Architecture d’un système d’exploitation: Allocation mémoire

Basé sur le cours de Marc Tchiboukdjian et les travaux de recherche de Sébastien Valat

Bibliographie

- La programmation sous UNIX de Jean-Marie Rifflet
- Système d’exploitation d’Andrew Tanenbaum
- What Every Programmer Should Know About Memory de Ulrich Drepper
- Contribution à l'amélioration des méthodes d'optimisation de la gestion de la mémoire dans le cadre du Calcul Haute Performance de Sébastien Valat
- Framework de R&D http://mpc.hpcframework.com

Déroulement du cours

- Allocation mémoire en espace noyau
- Allocation mémoire en espace utilisateur
- Allocation mémoire en contexte HPC

Allocation mémoire en espace noyau

La programmation sous UNIX de Jean-Marie Rifflet

Système d'exploitation d'Andrew Tanenbaum

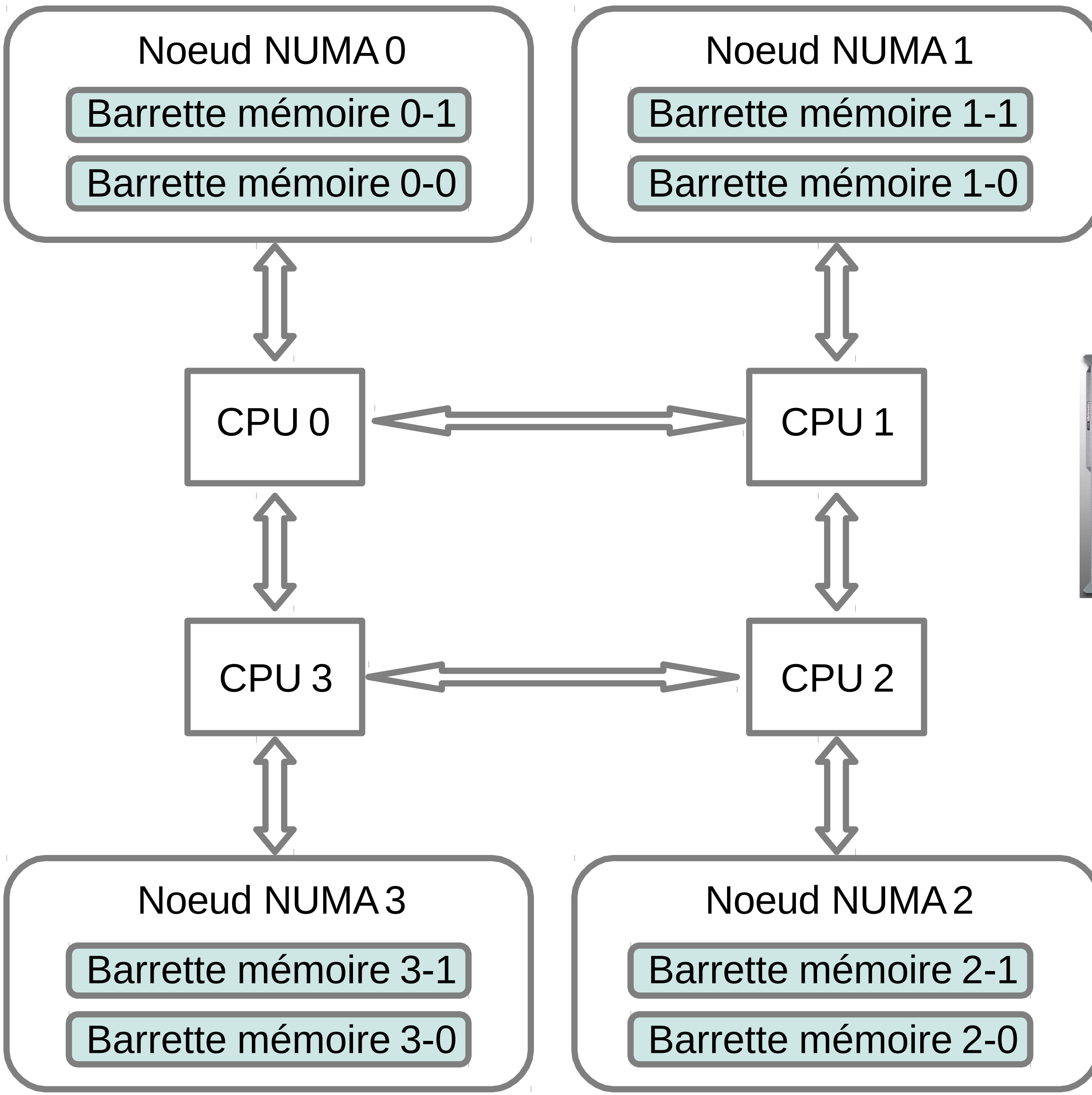
Mémoire physique

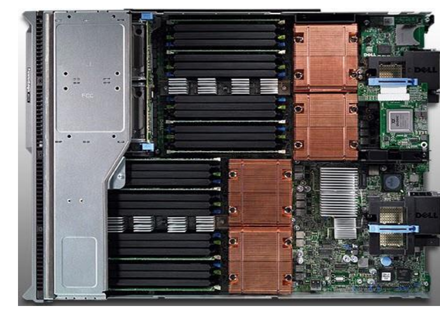
- Un tableau de cases ou cellules mémoire
  - Chaque case : un nombre défini de bits
  - Chaque case a un numéro : une adresse
  - Mot mémoire : information contenue dans une case

- RAM pour Random Access Memory

- Taille sur un système actuel : de quelques Go (téléphone portable) à 1 To (un nœud d'un super-calculateur)

Mémoire physique

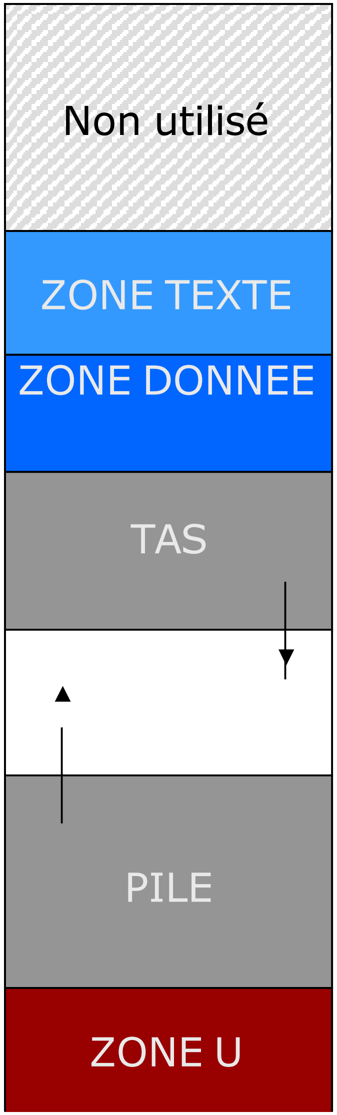




4 Nehalem EX processors

Mémoire vue par le programmeur

- Espace d'adressage logique
- Code du programme
  - Stocké dans la zone texte
- Variables allouées statiquement
  - Stockées dans la zone donnée
  - Durée de vie du programme
- Variables allouées automatiquement
  - Stockées dans la pile (stack)
  - Durée de vie liée à la fonction ou à un bloc
- Variables allouées manuellement
  - Stockées dans le tas (heap)
  - En C avec malloc/free
  - Durée de vie gérée par le programmeur

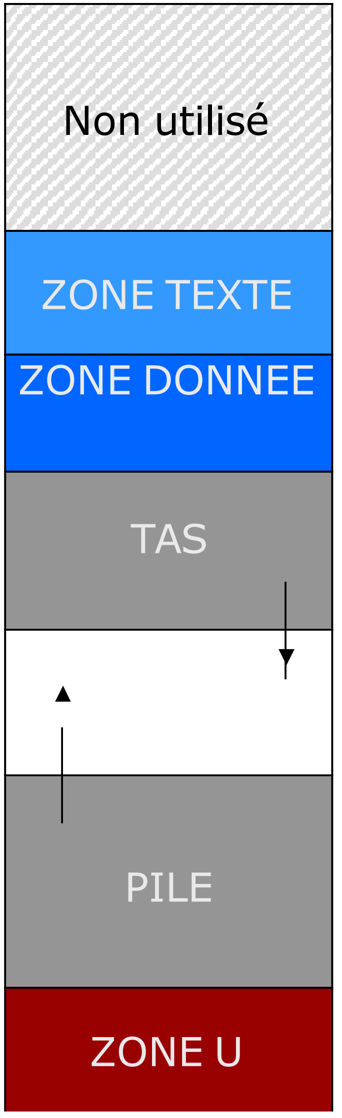


Variables statiques et automatiques

- Variables statiques (ou globales)
  - Durée de vie du programme
  - En C :
    - variable globale (déclarée à l'extérieure de toute fonction)
    - variable préfixée par le mot clé static
- Variables automatiques (ou locales)
  - Durée de vie associée à leur portée (fonction ou bloc)
  - En C : variable locale

Mémoire vue par le programmeur

- Code et les données sont chargées en mémoire juste avec l'exécution du programme par le loader
- La pile est gérée
  - En C : par le compilateur
  - En Assembleur : par le programmeur
- Le tas est géré par le programmeur
  - Interface : malloc/free
  - Implémentation : libc (allocateur mémoire)



Malloc/Free

- Utilisation
  - #include <stdlib.h>
  - void *malloc(size_t size);
  - void free(void *ptr);

- Fait partie de l'espace utilisateur
  - L'implémentation de l'allocateur mémoire fait des appels systèmes
    Groupe plusieurs appels à malloc (rapide) en un seul appel système (plus lent)
  - brk/sbrk : augmente la taille du tas
  - mmap : récupère de l'espace mémoire (libc : pour les grosses allocations)

- Critique pour les performances de certains codes (C,C++)
  - Parallélisme (plusieurs threads appellent malloc), Localité (NUMA)
  - D'autres implémentations que la libc existent (ex : Google TCMalloc)

Malloc/Free : problème de la fragmentation mémoire

- Après une série de malloc/free, la mémoire est fragmentée en parties utilisées et parties libérées

- Implémentation de l'allocateur mémoire
  - Choix de la zone où placer une nouvelle allocation
    - First fit / Best fit / Worse fit
  - Compromis performance / consommation mémoire

Problèmes posés par les adresses logiques

- Espace d'adresses logiques = abstraction de la mémoire
  - Simple pour le programmeur
  - Comment l'implémenter ?

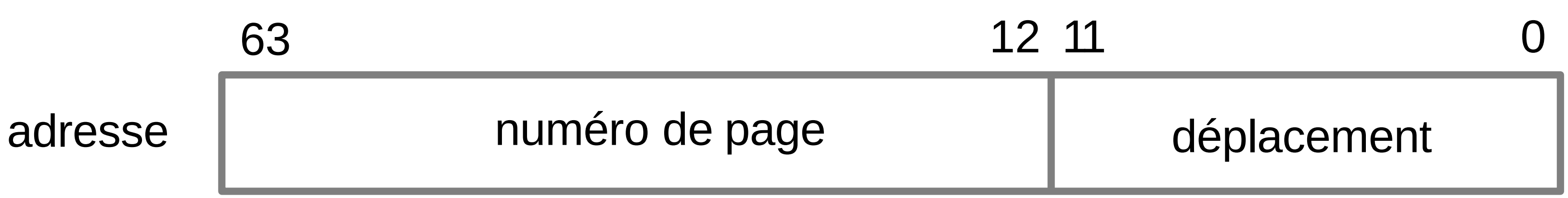
- La mémoire logique est plus grande que la mémoire physique. Comment fait on si on a moins de 2^{64} bits de mémoire ? (cas usuel)

- Multiprogrammation : plusieurs processus s'exécutent en « même temps »
  - Problème 1 : ils veulent utiliser la même adresse
  - Problème 2 : $\sum$ mémoires processus > mémoire principale

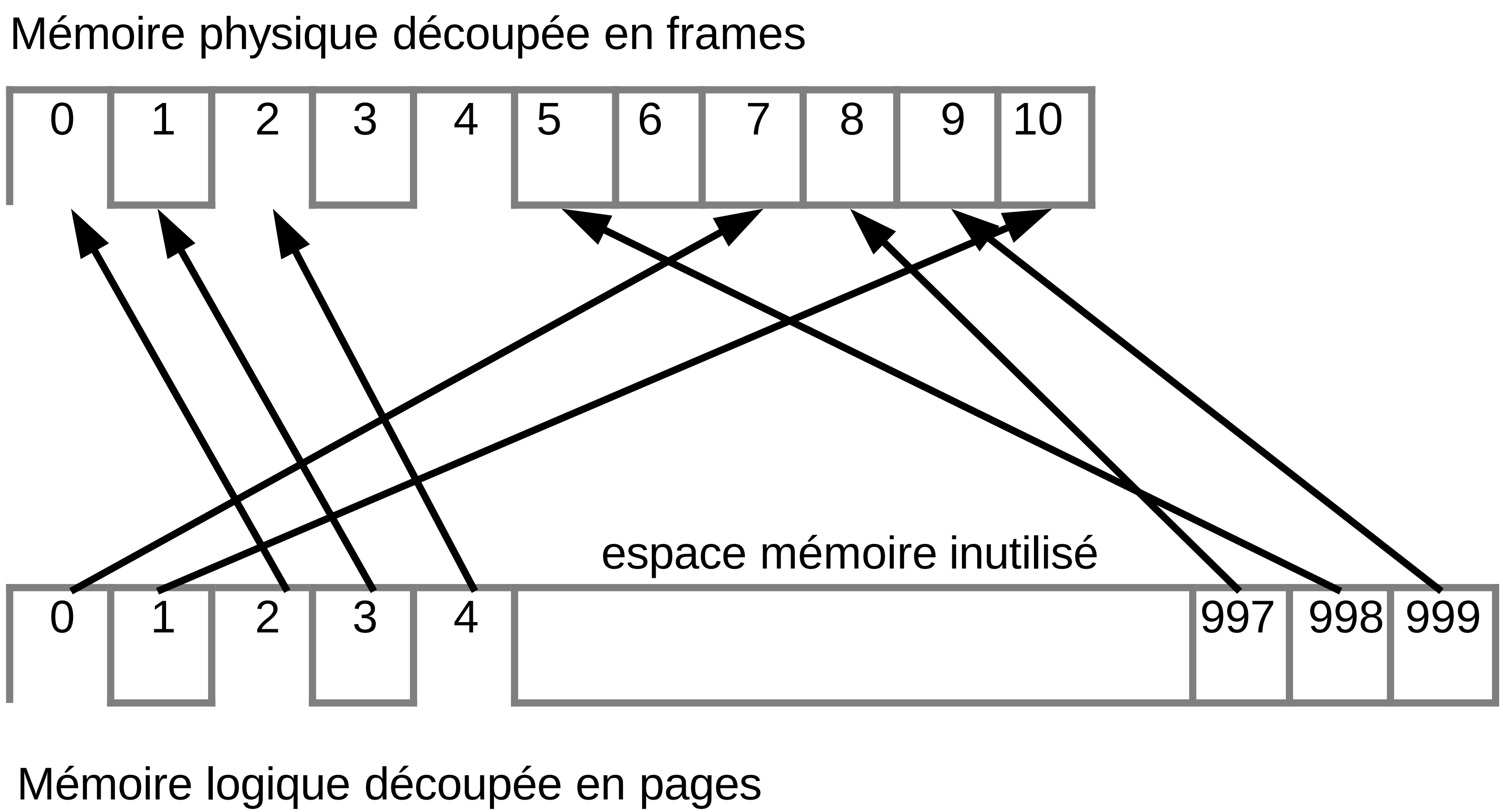
Solution : la mémoire virtuelle

- Mémoire physique et mémoire logique sont découpées en utilisant la même taille de zone
  - Mémoire physique : en frames
  - Mémoire logique : en pages
  - Taille usuelle : 4KO (huge page 2MO)

- Adresse logique = un numéro de page + un déplacement

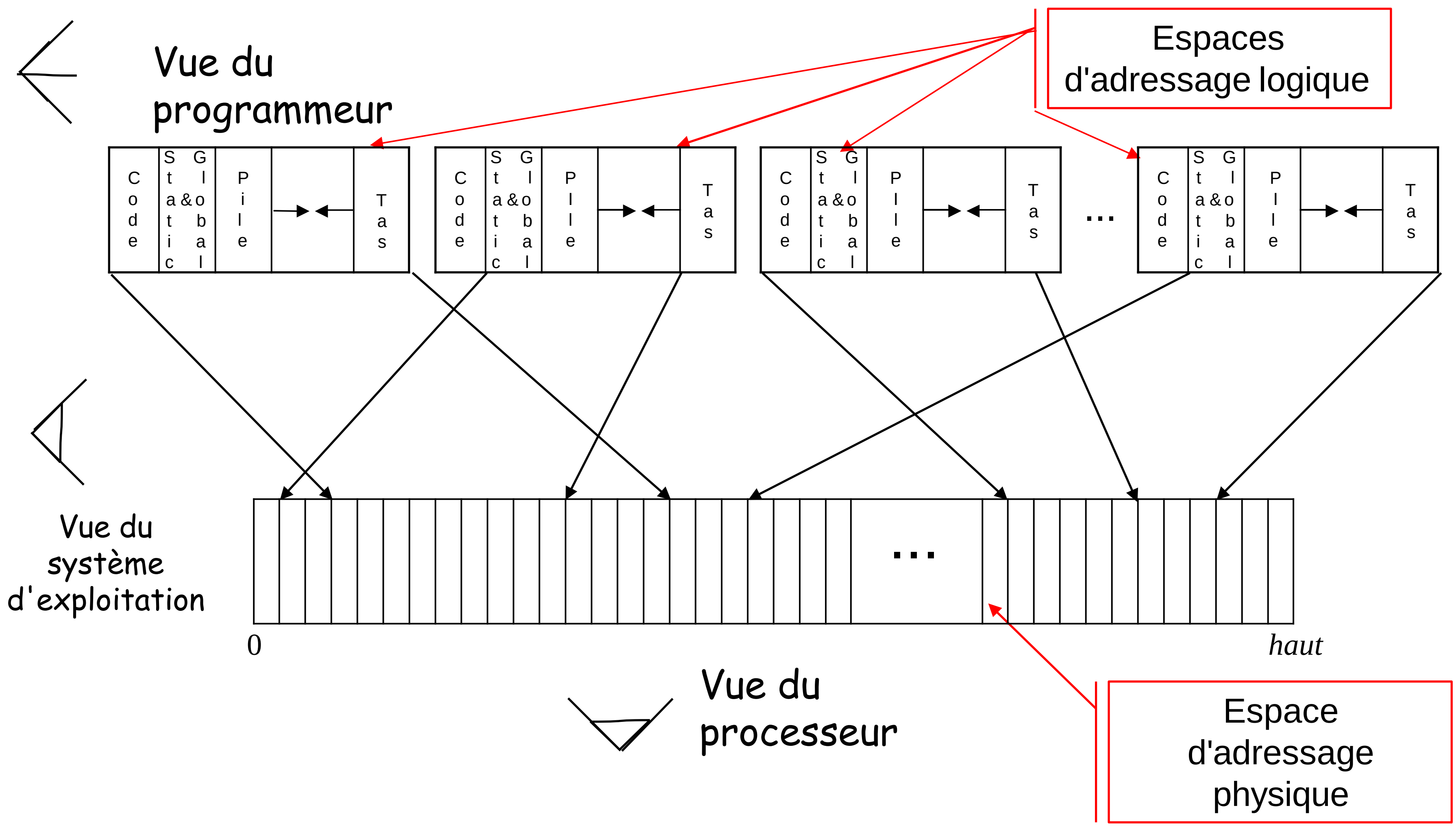


La mémoire virtuelle



La correspondance entre les pages et les frames est stockée dans la table des pages

