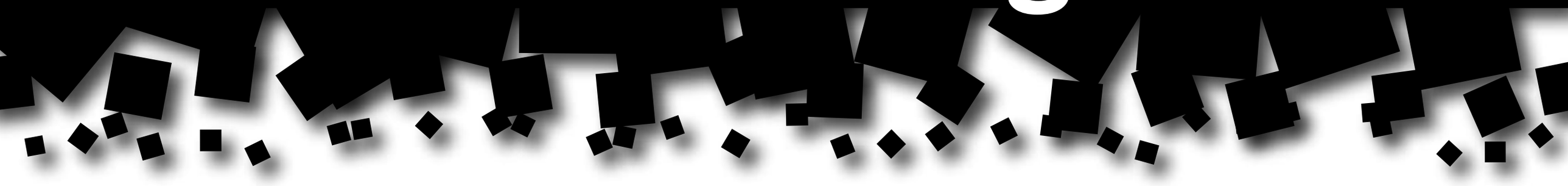
M2101 - Architecture 2

Cours n°5

Processus légers (threads)

Victor Poupet

Processus légers



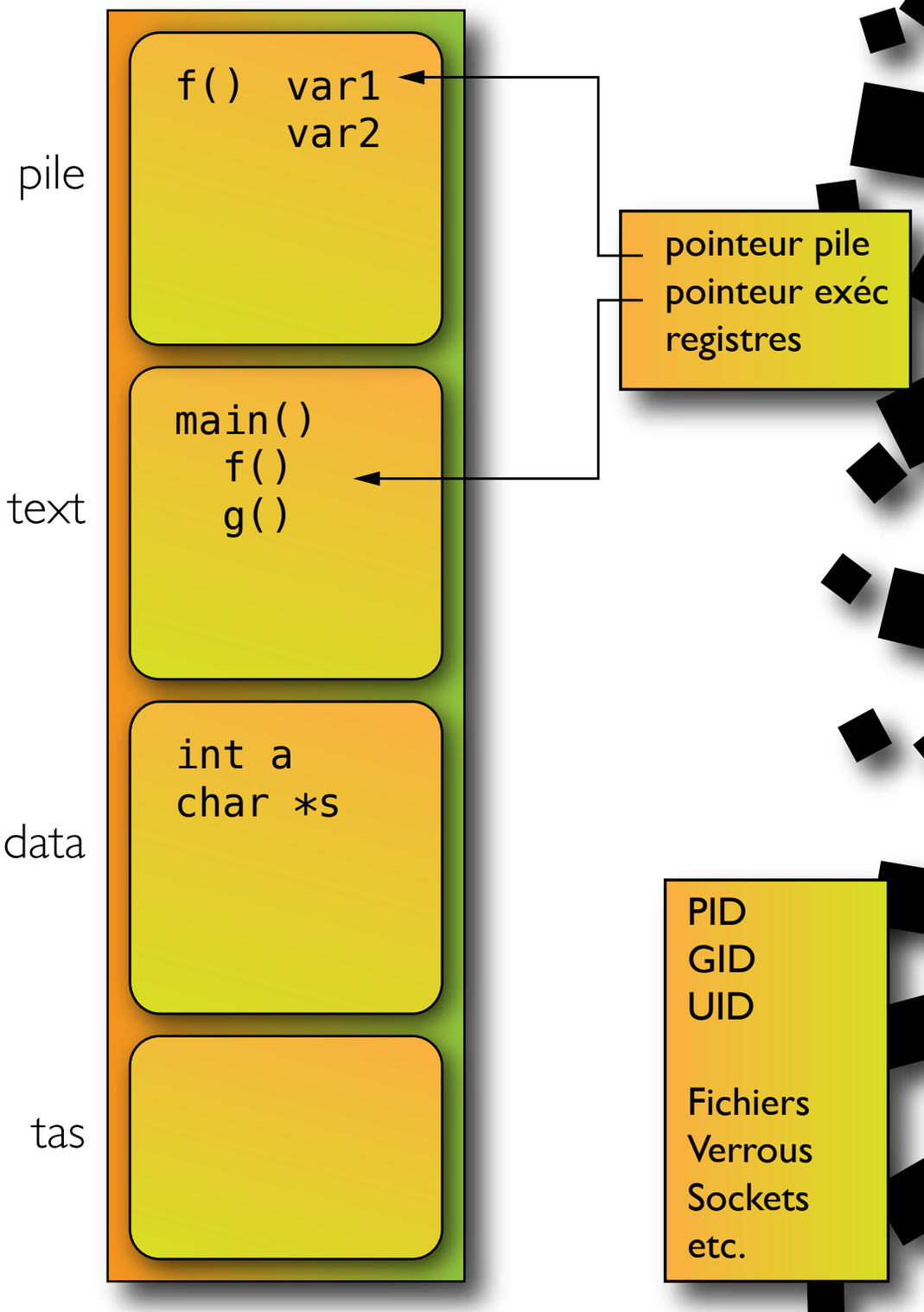
Si l'on veut exécuter plusieurs tâches en parallèle, on peut utiliser des processus différents

