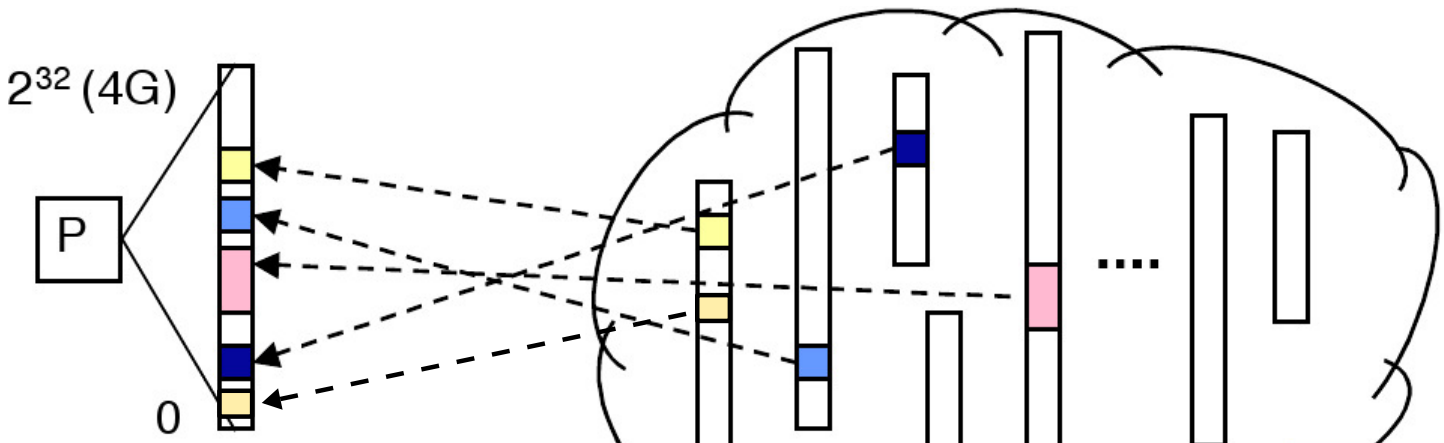
Mémoire virtuelle

Philippe Quéinnec

27 avril 2020

Objectifs

- Donner l'illusion d'une mémoire infinie à chaque processus
- Gérer le partage effectif de la mémoire physique

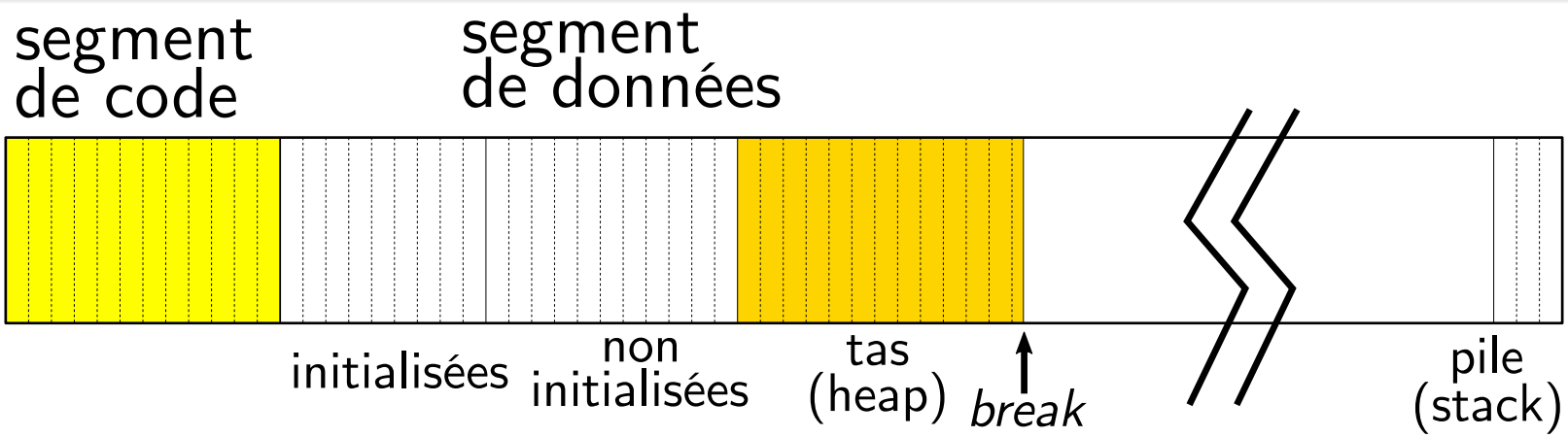


Information accessible à un instant donné
(sur 32 bits d'adresse)

Information potentiellement accessible

Source : S. Krakowiak ≪ Introduction aux systèmes et réseaux informatiques ≫.

