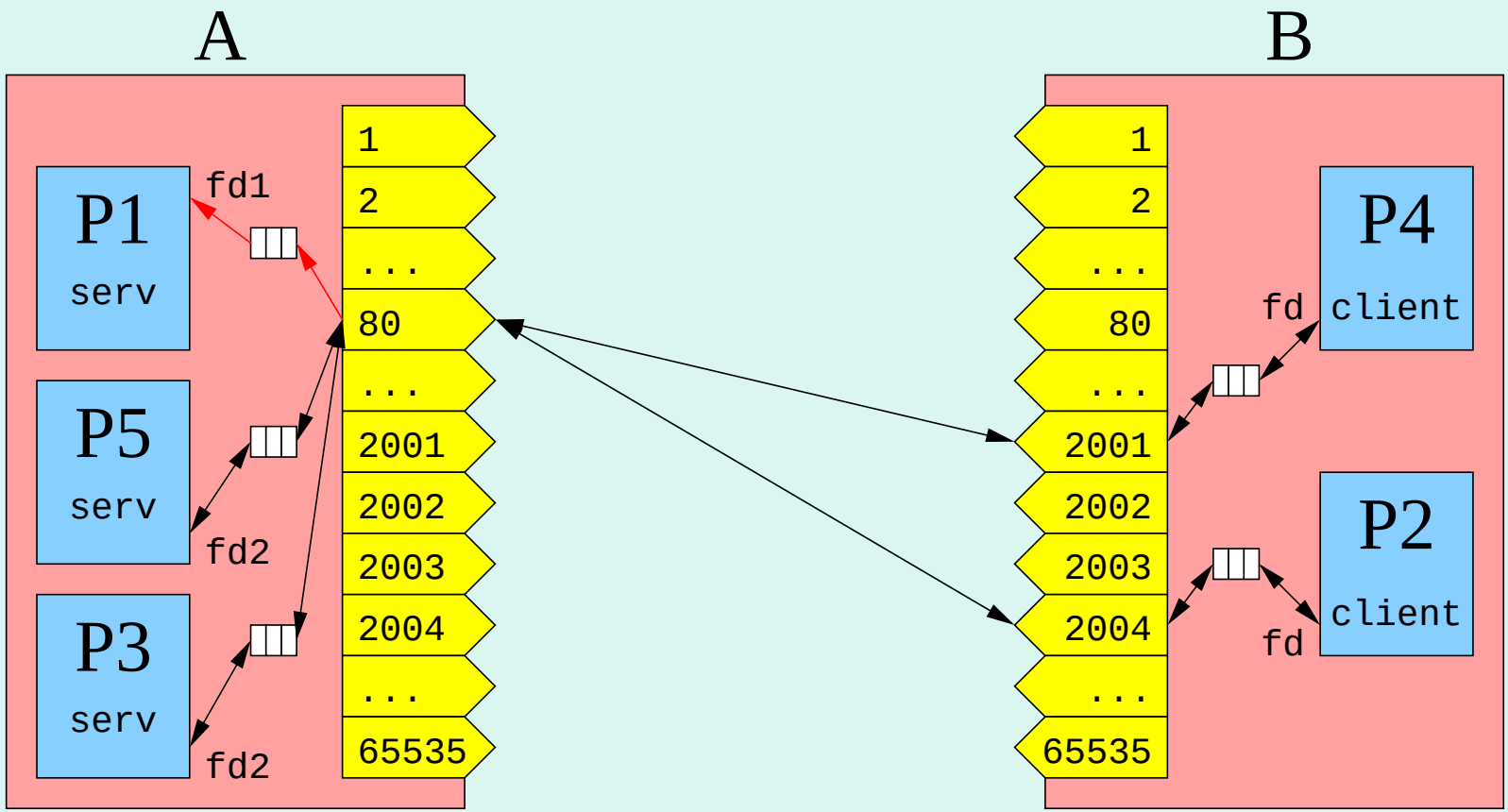
Exécution (7)

- Après `fork ()`, `close (fd1)` et `close (fd2)`



Exécution (8)

- Après connexion d'un autre client



- Le "socket pair" distingue les 2 connexions

Manipulation d'adresse IP (1)

- Routines pratiques pour manipuler des adresses IP :

```
in_addr_t inet_addr (char* addr);

struct hostent* gethostbyname(char* name);

struct hostent* gethostbyaddr(char* addr,
                              int len,
                              int type);
```

- Champs de `struct hostent`:

```
1. struct hostent
2. {
3.   char* h_name;        // Official name of host
4.   char** h_aliases;    // Alias list
5.   int h_addrtype;      // Host address type
6.   int h_length;        // Length of address
7.   char** h_addr_list; // List of addresses from name s
8. };
```

Manipulation d’adresse IP (2)

- Exemple :

```
in_addr_t adr;

adr = (132<<24)+(204<<16)+(24<<8)+179;

adr = inet_addr ("132.204.24.179");

server.sin_addr.s_addr = htonl (adr);

struct hostent *he;

he = gethostbyaddr ((char*)&adr, sizeof (adr), AF_INET
printf ("%s\n", he->h_name); // himalia.iro.umontreal.

he = gethostbyname ("www.iro.umontreal.ca");
server.sin_addr = *(struct in_addr*)he->h_addr_list[0]
```

`getsockname` et `getpeername` (1)

- Grace à `getsockname` et `getpeername`, un socket peut être interrogé pour connaître l'adresse IP et le port à chaque bout d'une connexion

```
int getsockname (int fd,
                 struct sockaddr* name,
                 int namelen);

int getpeername (int fd,
                 struct sockaddr* name,
                 int namelen);
```

getsockname et getpeername (2)


```
1.  struct sockaddr a;
2.  int len;
3.  getsockname(fd, &a, &len); print_addr(a);// mon bout
4.  getpeername(fd, &a, &len); print_addr(a);// l'autre bout
5.  ...
6.
7.  void print_addr (struct sockaddr addr)
8.  { struct hostent *h;
9.    struct sockaddr_in* p = (struct sockaddr_in*)&addr;
10.
11.   h = gethostbyaddr ((char*)&p->sin_addr.s_addr,
12.                      4,
13.                      AF_INET);
14.
15.   printf ("%s (%d.%d.%d.%d) port=%d\n",
16.           h->h_name,
17.           (ntohl (p->sin_addr.s_addr) >> 24) & 0xff,
18.           (ntohl (p->sin_addr.s_addr) >> 16) & 0xff,
19.           (ntohl (p->sin_addr.s_addr) >>  8) & 0xff,
20.           (ntohl (p->sin_addr.s_addr) >>  0) & 0xff,
21.           (ntohs (p->sin_port)));
22. }
23.
24. // sortie:
25. //   www.iro.umontreal.ca (132.204.24.179) port=80
26. //   mega.iro.umontreal.ca (132.204.26.92) port=1192
```

Caractéristiques de TCP (1)

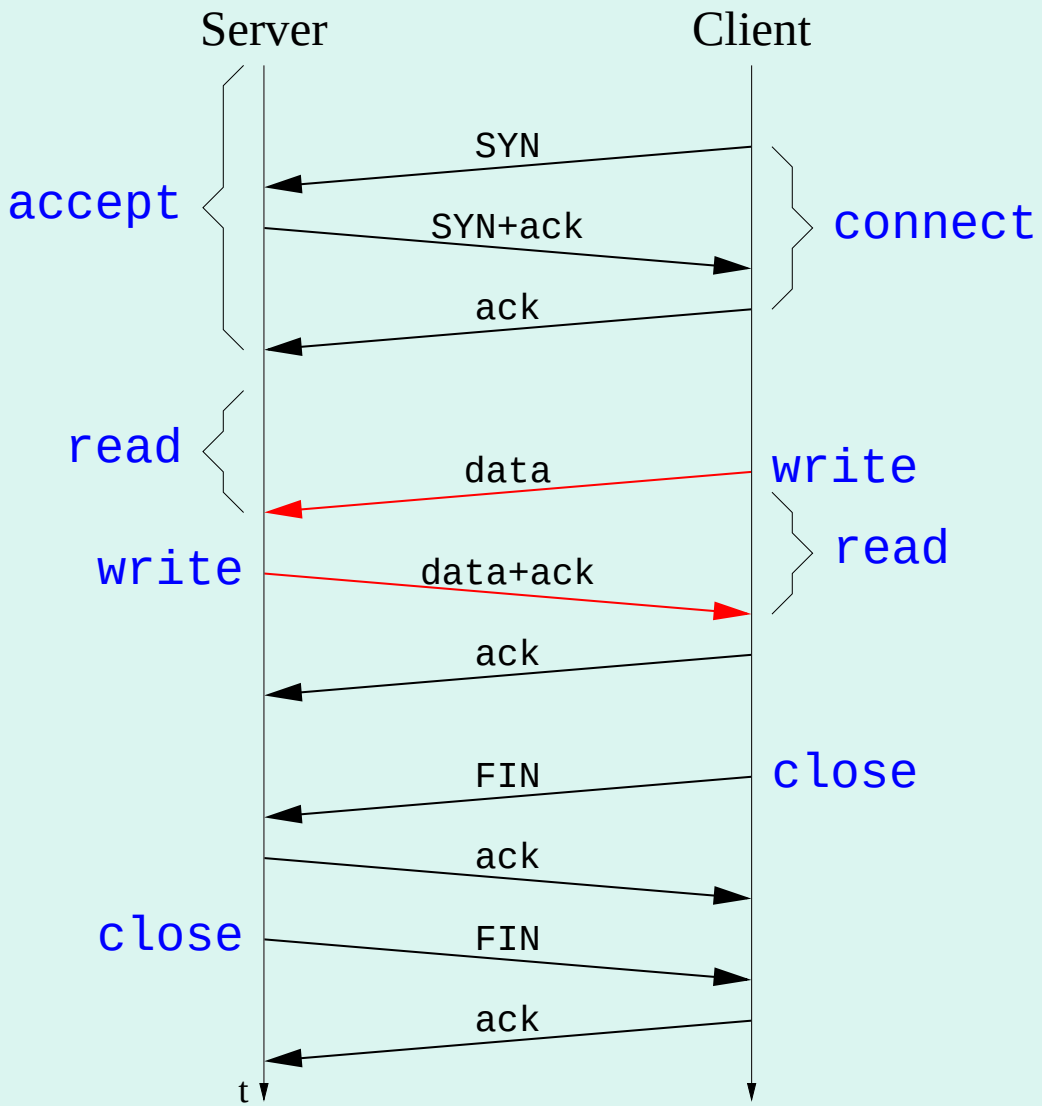
- Les données échangées sont divisées en **segments**
- Le receveur doit renvoyer un **accusé de réception** à l'envoyeur; si ce dernier ne le reçoit pas dans un certain laps de temps le segment est envoyé à nouveau (**retransmission**)
- L'accusé de réception est retardé d'une fraction de seconde pour qu'il puisse être inclu avec des données utiles dans le prochain segment venant du receveur

Caractéristiques de TCP (2)

- Un **“checksum”** permet de détecter les erreurs de communication; aucun accusé de reception (même négatif) n’est envoyé pour les segments corrompus
- Les segments sont numérotés (**numéro de séquence**) pour que le receveur puisse les assembler dans le bon ordre
- Les segments dupliqués sont ignorés
- Le flux des données est controlé par une **“fenêtre glissante”** qui indique à l’envoyeur combien d’espace libre il reste dans le FIFO du receveur

Vie d'une Connexion

- La vie d'une connexion passe par trois phases: **établissement**, **transfert de donnée**, et **fermeture**



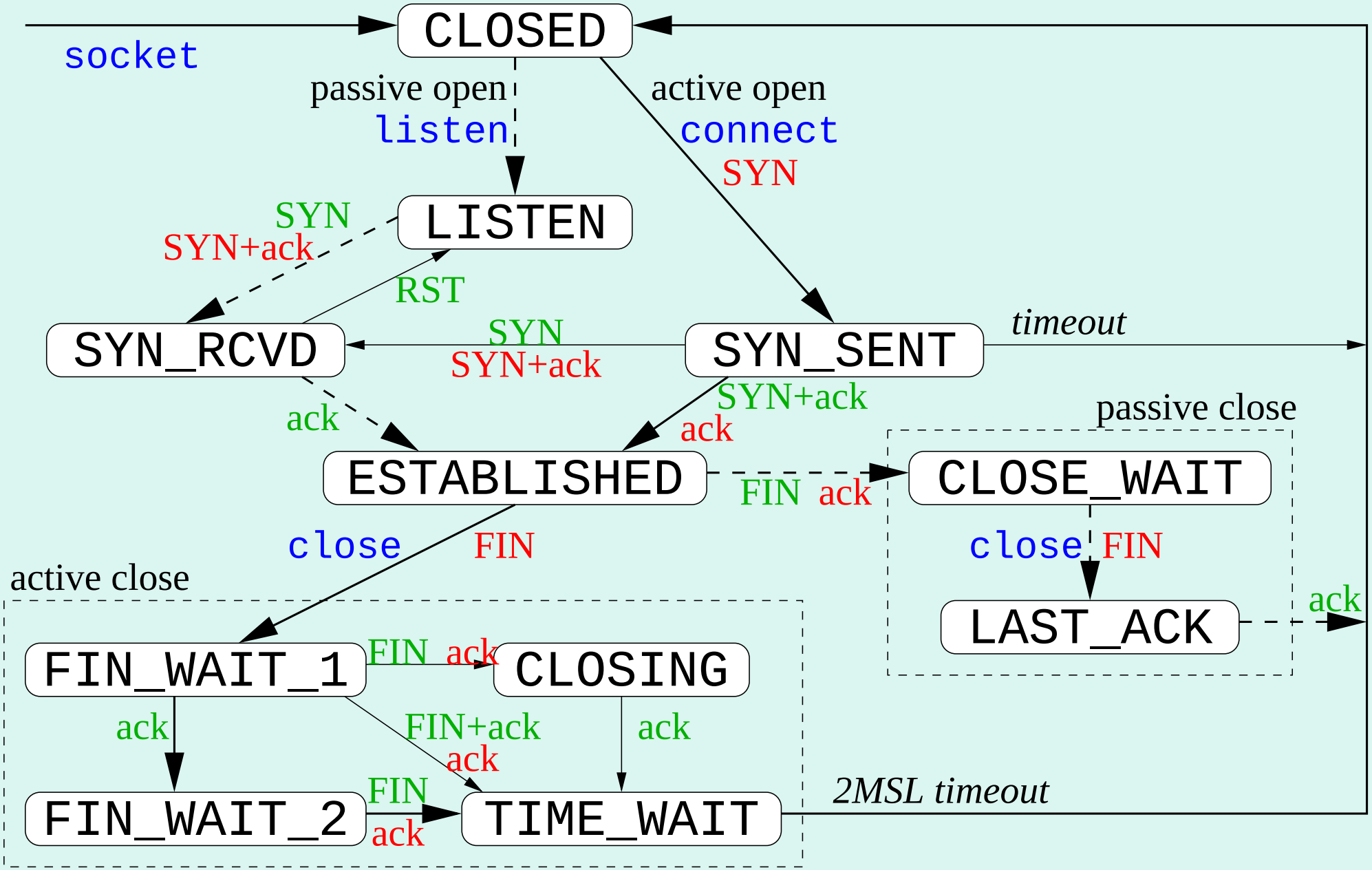
Flags d’un Segment TCP

- Chaque segment TCP peut contenir un des flags suivants:
  - **SYN**: synchroniser les accusés de reception
  - **FIN**: signaler la fin d’une connexion
  - **RST**: mise-à-zéro d’une connexion

- Chaque segment TCP contient aussi un accusé de reception qui indique combien de données ont été reçues correctement

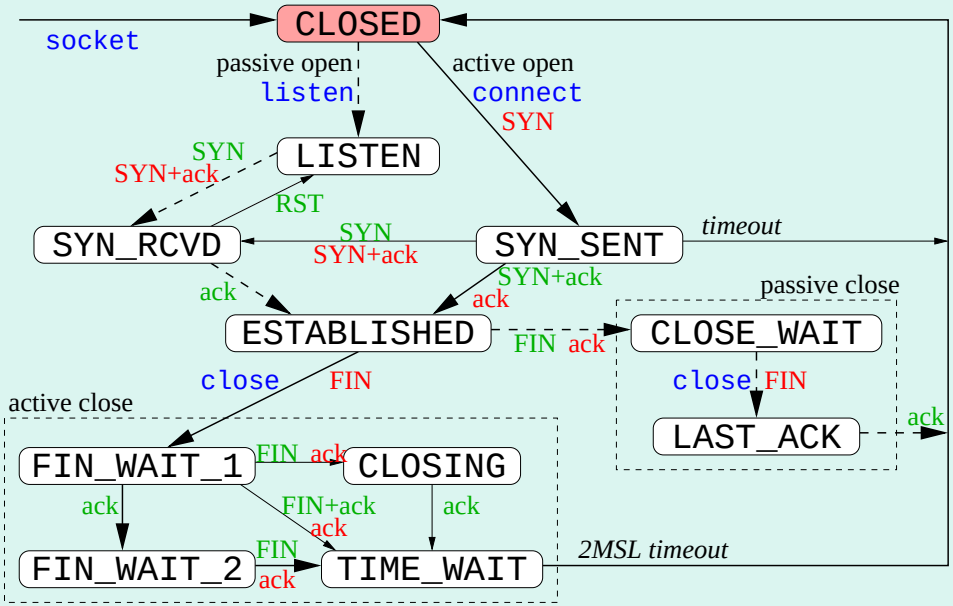
- Les données, flags et accusés de reception peuvent être combinées dans un même segment pour minimiser le nombre de paquets envoyés

États d'un Socket

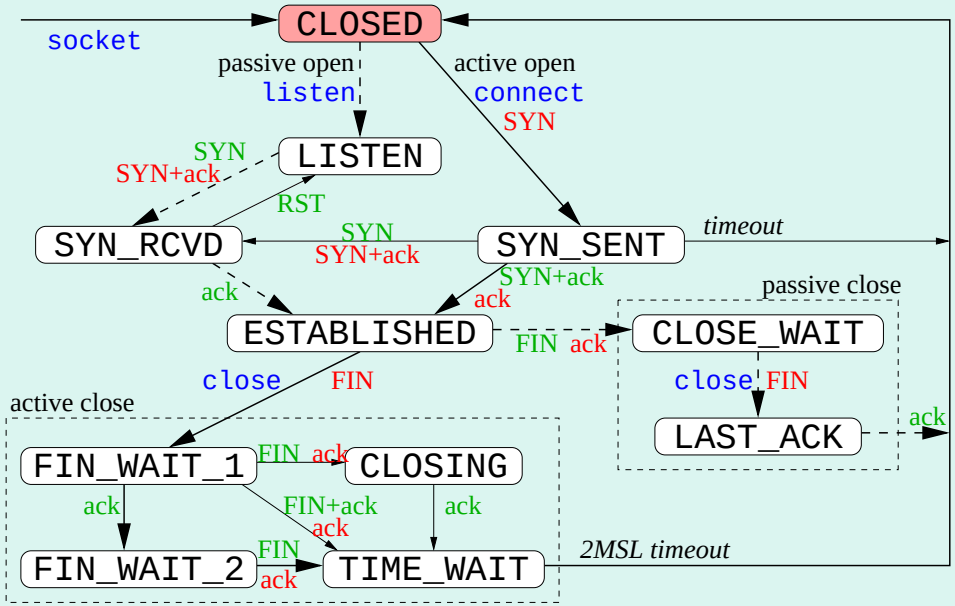


Établissement d'une Connexion (1)

1) Serveur

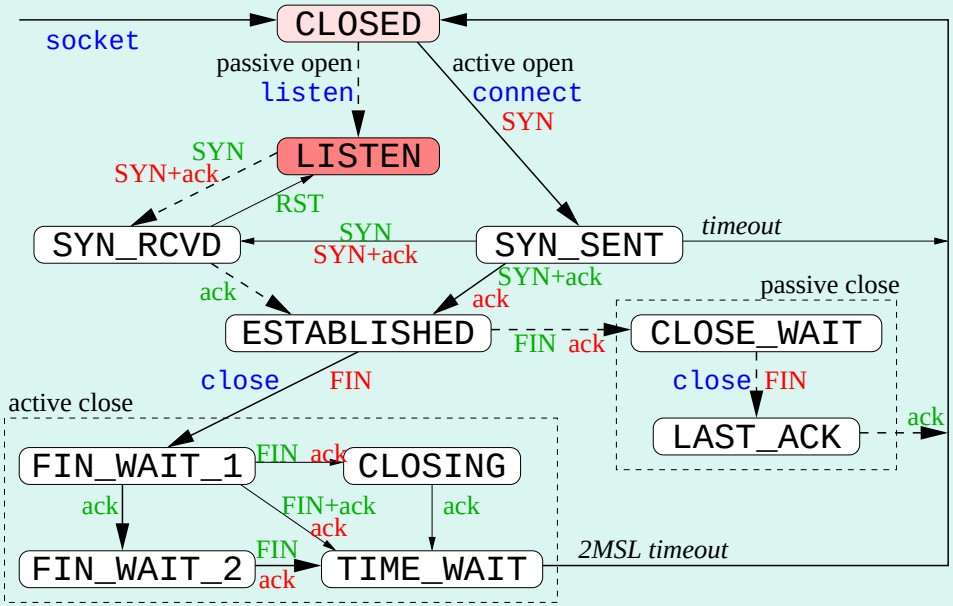


Client

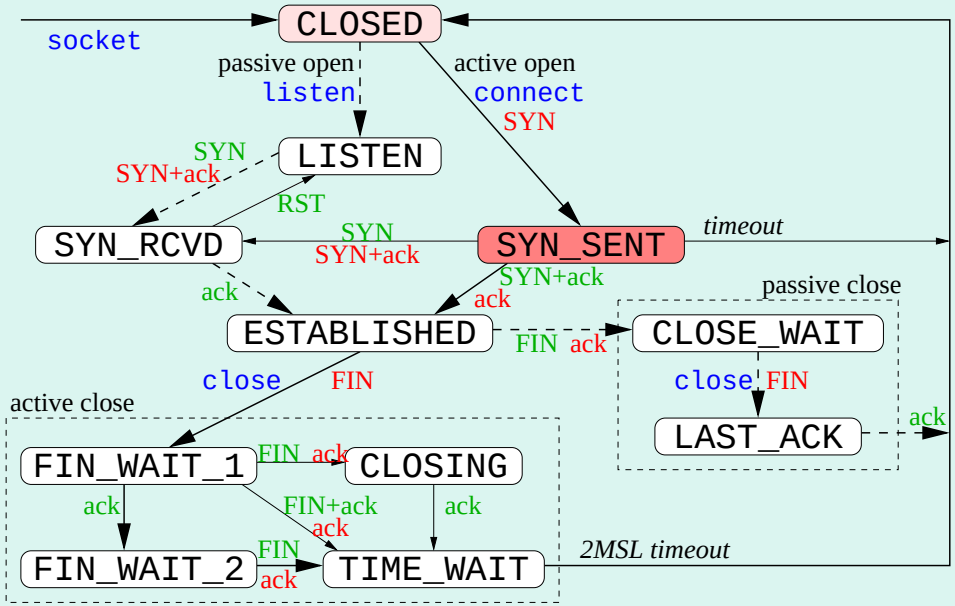


Établissement d'une Connexion (2)

