Les API classiques (IPC et threads)

Pierre David
pda@unistra.fr

Université de Strasbourg – Licence d'informatique

2017 – 2018

Plan

Plan

Introduction

Problème de la concurrence = partage de mémoire
$\Rightarrow$ comment partager la mémoire dans un programme ?

Introduction aux API [1] Unix et POSIX

- pour le partage de mémoire entre processus
- pour les threads

1. Application Programming Interface

Plan

Rappels – Processus

Qu'est-ce qu'un processus ?

Définition 1 (haut niveau) :

> *Un processus est une instance d'un programme en cours d'exécution*

Exemples :

- je tape « `ls /tmp` » : le programme `ls` est exécuté avec la donnée `/tmp` (et donc implicitement le contenu de `/tmp` car on sait ce que fait `ls`)
- je relance la même commande : ce n'est pas le même processus (c'est une autre exécution avec le même jeu de données)

Rappels – Processus

De plus, un processus possède des attributs :

- état
- identité (PID)
- propriétaire (UID)
- répertoire courant (CWD)
- terminal de contrôle
- ouvertures de fichiers
- actions définies pour les signaux
- etc.

Rappels – Processus

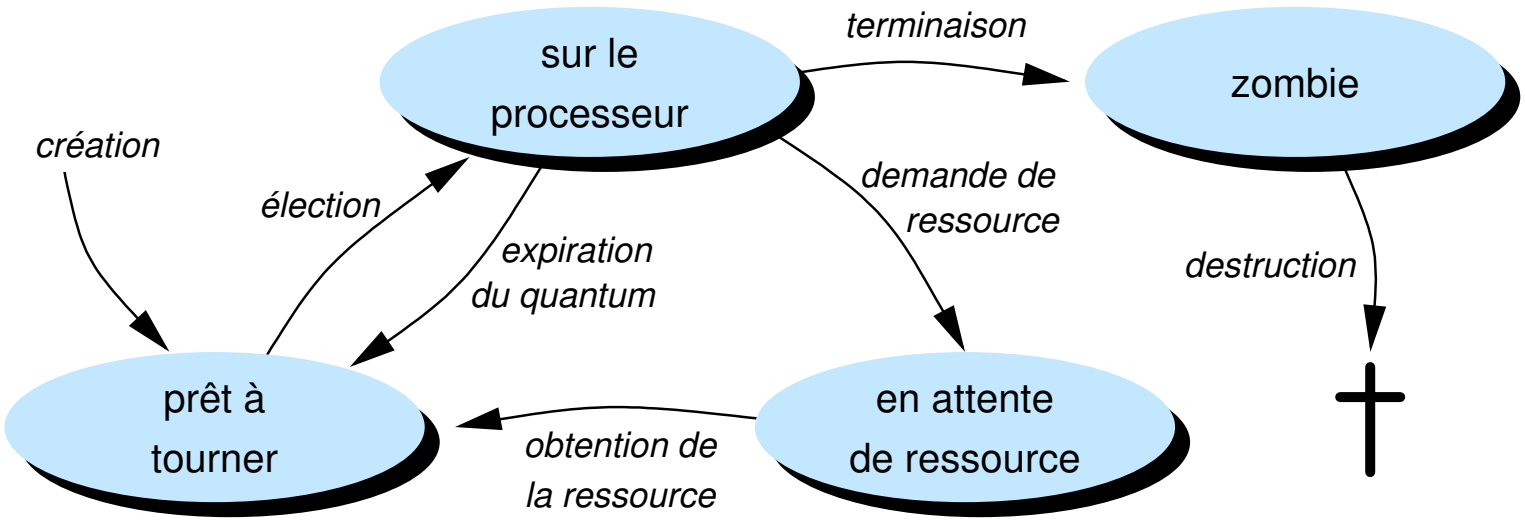
Primitives POSIX de gestion des processus :

- `pid_t fork (void)`
  Création d'un nouveau processus par duplication du processus courant
- `void exit (int val)`
  Terminaison du processus courant
- `pid_t wait (int *pval)` (et autres `wait*`)
  Attente de la terminaison d'un des processus fils

Les primitives `exec*` ne servent pas plus à la gestion des processus que `chdir`, `kill` ou `sigaction`.

Rappels

Exemple de diagramme d'états de processus



Ressources : E/S, date, événement d'un autre processus, etc.

Rappels – Processus

Qu'est-ce qu'un processus ?

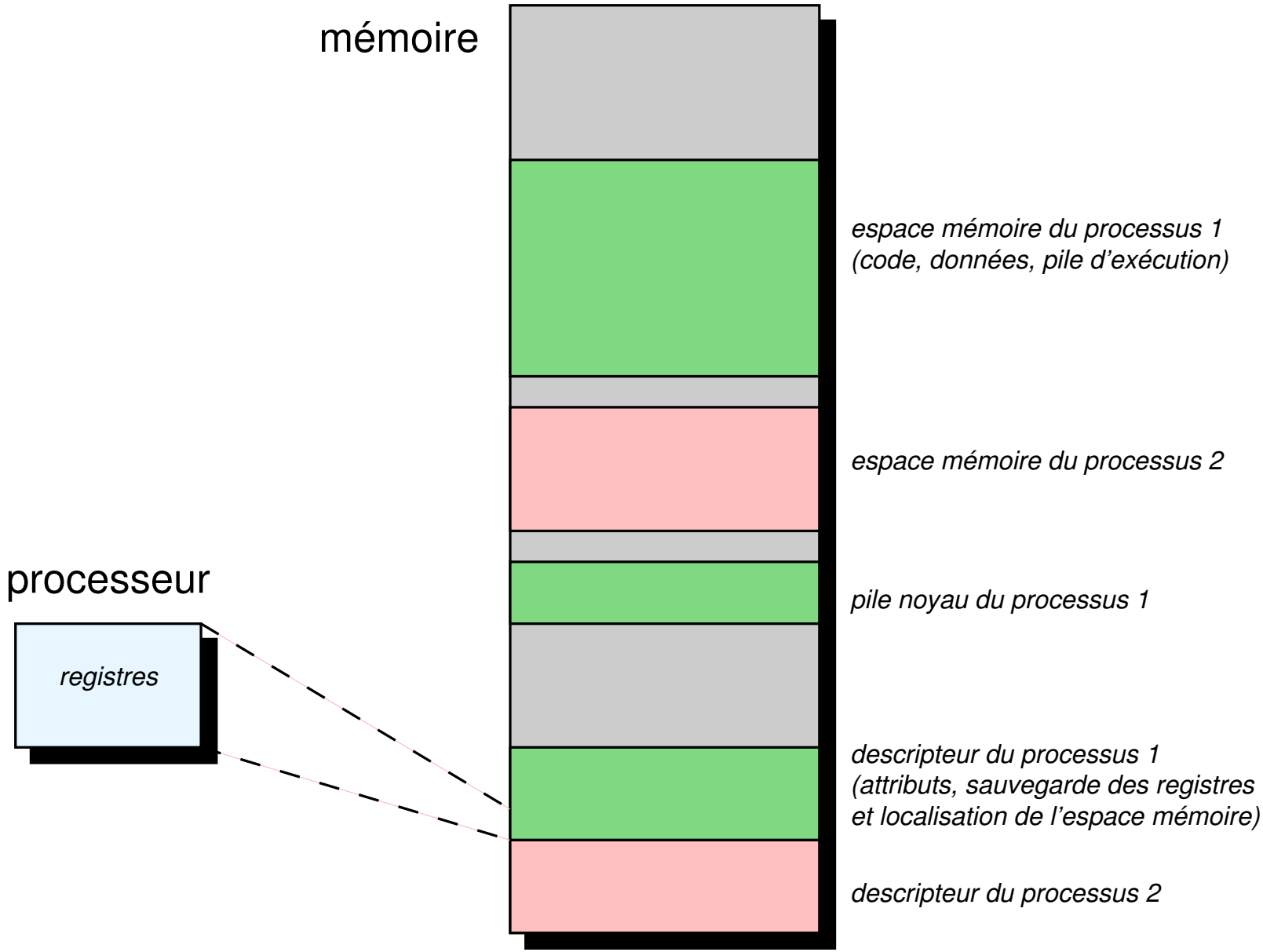
Définition 2 (bas niveau) :

> *Un processus est matérialisé par ses attributs, un contexte CPU et un espace mémoire*

Contexte CPU :

- registres du processeur
  - registres généraux (ex : EAX, EBX, etc. sur x86)
  - registre PC (*Program Counter*, EIP sur x86)
  - registre SP (*Stack Pointer*, ESP sur x86)
  - registre SR (*Status Register*, EFLAGS sur x86)
  - registres spécialisés (ex : registres MMX sur x86)
- ce contexte est sauvegardé en mémoire lorsque le processus n'est pas sur le processeur

Rappels – Processus



Rappels – Interruptions et exceptions

Entrée dans le noyau :

- interruption (ex : fin d'E/S, etc.)
  $\Rightarrow$ événement provoqué par un mécanisme externe (contrôleur d'E/S, horloge, etc.)
- exception (ex : violation d'adresse, division par 0, etc.)
  $\Rightarrow$ événement provoqué (volontairement ou non) par le programme
- appel système $\Rightarrow$ exception provoquée par une instruction spécifique (ex : INT sur x86, TRAP sur 68000)

Les traitements sont très proches

Rappels – Interruptions et exceptions

Actions du processeur suite à une exception :