Évolution de l'espace virtuel



- Initialement, l'espace virtuel d'un processus est constitué par :
  - le code (`execv` ou code parent si `fork`)
  - les données « statiques » initialisées et non initialisées (variables globales, constantes, chaînes de caractères. . .)
  - un tas vide
  - une amorce de pile (`main`), qui grandira à l'exécution
- À l'exécution, le *program break* peut être déplacé pour agrandir le tas : directement ou indirectement (`malloc`, `new`)

(très simplifié : chargement de code à la demande, plusieurs piles. . .)

Les principes

Les deux grandes idées

- La pagination : découper l'espace mémoire en pages de taille fixe, et distinguer adresse virtuelle / adresse physique
- Le va-et-vient : éjection de pages mémoire sur disque

Les méthodes

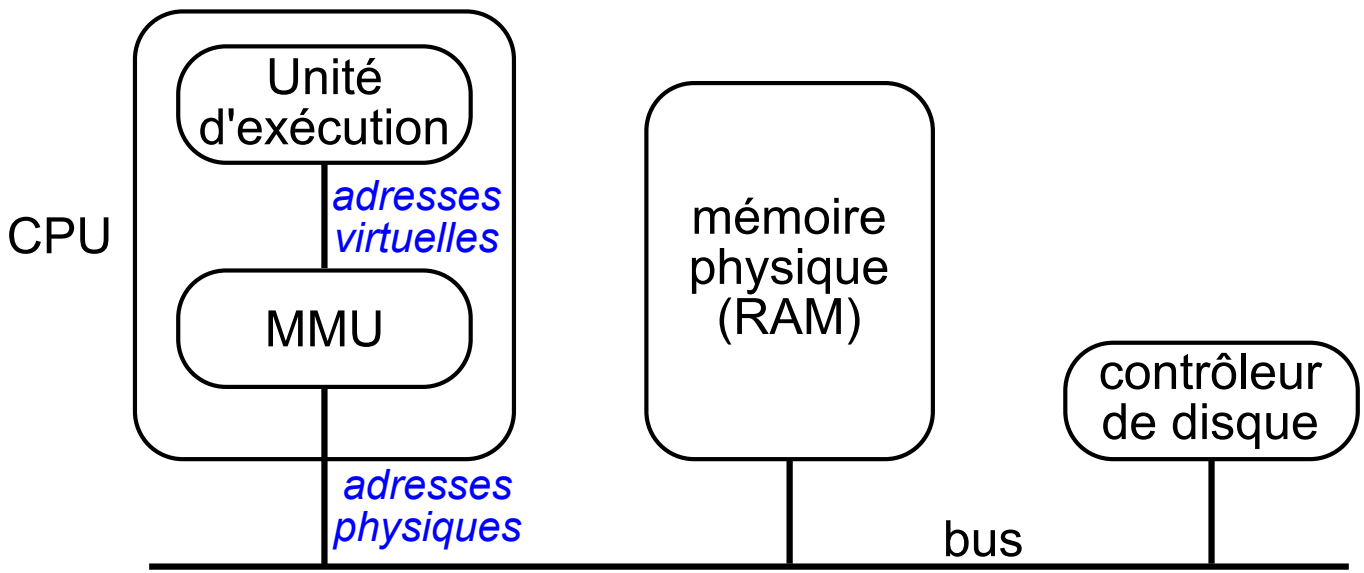
- Table des pages : correspondance page virtuelle $\leftrightarrow$ page physique
- Algorithmes de remplacement de pages
- Le partage et le verrouillage de pages

Plan

1. La pagination
   - Principe
   - La table des pages
   - Le partage de pages
2. Le va-et-vient
   - Expulsion sur disque
   - Algorithmes de remplacement de pages
   - Verrouillage
3. L'API Unix
4. Mise en œuvre dans notre noyau