2) Serveur

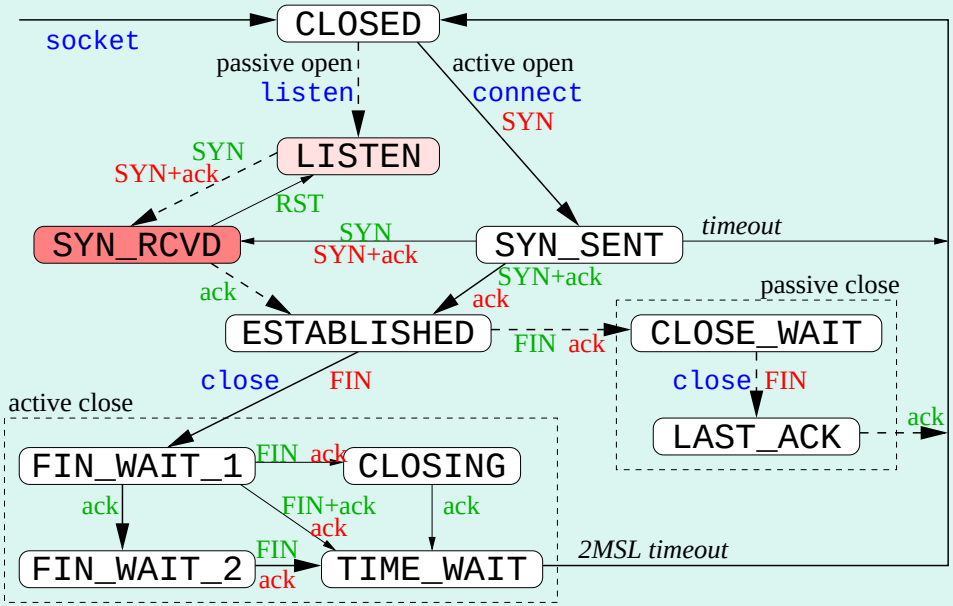


Client

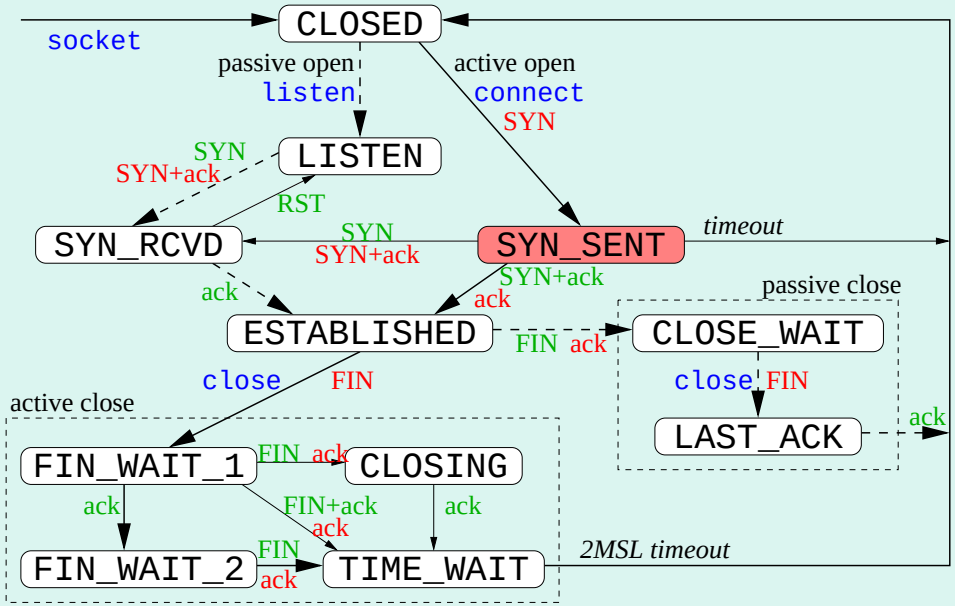


Établissement d'une Connexion (3)

3) Serveur

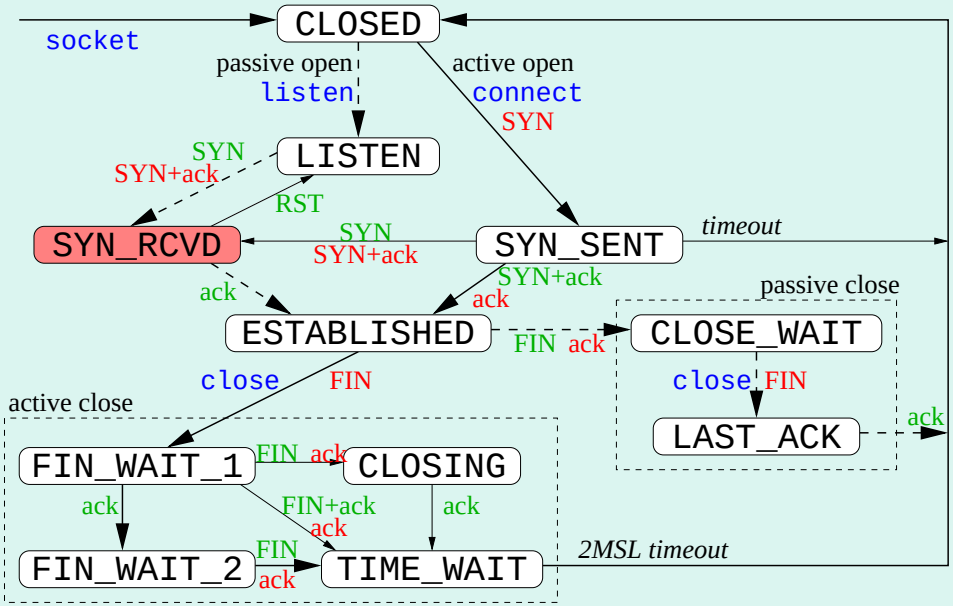


Client

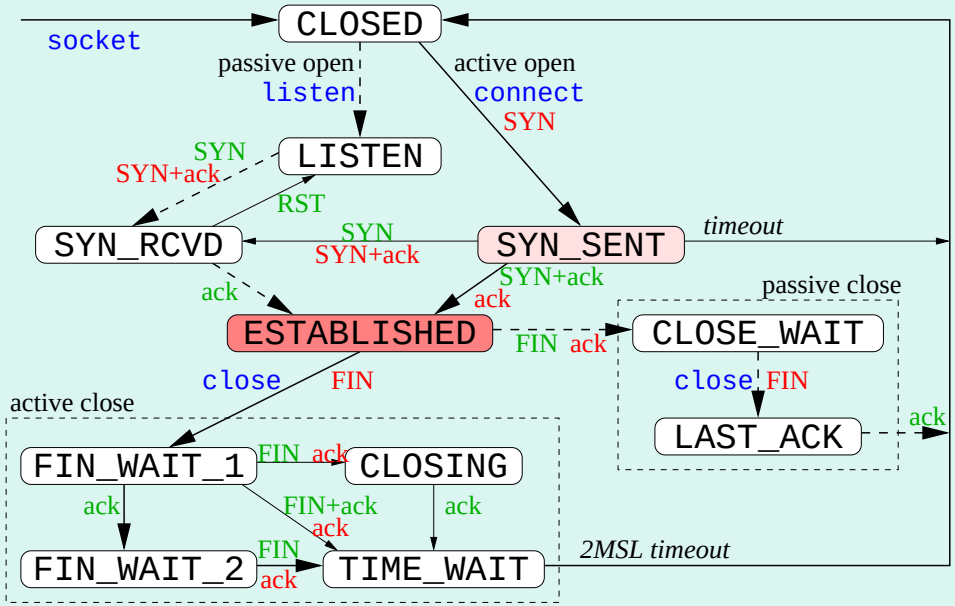


Établissement d'une Connexion (4)

4) Serveur

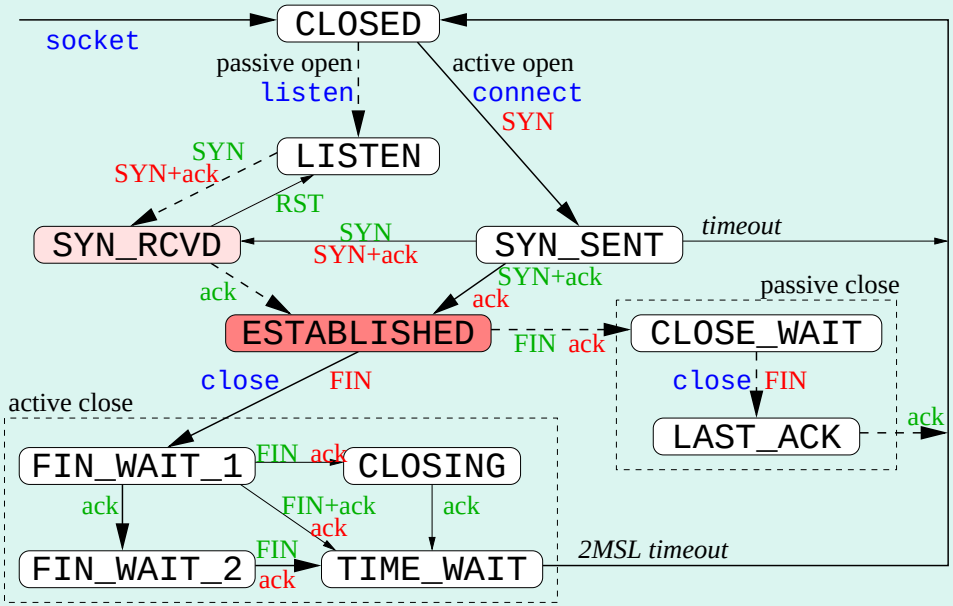


Client

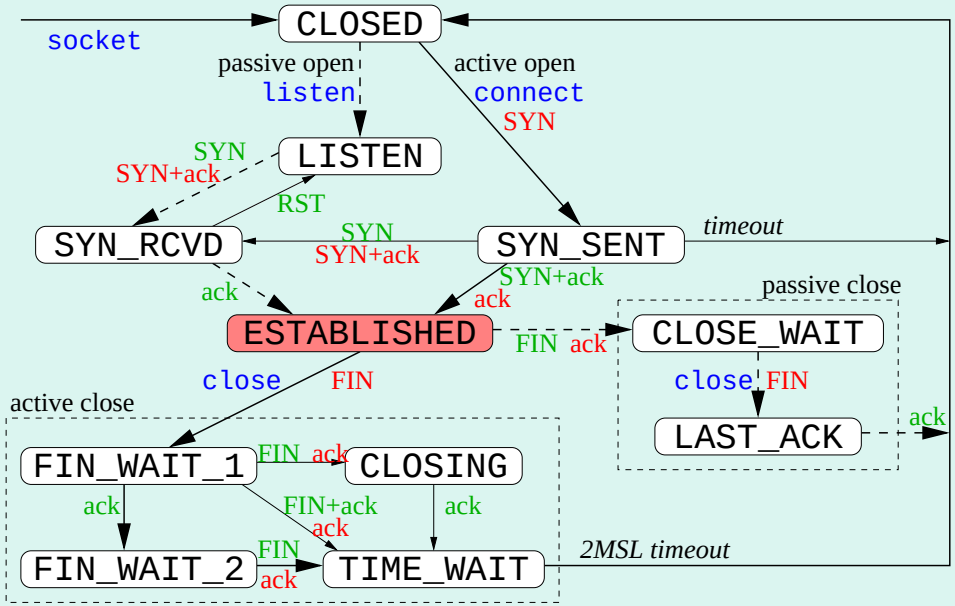


Établissement d'une Connexion (5)

5) Serveur

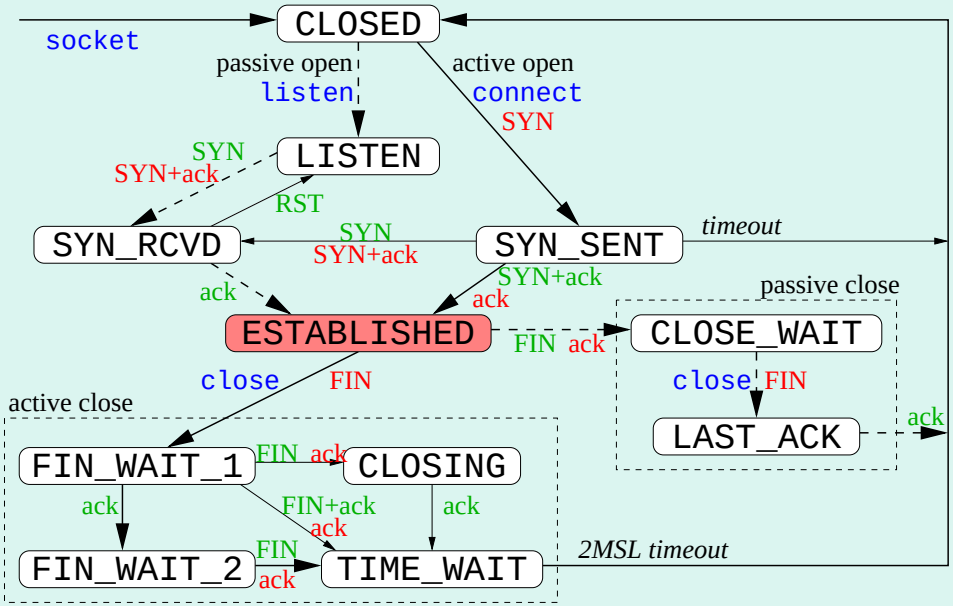


Client

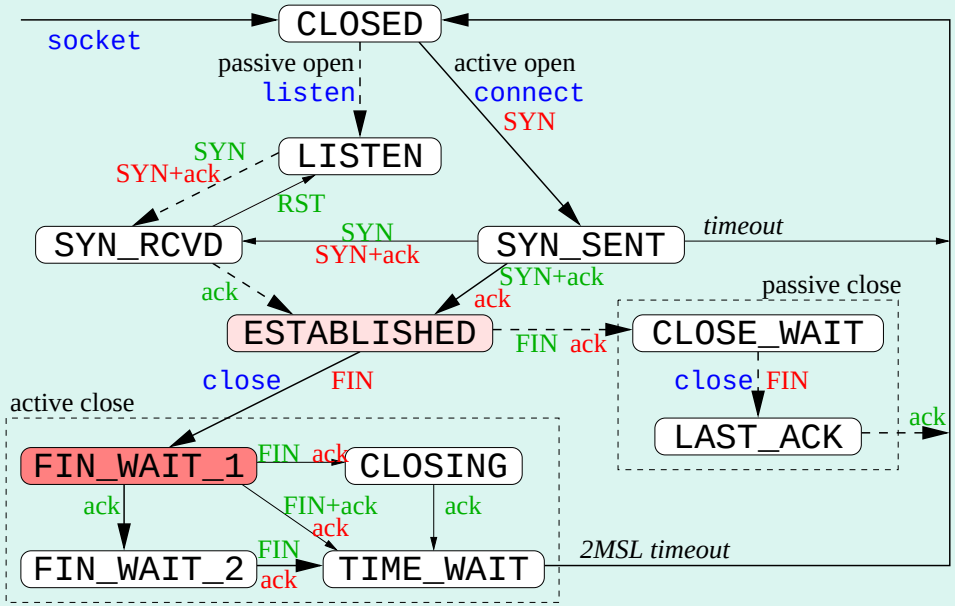


Fermeture d'une Connexion (1)

1) Serveur

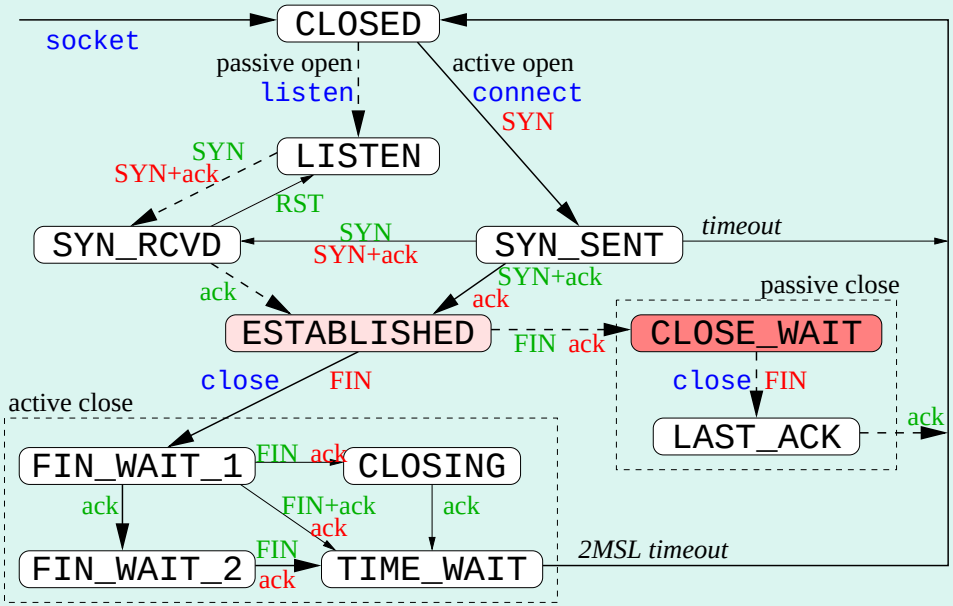


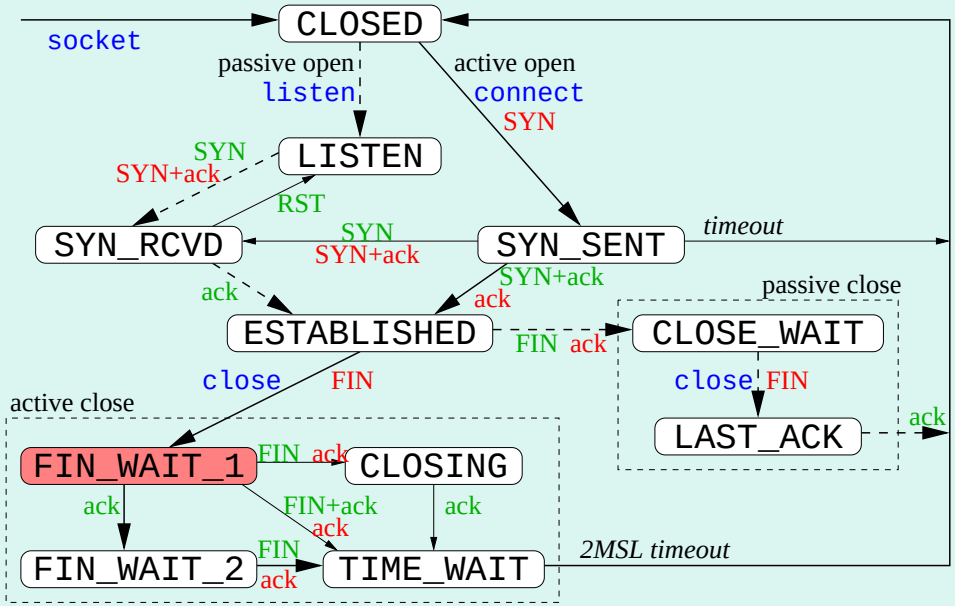
Client



Fermeture d'une Connexion (2)

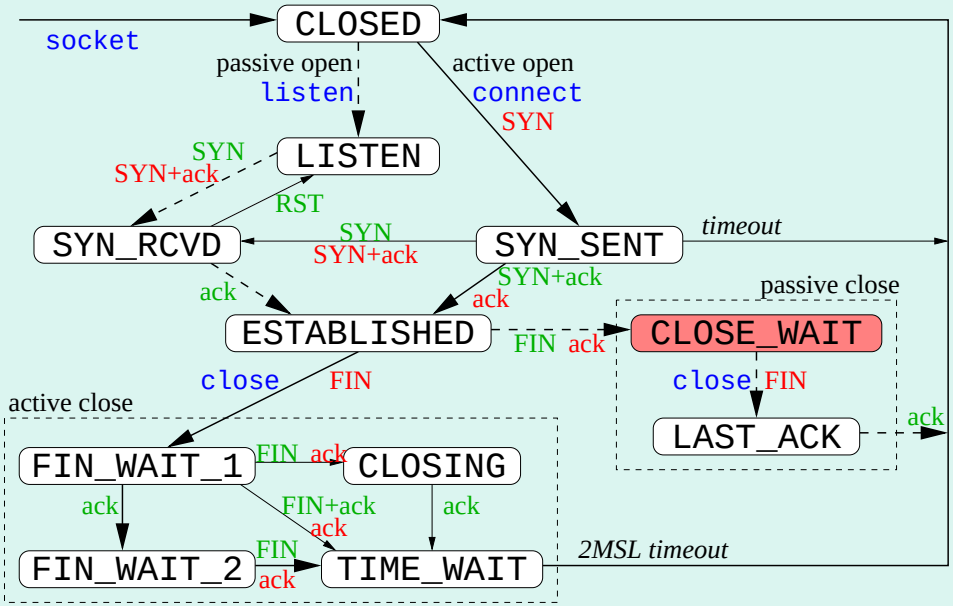
2)



