Systèmes d'exploitation

Chapitre 4 : Mémoires

Chargé de cours :
Emery Kouassi Assogba
Tél : 95 22 20 73
Emery.assogba@uac.bj/
emery.assogba@ptgfengineering.com

Hiérachie de mémoires

- Types de mémoires

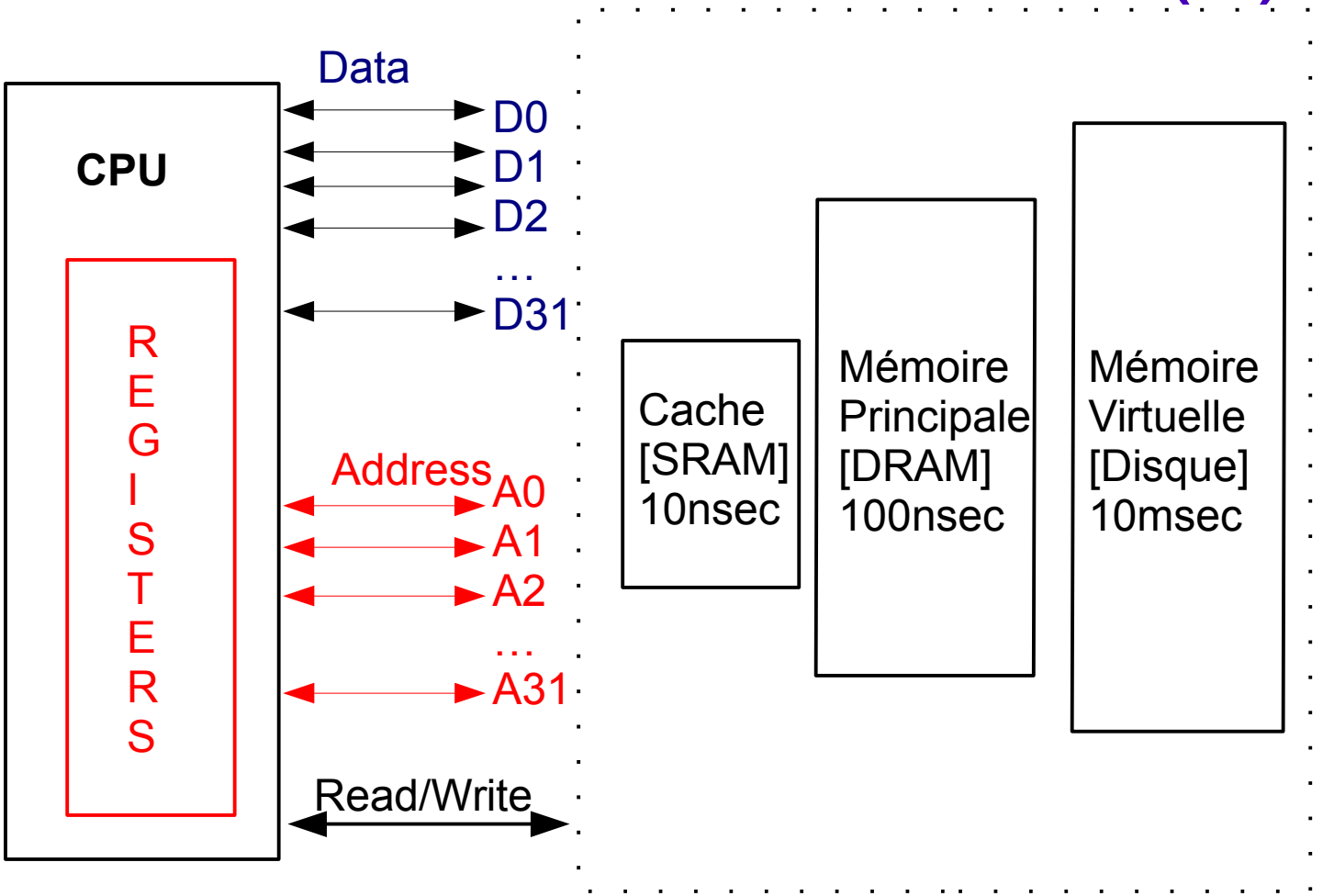
Type	Temps d'accès	Coût/MB
– SRAM	5-25 nsec	$100-$250
– DRAM	60-120 nsec	$5-$10
– disque dur	10-20 millions nsec	$0.1-$0.2

- Comment construire une mémoire d'ordinateur ?
  - uniquement des SRAMs
    - rapide, mais très couteux
  - uniquement des DRAMs
    - moins couteux mais moins rapide
  - uniquement des disques durs
    - bon marché, mais très lent

Hiérachie de mémoires (2)

- Solution
  - Utiliser les trois types de mémoire ensemble de façon à donner à l'utilisateur l'illusion d'une très vaste mémoire avec un très bon rapport coût/perf.
  - Combiner les trois types de mémoire en une profitant du principe de localité des programmes
    - localité temporelle
    - localité spatiale

Hiérarchie de mémoires (3)



- Objectif
  - L'accès aux différents niveaux de la hiérarchie doit
  - être le plus transparent possible pour le CPU

