Processus légers

Aloÿs Dufour

ATER, LIPN équipe LoCal
Université Paris-Nord XIII

3 février 2026

Threads

Partage des tâches en utilisant plusieurs processus : lent

- autant d'espaces mémoire
- commutation de contexte lent
- communication nécessitant des protocoles et appels systèmes

Threads

Partage des tâches en utilisant plusieurs processus : lent

- autant d'espaces mémoire
- commutation de contexte lent
- communication nécessitant des protocoles et appels systèmes

Autre méthode pour tirer avantage des machines multicœurs : *threads* ou processus légers.

Threads

Definition

Un *thread* est une entité ordonnancée par le noyau.

Threads

Definition

Un *thread* est une entité ordonnancée par le noyau.

Différences :

- un thread est une “partie” d’un processus
- un processus est l’exécution de plusieurs threads
- les threads partagent presque tout :
  espace mémoire, PID PPID, descripteurs de fichiers ouverts, UID, handlers des signaux, répertoire courant, masque de fichiers...
  mais pas les identifiants de threads, la pile, le masque des signaux, `errno`
- synchronisation à la main

Threads

Definition

Un *thread* est une entité ordonnancée par le noyau.

Différences :

- un thread est une “partie” d’un processus
- un processus est l’exécution de plusieurs threads
- les threads partagent presque tout :
  espace mémoire, PID PPID, descripteurs de fichiers ouverts, UID, handlers des signaux, répertoire courant, masque de fichiers...
  mais pas les identifiants de threads, la pile, le masque des signaux, `errno`
- synchronisation à la main

Problèmes :

- Utilisation des mêmes copies des librairies, doivent être *“MT-Safe”*
- Gestion de la synchronisation (mutex, sémaphores)
- En cas de mauvaise synchronisation : segfaults, interblocage

Threads Posix

Morceaux choisis (5/102)

```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 **retval);
pthread_t pthread_self(void);
int pthread_detach(pthread_t);
int pthread_equal(pthread_t, pthread_t);
```

Compilation avec l'option `-lpthread`

Création d'un thread

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

THREAD l'adresse mémoire où sera copié l'identifiant du nouveau thread,

ATTR des attributs (toujours `NULL` dans ce cours),

START_ROUTINE la fonction d'entrée du thread qui doit impérativement n'avoir qu'un seul paramètre de type `void *` et retourner une valeur de type `void *`,

ARG argument de ladite fonction.

Création d'un thread

- Le type opaque `pthread_t` permet de stocker un identifiant de thread,
- `pthread_create` met à l'adresse indiquée par son premier argument l'identifiant du thread créé,
- on y a accès depuis l'intérieur du thread via `pthread_self`,
- on peut comparer deux identifiant via `pthread_equal`.

Création d'un thread

- Le type opaque `pthread_t` permet de stocker un identifiant de thread,
- `pthread_create` met à l'adresse indiquée par son premier argument l'identifiant du thread créé,
- on y a accès depuis l'intérieur du thread via `pthread_self`,
- on peut comparer deux identifiant via `pthread_equal`.
- Évidemment, en cas d'erreur, retourne une valeur $\neq 0$ et remplit `errno` en conséquence, d'où

```
if (pthread_create(&t, NULL, fct, arg) != 0) {
    perror("pthread_create");
    return 1;
}
```

Attendre la terminaison d'un thread

```
int pthread_join(pthread_t thread, void **retval);
```

THREAD identifiant du thread dont la terminaison est attendue

RETVAL pour stocker la valeur de retour de la fonction exécutée par le thread, `NULL` dans une certaine quantité de cas.

Attendre la terminaison d'un thread

```
int pthread_join(pthread_t thread, void **retval);
```

THREAD identifiant du thread dont la terminaison est attendue

RETVAL pour stocker la valeur de retour de la fonction exécutée par le thread, `NULL` dans une certaine quantité de cas.

- `fork` ↔ `create`
- `wait` ↔ `join`
- `getpid` ↔ `pthread_self`

Attendre la terminaison d'un thread

```
int pthread_join(pthread_t thread, void **retval);
```

THREAD identifiant du thread dont la terminaison est attendue

RETVAL pour stocker la valeur de retour de la fonction exécutée par le thread, `NULL` dans une certaine quantité de cas.