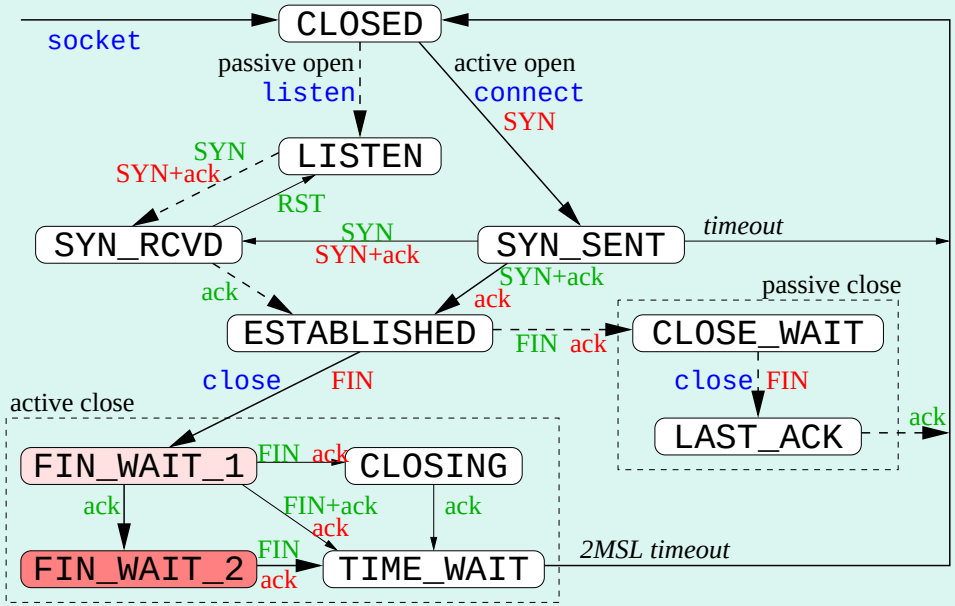


Fermeture d'une Connexion (3)

3) Serveur



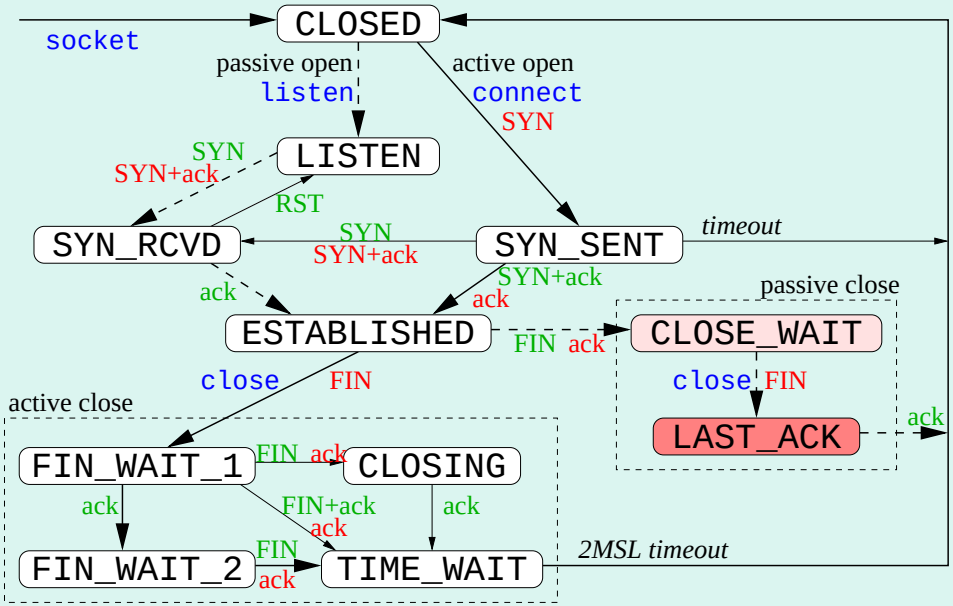
Client



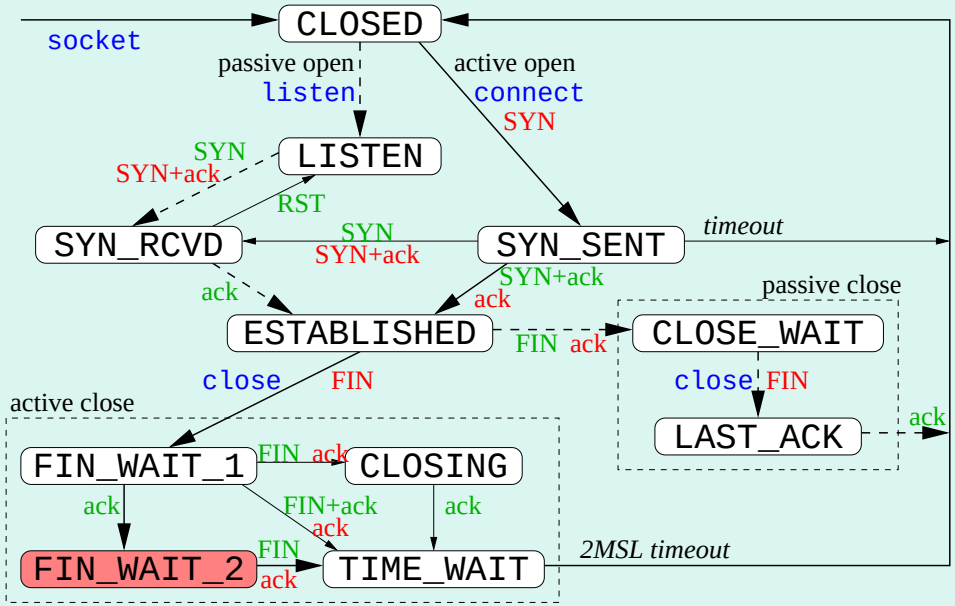
Fermeture d'une Connexion (4)

4)

Serveur

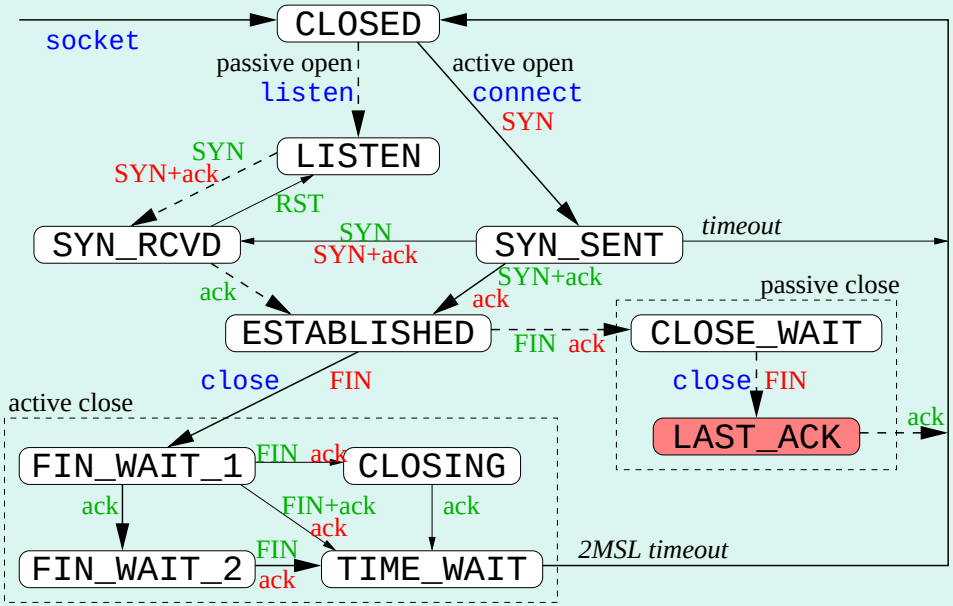


Client

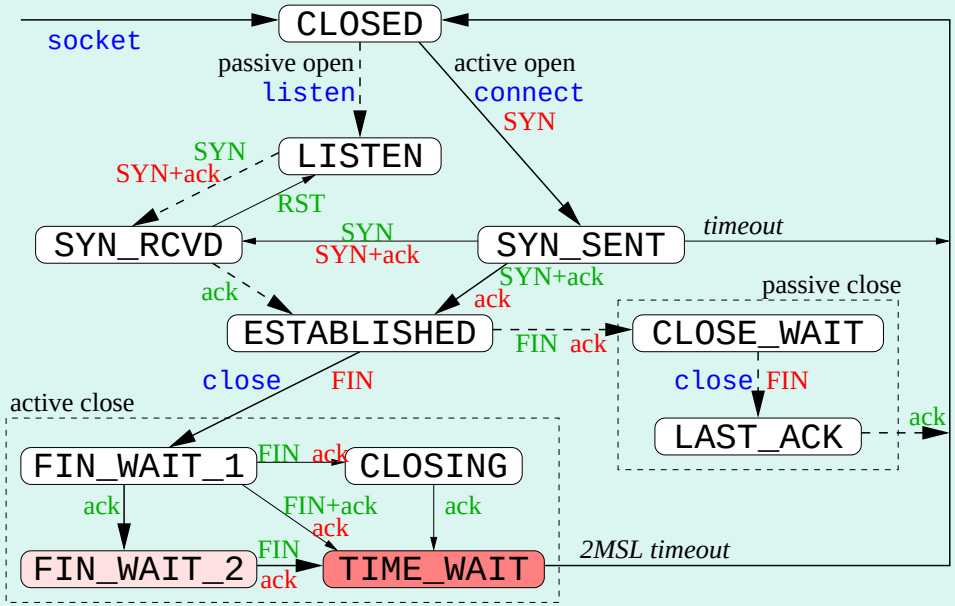


Fermeture d'une Connexion (5)

5) Serveur

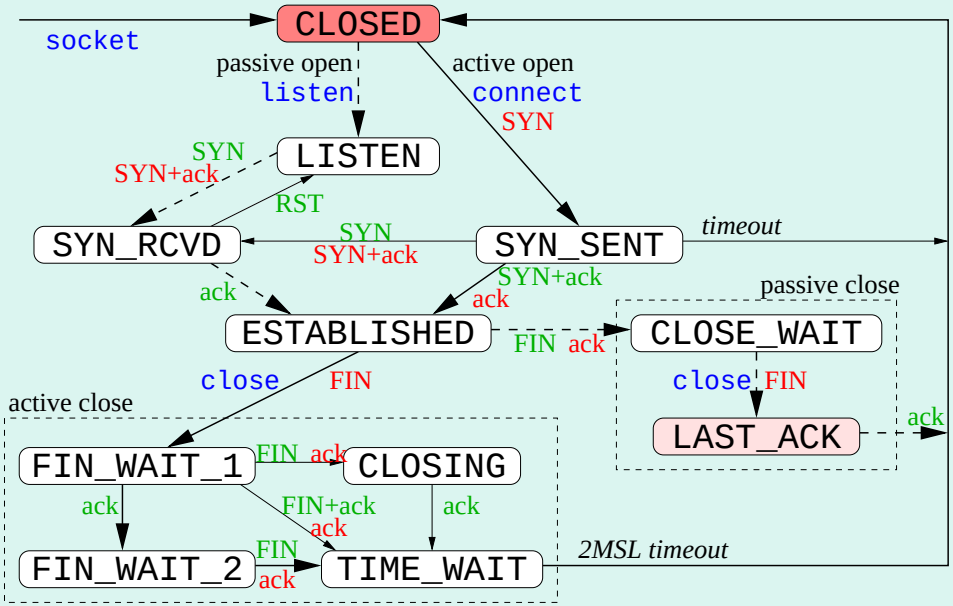


Client

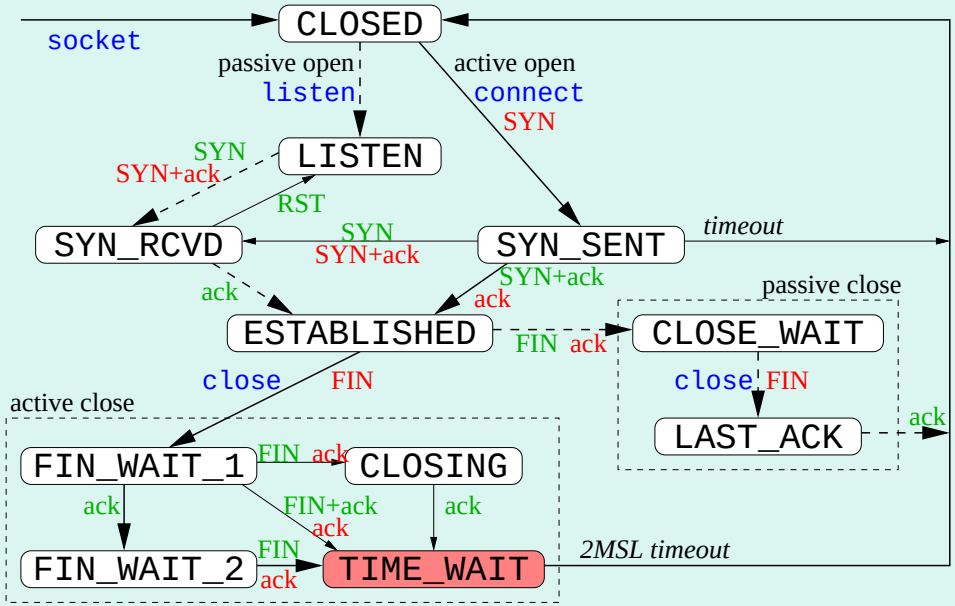


Fermeture d'une Connexion (6)

6) Serveur



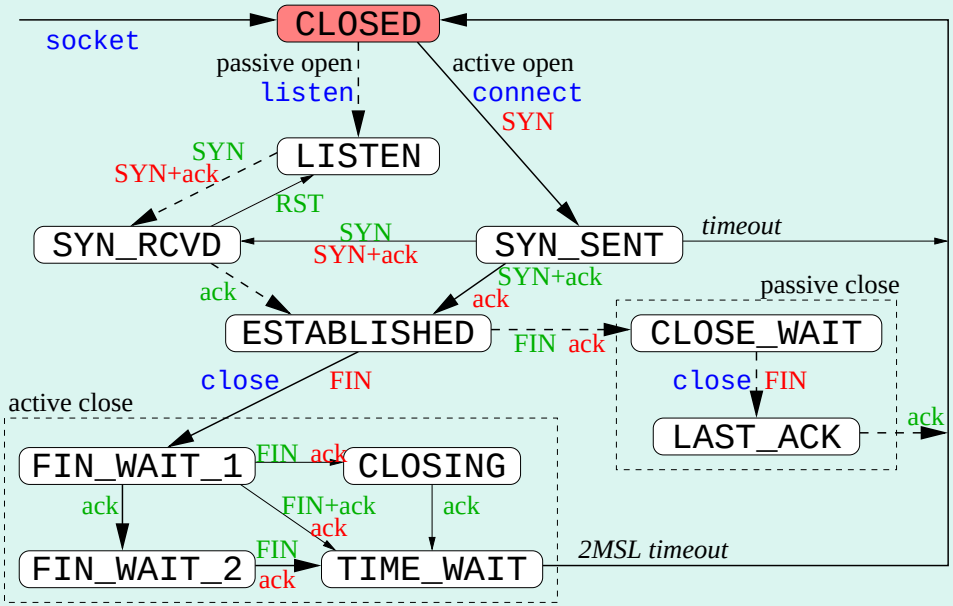
Client



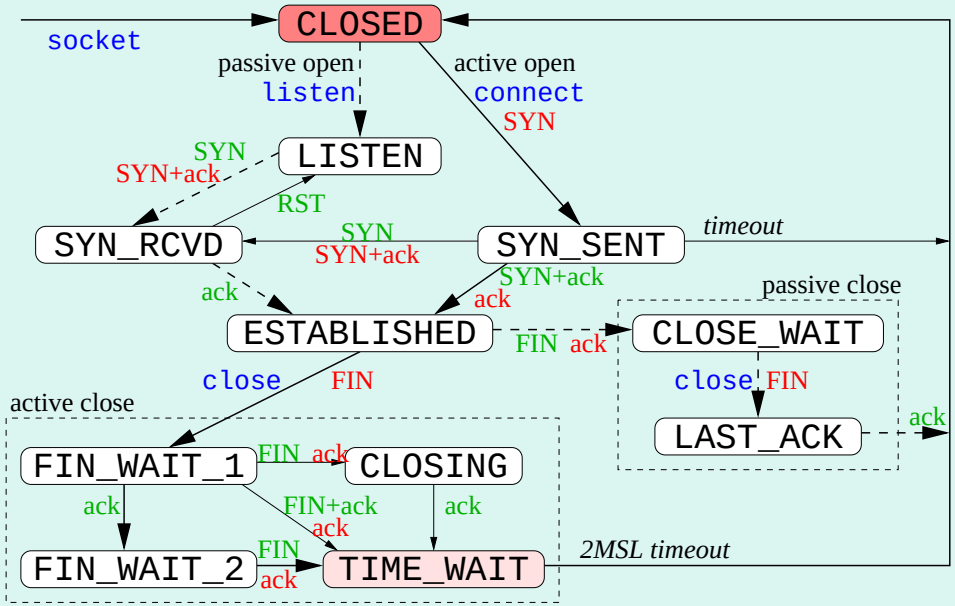
Fermeture d'une Connexion (7)

7)

Serveur



Client



Fermeture Partielle (1)

- Une lecture sur un socket bloque s'il n'y a **pas de données reçues** et la **connexion n'est pas fermée**
- Une lecture sur un socket retourne "fin de fichier" (`read` retourne 0) si la **connexion est fermée**
- Parfois un des processus (serveur ou client) désire indiquer à l'autre qu'il a **fini d'envoyer des données** sur la connexion mais qu'il peut encore en recevoir
- Par exemple, envois d'une requête de **longueur indéterminée** (une "fin de fichier" représente la fin de la requête pour le serveur)
- Si le client utilise `close` à la fin de la requête, le serveur va lire une fin de fichier **mais le client ne pourra pas lire la réponse du serveur**

Fermeture Partielle (2)

- L'appels système `shutdown` permet de fermer sélectivement chacun des sens d'une connexion (écriture ou lecture)

  `int shutdown (int` fd`, int` how`);`

  how doit être `SHUT_RD`, `SHUT_WR` ou `SHUT_RDWR`

- Exemple :

```
1.  int fd = socket (PF_INET, SOCK_STREAM, 0);
2.
3.  connect (fd, ...);
4.
5.  write (fd, ...);
6.
7.  shutdown (fd, SHUT_WR);
8.
9.  read (fd, ...);
10.
11. close (fd);
```

Mémoire Secondaire

- La mémoire principale (RAM) est rapide, mais **chère** et **volatile**
- La **mémoire secondaire** permet de stocker une **grande quantité** d'information de façon **permanente** et à **faible prix**
- Le temps d'accès est **beaucoup plus grand** que pour la RAM (dans le meilleur cas, quelques millisecondes par accès)
- Exemples typiques:
  - Disque dur (capacité élevée)
  - Floppy et ZIP (faible capacité)
  - CD-RW, CD-ROM, CD-R et WORM (capacité moy.)
  - Ruban magnétique (capacité très élevée)

Systèmes de Fichier (1)

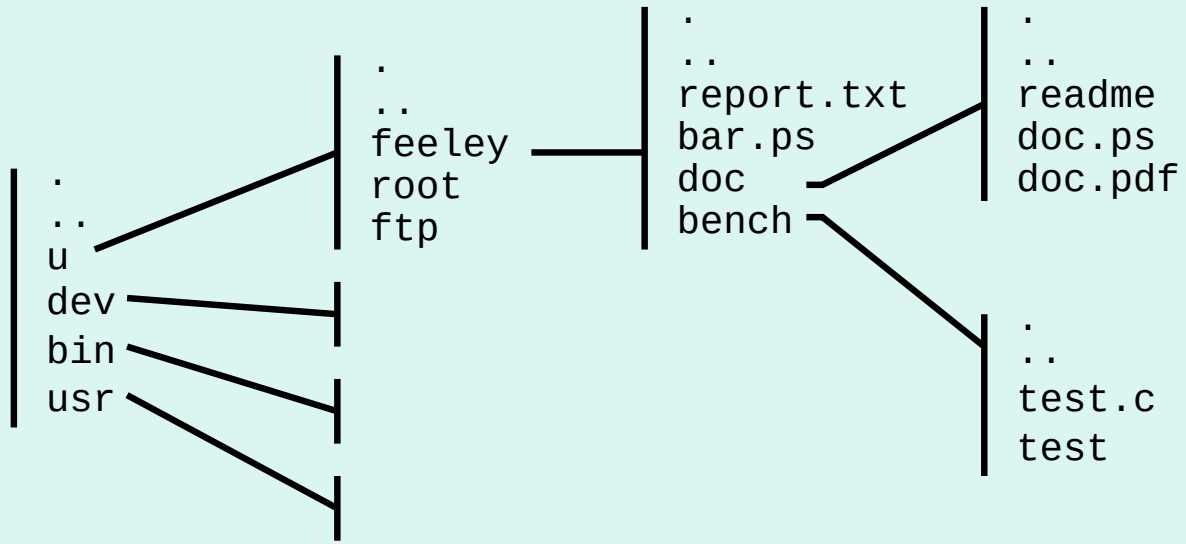
- Un système de fichier est une **organisation** des données permanentes dans la mémoire secondaire et des **méthodes pour gérer** ces données
- Facteurs à considérer:
  - **Identification**: syntaxe des noms de fichier
  - **Types permis**: réguliers (texte ou binaire), sous-répertoires, périphériques (“devices”), etc
  - **Attributs**: date de création/accès? permissions d’accès?
  - **Performance**: latence/débit d’accès aux fichiers séquentiel/aléatoire, utilisation économe de l’espace
  - **Fiabilité**: peut-il y avoir perte d’information? outils pour réparer les inconsistances? “backups”?