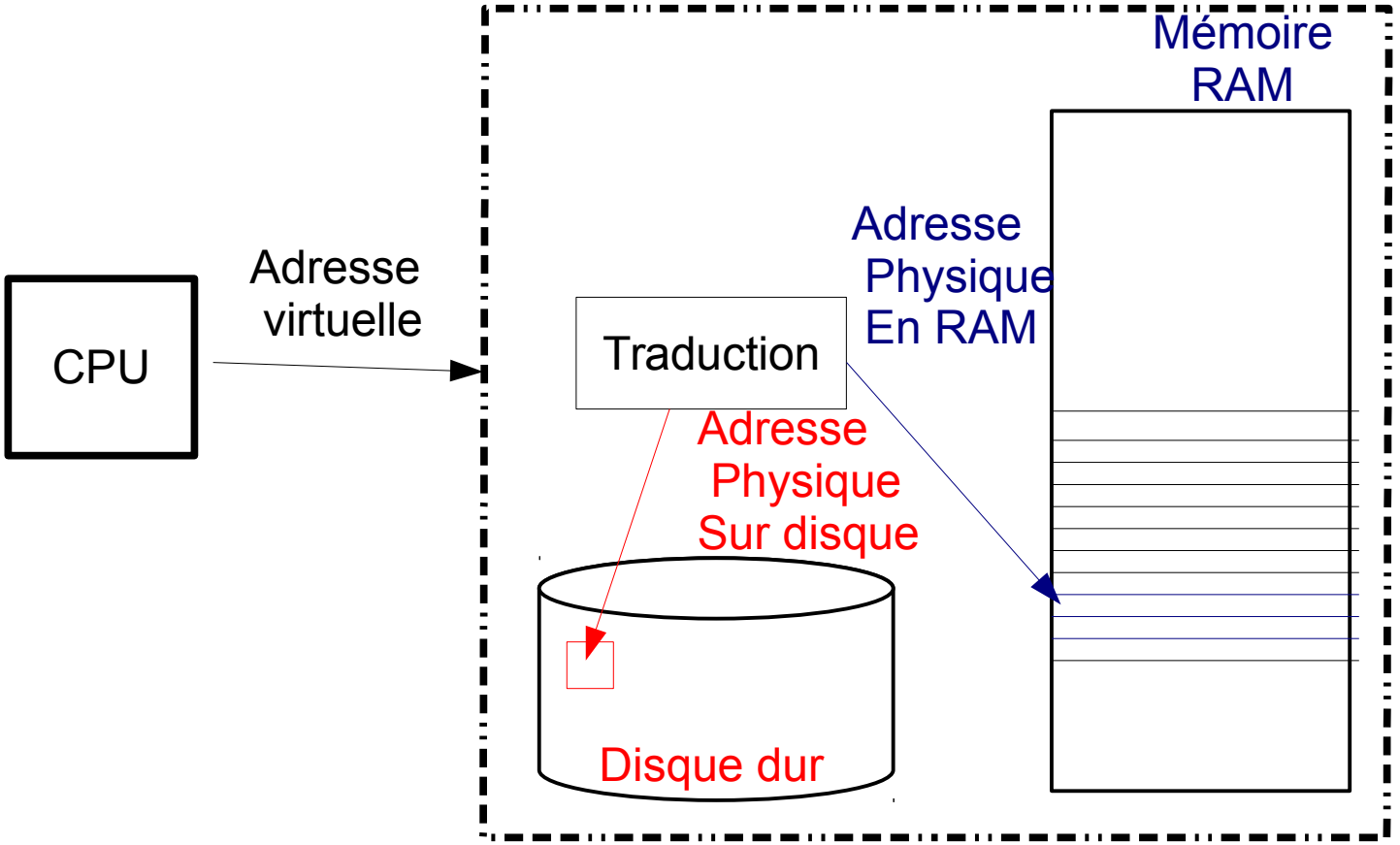
Mémoire cache

- Objectif
  - stocker dans une mémoire rapide les zones mémoire récemment utilisées par le CPU afin de bénéficier du principe de localité
- Accès mémoire avec une cache
  - consulter la cache pour voir si la zone mémoire demandée si trouve
    - si oui [hit] , fournir la donnée au CPU
    - si non [miss] , consulter la mémoire centrale pour obtenir la zone mémoire demandée

Accès à la mémoire cache

- En lecture
  - CPU fournit l'adresse demandée
    - Si la cache contient la zone correspondant à l'adresse demandée, elle répond directement
    - Sinon, la cache récupère en mémoire RAM la zone correspondant à l'adresse demandée
      - Si la cache est pleine, il faut trouver une place pour ces données !
- En écriture
  - Write through
    - Toute écriture se fait à la fois en RAM et en cache
      - Facile à implémenter, mais mauvaises performances
- Write back
    - Les écritures se font uniquement en cache
    - La RAM n'est mise à jour que lorsque l'on retire une information de la cache
      - Problèmes potentiels d'incohérences entre cache et RAM !

