Gestion de la mémoire

Michel Simatic

module CSC4508/M2

Avril 2018

Plan du document

1 Point de vue système

1.1 Introduction

- Un processus a besoin d'être présent en mémoire centrale pour s'exécuter
- Mémoire centrale divisée en deux parties :
  - L'espace réservé au système d'exploitation
  - L'espace alloué aux processus
- La Gestion mémoire concerne l'espace processus
- Capacités mémoire augmentent, mais les besoins aussi → Nécessité de plusieurs niveaux
  - Mémoire(s) rapide(s) (cache(s))
  - Mémoire centrale
  - Mémoire auxiliaire (disque)

  Principe d'inclusion pour limiter les mises à jour entre les différents niveaux

1.2 Besoins des processus

Allocation Trouver une zone mémoire disponible pour y stocker un programme

Libération quand un programme se termine, récupérer les espaces (en faisant des regroupements s'il y a eu fragmentation) qu'il occupait pour les consacrer à d'autres processus.

Protection Garantir l'intégrité de l'espace mémoire associé à chaque processus →

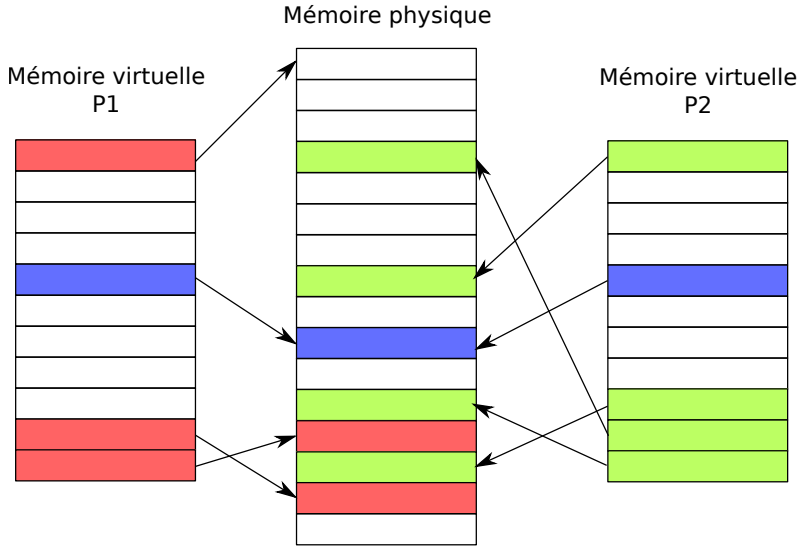
- Dispositif (matériel) de contrôle empêchant tout accès en dehors de l'espace attribué
- Toute détection d'une violation mémoire est récupérée par le système (déroutement)

Adressage Traduction des adresses logiques en adresses physiques : fonction appelée topographie ou *mapping*

- À la traduction/édition des liens
- Au chargement initial
- À l'exécution

1.3 Pagination

1.3.1 Généralités



- Objectif : offrir un espace adressable plus grand que la mémoire physique
- Principe
  - Espace adressable de chaque programme découpé en *pages*
  - Mémoire physique divisée en *cadres de pages*
- À l'exécution, un processus possède
  - Peu de pages en mémoire centrale (pages actives)
  - Le reste est

- Inexistant en mémoire (pages inactives jamais écrites)
- En mémoire secondaire (pages inactives qui ont déjà été écrites)

- Le dispositif de pagination
  - Effectue la correspondance d'adresse
  - Charge les pages nécessaires (déroutement par défaut de page)
  - (Éventuellement) décharge des pages actives en mémoire secondaire
- Donc, le programme n'est plus présent intégralement en mémoire centrale et il n'y a plus de continuité physique

1.3.2 Adresse logique

- Espace adressable divisé à partir des bits de poids forts de l'adresse

Adresse logique sur k bits	
Numéro de Page	Déplacement dans la page
(p bits)	($(k-p)$ bits)
N_p	d