Mémoire vue par le système d'exploitation

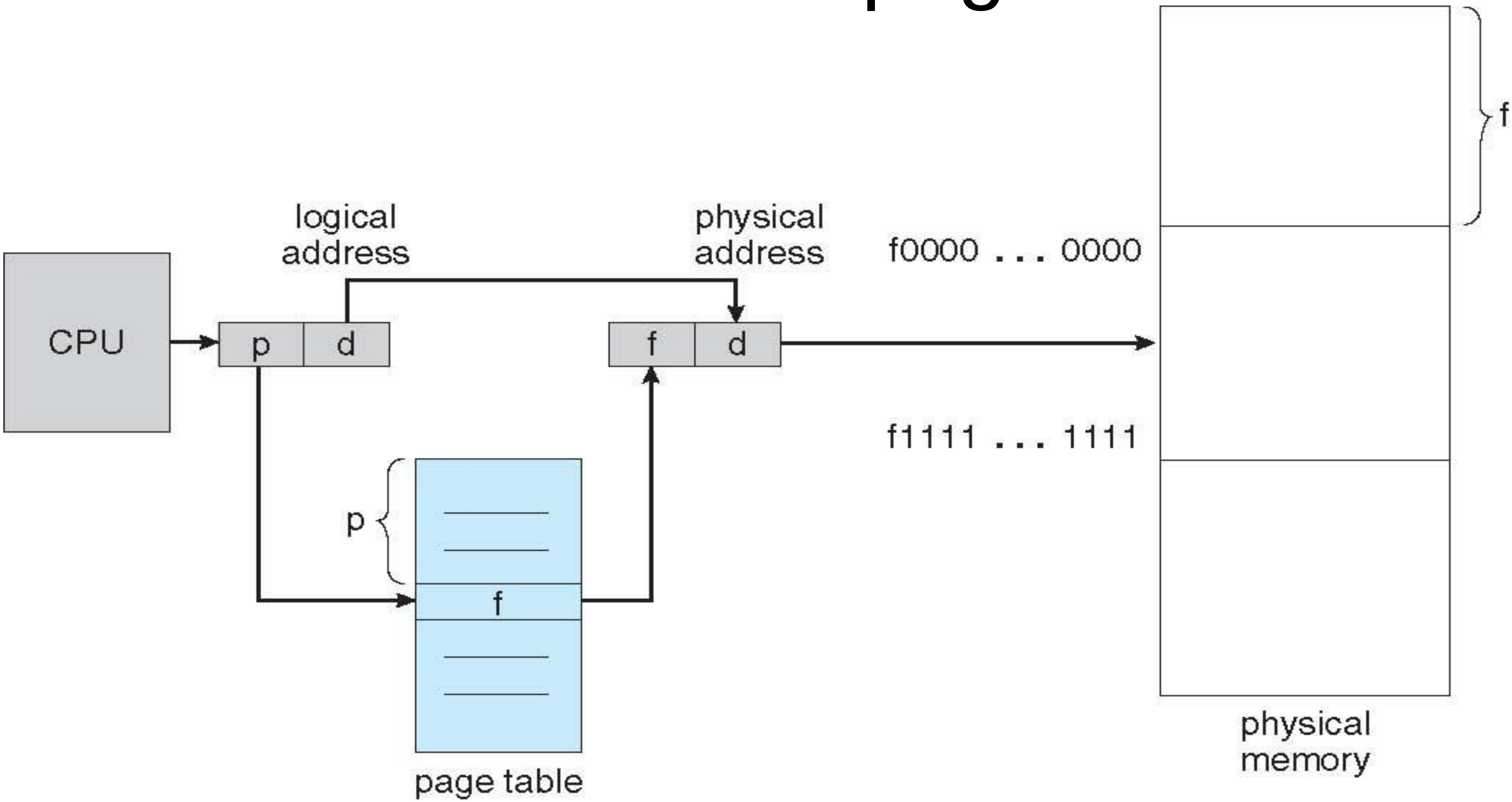


Gestion des frames et des pages

- Table des frames
  - Une case par frame
  - Libre ou allouée
  - Si allouée, information sur le processus et la page

- Table des pages
  - Par processus
  - Fait partie de son contexte

Traduction des adresses logiques en adresses physiques : la table des pages



Implémentation de la table des pages système

- Registre de table de pages
  - Table des pages en mémoire centrale
  - Un registre contient l'adresse de la table
  - Implique des accès mémoire lors de la traduction des adresses (lent)

- Mémoire cache dédiée
  - TLB : Translation Look-aside Buffer

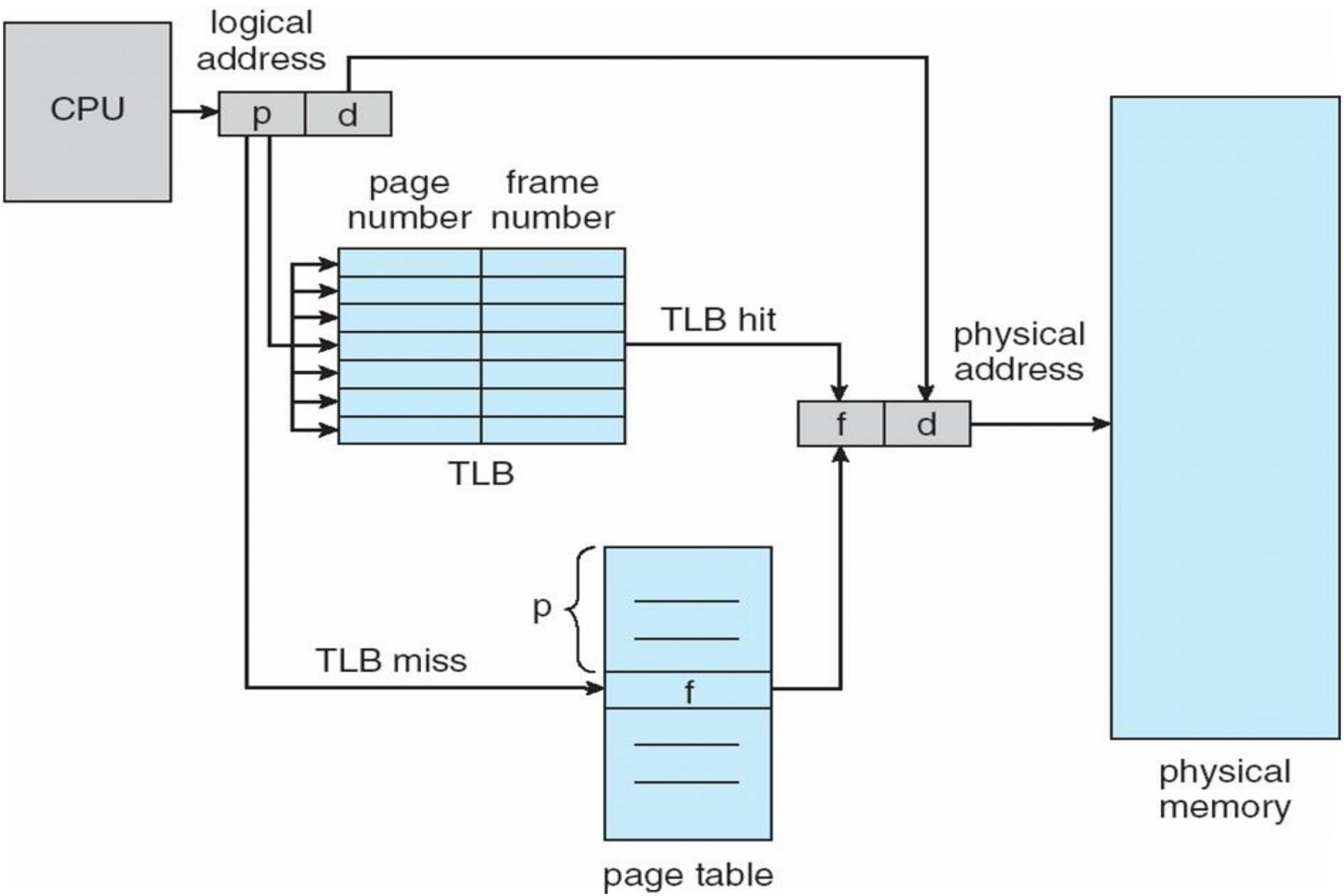
TLB : mémoire associative

- Mémoire associative : recherche très rapide en parallèle

Page#	Frame#

- Si le numéro de page est dans le TLB, on récupère le numéro de frame : très rapide car pas d'accès mémoire
- Sinon, récupérer le numéro dans la table des pages : lent car accès mémoire

Implémentation de la table des pages système avec TLB



Fichier d'échange / Swap

- Que faire si le nombre de pages utilisés par tous les processus est supérieur au nombre de frames (taille de la mémoire physique) ?

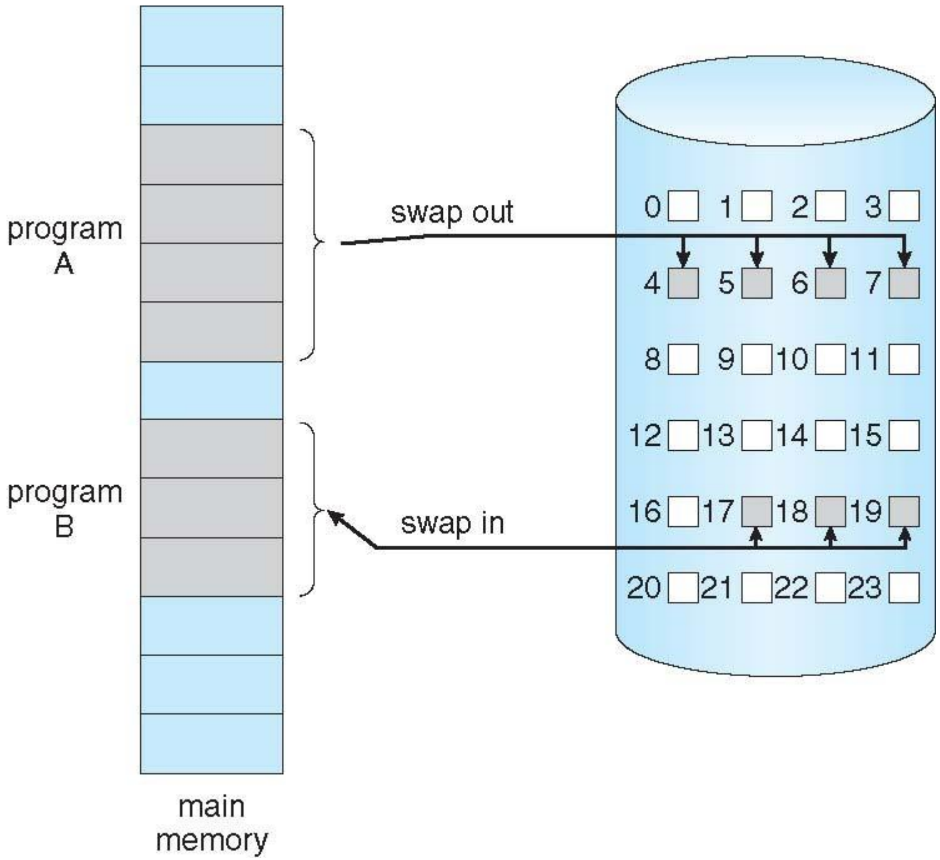
- Les pages supplémentaires sont stockées sur le disque dans une partition spécifique ou un fichier : le swap

- Si #pages x 4 KO > taille(mémoire physique + swap), le système d'exploitation tue un processus qui consomme beaucoup de mémoire

Fichier d'échange / Swap

- Les processus ne sont chargés que partiellement
- Les pages non chargées sont stockées sur le disque
- Le chargement est paresseux (lazy) : une page est chargée lors du premier accès à une adresse contenue dans la page

Fichier d'échange / Swap



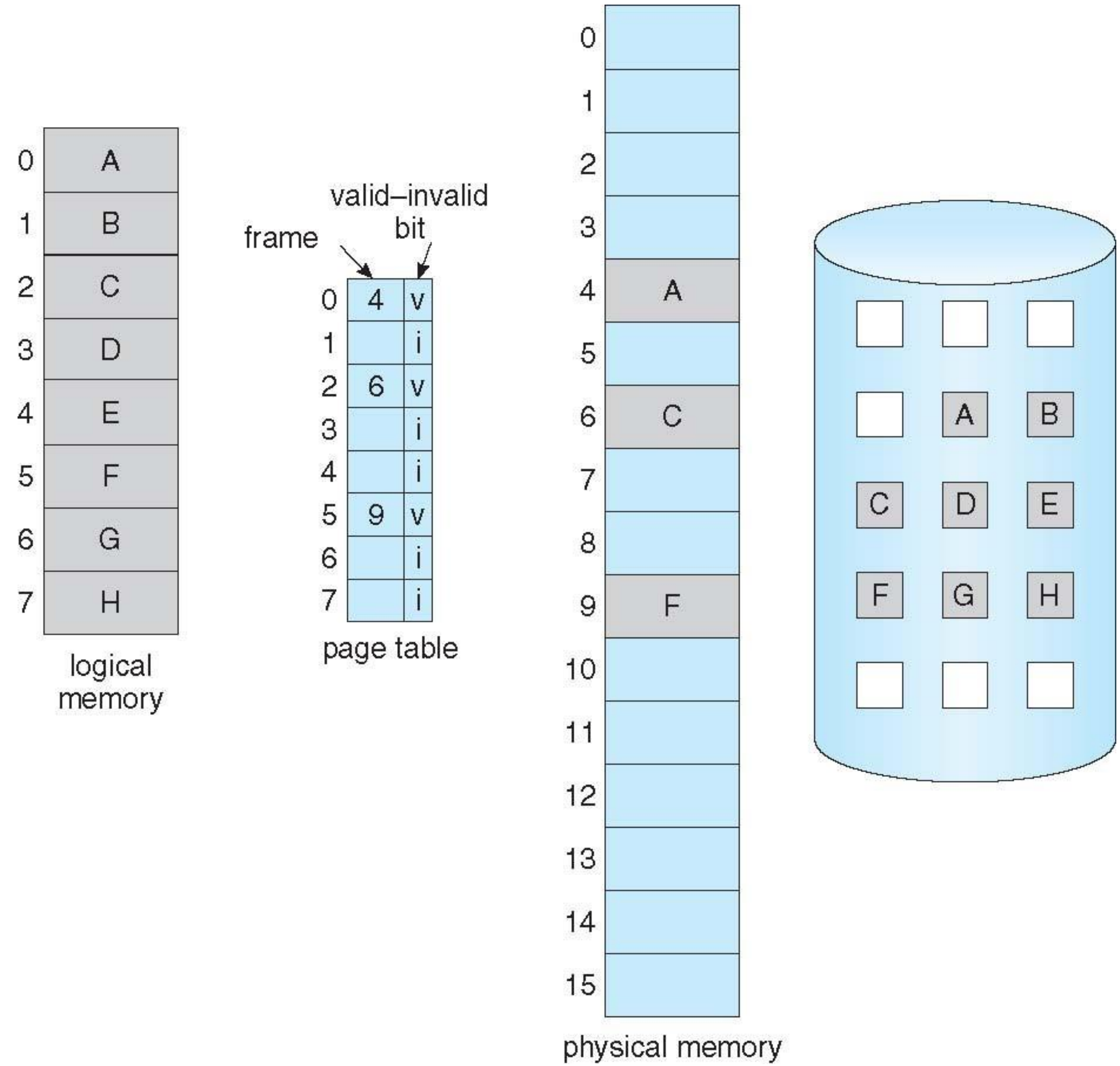
- Référence de page
  - Invalide
  - Page en mémoire : on continue l'exécution
  - Page sur disque : défaut de page (page fault)
    - On charge la page en mémoire
    - Si pas de place, remplacement de page