Mémoire virtuelle

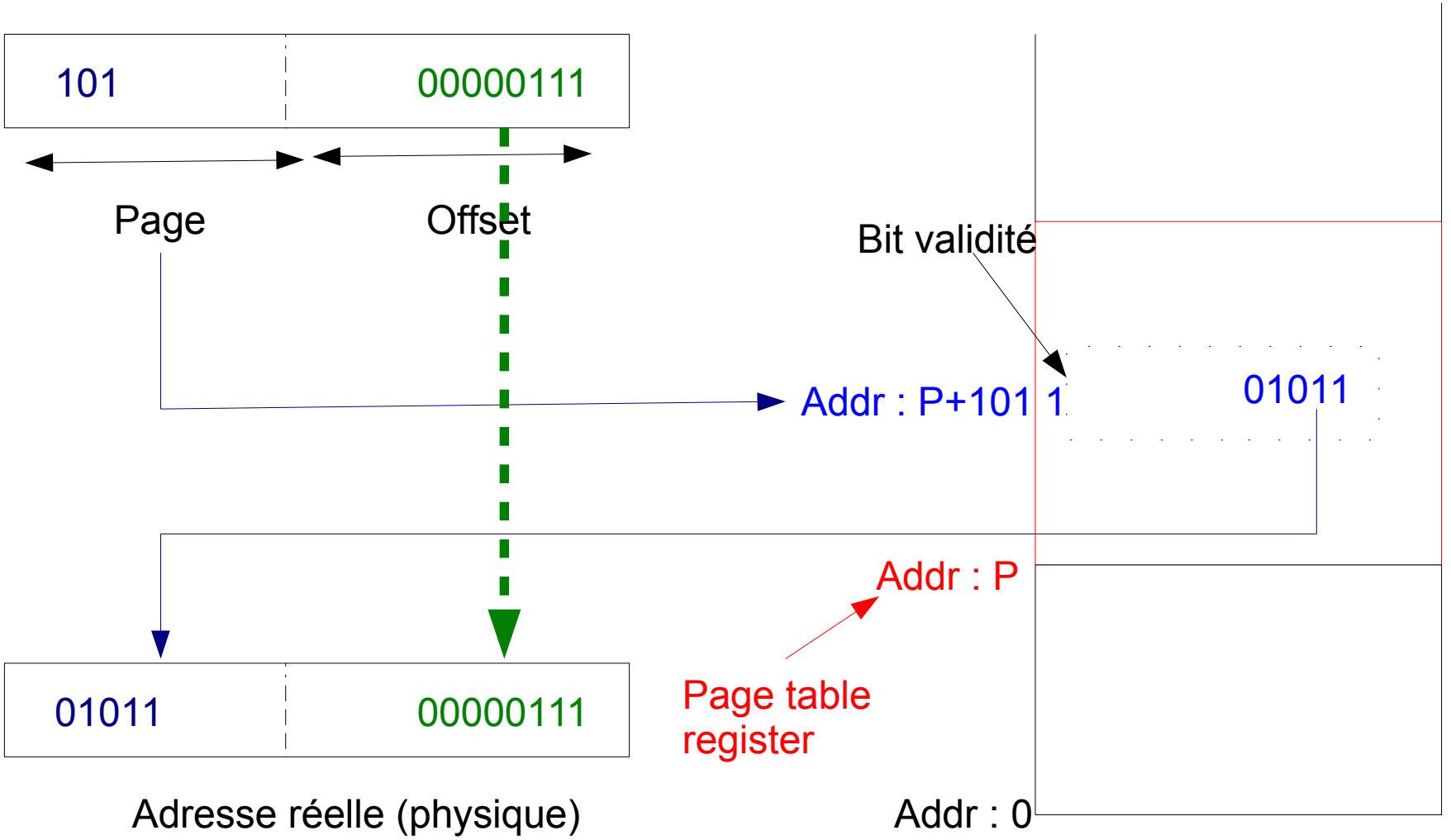


- Principe
  - l'information peut se trouver en RAM ou sur disque
  - le CPU y accède avec des adresses virtuelles

Mémoire virtuelle (2)

- Fonctionnement de la mémoire virtuelle
  - Disque dur
    - Mémoire de grande capacité
    - temps d'accès assez lent (10 millisecondes)
    - unité d'accès à l'information : secteur 512 octets ou +
- Mémoire RAM
    - Capacité moyenne
    - temps d'accès rapide par rapport au disque dur
    - unité d'accès à l'information : un mot de 32 bits
- Principes de base
    - le CPU utilise des adresses virtuelles qui sont
    - traduites en adresses physiques
    - diviser la mémoire virtuelle en pages
      - les pages fréquemment utilisées sont en RAM
      - les pages rarement utilisées sont sur disque dur
      - échanges fréquents entre le disque et la RAM

Une table des pages simple



Mémoire virtuelle : Lecture

- Accès à la mémoire virtuelle en lecture
  - décomposer l'adresse virtuelle en
    - numéro de page
    - offset
- consulter la table de pages à la ligne de cette page
    - si cette page est en mémoire, l'accès se fait en RAM grâce à l'adresse physique contenue dans la table des pages traduction via la table des pages
    - si cette page n'est pas en mémoire RAM, il faut la ramener en mémoire RAM
      - l'accès au disque est une opération complexe et lente
      - cette opération sera réalisée par le système d'exploitation
      - la donnée demandée pourra être utilisée lorsque la page complète aura été ramenée en mémoire RAM par l'OS
      - vu le coût d'accès au disque, il est intéressant d'éviter le plus possible d'accéder au disque dur