- `fork` ↔ `create`
- `wait` ↔ `join`
- `getpid` ↔ `pthread_self`

Si on ne veut pas à avoir à gérer la fin d'un thread, on peut le détacher avec

```
int pthread_detach(pthread_t thread);
```

Exemple avec valeur de retour

```c
int main(void)
{
    int tab[2] = { 33, 78 };
    pthread_t t;
    void *ret_addr;

    if (pthread_create(&t, NULL, f, tab) != 0) {
        perror("pthread_create");
        return 1;
    }
    pthread_join(t, &ret_addr);
    printf("%d + %d = %d\n", tab[0], tab[1], *((int *)ret_addr));
    free(ret_addr);
    return 0;
}

void *f(void *arg)
{
    int *tab = arg;
    int *resu = malloc(sizeof(int));
    *resu = tab[0] + tab[1];
    return resu;
}
```

Exemple avec valeur de retour

- le thread initial demande la création d'un second thread ;
- qui exécutera la fonction `f` ;
- avec l'adresse du tableau `tab` en argument.
- Le second thread commence par interpréter l'adresse transmise comme l'adresse d'un `int` ;

```
int *tab = arg;
```

- puis alloue de la mémoire sur le tas pour stocker le résultat de son calcul [1] ;

```
int *resu = malloc(sizeof(int));
```

- et retourne l'adresse de ce résultat.

[1] en toute rigueur, on devrait tester si `malloc` a retourné `NULL`

Exemple avec valeur de retour

- Le thread initial attend la terminaison du second thread, puis met dans la variable `ret_addr` l'adresse transmise par celui-ci.

```
void *ret_addr;
pthread_join(t, &ret_addr);
```

- Il peut ensuite l'interpréter comme l'adresse d'un `int` et libérer la mémoire allouée.

```
printf("%d + %d = %d\n", tab[0], tab[1], *((int *) ret_addr));
free(ret_addr);
```

Interlude sur les pointeurs : le typage

Le type d'un pointeur apporte :

- le nombre d'octets à sauter dans l'arithmétique des pointeurs,
- le nombre d'octets à récupérer ou à écrire lors d'un déréfencement,
- une aide au compilateur pour la détection d'erreurs de typage.

Interlude sur les pointeurs : le typage

Le type d'un pointeur apporte :

- le nombre d'octets à sauter dans l'arithmétique des pointeurs,
- le nombre d'octets à récupérer ou à écrire lors d'un déréfencement,
- une aide au compilateur pour la détection d'erreurs de typage.

Toute ces adresses font généralement la même taille (80), on peut abandonner les types avec les pointeurs génériques `void *`.

- Utilisé lorsqu'on veut monter en généralité (`memcpy` copie des octets).
- Utilisé parce qu'absance d'autres abstractions de type en C (absence de généricité, types paramétrés, types d'ordres supérieurs...)o

Interlude sur les pointeurs : des exemples bien connus

```
void *malloc(size_t size);
void free(void *ptr);
void *memcpy(void *dest, const void *src, size_t n);
ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Interlude sur les `void*` : faire ne pas faire

Ce que l'on peut faire avec `void *`

- y mettre n'importe quel type d'adresse dedans sans *cast* nécessaire,
- l'y mettre dans n'importe quel type de pointeur sans *cast* nécessaire.

Interlude sur les `void*` : faire ne pas faire

Ce que l'on peut faire avec `void *`

- y mettre n'importe quel type d'adresse dedans sans *cast* nécessaire,
- l'y mettre dans n'importe quel type de pointeur sans *cast* nécessaire.

Ce que l'on ne peut pas faire avec :

- de l'arithmétique de pointeur, puisqu'on ignore son type pointé et donc sa taille (à extensions non-standard de `gcc` près...)
- déréférencer un pointeur de type `void *`, même raison.

Interlude sur les void* : marche ne pas marche ?

```c
int n;
double x;
void *p;
p = &n;
p = &x;

int tab[] = { 33, 78 };
void *p = tab;
++p;

int n;
void *p = &n;
int *m = p;
m = malloc(sizeof(int) * 4);

void *p = malloc(12);
*p = 45;
int n = *p + 3;
```

Interlude sur les pointeurs : de fonctions

```
int img_0(int (*f) (int));
int compose(int (*f) (int), int (*g) (int), int n);
```

Interlude sur les pointeurs : de fonctions

```
int img_0(int (*f) (int));
int compose(int (*f) (int), int (*g) (int), int n);
```

Facilité d'écriture : lorsqu'un pointeur de fonction est appliqué à des arguments, il est automatiquement déréférencé.