Systèmes de Fichier (2)

- Un fichier normal est une **séquence d'octets** d'une certaine longueur
- Opérations sur les fichiers (et fonction UNIX)
  - **Création**: allouer une entrée de répertoire (`creat`)
  - **Ouverture/Fermeture**: début/fin des accès au contenu du fichier, le descripteur obtenu à l'ouverture contient un **index** de lecture/écriture (`open`/`close`)
  - **Écriture**: écrit des données à l'index (ce qui demande possiblement d'allouer de l'espace pour étendre le fichier) et l'avance (`write`)
  - **Lecture**: lit des données à l'index et l'avance (`read`)
  - **Repositionnement**: change l'index (`lseek`)
  - **Élimination**: élimine le fichier et libère l'espace associé (`unlink`)

Identification de Fichiers (1)

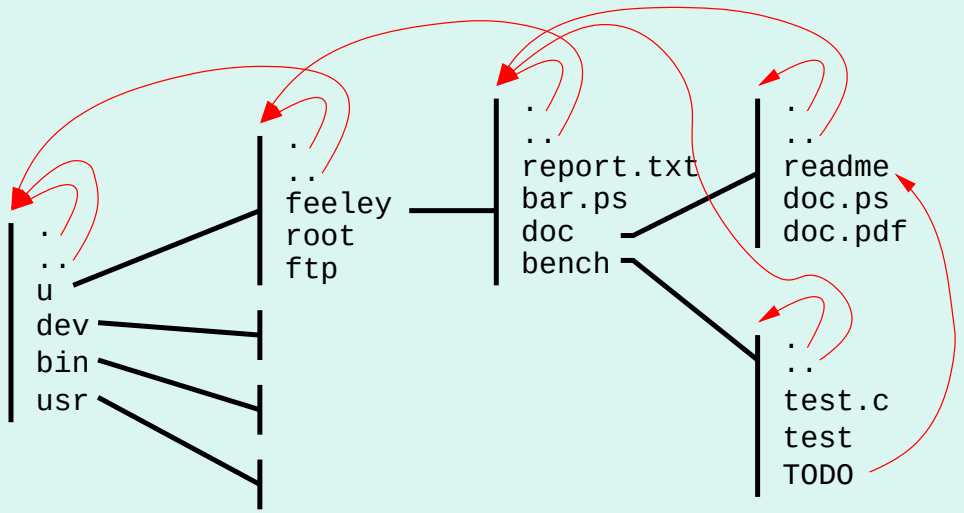
- La majorité des systèmes de fichier utilisent une **organisation hiérarchique** sous forme d'arbre
- Le **répertoire racine** ("root") est à la racine de l'arbre; les répertoires peuvent contenir des **sous-répertoires**



- Une position particulière dans l'arbre peut-être identifiée de façon unique par le **chemin** qui mène à cette position depuis la racine

Identification de Fichiers (2)

- Certains systèmes de fichier permettent des **alias** (un fichier peut avoir plus d'un nom)
- UNIX permet les liens **durs** ("hard link") et **symboliques**, qui peuvent introduire des cycles
  Pour créer un lien: `ln [-s] existant nouveau`
- Les répertoires UNIX contiennent toujours les sous-répertoires "`.`" et "`..`" qui sont des liens durs vers le répertoire lui-même et son parent



Identification de Fichiers (3)

- Un **compteur de référence** est requis pour compter les **liens durs** et effacer le fichier lorsque = 0

```
% mkdir test
% cd test
% echo aaa > a
% echo bbbbb > b
% ln b b2
% echo ccccccc > c
% ln c c2
% ln -s c c3
% ls -al
total 36
drwxrwxr-x    2 feeley   feeley        4096 Dec  3 08:28 .
drwxrwxr-x    7 feeley   feeley       12288 Dec  3 08:00 ..
-rw-rw-r--    1 feeley   feeley           4 Dec  3 08:30 a
-rw-rw-r--    2 feeley   feeley           6 Dec  3 08:30 b
-rw-rw-r--    2 feeley   feeley           6 Dec  3 08:30 b2
-rw-rw-r--    2 feeley   feeley           8 Dec  3 08:30 c
-rw-rw-r--    2 feeley   feeley           8 Dec  3 08:30 c2
lrwxrwxrwx    1 feeley   feeley           1 Dec  3 08:30 c3 -> c
% cat c3
ccccccc
% rm c
% cat c2
ccccccc
% cat c3
cat: c3: No such file or directory
% rm c3
% cat c2
ccccccc
% rm c2
```

Identification de Fichiers (4)

- Chaque processus UNIX a un **répertoire de travail présent**, qui peut être obtenu avec `getwd` et se changer avec `chdir`
- Les noms de fichiers non-absolus sont relatifs au répertoire de travail présent
- La **syntaxe des chemins de fichier** varie entre systèmes
  - **UNIX**: `/a/b/c` (absolu), `xyz` et `x/y/z` (relatifs)
  - **Windows**: `C:\a\b\c` (absolu), `\a\b\c` (relatif au disque présent), `xyz` et `x\y\z` (relatifs)
  - **MACOS**: `a:b:c` (absolu sur disque a), `xyz` et `:x:y:z` et `:::x:y:z` (relatifs)

Routines `chdir` et `getwd` (1)

- L'appel système `chdir` change le "répertoire de travail" du processus (`open` se servira de ce répertoire si le nom de fichier est relatif)

```
int chdir (char* path);
```

- L'appel système `getwd` copie le chemin du "répertoire de travail" dans le tableau buf (au plus `PATH_MAX` caractères dans un chemin)

```
char* getwd (char* buf);
```

Routines `chdir` et `getwd` (2)

- Exemple :

```
int main (int argc, char* argv[])
{ char dir[PATH_MAX+1];
  getwd (dir);
  printf ("working dir = %s\n", dir);
  chdir ("/etc");
  getwd (dir);
  printf ("working dir = %s\n", dir);
  return 0;
}
```

- Exécution:

```
% pwd
/home/feeley/ift2245
% ./a.out
working dir = /home/feeley/ift2245
working dir = /etc
```

Routine `creat`

- L'appel système `creat` permet de créer une nouvelle entrée dans un répertoire

```
int creat (char* filename, mode_t mode);
```

- Cela est équivalent à un appel à `open` :

```
open (filename, O_CREAT | O_TRUNC |
O_WRONLY, mode);
```

Routine `umask`

- L'appel système `umask` change le mode de création de fichier du processus

  `mode_t umask (mode_t` $mask$`);`

- Chaque bit à 1 dans le $mask$ est un bit de permission qui sera **retiré** du $mode$ passé à `open` lors de la création d'un fichier

- Le masque de défaut est 022 (ne pas permettre les écritures du groupe ou des autres usagers)

Routines `link`, `symlink` et `unlink` (1)

- Les appels système `link` et `symlink` permettent de créer des liens durs et symboliques vers des fichiers existants, et l'appel système `unlink` permet d'éliminer un lien dur

```
int link (char* existant, char* nouveau);
int symlink (char* existant, char* nouveau);
int unlink (char* existant);
```

- Pour `link` *existant* et *nouveau* doivent être sur le même système de fichier

- `link` incrémente le nombre de liens vers *existant*

- `unlink` décrémente le nombre de liens vers *existant*; lorsque ça tombe à 0 **et** plus aucun processus n'a ce fichier d'ouvert, l'espace de ce fichier est récupéré

Routines link, symlink et unlink (2)

- Exemple :

```
int main (int argc, char *argv[])
{ int fd = creat ("test1", 0644);
  write (fd, "allo\n", 5);
  close (fd);
  link ("test1", "test2");
  link ("test2", "test3");
  unlink ("test1");
  return 0;
}
```

- Exécution:

```
% ./a.out
% ls -l test*
-rw-r--r--   2 feeley  feeley      5 Apr  2 21:27 test2
-rw-r--r--   2 feeley  feeley      5 Apr  2 21:27 test3
% cat test2
allo
% cat test3
allo
```

Routines `stat` et `lstat` (1)

- L'appel système `stat` obtient des informations sur un fichier donné (et `lstat` ne traverse pas les liens symboliques)

  `int stat (char*` *filename*`, struct stat*` *buf*`);`

- Structure `struct stat`:

```
struct stat {
  dev_t           st_dev;       /* device */
  ino_t           st_ino;       /* inode */
  mode_t          st_mode;      /* type and 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 devic
  off_t           st_size;      /* total size, in bytes */
  unsigned long st_blksize;  /* blocksize for filesystem I/
  unsigned long 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 inode change *
};
```

Routines `stat` et `lstat` (2)

- Exemple :

```
void info (char* filename)
{ struct stat s;

  if (lstat (filename, &s) < 0) return;

  printf ("file %s is a ", filename);
  if (S_ISREG(s.st_mode))  printf ("regular file");
  if (S_ISDIR(s.st_mode))  printf ("directory");
  if (S_ISCHR(s.st_mode))  printf ("character device");
  if (S_ISBLK(s.st_mode))  printf ("block device");
  if (S_ISFIFO(s.st_mode)) printf ("fifo");
  if (S_ISLNK(s.st_mode))  printf ("symbolic link");// lsta
  printf (" with size=%ld\n", s.st_size);
}

int main (int argc, char* argv[])
{ info ("/etc/passwd");
  info ("/dev");
  info ("/dev/ttyS0");
  info ("/dev/modem");
  info ("/dev/hda1");
  info ("/u/feeley/x");
  return 0;
}
```

Routines `stat` et `lstat` (3)

- Exécution

```
% mkfifo ~/x
% ls -ld /etc/passwd /dev /dev/ttyS0 /dev/modem /dev/hda1 ~/x
drwxr-xr-x 8 root   root  36864 Dec 31 20:48 /dev
brw-rw---- 1 root   disk  3,  1 May  5  1998 /dev/hda1
lrwxrwxrwx 1 root   root      5 Jun  4  2001 /dev/modem -> ttyS0
crw------- 1 root   tty   4, 64 May  5  1998 /dev/ttyS0
-rw-r--r-- 1 root   root    720 Nov 11  2001 /etc/passwd
prw-r----- 1 feeley users     0 Jan 13  2005 /u/feeley/x
% ./a.out
file /etc/passwd is a regular file with size=720
file /dev is a directory with size=36864
file /dev/ttyS0 is a character device with size=0
file /dev/modem is a symbolic link with size=5
file /dev/hda1 is a block device with size=0
file /u/feeley/x is a fifo with size=0
% cat < ~/x &
% cat > ~/x
hello
hello
world
world
<CTRL-D>
%
```

Routines `chmod` et `chown`

- Les appels système `chmod` et `chown` permettent de modifier les permissions et le propriétaire et groupe d'appartenance d'un fichier

```
int chmod (char* filename, mode_t mode);
int chown (char* filename, uid_t u, gid_t g);
```

-1 pour u ou g ne modifie pas ce champ

Routine `utime` (1)

- L'appel système `utime` permet de modifier les dates de dernier accès et de dernière modification

  `int utime (char*` $filename$`, struct utimbuf*` am`);`

  am pointe vers une structure contenant les dates de dernier accès et de dernière modification

- Structure `struct utimbuf`:

```
struct utimbuf {
  time_t actime;  /* access time */
  time_t modtime; /* modification time */
};
```

- `time_t` est un type entier; le nombre de **secondes depuis l'epoch** (minuit, 1 janvier 1970)

Routine utime (2)


```
void info (char *filename, time_t *access, time_t *modif)
{ struct stat s;

  if (lstat (filename, &s) < 0) exit (1);

  *access = s.st_atime;   *modif = s.st_mtime;

  printf ("access = %s", ctime (access));
  printf ("modif  = %s", ctime (modif));
}

int main (int argc, char *argv[])
{ char *filename = argv[1];
  time_t access, modif;
  struct utimbuf t;

  info (filename, &access, &modif);

  t.actime = access-10;   t.modtime = 0;

  if (utime (filename, &t) < 0) exit (1);

  info (filename, &access, &modif);

  return 0;
}
```

Routine utime (3)

- Exécution

```
% date
Sun Apr  2 20:41:42 EDT 2006
% echo allo > test
% date
Sun Apr  2 20:42:15 EDT 2006
% cat test
allo
% date
Sun Apr  2 20:42:22 EDT 2006
% ls -l test
-rw-r--r--   1 feeley  feeley  5 Apr  2 20:41 test
% ./a.out test
access = Sun Apr  2 20:42:20 2006
modif  = Sun Apr  2 20:41:55 2006
access = Sun Apr  2 20:42:10 2006
modif  = Wed Dec 31 19:00:00 1969
% ls -l test
-rw-r--r--   1 feeley  feeley  5 Dec 31  1969 test
```

Routines `mkdir` et `rmdir` (1)

- La création d'un répertoire UNIX se fait avec `mkdir`

  `int mkdir (char*` $path$`, mode_t` $mode$`);`

- Le $mode$ indique les permissions d'accès:

  0700 = lecture, écriture, accès par nom (propriétaire)

  0070 = lecture, écriture, accès par nom (groupe)
  0007 = lecture, écriture, accès par nom (autres usagers)

  En $mode$ "lecture", on peut obtenir l'ensemble des noms de fichiers dans le répertoire

  En $mode$ "accès par nom", on peut seulement accéder à un fichier si on connait son nom (et on a en plus les permissions nécessaires sur ce fichier)

- `rmdir` élimine un répertoire **vide**

  `int rmdir (char*` $path$`);`

Routines mkdir et rmdir (2)

- Exemple

```
/* create directory "test" listable by group and
 * accessible by specific name by all, and file
 * "test/xxx" readable by everyone. */

int fd;
if (mkdir ("test", 0751) < 0)
  perror ("mkdir");
else if ((fd = creat ("test/xxx", 0444)) < 0)
  perror ("creat");
else if (close (fd) < 0)
  perror ("close");
else if (rmdir ("test") < 0)
  perror ("rmdir");
```

```
% ./a.out
rmdir: Directory not empty
% ls -la test
total 16
drwxr-x--x  2 feeley feeley   4096 Jan  2 16:35 .
drwx--x--x  7 feeley feeley  12288 Jan  2 16:35 ..
-r--r--r--  1 feeley feeley      0 Jan  2 16:35 xxx
% ./a.out
mkdir: File exists
```

Routines `opendir`, `readdir`, etc (1)

- `opendir` permet d'énumérer les entrées d'un répertoire

```
DIR* opendir (char* path);
struct dirent* readdir (DIR* dir);
int closedir (DIR* dir);
```

- Structure `struct dirent`:

```
struct dirent {
  ...
  unsigned char d_type;  /* file type */
  char d_name[256];      /* file name */
};
```

Routines opendir, readdir, etc (2)

- Exemple : parcours récursif d'un répertoire

```
% ./a.out /home/feeley/test
|-> .
|-> ..
|-> readme       [size=1000]
|-> doc
|     |-> .
|     |-> ..
|     |-> report.doc      [size=10000]
|     |-> report.ps       [size=50000]
|-> test.c       [size=2000]
|-> bin
|     |-> .
|     |-> ..
|     |-> i386
|     |     |-> .
|     |     |-> ..
|     |     |-> a.out     [size=4000]
|     |     |-> find      [size=8000]
|     |-> ppc
|     |     |-> .
|     |     |-> ..
|     |     |-> a.out     [size=9000]
total = 84000 bytes
```

```
int list (char* path, int indent)
{ int i, n = 0;
  char old[PATH_MAX+1];
  getwd (old);
  if (chdir (path) == 0)
    { DIR* dir = opendir (".");
      if (dir != NULL)
        { struct dirent* e;
          while ((e = readdir (dir)) != NULL)
            { struct stat s;
              for (i=0; i<indent; i++) printf ("|    ");
              printf ("|-> %s", e->d_name);
              if (lstat (e->d_name, &s) == 0)
                { if (S_ISDIR(s.st_mode))
                    { printf ("\n");
                      if (strcmp (e->d_name, ".") != 0 &&
                          strcmp (e->d_name, "..") != 0)
                        n += list (e->d_name, indent+1);
                    }
                  else
                    { printf ("\t[size=%ld]\n", s.st_size);
                      n += s.st_size;
                    }
                }
            }
          closedir (dir);
        }
      chdir (old);
    }
  return n;
}
```

```
int main (int argc, char* argv[])
{ printf ("total = %d bytes\n", list (argv[1], 0));
  return 0;
}
```

Routines `lseek` et `lseek64` (1)

- Le cannal d'E/S créé lors de l'ouverture d'un fichier mémorise la **position dans le fichier du prochain octet qui sera lu ou écrit**

- Les appels système `lseek` et `lseek64` permettent de connaître la position et la changer

```
off_t lseek (int fd, off_t pos, int rel);
off64_t lseek64 (int fd, off64_t pos, int rel);
```

`off_t` et `off64_t` sont des types entiers représentant une position

rel est un code qui indique comment interpréter pos :
`SEEK_SET` = à partir du début,
`SEEK_CUR` = à partir de la position présente,
`SEEK_END` = à partir de la fin

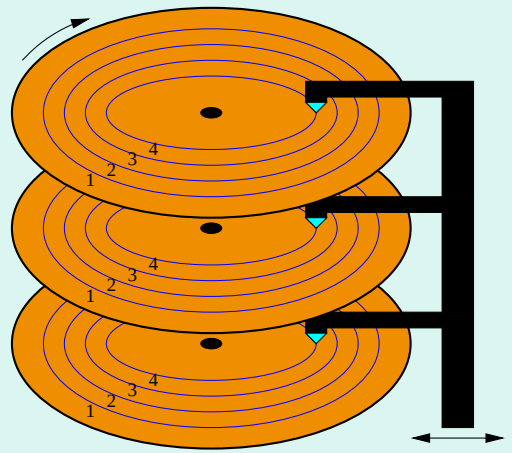
```
#define N 100000

struct dossier { char nom[43], tel[11], nas[10]; };

int tab; /* descripteur de fichier de la table */

void init (void)
{ int i = N;
  struct dossier d;
  d.nom[0] = '\0';
  while (i-- > 0) write (tab, &d, sizeof (d));
}

void lire (struct dossier* d, int i)
{ lseek (tab, i * sizeof(struct dossier), SEEK_SET);
  read (tab, (char*)d, sizeof(struct dossier));
}

void ecrire (struct dossier* d, int i)
{ lseek (tab, i * sizeof(struct dossier), SEEK_SET);
  write (tab, (char*)d, sizeof(struct dossier));
}
```

```
int hash (char* str)
{ char* p = str;
  int h = 0;
  while (*p != '\0')
    h = (((h>>8) + *p++) * 331804471) & 0x7fffffff;
  return h;
}

void ajouter (struct dossier* d)
{ struct dossier t;
  int i = hash (d->nom) - 1;
  do { i = (i+1) % N;
       lire (&t, i);
     } while (t.nom[0] != '\0' &&
              strcmp (d->nom, t.nom) != 0);
  ecrire (d, i);
}

void chercher (char* nom, struct dossier* d)
{ int i = hash (nom) - 1;
  do { i = (i+1) % N;
       lire (d, i);
     } while (d->nom[0] != '\0' &&
              strcmp (nom, d->nom) != 0);
}
```

```
int main (int argc, char *argv[])
{
  struct dossier d;

  tab = open ("bd", O_RDWR|O_CREAT, 0644);

  init ();

  strcpy (d.nom, "etienne");
  strcpy (d.tel, "3435766");
  strcpy (d.nas, "111111111");
  ajouter (&d);

  strcpy (d.nom, "marc");
  strcpy (d.tel, "5551212");
  strcpy (d.nas, "222222222");
  ajouter (&d);

  chercher ("etienne", &d);

  printf ("tel = %s\n", d.tel);

  return 0;
}
```

Disque Dur (1)

- Mémoire secondaire la plus répandue = **disque dur**
- Dans un enclos hermétique (sans poussière) un certain nombre de **plateaux** recouverts d'une substance magnétique tournent rapidement (p.e. 5400 RPM)
- Un ensemble de têtes de lectures qui peuvent **lire** et **écrire** des informations sur les plateaux se déplacent en unisson ("**seek**") en fonction des commandes envoyées par le contrôleur de disque dur



Disque Dur (2)

- Chaque plateau contient un certain nombre de **pistes** (“track”); chaque piste contient un certain nombre de **secteurs**
- L’ensemble des $i^{\text{ème}}$ pistes de tous les plateaux forment un **cylindre**
- La capacité totale du disque est

$$\text{capacité} = N_P * N_T * N_S * OS$$

où N_P=nombre de pistes par plateau, N_T=nombre de têtes (normalement 2 par plateau), N_S=nombre de secteurs par piste, OS=nombre d’octets par secteur

- Exemple typique: $N_P = 1011$, $N_T = 15$, $N_S = 44$, $OS = 512$, capacité=325MB

Disque Dur (3)

- L'unité de transfert de donnée est le **secteur** (typiquement de 512 à 4096 octets)
- Chaque secteur a une adresse **"CHS"** qui l'identifie (sur PC: Cylinder=0..$N_P - 1$, Head=0..$N_T - 1$, Sector=1..N_S)
- L'approche **"LBA"** (Logical Block Addressing) est une autre façon de numéroter les secteurs (de 0 à $N - 1$, où N = nombre total de secteurs)

$$\text{adresse LBA} = (C * N_T + H) * N_S + S - 1$$

Disque Dur (4)

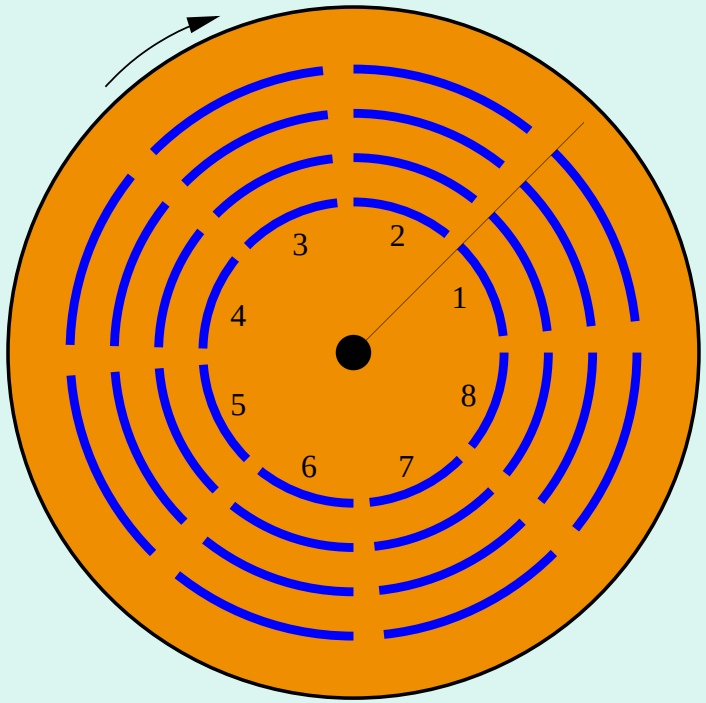
- Le temps d'accès dépend du temps de positionnement de la tête sur la bonne piste ST (**"seek time"**), la latence additionnelle de rotation du disque RL (**"rotational latency"**), le temps de transfert des secteurs accédés TT

$$\text{temps d'accès} = ST + RL + TT$$

- Valeurs typiques: ST = 0 .. 40 msecs, RL = 0 .. 11 msecs (pour un disque 5400 RPM)
- ST dépend de la distance de déplacement d (nombre de pistes), l'accélération maximale des têtes, la vitesse maximale des têtes, et le temps de stabilisation
- Un modèle simple: $ST = d * k_1 + k_2$
- Une mesure moyenne de ST = 1/3 du ST maximal

Disque Dur (5)

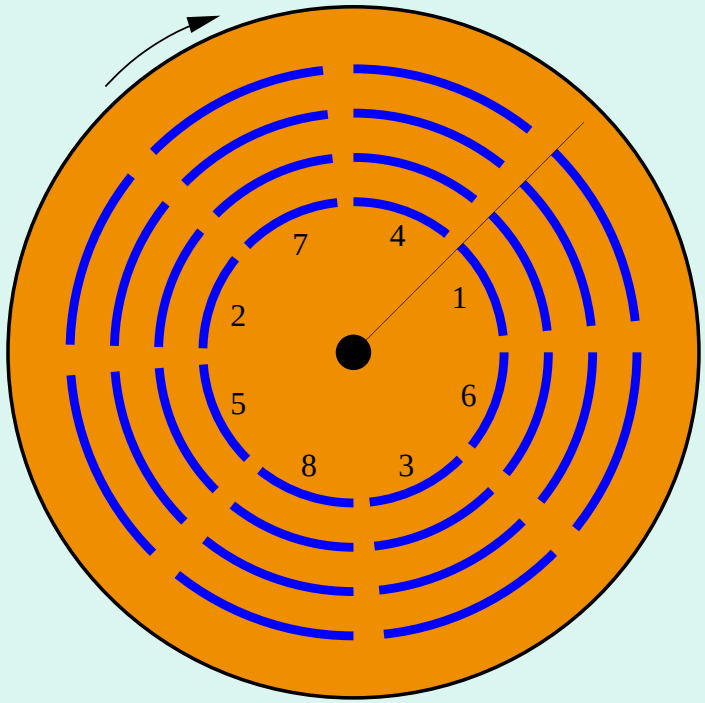
- Une organisation simple est de placer le secteur $i + 1$ immédiatement après le secteur i



- Dans ce cas, RL est très petit si après le secteur i on accède au secteur $i + 1$

Disque Dur (6)

- Lorsque le temps entre les accès aux secteurs i et $i+1$ n'est pas petit, il peut être avantageux d'**entrelacer les secteurs** pour que l'accès au secteur $i+1$ n'ait pas à attendre une rotation complète du disque



- L'entrelacement optimal dépend de la vitesse du processeur, du SE, de l'application, etc.