Bit de validité

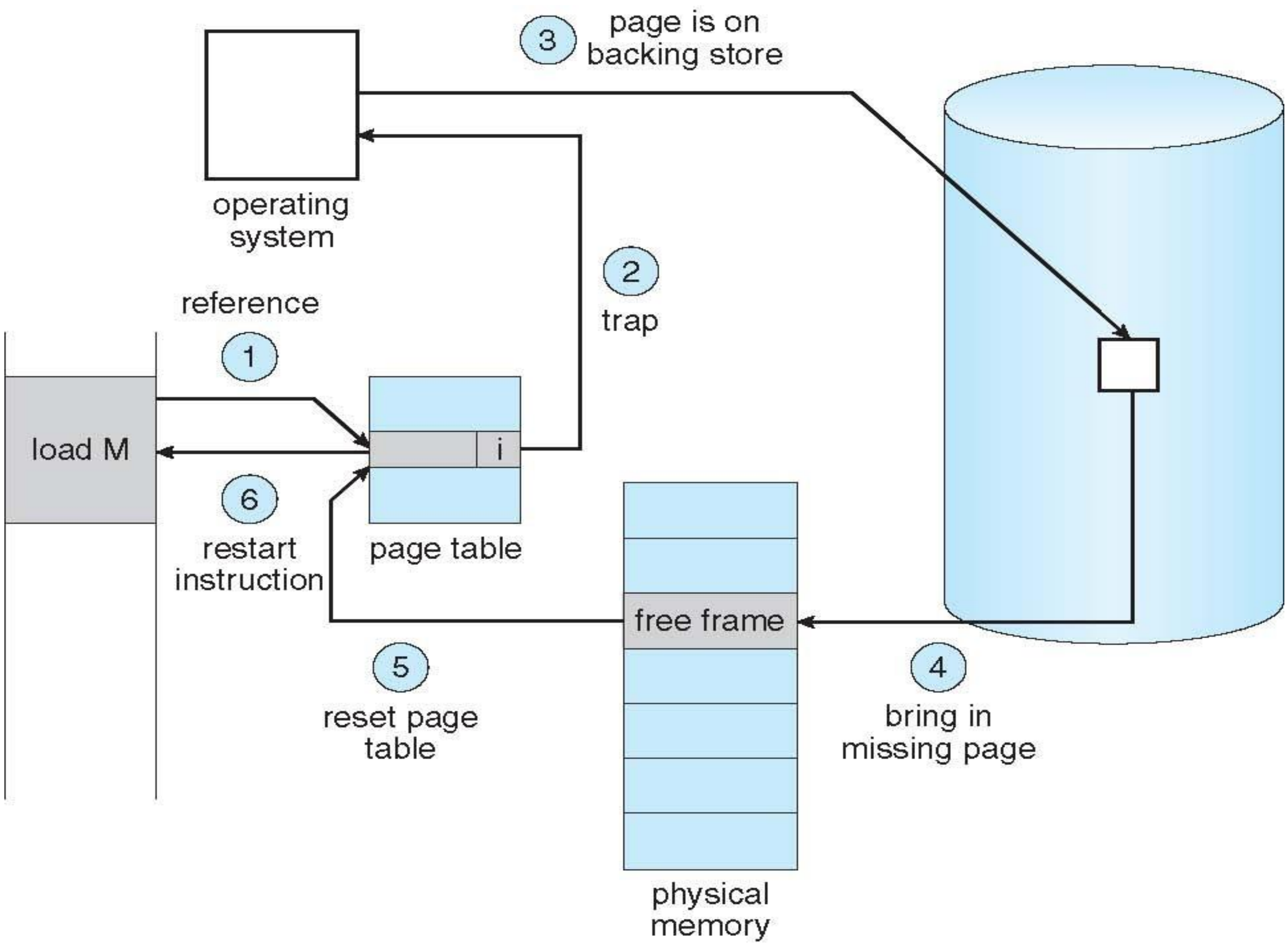
- Comment déterminer si la page est en mémoire ou sur disque ?
- On rajoute un bit de validité dans la table des pages

- Si le bit de validité est à 1, la page est en mémoire à l'emplacement de la frame
- Si le bit de validité est à 0, la page est sur le disque : déclenchement d'un défaut de page

Bit de validité



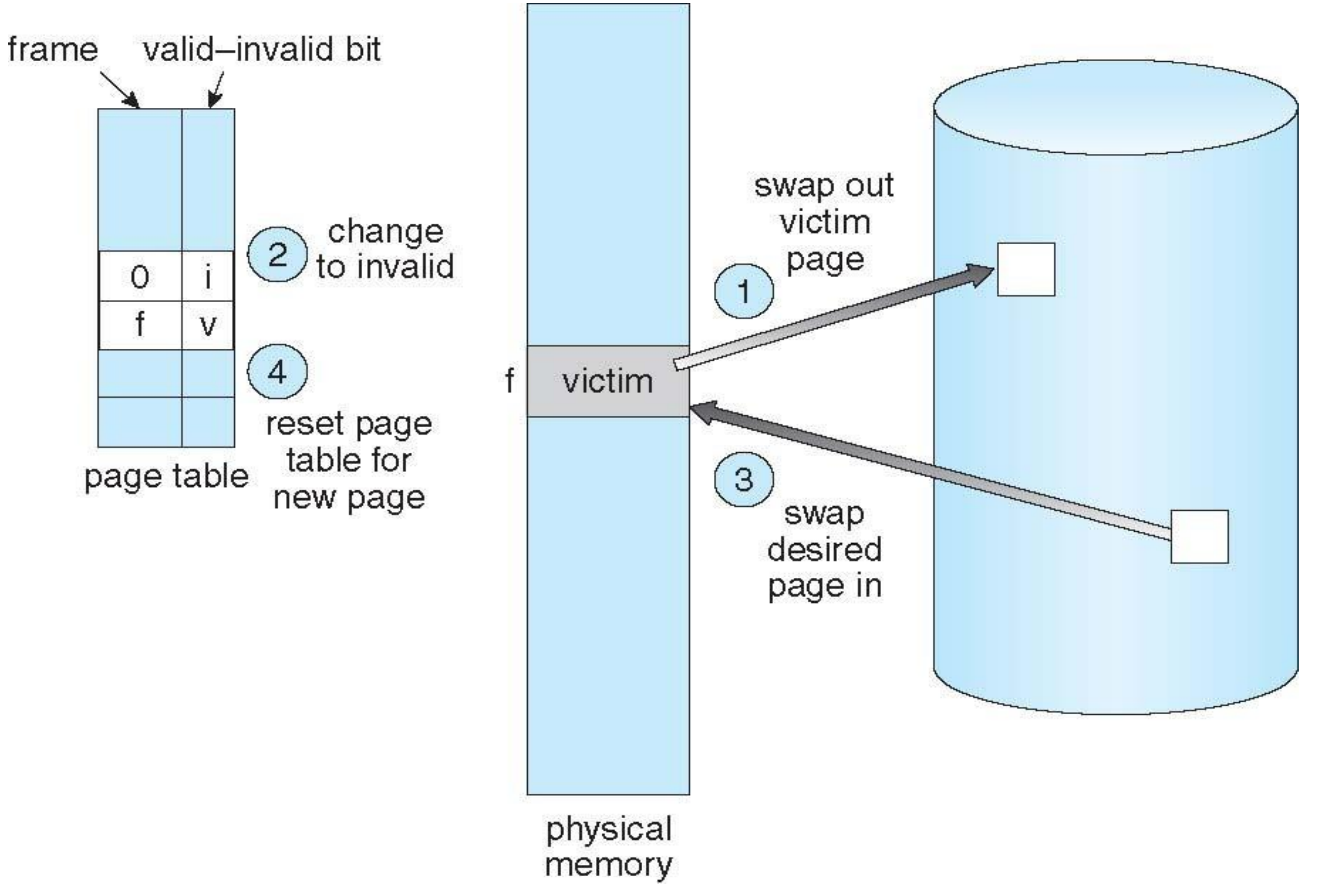
Gestion des défauts de page



Gestion des défauts de page

- Algorithme de préchargement
  - Quelles sont les pages à charger au lancement du processus ?
  - Cas extrême : aucune
  - Pagination à la demande pure
- Algorithme de remplacement
  - Si il n'y a pas de frame libre, laquelle des pages décharger ?
- Algorithme de répartition des frames entre processus
  - Combien de pages un processus a le droit de charger ?

Remplacement de page



Bit modifié (dirty)

- En plus du bit de validité, la table des pages comprend aussi un bit modifié
- Si la page a été modifié, le bit est à 1, on doit ré-écrire la page sur disque
- Si la page n'a pas été modifié, le bit est à 0, ce n'est pas la peine de ré-écrire la page sur disque

Algorithmes de remplacement

- Comment choisir la page victime ?
  - But : minimiser le nombre de défauts de page

- Algorithmes proposés
  - Random
  - FIFO (first in first out)
  - FIF (furthest in future)
  - LRU (least recently used)
  - Clock

Remplacement FIFO

- FIFO : first in first out
- Pages déchargées suivant l'ordre d'arrivée
- Gestion simple : une file d'attente FIFO

```
Pages adressées :
   7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Contenu des 3 frames :
f1 7 7 7 2 2 2 2 4 4 4 0 0 0 0 0 0 0 7 7 7
f2   0 0 0 0 3 3 3 2 2 2 2 2 1 1 1 1 1 0 0
f3     1 1 1 1 0 0 0 3 3 3 3 3 2 2 2 2 2 1
                        15 défauts de page
```

Remplacement FIF

- FIF : furtherst in future
- Remplacer la page qui sera accédée le plus tard
- Optimal mais nécessite de connaître le futur !

```
Pages adressées :
   7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Mémoire de 3 blocs :
b1 7 7 7 2   2   2     2     2       7
b2   0 0 0   0   4     0     0       0
b3     1 1   3   3     3     1       1
                         9 défauts de page
```

Remplacement LRU

- LRU : Least Recently Used
- Remplacer la page non utilisée depuis le plus longtemps

```
Pages adressées :
   7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Mémoire de 3 blocs :
b1 7 7 7 2 2 2   4 4 4 0     1   1   1
b2   0 0 0 0 0   0 0 3 3     3   0   0
b3     1 1 1 3   3 2 2 2     2   2   7
                                      12 défauts de page
```

- Utilisé, considéré performant, implémentation coûteuse

Remplacement Clock

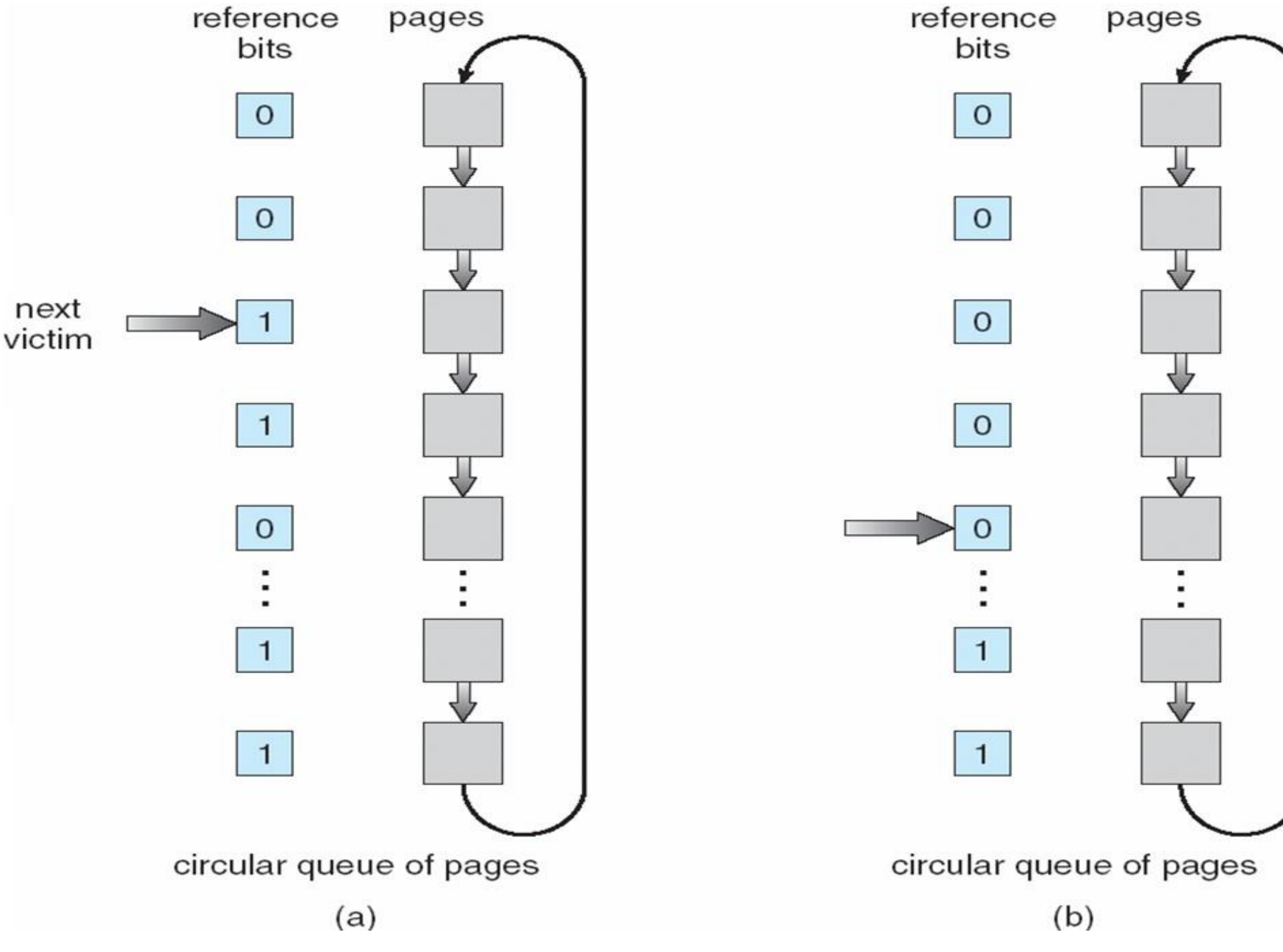
- Ou algorithme de la deuxième chance

- Bit de référence
  - Initialement page non utilisée : bit = 0
  - Utilisation de la page : bit = 1

- Deuxième chance
  - Liste circulaire FIFO
  - Si bit == 0, remplacement
  - Si bit == 1, bit ← 0, vers page suivante

- Approximation de LRU moins coûteux à implémenter

Remplacement Clock



Classement des algorithmes de remplacement de pages

- FIF
- LRU
- Clock
- FIFO
- Random

- Dépend des programmes

Répartition des frames entre les processus

- Comment partager les frames entre les processus ?
- Égalité entre processus
  - avantages / inconvénients ?
- Variations dynamiques

Gestion globale ou locale des frames

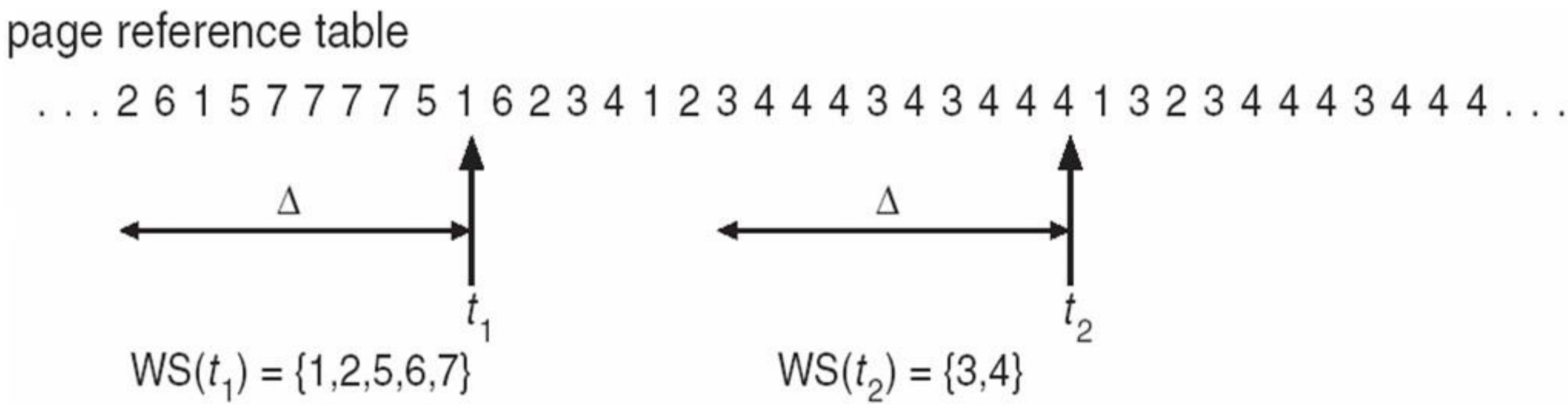
- Gestion globale
  - Quand une nouvelle frame est demandée par un processus, la frame allouée peut appartenir à un autre processus
  - Variation dynamique du nombre de frames par processus
- Gestion locale
  - Quand une page doit être déchargée, ne sont considérées que les frames allouées au processus
  - Le nombre de frames par processus reste constant

Problème du remplacement global

- Considérons la situation suivante
  - Un processus s'exécute et essaye d'accéder à des pages non chargées
  - Le chargement des pages implique des E/S disque, le processus est en attente et peut relâcher le processeur
  - Les autres processus n'ont pas assez de mémoire, ils génèrent aussi des défauts de page
  - Le processeur est inoccupé et aucun processus en mémoire n'est prêt, on va essayer de charger de nouveaux processus prêts
  - L'utilisation du processeur diminue, augmentation du nombre de processus en exécution
- Les processus passent plus de temps à charger des pages qu'à calculer : crash du système (trashing)

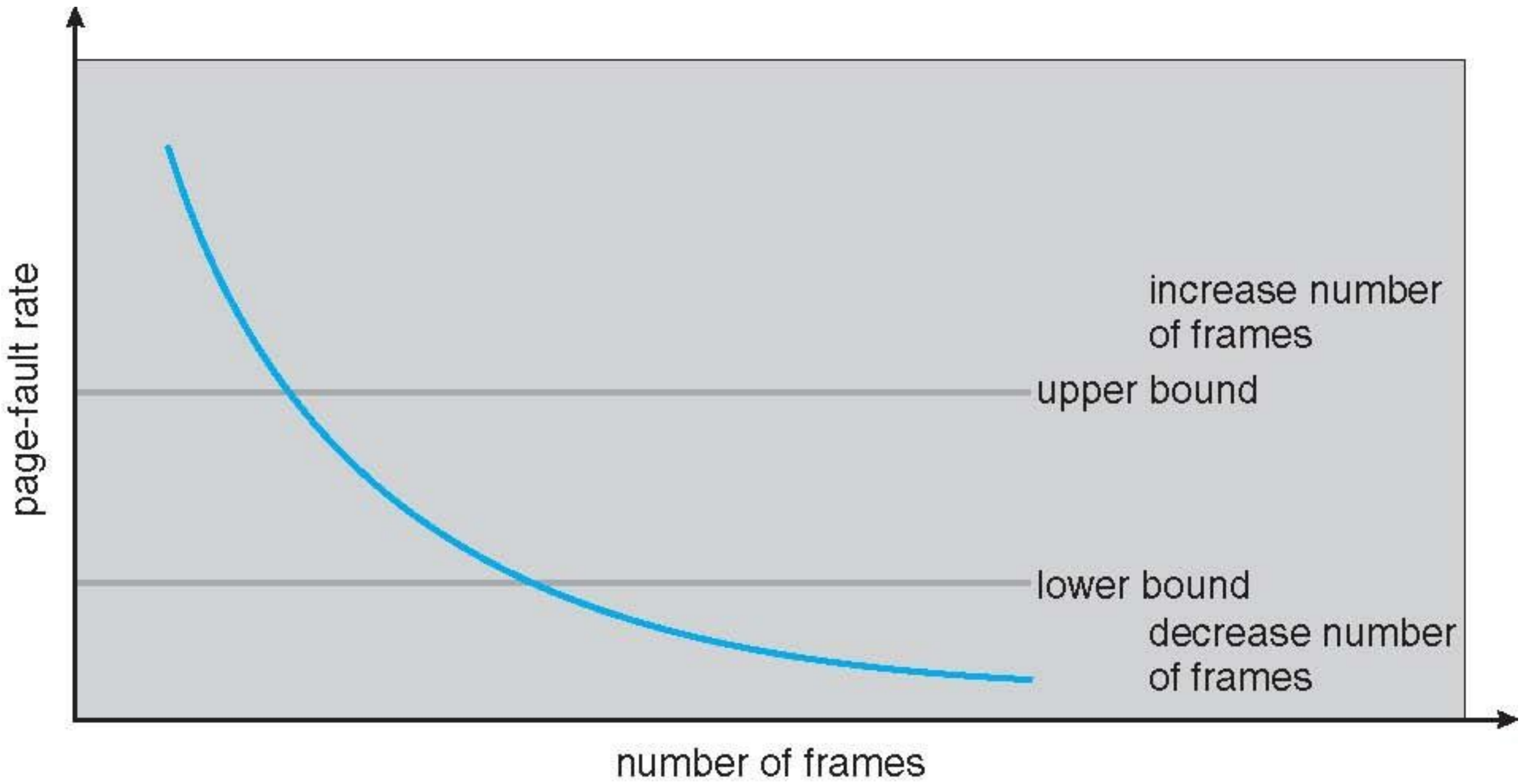
Remplacement local

- Comment décider du nombre de frames par processus ?
- Notion d'ensemble de travail (working set)
  - Ensemble des pages accédées dans le dernier intervalle de temps