```
int img_0(int (*f) (int))
{
    return f(0); /* déréférencement implicite */
}
```

Interlude sur les pointeurs : de fonctions

```
int img_0(int (*f) (int));
int compose(int (*f) (int), int (*g) (int), int n);
```

Facilité d'écriture : lorsqu'un pointeur de fonction est appliqué à des arguments, il est automatiquement déréférencé.

```
int img_0(int (*f) (int))
{
    return f(0); /* déréférencement implicite */
}
```

Appels :

```
n = img_0(&triple);   /* adresse de triple donnée en argument */
m = img_0(plus_deux); /* ou cast automatique en pointeurs de fonctions */
```

Interlude sur les pointeurs : de fonctions

- `r (*)(p1, …, pn)` : type pointeur de fonction `p1` $\times \cdots \times$ `pn` $\rightarrow$ `r`,
- Partout où l'adresse d'une fonction du bon type est attendue, on peut donner le nom avec ou sans `&`,
- de même, pour l'appeler, avec ou sans `*`.

Interlude sur les pointeurs : de fonctions

- `r (*)(p1, …, pn)` : type pointeur de fonction `p1` $\times \cdots \times$ `pn` $\rightarrow$ `r`,
- Partout où l'adresse d'une fonction du bon type est attendue, on peut donner le nom avec ou sans `&`,
- de même, pour l'appeler, avec ou sans `*`.

Le standard C n'autorise pas :

- les casts entre différents types de pointeurs de fonction,
- les casts entre pointeurs de fonction et pointeurs simples (même avec `void *`),
- mais certains compilateurs peuvent décider de l'autoriser dans certains cas.

Interlude sur les pointeurs : devinettes sur les types

```
char l[10]
int *p
double f()
```

Interlude sur les pointeurs : devinettes sur les types

```
char l[10]          /* tableau de 10 caractères */
int *p              /* pointeur d'entier */
double f()          /* fonction retournant un double */
```

Interlude sur les pointeurs : devinettes sur les types

```
char l[10]          /* tableau de 10 caractères */
int *p              /* pointeur d'entier */
double f()          /* fonction retournant un double */

char *ch[10]
int m[100][40]
char **argv
```

Interlude sur les pointeurs : devinettes sur les types

```
char l[10]          /* tableau de 10 caractères */
int *p              /* pointeur d'entier */
double f()          /* fonction retournant un double */

char *ch[10]        /* tableau de 10 pointeurs de caractères */
int m[100][40]      /* tableau de 100 éléments, étant chacun un tableau de 40 entier */
char **argv         /* pointeur de pointeur de caractère */
```

Interlude sur les pointeurs : devinettes sur les types

```
char l[10]          /* tableau de 10 caractères */
int *p              /* pointeur d'entier */
double f()          /* fonction retournant un double */

char *ch[10]        /* tableau de 10 pointeurs de caractères */
int m[100][40]      /* tableau de 100 éléments, étant chacun un tableau de 40 entier */
char **argv         /* pointeur de pointeur de caractère */

int (*tab)[10]
char (*f)()
char *(*g)()
float *(*t[20])()
```

Interlude sur les pointeurs : devinettes sur les types

```
char l[10]          /* tableau de 10 caractères */
int *p              /* pointeur d'entier */
double f()          /* fonction retournant un double */

char *ch[10]        /* tableau de 10 pointeurs de caractères */
int m[100][40]      /* tableau de 100 éléments, étant chacun un tableau de 40 entier */
char **argv         /* pointeur de pointeur de caractère */

int (*tab)[10]      /* pointeur de tableau de 10 entiers */
char (*f)()         /* pointeur de fonction retournant un caractère */
char *(*g)()        /* pointeur de fonction retournant un pointeur de caractère */
float *(*t[20])()   /* tableau de 20 pointeurs de fonction retournant un pointeur de float */
```

Interlude sur les pointeurs : devinettes sur les types