Mémoire virtuelle : Ecriture

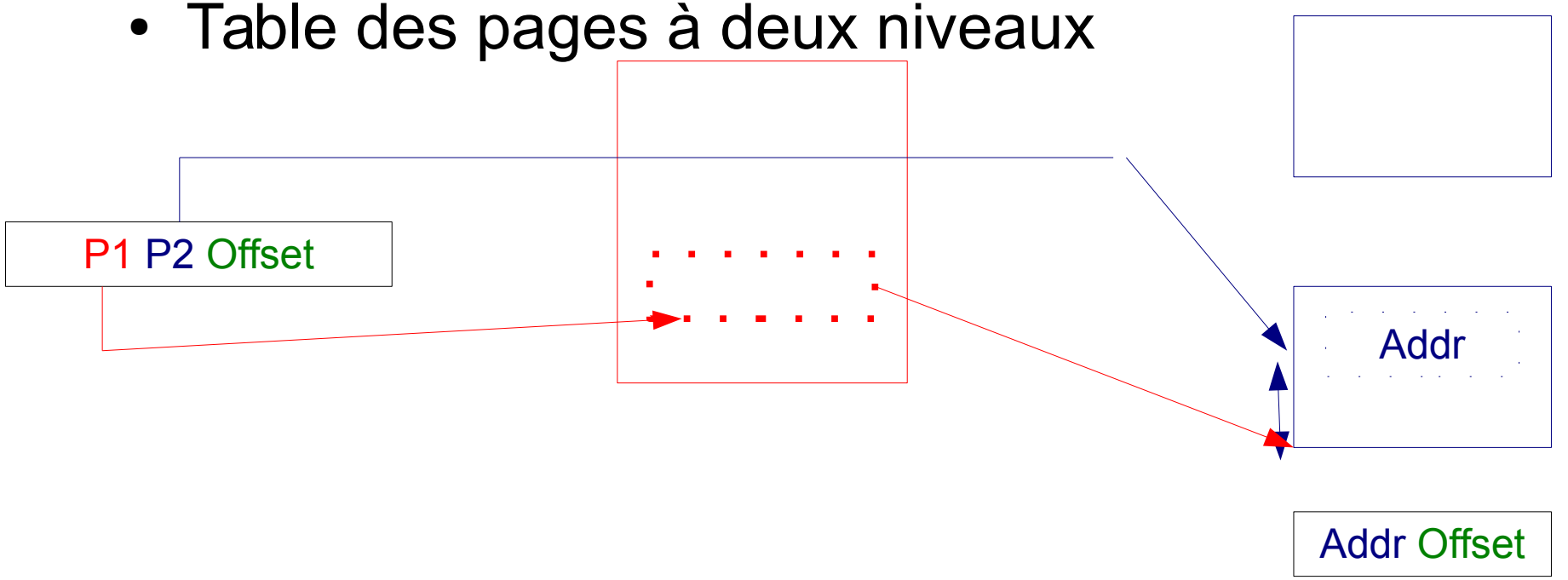
- Accès à la mémoire virtuelle en écriture
  - décomposer l'adresse virtuelle en
    - numéro de page
    - offset
  - consulter la table de pages à la ligne de cette page
    - si cette page est en mémoire, l'écriture se fait directement en RAM grâce à l'adresse physique contenue dans la table des pages
    - si cette page n'est pas en mémoire, il faut demander à l'OS de la ramener en mémoire avant de pouvoir y écrire
    - vu le temps d'accès au disque, le processus est suspendu
    - après écriture, la page modifiée sera conservée en RAM

Mémoire virtuelle et multitâche

- Comment permettre à plusieurs programmes de se partager la mémoire RAM ?
  - Offrir à chaque processus l'illusion qu'il dispose de toute la mémoire virtuelle
  - Au lancement d'un programme, l'OS créée une table des pages pour chaque processus qu'il lance
    - la localisation en mémoire RAM de la table des pages est indiquée dans un registre spécial (page table register)
      - page table registrer contient l'adresse de la table des pages du processus courant
      - lorsque l'OS passe d'un processus à l'autre, il met à jour le page table register

Améliorations pour les machines 64bits

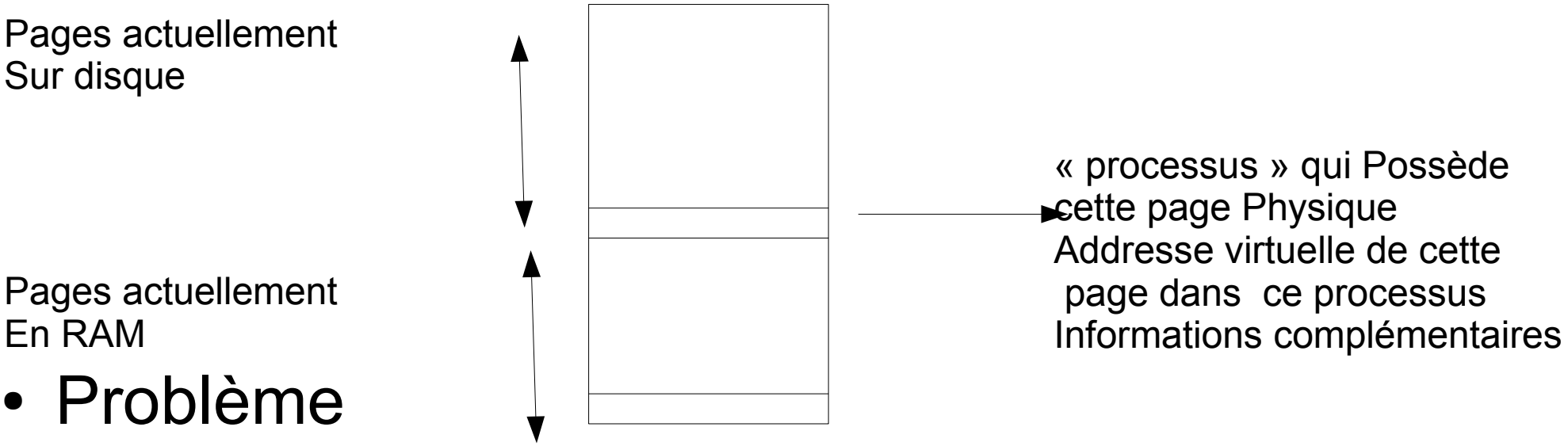
- Table des pages à deux niveaux



- Avantage
  - Seules les tables de second niveau qui pointent vers des pages utilisées existent
  - la plupart des lignes de la table des tables des pages pointent vers NULL