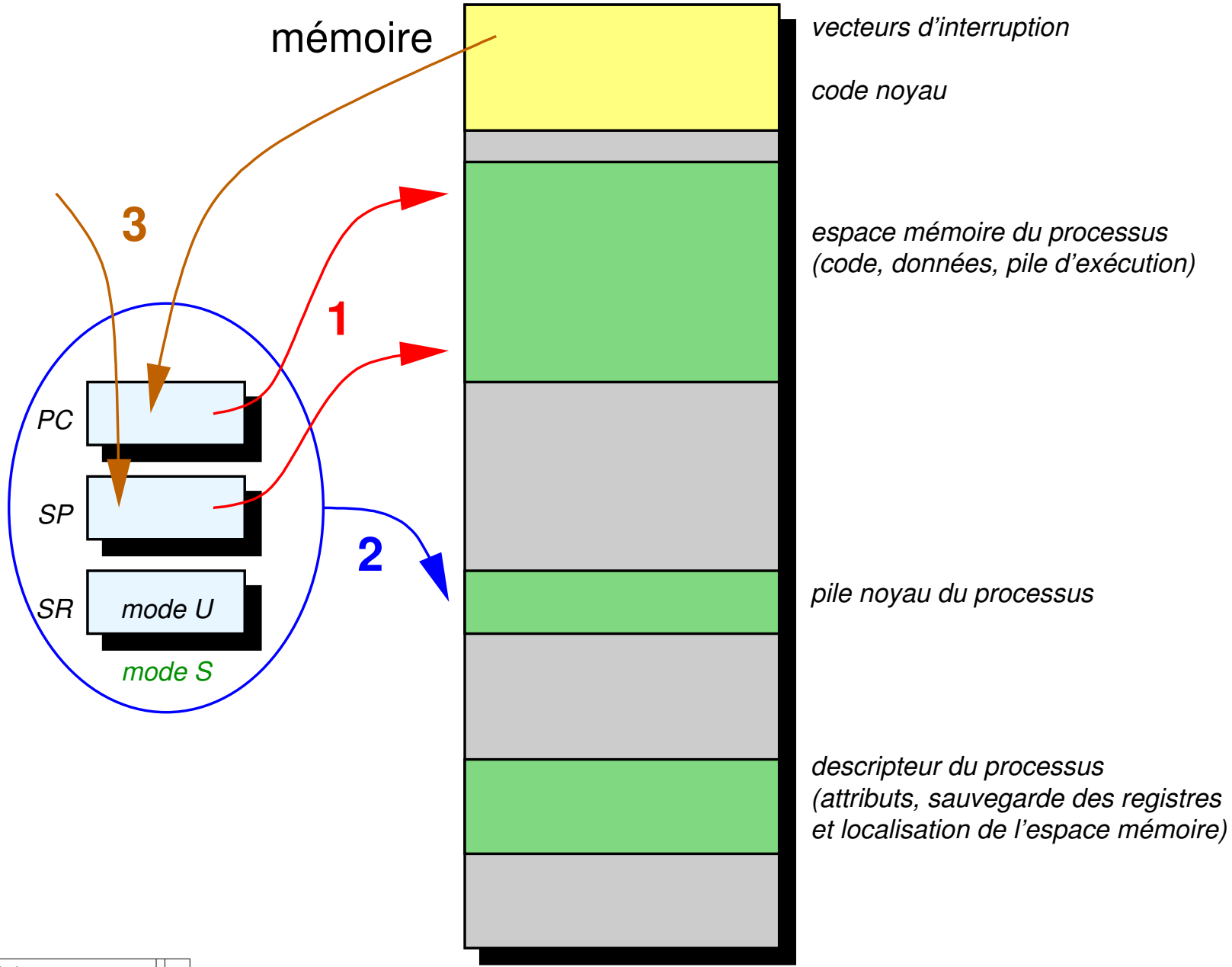
1. lorsque l'exception se produit, PC pointe dans le code du processus, SP dans la pile du processus et SR indique qu'on est en mode « utilisateur » (par exemple)

   En cas d'interruption, le processeur bloque (masque) en plus les interruptions en modifiant le masque d'interruptions dans SR
2. le processeur sauvegarde ces registres dans la pile d'exécution noyau associée au processus

   Note : la pile d'exécution du noyau est retrouvée par un mécanisme dépendant du processeur (registre spécifique pour Motorola 68000, indirection via lecture du TSS pour x86)
3. le processeur initialise PC à partir du vecteur d'interruption, SP à partir de l'adresse de la pile noyau, et SR pour indiquer le mode « système »

$\Rightarrow$ tout ceci est effectué par le matériel

Rappels – Interruptions et exceptions



Rappels – Interruptions et exceptions

Une fois le contexte (PC, SP, SR) initialisé, le processeur exécute le code du système d'exploitation :

1. (en assembleur) sauvegarde du reste du contexte CPU (registres généraux, etc.)
2. (en assembleur) mise en place d'un contexte de pile pour un appel de procédure en langage de haut niveau (ex : C)
3. (en assembleur) branchement à une adresse
4. (en C) vérification de la raison de l'exception
5. (en C) action correspondant à l'exception

Rappels – Interruptions et exceptions

Au retour :

- actions logicielles symétriques en fin d'exception (en C puis en assembleur)
- actions (en matériel) symétriques à la prise en compte de l'exception : instruction spéciale (IRET pour x86, RTE pour 68000)

Rappels – Commutation de processus

La commutation de processus survient :

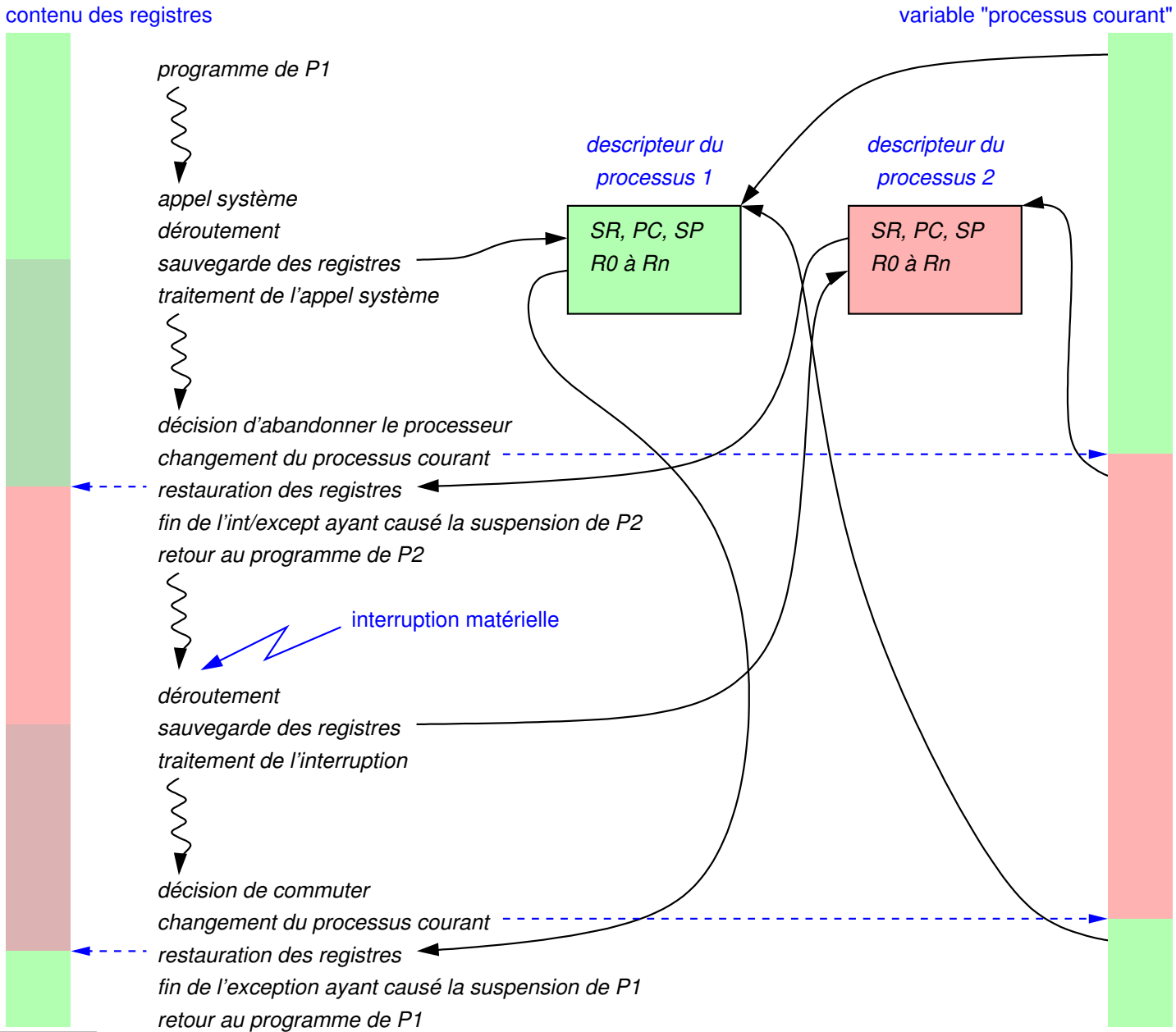
- suite à l'interruption de l'horloge entraînant le recalcul des priorités des processus
- suite à l'abandon du processeur par un processus via un appel de primitive système (i.e. une exception)
  $\Rightarrow$ demande d'E/S, attente d'un signal, etc.

Rappels – Commutation de processus

Techniquement, la commutation de processus ($P_1 \rightarrow P_2$) est :

- le changement de la variable du noyau indiquant le processus courant
  (variable multiple pour les multi-processeurs)
- la restauration des registres du processus remis sur le processeur
  $\Rightarrow$ fin du traitement de l'interruption ayant entraîné la suspension de P_2

Rappels – Commutation de processus

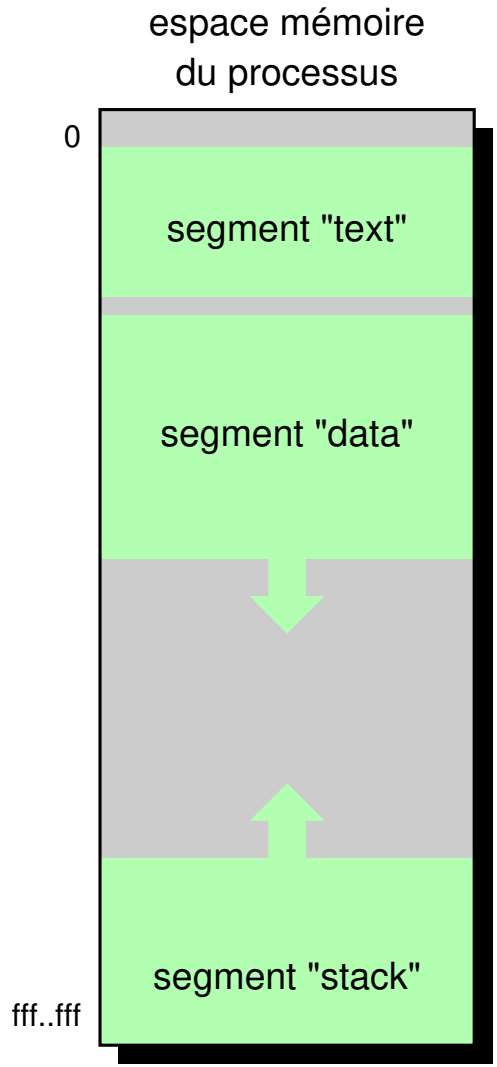


Rappels – Adressage logique

Espace d'adressage logique d'un processus

Chaque processus dispose :

- du segment « *text* » (non modifiable)
- du segment « *data* » (extensible via `malloc`)
- du segment « *stack* » (extensible automatiquement)
- et éventuellement d'autres segments (bibliothèques dynamiques...)