- la création d'un processus est une opération coûteuse
- chaque processus occupe un segment de mémoire séparée
- le code à exécuter est copié dans chaque nouveau processus
- la communication entre processus est difficile (tubes, fichiers, sockets, etc.)

Pour exécuter plusieurs instances de la même tâche en parallèle, on peut utiliser des *threads* (processus légers) à la place des processus

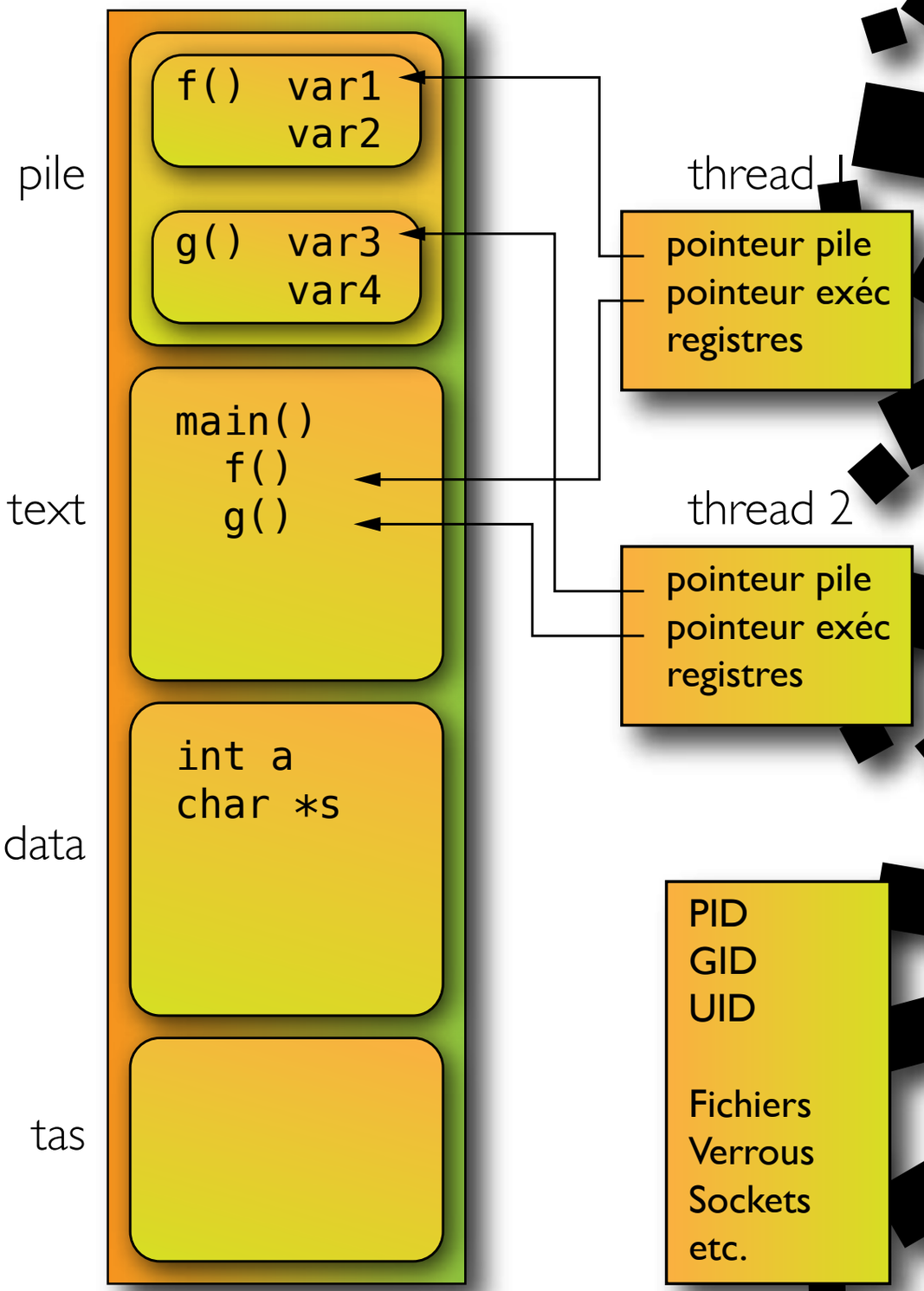
- Un unique processus peut exécuter plusieurs threads
- L'ordonnancement gère les threads d'un même processus comme des tâches séparées (potentiellement sur des processeurs différents)
- Les threads sont terminés lorsque le processus termine