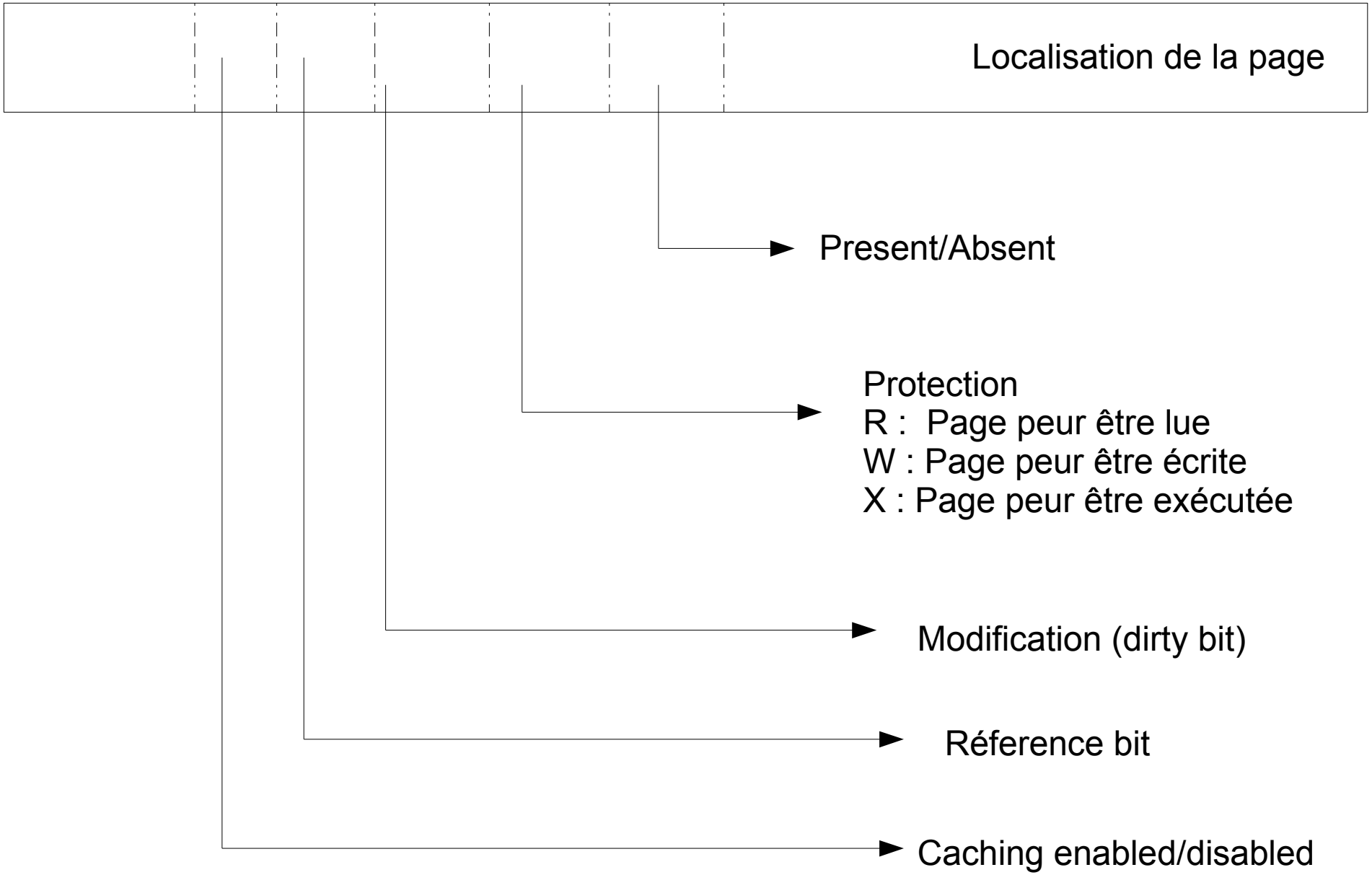
Améliorations pour les machines 64bits (2)

- Tables des pages inversées
  - contient une info pour chaque page physique



- Problème
  - Chaque traduction d'adresse nécessite une recherche !
  - recherche séquentielle dans la table est complexe
    - utiliser une table de hachage plutôt