```
char l[10]          /* tableau de 10 caractères */
int *p              /* pointeur d'entier */
double f()          /* fonction retournant un double */

char *ch[10]        /* tableau de 10 pointeurs de caractères */
int m[100][40]      /* tableau de 100 éléments, étant chacun un tableau de 40 entier */
char **argv         /* pointeur de pointeur de caractère */

int (*tab)[10]      /* pointeur de tableau de 10 entiers */
char (*f)()         /* pointeur de fonction retournant un caractère */
char *(*g)()        /* pointeur de fonction retournant un pointeur de caractère */
float *(*t[20])()   /* tableau de 20 pointeurs de fonction retournant un pointeur de float */

void (*(*f[])())()
```

Interlude sur les pointeurs : devinettes sur les types

```
char l[10]          /* tableau de 10 caractères */
int *p              /* pointeur d'entier */
double f()          /* fonction retournant un double */

char *ch[10]        /* tableau de 10 pointeurs de caractères */
int m[100][40]      /* tableau de 100 éléments, étant chacun un tableau de 40 entier */
char **argv         /* pointeur de pointeur de caractère */

int (*tab)[10]      /* pointeur de tableau de 10 entiers */
char (*f)()         /* pointeur de fonction retournant un caractère */
char *(*g)()        /* pointeur de fonction retournant un pointeur de caractère */
float *(*t[20])()   /* tableau de 20 pointeurs de fonction retournant un pointeur de float */

void (*(*f[])())()  /* tableau de pointeurs de fonctions retournant
                     * des pointeurs de fonction ne retournant rien */
```

Retour au threads : comportement avec exec et fork

exec si un thread fait lui fait appel (et n'échoue pas), tous les autres threads sont tués.

fork seul le thread l'appelant est dupliqué. Problème : comme les autres threads n'existent plus dans le fils, il se peut qu'un mutex ne soit jamais libéré.
Une solution est d'utiliser `pthread_atfork` qui rajoute des handlers dans ce cas.

Le début des problèmes

Les threads partageant leur mémoire, les librairies utilisées doivent être prévues pour un usage multi-thread.
Lors de la communication par mémoire partagée, il faut faire attention à ce que deux threads ne travaillent pas sur la même zone mémoire en même temps, sinon plantages, exploits, heisenbugs…

Le début des problèmes

Les threads partageant leur mémoire, les librairies utilisées doivent être prévues pour un usage multi-thread.
Lors de la communication par mémoire partagée, il faut faire attention à ce que deux threads ne travaillent pas sur la même zone mémoire en même temps, sinon plantages, exploits, heisenbugs...

Definition

Une fonction est appelée *ré-entrante* si elle peut être interrompue, et rappelée (de manière sûre).

Ré-entrance problématique : exemple sur strtok

```c
#include <string.h>
char *strtok(char *str, const char *delim);
/* strtok coupe str aux caractères présents dans delim
 * Si str=NULL, elle continue de découper la chaîne précédente */
char w[] = "2:5:6:1:2:6:3";
char *p = strtok(w, ":");
while (p) {
    printf("%s", p);
    p = strtok(NULL, ":");
}
```

Ré-entrance problématique : exemple sur strtok

```c
#include <string.h>
char *strtok(char *str, const char *delim);
/* strtok coupe str aux caractères présents dans delim
 * Si str=NULL, elle continue de découper la chaîne précédente */
char w[] = "2:5:6:1:2:6:3";
char *p = strtok(w, ":");
while (p) {
    printf("%s", p);
    p = strtok(NULL, ":");
}
```

Sortie : 2 5 6 1 2 6 3

Ré-entrance problématique : exemple sur strtok

```c
#include <string.h>
char *strtok(char *str, const char *delim);
/* strtok coupe str aux caractères présents dans delim
 * Si str=NULL, elle continue de découper la chaîne précédente */
char w[] = "2:5:6:1:2:6:3";
char *p = strtok(w, ":");
while (p) {
    printf("%s", p);
    p = strtok(NULL, ":");
}
```

Sortie : 2 5 6 1 2 6 3

Problème avec les programmes multithread : `strtok` garde en interne un pointeur sur la position courante dans la chaîne.

Ré-entrance problématique : exemple sur strtok

```c
#include <string.h>
char *strtok(char *str, const char *delim);
/* strtok coupe str aux caractères présents dans delim
 * Si str=NULL, elle continue de découper la chaîne précédente */
char w[] = "2:5:6:1:2:6:3";
char *p = strtok(w, ":");
while (p) {
    printf("%s", p);
    p = strtok(NULL, ":");
}
```

Sortie : 2 5 6 1 2 6 3

Problème avec les programmes multithread : `strtok` garde en interne un pointeur sur la position courante dans la chaîne.

Version ré-entrante :

