



Cours de système

La mémoire

Sébastien Paumier



Types de mémoires

- plusieurs types de mémoires
- le coût augmente avec la vitesse d'accès, et la capacité de stockage diminue
 - registres du processeur
 - mémoire cache
 - mémoire centrale
 - disques



Les registres

- en dur dans le processeur
- modifiables par programme, mais pas tous sans privilège
- certains registres ne peuvent être modifiés en mode utilisateur (comme celui qui indique qu'on est dans ce mode)



Le cache

- petite mémoire très rapide servant de buffer entre le processeur et un matériel (mémoire centrale, disques, ...)
- exemple: travailler avec des variables locales stockées en cache est beaucoup plus rapide que de faire des allers-retours dans la mémoire centrale



La mémoire centrale

- grand tableau d'octets
- vue comme telle par le processeur
- composant central d'un ordinateur, car c'est par elle que le CPU communique avec le matériel
- au système de protéger la mémoire pour éviter que des programmes fassent n'importe quoi



Allocation contiguë

- les stratégies d'allocation ont beaucoup évolué avec le temps
- technique basique: allocation contiguë
 - on attribue à chaque processus une zone de la mémoire physique
- sur les machines primitives, aucune protection:
 - un programme pouvait écrire n'importe où, y compris sur son code ou celui du système!!



Protection par barrière

- le degré 0 de la protection: empêcher de toucher à la mémoire système
- idée:
 - mettre le système au début de la mémoire
 - utiliser un registre barrière pour vérifier la légalité des accès mémoire des programmes

0

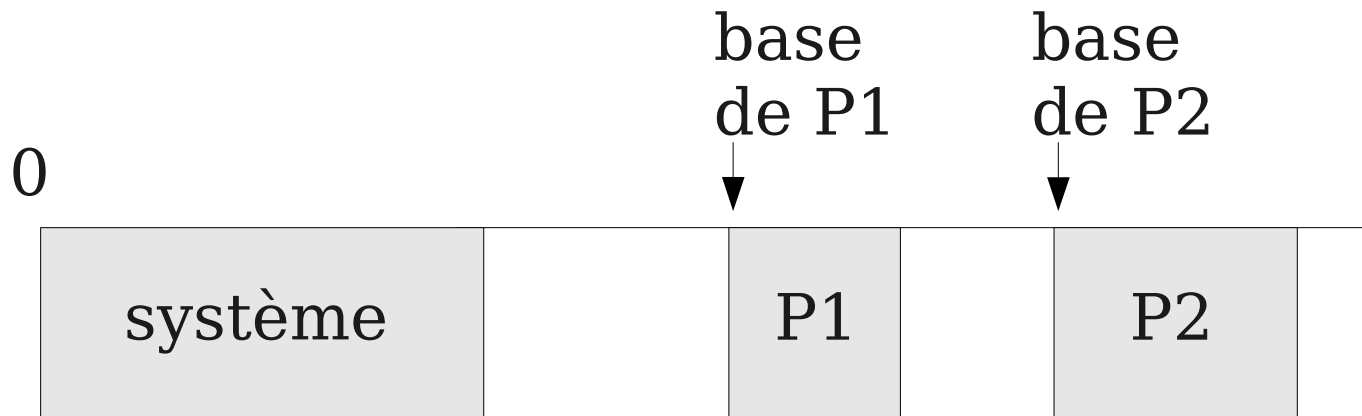
barrière





Protection par barrière

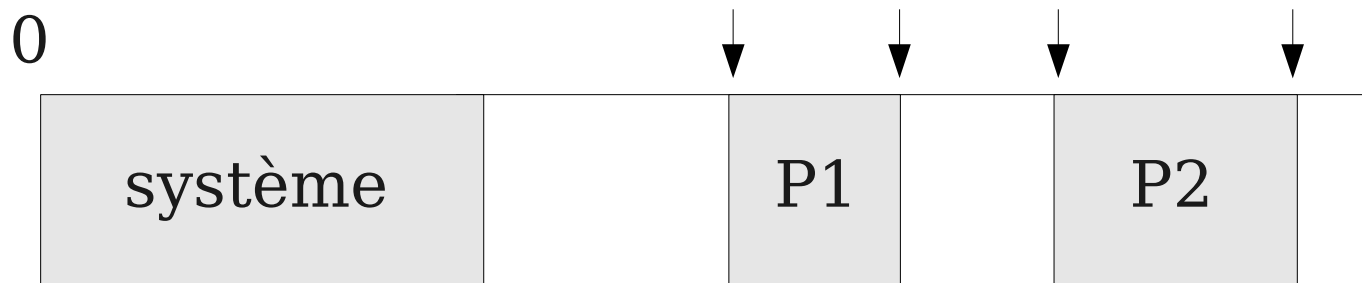
- inconvénients:
 - ne protège que le système
 - pas les processus entre eux
 - interdit certaines adresses aux processus
- 2e idée: utiliser un registre *base* qui va s'ajouter aux adresses commençant à 0