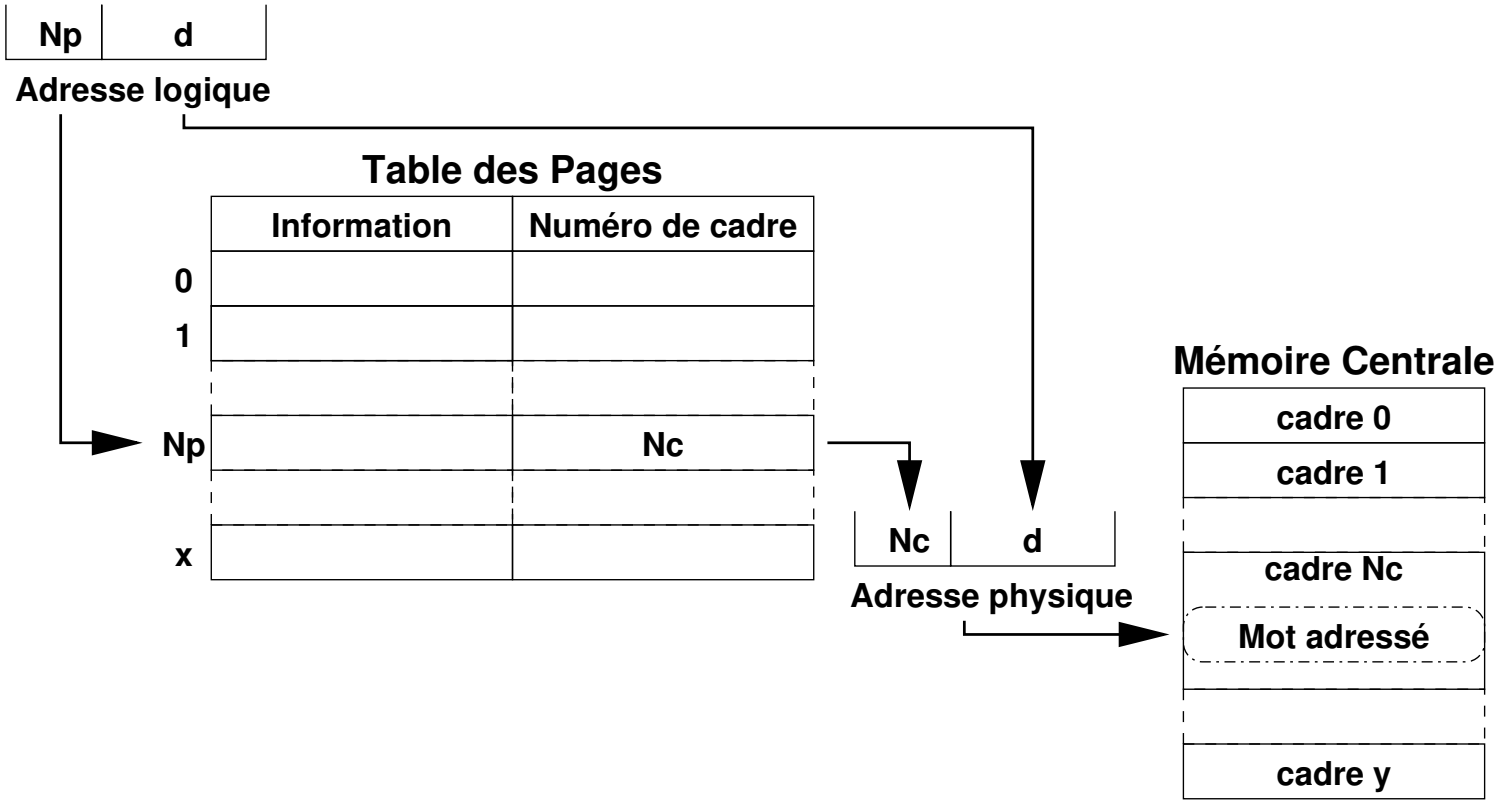
  $\rightarrow 2^p$ pages et une page contient 2^{k-p} octets

- Taille d'une page
  - À l'origine : 512 octets ou 1 Ko
  - Aujourd'hui : 2 Ko, 4 Ko (k-p = 12 bits, donc p = 20 bits) et plus

  Choix = compromis entre divers critères opposés
  - Dernière page à moitié gaspillée
  - Temps de transfert d'une page faible par rapport au temps d'accès total
  - Mémoire de petite capacité : petites pages