Disque Dur (7)

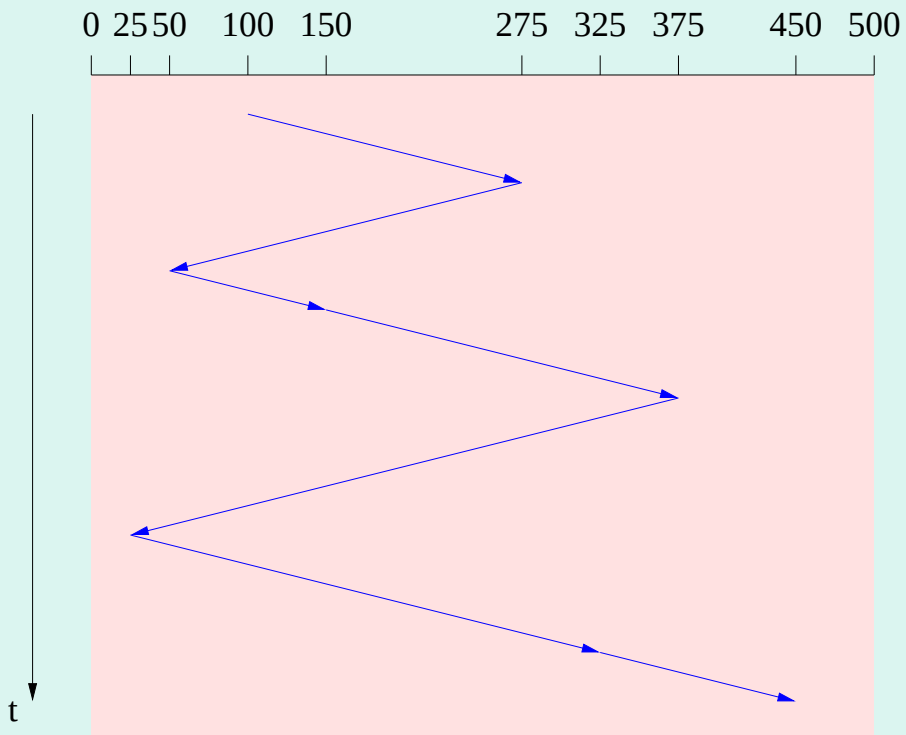
- Les disques récents n'ont pas une géométrie CHS fixe
- Le nombre de secteurs par piste n'est pas constant, il **augmente à la périphérie du disque** pour que la densité des bits par unité de surface soit approximativement constant
- La relation entre CHS et LBA est plus complexe
- Pour demeurer compatible avec les vieux SE qui supposent une géométrie CHS fixe dans leurs échanges avec le contrôleur de disque, le disque déclare des valeurs N_P, N_T et N_S qui ne correspondent pas à la réalité et **font eux même une traduction de ce CHS logique en un CHS physique** (par exemple $N_P = 16383$, $N_T = 16$ et $N_S = 63$, pour un disque de 8GB)

Ordonnancement des Accès au Disque

- Dans un contexte **multitasking**, plusieurs processus peuvent être en attente d'un accès au disque
- Puisque le disque est relativement lent il est avantageux d'**ordonnancer les accès au disque**
- L'ordonnanceur d'accès au disque peut s'implanter comme un **processus serveur**; les processus clients envoient des requêtes d'accès qui sont mis dans une **file d'attente**
- L'ordonnanceur choisit **l'ordre dans lequel ces requêtes sont traitées** de façon à **maximiser le débit du disque**

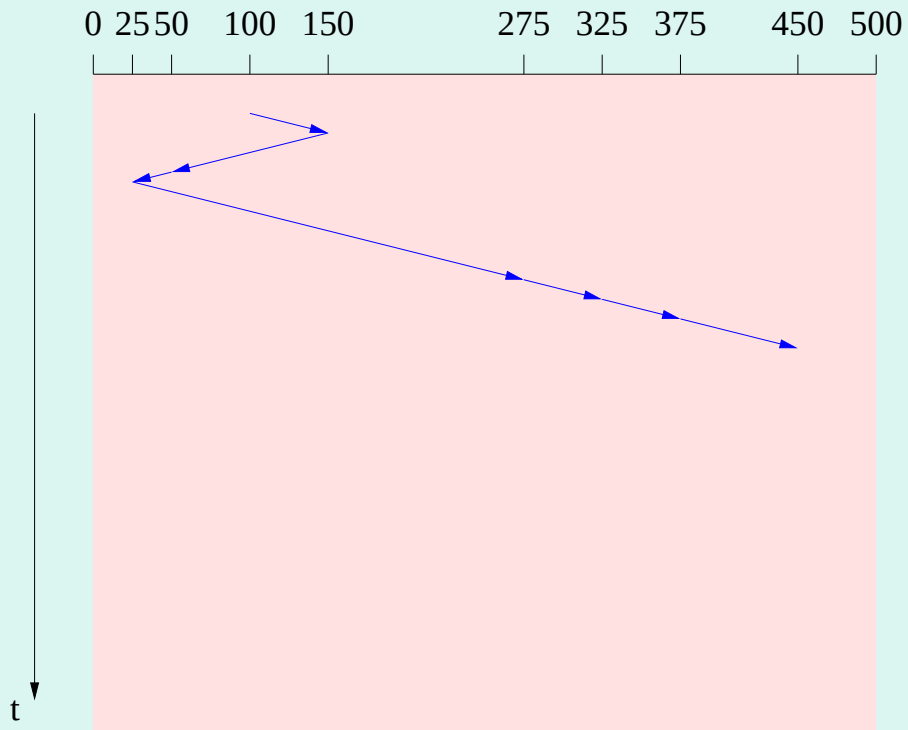
Ordonnancement “FCFS”

- Une approche simple consiste à traiter les requêtes dans l’ordre d’arrivée; premier arrivé premier servi (“FCFS”)
- Exemple : des requêtes arrivent dans l’ordre de numéro de piste 100, 275, 50, 150, 375, 25, 325, 450



Ordonnancement “SSTF”

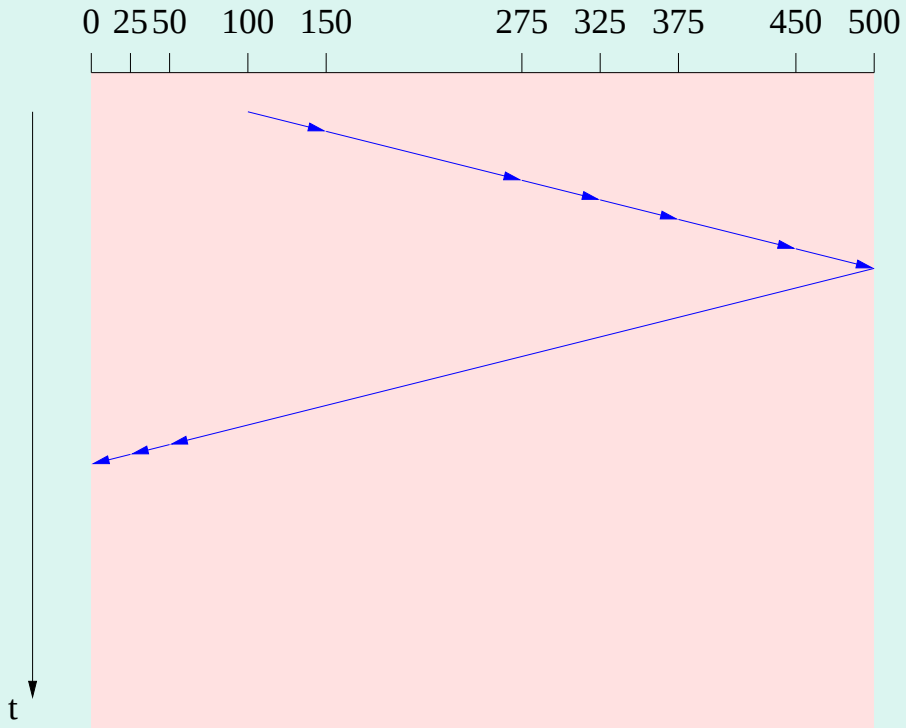
- L’approche SSTF (Shortest Seek Time First) tente de réduire ST en traitant les accès proches en premier



- Il y a un danger de “famine”, si des accès proches arrivent sans arrêt

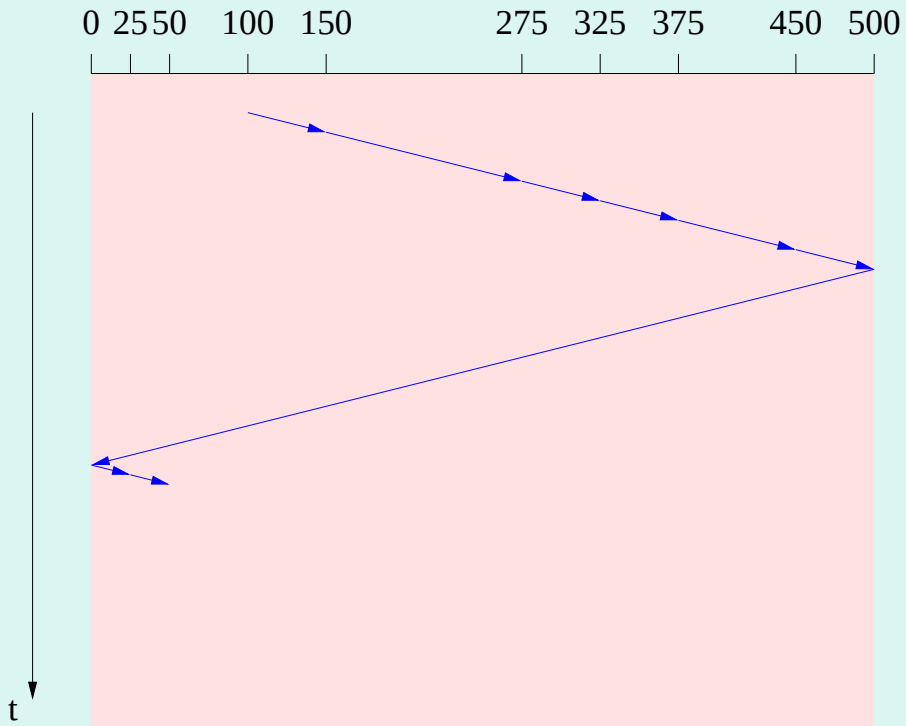
Ordonnancement “SCAN”

- L’approche SCAN est aussi connu sous le nom d’**algorithme de l’ascenseur**: les têtes changent de direction seulement aux extrémités du disque



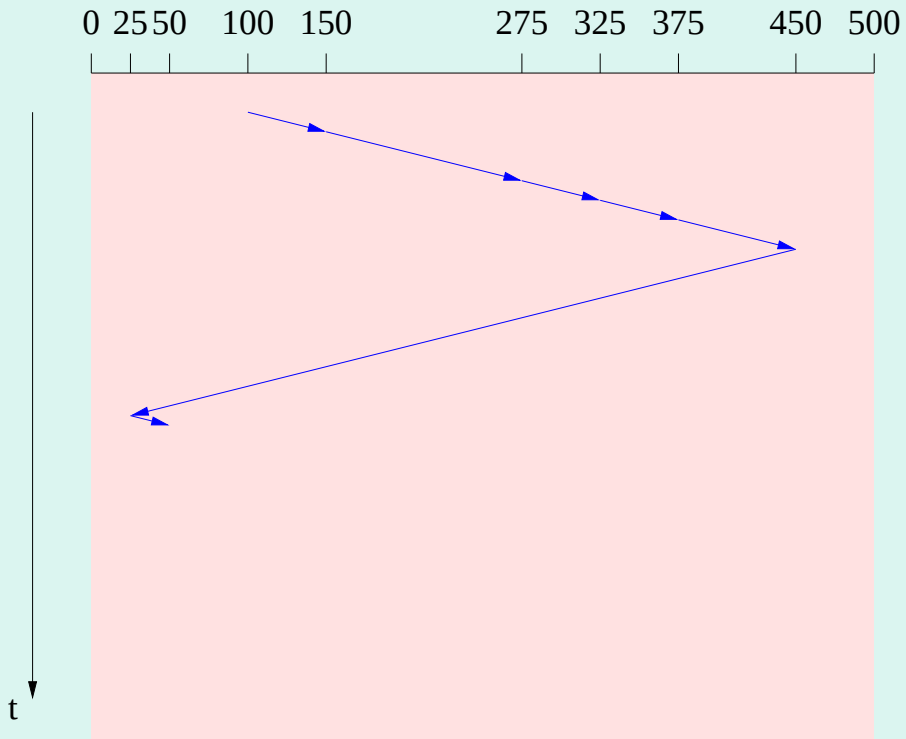
Ordonnancement “C-SCAN”

- L’approche SCAN cyclique est comme SCAN mais les accès se font seulement pendant la phase montante, ce qui donne une distribution plus uniforme du temps d’attente



Ordonnancement “C-LOOK”

- L’approche C-LOOK est comme C-SCAN mais les déplacements de tête sont limités à la piste minimale et maximale

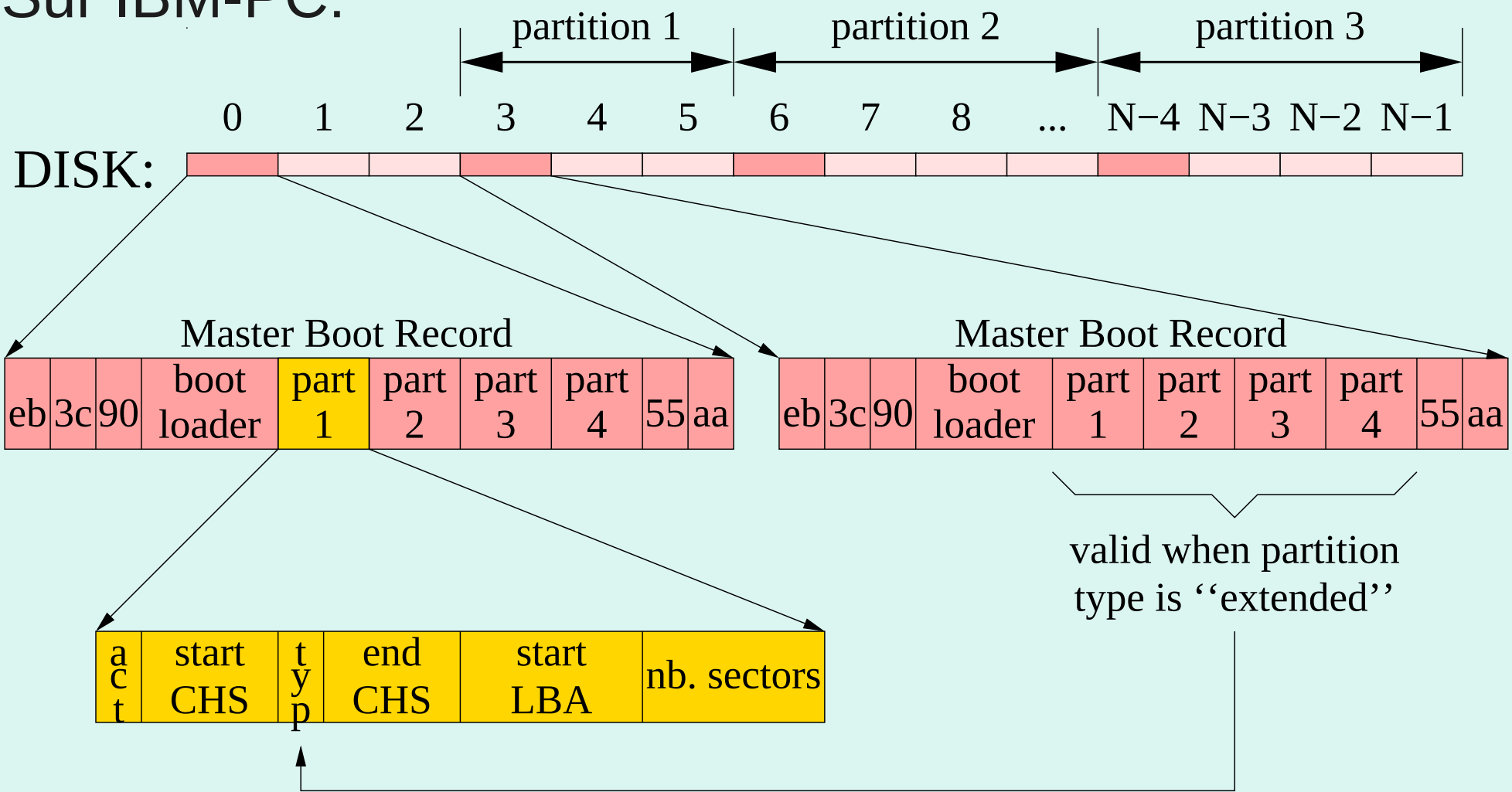


Comparaison

- SSTF est **facile à implanter** mais il y a danger de famine
- SCAN et C-SCAN ont des bonnes performances lorsque la charge du disque est élevée
- Le patron et fréquence d'accès au disque, l'organisation des données sur disque ("file system"), et le type d'application sont tous à considérer dans le choix d'un algorithme d'ordonnancement
- SSTF et C-LOOK ont des bonnes performances en général
- Ces algorithmes peuvent aussi être modifiés pour tenir compte de la **priorité des processus**

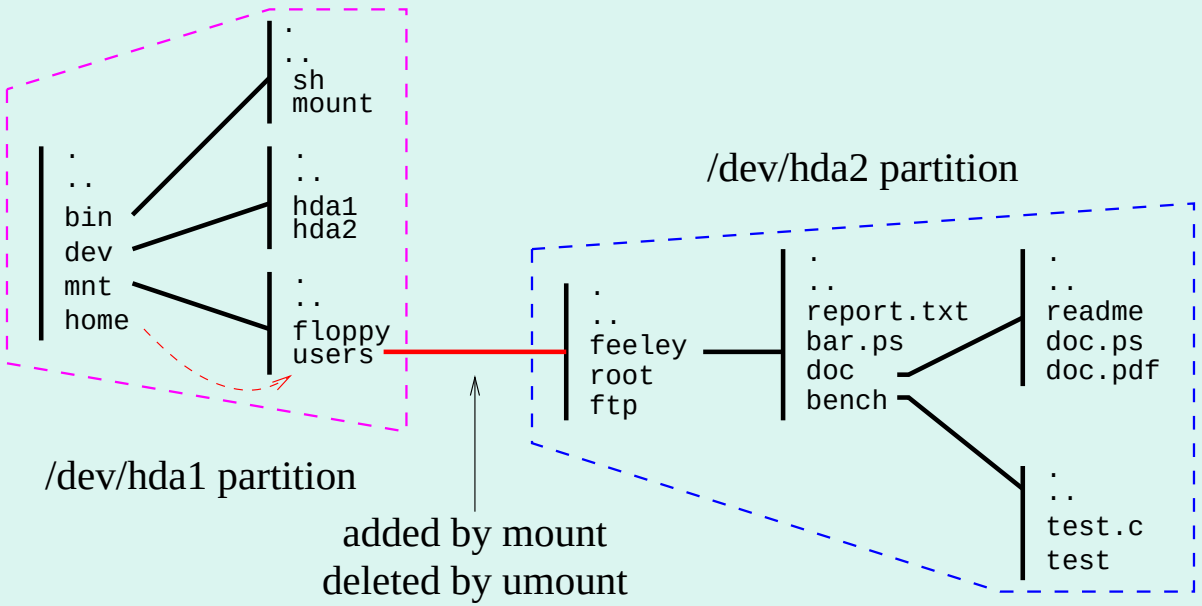
Organisation du Disque Dur

- Une **partition** est un disque abstrait (il peut y avoir plusieurs partitions sur un même disque, ou une partition peut couvrir plusieurs disques)
- Sur IBM-PC:



Montage de Partition (1)

- Sous UNIX, chaque partition qui contient un système de fichier utile doit être montée avec `mount`
- à la commande `mount` il faut spécifier: le device correspondant à la partition à monter (par exemple `/dev/hda2`) et le répertoire où il faut **greffer** ce système de fichier
- Par exemple: `mount /dev/hda2 /mnt/users`



Montage de Partition (2)

- Au montage d'une partition, le SE s'assure que le système de fichier ne contient pas d'incohérences et initialise les structures de données internes pour permettre l'accès à cette partition

- Le programme `fsck` vérifie la cohérence d'une partition

- Cela se fait automatiquement au démarrage du système (par `init`), en consultant le fichier `/etc/fstab`

```
# /etc/fstab
/dev/hda1   /             ext2    defaults          1 1
/dev/hda2   /home         ext2    defaults          1 2
/dev/hda3   swap          swap    defaults          0 0
/dev/hdb1   /win95        fat     defaults          0 0
/dev/fd0    /mnt/floppy   auto    noauto,owner      0 0
/dev/cdrom  /mnt/cdrom    iso9660 noauto,unhide,owner,ro 0 0
```

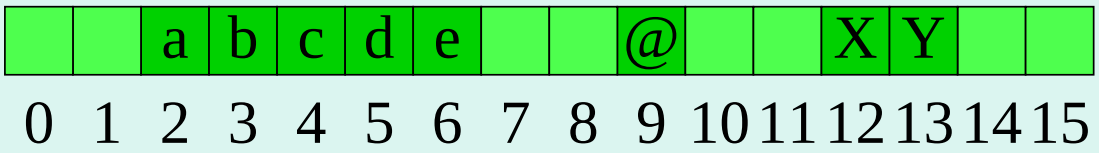
Allocation de l'Espace (1)

- Le disque contient principalement des **fichiers réguliers** (contenant une séquence d'octêts) et des **répertoires** (contenant une séquence d'entrées)
- Souvent, les **répertoires sont traités comme des fichiers** dont le contenu décrit les entrées du répertoire (selon un format fixe ou variable)
- Le problème de l'allocation de l'espace disque est similaire à celui de l'allocation de la mémoire principale
- À cause de la lenteur des disques, leur grande capacité, et les différences de manipulation des données, des **algorithmes spécialisés** sont utilisés
- L'espace disque est normalement alloué en **blocs** dont la taille est un multiple de secteurs; il y a un **espace inutilisé** dans les blocs qui ne sont pas pleins

Allocation de l'Espace (2)

- Une méthode simple consiste à allouer tous les blocs de chaque fichier dans un **espace contigü**

file 1: length=5 start=2
file 2: length=1 start=9
file 3: length=2 start=12

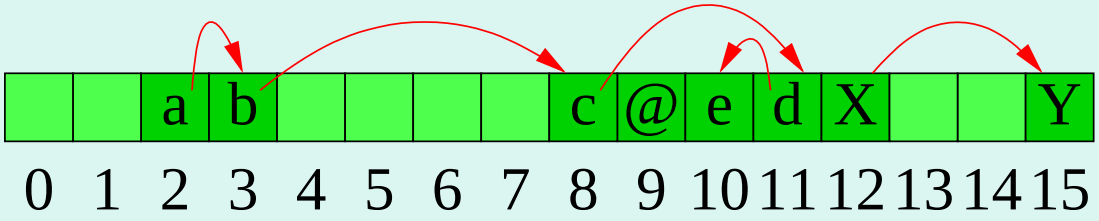


- La représentation est **compacte** (à part l'entrée du répertoire, il faut uniquement des blocs pour le contenu du fichier)
- Les **accès séquentiels et aléatoires sont rapides** (étant donné la position d'un octêt, il est facile de trouver le bloc qui contient cet octet)
- Pour **étendre un fichier**, il faut possiblement **le copier**
- **Fragmentation** de l'espace libre

Allocation de l'Espace (3)

- Pour ne pas avoir à déplacer les fichiers, les blocs peuvent être **chaînés**

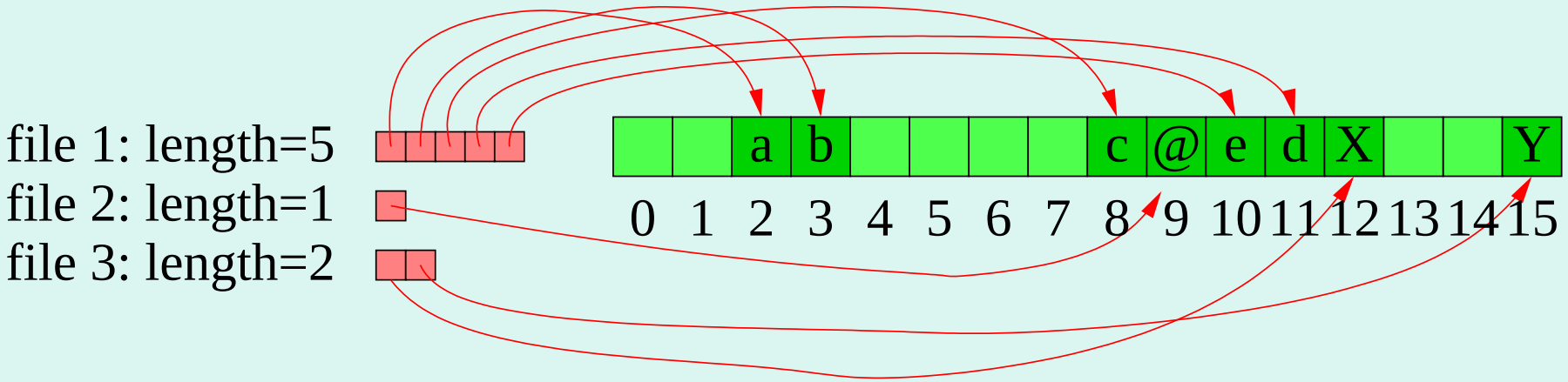
file 1: length=5 start=2
file 2: length=1 start=9
file 3: length=2 start=12



- Il faut stocker un pointeur additionnel par bloc
- Accès séquentiel rapide; **accès aléatoires lent**: il faut suivre la chaîne pour trouver le $n^{\text{ème}}$ bloc
- Lors de l'extension d'un fichier, il est bon d'allouer le prochain bloc le plus proche du dernier (pour minimiser les déplacements de tête), ce qui est simplifié par l'utilisation d'un **bitmap des blocs libres**
- **Fragmentation** minimale de l'espace libre

Allocation de l'Espace (4)

- Une autre approche conserve pour chaque fichier la position des blocs dans une table, pour pouvoir les **indexer** rapidement

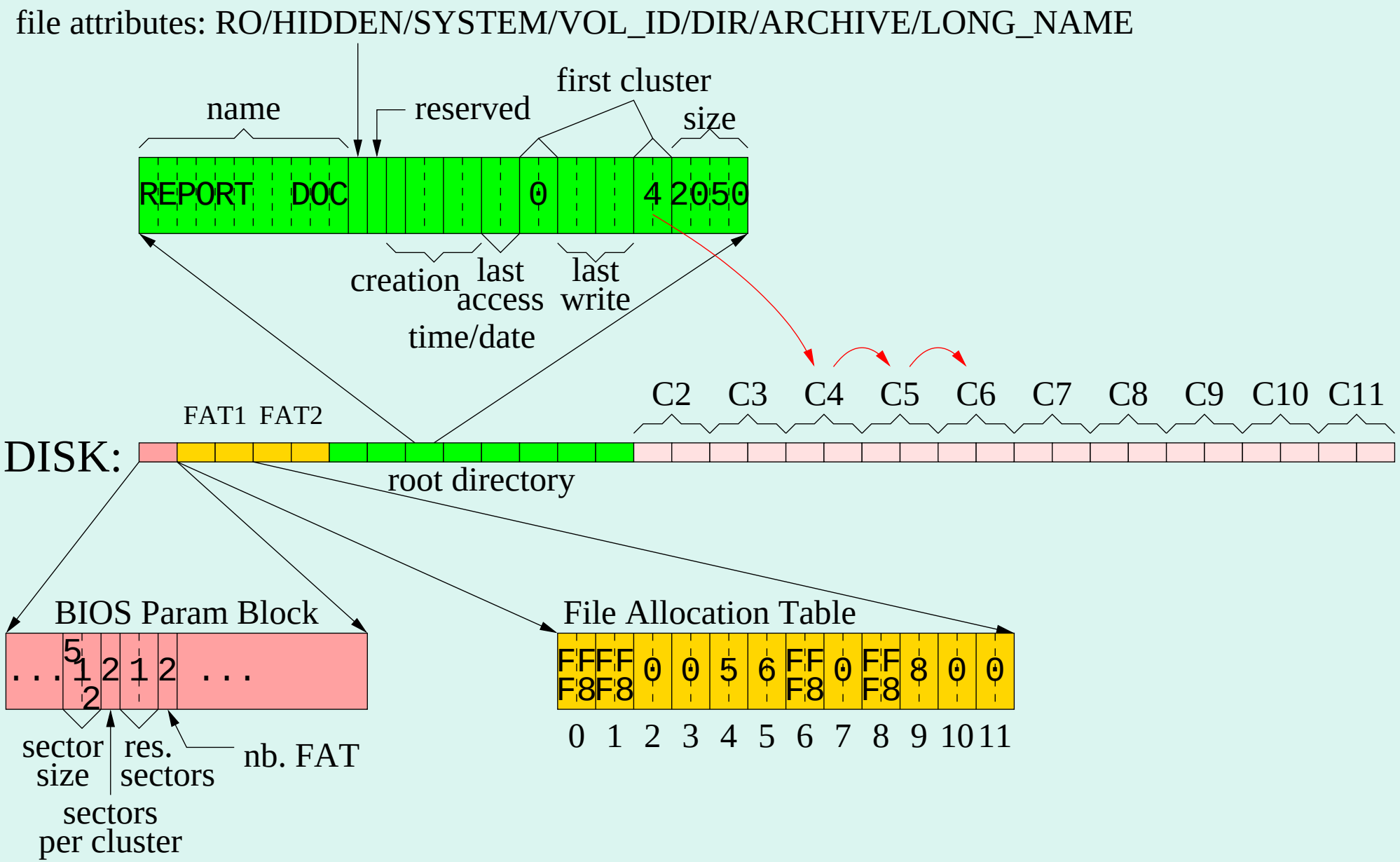


- Les **accès séquentiels et aléatoires sont rapides**
- **La table des blocs doit être étendue** lorsque le fichier est étendu
- **Fragmentation** minimale de l'espace libre

Le Système de Fichier FAT16 (1)

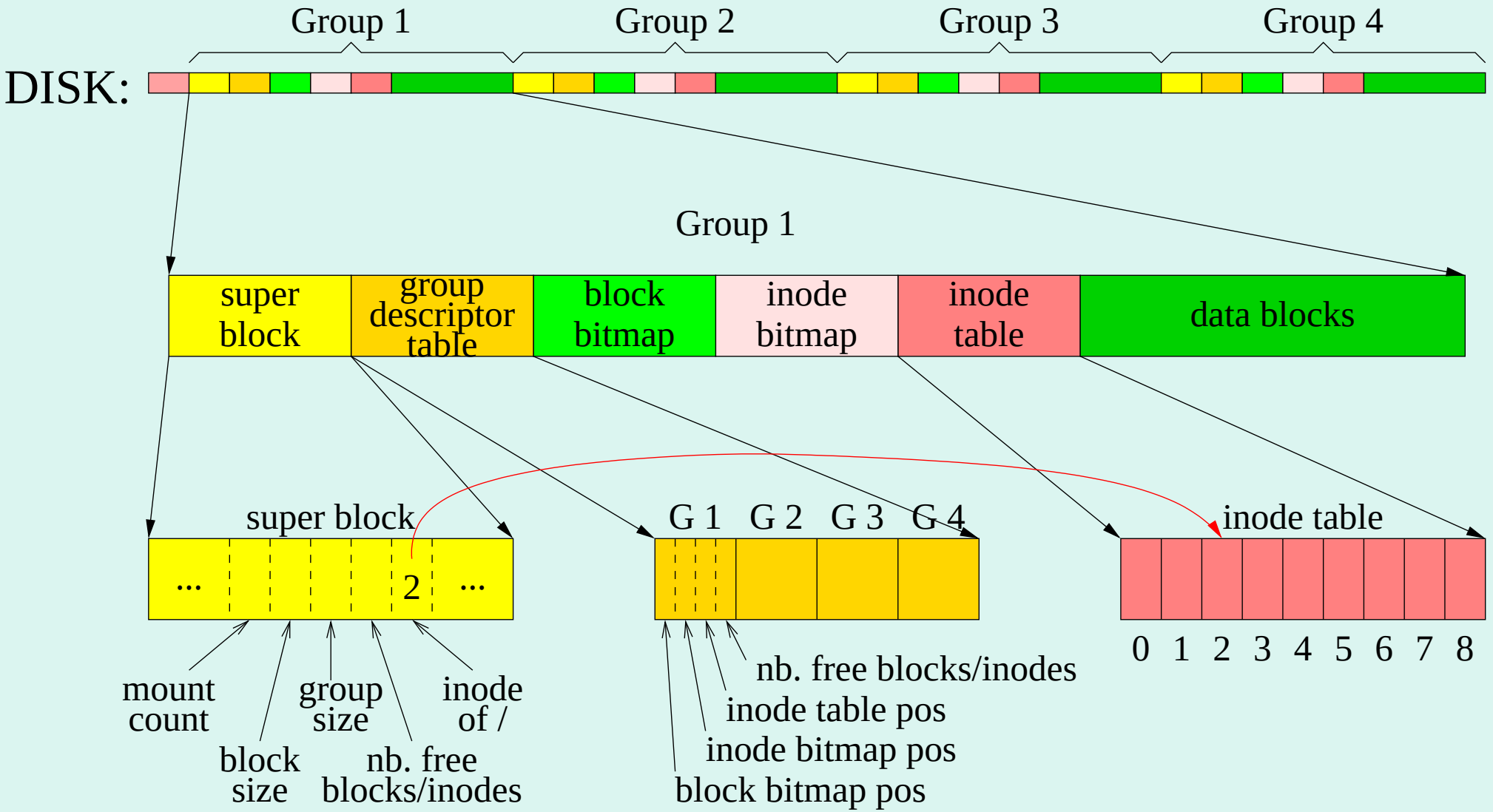
- Chaque fichier est une **chaîne de “clusters”** (cluster = 1, 2, 4, 8, 16, 32, 64 ou 128 secteurs)
- Le **“File Allocation Table”** (FAT) est un tableau d’entiers de **16 bits** qui chaîne les clusters des fichiers et indique quels clusters sont libres
  - 0 = libre
  - 0xFFF7 = cluster defectueux
  - 0xFFF8 = dernier
  - autre = pointeur vers suivant
- Puisqu’il y a un maximum de 65000 clusters, les disques à grande capacité doivent utiliser des clusters de 128 secteurs (64KBytes), ce qui donne une **très mauvaise utilisation de l’espace pour les fichiers courts**

Le Système de Fichier FAT16 (2)



Le Système de Fichier UNIX (1)

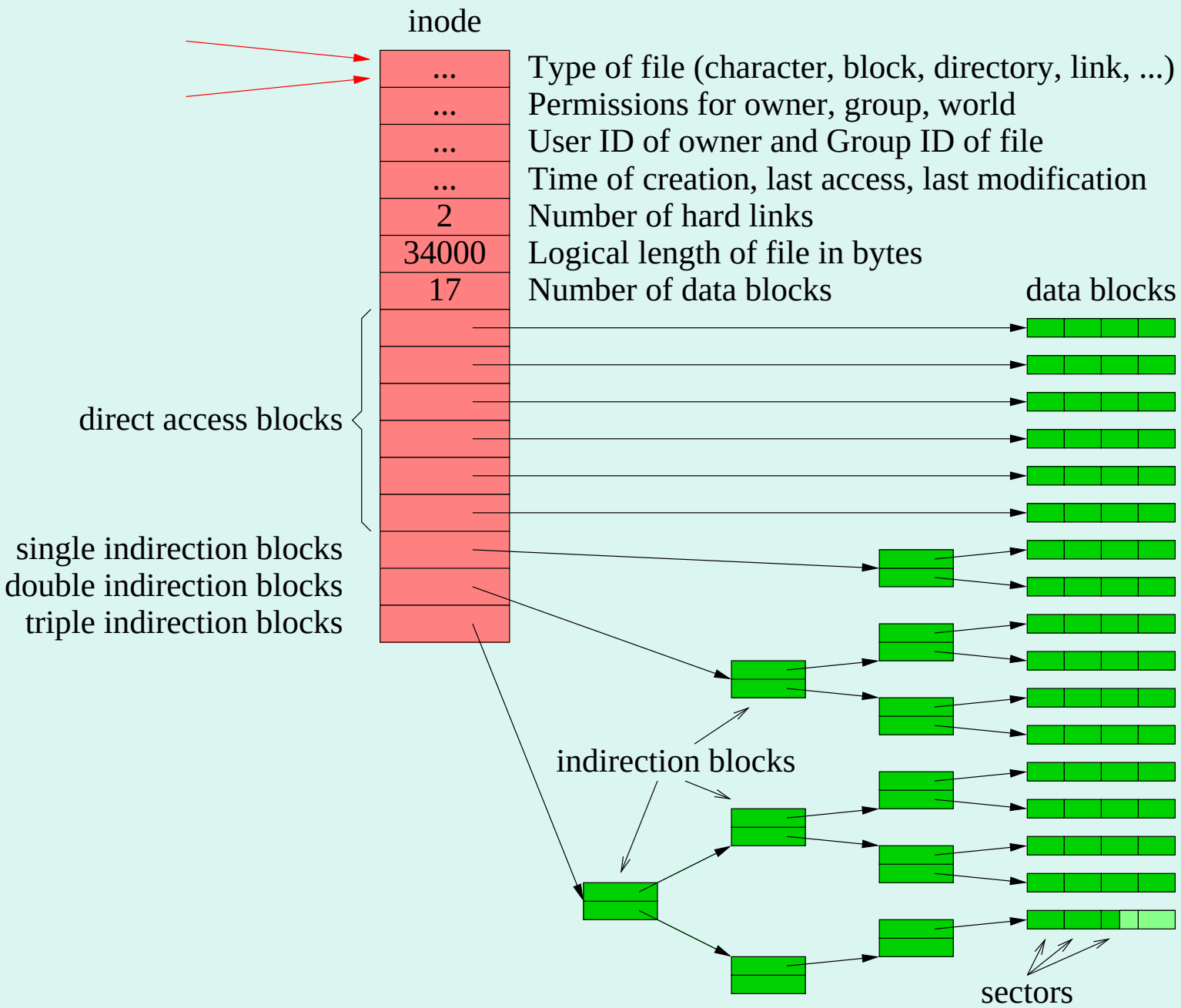
- Un fichier est représenté par un “**inode**”; les répertoires associent un (ou des) nom(s) à un inode donné



Le Système de Fichier UNIX (2)

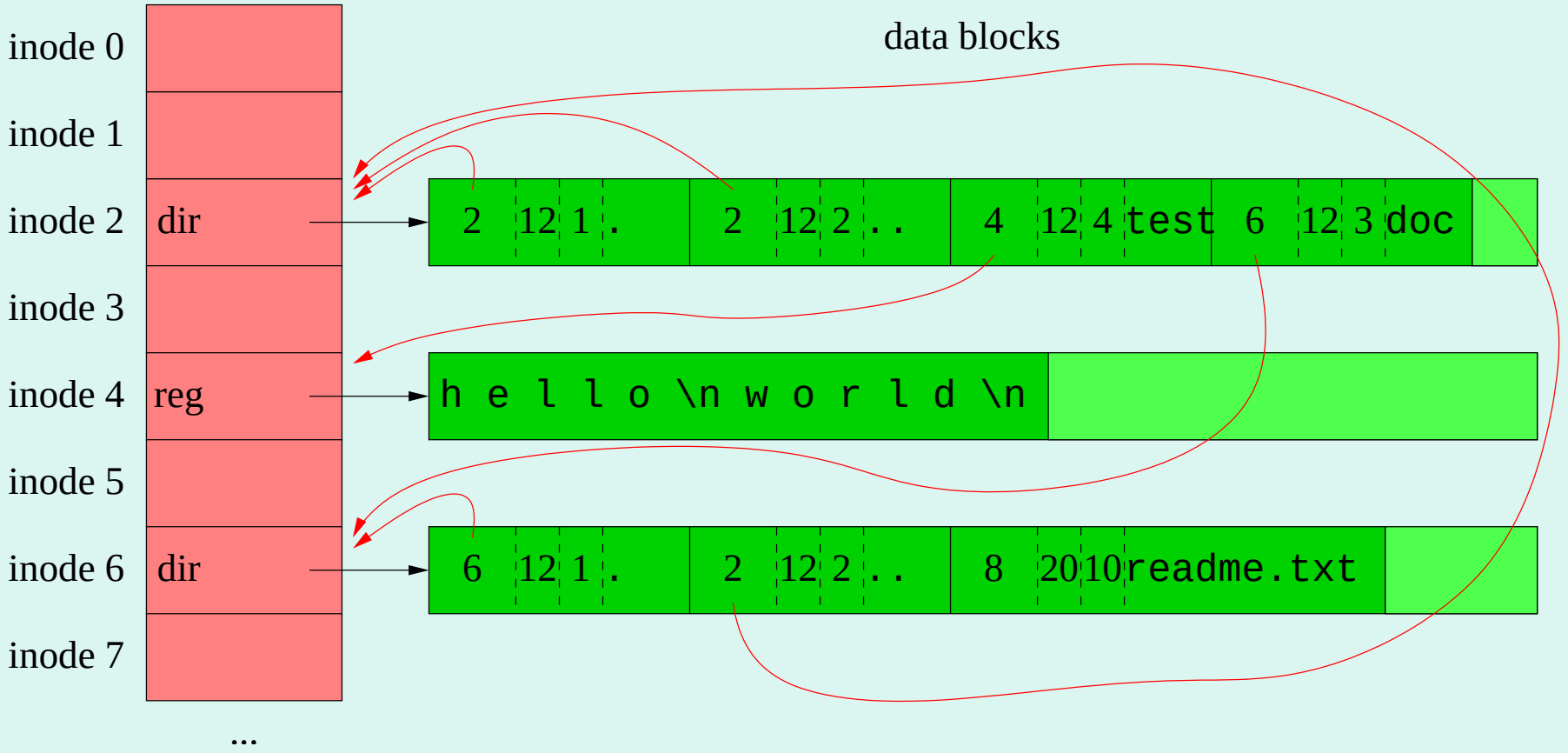
- Chaque groupe contient une copie du premier “super block” et “group descriptor table”, ce qui permet de récupérer le contenu du disque suite à une panne
- Pour améliorer la localité:
  - l’allocation d’un nouveau inode se fait (si possible) dans le même groupe que le répertoire parent
  - l’allocation d’un nouveau bloc pour étendre un fichier se fait (si possible) dans le même groupe
- Les blocs d’un fichier sont accédés par **indexation multiple**
- Les fichiers courts (qui sont statistiquement fréquents) n’ont pas besoin d’indirection