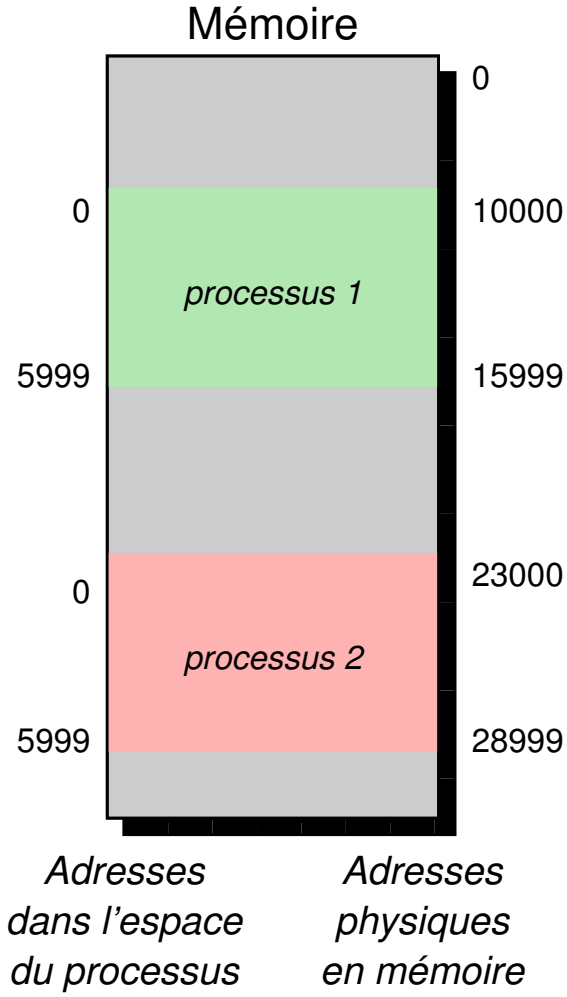


Rappels – Adressage physique

Chaque processus a son propre espace d'adressage :

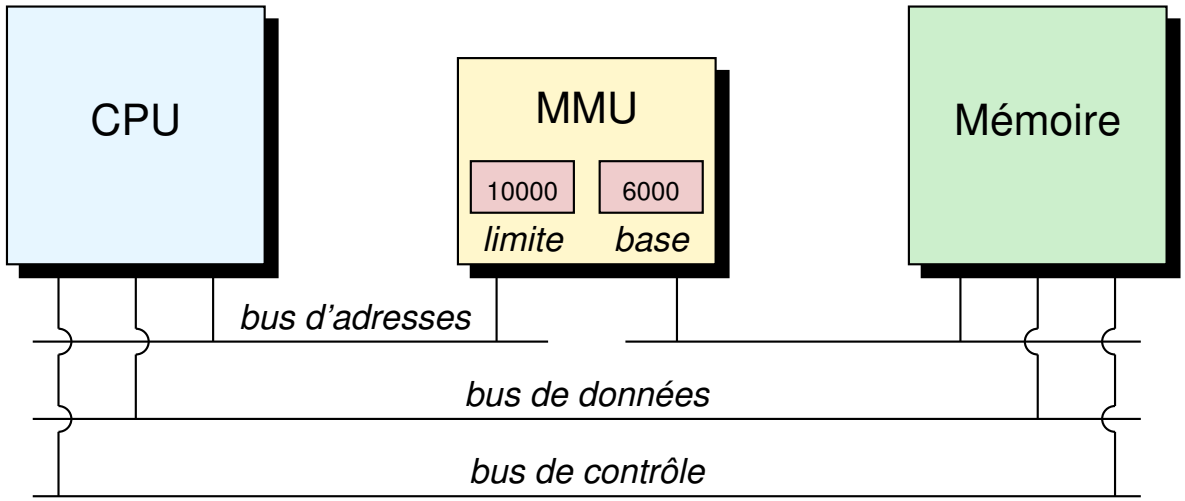
Les programmes utilisent des adresses *relatives* à l'espace d'adressage du processus.

Exemple : un programme utilise la case mémoire d'adresse 1000. Si deux processus exécutent ce programme, chacun doit avoir sa propre case d'adresse 1000. Ces deux cases ont une adresse réelle différente en mémoire.



Rappels – Traduction d'adresses

Unité de gestion mémoire (MMU) :



- CPU présente une adresse logique sur le bus d'adresses
- si adresse $\geq$ limite, alors exception
- si adresse $<$ limite, alors MMU calcule l'adresse physique (adresse physique = adresse logique + limite)

Rappels – Traduction d'adresses

Cette MMU est irréaliste : trop simple pour les cas réels

- pas de prise en compte de l'adresse 0
- les processus ont des « trous » dans leur espace d'adressage
  (ex : espace entre segment « data » et « stack »)
- les segments d'un processus ne sont pas forcément contigüs en mémoire physique
- pas de possibilité de partager un segment
  (ex : partage du segment « text » entre processus)

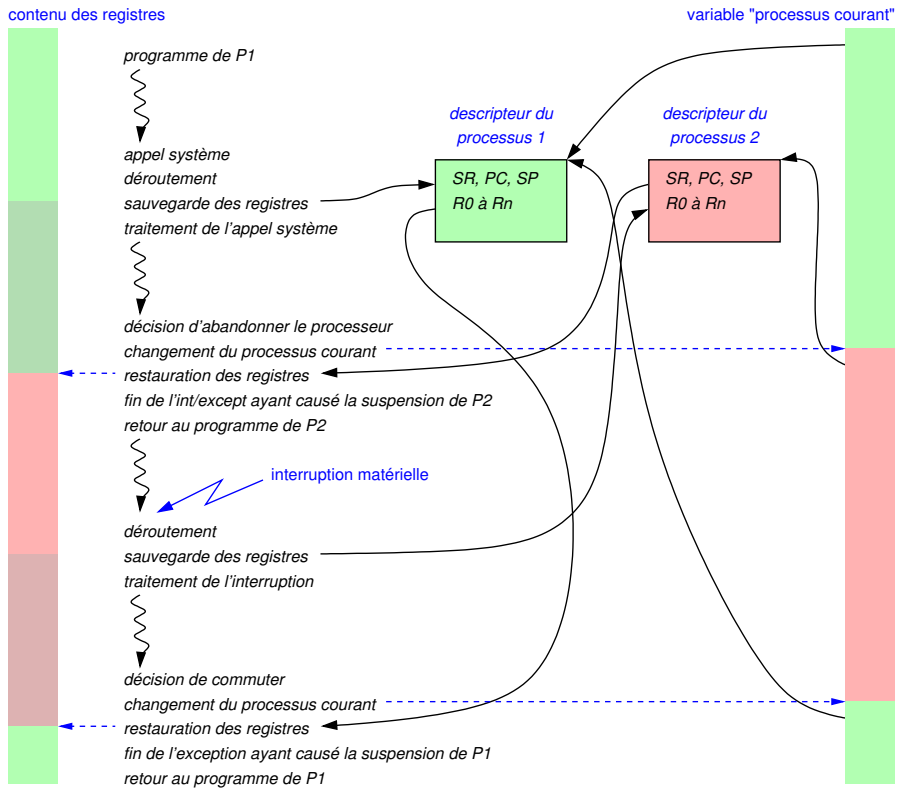
$\Rightarrow$ mais elle est pédagogique !

Rappels – Traduction d'adresses

Retour sur la commutation de processus

Avec notre MMU simpliste, il suffit :

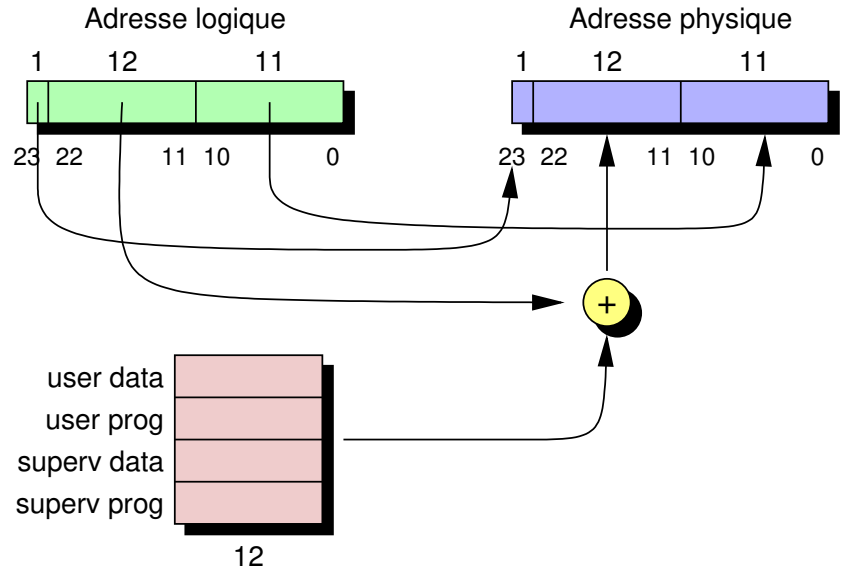
- de considérer qu'il n'y a pas de traduction d'adresse lorsque le CPU est en mode « système »
- d'intégrer les deux registres « limite » et « base » lors de la sauvegarde et de la restauration des registres
- vider le cache



(Même dessin que précédemment)

Rappels – Traduction d'adresses

Unité de gestion mémoire du HP 9000-IPC :

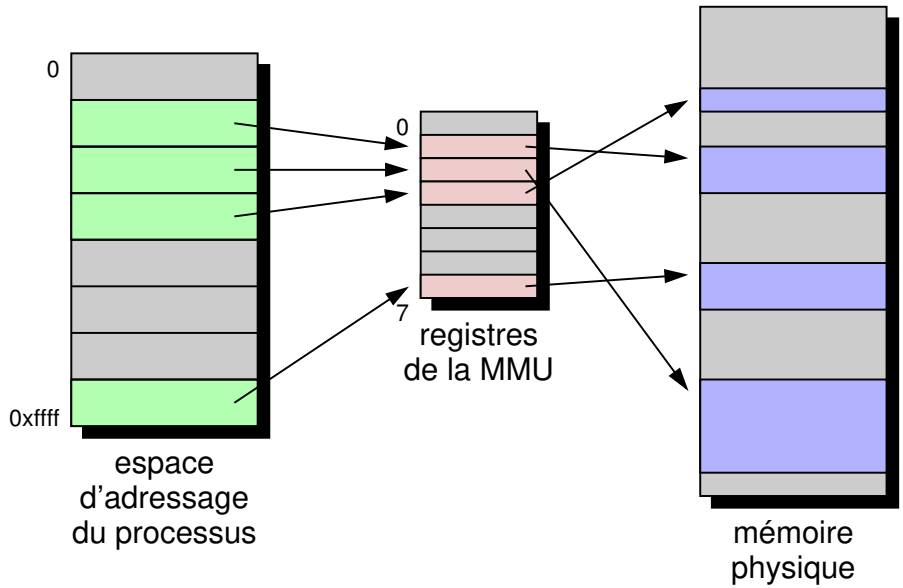


bit 23 utilisé pour activer la MMU (sinon accès en ROM)

- MMU « câblée »
- type d'accès mémoire (U/S, D/P) $\Rightarrow$ registre consulté
- additionneur seul $\Rightarrow$ pas de vérification de limite (machine mono-utilisateur)