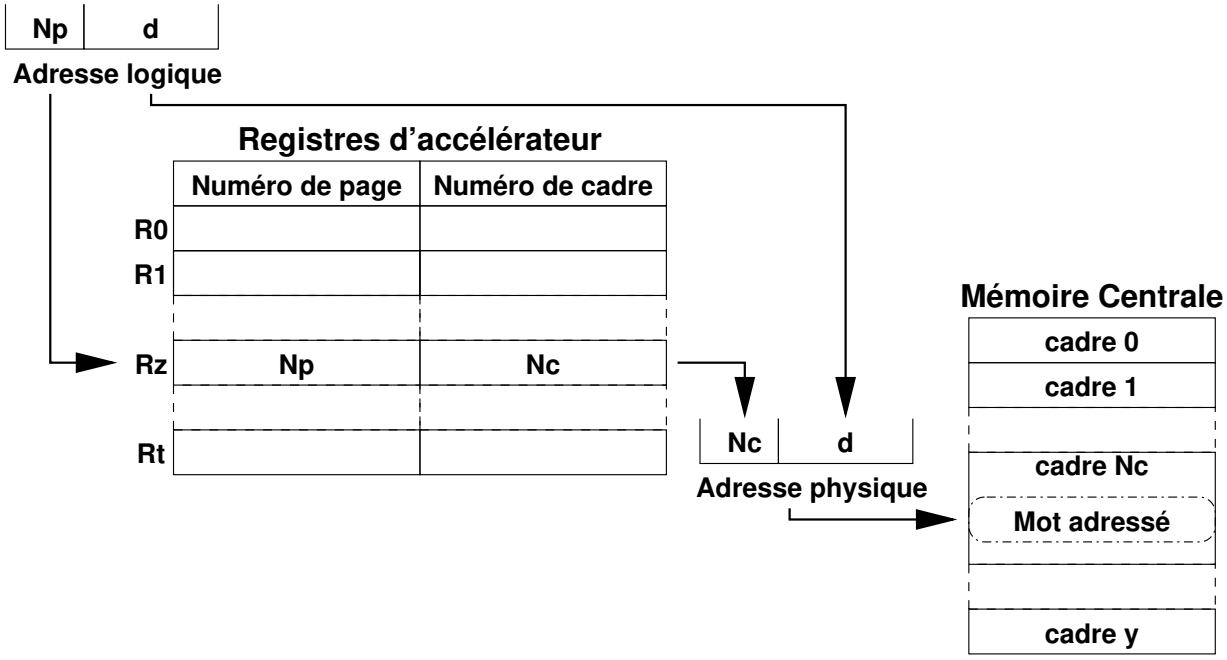
1.3.3 Table des pages

- La correspondance entre adresse logique et adresse physique se fait avec une table des pages contenant
  - Numéro de cadre de page
  - Bits d'information (présence, accès, référence, écriture, date de chargement...)



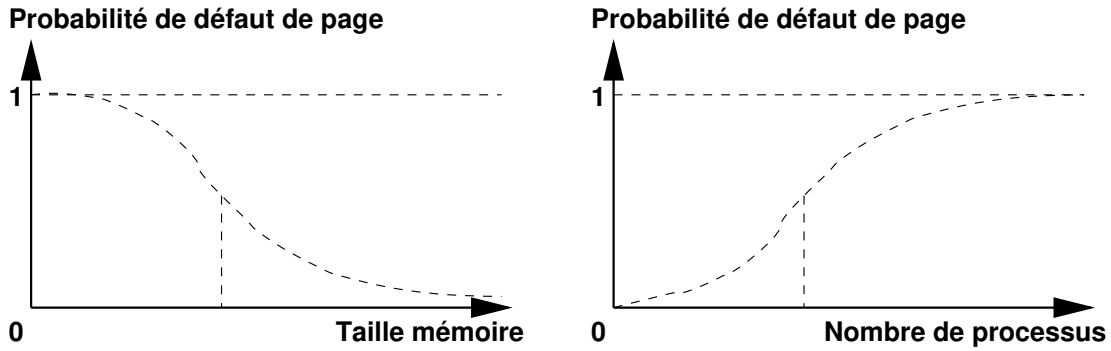
1.3.4 Accélérateurs (caches d'adresses)