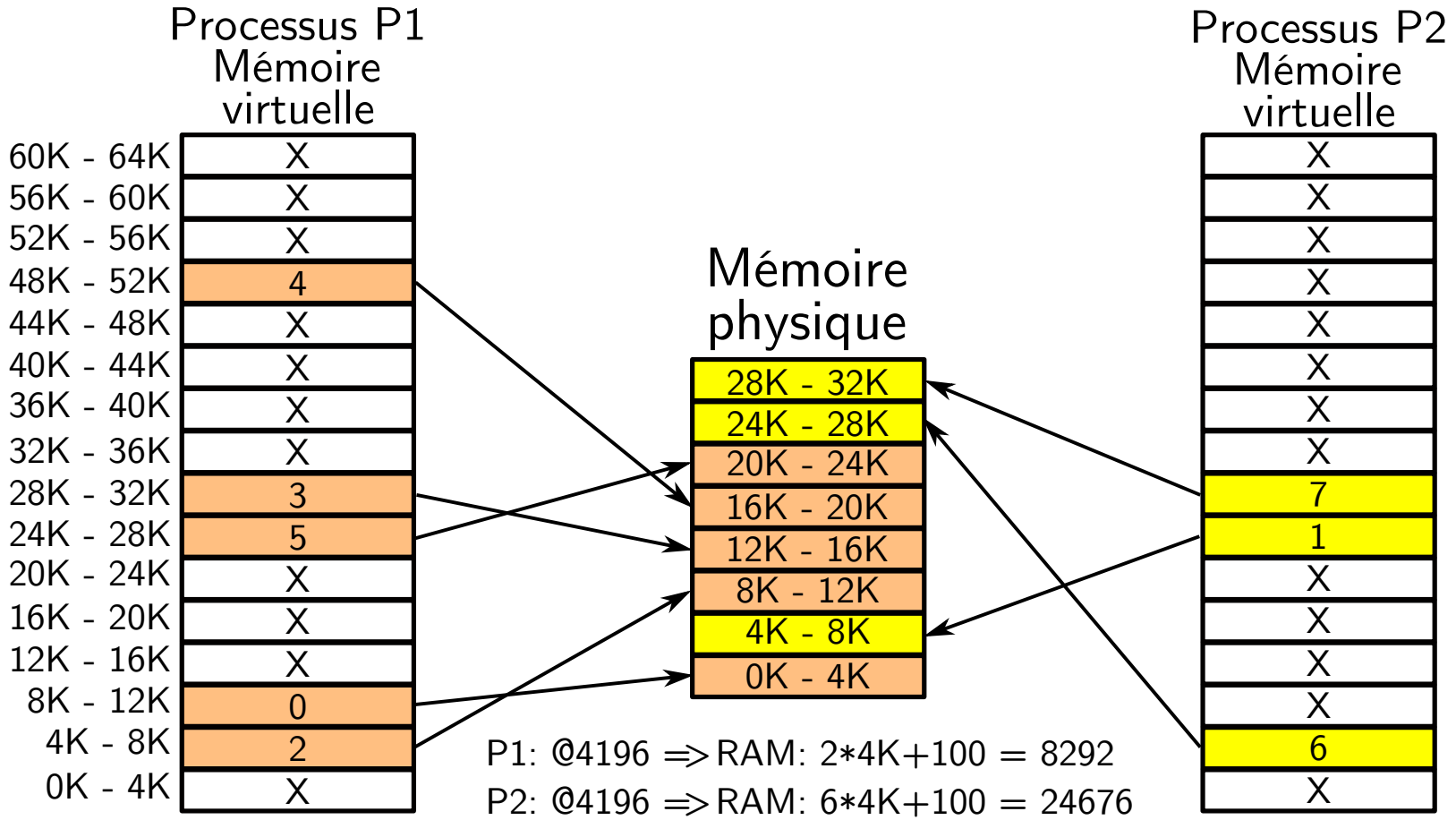
La pagination

- La mémoire virtuelle est découpée en pages (virtuelles)
- La mémoire physique est découpée en pages (physiques)
- Les pages ont une taille P fixe
- La MMU (Memory Management Unit) traduit les adresses virtuelles en adresses physiques



Translation d'adresse

À chaque page virtuelle correspond une page physique ou une erreur.



Translation d'adresse

À chaque page virtuelle correspond une page physique ou une erreur (accès illégal, le célèbre *segmentation fault*).
Adresse virtuelle = (n° page virtuelle, déplacement dans la page)

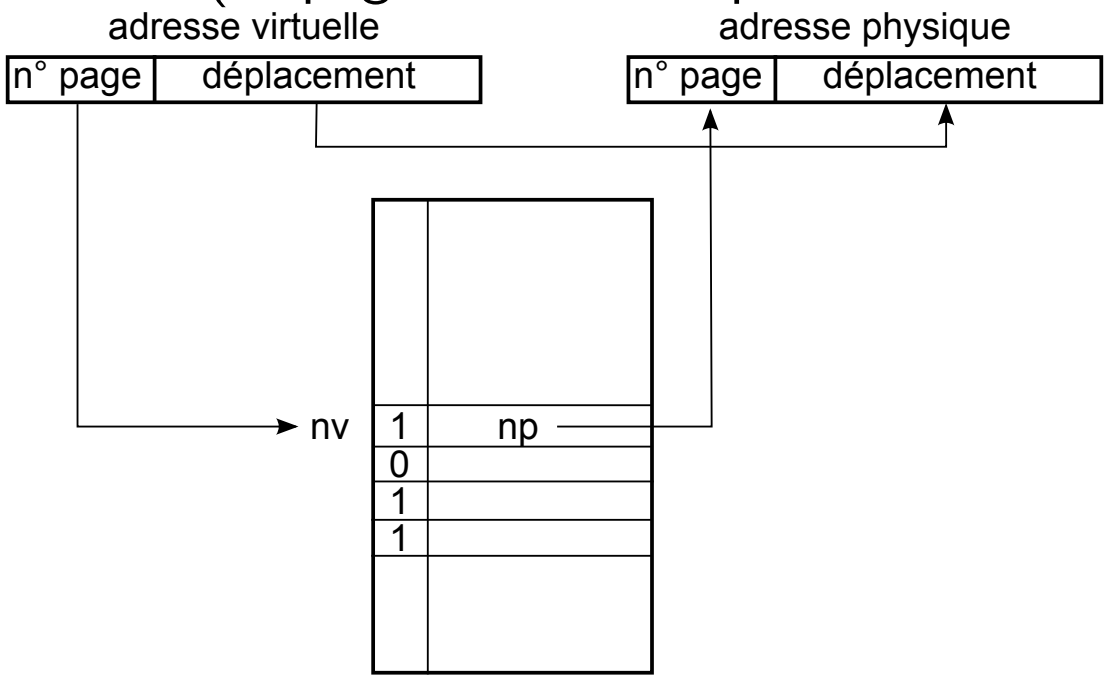


Table des pages élémentaire

Réalisation directe :