- Algorithme du working set
  - si la somme des WS de tous les processus est supérieure à la mémoire physique
  - on suspend l'exécution d'un des processus et on décharge ses pages de la mémoire

Working Set



page-fault rate
increase number of frames
upper bound
lower bound
decrease number of frames
number of frames

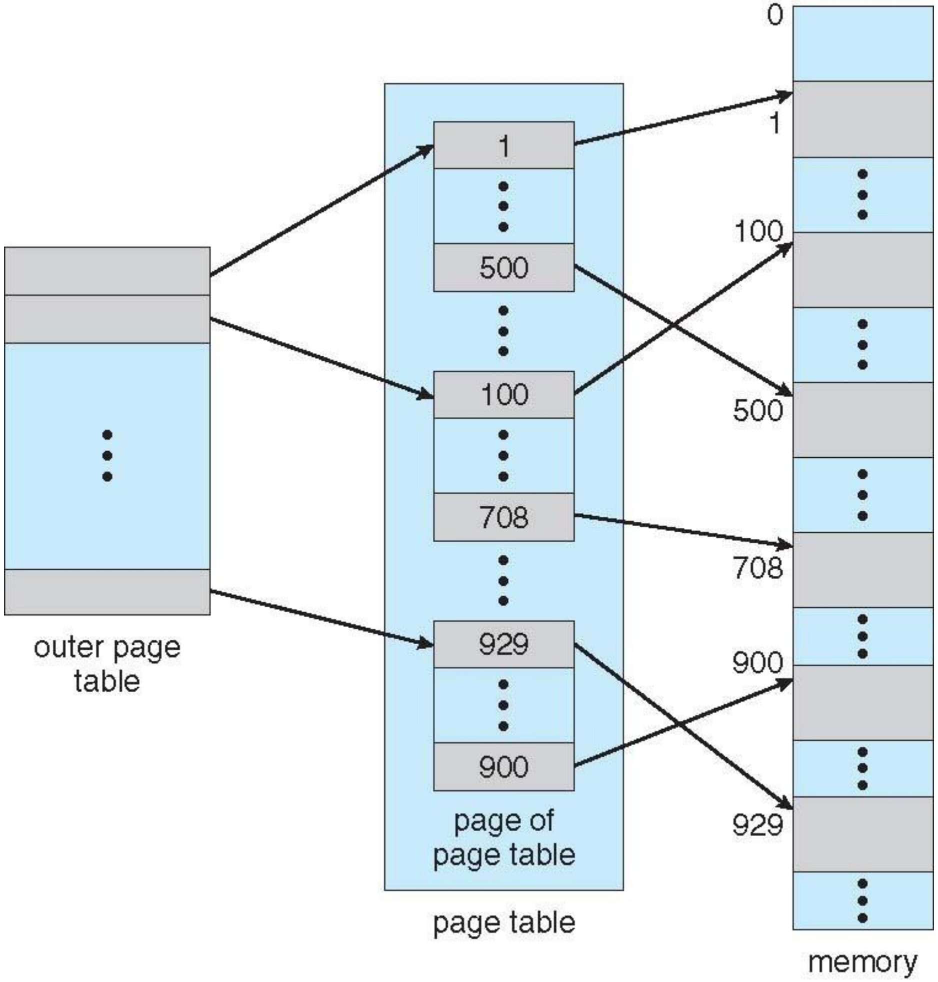
Table des pages hiérarchique

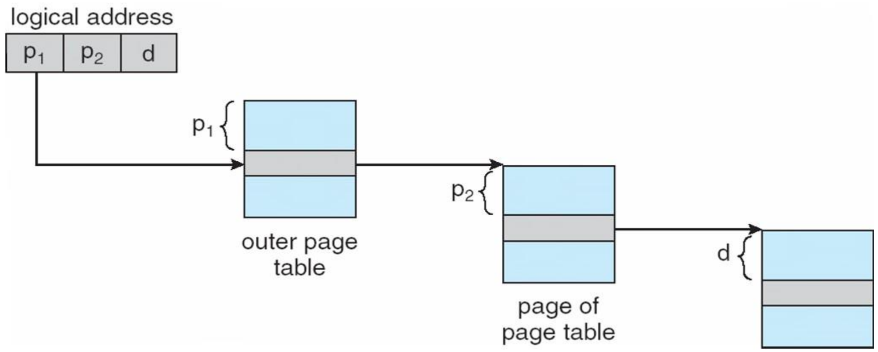
- Taille d'un table de page de 4KO en 64bit
  - Taille d'une page : 2^12
  - Nombre de # de pages : 2^52
  - Taille de la table : 2^52 * 4O → énorme !

- Découper la table des pages en plusieurs niveaux

	63 … 33	32 … 12	11 … 0
adresse	# page externe	# page interne	déplacement

Table des pages hierarchiques





mmap

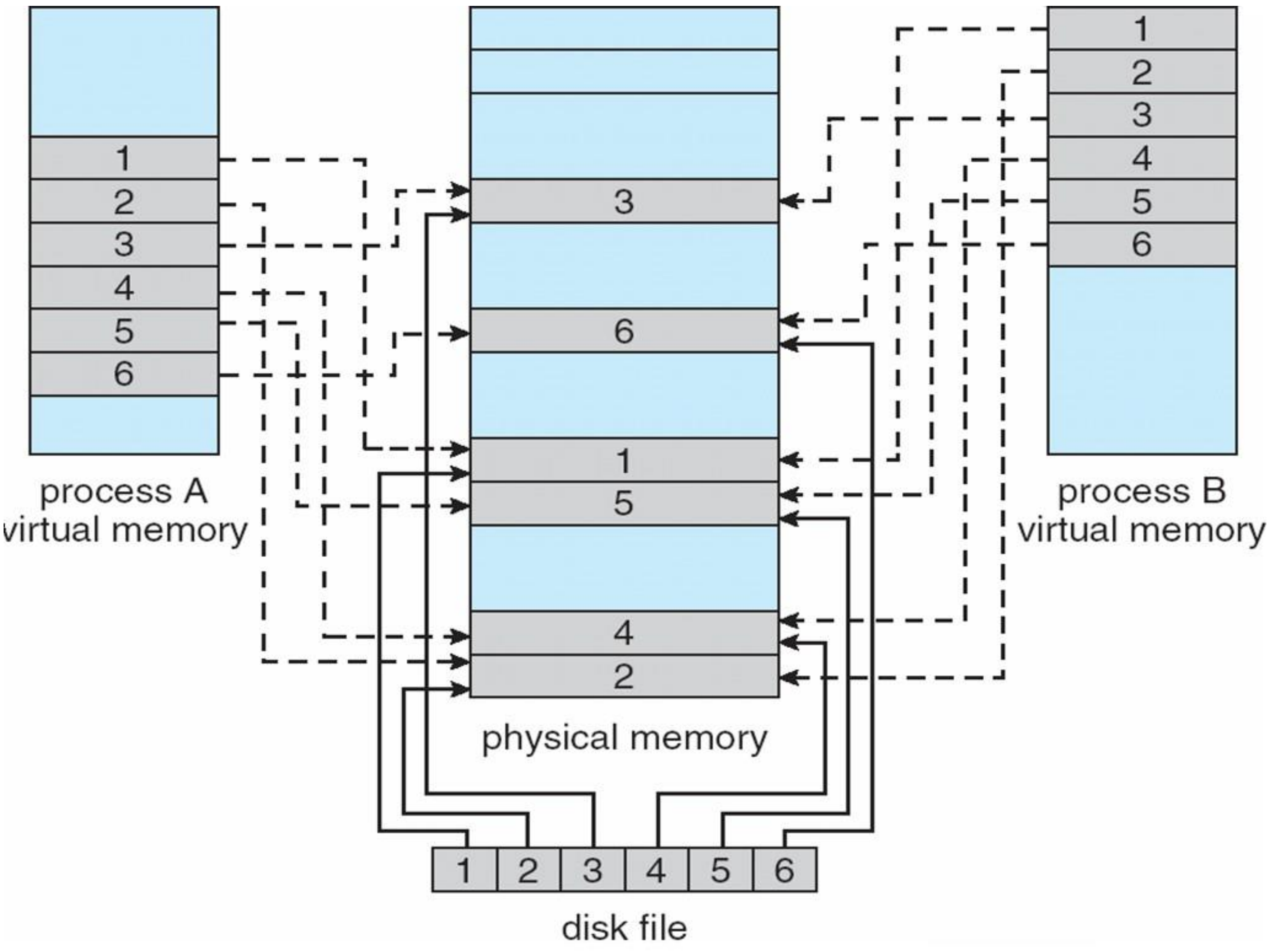
- void * mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);
  - mmap demande la projection en mémoire de length octets commençant à la position offset depuis un fichier indiqué par le descripteur fd

- remplace les E/S dans des fichiers par des opérations en mémoire : plus simple, plus rapide

- pagination à la demande : le fichier n'est pas intégralement chargé en mémoire

- permet de recouvrir les chargements par des calculs

mmap



1
2
3
4
5
6
process A
virtual memory
3
6
1
5
4
2
physical memory
1
2
3
4
5
6
process B
virtual memory
1
2
3
4
5
6
disk file

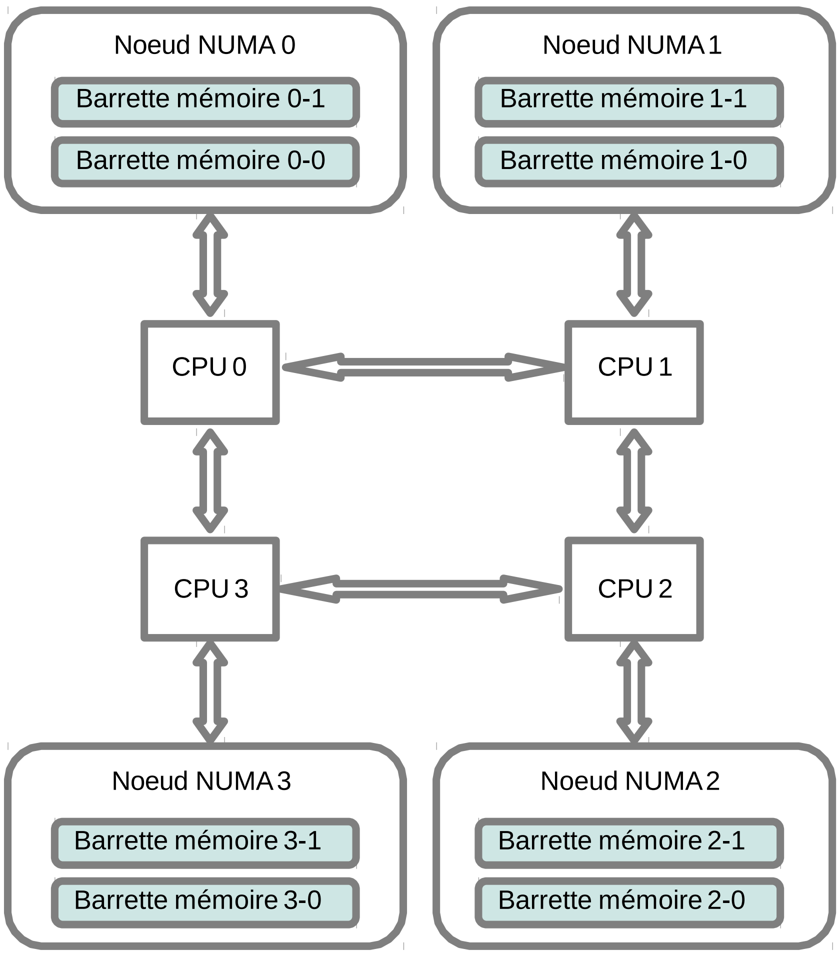
NUMA et la politique first touch

- Affinité mémoire : critique pour les performances
  - Bande passante
  - Latence

- First touch
  - premier accès à la page
  - Polique de linux par défaut

- Lorsqu'une page est accédée pour la première fois, on alloue une frame dans une barrette mémoire sur le même nœud NUMA que le thread qui a fait le défaut de page

- On peut modifier cette politique avec, par exemple, la libnuma
  - numactl --interleave



Mémoire virtuelle : conclusion

- 4 endroits où l'OS intervient

1. Création de processus
   - décider de la taille de la table
   - créer la table des pages
2. Exécution de processus
   - remise à zéro du TLB
3. Défaut de page
   - déterminer l'adresse virtuelle qui fait défaut
   - swap entre page de remplacement et page demandée
4. Terminaison
   - libérer la table des pages, les pages en mémoire

Allocation mémoire en espace utilisateur

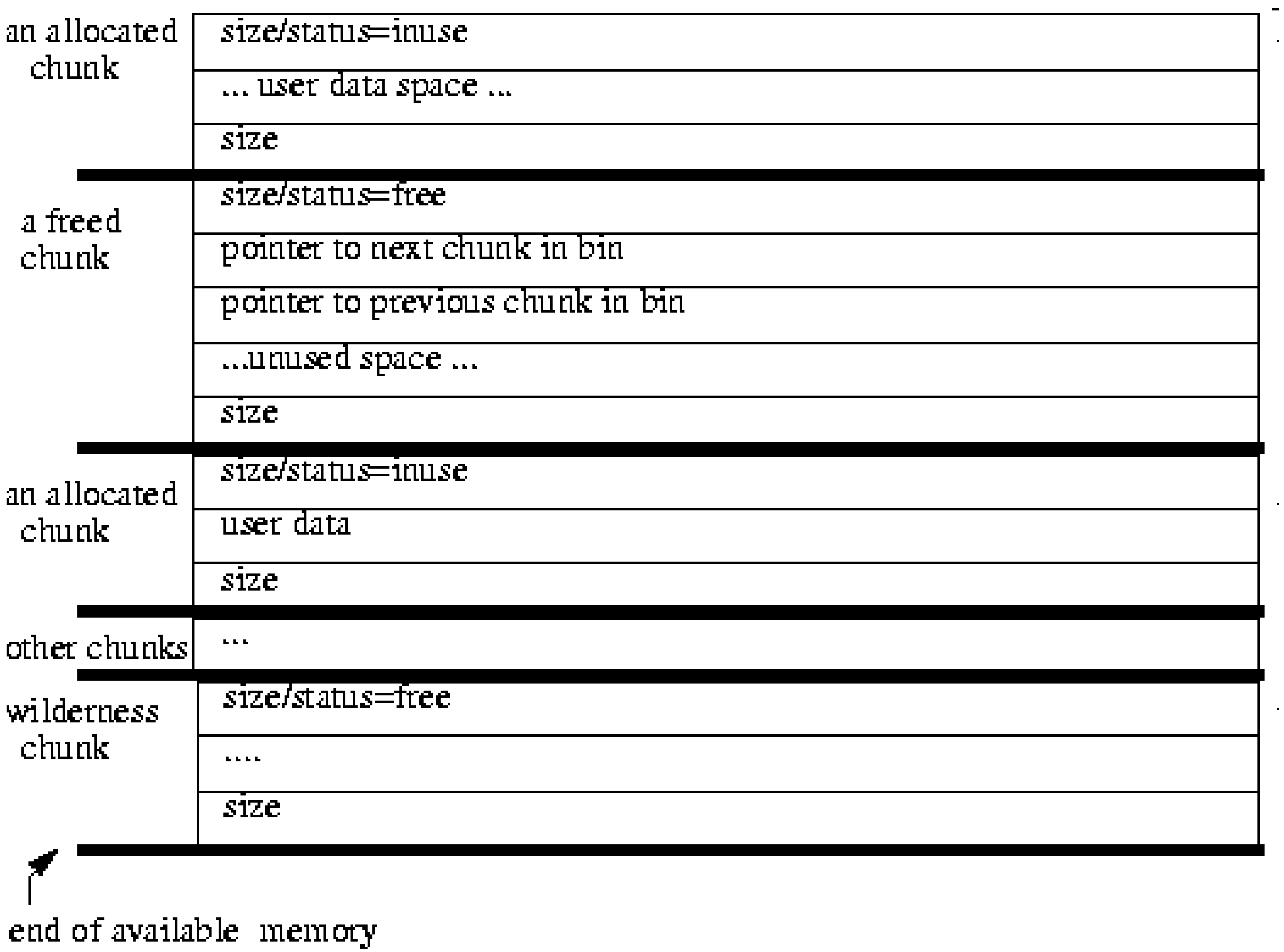
Issue de What Every Programmer Should Know About Memory de Ulrich Drepper
http://futuretech.blinkenlights.nl/misc/cpumemory.pdf

Fonctionnement de malloc/free: Linux sans arena

Approche centralisée

- Verrou en entrée de fonction
- Pas de distinction entre threads
- Pas de notion de NUMA
- First touch
- Pas de lien avec l’ordonnanceur