- Problème : tout accès à une information nécessite deux accès mémoire
- Solution : utiliser des mémoires associatives (registres d'accès rapide)
- Principe
  - On a un certain nombre de registres à disposition
  - Numéro de page logique N_p comparé au contenu de chaque registre
  - Égalité trouvée $\rightarrow$ en sortie numéro N_c du cadre correspondant
  - Sinon utilisation de la table des pages



1.3.5 Écroulement

- Effet secondaire de la pagination : écroulement (*thrashing*) dû à un taux de défauts de page trop important
- Deux facteurs principaux : taille mémoire et degré de multiprogrammation (nombre de processus)



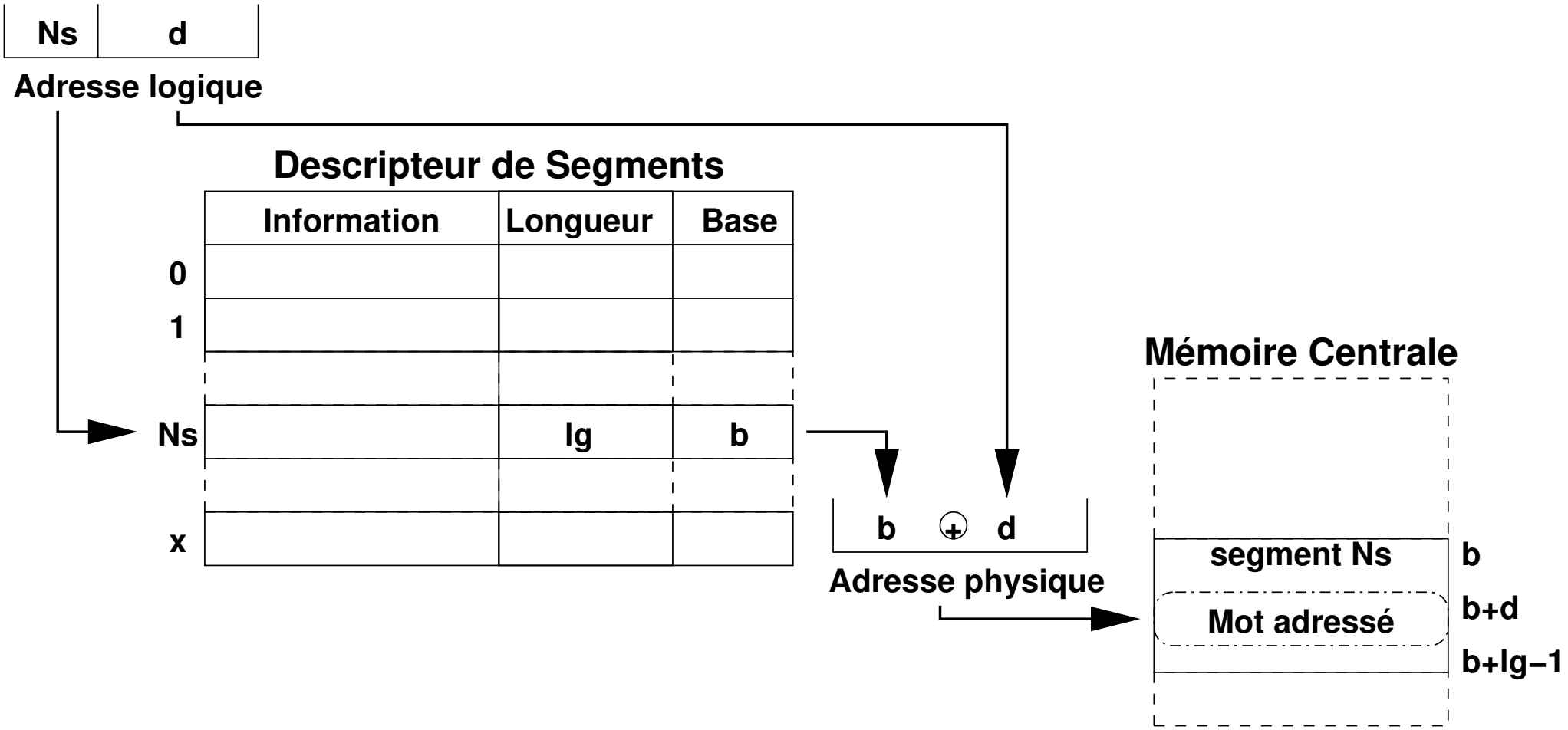
- Solutions
  - Augmentation de la capacité mémoire
  - Ordre des modules utilisés
  - Prise en compte lors du développement
  - Régulation de la charge (*load levelling*) : limitation du nombre de processus
  - Définition d'un espace vital (*working set*) nécessaire
  - Gestion du taux de défauts de page : moduler le nombre de pages allouées