Une entrée de la table des pages



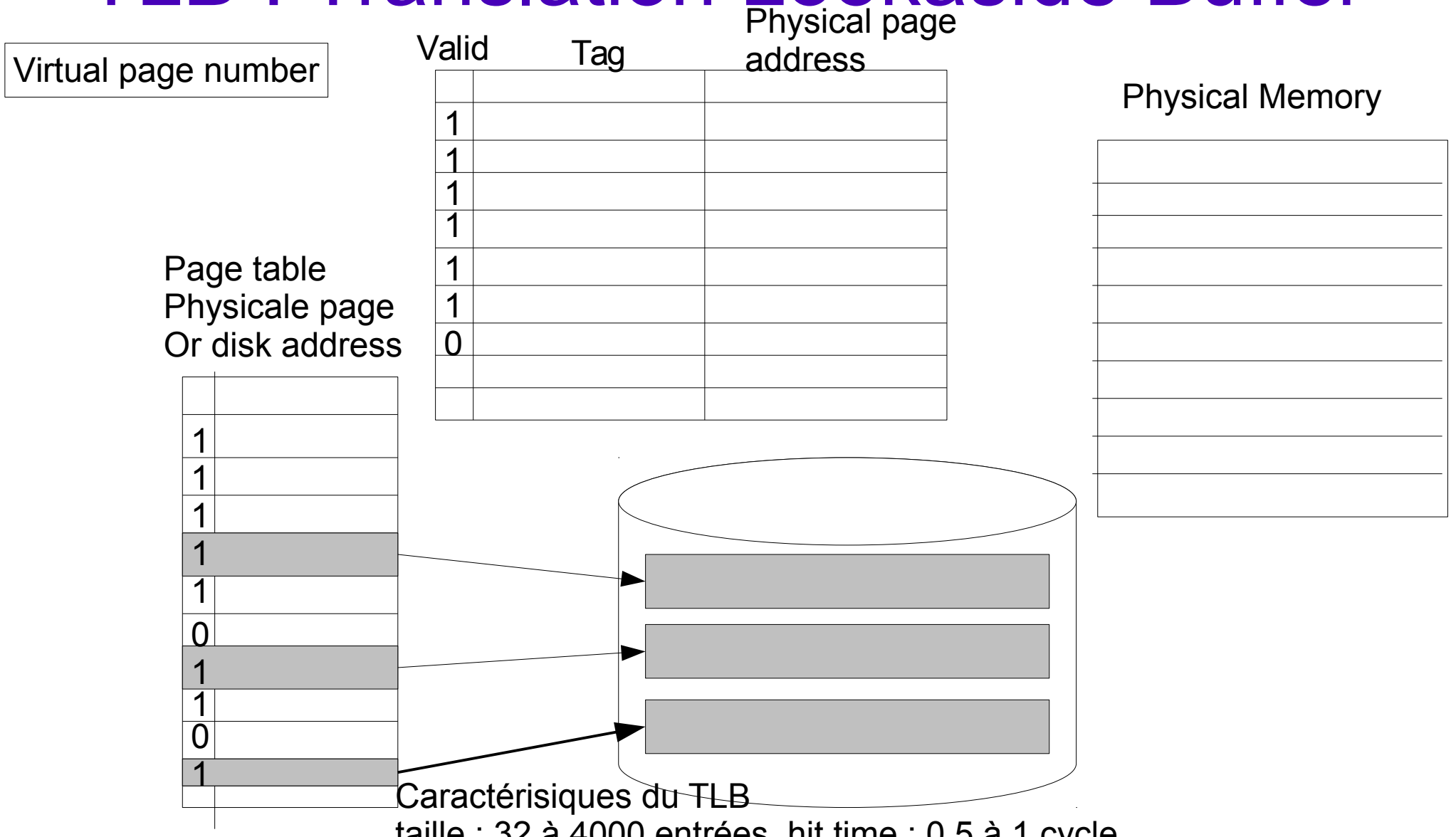
Mémoire virtuelle : remplacement

- Lorsque la RAM est pleine, il peut être nécessaire de sauver une page en RAM pour ramener une autre page du disque dur
  - Quelle page retirer de la RAM ?
    - Least Recently Used
      - difficile à implémenter exactement dans une table des pages
  - Approximation : use bit ou reference bit
    - mis à 1 par le hardware en cas d'accès à la page
    - mis périodiquement à 0 par l'OS
    - sélectionner pour remplacement une page dont le use bit est à 0
  - Comment sauver sur disque la page qui est en RAM?
    - Ecrire la page sur disque uniquement si elle a changé
      - dirty bit indiquant dans la table des pages qu'une page a été modifiée

Mémoire virtuelle : performance

- Quelles sont les performances d'une telle mémoire virtuelle ?
  - Comment accéder à l'adresse virtuelle x ?
    - accès à la table des pages pour avoir l'adresse physique
    - accès à l'adresse physique correspondante
  - deux accès mémoire au lieu d'un !
- Amélioration des performances
  - TLB : Translation-lookaside-buffer
    - mémoire cache spécialisée contenant une partie de la table des pages
    - Souvent mis à jour par l'OS et fonctionnant en write-back

TLB : Translation Lookaside Buffer



Caractérisiques du TLB
taille : 32 à 4000 entrées, hit time : 0.5 à 1 cycle
miss penalty : 10-30 cycles, Miss rate : 1-0.01%

Accès en mémoire avec le TLB

- Fonctionnement du TLB
  - Accès à l'adresse virtuelle x
    - consulter le TLB
      - TLB hit : l'adresse physique obtenue permet de faire l'accès mémoire
        - écriture : mettre le dirty bit à 1 dans la table des pages
        - écriture/lecture : mettre reference bit à 1 dans la table des pages
      - TLB miss : deux possibilités
        - la page est en mémoire, mais l'adresse n'est pas dans le TLB
          - interruption pour faire appel à l'OS et consulter la table des pages et mettre le TLB à jour
        - la page n'est pas en mémoire
          - interruption pour demander à l'OS de charger la page
          - lorsque la page est ramenée en RAM par l'OS, insérer l'adresse physique dans le TLB et faire l'accès mémoire
            - écriture : mettre dirty bit à 1
            - écriture/lecture : mettre reference bit à 1

Mémoire virtuelle : protection

- La table des pages contient de l'info critique
  - Comment contrôler l'accès à cette table ?
  - Les programmes normaux ne doivent pas y accéder
- Solution
  - deux modes de fonctionnement du CPU
    - mode utilisateur
      - mode dans lequel fonctionnent les programmes
    - mode superviseur/privilégié
      - mode dans lequel fonctionne l'OS
      - l'accès en écriture au page table register et au TLB n'est permis qu'en mode superviseur
  - mécanismes pour passer d'un mode à l'autre
  - utilisateur -> superviseur
    - syscall dans les processeurs MIPS, interruption logicielle
  - superviseur -> utilisateur
    - rfe dans les processeurs MIPS, fin d'interruption logicielle

Mémoire virtuelle : partage d'exécutable

- Problème
  - dans un système multi-utilisateurs, plusieurs instances du même exécutables peuvent se trouver en même temps en mémoire
- comment n'avoir qu'une seule copie en RAM ?
  - chaque processus dispose de sa table des pages
  - plusieurs adresses virtuelles peuvent pointer vers la même adresse physique
    - permet le partage de l'exécutable
    - chaque exécutable peut être mappé vers le fichier qui le
    - contient lors de l'appel système exec
  - comment supporter les écritures sur ces pages?
    - write access bit dans le TLB/table pages pour autoriser l'écriture individuellement à chaque page
    - l'OS doit gérer le write access bit pour permettre à certains processus de partager certaines zones de la mémoire
    - copy on write

Shared memory

- Comment permettre à deux processus de se partager explicitement de la mémoire ?
  - int shmget(key_t key, int size, int shmflg);
    - OS créée un descripteur de segment de mémoire partagée et réserve la mémoire nécessaire
      - Descripteur de segment de mémoire partagée
        - struct shmid_ds {
        - struct ipc_perm shm_perm; /* operation perms */
        - int shm_segsz; /* size of segment (bytes) */
        - time_t shm_atime; /* last attach time */
        - time_t shm_dtime; /* last detach time */
        - time_t shm_ctime; /* last change time */
        - unsigned short shm_cpid; /* pid of creator */
        - unsigned short shm_lpid; /* pid of last operator */
        - short shm_nattch; /* no. of current attaches */
        - };