Fonctionnement de malloc/free: Linux sans arena



Fonctionnement de malloc/free: Linux avec arena

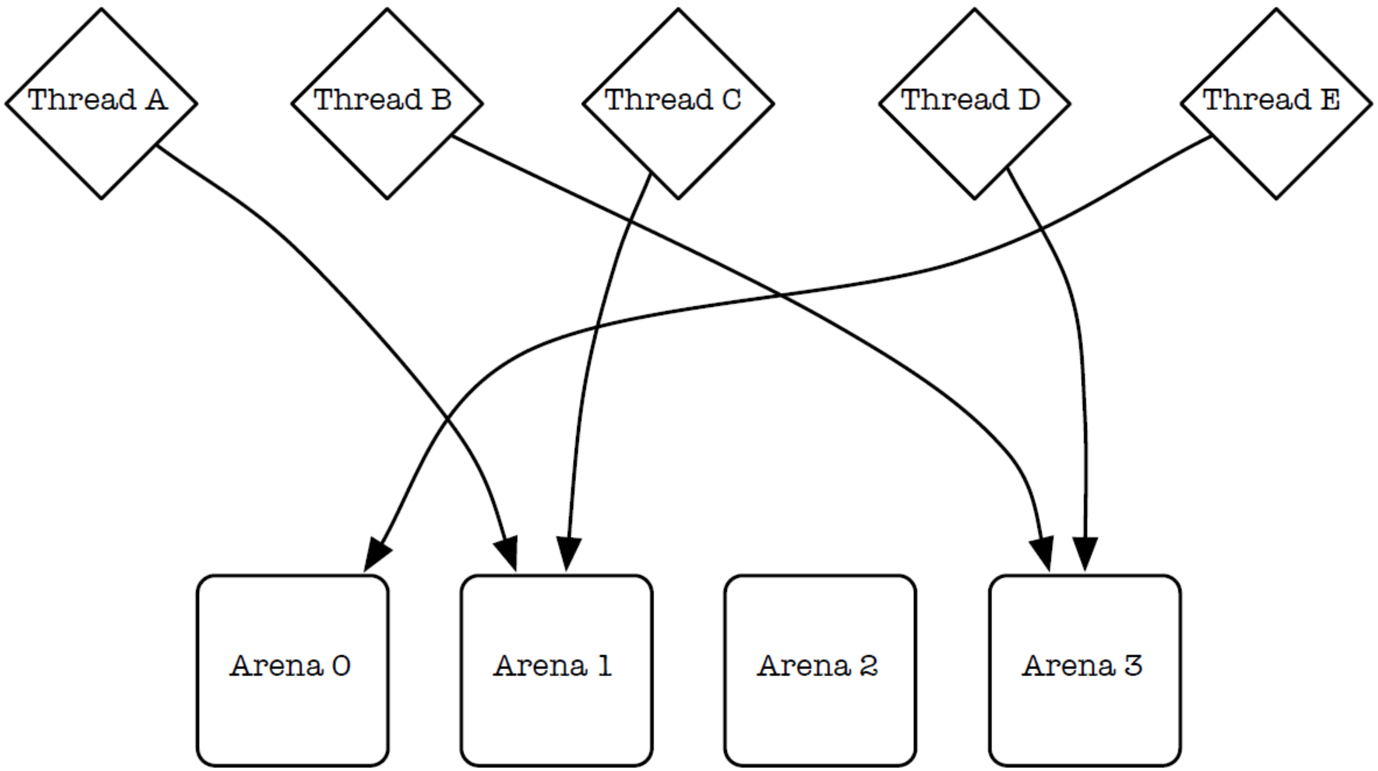


Figure 2: Larson and Krishnan (1998) hash thread identifiers in order to permanently assign threads to arenas. This is a pseudo-random process, so there is no guarantee that arenas will be equally utilized.

Fonctionnement de malloc/free: BSD

Approche hiérarchique

- Notion de thread
- Verrous distribués
- Pas de notion de NUMA
- Pas de lien avec l’ordonnanceur

Fonctionnement de malloc/free: BSD

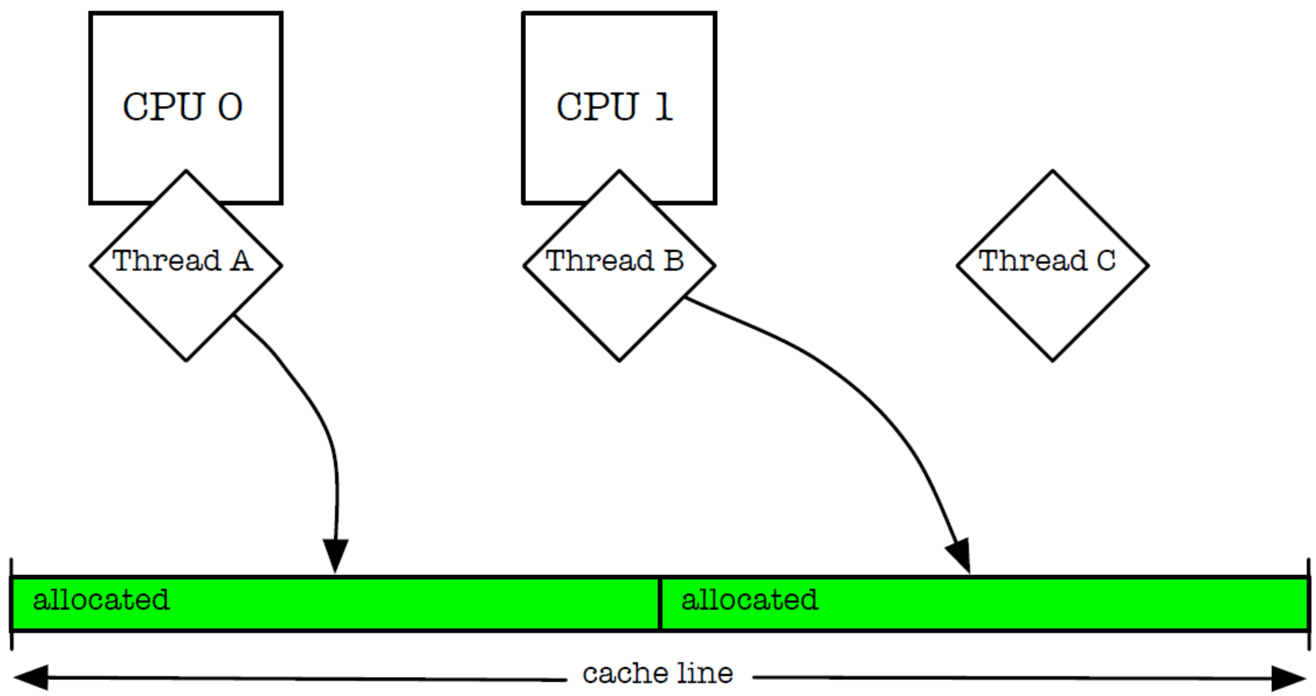


Figure 1: Two allocations that are used by separate threads share the same line in the physical memory cache (false cache sharing). If the threads concurrently modify the two allocations, then the processors must fight over ownership of the cache line.

Fonctionnement de malloc/free: BSD

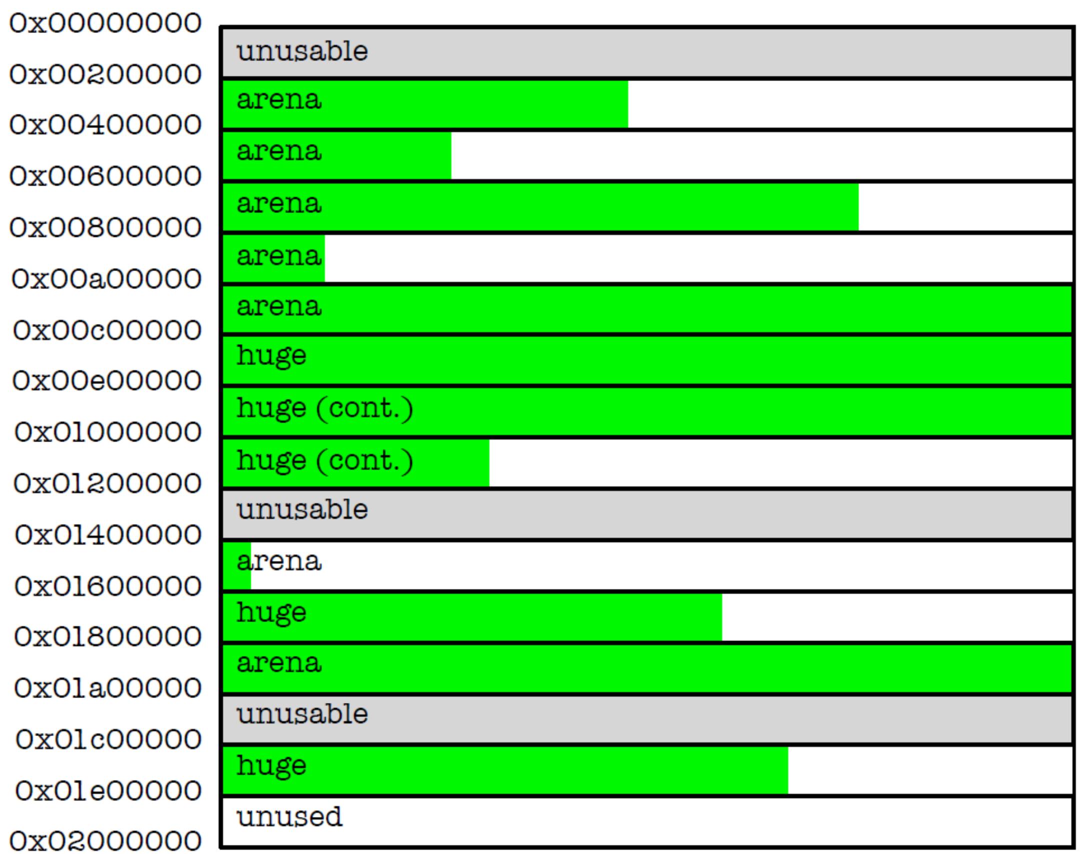


Figure 3: Chunks are always the same size, and start at chunk-aligned addresses. Arenas carve chunks into smaller allocations, but huge allocations are directly backed by one or more contiguous chunks.

Fonctionnement de malloc/free: BSD

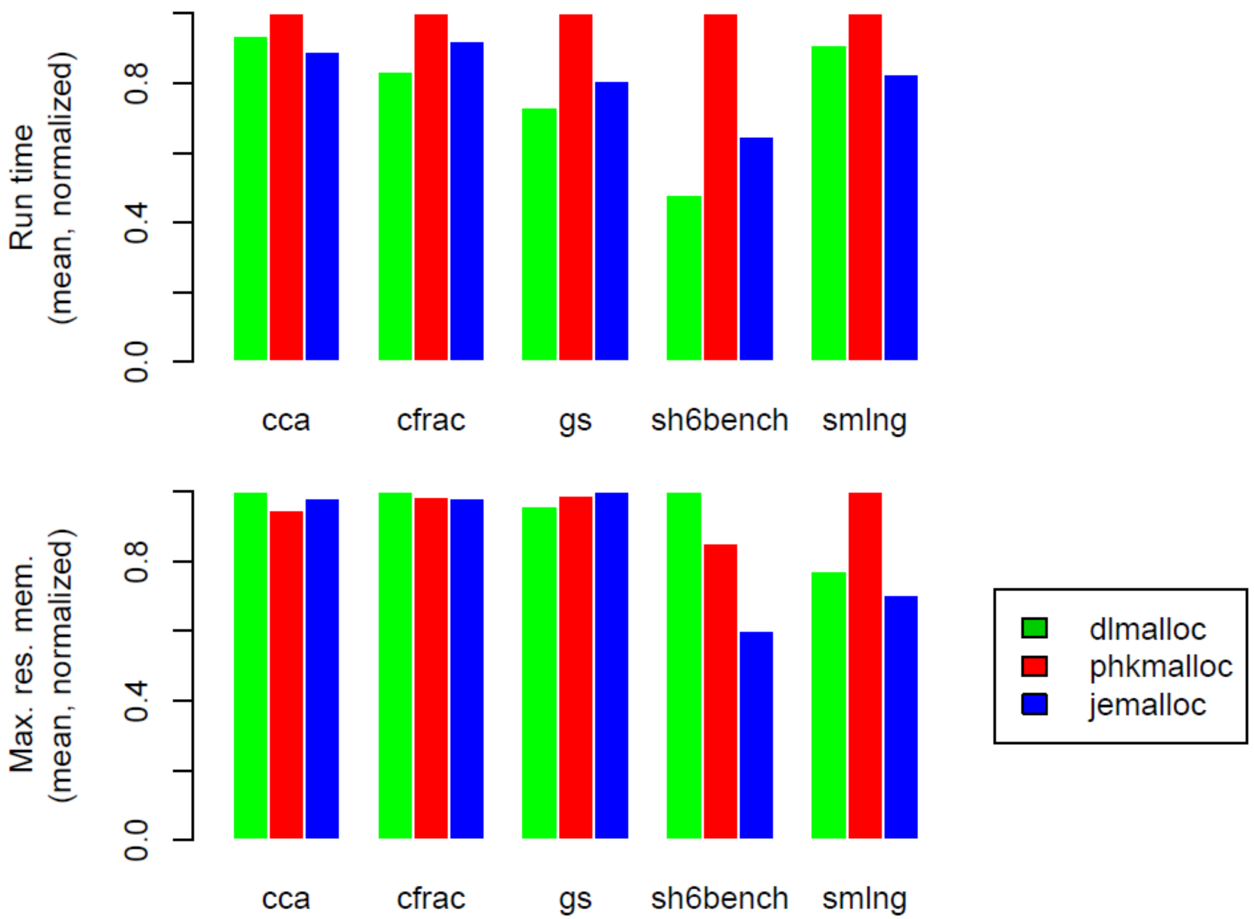


Figure 8: Scaled run time and maximum resident memory usage for five single-threaded programs. Each graph is linearly scaled such that the maximum value is 1.0.

Fonctionnement de malloc/free: BSD

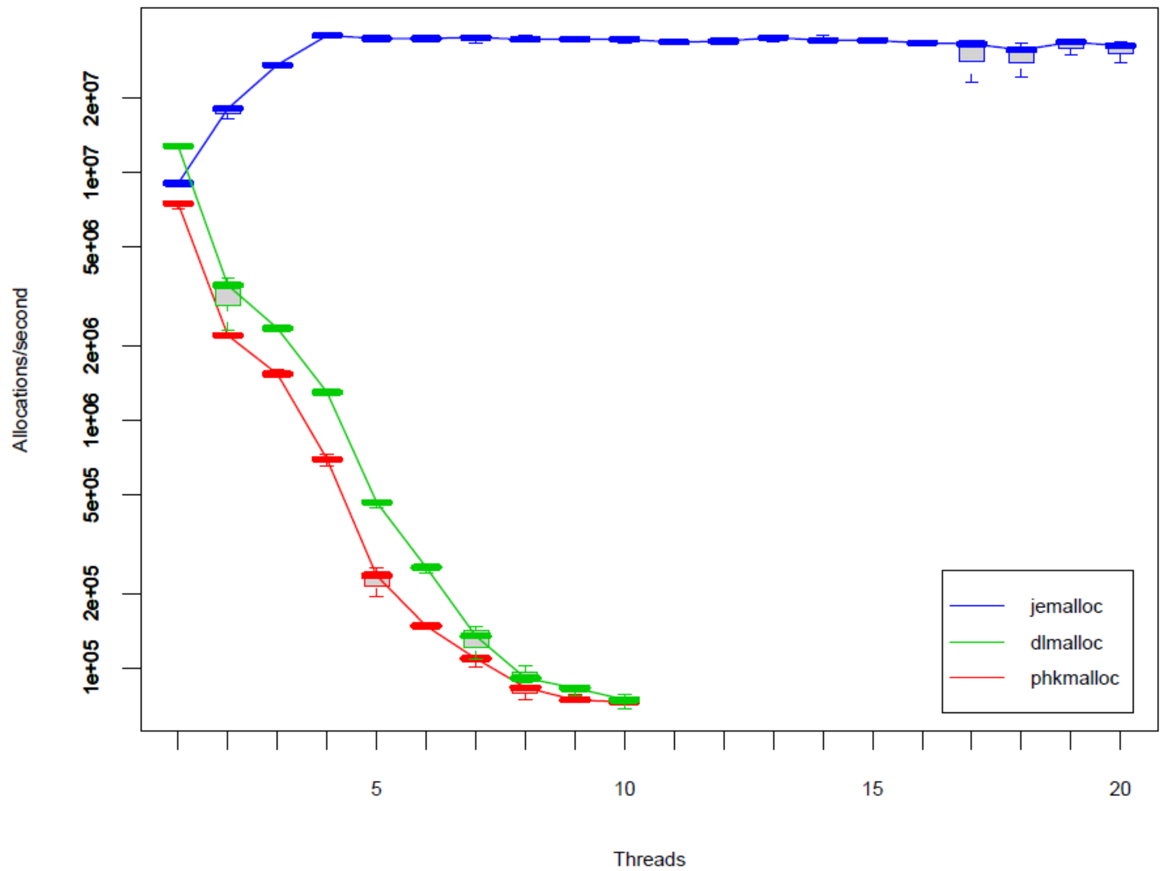


Figure 6: Allocator throughput, measured in allocations/second, for increasing numbers of threads. Each run performs a total of 40,000,000 allocation/deallocation cycles, divided equally among threads, creating one 512-byte object per cycle. All configurations are replicated three times, and the results are summarized by box plots, where the central lines denote the medians and the whiskers represent the most extreme values measured.