1.4 Segmentation

1.4.1 Généralités

- Objectif : refléter en mémoire centrale la structure des programmes (code, données, pile)
- L'utilisateur définit des segments (entités logiques)
- Programme = ensemble de segments ayant une taille et des attributs propres (lecture seule, écriture, partage)
- Pour disposer d'une segmentation, il faut un dispositif chargé de
  - Effectuer la correspondance entre une adresse logique et l'adresse physique
  - Assurer la protection
  - Charger les segments en mémoire
- Correspondance réalisée par une table des descripteurs de segments contenant
  - Adresse de base d'implantation
  - Longueur du segment
  - Informations de contrôle d'accès
- Protection assurée par un double contrôle : si le déplacement est négatif ou supérieur à la longueur du segment, alors il y a violation mémoire et déroutement

1.4.2 Descripteur de segments



- Doublement des accès mémoire → Mêmes dispositifs accélérateurs (caches d'adresses) que pour la pagination.

1.5 Pagination versus Segmentation

- Mise en oeuvre analogue, mais concepts différents
  - Pagination = division physique *vs* Segmentation = division logique
  - Pagination transparente à l'utilisateur *vs* Segmentation déterminée par l'utilisateur
  - Taille des pages fixes *vs* Taille des segments variable (avec contrôle de non-dépassement)
- On utilise donc pagination *et* segmentation !

Adresse logique sur k bits		
N° de Segment	**Déplacement dans le segment**	
N° de Segment	**N° de Page**	**Déplacement dans la page**
Ns	**Np**	**d**

  - Avantage : mémoire virtuelle (segments chargés en partie) et organisation logique
  - Inconvénient : Surcoût (limité par l'utilisation de caches d'adresses dans le processeur)

1.6 Algorithmes pour la gestion des pages

1.6.1 Chargement

- Un programme démarre. On décide des pages à charger :
  - À la demande c'est-à-dire en cas de besoin
  - Par anticipation

    Exemple : un programme travaille en général (principe de la séquentialité des programmes) sur des pages contiguës en y accédant de manière croissante : il accède à une page, puis à la suivante (dans la plage d'adresse), puis à la suivante.... De ce fait, quand un programme accède à une page, certains systèmes (Exemple : Sun) anticipent le fait qu'il accédera bientôt à la suivante : il la précharge.

1.6.2 Remplacement (déchargement)

- Pour les systèmes paginés lorsqu'il n'y a plus de page libre

  FIFO (*First In First Out*) Remplacer la page la plus ancienne

  LRU (*Least Recently Used*) Remplacer la page la moins récemment utilisée

  LFU (*Least Frequently Used*) Remplacer la page la moins fréquemment utilisée

  Algorithme de Belady dit « optimal » (OPT ou MIN) Remplacer la page qui ne sera plus utilisée s'il en existe une et sinon celle qui le sera le plus tardivement