Le Système de Fichier UNIX (3)



Le Système de Fichier UNIX (4)

- Les entrées des répertoires contiennent simplement un **nom** et un **numéro de inode**



Fiabilité (1)

- Si le disque est corrompu, en particulier à l'endroit où est stocké un répertoire ou le FAT, il est **impossible de savoir quels blocs vont avec quels fichiers**
- Des utilitaires comme `CHKDSK` et `SCANDISK` (Windows) et `fsck` (UNIX) permettent de réparer le système de fichier suite à une perte mineure d'information
- Ces utilitaires vérifient la cohérence des informations du disque
  - bloc marqué libre jamais utilisé dans un fichier
  - bloc marqué occupé utilisé dans un seul fichier
  - longueur en octets qui concorde au nombre de blocs utilisés
  - absence de cycles dans les repertoires
  - compteur de référence correct

Fiabilité (2)

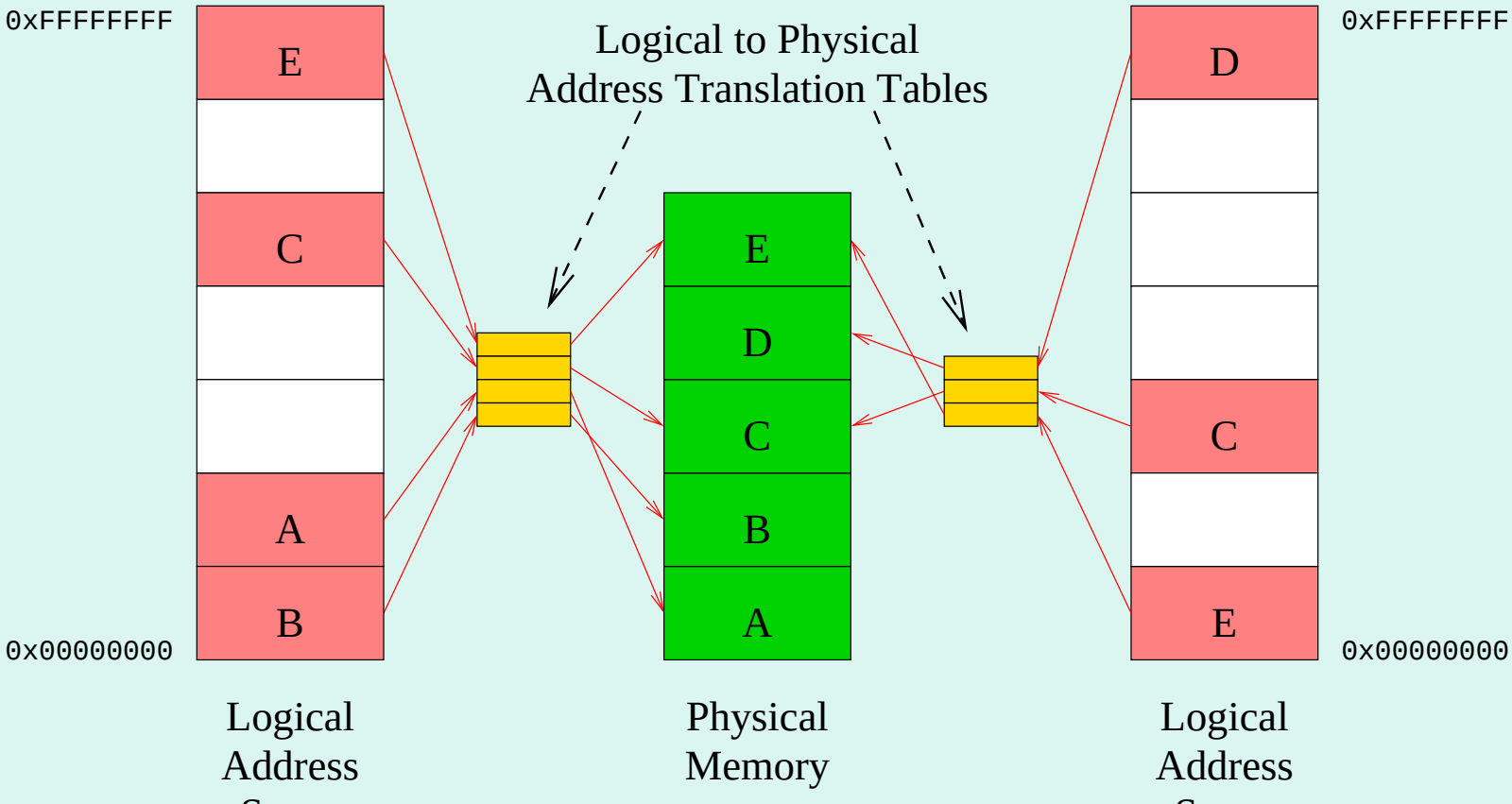
- Ces utilitaires se font appeler: lorsqu'un disque est monté sans avoir été dé-monté (signe d'un arrêt anormal de l'ordinateur), après avoir monté le disque un certain nombre de fois (par précaution)
- Les algorithmes utilisés sont essentiellement ceux qu'on retrouve dans un **garbage collector** pour récupérer la mémoire principale:
  - visiter le répertoire racine
  - pour chaque répertoire visité, visiter les entrées du répertoire
  - pour chaque fichier régulier visité, parcourir la liste des blocs associés
- Après il faut vérifier que les bitmaps de blocs libres (ou le FAT) sont cohérents avec l'information accumulée

Mémoire Virtuelle (1)

- Dans un système multitasking, on désire
  - avoir un **espace d'adressage propre à chaque processus**: chaque processus à l'illusion d'avoir un espace d'adressage qui ne change pas d'une exécution à l'autre (de 0 à N); **compilation et édition de liens plus simple**; les processus sont isolés
  - assigner la mémoire principale aux processus en fonction des besoins de chaque processus; les **besoins peuvent changer pendant l'exécution**
- Solution: la **mémoire virtuelle**
  - espace d'adressage **physique** (niveau matériel)
  - espace d'adressage **logique** ou **virtuel** (utilisé par les processus)

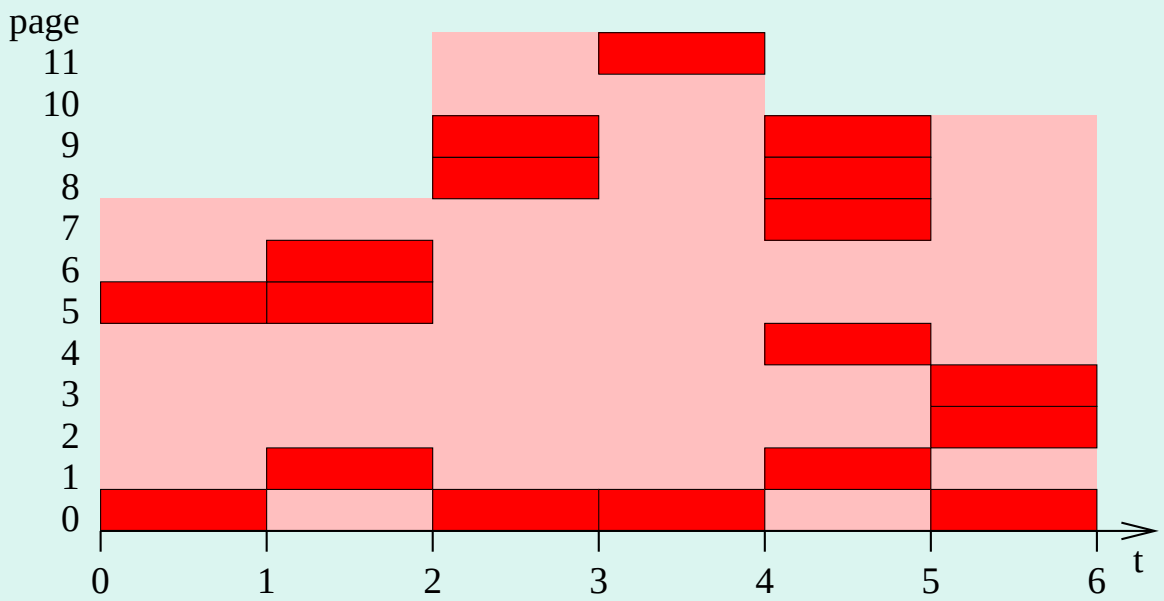
Mémoire Virtuelle (2)

- C'est le SE (avec l'aide du matériel) qui s'occupe de faire la **traduction de l'adresse logique à physique**
- Lorsque le SE donne le CPU à un processus, les **tables de traduction** sont configurées pour que la correspondance appropriée à ce processus soit en effet



Mémoire Virtuelle (3)

- Le “**working set**” c’est l’**ensemble des pages de mémoire** qu’un processus a accédé depuis un certain intervalle de temps Δ
- Le “**working set**” est normalement une petite fraction de l’espace total accessible au processus et varie en fonction des phases de calcul
- Exemple avec Δ = 1 seconde

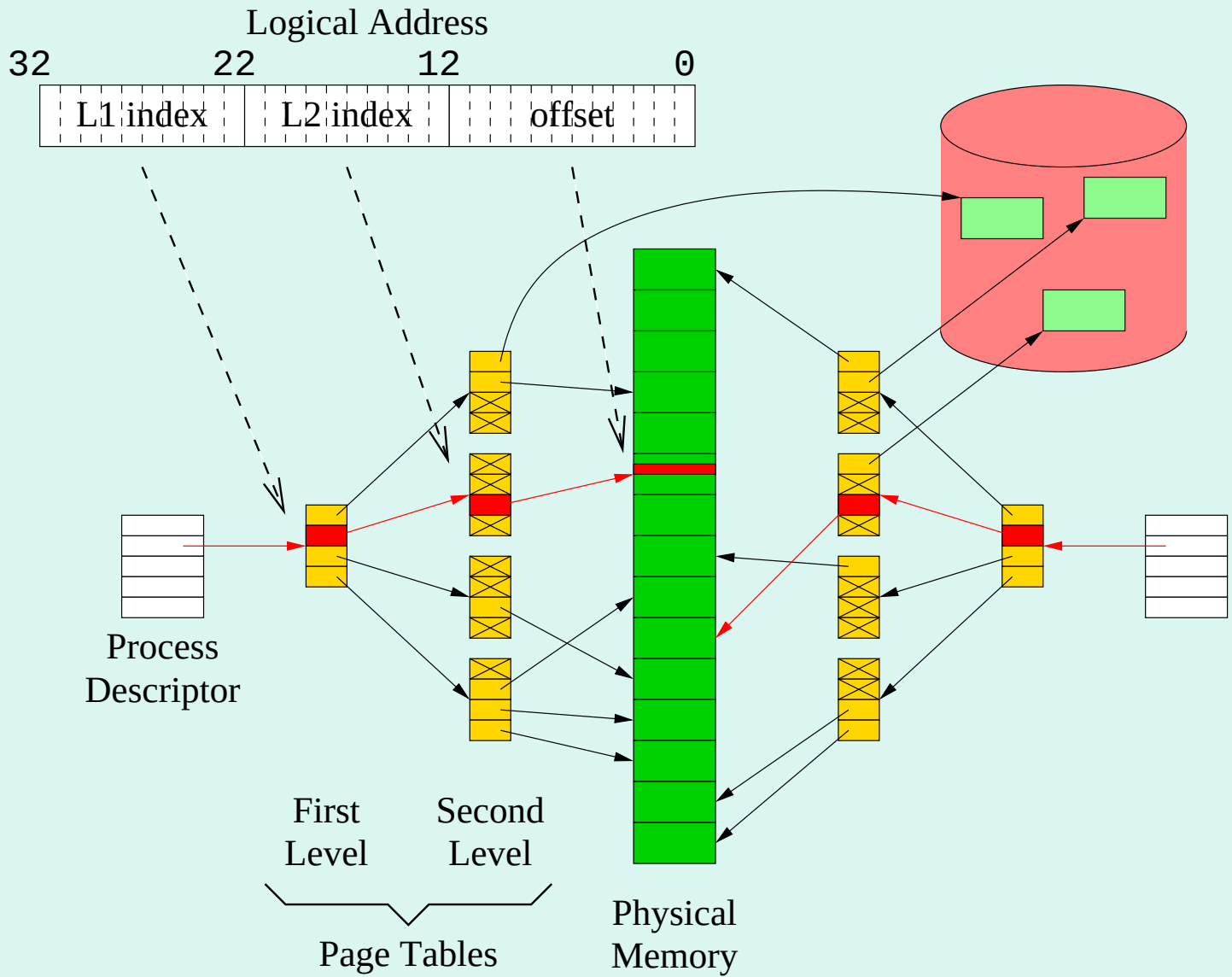


Mémoire Virtuelle (4)

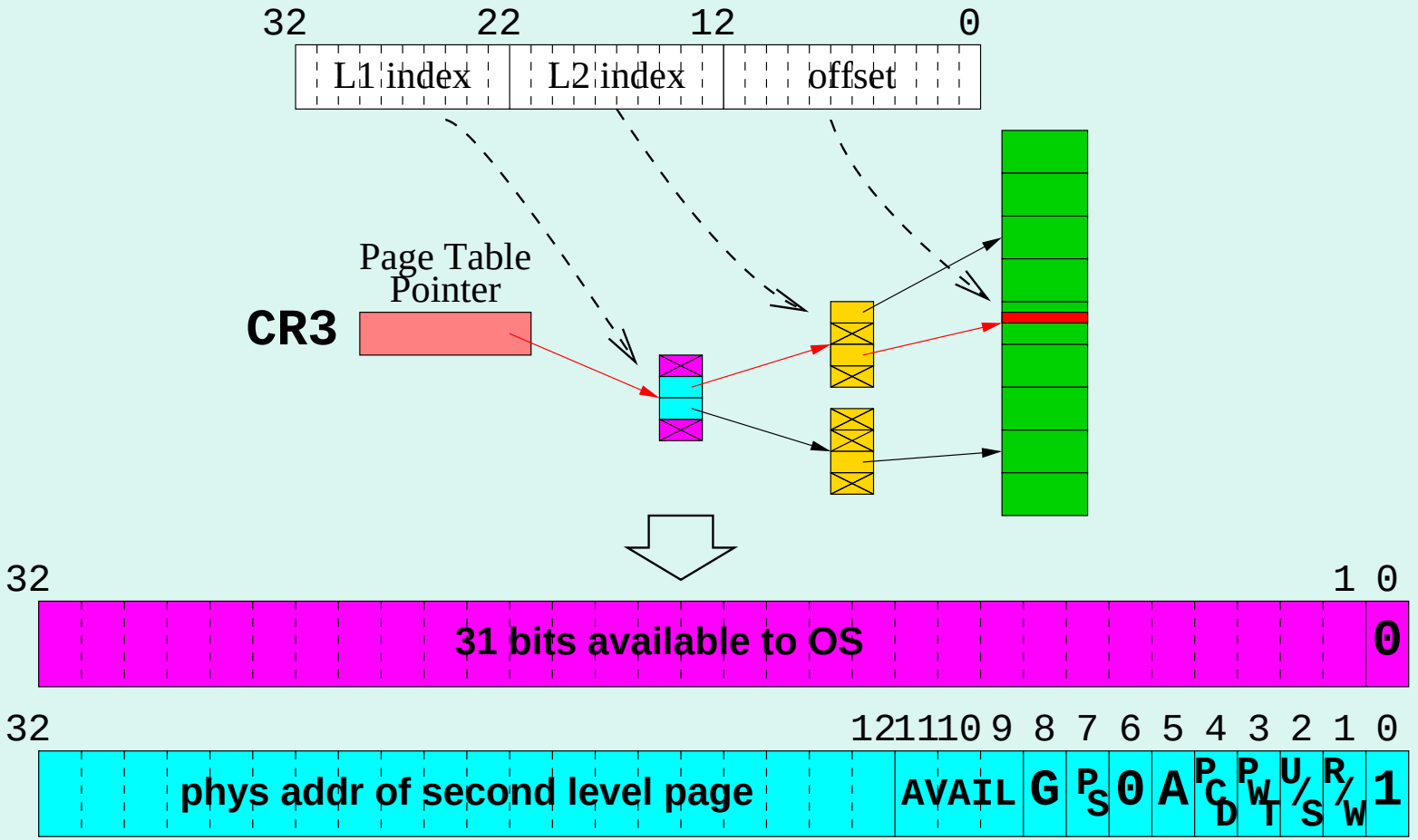
- Pour exécuter **des processus qui demandent plus de mémoire virtuelle que physique**, le SE utilise la mémoire principale comme une **antémémoire** pour les pages du “working set”; les autres pages sont en **mémoire secondaire** (disque de “swap”)
- Un processus sera approximativement aussi performant que si toutes les pages du processus étaient en mémoire principale (les seules pertes de temps sont dans les **transitions d’un working set à un autre**, car des pages doivent être récrites et lues du disque)
- La table de traduction indique donc pour chaque page d’adresse logique:
  - si la page est **invalide** (i.e. non accessible)
  - **l’endroit** où elle se trouve (disque/mémoire

Mémoire Virtuelle (5)

- Les tables de traduction sont souvent **hiérarchiques**
- Par exemple 2 niveaux sur le i386:

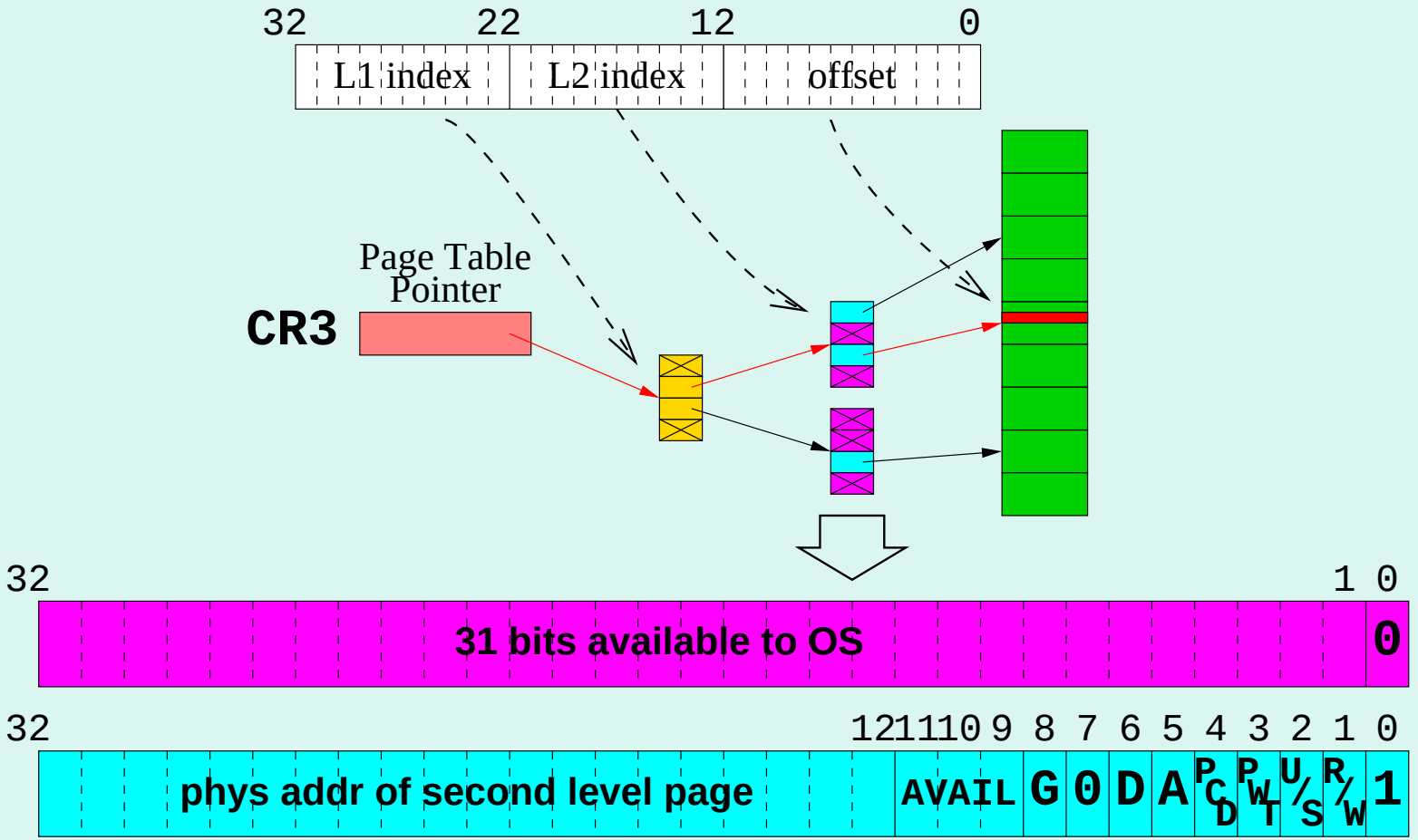


Pages de niveau 1 sur i386 (4 kbytes)