Protection par barrière

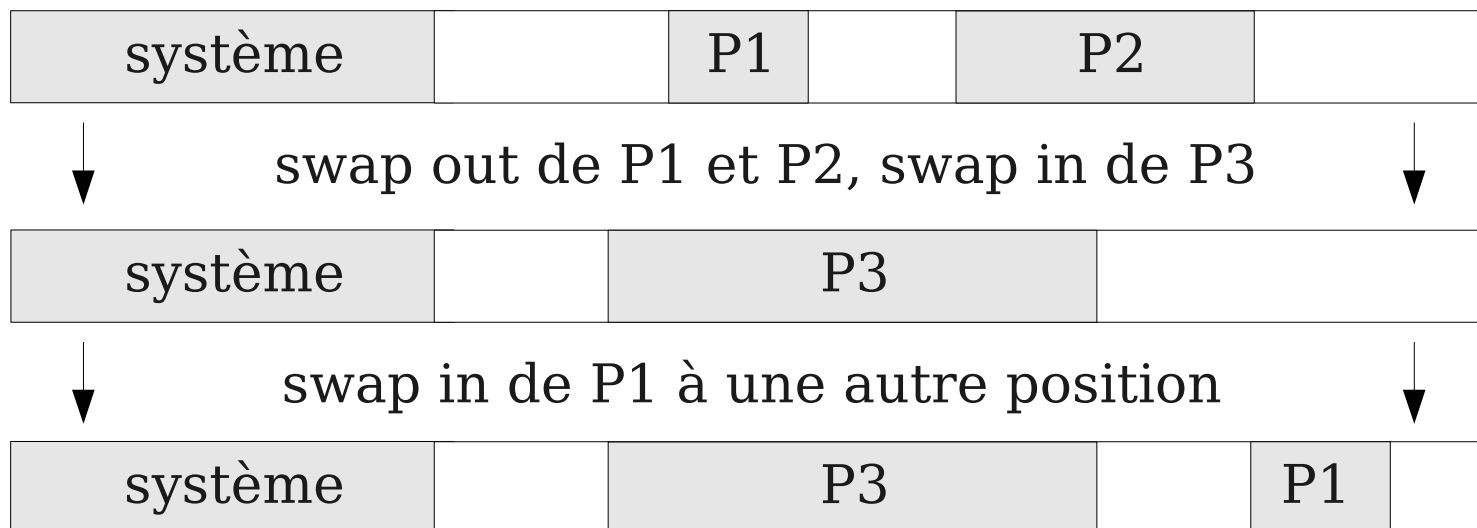
- les processus ne sont toujours pas protégés entre eux
- pour ça, il faut un autre registre qui donne, en plus de la base, la *limite* de la zone mémoire
- grâce au mode protégé, on peut alors faire respecter les zones par les processus





Le swap

- problème: soit on n'utilise pas plus que la mémoire disponible, soit on doit pouvoir enlever et remettre des processus en mémoire
- principe du swap





Le swap

- inconvénient: ça coûte très cher en E/S
- possibilité d'amélioration en ne swappant pas tout, mais juste le strict nécessaire
- pour ça, il faut connaître à tout instant la taille de tous les processus:
 - possible si tous les changements de taille se font via un appel système (**sbrk**)
- mais chaque processus doit garder longtemps le processeur pour que ça puisse valoir le coup



Swap et E/S

- si on swappe un processus en attente d'une E/S, les données seront peut-être lues ou écrites dans une zone utilisée par un autre processus
- solution 1: pas de swap pendant une E/S
- solution 2: utiliser des buffers du noyau en recopiant ensuite au bon endroit après le swap



Allocation non contiguë

- même avec le swap, on doit avoir les processus entiers en mémoire
 - peu de processus peuvent cohabiter
 - impossible d'avoir des processus plus grands que la mémoire physique
 - l'ordonnancement devient très critique!!
- remarque: un processus n'a jamais besoin de toute sa mémoire tout le temps
- idée: ne lui donner que ce dont il a besoin à un instant t



Allocation non contiguë

- pour ça, on découpe la mémoire en *pages*
- chaque adresse est composée d'un couple (n° de page, adresse dans la page)
 - translation d'adresses par la MMU
- on peut donc associer à un processus un ensemble non contigu de pages de taille fixe qui seront plus faciles à allouer et à swapper
- le système n'a plus qu'à tenir à jour une table des pages