Rappels – Traduction d'adresses

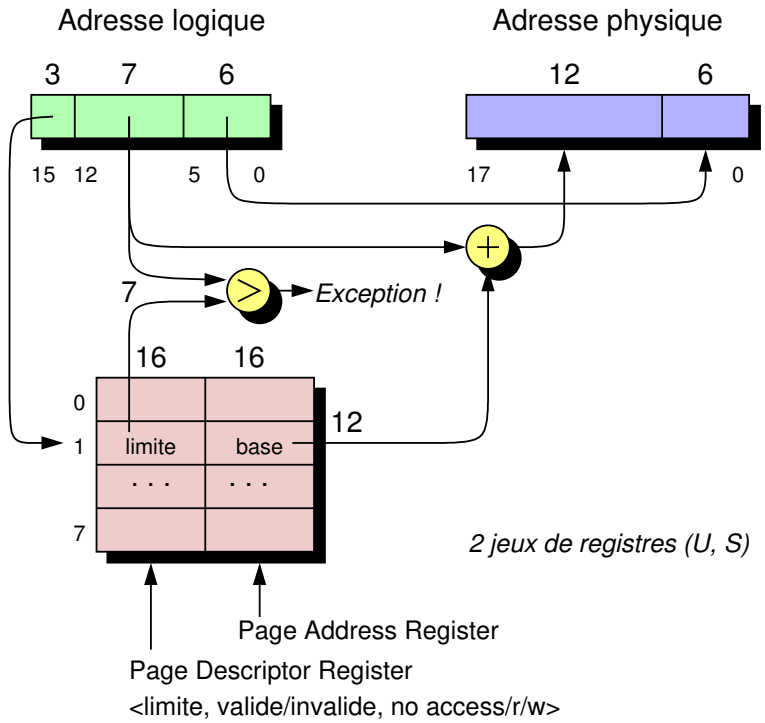
Principe de l'unité de gestion mémoire du PDP 11/45 :



- espace d'adressage d'un processus découpé en 8 segments
- mémoire physique découpée en blocs de 2^6 octets
- chaque segment est traduit par un registre de la MMU

Rappels – Traduction d'adresses

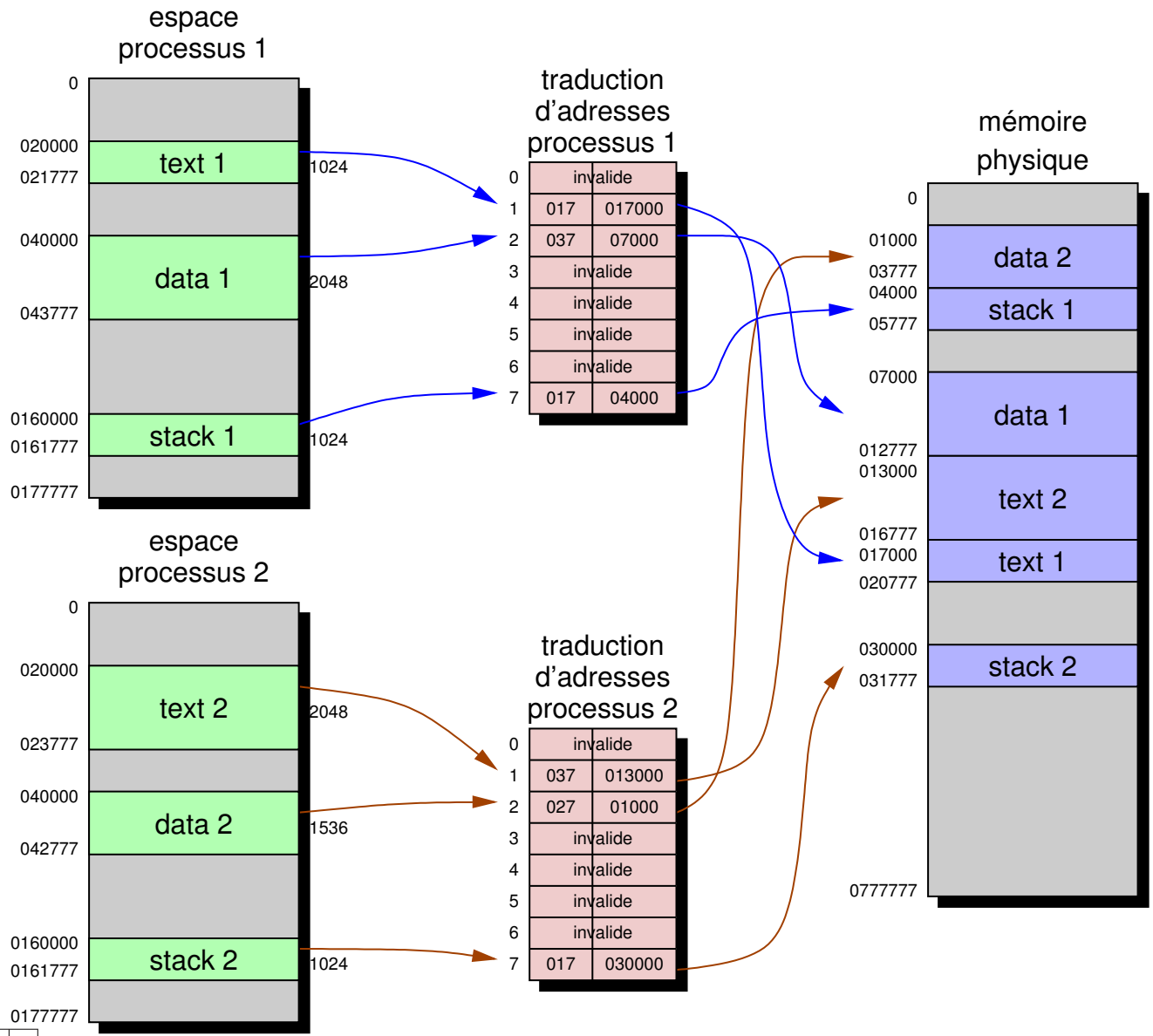
Traduction d'adresse sur le PDP 11/45 :



- mémoire physique découpée en blocs de 2^6 octets
- limite d'un segment : sur 7 bits
- jusqu'à 8 segments entre 1 et 2^7 blocs (64 à 8192 octets)

Rappels – Traduction d'adresses

Exemple avec 2 processus (rappel : `0123` = 123 en octal)



Rappels – Mémoire virtuelle

Jusqu'ici, traduction mémoire « simple »

- mémoire d'un processus : divisée en p blocs de 2^k octets
  Sur PDP 11/45 : $p = 2^3 = 8$, $k = 13 \Rightarrow 2^k = 8192$ octets
- adapté pour les 3 segments d'un processus
  et peut-être d'autres segments supplémentaires
  (bibliothèques dynamiques, mémoire partagée, etc.)
- pour que le processus soit mis sur le processeur, il faut que tout son espace d'adressage soit en mémoire

Rappels – Mémoire virtuelle

Besoins nouveaux :

- exécuter des programmes de taille supérieure à la mémoire physique
- rationnaliser l'utilisation de la mémoire
  un processus n'a pas besoin de tout son espace d'adressage pendant toute sa durée de vie
  Exemple : code d'initialisation parcouru une seule fois

$\Rightarrow$ Mémoire virtuelle (pagination)

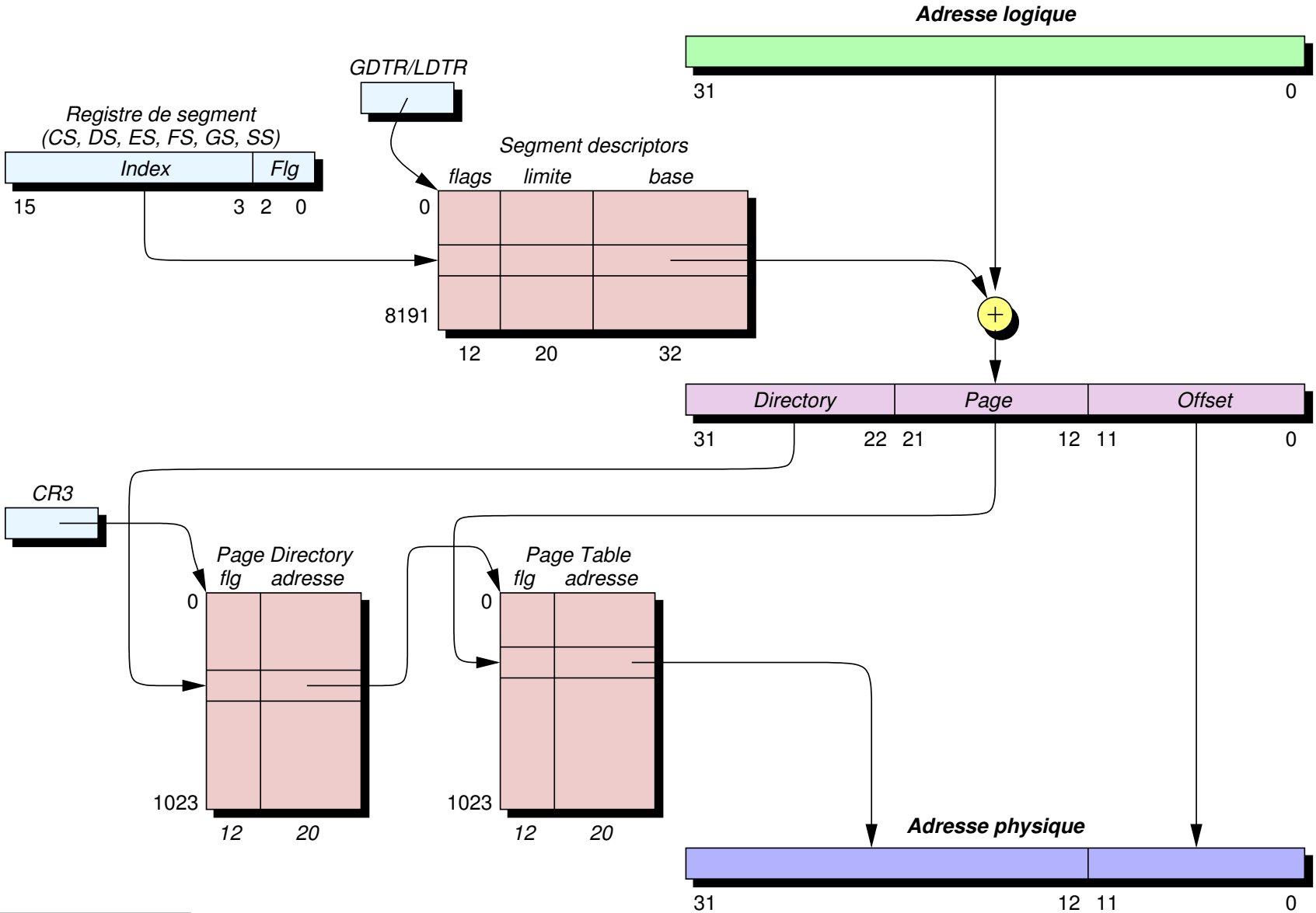
Rappels – Mémoire virtuelle

Principes :