- Table page virtuelle → page physique
- Une table par processus
- Informations supplémentaires :
  - présent/absent en mémoire (pour le va-et-vient)
  - protection : accessibilité en lecture / écriture / exécution
  - modifiée et/ou référencée (pour l'algorithme de remplacement)
- Commutation de processus ⇒ commutation de la table des pages dans la MMU
- Mise à jour de la table :
  - Création d'un processus = espace virtuel initial → allocation de pages physiques libres pour code et données initiales
  - Quand un processus grossit (`malloc`) = utilisation d'une nouvelle page virtuelle → allocation d'une page physique libre
  - Quand un processus termine → libération de toutes les pages physiques associées à ses pages virtuelles

Table des pages élémentaires

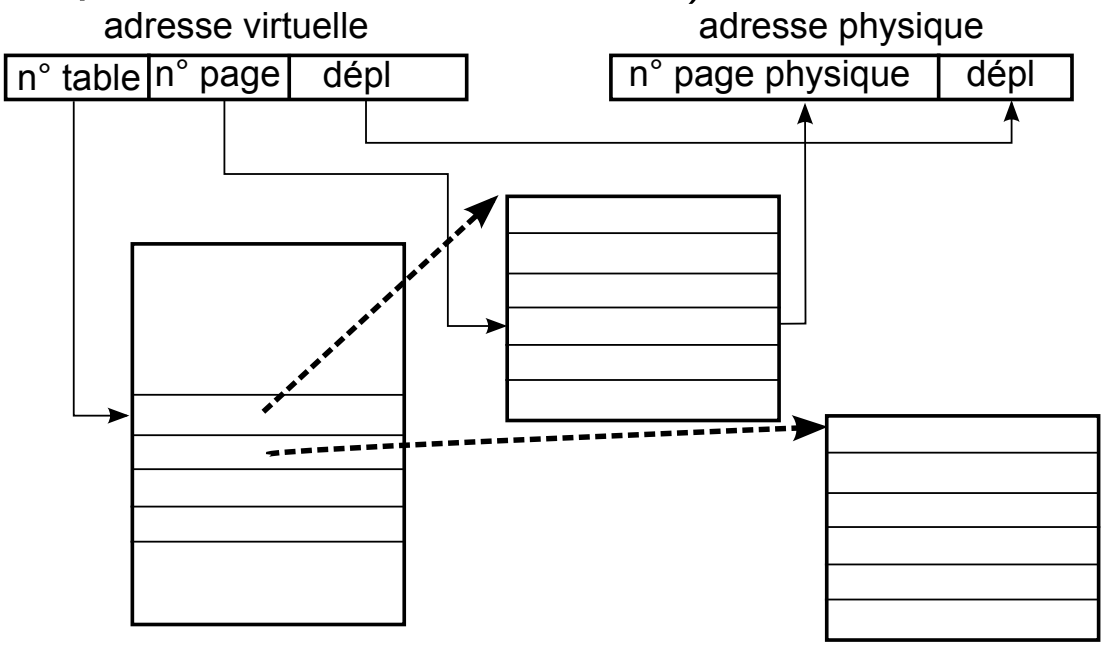
Ceci est un modèle, pas (plus) une implantation !

- Espace d'adressage sur 32 bits et pages de 4Ko $\rightarrow 2^{20}$ pages avec 20+6 bits d'informations $\rightarrow$ 3,25 Mo par processus.
  Machine 64 bits et pages de 4 Ko $\rightarrow$ 13312 Po !
  $\Rightarrow$ on va devoir ruser.
- Trop gros pour être dans la MMU + basculement à chaque commutation
  $\Rightarrow$ table en mémoire RAM avec un registre la pointant dans la MMU
  $\Rightarrow$ tout accès par une adresse virtuelle nécessite deux accès en RAM (un pour lire la table de translation, un pour faire l'opération lecture ou écriture demandée)

Table des pages multi-niveaux

Objectif : réduire la taille des informations nécessaires

- Décomposer l'adresse en (n° de table, n° de page, dépl.)
- Utiliser une double indirection (table de tables)
- Espérer que la majorité des premiers niveaux soient inutilisés
  $\Rightarrow$ beaucoup moins gros
- Tout accès virtuel nécessite trois accès en RAM (à moins de mettre le premier niveau en MMU)



Mémoire associative / Translation Lookaside Buffer (TLB)

Objectif : optimiser la translation d'adresse

- Conserver dans un circuit périphérique à la MMU un petit nombre de correspondances page virtuelle $\rightarrow$ page physique
- Recherche simultanée (en parallèle) dans toutes les entrées
- En cas d'insuccès, utiliser la méthode normale (coûteuse)
- Mise à jour de la mémoire associative en fonction des pages accédées (stratégie LRU ou NRU, similaire à la gestion du cache processeur)
- En pratique taille du TLB = 16 à 512 entrées
- Localité des accès $\Rightarrow$ énorme gain d'efficacité.