Table des pages

- il faut savoir quelles pages sont occupées et à quels processus elles appartiennent
- mécanisme de protection des pages qui permet par exemple de partager une page de code entre plusieurs processus (il suffit de la protéger en écriture)
- le rôle du système est de fournir les pages quand les processus en ont besoin



Mémoire virtuelle

- rien n'empêche de manipuler plus de pages qu'il ne peut en tenir en mémoire
 - principe de la mémoire virtuelle
- pagination à la demande:
 - si un processus demande une adresse dans une page absente de la mémoire, le processeur génère une page fault
 - le système charge la page demandée et redonne la main au processus
 - nécessite peut-être de supprimer une autre page



Mémoire virtuelle

- comme il y a des accès disque, c'est lent
- il faut minimiser les fautes de pages pour éviter de vampiriser le système
- le système: choisir au mieux les pages à avoir en mémoire
- le programmeur: ne pas parcourir la mémoire n'importe comment

```
arch/x86/mm/fault.c: do_page_fault
```



Remplacement de page

- comment choisir quelle page retirer de la mémoire, pour minimiser les fautes de pages ?
- tests des différents algorithmes pour les données suivantes:
 - 3 pages disponibles en mémoire
 - pages demandées dans l'ordre suivant :
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



Remplacement idéal

- retirer la page qui sera inutile le plus longtemps

7 : 7xx	0 : 203	=9 fautes de pages problème: il faudrait pouvoir prédire l'avenir...
0 : 70x	3 : 203	
1 : 701	2 : 203	
2 : 201	1 : 201	
0 : 201	2 : 201	
3 : 203	0 : 201	
0 : 203	1 : 201	
4 : 243	7 : 701	
2 : 243	0 : 701	
3 : 243	1 : 701	



Remplacement FIFO

- retirer la page la plus ancienne

7 : 7xx	0 : 023
0 : 70x	3 : 023
1 : 701	2 : 023
2 : 201	1 : 013
0 : 201	2 : 012
3 : 231	0 : 012
0 : 230	1 : 012
4 : 430	7 : 712
2 : 420	0 : 702
3 : 423	1 : 701

=15 fautes de pages

anomalie de Belady:
cet algo peut produire
plus de fautes de pages
quand il y a plus de
pages disponibles !!



Remplacement LRU

- Last Recently Used: garder les N-1 pages auxquelles on a accédé en dernier

7 : 7xx	0 : 032	=12 fautes de pages comme FIFO, nécessite de représenter les dates d'accès aux pages
0 : 70x	3 : 032	
1 : 701	2 : 032	
2 : 201	1 : 132	
0 : 201	2 : 132	
3 : 203	0 : 102	
0 : 203	1 : 102	
4 : 403	7 : 107	
2 : 402	0 : 107	
3 : 432	1 : 107	



Remplacement LRU

- plusieurs façons de gérer ces dates
- compteurs:
 - tenir à jour un compteur de temps par page
 - on doit aussi le swapper quand la page change
- pile:
 - mettre la page utilisée en haut de pile
 - le bas de pile désigne celle qui a été utilisée le moins récemment



Remplacement LRU

- masques:
 - chaque page a un octet
 - à chaque accès, on met à 1 le bit de poids fort
 - régulièrement, le système décale tous les octets d'un bit vers la droite
 - l'octet le plus petit correspond ainsi à la page à supprimer (10010111 est plus récent que 00111011)
 - à numéro égal, on prend la première page rencontrée



Algo de la 2ème chance

- chaque page possède un bit qu'on met à 1 quand on y accède
- quand on doit supprimer une page, on regarde ce bit:
 - s'il vaut 0, on supprime la page
 - s'il vaut 1, on met le bit à 0 et on cherche une autre victime



Le dirty bit

- si la page a déjà une copie identique dans le swap, on n'aura pas besoin de la sauver si on la retire de la mémoire
- le dirty bit permet de savoir si une telle économie est possible
- utilisé comme heuristique complémentaire par les algorithmes de remplacement de pages

```
mm/swap.c: mark_page_accessed  
mm/page-writeback.c
```



Allocation de pages

- bien choisir sa stratégie:
 - ne pas donner trop de pages à un processus qui les sous-utilise
 - ne pas en donner trop peu pour minimiser les fautes de pages et le swap
- principe du Copy-On-Write:
 - éviter de dupliquer des pages
 - pratique quand on enchaîne un fork et un exec
 - précurseur=vfork (deprecated)