Fonctionnement de malloc/free

- Peu optimisé pour le contexte multithread
  - Support Linux des threads très limité (thread safe uniquement)
  - Non lié à l'ordonnanceur (migration de pages)
  - Pas de support OpenMP
- Pas de problème en MPI
  - Vision séquentielle
  - Pas de migration
  - Localité intrinsèque au modèle

Allocation mémoire en contexte HPC

Issue des travaux de thèse de Sébastien Valat
https://tel.archives-ouvertes.fr/tel-01253537/document

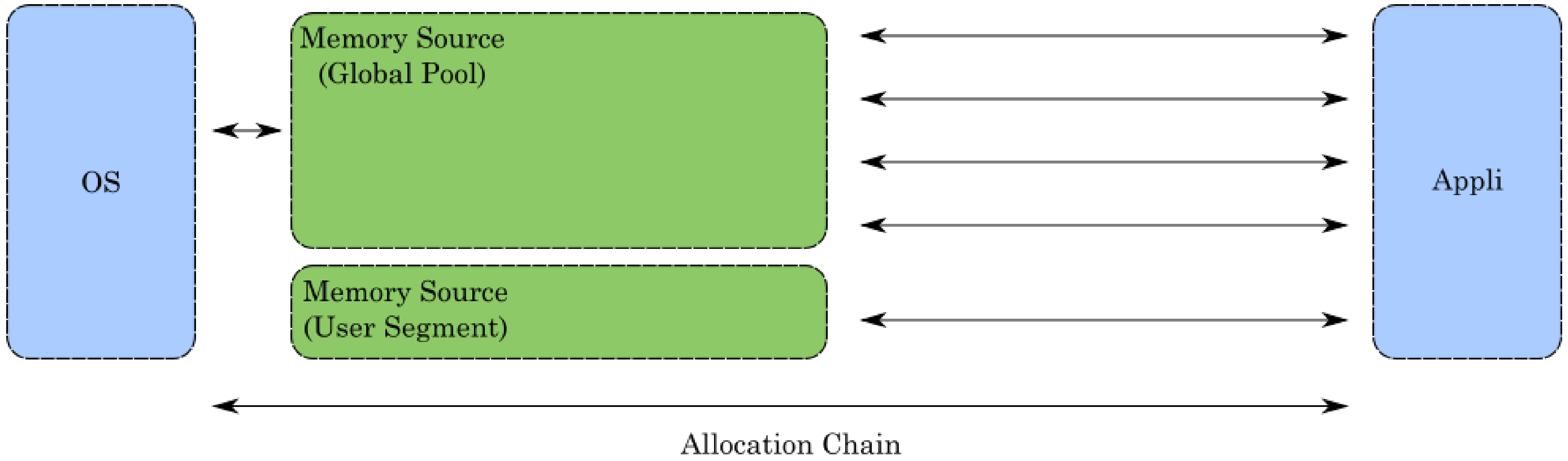
Memory Allocation in User Space

User Space

- Reduce the number of requests to the OS

Drawbacks

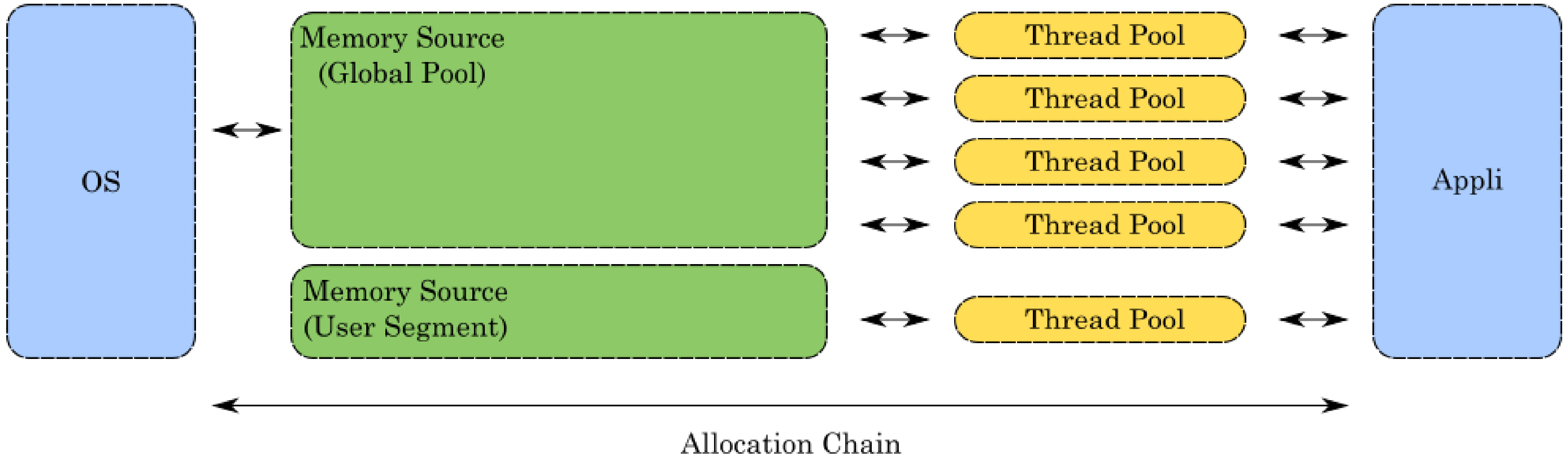
- Inefficient NUMA support
  - Data are allocated without NUMA knowledge
- Inefficient multithread application support
  - Contention issue if threads target the same memory area



Data Locality in multithreaded context

Data locality management with thread pools

- Thread pools handle local management of allocations
  - List of free blocs (coming from macro-blocs splitting)
  - First level for algorithms (decision, splitting, fusion)
- Exchange with Memory source via macro-blocs (> 2MB)
- Ensure local memory accesses
  - Avoid false sharing
  - Easy when threads are bound
  - Enable thread migration and keep data locality



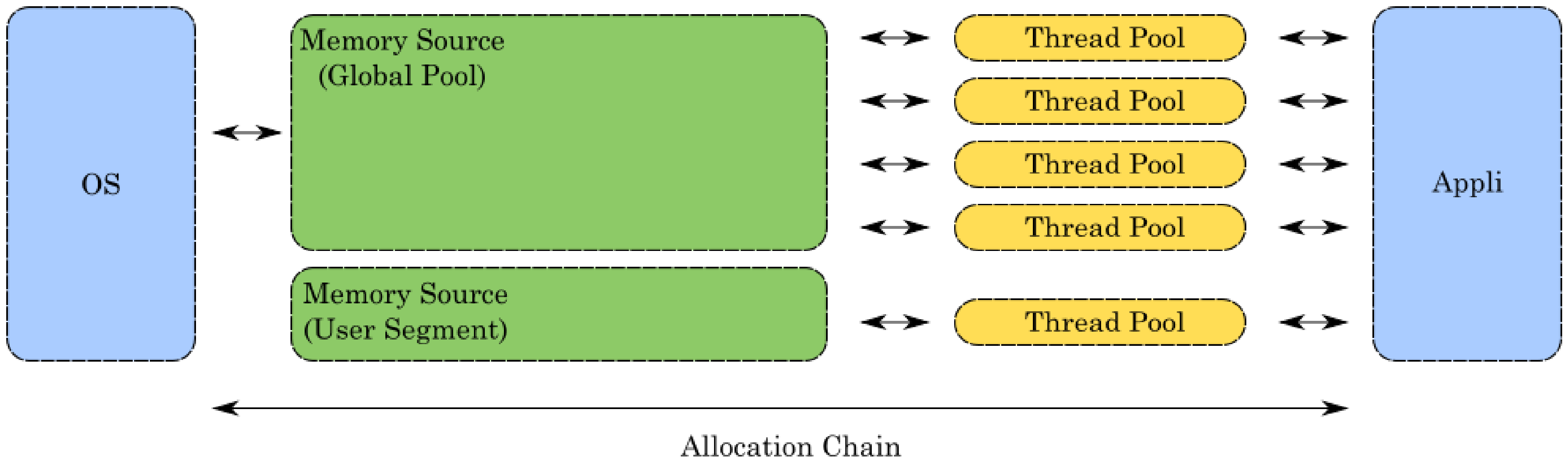
Data Locality in multithreaded context

Drawbacks

- Directly handle only allocation < 1 MB
  - Huge segment allocation are directly transferred to Memory Source
- Multiple threads with huge segment allocation will cause contention of Memory Source

Solution implemented in MPC allocator:

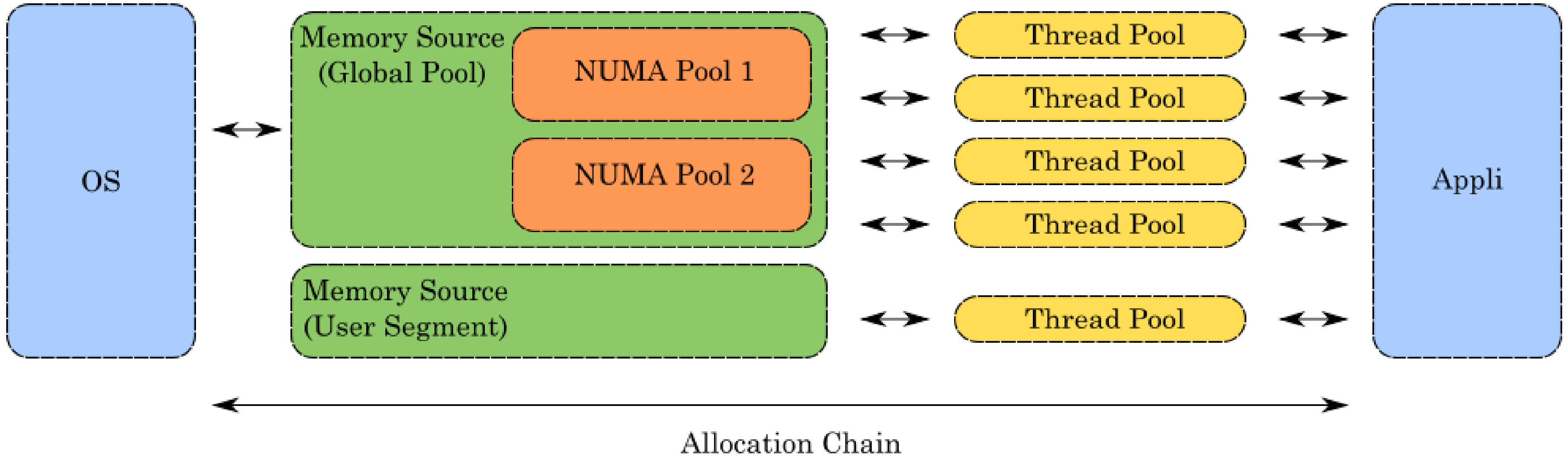
- Do not tackle directly the Global Pool
  - Levels of NUMA pools to avoid contention and NUMA effects with huge segments



Data Locality in multithreaded context

Keep data locality

- Efficient NUMA support
- Efficient highly multithread application support
- How to handle thread migration
  - Link between thread scheduler and memory allocator
  - Next touch policy
- Allow to handle concurrency more precisely on each NUMA level
- Allow page "recycling" between threads in the same NUMA level



Huge segments issues

Huge segment allocations still go through the OS

- Performance loss due to system calls
- For sizes > 1MB, allocation cost is driven by page-faulting

MPC solution

- The memory source is an allocator by itself
  - It keeps track of free macro-blocs for future reuse
    - If the segment is not too large
    - If the total amount of memory used for this macro-blocs cache is not tool large
  - Possibility to fuse adjacent macro-blocs to provide greater segments if necessary
- Use *mremap* to resize segments if no matching segment is available

Drawbacks

- Greater latency when the system call is still necessary
  - Set up of macro-blocs buffers for each NUMA Pool to avoid system calls
- Increase memory consumption
  - Possibility to adjust dynamically the amount of memory available for macro-blocs caches

AMR Code + MPC on Dual-Westmere (2*6 cores)

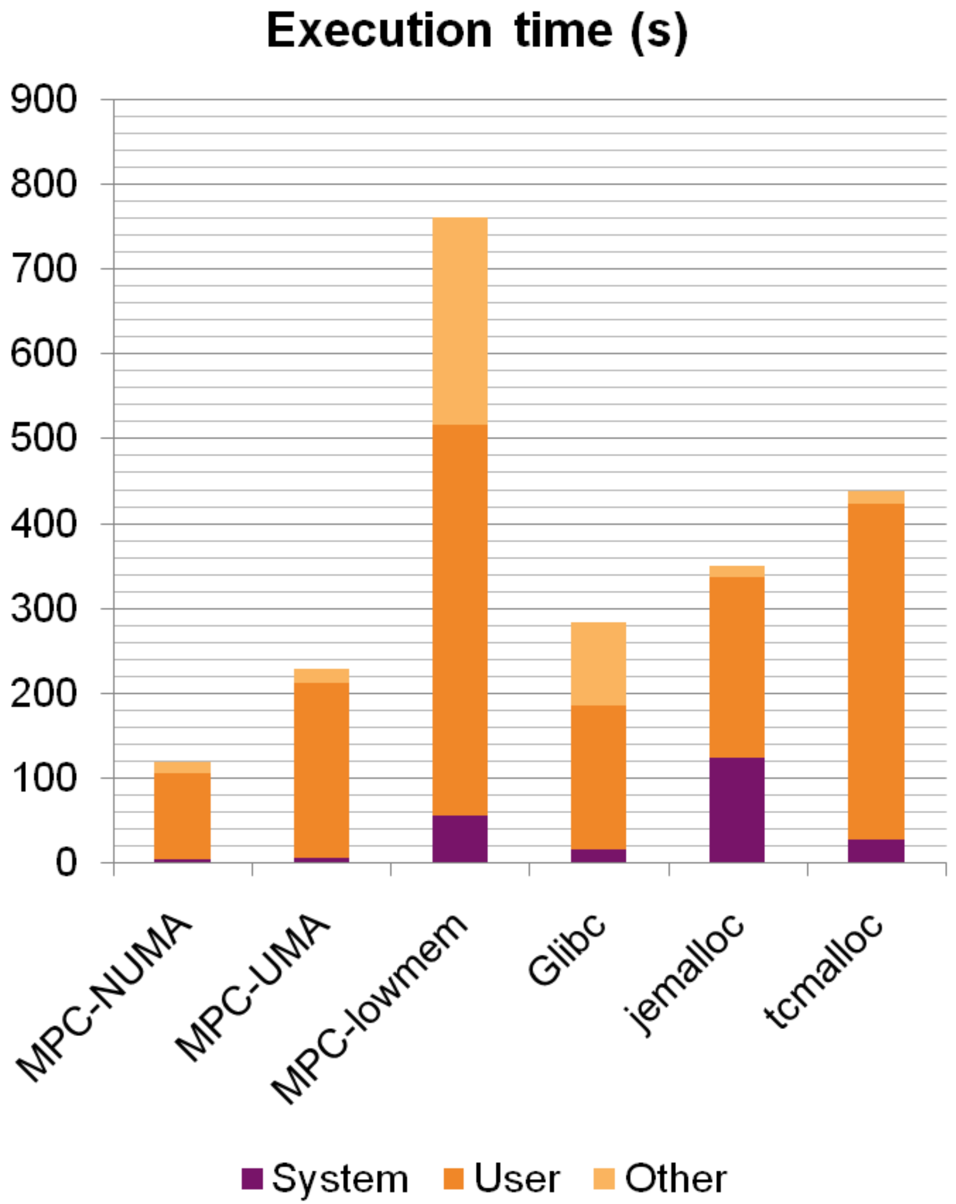
Standard **4K pages**

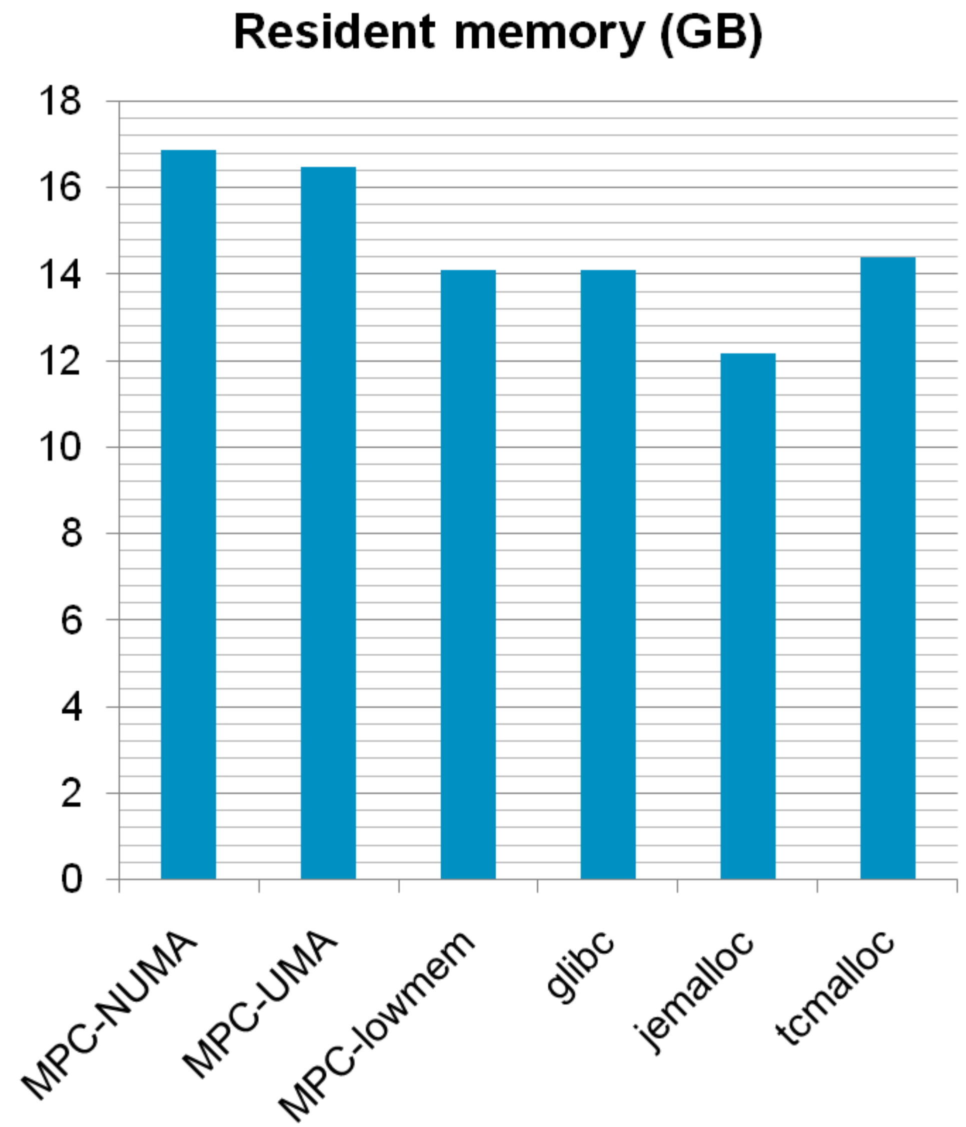
Allocator	Kernel	Total (s)	Sys. (s)	Mem. (GB)
MPC-NUMA	Std.	135.14	1.79	4.3
MPC-Lowmem	Std.	161.58	15.97	2.0
Jemalloc	Std.	143.05	14.53	1.9