Table des pages inversée

Objectif : réduire drastiquement les informations nécessaires, sous l'hypothèse que la mémoire physique est largement plus petite que la mémoire virtuelle

- Table inversée : page physique $\rightarrow$ page virtuelle
- RAM 1Go, page 4Ko $\rightarrow 2^{18}$ pages $\rightarrow$ 4 Mo d'informations (pour tout le système)
- Recherche inefficace $\Rightarrow$ table de hachage indexée par les pages virtuelles, de taille approximative le nombre de pages physiques $\Rightarrow$ peu de conflit O(1)
- $+$ TLB $\Rightarrow$ bonnes performances pour des espaces d'adressage 64 bits
- Encore mieux : remplacer la table inversée par un arbre équilibré

Partage

- Possibilité d'avoir une même page physique qui correspond à des pages virtuelles de plusieurs processus
- Intéressant pour des données non modifiables : code, librairies partagées
- Le code doit être non localisé = indépendant de son emplacement en mémoire (il n'est pas nécessairement à la même adresse virtuelle dans tous les processus)

Copie sur écriture

- Deux processus travaillent sur la même page physique, autorisée en lecture seulement.
- Si l'un écrit, la page est dupliquée, chacun des exemplaires est associé à la mémoire virtuelle d'un des processus, et les pages deviennent modifiables.
- Retarde la duplication à la première modification.
- Utilisé pour l'implantation de `fork` : le processus fils partage (virtuellement) toutes les pages du père, la copie effective n'a lieu que si nécessaire. Grosse économie en cas de `fork` suivi d'`exec`.

Plan

1. La pagination
   - Principe
   - La table des pages
   - Le partage de pages
2. Le va-et-vient
   - Expulsion sur disque
   - Algorithmes de remplacement de pages
   - Verrouillage
3. L'API Unix
4. Mise en œuvre dans notre noyau

Va-et-vient (swapping)

- Hiérarchie mémoire : RAM / disque = mémoire principale / mémoire secondaire
- Capacité ↑, vitesse ↓
  (disque 20 à 200 fois plus lent que la RAM)
- Expulser des pages de la mémoire principale vers la mémoire secondaire quand il n'y a plus de place en mémoire principale
- Plusieurs approches :
  - Approche globale : expulser l'intégralité du processus (approche historique)
  - Approche dynamique : expulser page par page au fur et à mesure des besoins
  - Anticipation : profiter de l'inactivité disque pour écrire certaines pages

Expulsion de page

Un processus référence une page de sa mémoire virtuelle qui n'est pas en mémoire physique → défaut de page.
La gestion du défaut de page doit :

- vérifier que l'adresse virtuelle est légale (qu'il y a un contenu associé)
- trouver une page physique disponible (libre)
- pour cela, choisir une page et l'expulser : algorithmes de remplacement de pages
- quand on expulse une page, nécessité de savoir s'il faut l'écrire sur disque : indication de page modifiée positionnée lors d'une écriture dans cette page
- charger depuis le disque le contenu de la page virtuelle qui a causé le défaut de page
- 1 accès mémoire avec défaut de page ⇒ 2 accès disque !

Gestion du swap

- Zone banalisée sur disque
- Une page = N blocs consécutifs
- Savoir si un groupe de blocs est occupé ou pas ⇒ table de bits
- Réservation au démarrage du processus (combien ?) ou à la demande (risque d'échec)
- Optimisation : pour les pages non modifiables d'un programme (code), il n'est pas nécessaire de l'écrire dans le swap : utiliser directement le fichier exécutable comme zone d'expulsion (sans écriture nécessaire)

Gestion du défaut de page

1. MMU déclenche « accès impossible » ⇒ *trap* vers le noyau ;
2. Le noyau calcule le numéro de page virtuelle et vérifie qu'elle est légale ; sinon envoi d'un signal (SIGSEGV) au processus ;
3. Le processus ayant causé le défaut de page est suspendu ;
4. Le noyau détermine si une page physique est libre. Si non, il en choisit une (algorithme de remplacement) ;
5. Si cette page physique a été modifiée, il donne l'ordre au contrôleur d'E/S de l'écrire dans le swap et fait une commutation de contexte pour exécuter un autre processus ;
6. Quand la page physique est libre (non modifiée ou écriture terminée), le noyau détermine la page disque (ensemble de blocs) à charger et demande le chargement ; nouvelle commutation ;
7. Quand la lecture est terminée, les tables de page sont mises à jour, le processus ayant causé le défaut de page est remis à l'état prêt et de telle manière que l'instruction responsable soit reéxécutée (modification du compteur ordinal ou support matériel).