DMA

- Direct Memory Access
- principe: ne pas bloquer le processeur pendant un transfert de données depuis un périphérique
- le CPU initie le transfert
- celui-ci a lieu indépendamment sur le bus DMA
- le CPU est prévenu par interruption quand c'est terminé



Pages empoisonnées

- le matériel peut détecter des erreurs physiques dans la mémoire
- les pages sont marquées "empoisonnées"
 - une page empoisonnée qui a une copie sur disque peut être éliminée
 - si pas de copie, l'application doit être tuée
 - une page empoisonnée ne sera plus jamais utilisée

`mm/memory-failure.c`



Mémoire du noyau

- le noyau doit avoir de la mémoire accessible immédiatement, sans swap
- principe du slab:
 - pré-réserver des buffers de certaines tailles adaptées aux objets courants du noyau (locks, inodes, etc)
 - évite la fragmentation
 - **kmalloc/kfree**: trouver vite des slabs disponibles adaptés aux demandes
 - zones mémoire contiguës limitées à 128Ko (cf. **vmalloc/vfree** pour plus)

mm/slab.c



Fichiers mappés

- `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`
- permet de mapper en mémoire le contenu d'un fichier
- on accédera au contenu comme à un tableau
- c'est le système qui gère automatiquement les accès disques et les changements de page



mmap

- **addr**: adresse de base à laquelle on veut mapper le fichier; si **NULL**, le noyau choisit
- **length**: taille de la zone à projeter
- **fd**: descripteur du fichier
- **offset**: position de départ dans le fichier



mmap

- **prot**: mode de protection de la zone (OU binaire)
 - PROT_READ: accessible en lecture
 - PROT_WRITE: accessible en écriture
 - PROT_EXEC: accessible en exécution
 - PROT_NONE: inaccessible



mmap

- **flags:**

- MAP_SHARED: les modifications sont visibles par tous les processus manipulant la zone et sont répercutées dans le fichier sous-jacent (de façon asynchrone)
- MAP_PRIVATE: chaque processus à sa copie privée grâce au copy-on-write; les modifications ne sont pas répercutées sur le fichier

mmap.cpp



munmap

- `int munmap(void *addr, size_t length);`
- unmappe une zone, en sauvegardant les modifications sur le fichier
- possibilité de demander la synchronisation avec le fichier grâce à

```
int msync(void *addr, size_t  
length, int flags);
```



madvise

- `int madvise(void *addr, size_t length, int advice);`
- **advice** suggère une politique au noyau:
 - `MADV_SEQUENTIAL`: les pages vont être lues dans l'ordre et oubliées aussitôt (on peut en précharger)
 - `MADV_RANDOM`: lecture aléatoire (pas besoin de précharger)
 - etc.



mprotect

- `int mprotect(const void *addr, size_t len, int prot);`
- `prot` demande au noyau de protéger la zone contre la lecture (`PROT_READ`), l'écriture (`PROT_WRITE`), l'exécution (`PROT_EXEC`), ou tout à la fois (`PROT_NONE`)
- une violation entraînera un `SIGSEGV`

`mprotect.cpp`



mlock

- `int mlock(const void *addr, size_t len);`
- `int munlock(const void *addr, size_t len);`
- permet de demander au noyau de ne pas swapper la zone
- utile pour:
 - le temps réel, pour éviter les ralentissements imprévus dûs au swap
 - la sécurité, pour éviter que des données sensibles (mots de passe) se retrouvent dans la zone de swap sur disque



Map anonyme

- avec le flag `MAP_ANONYMOUS`, on peut demander à `mmap` une zone mémoire qui ne soit pas basée sur un fichier, et sans modifier la taille du processus avec `sbrk`
- grâce à `MAP_PRIVATE`, on a ainsi une zone qui se duplique correctement à travers `fork`
- utile pour implémenter `malloc`



My malloc

- la variable d'environnement `LD_PRELOAD` permet de précharger en priorité des bibliothèques
- possibilité de hook sur toutes les fonctions de bibliothèques, y compris la `libc`
- on alloue de la mémoire au chargement de la bibliothèque et on affiche des statistiques à sa libération



My malloc

- exécution comparée avec valgrind:

```
$>LD_PRELOAD=./mymalloc.so ps
```

mymalloc.cpp

```
...
```

```
132 mallocs (3 reallocs, 9 callocs), 197 frees
```

```
43392 bytes allocated
```

```
441 bytes unfreed
```

```
$>valgrind ps
```

```
...
```

```
==4375== HEAP SUMMARY:
```

```
==4375==      in use at exit: 441 bytes in 18 blocks
```

```
==4375==    total heap usage: 144 allocs, 126 frees, 43,392
```

```
bytes allocated
```

```
==4375==
```

```
...
```