- l'espace mémoire est découpé en pages de taille 2^n
  Ex : $n = 12 \Rightarrow 2^n = 4096$ octets (cas le plus courant)
  Ex : $n = 22 \Rightarrow 2^n = 4$ Mo (*superpages* sur x86)
- chaque page peut être chargée en mémoire ou non
- la MMU gère une « table des pages » par processus
- « défaut de page » : la MMU génère une exception
  $\Rightarrow$ le noyau traite l'exception
- lorsque le noyau aura chargé la page en mémoire et positionné la table des pages, le processeur pourra à nouveau tenter l'exécution de l'instruction « fautive »
  $\Rightarrow$ les instructions sont « redémarrables »

Rappels – Mémoire virtuelle

Unité de gestion mémoire du i386 :



Rappels – Mémoire virtuelle

Exemple du i386 :

- segmentation : poids du passé, on oublie...
- adresse après segmentation sur 32 bits
- la table « page directory » est :
  - référencée par le registre CR3 du processeur
  - indexée par les 12 bits de poids fort de l'adresse
- chaque table des pages est :
  - référencée par l'entrée dans « page directory »
  - indexée par les 12 bits de poids milieu de l'adresse
- l'adresse physique est constituée :
  - de l'adresse de la page trouvée dans la table des page
  - de l'offset dans la page récupéré de l'adresse originale

Rappels – Mémoire virtuelle

Les tables des pages résident en mémoire :

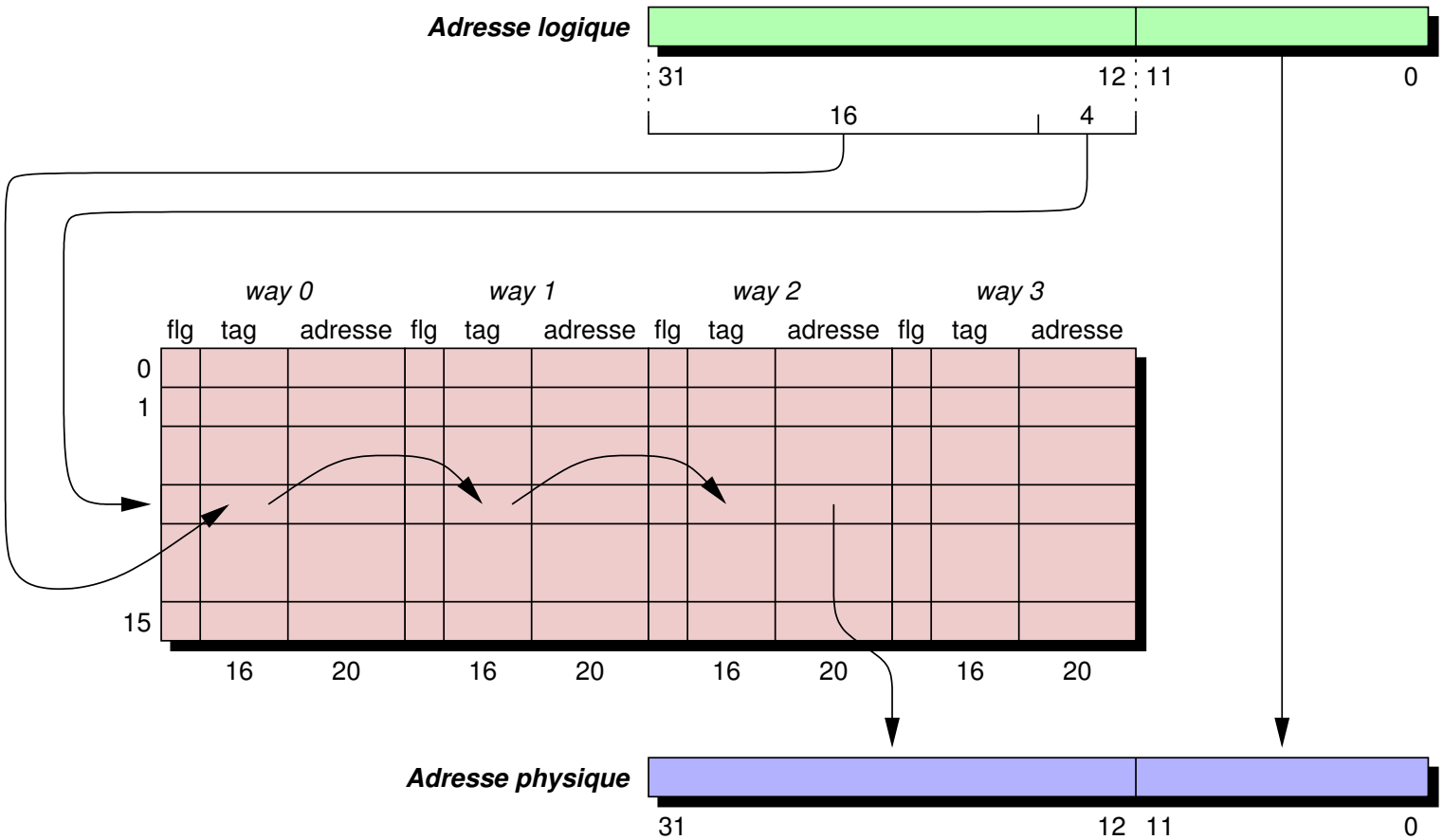
- commutation de processus $\Rightarrow$ changer l'adresse de la table des pages
  (registre CR3 sur x86)
- les tables de pages peuvent être swappées sur le disque
- la traduction d'une adresse peut elle-même conduire à un défaut de page
- traduction lente, même sans défaut de page

Optimisation des traductions :

- TLB (*Translation Lookaside Buffer*)
  $\Rightarrow$ cache des dernières traductions d'adresse

Rappels – Mémoire virtuelle

Exemple pour Pentium III (x86) : TLB « *4-way set-associative* »



Ne pas oublier de vider le TLB lors de la commutation de processus...

Plan

Partage de mémoire en processus

Objectif de la gestion mémoire dans un système : assurer l'étanchéïté des espaces mémoires entre processus
($\Rightarrow$ objectif de sécurité du système)

Besoin de performances en multi-processeurs
$\Rightarrow$ besoin de partager des portions de mémoire
$\Rightarrow$ développement d'interfaces de programmation idoines

Partage de mémoire en processus

API disponibles :

- IPC System V (`shmget`/`shmat`/etc)
  $\Rightarrow$ origine AT&T, vieille interface, orthogonale avec le reste du système, très répandue
- `mmap`/`munmap`
  $\Rightarrow$ origine Berkeley, partage de mémoire = effet de bord de la projection d'un fichier en mémoire, simple, peu répandue
- `shm_open`/`shm_unlink`
  $\Rightarrow$ origine POSIX, simple et élégante, encore peu répandue

Dans ce cours : IPC System V

IPC System V

Les IPC System V fournissent :

- des files de messages
  $\Rightarrow$ dans ce cours : exemple d'utilisation des IPC System V
- des segments de mémoire partagée
  $\Rightarrow$ l'objet principal de ce cours
- des groupes de sémaphores
  $\Rightarrow$ cours ultérieur

IPC System V

Les « objets » IPC System V :

- ont la durée de vie du système
  Ils sont conservés dans la mémoire du noyau
  $\Rightarrow$ effacés lorsque le noyau s'arrête ou lors d'une suppression explicite
- sont désignés par une *clef* (type `key_t`)
  Sorte de nom = convention entre tous les programmes participant à la communication
- sont identifiés par un *identificateur* (type `int`)
  Un identificateur identifie un objet *existant*
- sont créés (ou accédés s'ils existent) via les primitives :
  - `int id = msgget (clef, ...)`
  - `int id = shmget (clef, ...)`
  - `int id = semget (clef, ...)`

IPC System V

Analogie avec les fichiers :

	IPC System V	**Fichier**
Nom	clef (`key_t`)	nom de fichier (`char *`)
Durée de vie	arrêt du système ou suppression explicite	suppression du fichier
Accès/création	primitives `xxxget()`	primitive `open()`
Descripteur	id (`int`)	descripteur de fichier (`int`)

IPC System V

Comment choisir une clef ?
$\Rightarrow$ convention entre les participants à la communication

- choisir un nombre au hasard ?
  $\Rightarrow$ possibilité de collisions
- faire appel à un annuaire de clefs ?
  $\Rightarrow$ bonne idée, mais ça n'existe pas...

Solution (*hack*) : fonction de bibliothèque `ftok`

IPC System V

```
key_t ftok (char *fichier, int projet)
```

Élaboration de la clef, par exemple à partir de :

16 bits	numéro d'inode du fichier
8 bits	numéro de périphérique (mineur) du fichier
8 bits	`projet`

- avec de la chance, cette clef est unique (personne ne l'utilise déjà)
- si tout le monde utilise `ftok` avec des valeurs différentes, il y a des chances que cette clef soit et reste unique

IPC System V

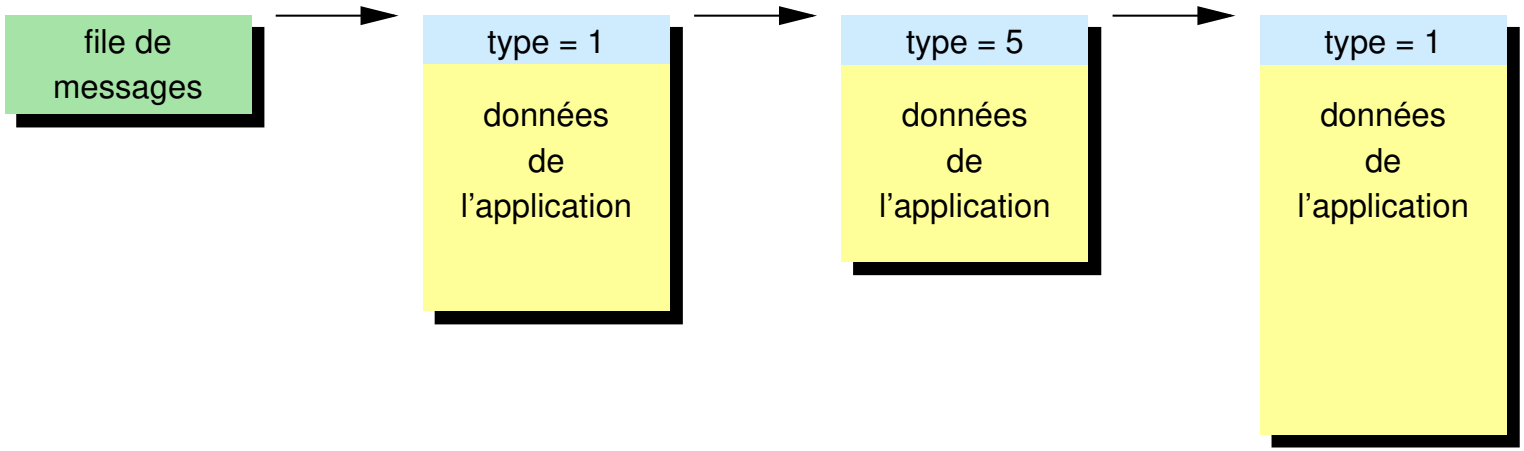
Cas spécial de clef : `IPC_PRIVATE`

- primitives `xxxget` créent un objet « anonyme »
  (aucune clef ne correspond à un tel objet)
- l'objet dispose d'un identificateur
  (retourné par les primitives `xxxget`)
- l'objet est partagé par le créateur et ses descendants
  (cf descripteurs de tube anonyme)
- attention toutefois : l'objet n'est pas implicitement détruit
  (suppression explicite nécessaire)