Remplacement de pages : objectifs

- Comment déterminer la page à expulser ?
- Idéalement, celle qui sera utilisée le plus loin dans le futur.
- En pratique, approximer le futur par le passé : principe de localité temporelle des accès.

NRU – Not Recently Used

- A chaque page est associée deux bits : bit R (référencée) + bit M (modifiée/dirty)
- Bit mis à jour à chaque accès mémoire (hardware de la MMU)
- Quand une page est chargée depuis le disque suite à un défaut de page : R=1, M=0
- Périodiquement (à chaque interruption d'horloge / chaque préemption) : R=0 pour toutes les pages
- Quatre classes :
  - classe 0 : non référencée, non modifiée
  - classe 1 : non référencée, modifiée
  - classe 2 : référencée, non modifiée
  - classe 3 : référencée, modifiée
- Algorithme NRU : choix d'une page au hasard dans la classe la plus basse non vide
- Algorithme grossier, approximatif mais très peu coûteux

LRU – Least Recently Used

- Un compteur global + un compteur pour chaque page
- À chaque top, incrémenter le compteur global
- À chaque accès, compteur de la page ← compteur global
- Algorithme LRU = choix de la page la plus ancienne
- Algorithme excellent mais difficile à implanter à faible coût (parcours des pages + taille des compteurs)

NFU – Not Frequently Used

- Un compteur sur n bits par page
- À chaque top et pour *toutes* les pages, décaler le compteur à *droite* et positionner bit de poids *fort* à la valeur du bit R (référence)
- Algorithme NFU = choix d'une page ayant le plus petit compteur (= référencée le plus anciennement)
- Avec n bits, le système « oublie » un référencement après n tops
- Bonne approximation du LRU avec mémoire bornée

Ensemble de travail / working set

- Constat qu'un processus n'utilise, sur une période donnée, qu'un petit ensemble de pages
- Garder trace de cet ensemble
- précharger les pages $\Rightarrow$ pas de défaut de page, peu de risque d'expulser une page nécessaire « bientôt »
- existence d'implantations simples et performantes

Verrouillage en mémoire

- Parfois nécessaire d'interdire l'expulsion
- Exemple : entrée-sortie directe en mémoire

Plan

1 La pagination
- Principe
- La table des pages
- Le partage de pages

2 Le va-et-vient
- Expulsion sur disque
- Algorithmes de remplacement de pages
- Verrouillage

3 L'API Unix

4 Mise en œuvre dans notre noyau

Changer la protection d'une zone mémoire

mprotect

```
int mprotect(void *addr, size_t len, int prot);
```

où `prot` est un « ou » logique entre :

- `PROT_NONE` zone inaccessible
- `PROT_READ` zone accessible en lecture
- `PROT_WRITE` zone accessible en écriture
- `PROT_EXEC` zone accessible en exécution (code)

Un accès illégal se traduit par la réception d'un SIGSEGV.

Rq : la protection porte sur l'ensemble des pages *virtuelles* du processus appelant, et contenant la zone [addr, addr+len[.

Projection en mémoire

mmap

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

Projection en espace virtuel de fichier

Flags (« ou » logique) :

- `MAP_PRIVATE` ou `MAP_SHARED` : privé au processus ou partageable
- `MAP_FILE` ou `MAP_ANONYMOUS` : associé à un fichier ou pas

mumap

```
int munmap(void *addr, size_t length);
```

Fin de la projection.

Les actions portent sur l'ensemble des pages virtuelles du processus appelant contenant la zone [addr, addr+length[.

Cas d'utilisation

- Projeter une portion de fichier en mémoire :

```
int fd = open("toto", O_RDONLY);
char *adr = mmap(NULL, 256, PROT_READ,
                 MAP_PRIVATE|MAP_FILE, fd, 0);
for (i = 0; i < 256; i++) { x = x + adr[i]; }
```

- Projeter une portion de fichier en modification :

```
int fd = open("toto", O_RDWR);
char *adr = mmap(NULL, 256, PROT_READ|PROT_WRITE,
                 MAP_SHARED|MAP_FILE, fd, 0);
for (i = 0; i < 256; i++) {
    x = x + adr[i]; adr[i] = adr[i] * 2;
}
munmap(adr, 256);
```

Cas d'utilisation (2)

- Partager une zone mémoire entre deux processus :

```
char *adr = mmap(NULL, 256, PROT_READ|PROT_WRITE,
                 MAP_SHARED|MAP_ANONYMOUS, -1, 0);
if (fork() == 0) { /* fils */
    adr[10] = 'a';
} else { /* père */
    sleep(1);
    printf("%c\n", adr[10]); /* ça affiche 'a' ! */
}
```

Verrouillage en mémoire

mlock

```
int mlock(void *addr, size_t len);
int munlock(void *addr, size_t len);
```

Verrouille en mémoire physique les pages virtuelles de l'intervalle [addr,addr+len[

Nécessite des droits particuliers.