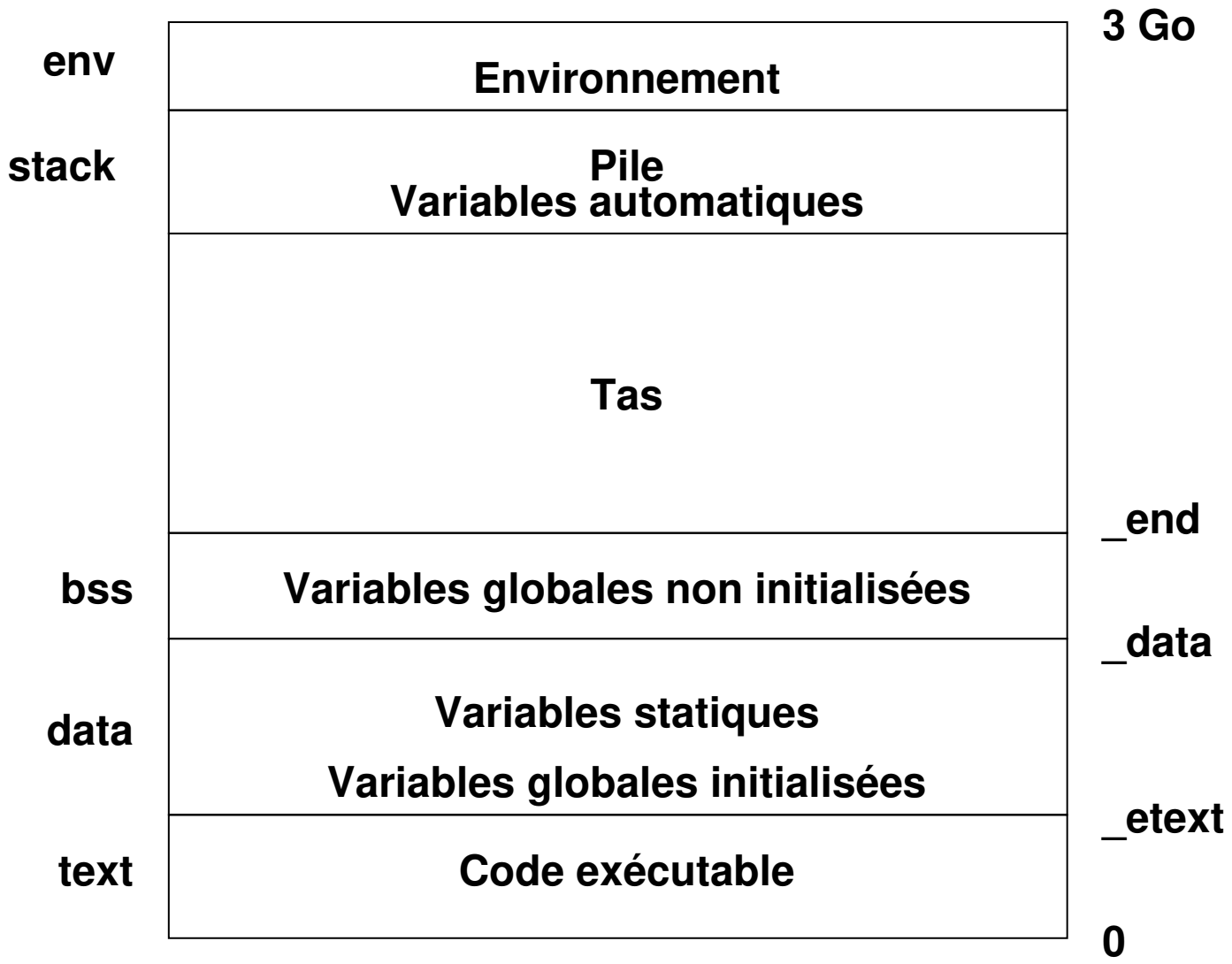
  Random Remplacer une page choisie au hasard

  Algorithme « seconde chance » Compromis tant au niveau surcoûts qu'en termes de transferts

  - Variante de l'algorithme LRU couplée au principe FIFO
  - Un bit U associé à chaque page et mis à un lors de chaque accès
  - L'algorithme parcourt la liste des pages dans l'ordre FIFO et met à 0 tout bit U qui vaut 1 ou retire la première page dont le bit U est à 0
  - C'est l'algorithme utilisé par Linux

2 Point de vue processus

2.1 Espace d'adressage d'un processus



2.2 Observation et contrôle de l'utilisation de la mémoire

2.2.1 Observation de l'utilisation des pages

- `ps -u` et `top` donnent des indications
- `int getrusage (int who, struct rusage *usage)`

  permet de connaître
  - `ru_minflt` : nombre de défauts de page mineurs, c'est-à-dire pages inactives jamais écrites finalement chargées en mémoire
  - `ru_majflt` : nombre de défauts de page majeurs, c'est-à-dire pages inactives déjà écrites et déchargées (swappées) sur disque
- `/usr/bin/time` *`nomExécutable`*

  affiche toutes ces informations pour le processus `nomExécutable` lancé en mode commande

2.2.2 Observation du type d'accès à la mémoire

- L'outil `valgrind` (logiciel libre, `http://valgrind.org`) permet d'observer tous les types d'accès à la mémoire effectués par un programme

```
valgrind --tool=cachegrind nomProgramme
```