Processus légers



- Chaque thread a ses propres variables locales, mais elles sont toutes dans la pile du processus (un thread a donc accès à la pile des autres)
- Les threads d'un processus partagent les segments **`text`** et **`data`** du processus, ainsi que le tas
- Chaque thread a ses propres pointeurs de pile et d'exécution, ainsi que l'état des registres du processeur

Processus légers



- Chaque thread a ses propres variables locales, mais elles sont toutes dans la pile du processus (un thread a donc accès à la pile des autres)
- Les threads d'un processus partagent les segments **`text`** et **`data`** du processus, ainsi que le tas
- Chaque thread a ses propres pointeurs de pile et d'exécution, ainsi que l'état des registres du processeur

En C

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
const pthread_attr_t *attr, void
*(*start_routine)(void *), void *arg)

int pthread_join(pthread_t thread, void
**value_ptr)

void pthread_exit(void *value_ptr)
```

La bibliothèque **pthread** permet de créer et gérer des threads dans un processus

- **pthread_create** pour démarrer un nouveau thread
  - **start_routine** est la fonction à exécuter dans le thread
  - cette fonction prend un unique argument **arg** de type **void***
- **pthread_join** pour attendre la fin d'un thread en cours
  - **value_ptr** est un pointeur où écrire le résultat de la fonction du thread qui a terminé
- **pthread_exit** permet de terminer un thread (appelée automatiquement si la fonction **start_routine** termine)

Exemple

```c
#include <stdio.h>

#define NB_CASES 1000

void tache(int *tab) {
    int i;
    for (i = 0; i < NB_CASES; i++) {
        tab[i]=i*i;
    }
}
```

```c
int main(void) {
    int i, tab[NB_CASES];

    tache(tab);

    for (i = 0; i < NB_CASES; i++) {
        printf("%d ", tab[i]);
    }
    return 0;
}
```

Exemple

```c
#include <stdio.h>

#define NB_CASES 1000
#define NB_THREADS 4

void tache(int debut, int fin, int *tab) {
    int i;
    for (i = debut; i < fin; i++) {
        tab[i]=i*i;
    }
}
```

```c
int main(void) {
    int i, tab[NB_CASES];

    for (i = 0; i < NB_THREADS; i++) {
        debut = i * NB_CASES / NB_THREADS;
        fin = (i+1) * NB_CASES / NB_THREADS;
        tache(debut, fin, tab);
    }

    for (i = 0; i < NB_CASES; i++) {
        printf("%d ", tab[i]);
    }
    return 0;
}
```

Exemple

```c
#include <pthread.h>
#include <stdio.h>

#define NB_CASES 1000
#define NB_THREADS 4

struct ThreadArgs {
    int debut;
    int fin;
    int *tab;
};

void* tache(void* args) {
    struct ThreadArgs *a = args;
    int i;
    for (i = a->debut; i < a->fin; i++) {
        a->tab[i]=i*i;
    }
    return NULL;
}
```

```c
int main(void) {
    int i, tab[NB_CASES];
    struct ThreadArgs args[NB_THREADS];
    pthread_t threads[NB_THREADS];

    for (i = 0; i < NB_THREADS; i++) {
        args[i].debut = i * NB_CASES / NB_
THREADS;
        args[i].fin = (i+1) * NB_CASES / NB_
THREADS;
        args[i].tab = tab;
        pthread_create(&threads[i], NULL, tache,
&args[i]);
    }

    for (i = 0; i < NB_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    for (i = 0; i < NB_CASES; i++) {
        printf("%d ", tab[i]);
    }
    return 0;
}
```

Concurrence