Conseils de pagination

madvise

```
int madvise(void *addr, size_t len, int advice);
```

où advice vaut :

MADV_NORMAL comportement par défaut

MADV_RANDOM utilisation disparate ⇒ lecture anticipée peu utile

MADV_SEQUENTIAL utilisation séquentielle ⇒ anticiper la page suivante

MADV_WILLNEED cette plage d'adresse va bientôt servir ⇒ anticiper le chargement et éviter l'expulsion

MADV_DONTNEED cette plage d'adresse ne va plus servir prochainement ⇒ peut être expulsée sur disque

⋮

(en pratique, peu d'intérêt)

SIGSEGV

Que se passe-t-il en cas d'accès illégal (adresse virtuelle sans adresse physique existante, lecture d'une zone protégée en lecture, modification d'une zone protégée en écriture) ?

$\Rightarrow$ au niveau matériel : interruption
au niveau unix : signal SIGSEGV (segmentation violation)

S'il y a un gestionnaire de signal connecté via `sigaction`, le gestionnaire est appelé *puis l'instruction responsable est reexécutée à l'identique*.
Le gestionnaire reçoit en paramètre (entre autre) l'adresse responsable et la cause de l'accès illégal (p.e. permission invalide).

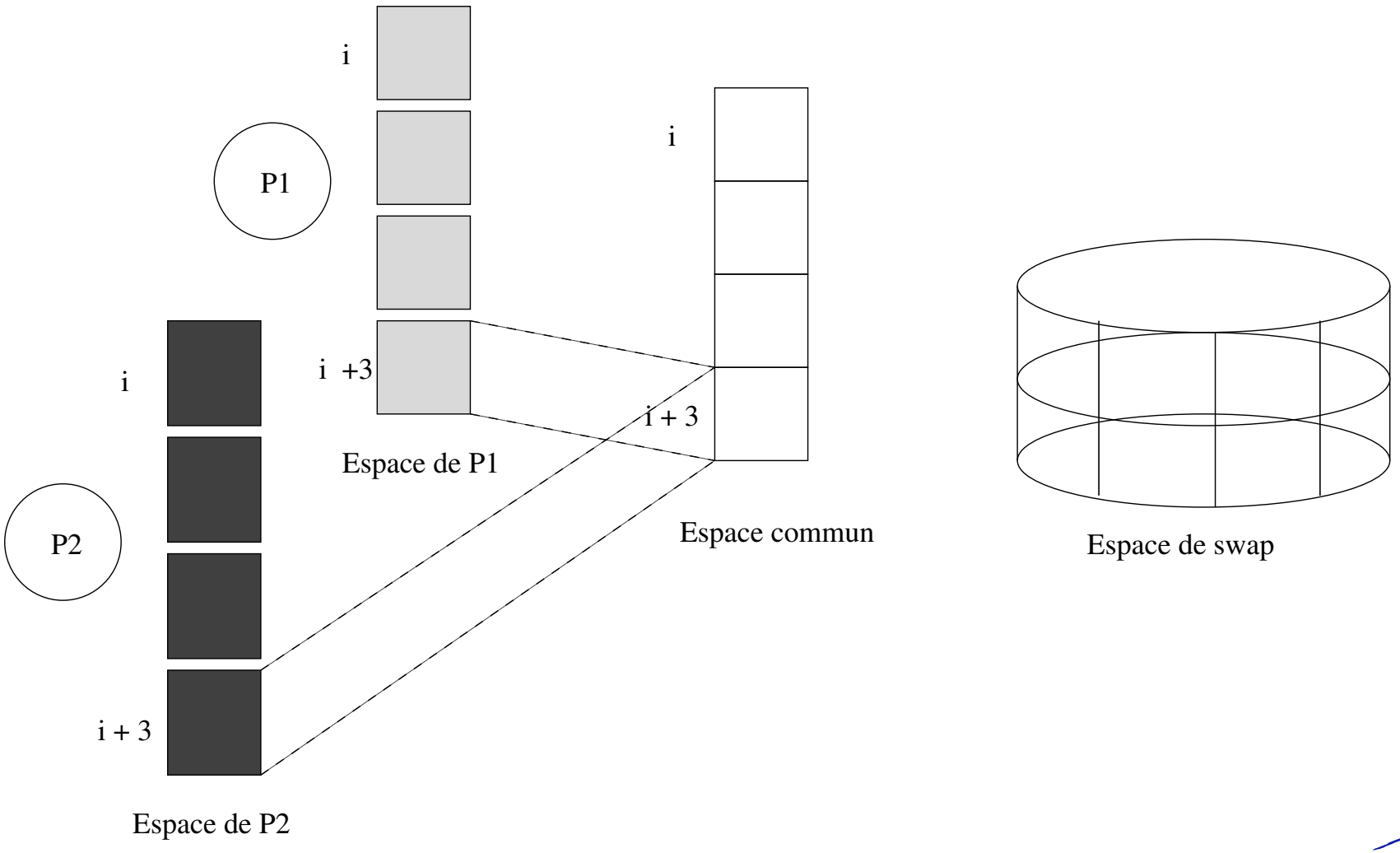
Plan

1. La pagination
   - Principe
   - La table des pages
   - Le partage de pages
2. Le va-et-vient
   - Expulsion sur disque
   - Algorithmes de remplacement de pages
   - Verrouillage
3. L'API Unix
4. Mise en œuvre dans notre noyau