Shared memory (2)

- Lorsqu'un segment de mémoire partagée a été créé, n'importe quel processus peut l'utiliser
  - pour autant qu'il connaisse son identifiant et possède les bonnes permissions
  - Appels système
    - void *shmat(int shmid, const void *shmaddr, int shmflg);
      - attache un processus à un segment existant
      - l'OS vérifie les permissions sur base du descripteur, modifie la table des pages du processus et éventuellement fixe la valeur du bit de protection
    - int shmdt(const void *shmaddr);
      - détache un segment de mémoire partagée d'un processus

Fork, exec, exit

- Exemple
- Processus père effectue fork
  - OS modifie la table des processus et crée une table des pages pour le fils
    - toutes les lignes de la table des pages du fils pointent vers celles du père et sont en copy-on-write
      - OS met à jour la liste des pages physiques de façon à savoir qu'une page est utilisée par plusieurs processus
        - liste des processus qui utilisent chaque page
        - nombre de processus utilisant chaque page
      - les segments de mémoire partagée éventuels sont aussi accessibles depuis le fils
        - implicitement, le fils fait donc un shmat

Fork, exec, exit(2)

- Processus fils effectue exec
  - OS supprime dans la table des pages du fils toutes les références vers l'ancien processus
    - il doit également mettre à jour la liste des pages physiques si copy-on-write est utilisé
    - si il y avait des segments de mémoire partagée, il faut les détacher
  - OS localise l'exécutable sur disque et modifie la table des pages de façon à pointer vers cet exécutable
    - les pages sont marquées avec le bit de validité à faux
      - seules les pages utilisées par le processus seront chargées
  - OS crée le stack initial (arguments et env) et met à jour la table des pages du processus fils

Fork, exec, exit(3)

- Processus fils effectue exit
  - OS libère les segments de mémoire partagée qui ont été attachés par le processus fils
    - l'OS fait donc shmdt à la place du fils si nécessaire
  - OS parcoure la liste des pages physiques pour
    - marquer comme libres les pages que seul le processus fils utilisait
    - supprimer le processus fils de la liste des utilisateurs des pages partagées
  - OS supprime la table des pages du processus qui se termine
    - les pages correspondantes en mémoire sont libérées
  - OS met à jour la table des processus et fournit la valeur de retour du fils à son père

Remplacement de pages

- Que faire lorsqu'un processus a besoin d'une page supplémentaire en RAM ?
  - Il reste des pages libres en RAM
    - Cela nécessite que l'OS maintienne une table avec la liste des pages libres et utilisées en RAM
    - L'OS choisit une page et met à jour la table des pages du processus
  - Il ne reste plus de page libre en RAM
    - L'OS doit sélectionner une ou plusieurs pages qu'il peut déplacer sur le disque
    - Pour cela, l'OS doit maintenir la liste des pages actuellement en mémoire RAM et avoir des statistiques sur leur utilisation
    - Le processus est interrompu tant que l'OS n'a pas pu libérer de place dans la mémoire RAM

Remplacement de pages:FIFO

- Objectif
  - Obtenir un mécanisme de remplacement de pages facile à implémenter sans support du hardware
- Implémentation
  - OS maintient une queue avec l'ensemble des pages actuellement en mémoire
  - Lorsqu'une page est créée ou ramenée en mémoire, elle est placée en fin de cette queue
  - Lorsqu'il faut remplacer une page, l'OS sélectionne celle qui est en tête de la queue
- Inconvénient
  - Mauvaises performances car une page utilisée en permanence risque d'aller sur le disque

Remplacement de pages:second chance

- Objectif
  - Améliorer la stratégie FIFO en ajoutant un bit de Reference supporté par le hardware
- Implémentation
  - OS maintient une queue avec l'ensemble des pages actuellement en mémoire
  - Lorsqu'une page est créée ou ramenée en RAM, elle est mise en fin de cette queue et R=0
  - Lorsqu'il faut remplacer une page, l'OS examine celle qui est en tête de la queue
    - Si R=0, la page est placée sur disque
    - Si R=1, la page est remise en fin de queue, R est remis à 0 et on examine la page suivante dans la queue
- Amélioration
  - Clock

Remplacement de pages Not Recently Used

- Objectif
  - Supprimer de la RAM une page qui n'a pas été récemment utilisée en supposant qu'elle ne le sera pas non plus prochainement
- Implémentation
  - Referenced et Modified bits dans table pages
  - A chaque accès, hardware met à jour R et M
  - Lorsque processus démarre, R=0 et M=0 partout
  - Périodiquement, l'OS met R=0 pour toute page
  - Lorsqu'il faut libérer une page, OS choisit une dans la classe la plus basse
    - Classe 0 : pages non référencées et non modifiées
    - Classe 1: pages non référencées mais modifiées
    - Classe 2: pages référencées mais non modifiées
    - Classe 3 : pages référencées et modifiées

Remplacement de pages Not Frequently Used

- Objectif
  - Supprimer des pages qui n'ont pas été fréquemment utilisées
- Principe
  - Ajouter dans la table des pages un compteur
  - Régulièrement l'OS incrémente le compteur de la valeur du bit R associé à la page
  - Les pages avec le compteur le plus faible sont choisies en cas de manque de pages
  - Amélioration
    - A chaque incrémentation du compteur, décaler la valeur du compteur à droite d'un bit et insérer le bit R comme bit de poids fort
    - permet d'éviter d'une ancienne page inutilisée ne reste en RAM