2.2.3 Être attentif à l'alignement des structures

- Par défaut, le compilateur aligne sur des frontières de 4 octets (2 octets pour les `short`).
- Le compilateur peut donc générer des octets inutilisés au sein d'une structure. Pour récupérer cet espace perdu
  - Le mieux est de réordonnancer les feuilles des structures
  - On peut aussi compacter toutes les structures d'un source

    ```
    gcc -fpack-struct
    ```
  - Ou bien compacter juste une structure donnée

    ```
    typedef struct {
      ...
    } __attribute__((packed)) uneStructure;
    ```
- Cet alignement peut être insuffisant (besoin d'aligner sur 16 octets, par exemple). Pour forcer l'alignement :

- `typVariable nomVariable __attribute__((aligned (nombre)));`
- Utilisation de `posix_memaligned` pour les variables de type pointeur

2.2.4 Contrôle des défauts de page

- Lutte contre les défauts de page *mineurs*
  - Il suffit d'écrire un octet dans la page
- Lutte contre les défauts de page *majeurs*
  - `int mlock(const void *addr, size_t len)`

    permet de verrouiller en mémoire centrale `len` octets (à partir de `addr`) de la mémoire virtuelle du processus
  - `int munlock(const void *addr, size_t len)`

    déverrouille
  - `int mlockall(int flags)`

    permet de tout verrouiller en mémoire centrale. `flags` vaut une combinaison de
    - `MCL_CURRENT` : Verrouiller toutes les pages correspondant actuellement à l'espace d'adressage du processus
    - `MCL_FUTURE` : Verrouiller toutes les pages qui *seront* dans l'espace d'adressage
  - `int munlockall(void)`

    déverrouille tout

2.3 Allocation/Désallocation dynamique de mémoire

2.3.1 Allocation/Désallocation mémoire standard

- `void *malloc (size_t size)`

  renvoie un pointeur sur une zone de `size` octets

- `void *realloc (void *ptr, size_t size)`

  change la taille d'une zone mémoire réservée précédemment par `malloc(3)`

- `void *calloc (size_t nmemb, sizeoi_t size)`

  Même rôle que `malloc`, mais avec initialisation de la mémoire à 0

- `int posix_memalign(void **memptr, size_t alignment, size_t size)`

  Même rôle que `malloc`, l'adresse de la zone mémoire renvoyée étant un multiple de `alignment`

  `_aligned_malloc` est la fonction correspondante sur compilateur Microsoft ou Intel

- `void free (void *ptr)`

  Libération de zone

- `void _aligned_free (void *ptr)`

  Libération de zone mémoire allouée par `_aligned_malloc`

- `int mallopt (int parametre, int valeur)`

  Contrôle de paramètres de fonctionnement des fonctions précédentes

2.3.2 Digression : algorithmes pour malloc

- Dans la suite, on appelle « Trou », une zone libre de taille x_i (supérieure à un minimum) et d'adresse a_i
- Meilleur choix (*best fit*) : trous dans l'ordre croissant des tailles $(x_1 < x_2 < \cdots < x_n)$ et choix du plus petit trou de taille suffisante
- Plus mauvais choix (*worst fit*) : trous dans l'ordre décroissant des tailles $(x_1 > x_2 > \cdots > x_n)$ et allocation de tout ou partie du premier trou
- Premier trouvé (*first fit*) : trous dans l'ordre croissant des adresses $(a_1 < a_2 < \cdots < a_n)$ et choix du premier trou de taille suffisante
- Frères siamois (*Buddy system*) basé sur l'allocation de zones ayant une taille multiple d'une puissance de 2
  - Principe : si demande de taille T, recherche de la puissance i telle que $2^{i-1} < T \leq 2^i$, puis allocation d'un trou de taille 2^i
  - Algorithme récursif : si liste 2^i vide, recherche d'un trou de taille 2^{i+1}