Idée

- Tel qu'implanté, nos processus partagent le même espace de mémoire. Ils ne possèdent en propre que leur pile d'exécution (et encore. . .).
- Objectif : assurer à chaque processus un espace mémoire privé de taille fixée. Pour tous les processus, cet espace sera vu *dans la même plage d'adresses*.
- L'adresse de début de cet espace et sa taille en nombre de pages seront connues de tout processus.
- Analogue à l'implantation d'une mémoire virtuelle avec pagination mais sans translation d'adresse.

Partage de la zone commune



Solution 1

À chaque changement de processus élu (commutation, suspension, mort) :

- écrire toutes les pages virtuelles de l'ancien processus élu sur disque ;
- charger toutes les pages virtuelles du nouveau processus élu.

Mais c'est inefficace !

Solution 2

À chaque changement de processus élu (commutation, suspension, mort) :

- Interdire totalement l'accès à tout l'espace commun.

Tentative d'accès ⇒ erreur de segmentation ⇒ signal.
Dans le gestionnaire de signal :

- déterminer à partir de l'adresse le numéro de page concernée ;
- écrire le contenu courant de la page sur disque (le swap) ;
- charger le contenu correspondant à l'actuel processus élu.

Nécessite de connaître :

- Pour chaque page virtuelle : à quel processus appartient-elle ?
- Pour chaque processus (TSD) : pour chaque page virtuelle, où est-elle dans le swap ?

Affinage

- La page courante en mémoire virtuelle peut déjà appartenir au processus qui y accède : inutile d'expulser/charger.
- Pour écrire la page sur disque, il faut disposer d'une page dans le swap :
  - Allocation des N pages au démarrage du processus
  - Ou attendre la première écriture
    $\rightarrow$ Nécessite de savoir pour chaque page disque : libre ou pas ?
- Inutile d'écrire une page qui n'a pas été modifiée. Comment le détecter ?

Constantes et structures de données

Constantes

- base : adresse de début de la mémoire centrale
- pagesize : taille d'une page
- taille_espace_centr

Structures de données

- `info_memcentrale` :
  table n° page virtuelle → processus propriétaire
- `info_swap` :
  table n° page swap → booléen ou processus propriétaire
- Pour chaque processus (en TSD) :
  table n° page virtuelle → n° page swap

Raffinage 0

À chaque changement d'élu

Dans `pre_commuter`, interdire l'accès à la la mémoire centrale (pour déclencher SIGSEGV) :

```
mprotect(base, taille_espace_centr, PROT_NONE);
```

Gestionnaire du signal SIGSEGV

1. Vérifier que l'adresse responsable est dans base..base+taille_espace_centr
2. numpage ← (@ - base) / pagesize
   [n° page correspondant à l'@]
3. Rendre la page lisible et modifiable (`mprotect`)
4. `vider_page` : Vider le contenu actuel de la page sur disque
5. `charger_page` : Charger le bon contenu depuis le disque
6. Reconnecter le gestionnaire du signal SIGSEGV

Raffinage 1 – `vider_page(npage)`

ancien ← ancien propriétaire de la page
(`info_memcentrale[npage].propio`)
nouveau ← processus courant (`proc_self()`)
s'il y a un ancien proprio et ancien ≠ nouveau alors
// il faut sauver la page
infopages ← table virt2swap de l'ancien proprio (en TSD)
si la page virtuelle n'a pas encore de page de swap alors
allouer une page de swap inutilisée (infos_swap)
et mettre à jour la table virt2swap du proc
finsi
pageswap ← infospage[npage].swap
écrire la zone mémoire [base+npage*pagesize .. base+(npage+1)*pagesize[
dans le swap à l'emplacement pageswap*pagesize :
lseek(swapfd, pageswap * pagesize, SEEK_SET) ;
write(swapfd, base + npage * pagesize, pagesize) ;
finsi

Raffinage 1 – `charger_page(npage)`

ancien ← ancien propriétaire de la page
(`info_memcentrale[npage].propio`)
nouveau ← processus courant (`proc_self()`)
si ancien ≠ nouveau alors
    // il faut peupler la page
    infopages ← table virt2swap du proc courant (en TSD)
    s'il n'y a pas de page de swap allouée alors
        // premier accès à cette page
        remplir la page avec des 0
        `memset(base+npage*pagesize, 0, pagesize)`
    sinon
        lire depuis le swap (lseek + read)
    finsi
    mettre à jour le proprio de la page virtuelle
finsi

Résumé

- Mémoire virtuelle vs mémoire physique
- La pagination : translation d'adresses
- Le va-et-vient (swap) : utilisation de l'espace disque, algorithmes de remplacement