$\Rightarrow$ intéressant pour une communication entre processus apparentés (i.e. avec un ancêtre commun)

Files de messages

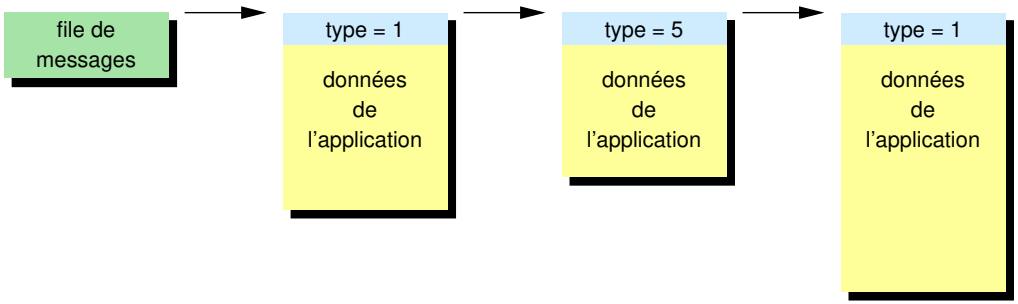
Exemple d'utilisation : les files de messages



Une file de message possède des attributs :

- permissions : lecture/écriture, utilisateur/groupe/autres
- dates de dernière opération : envoi, réception
- taille : nombre de messages, nombre d'octets
- etc.

Files de messages



Chaque message individuel :

- comprend en premier un `long int` (toujours > 0)
  $\Rightarrow$ permet au récepteur de sélectionner les messages
- le format du reste du message est à la charge de l'application
- lorsqu'on donne une taille de message, celle-ci est *sans* le `long int` (cf `msgsnd` et `msgrcv`)

Files de messages

```
int creer_file (void)
{
    key_t k ;
    int id ;

    k = ftok ("/etc/passwd", 'F') ;
    if (k == -1)
        raler ("ftok") ;

    id = msgget (k, IPC_CREAT | 0666) ;
    if (id == -1)
        raler ("msgget") ;

    return id ;
}
```

Files de messages

```
int acceder_file (void)
{
    key_t k ;
    int id ;

    k = ftok ("/etc/passwd", 'F') ;
    if (k == -1)
        raler ("ftok") ;

    id = msgget (k, 0) ;
    if (id == -1)
        raler ("msgget") ;

    return id ;
}
```

Files de messages

```
void supprimer_file (int id)
{
    int r ;

    r = msgctl (id, IPC_RMID, NULL) ;
    if (r == -1)
        raler ("msgctl") ;
}
```

Files de messages

```
struct message
{
    long mtype ;
    int val ;
} ;

void ecrire_valeur (int id, int val)
{
    struct message m ;   int r ;

    m.mtype = 25 ;
    m.val = val ;
    r = msgsnd (id, &m, sizeof m - sizeof m.mtype, 0) ;
    if (r == -1)
        raler ("msgsnd") ;
}
```

Files de messages

```
int lire_valeur (int id)
{
    struct message m ;   int r, val ;

    r = msgrcv (id, &m, sizeof m - sizeof m.mtype, 25, 0) ;
    if (r == -1)
        raler ("msgrcv") ;
    return m.val ;
}
```

Files de messages

Le type du message (toujours > 0) est utilisé comme critère d'extraction d'un message de la file (avec `msgrcv`)

Selon la valeur du paramètre `type` de `msgrcv` :

- si `type` = 0 : le premier message en attente est lu, quel que soit son type
- si `type` > 0 : le premier message en attente du type demandé est lu
- si `type` < 0 : le premier message en attente dont le type est inférieur ou égal à -`type` est lu
  exemple : si `type` = -25, lecture du premier message dont le type $\in$ [1..25]

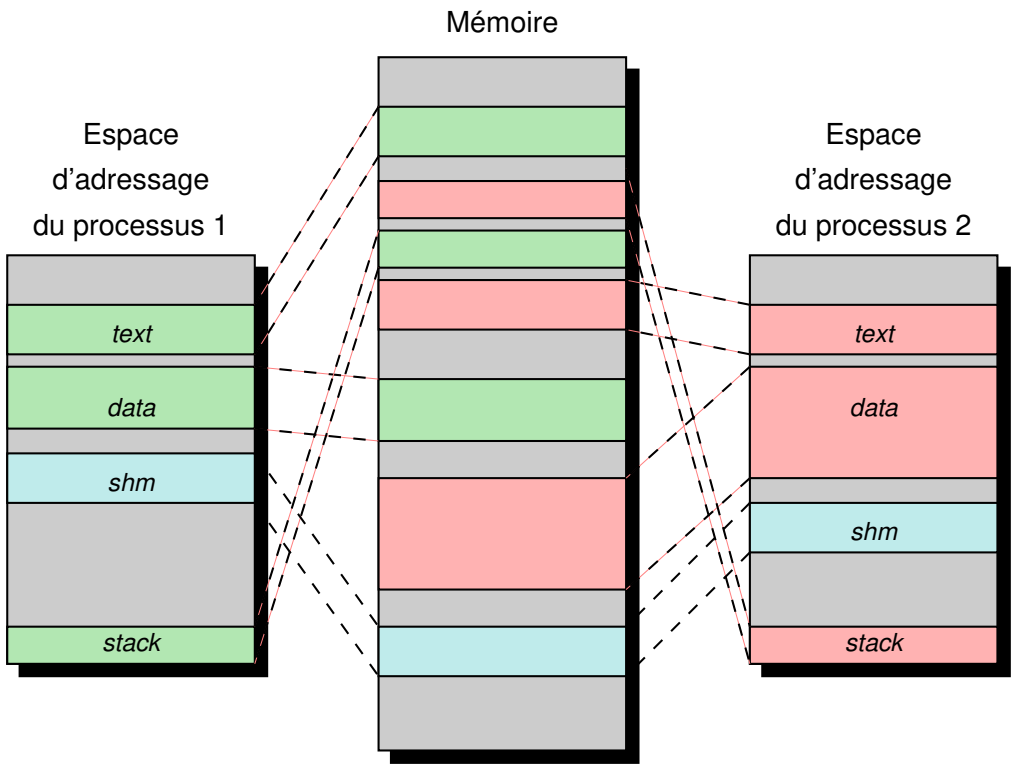
Mémoire partagée

Primitives d'accès à la mémoire partagée :

- `shmget` : crée ou accède à un objet de type « segment de mémoire partagée »
- `shmctl` : opérations (p. ex. suppression) sur un segment
- `shmat` : attache un segment existant dans l'espace d'adressage du processus courant
- `shmdt` : retire (détache) le segment de l'espace d'adressage

Mémoire partagée

Attachement d'un segment de mémoire :



Attention : le segment de mémoire n'est pas forcément attaché à la même adresse virtuelle dans les différents processus !

Mémoire partagée

```
int creer_segment (void)
{
    key_t k ;
    int id ;

    k = ftok ("/etc/passwd", 'M') ;
    if (k == -1)
        raler ("ftok") ;

    id = shmget (k, 4096, IPC_CREAT | 0666) ;
    if (id == -1)
        raler ("shmget") ;

    return id ;
}
```

Mémoire partagée

```
int acceder_segment (void)
{
    key_t k ;
    int id ;

    k = ftok ("/etc/passwd", 'M') ;
    if (k == -1)
        raler ("ftok") ;

    id = shmget (k, 0, 0) ;
    if (id == -1)
        raler ("shmget") ;

    return id ;
}
```

Mémoire partagée

```
void supprimer_segment (int id)
{
    int r ;

    r = shmctl (id, IPC_RMID, NULL) ;
    if (r == -1)
        raler ("shmctl") ;
}
```

Mémoire partagée

```
void ecrire_valeurs (int id, int v1, int v2)
{
    int *adr ;

    adr = shmat (id, NULL, 0) ;
    if (adr == (void *) -1)
        raler ("shmat␣ecrire") ;

    adr [0] = v1 ;
    adr [1] = v2 ;

    if (shmdt (adr) == -1)
        raler ("shmdt") ;
}
```

Mémoire partagée

```
void lire_valeurs (int id, int *v1, int *v2)
{
    int *adr ;

    adr = shmat (id, NULL, 0) ;
    if (adr == (void *) -1)
        raler ("shmat␣lire") ;

    *v1 = adr [0] ;
    *v2 = adr [1] ;

    if (shmdt (adr) == -1)
        raler ("shmdt") ;
}
```

Mémoire partagée

Attention : le segment de mémoire n'étant pas forcément attaché à la même adresse virtuelle dans les différents processus, ne jamais mettre de pointeur dans les segments de mémoire partagée.

Plan

Threads

Les processus ne sont pas adaptés au parallélisme à grain fin

- commutation de processus lente
  $\Rightarrow$ sauvegarde et restauration du contexte, sélection du processus, *changement d'espace d'adressage* et *invalidation du TLB*
- communication inter-processus (tubes, messages) lente
  $\Rightarrow$ création de la mémoire partagée
- manque d'outils de synchronisation

Threads

Début des années 1980 : apparition des *threads* (ou *processus légers*)

- plusieurs fils d'exécution
- un seul espace mémoire

Idée : alléger la commutation de processus
$\Rightarrow$ faire le maximum de choses en mode utilisateur

- éliminer la commutation d'espace mémoire (sauvegardes/restaurations du contexte MMU)
- permettre l'ordonnancement en mode utilisateur
- accès direct au support matériel pour la synchronisation

Threads

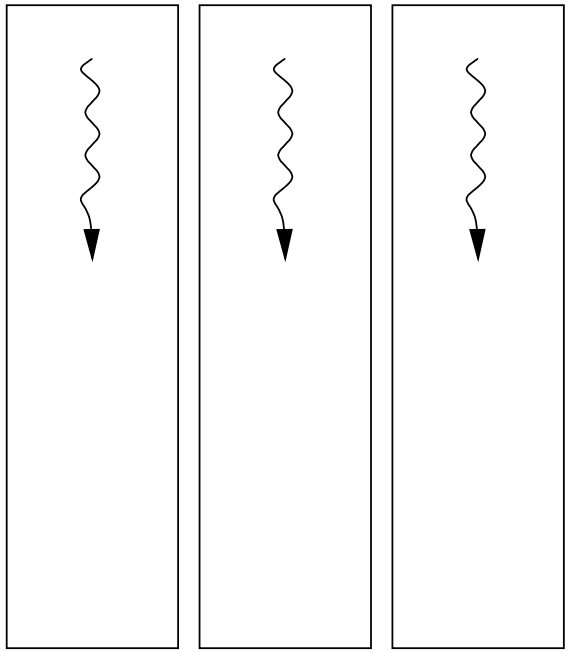
Les threads d'un processus partagent :

- l'identificateur du processus et son propriétaire
- l'*espace mémoire unique* du processus
- les fichiers ouverts
- les actions associées aux signaux
- etc.