```c
#include <pthread.h>
#include <stdio.h>
#define NB_THREADS 4

void* incr(void *arg) {
    int i;
    int *c = arg;
    for (i = 0; i < 10000; i++) {
        (*c)++;
    }
    return NULL;
}

int main(void) {
    int i, c;
    pthread_t threads[NB_THREADS];

    for (i = 0; i < NB_THREADS; i++) {
        pthread_create(&threads[i], NULL,
incr, &c);
    }

    for (i = 0; i < NB_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("%d\n", c);
    return 0;
}
```

Les threads partagent le même espace mémoire

- La modification d'une variable par un thread affecte les lectures sur cette variable par les autres threads
- Problèmes d'accès concurrents
- Il faut utiliser des mécanismes complexes pour s'assurer du bon déroulement du programme
    - sémaphores
    - mutex
    - barrières

Thread safety

```c
#include <pthread.h>

int incr() {
    static int c = 0;
    static pthread_mutex_t mutex;
    mutex = PTHREAD_MUTEX_INITIALIZER;

    // bloquer le verrou
    pthread_mutex_lock(&mutex);

    c++;
    int r = c; // sauvegarder résultat

    // libérer le verrou
    pthread_mutex_unlock(&mutex);

    return r;
}
```

- assurer que les sections critiques (manipulations des variables partagées) ne soient accessibles que par un thread
- attention aux blocages potentiels
- difficile à tester (certains effets se produisent rarement)
- les verrouillages ralentissent le parallélisme (goulot d'étranglement)

Réentrance

```
// cette fonction est réentrante
void echange(int *x, int *y) {
    int t;
    t = *x;
    *x = *y;
    *y = t;
}
```

Une fonction est dite *réentrante* si elle se comporte correctement lorsqu'elle est appelée pendant une exécution d'elle-même

- récursivité
- exécution lors d'une interruption
- exécution concurrente (threads)

Réentrance

```c
int t;

// cette fonction n'est pas réentrante
void echange(int *x, int *y) {
    t = *x;
    *x = *y;
    *y = t;
}
```

Une fonction est dite *réentrante* si elle se comporte correctement lorsqu'elle est appelée pendant une exécution d'elle-même

- récursivité
- exécution lors d'une interruption
- exécution concurrente (threads)

Réentrance

```
int t;

// cette fonction est réentrante
void echange(int *x, int *y) {
    int s = t;
    t = *x;
    *x = *y;
    *y = t;
    t = s;
}
```

Une fonction est dite *réentrante* si elle se comporte correctement lorsqu'elle est appelée pendant une exécution d'elle-même

- récursivité
- exécution lors d'une interruption
- exécution concurrente (threads)

Compilation

- Pour utiliser les fonctions de la bibliothèque **`pthread`**, il faut demander au compilateur de lier l'exécutable à la bibliothèque : **`-lpthread`**
- Par ailleurs, il faut indiquer que les fonctions doivent être *réentrantes* : **`-D_REENTRANT`**
  - certaines fonctions ont des variantes réentrantes (ex : **`strtok_r`** au lieu de **`strtok`**)
  - certaines macros sont remplacées par des fonctions (ex : **`getc`** et **`putc`**)
  - chaque thread dispose d'une instance différente de la variable **`errno`**