2.3.3 Digression : algorithmes pour free

- Fragmentation : demande refusée car plus de zone de taille suffisante mais somme des tailles des zones libres supérieure à la taille demandée
- Solution : retassement ou compactage : regrouper les zones libres pour en créer une la plus grande possible
- Objectif : reconstruire la liste des zones libres en cherchant à reconstruire le plus grand trou possible
- Plusieurs cas à envisager
  - Zone libérée entre 2 zones occupées
  - Zone libérée entre une zone occupée et une zone libre
  - Zone libérée entre 2 zones libres
- Un algorithme performant au niveau de l'allocation peut s'avérer être plus complexe pour la libération
- Algorithme des frères siamois : la libération d'une zone (un *buddy*) est récursive

2.3.4 Désallocation automatique avec alloca

- `void *alloca (size_t size)`

  alloue `size` octets dans l'espace de pile de l'appelant

- Donc, quand on retourne de la fonction qui a appelé `alloca`, l'espace alloué est automatiquement libéré

- Problème : On risque de déborder de la pile... sans aucun avertissement du système

2.3.5 Mécanisme d'allocation/désallocation dédié

- Principe
  - Au démarrage du programme, on construit une liste chaînée de blocs de N octets
  - Quand on a besoin d'allouer un bloc de N octets, on récupère le premier élément de cette liste
  - Quand on veut désallouer un bloc de N octets, on remet ce bloc en tête de liste
- Avantage : Vitesse d'allocation/désallocation
- Inconvénient : Les blocs alloués/désalloués ont tous la même taille

2.3.6 Allocation/Désallocation mémoire au niveau système

- `void *sbrk(ptrdiff_t incrément)`

  incrémente l'espace de données du programme de `incrément` octets

- `void * mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset)`

  Permet de mapper un fichier en mémoire. En plus des options qui seront étudiées dans le chapitre « Entrées/sorties », `flags` peut prendre la valeur `MAP_ANON` (ou `MAP_ANONYMOUS`) pour indiquer qu'on ne souhaite pas réellement travailler sur un fichier, mais sur une zone mémoire vierge emplie de zéros

2.4 Déverminage des accès mémoire

2.4.1 Détecter statiquement les accès erronés à des zones allouées dynamiquement

- Objectif : quand un programme accède à une zone allouée dynamiquement, vérifier, au moment de la compilation, que
  - Il n'accède pas en dehors de la zone (ou que la zone n'est pas libérée)
  - Il ne lit pas un octet jamais initialisé
  - ...
- Outils
  - `insure++` (société *Parasoft*) : propose un outil d'analyse des sources
  - `splint` (logiciel libre, `http://www.splint.org/`) : outre des contrôles « paranoïaques » sur le source, vérifie certains problèmes d'accès mémoire
- Limite = Les vérifications ne peuvent dépasser le cadre d'un source

2.4.2 Détecter dynamiquement les accès erronés à des zones allouées dynamiquement

- Objectif : quand un programme accède à une zone allouée dynamiquement, vérifier, à l'exécution, que
  - Il n'accède pas en dehors de la zone (ou que la zone n'est pas libérée)
  - ...
- Outils
  - `insure++` (société *Paradox*) : au moment de la compilation, cet outil ajoute aux sources (`.c`) des instructions de contrôle au niveau des différentes instructions d'affectation
  - `Rational Purify` (société *IBM*) : au moment du link, cet outil ajoute aux objets (`.o`) des instructions de contrôle au niveau des différentes instructions (assembleur) d'affectation
  - `valgrind` (logiciel libre, `http://valgrind.org`) : l'exécutable est lu par un simulateur de processeur x86 qui se charge de détecter toutes les anomalies d'accès
- Limite = On n'est jamais sûr d'être passé par toutes les branches

2.4.3 Visualiser quand une zone mémoire est accédée/modifée

- *Watchpoint* sous *gdb*
  - *gdb* offre la possibilité de mettre en place des *watchpoint* sur des zones mémoire, en lecture ou en lecture/écriture.
  - Si une telle zone mémoire est accédée, *gdb* stoppe le programme.
- Contrôle de l'accès à la mémoire
  - `int mprotect(const void *addr, size_t *len, int prot)`

    contrôle les autorisations d'accès (lecture et/ou écriture) à une portion de la mémoire. Si un accès interdit se produit, le programme reçoit `SIGSEGV`
  - Utilisation : toute situation où une donnée est modifiée et le contexte ne permet pas d'utiliser des outils de type *valgrind* ou *gdb*

Bibliographie du chapitre