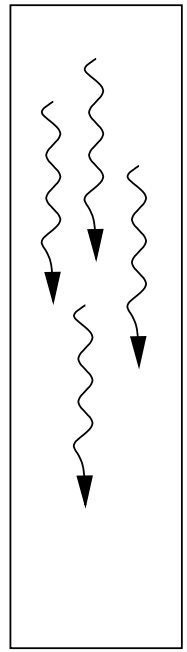
Chaque thread dispose :

- de son identificateur de thread
- de son propre fil d'exécution
  $\Rightarrow$ sauvegarde et restauration du contexte CPU (sans le contexte MMU)
- de sa propre pile d'exécution

Threads



Processus "lourds"
un fil d'activité par processus



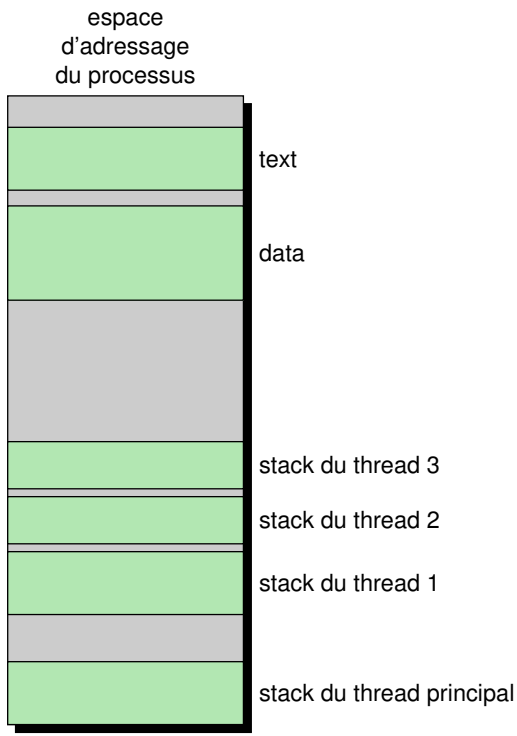
Threads

- chaque processus a son propre espace d'adressage et un fil d'activité
- les différents threads sont autant de fils d'activité se partageant un même espace d'adressage

Threads

Espace d'adressage du processus :

- zone « data » : variables globales
  $\Rightarrow$ partagées par les threads
- zone « stack » : chaque thread a sa propre pile d'exécution
  Attention : taille de la pile bornée
  $\Rightarrow$ pas d'accroissement possible
  $\Rightarrow$ taille fixée à la création
  $\Rightarrow$ pas de détection de saturation !



Plan

Threads

API normalisée par POSIX

- fichier d'inclusion : `#include <pthread.h>`
- édition de liens : `cc -o ...-l pthread`

Types :

- `pthread_t` : identificateur de thread
- `pthread_attr_t` : attribut de thread

Attention : contrairement aux primitives systèmes classiques, les fonctions `pthread_*` renvoient 0 en cas de succès ou un numéro d'erreur ($>$ 0) sinon.

Threads

Fonctions de l'API :

- création de thread :

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

- fin de thread :

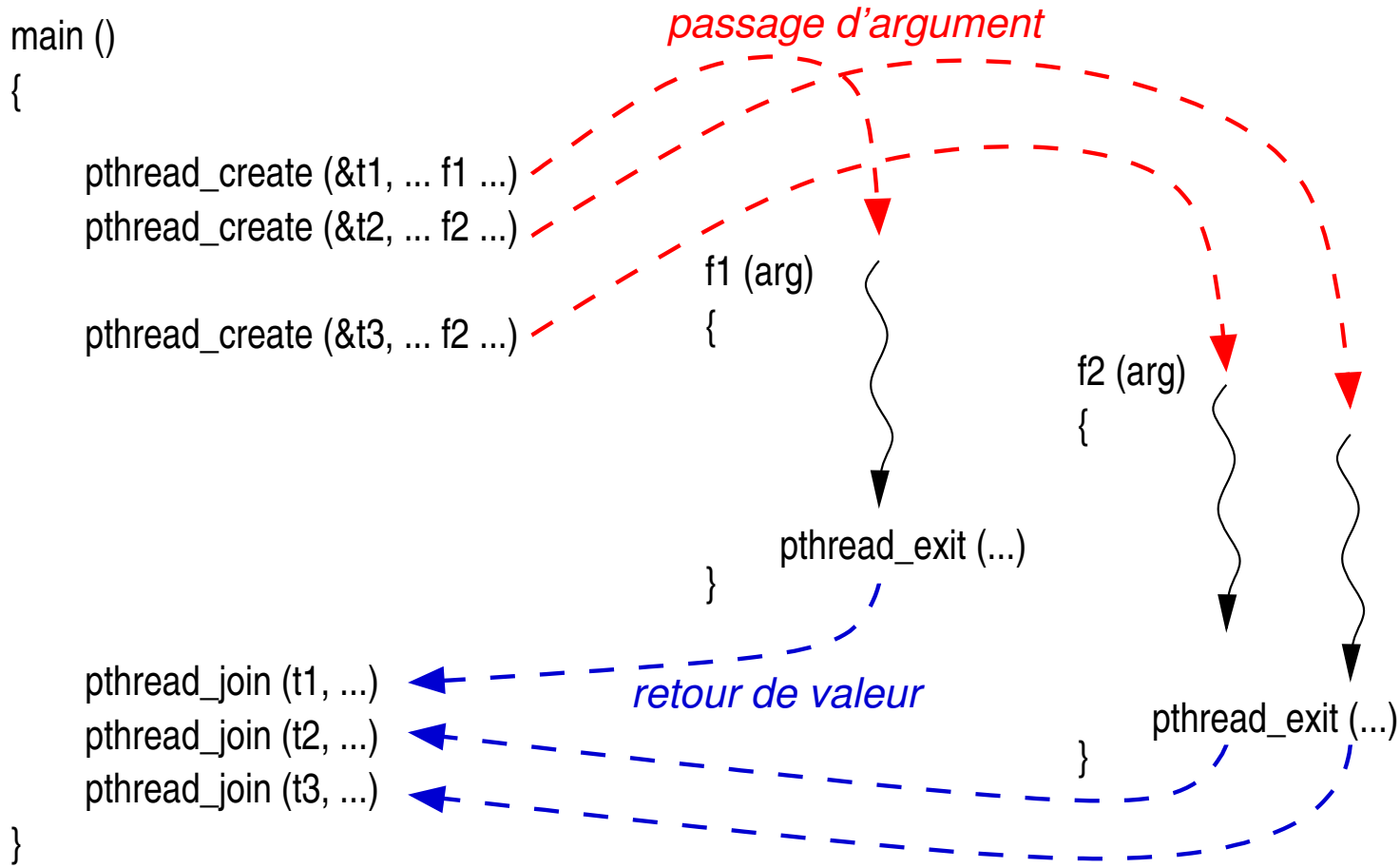
```
int pthread_exit (void *valretour)
```

- attente de fin de thread :

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

Threads

Illustration : 4 threads (principal + 3 nouveaux)



Threads

```
int nthread ;

void *f (void *arg)
{
    int i = * (int *) arg ;
    int *p ;

    printf ("Je␣suis␣le␣thread␣%d␣(sur␣%d),␣tid␣=%ld\n",
                i, nthread, pthread_self ()) ;

    p = malloc (sizeof (int)) ;
    *p = i * 2 ;

    pthread_exit (p) ;          /* <=> return p */
}
```

Threads

```
int main (int argc, char *argv [])
{
    pthread_t *tid ;       int e, i, *ti ;

    nthread = atoi (argv [1]) ;
    tid = malloc (nthread * sizeof (pthread_t)) ;
    ti = malloc (nthread * sizeof (int)) ;
    for (i = 0 ; i < nthread ; i++)
    {
        ti [i] = i ;
        if ((e=pthread_create(&tid[i], NULL, f, &ti[i])) != 0)
            raler (e, "pthread_create") ;
    }

    for (i = 0 ; i < nthread ; i++)
    {
        void *r ;
        if ((e = pthread_join (tid [i], &r)) != 0)
            raler (e, "pthread_join") ;
        printf ("Retour␣du␣thread␣%d␣=␣%d\n", i, * (int *) r) ;
        free (r) ;
    }
    free (tid) ; free (ti) ;
}
```

Fonctions « thread-safe »

La plupart des fonctions de bibliothèque ont été créées avant l'apparition des threads

- La plupart n'ont pas nécessité de modifications
  Exemples : `strlen`, `isalpha`, etc.
- Pour certaines, l'implémentation a dû être modifiée
  $\Rightarrow$ ajout de synchronisations (cf cours suivant)
- Quelques unes **ne peuvent pas** être compatibles avec les threads :
  - stockage d'un état dans une variable unique
  - retour d'une adresse pointant dans une variable unique

  $\Rightarrow$ Problème de conception des fonctions

Fonctions « thread-safe » – État global

Stockage d'un état dans une variable unique :

- Exemple (découpage d'une chaîne en tokens) :

```
char *strtok (char *str, char *delim) {
  static char *last ;
  /* variable last unique pour tous les threads */
  ...
}
```

- L'état actuel du parcours dans `str` est mémorisé dans `last`
  - Si deux threads utilisent `strtok` sur des chaînes différentes, la variable `last` renvoie à l'un des threads un pointeur dans la chaîne de l'autre thread
- Problème inhérent à la définition de `strtok`

Fonctions « thread-safe » – État global

- Définition d'une nouvelle fonction
- `char *strtok_r (char *s, char *delim, char **parcours)`
- La variable `last` est « sortie » de `strtok` via `parcours`

Fonctions « thread-safe » – Retour global

Retour d'adresse pointant dans une variable unique :

- Exemple (lecture de répertoire) :

```
struct dirent *readdir (DIR *dirp) {
  /* variable d unique pour tous les threads */
  static struct dirent d ;
  ...
  return &d ;
}
```

  - Si deux threads utilisent `readdir`, la lecture par le deuxième thread remplace la donnée que le premier thread n'a peut être pas encore exploitée
  - `readdir` ne peut pas faire autrement : la spécification des fonctions `xxxdir` ne permet pas d'allouer ou de libérer la mémoire utilisée par `readdir`

- Problème inhérent à la définition de `readdir`

Fonctions « thread-safe » – Retour global

- Définition d'une nouvelle fonction
- `int readdir_r (DIR *dirp, struct dirent *entry, struct dirent **result)`
- Le résultat est placé dans une zone de mémoire allouée par l'appelant
  $\Rightarrow$ adresse précisée par `entry`
  $\Rightarrow$ adresse de `entry` placée dans `result`
- En fin de répertoire, `NULL` est placé dans `result`

Fonctions « thread-safe »

Attention donc aux fonctions qui ne sont pas « thread-safe » :

- `rand` ⇒ `rand_r`
- `getlogin` ⇒ `getlogin_r`
- `ttyname` ⇒ `ttyname_r`
- `ctime` ⇒ `ctime_r`
- etc.