```
$ gcc prog.c -lpthread -D_REENTRANT
```

Compilation

```
$ gcc prog.c -pthread
```

- Lorsqu'elle est disponible, l'option **`-pthread`** se charge d'activer les options nécessaires, spécifiques au système courant (c'est la solution à préférer)
- la plupart du temps, cela correspond à **`-lpthread -D_REENTRANT`**

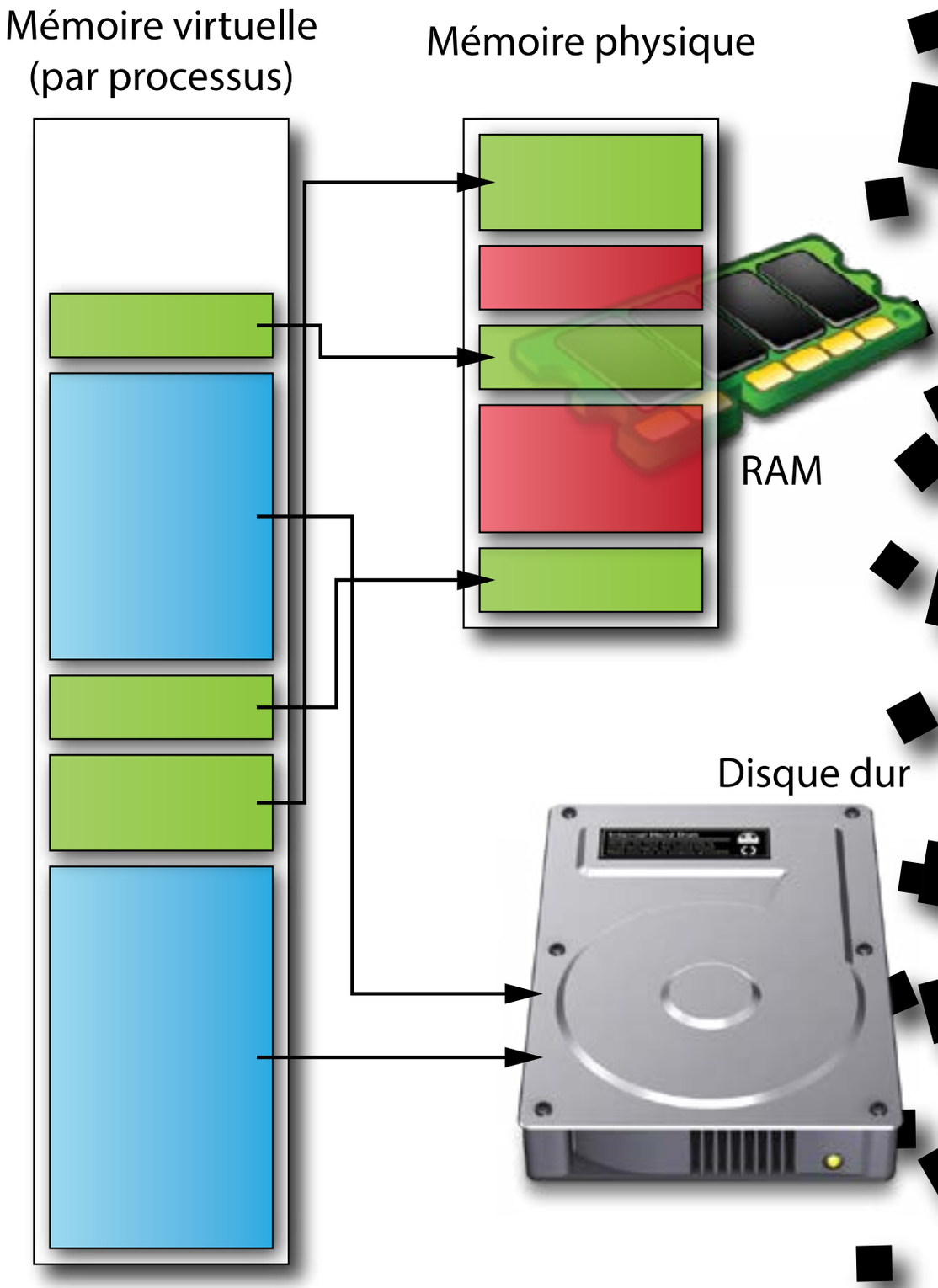
Processus

- Espace de mémoire virtuelle (apparaît comme connexe) séparée
- Identifiant unique au niveau OS
- Les processus sont disjoints
- Peuvent fonctionner sur des machines distinctes

Thread

- Sous-division d'un processus
- Mémoire partagée (même espace d'adressage virtuel)
- Sur la même machine (potentiellement sur des processeurs différents)
- Plus simple à créer
- Tous les threads d'un même processus ont le même code

Virtualisation



Les processus n'ont pas directement accès aux différentes mémoires physiques

- Ils disposent d'un espace de mémoire virtuelle
- L'unité de gestion de mémoire (MMU) s'occupe de la correspondance
- Permet d'isoler les processus
- Masque les problèmes de fragmentation (le noyau s'en occupe)