```c
char *strtok_r(char *str, const char *delim, char **restrict saveptr);
```

ne garde pas un pointeur interne, il faut lui en fournir un.

Problèmes de synchronisation

Un thread peut être arrêté n'importe quand pour laisser sa place à un autre thread

- y compris au milieu d'une ligne
- y compris au milieu d'une instruction basique en C (ex : `i++`)

Donc si thread *A* travaille sur une zone mémoire *M*, et est arrêté alors que *M* est inconsistante, le thread *B* trouvera *M* en état inconsistant...
comportement imprévisible

Atomicité

Codde assembleur de `i++`

```
movq i(%rip) , %rax
addq $1, %rax
movq %rax, i(%rip)
```

On ne veut pas que ces instructions puissent être interrompues.

Definition

Une instruction est dite *atomique* si elle ne peut être interrompue.

Pour rendre atomique des portions de *code critique* : mécasisme d'exclusion mutuelle, mutex !

Exo : bien ou bien ?

Dans chacun des cas suivants, dire ce qu'affiche le programme, ou bien, s'il y a un comportement imprévisible ou des erreurs possibles à l'exécution, dire pourquoi. Pour des raisons de concision, les appels systèmes ne sont pas testés.

```c
void *f(void *arg) {
    printf("hop\n");
    return NULL;
}
int main(void) {
    pthread_t p;
    pthread_create(&p, NULL, f, NULL);
    pthread_join(p, NULL);
    printf("plop\n");
    return 0;
}
```

Exo : bien ou bien ?

Dans chacun des cas suivants, dire ce qu'affiche le programme, ou bien, s'il y a un comportement imprévisible ou des erreurs possibles à l'exécution, dire pourquoi. Pour des raisons de concision, les appels systèmes ne sont pas testés.

- ```
  void *f(void *arg) {
      printf("hop\n");
      return NULL;
  }
  int main(void) {
      pthread_t p;
      pthread_create(&p, NULL, f, NULL);
      pthread_join(p, NULL);
      printf("plop\n");
      return 0;
  }
  ```
- L'affichage est `hop` suivi de `plop`. L'appel à `join` assure que le *thread* initial attend la fin du *thread* créé avant de lui-même afficher.
  ```

Exo : bien ou bien ?

Dans chacun des cas suivants, dire ce qu'affiche le programme, ou bien, s'il y a un comportement imprévisible ou des erreurs possibles à l'exécution, dire pourquoi. Pour des raisons de concision, les appels systèmes ne sont pas testés.

- ```
  void *f(void *arg) {
      printf("hop\n");
      return NULL;
  }
  int main(void) {
      pthread_t p;
      pthread_create(&p, NULL, f, NULL);
      pthread_join(p, NULL);
      printf("plop\n");
      return 0;
  }
  ```
- L'affichage est `hop` suivi de `plop`. L'appel à `join` assure que le *thread* initial attend la fin du *thread* créé avant de lui-même afficher.

- ```
  void *f(void *arg) {
      printf("hop\n");
      return NULL;
  }
  int main(void) {
      pthread_t p;
      pthread_create(&p, NULL, f, NULL);
      printf("plop\n");
      return 0;
  }
  ```

Exo : bien ou bien ?

Dans chacun des cas suivants, dire ce qu'affiche le programme, ou bien, s'il y a un comportement imprévisible ou des erreurs possibles à l'exécution, dire pourquoi. Pour des raisons de concision, les appels systèmes ne sont pas testés.

- ```
  void *f(void *arg) {
      printf("hop\n");
      return NULL;
  }
  int main(void) {
      pthread_t p;
      pthread_create(&p, NULL, f, NULL);
      pthread_join(p, NULL);
      printf("plop\n");
      return 0;
  }
  ```
- L'affichage est `hop` suivi de `plop`. L'appel à `join` assure que le *thread* initial attend la fin du *thread* créé avant de lui-même afficher.

- ```
  void *f(void *arg) {
      printf("hop\n");
      return NULL;
  }
  int main(void) {
      pthread_t p;
      pthread_create(&p, NULL, f, NULL);
      printf("plop\n");
      return 0;
  }
  ```
- Comportement imprévisible. C'est l'échec, aucune synchronisation, tout peut arriver, y compris (et c'est ce qui arrivera le plus souvent) que le *thread* initial atteigne la fin de `main` et mette fin au processus, avant que le *thread* créé n'ai eu le temps d'afficher.