Le manuel est votre ami : le manuel d'une fonction `x` mentionne également la fonction `x_r` si elle existe

Fonctions « thread-safe » – Variable errno

Et la variable `errno` ?

- Avant les threads, `errno` était une variable globale
- Avec les threads, `errno` devient une variable propre à chaque thread

  $\Rightarrow$ il est donc possible d'écrire :

```
errno = pthread_... (...) ;
if (errno != 0)
    raler ("pthread_...") ;
```

Attributs de threads

Les threads peuvent avoir des attributs
$\Rightarrow$ positionnés lors de la création du thread

- `int pthread_attr_init (pthread_attr_t *attr)`
- `int pthread_attr_destroy (pthread_attr_t *attr)`

Le type `pthread_attr_t` :

- est opaque
- peut référencer plusieurs attributs
- est ensuite passé à `pthread_create`

Attributs de threads

Exemple :

```
pthread_attr_t pa ;

errno = pthread_attr_init (&pa) ;
if (errno != 0)
    raler ("pthread_attr_init") ;

errno = pthread_attr_setstacksize (&pa, 2 * PAGE_SIZE) ;
if (errno != 0)
    raler ("pthread_attr_setstacksize") ;

errno = pthread_create (..., &pa, ....) ;
if (errno != 0)
    raler ("pthread_create") ;

pthread_attr_destroy (&pa) ;
```

Plan

Threads et signaux

Les threads d'un processus partagent :

- l'action associée à chaque signal

```
int sigaction (int signum, const struct sigaction
*new, struct sigaction *old)
```

Chaque thread a en propre :

- son masque de signaux (hérité du masque du processus)

```
int pthread_sigmask (int comment, const sigset_t
*new, sigset_t *old)
```

- les signaux en attente de réception

Threads et signaux

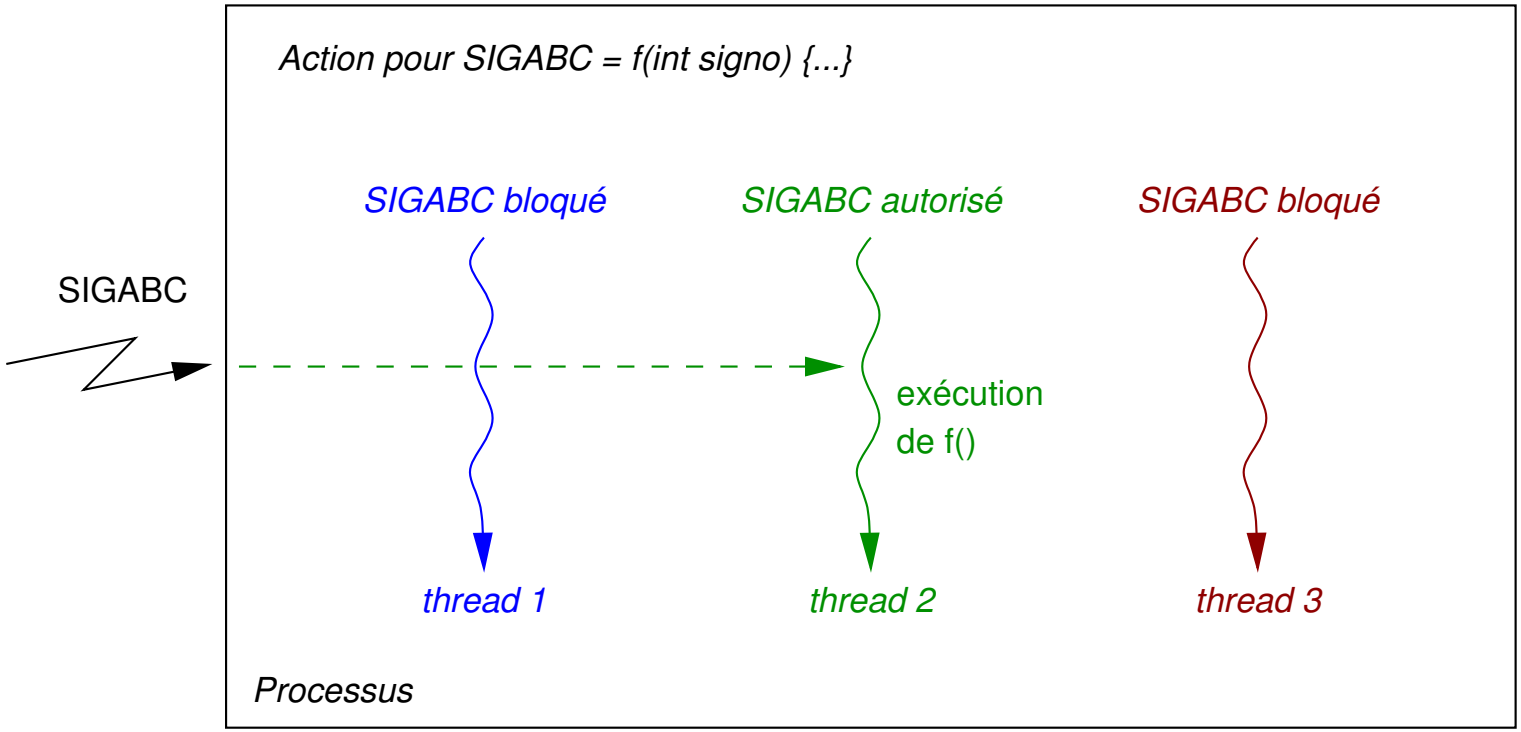
Un signal peut être envoyé à un thread en particulier :

- à un thread désigné
  `int pthread_kill (pthread_t id, int sig)`
- au thread fautif
  Exemple : SIGSEGV en cas de problème de pointeur

Si un signal n'est pas dirigé vers un thread (`kill()` ou événement externe comme SIGINT), il est envoyé à un thread arbitraire :

- à l'un des threads qui attendent le signal
  `int sigwait (const sigset_t *set, int *sig)`
- à l'un des threads qui autorisent le signal

Threads et signaux



Plan

Barrières

Les threads peuvent utiliser de nombreux mécanismes de synchronisation :

- verrous
- variables de condition
- barrières
- etc.

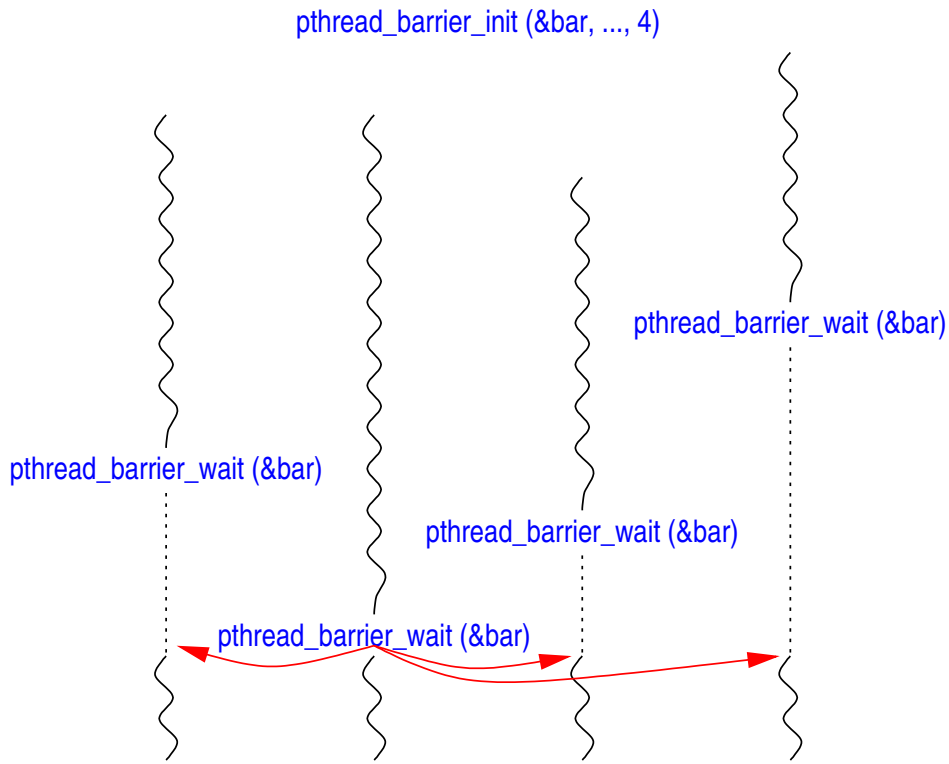
Commençons par un mécanisme simple : la barrière

Barrières

Principe :

- une barrière de n threads est bloquante pour les $n - 1$ premiers threads
- lorsque le n-ème thread se présente à la barrière, les n threads sont libérés
- la barrière est alors de nouveau prête à fonctionner

Barrières



Valeur de retour de `pthread_barrier_wait` :

- `PTHREAD_BARRIER_SERIAL_THREAD` pour l'un des n threads
- `0` pour les $n - 1$ autres threads.

Barrières – Attributs

Les mécanismes de synchronisation peuvent aussi avoir des attributs :

- `int pthread_barrierattr_init (pthread_barrierattr_t *attr)`
- `int pthread_barrierattr_destroy (pthread_barrierattr_t *attr)`

Exemple : barrière partagée entre plusieurs processus ou restreinte aux threads du processus courant

- `pthread_barrierattr_setpshared ()` `pthread_barrierattr_getpshared ()`
- valeur `PTHREAD_PROCESS_SHARED` ou `PTHREAD_PROCESS_PRIVATE`

Si la barrière est partagée entre processus, il faut qu'elle soit en mémoire partagée.

Barrières – Attributs

Exemple :

```
pthread_barrier_t b ;
pthread_barrierattr_t ba ;

errno = pthread_barrierattr_init (&ba) ;
if (errno != 0)
    raler ("pthread_barrierattr_init") ;

errno = pthread_barrierattr_setpshared (&ba,
                                PTHREAD_PROCESS_SHARED) ;
if (errno != 0)
    raler ("pthread_barrierattr_setpshared") ;

errno = pthread_barrier_init (&b, &ba, 4) ;
if (errno != 0)
    raler ("pthread_barrier_init") ;

pthread_barrierattr_destroy (&ba) ;
```