Transparent Huge Pages

Allocator	Kernel	Total (s)	Sys. (s)	Mem. (GB)
MPC-NUMA	Std.	137.89	1.86	6.2
MPC-Lowmem	Std.	196.51	28.24	3.9
Jemalloc	Std.	144.72	14.66	2.5

AMR Code + MPC on Nehalem-EX (128: 4*4*8 cores)





Memory allocation

Memory Semantic

Malloc only modify virtual memory space

Requests via *mmap* or *brk* system calls

Lazy Memory Allocation: Page Fault

Physical pages are then provided upon a first touch policy via a page fault

Allocation cost is not limited to *malloc*

Cache usage is linked to physical pages allocation

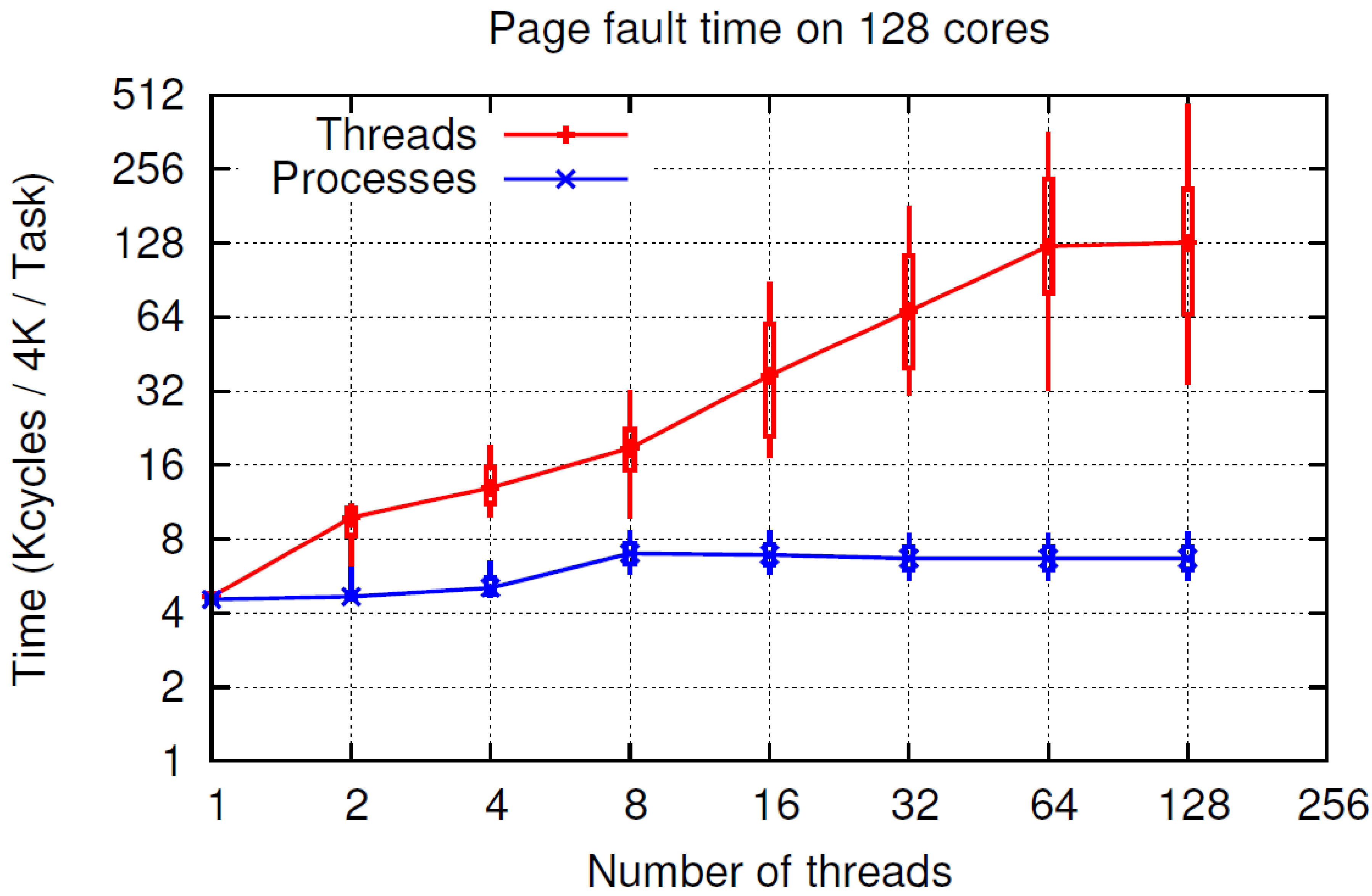
Reset memory pages

Security reason

Before providing page to the application

In kernel space

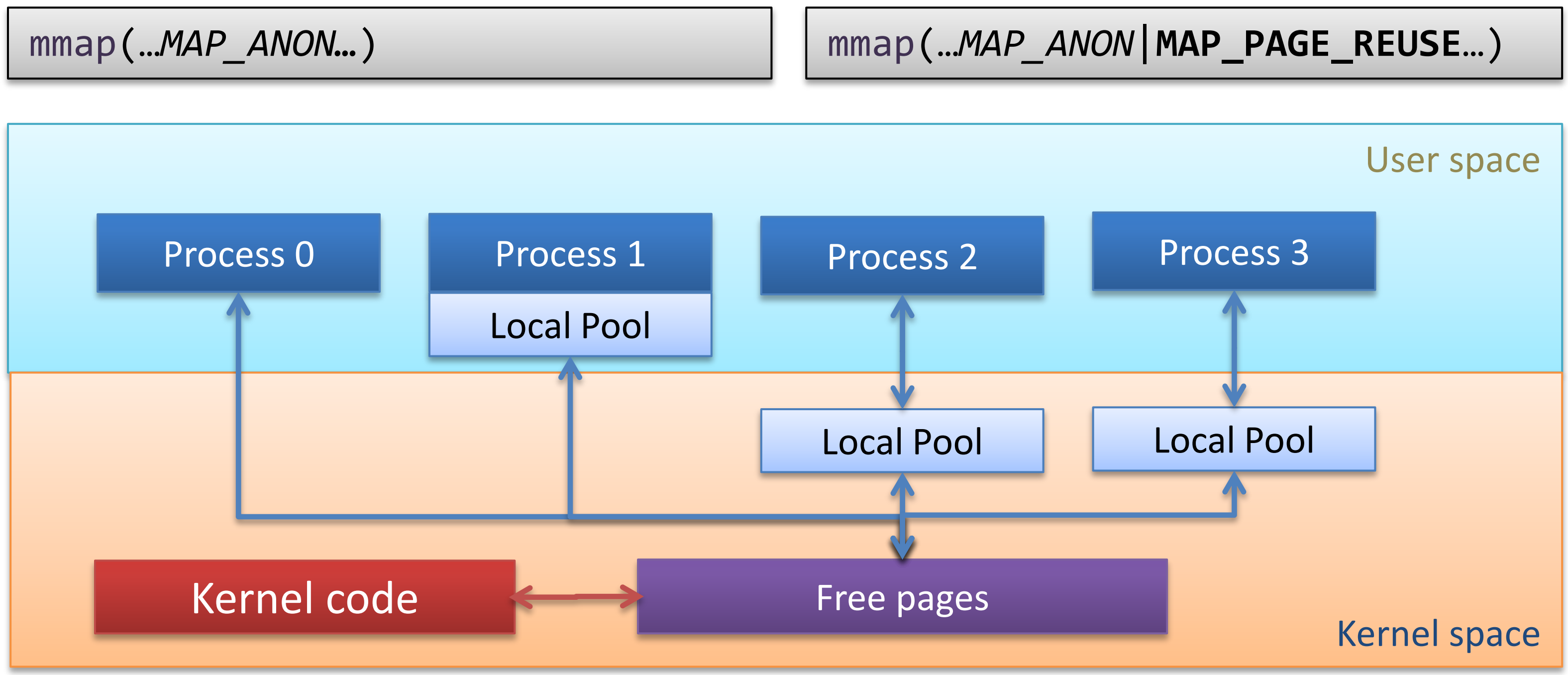
Page fault cost



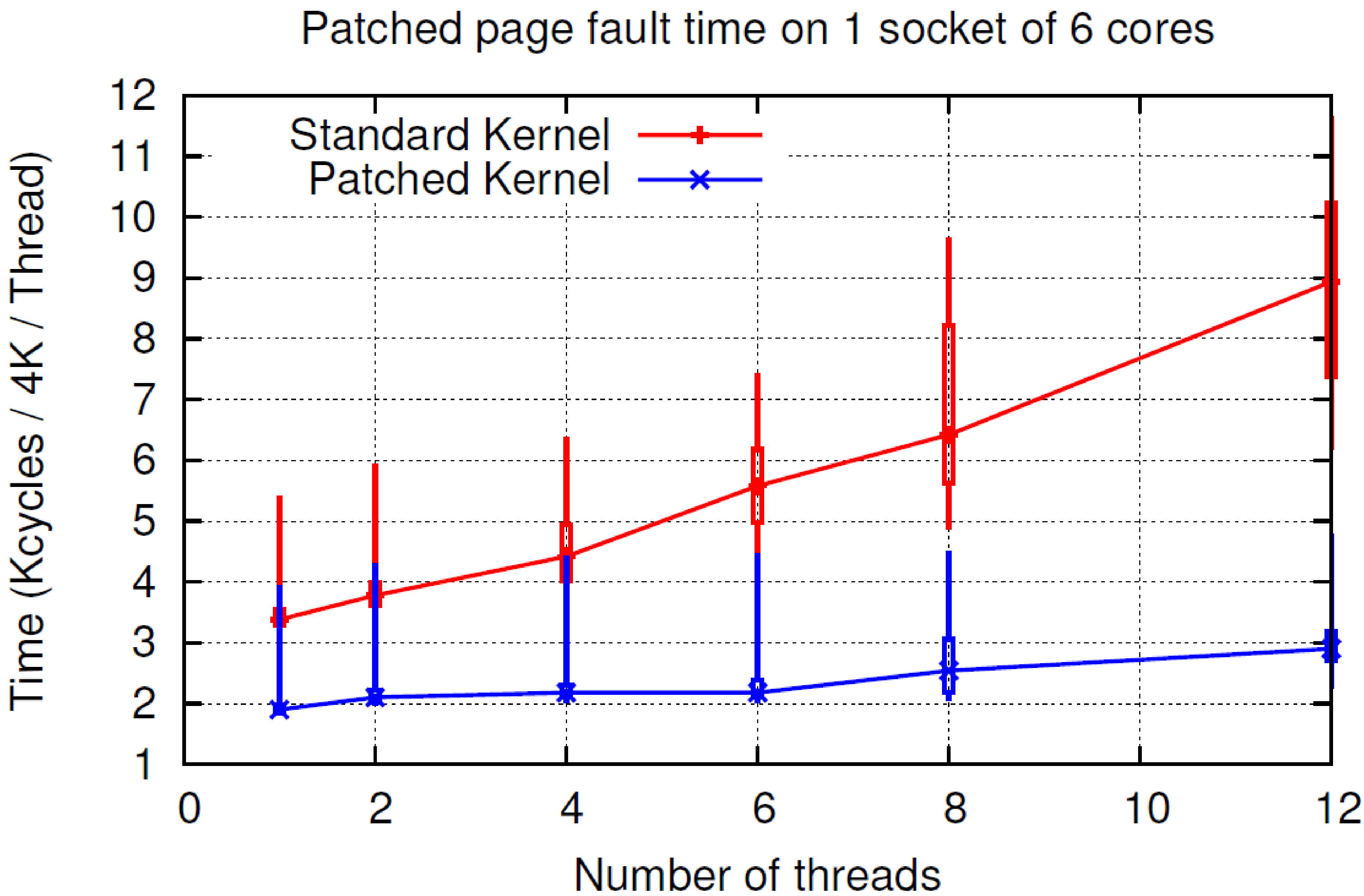
Page zoring

Kernel Space

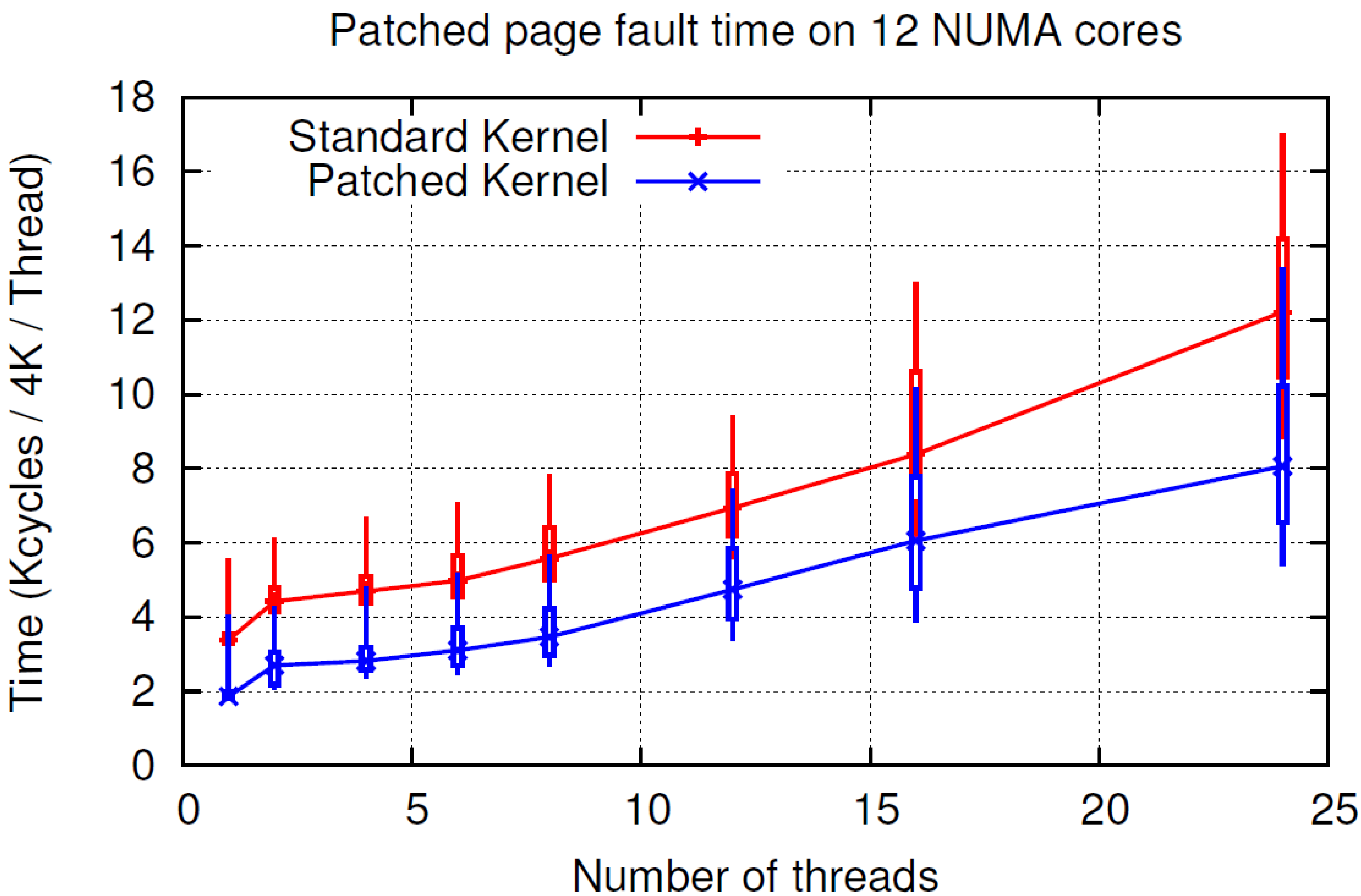
40% of page fault execution time is due to zero-page.
Zero-page is useless from the application point of view.
Reuse dirty pages within the same process to avoid page zeroing.
Kernel patch using an extension to the mmap system call.