AVAIL (disponible pour le système d'exploitation)
G (page globale) ignoré
PS (taille de page) PS=0 pour 4kbytes, PS=1 pour 4Mbytes
A (accédé) mis à 1 à chaque accès à la page de niveau 2
PCD et PWT, configuration de l'antémémoire
U/S (permission user/supervisor)
R/W (permission read/write)

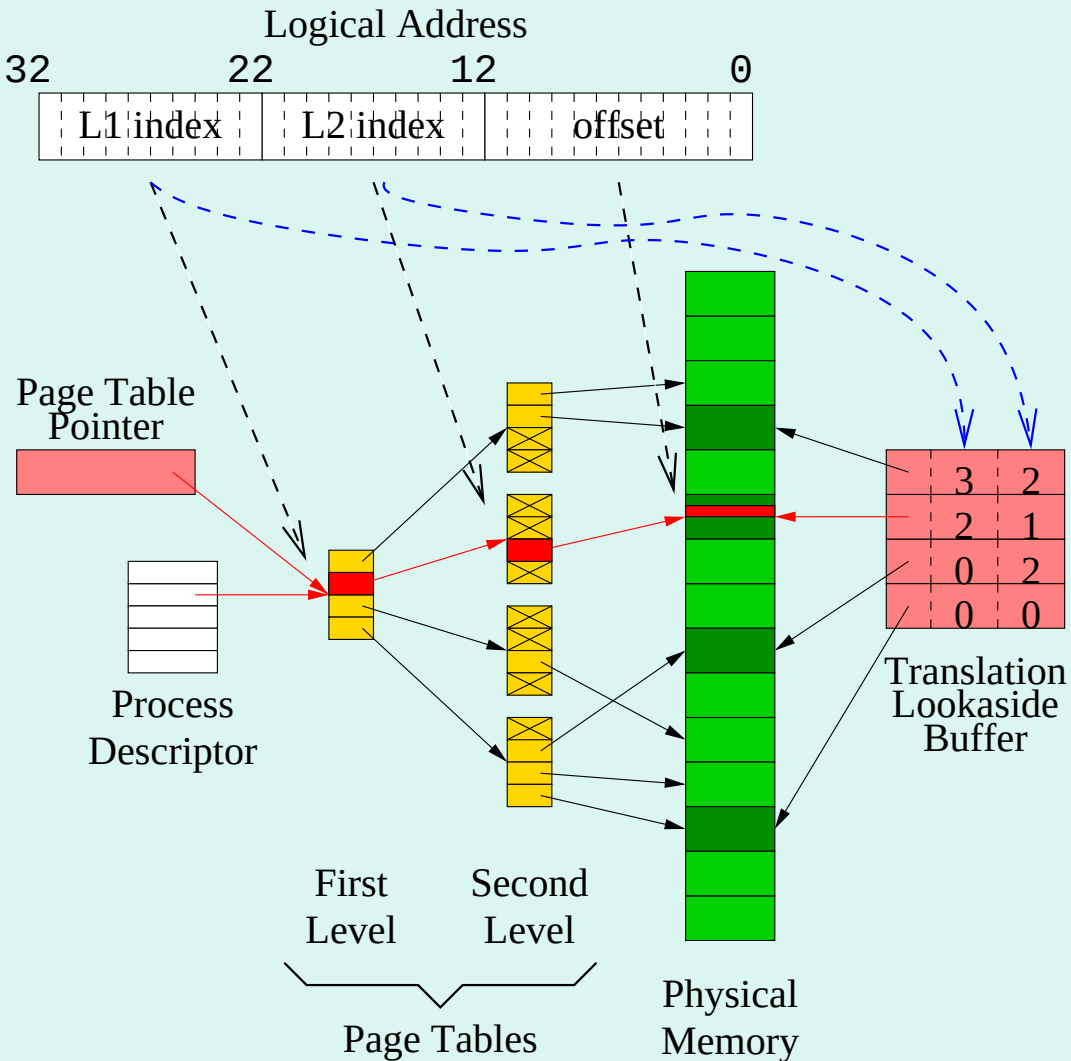
Pages de niveau 2 sur i386 (4 kbytes)



AVAIL (disponible pour le système d'exploitation)
G (page globale)
D (page sale) mis à 1 à chaque écriture à la page
A (accédé) mis à 1 à chaque accès à la page
PCD et PWT, configuration de l'antémémoire
U/S (permission user/supervisor)
R/W (permission read/write)

Translation Lookaside Buffer

- Pour ne pas avoir à consulter les tables de traduction (2 lectures) à chaque accès logique, une petite antémémoire est maintenue (**Translation Lookaside Buffer**)



Accès à la mémoire virtuelle (1)

- Pour traduire l'adresse logique L en adresse physique P le CPU procède comme suit:
  - si L est **dans la TLB**, l'utiliser pour calculer P
  - sinon, indexer 1er et 2eme niveaux des tables de pages
  - si la page est invalide, interruption "**segment violation**"
  - si la page est en mémoire physique, **mettre-à-jour la TLB** et calculer P
  - si la page est sur disque, interruption "**page fault**"

Accès à la mémoire virtuelle (2)

- Le traiteur de “**page fault**”:
  1. suspends le processus
  2. lit la page du disque
  3. met-à-jour les tables de traduction du processus
  4. résume le processus à l’instruction d’accès

- Si toutes les pages en mémoire physique sont en utilisation (ce qui est le cas normal), il faut avant tout **libérer une page physique en la retirant d’une table de traduction** (il faut l’écrire sur disque si elle a été modifiée depuis sa lecture, i.e. si elle est “sale”)

- Quelle page choisir?

Algorithmes de Remplacement (1)

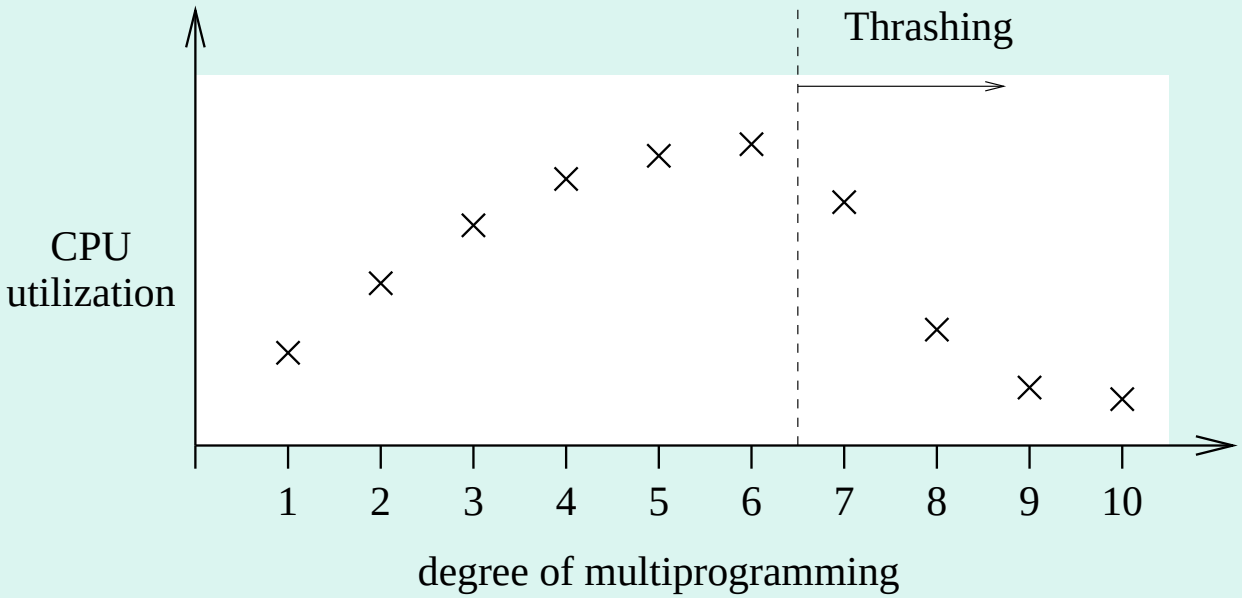
- Il y a plusieurs algorithmes
  - **FIFO**: cycle sur toutes les pages
  - **optimal**: choisit la page qui sera accédée le plus loin dans le futur
  - **LRU**: choisit la page qui a été accédée depuis le plus longtemps
- Exemple s'il y a 3 pages de mémoire principale:

Algorithmes de Remplacement (2)

- **Optimal** est difficile à implanter; surtout utile pour évaluer l'efficacité d'un autre algorithme
- Implantations de **LRU**:
  - stocker un "**time stamp**" à chaque accès; choisir la page avec le plus petit
  - conserver une pile des pages; déplacer la page accédée au dessus de la pile; choisir la plus profonde
  - approximation **1 bit d'accès**: 1 bit par page; mis-à-zéro à chaque X secondes; mis-à-un à chaque accès; choisir une page avec bit à zéro
  - approximation "**second chance**": sélection FIFO + 1 bit d'accès; si bit à zéro remplacer la page, sinon mettre bit à zéro, laisser la page en mémoire et essayer la prochaine page

Thrashing (1)

- Il y aura "**thrashing**" si la somme des "working sets" est **supérieure à la taille de la mémoire principale**
- Chaque processus tente de charger son working set en mémoire, ce qui déplace les pages des working sets des autres processus
- **Le CPU passe son temps en E/S et ne fait plus de travail utile**



Thrashing (2)

- Exemple : un programme qui a un working set de 40 MBytes

```
int main (int argc, char* argv[])
{
  char buf[10000];
  int i, j;

  double* t = malloc (5000000 * sizeof (double));

  int fd = open ("big", O_RDONLY);

  for (i=0; i<100; i++)
    for (j=0; j<5000000; j++)
      {
        t[j] = 0;
        if (j % 100000 == 0)
          read (fd, buf, 10000);
      }

  return 0;
}
```

Thrashing (3)

- Exécution sur une machine avec 128 MBytes de RAM

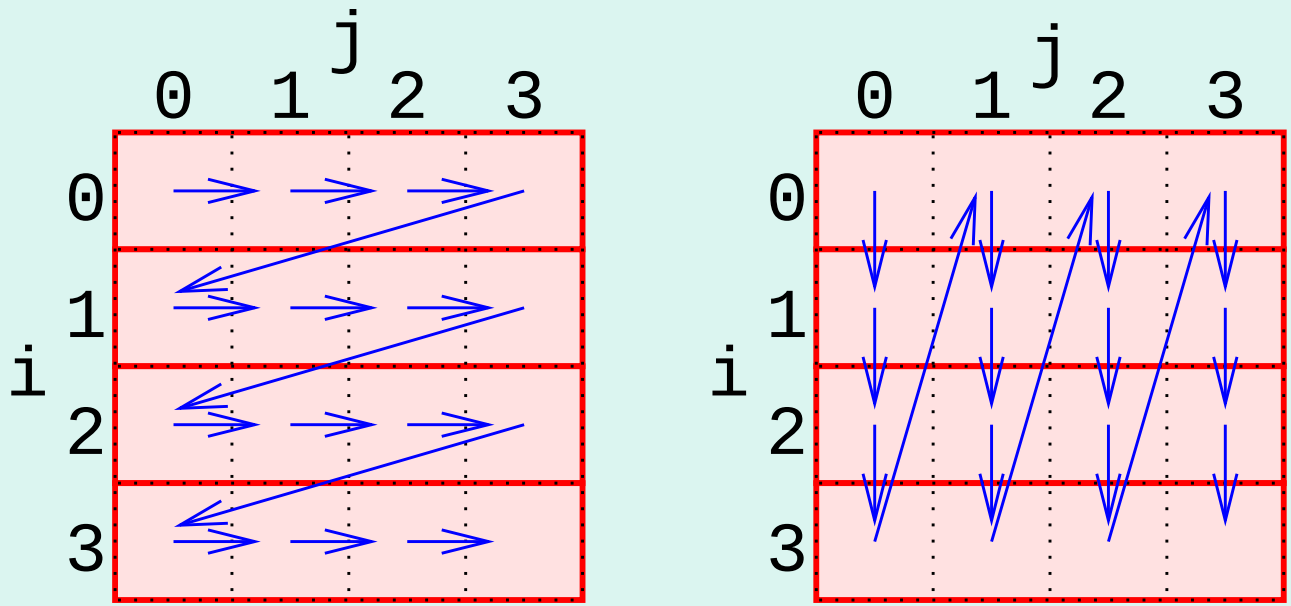
```
% time ./a.out
./a.out  19.24s user 0.97s system 97% cpu 20.658 total
% time ./a.out | ./a.out
./a.out  19.25s user 1.06s system 46% cpu 43.928 total
./a.out  19.42s user 0.95s system 46% cpu 43.863 total
% time ./a.out | ./a.out | ./a.out
./a.out  19.20s user 2.04s system 16% cpu 2:07.26 total
./a.out  19.35s user 2.05s system 16% cpu 2:07.25 total
./a.out  19.21s user 1.25s system 44% cpu 46.384 total
% time ./a.out | ./a.out | ./a.out | ./a.out
./a.out  19.30s user 108.93s system 0% cpu 7:00:29.98 total
./a.out  20.50s user 112.49s system 0% cpu 7:02:23.44 total
./a.out  18.14s user 109.59s system 0% cpu 6:58:55.18 total
./a.out  18.94s user 109.54s system 0% cpu 6:59:19.03 total
```

- Dans le dernier cas l'exécution est 300 fois plus lente que lorsque le working set tient en RAM!

Thrashing (4)

- Le **patron d'accès mémoire** du programme à un impact important sur la taille du working set:

```
int t[1024][1024];
int i, j;

for (i=0; i<1024; i++)
  for (j=0; j<1024; j++)
    t[i][j] = 0; // WS = 1 page per inner loop, time = 0.04s

for (j=0; j<1024; j++)
  for (i=0; i<1024; i++)
    t[i][j] = 0; // WS = 1024 pages per inner loop, time = 0.28s
```



Thrashing (5)

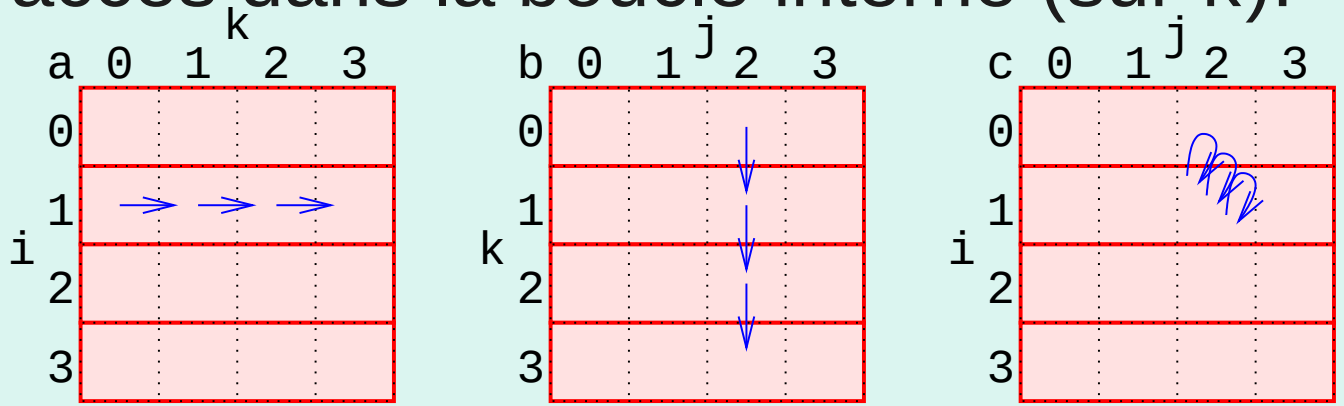
- Exemple : multiplication de matrice

```
#define N 512

typedef double mat[N][N];

void matmul1 (mat a, mat b, mat result)
{ int i, j, k;

  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      result[i][j] = 0;

  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      for (k=0; k<N; k++)
        result[i][j] += a[i][k] * b[k][j];
}

mat a, b, c;

int main (int argc, char* argv[])
{ matmul1 (a, b, c);
  return 0;
}
```

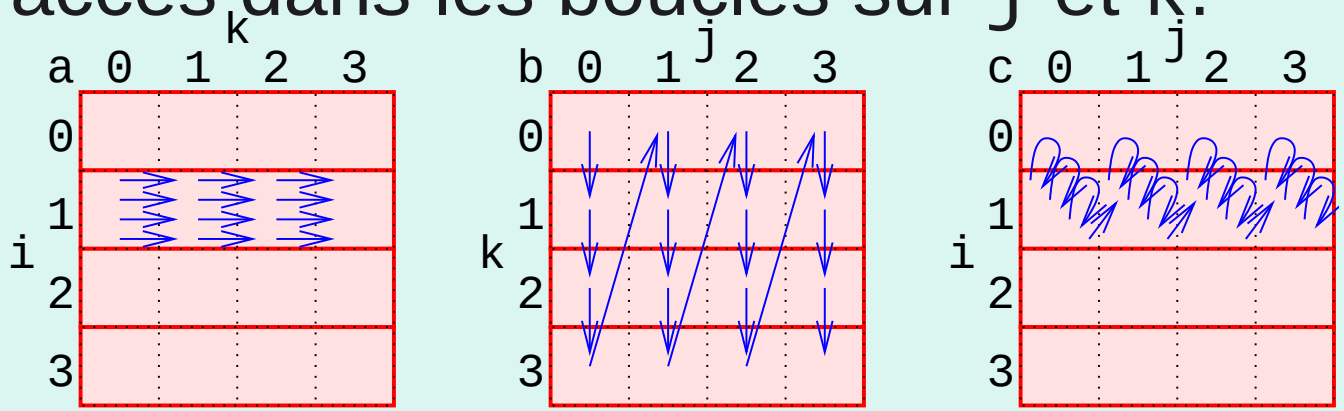
Thrashing (6)


```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      result[i][j] += a[i][k] * b[k][j];
```

- Patron d'accès dans la boucle interne (sur k):



- Patron d'accès dans les boucles sur `j` et `k`:

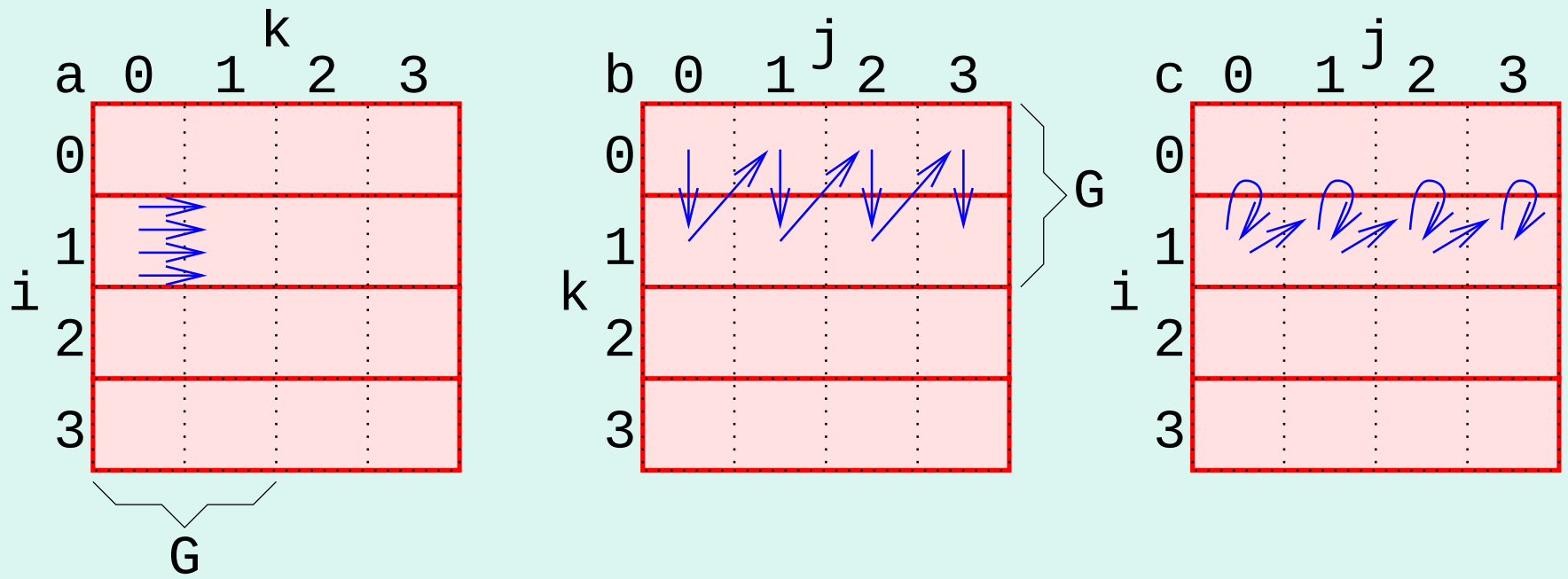


Thrashing (7)

- `matmul2`: découper la boucle sur `k` en blocs

```
for (kk=0; kk<N; kk+=G)
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      for (k=kk; k<kk+G; k++)
        result[i][j] += a[i][k] * b[k][j];
```

- Patron d'accès dans les boucles sur `j` et `k`:



Thrashing (8)

- Exécution sur une machine avec des pages de 4096 octets et une TLB de 64 entrées et G=32

```
% time ./matmul1
./matmul1  14.93s user 0.02s system 99% cpu 15.026 total
% time ./matmul2
./matmul2   4.00s user 0.03s system 98% cpu  4.110 total
```

- Le deuxième algorithme est plus de 3 fois plus rapide que l’algorithme classique!