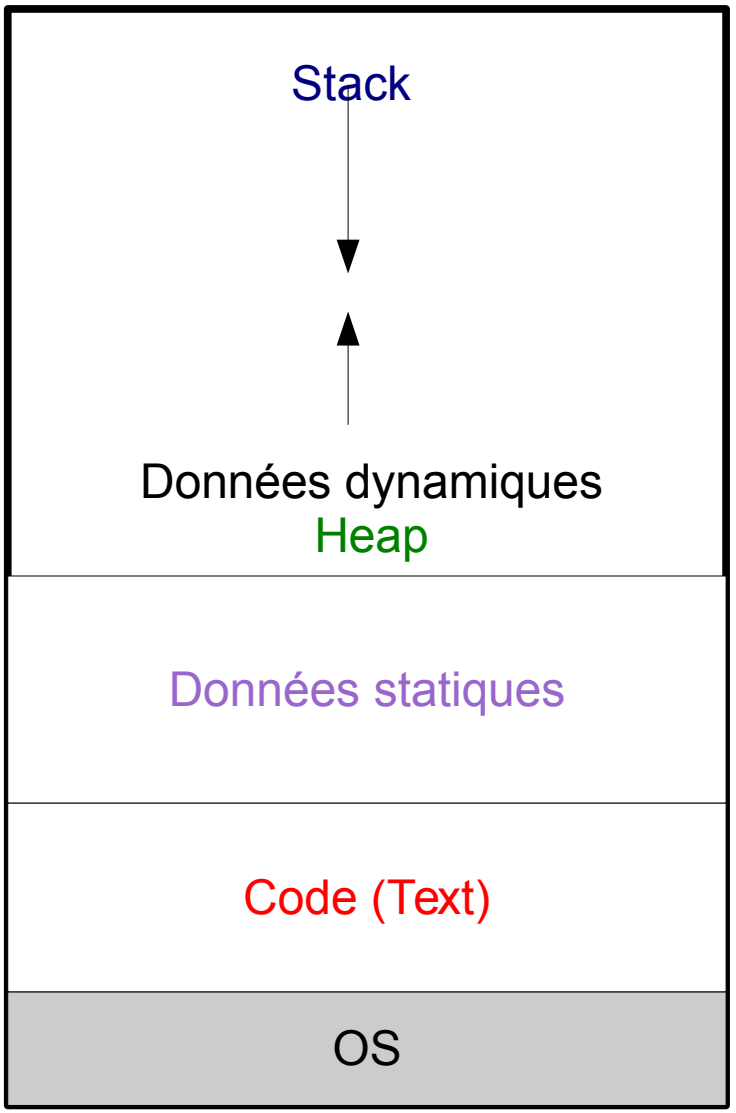
Swapping

- Objectif
  - Libérer plus de mémoire RAM en sauvegardant toutes les pages de processus inactifs sur disque
- Implémentation
  - Lorsque la mémoire RAM est fort remplie, régulièrement l'OS
    - inspecte la liste des processus dans l'état blocked
    - Si un processus est bloqué depuis longtemps, par exemple plusieurs dizaines de secondes
      - sauvegarder toutes les pages du processus sur disque
      - la table des pages du processus peut également être placée sur le disque dur
    - Si il n'y a pas de processus bloqué et que le manque de pages est important, on peut quand même swapper
    - Pour cela, il est nécessaire de maintenir des statistiques dans la table des processus

Example of implementation mistakes Buffer overflow

- The problem
  - For performance reasons, C and C++ do not perform bound checking when using arrays
  - Programmers do not always correctly use library functions in the standard C library
- Example
  - char *strcpy(char *dest, const char *src);
    - The strcpy() function copies the string pointed to by src (including the terminating `\0' character) to the array pointed to by dest. The strings may not overlap, and the destination string dest must be large enough to receive the copy.
    - if the dest array is smaller than the src array, memory beyond dest will be overwritten
  - Safer alternative
    - char *strncpy(char *dest,const char *src,size_t n);

Example of implementation mistakes Buffer overflow (2)



Contient les zones mémoires
Allouées par malloc

Contient les données initialisées
Au démarrage du programme
-Chaîne de car. À afficher
- tableaux globaux

Contient le code du programme
En langage machine

Example of implementation mistakes Buffer overflow on the stack

- Information stored on the stack
  - Local arrays and local variables of functions
  - return addresses
- Example of bad code

```
/* a simple buffer overflow with strcpy */
void f() {
unsigned char *in="A long message.....";
unsigned char out[5];
strcpy(out,in); /* use strncpy instead */
}
int main(int argc, char **argv) {
f();
printf("done\n");
}
```

Example of implementation mistakes Buffer overflow on the heap

- Information stored on the heap
  - Any type of dynamically allocated memory
    - arrays, strings, integers, structures, sometimes pointers to functions
- Example

```
char *gin="A long message.....";
char *msg, *gout;
int main(int argc, char **argv){
gout=(char *)malloc(5*sizeof(char));
msg=(char *)malloc(1*sizeof(char));
*msg='A';
strcpy(gout,gin); /* use strncpy instead */
printf("msg:%c\n",*msg);
}
```

- ./a.out
- msg:.

Can you spot problems ?

- A simple phonebook to be used by students

```
/* phone – expects name=foo value on STDIN */
static char cmd[128];
static char format[] = “grep %s phone.list\n”;
int main(int argc, char *argv[])
{
char buf[256];
gets(buf);
sprintf(cmd,format,buf+5);
write(1,”Phone number is ........\n\n”,27);
system(cmd);
}
```

Is it secure ?

Can you spot problems ?

- A simple phonebook to be used by students

```
/* phone – expects name=foo value on STDIN */
static char cmd[128];
static char format[] = “grep %s phone.list\n”;
int main(int argc, char *argv[])
{
char buf[256];
gets(buf);
sprintf(cmd,format,buf+5);
write(1,”Phone number is ........\n\n”,27);
system(cmd);
}
```

Is it secure ?