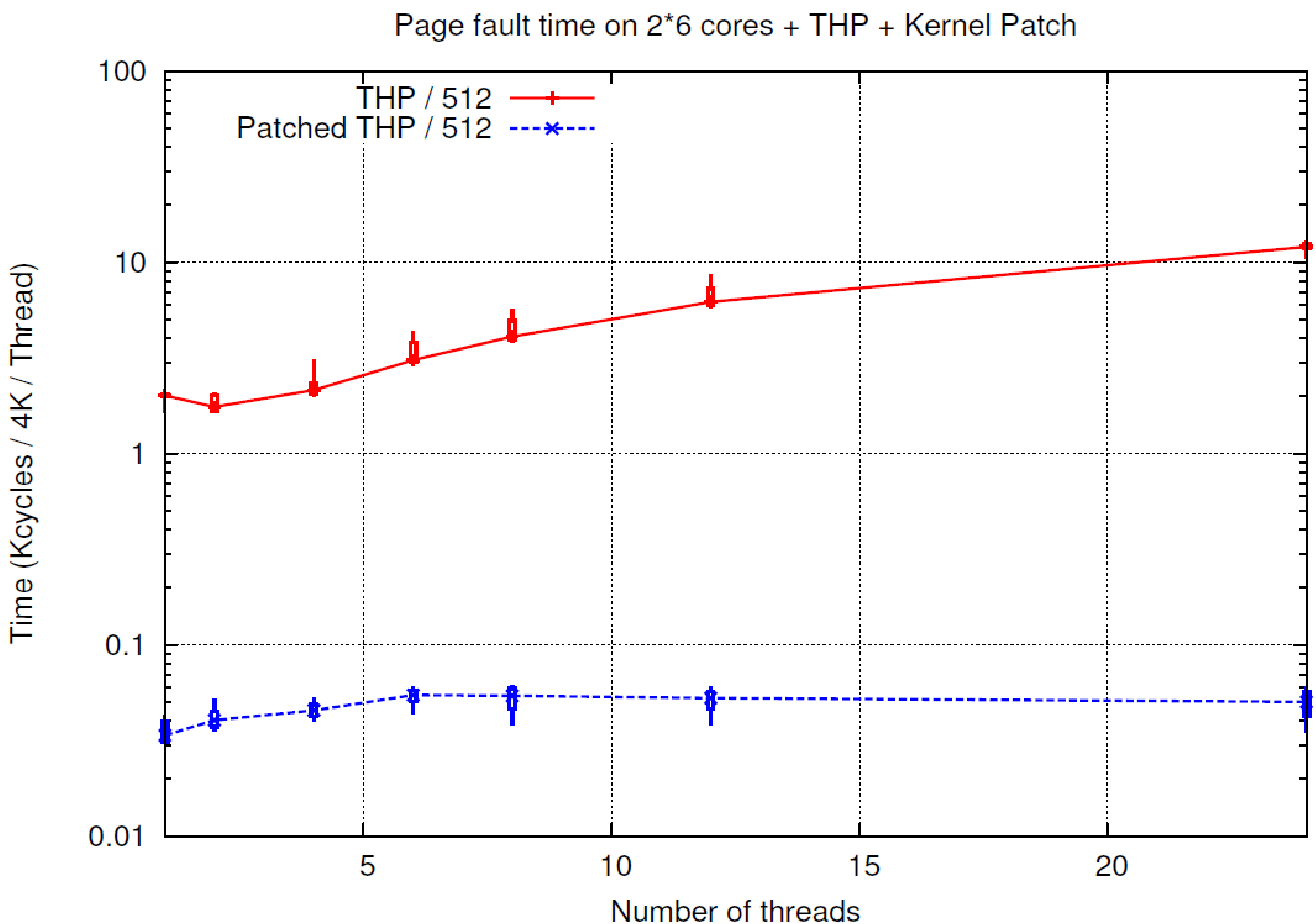
Page fault performance results



Page fault performance results



Page fault performance results (3/3)



MPC on Dual-Westmere (2*6 cores)

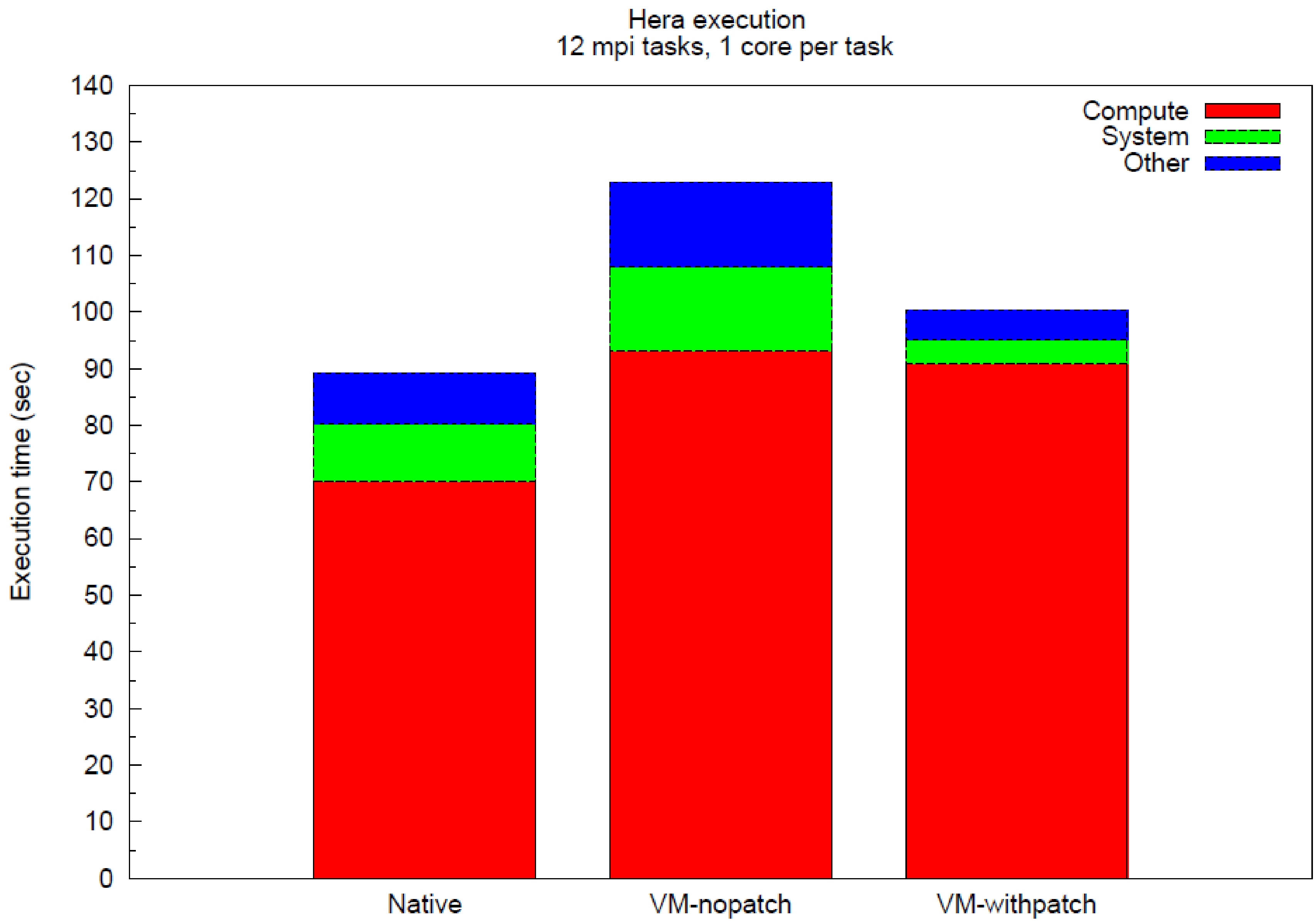
Kernel patch and standard **4K pages**

Allocator	Kernel	Total (s)	Sys. (s)	Mem. (GB)
MPC-NUMA	Std.	135.14	1.79	4.3
MPC-Lowmem	Std.	161.58	15.97	2.0
MPC-Lowmem	Patched	157.62	10.60	2.0
Jemalloc	Std.	143.05	14.53	1.9
Jemalloc	Patched	140.65	9.32	3.2

Kernel patch and **Transparent Huge Pages**

Allocator	Kernel	Total (s)	Sys. (s)	Mem. (GB)
MPC-NUMA	Std.	137.89	1.86	6.2
MPC-Lowmem	Std.	196.51	28.24	3.9
MPC-Lowmem	Patched	138.77	2.90	3.8
Jemalloc	Std.	144.72	14.66	2.5
Jemalloc	Patched	138.47	6.40	3.2

Results on Dual-Westmere (2*6 cores)



Physical page allocation

Linux

- Random page distribution

Linux Transparent Huge Pages

- Random page distribution
- Huge pages

OpenSolaris

- Page coloring
- Hash on virtual page addresses
- Without PID

FreeBSD

- Page coloring
- Hash on virtual page addresses
- Without PID
- Superpages

Physical page allocation

Experimental setup

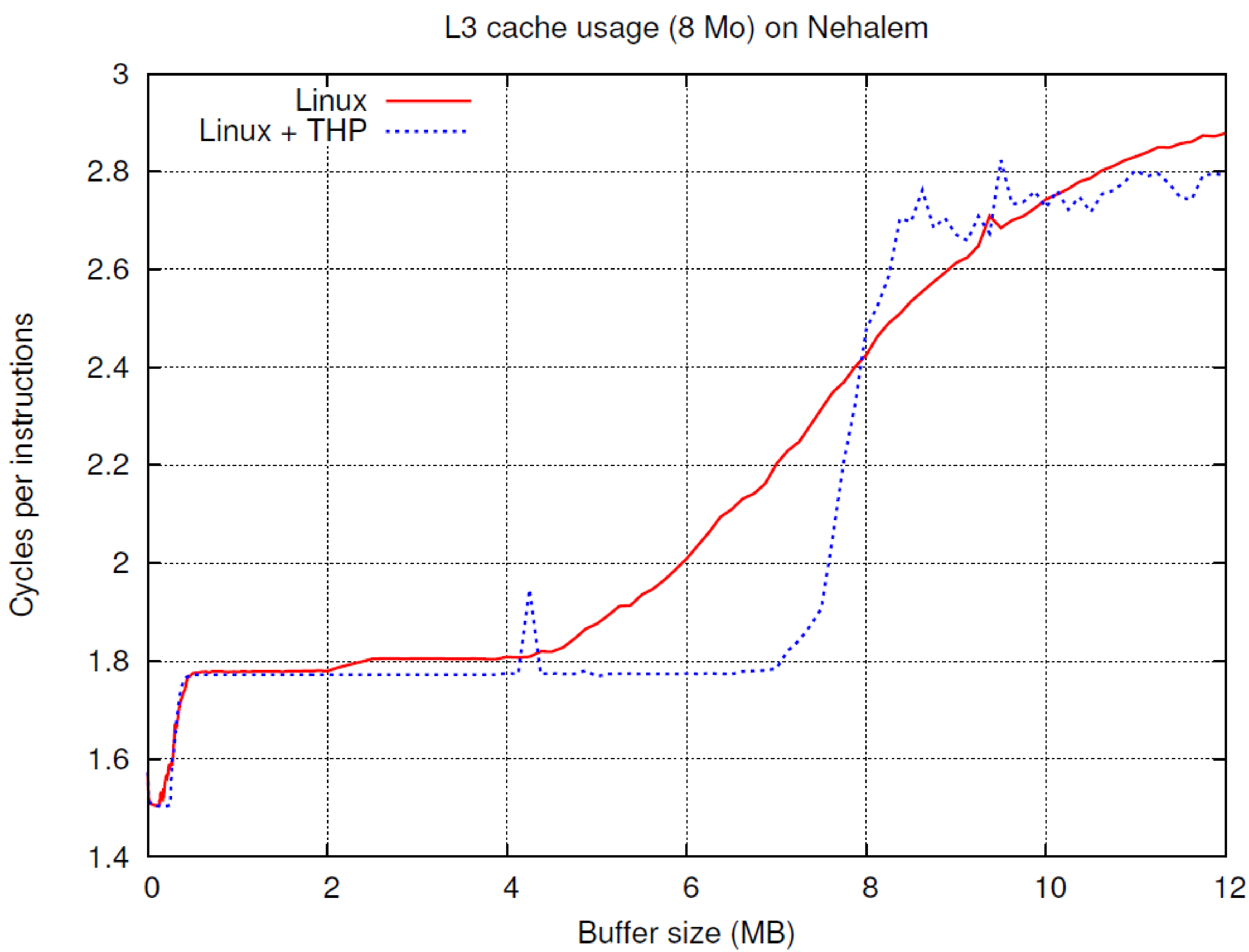
Processor Bi Intel Xeon-E5502 (quad core Nehalem)
Frequency 2.27 GHz
Cache L3 8 MB
Cache L2 256 KB
Cache L1 32 KB
Memory 24 GB

Software stack

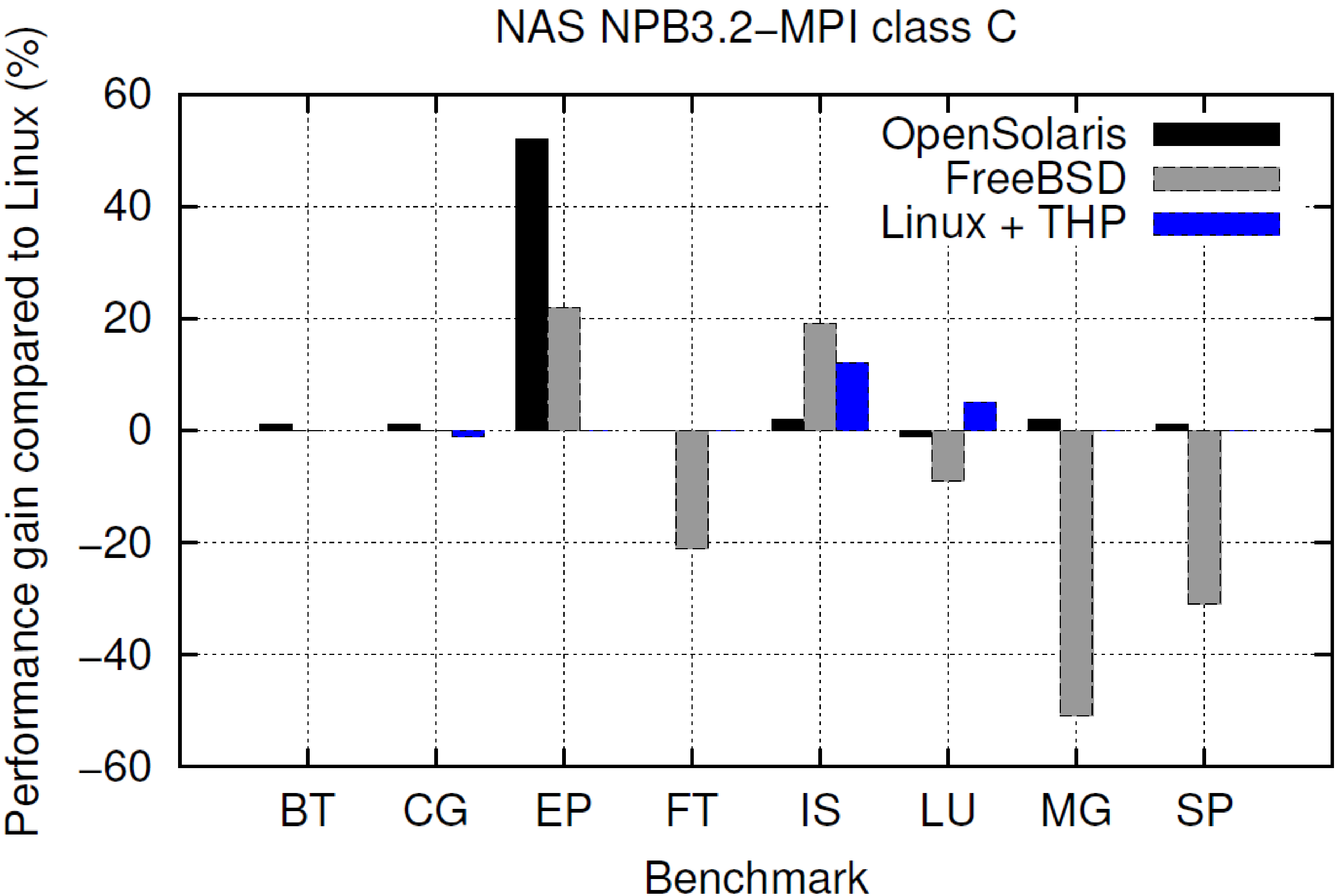
GCC 4.4.1
Identical software stack
Identical compilation options

Native execution

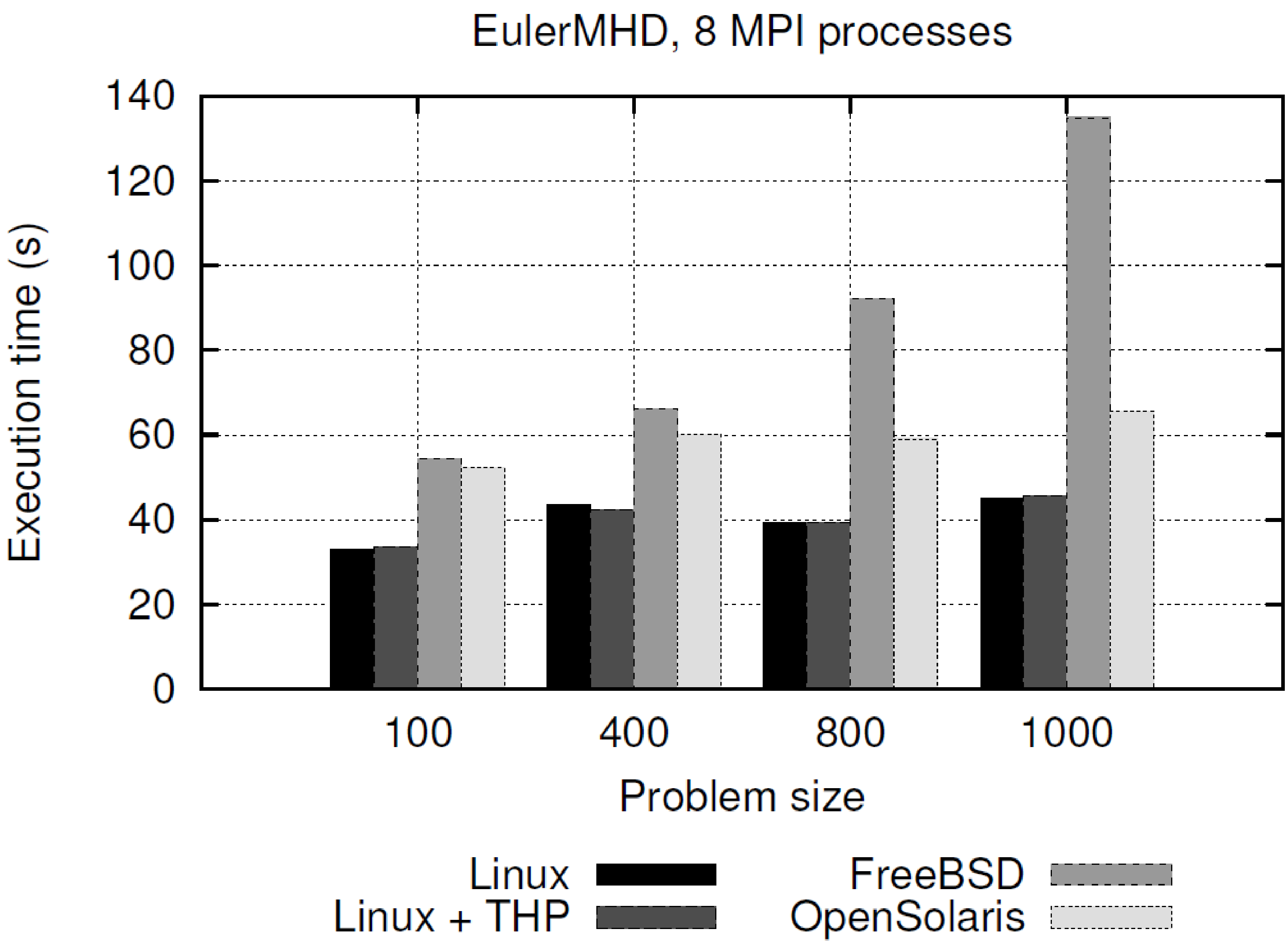
Cache usage


L3 cache usage (8 Mo) on Nehalem
Linux
Linux + THP
Cycles per instructions
Buffer size (MB)
3
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
0
2
4
6
8
10
12

Impact of page allocation policy



Impact of page allocation policy



Conclusion

Page zeroing

- Kernel-space patch in Linux 2.6.32 and 2.6.36
- Performance improvements
  - up to 45% on sequential page faults
  - up to 66% for 12 threads
- Limitation for standard usage (outside HPC)
  - No support for SWAP
  - Need pool cleaning method

Physical pages allocation

- Huge impact on execution time
  - Up to 51% performance improvement and 91% performances decrease
- POC: kernel module with contiguous physical pages allocation to improve cache usage