Exo : bien ou bien ?

```c
void *f(void *arg) {
    printf("hop\n");
    kill(getpid(), SIGTERM);
    return NULL;
}
int main(void) {
    pthread_t p;
    pthread_create(&p, NULL, f, NULL);
    printf("plop\n");
    return 0;
}
```

Exo : bien ou bien ?

- ```
  void *f(void *arg) {
      printf("hop\n");
      kill(getpid(), SIGTERM);
      return NULL;
  }
  int main(void) {
      pthread_t p;
      pthread_create(&p, NULL, f, NULL);
      printf("plop\n");
      return 0;
  }
  ```
- Comportement imprévisible. Le signal est envoyé à tout le processus, donc il est tout à fait possible que le *thread* créé mette fin au processus avant que le *thread* initial n'ait le temps d'afficher quoi que ce soit.
  ```

Exo : bien ou bien ?

- ```
  void *f(void *arg) {
      printf("hop\n");
      kill(getpid(), SIGTERM);
      return NULL;
  }
  int main(void) {
      pthread_t p;
      pthread_create(&p, NULL, f, NULL);
      printf("plop\n");
      return 0;
  }
  ```
- Comportement imprévisible. Le signal est envoyé à tout le processus, donc il est tout à fait possible que le *thread* créé mette fin au processus avant que le *thread* initial n'ait le temps d'afficher quoi que ce soit.

- ```
  void *f(void *arg) {
      printf("hop\n");
      strcpy(arg, "plop\n");
      return NULL;
  }
  int main(void) {
      pthread_t p;
      char ch[256];
      pthread_create(&p, NULL, f, ch);
      pthread_join(p, NULL);
      printf(ch);
      return 0;
  }
  ```

Exo : bien ou bien ?

- ```
  void *f(void *arg) {
      printf("hop\n");
      kill(getpid(), SIGTERM);
      return NULL;
  }
  int main(void) {
      pthread_t p;
      pthread_create(&p, NULL, f, NULL);
      printf("plop\n");
      return 0;
  }
  ```
- Comportement imprévisible. Le signal est envoyé à tout le processus, donc il est tout à fait possible que le *thread* créé mette fin au processus avant que le *thread* initial n'ait le temps d'afficher quoi que ce soit.

- ```
  void *f(void *arg) {
      printf("hop\n");
      strcpy(arg, "plop\n");
      return NULL;
  }
  int main(void) {
      pthread_t p;
      char ch[256];
      pthread_create(&p, NULL, f, ch);
      pthread_join(p, NULL);
      printf(ch);
      return 0;
  }
  ```
- L'affichage est `hop` suivi de `plop`. Le *thread* créé a écrit dans la pile du *thread* initial la chaîne à afficher.

Exo : bien ou bien ?

```c
void *f(void *arg) {
    printf("hop\n");
    char *ch = malloc(256);
    strcpy(ch, "plop\n");
    return ch;
}
int main(void) {
    pthread_t p;
    void *ret;
    pthread_create(&p, NULL, f, NULL);
    pthread_join(p, &ret);
    printf(ret);
    free(ret);
    return 0;
}
```

Exo : bien ou bien ?

- ```
  void *f(void *arg) {
      printf("hop\n");
      char *ch = malloc(256);
      strcpy(ch, "plop\n");
      return ch;
  }
  int main(void) {
      pthread_t p;
      void *ret;
      pthread_create(&p, NULL, f, NULL);
      pthread_join(p, &ret);
      printf(ret);
      free(ret);
      return 0;
  }
  ```
- L'affichage est `hop` suivi de `plop`. Le *thread* créé a écrit dans le tas la chaîne `plop`. Le *thread* initial y a accédé puis a libéré la mémoire.
  ```

Exo : bien ou bien ?

- ```
  void *f(void *arg) {
      printf("hop\n");
      char *ch = malloc(256);
      strcpy(ch, "plop\n");
      return ch;
  }
  int main(void) {
      pthread_t p;
      void *ret;
      pthread_create(&p, NULL, f, NULL);
      pthread_join(p, &ret);
      printf(ret);
      free(ret);
      return 0;
  }
  ```
- L'affichage est `hop` suivi de `plop`. Le *thread* créé a écrit dans le tas la chaîne `plop`. Le *thread* initial y a accédé puis a libéré la mémoire.

- ```
  void *f(void *arg) {
      printf("hop\n");
      char ch[256];
      strcpy(ch, "plop\n");
      return ch;
  }
  int main(void) {
      pthread_t p;
      void *ret;
      pthread_create(&p, NULL, f, NULL);
      pthread_join(p, &ret);
      printf(ret);
      return 0;
  }
  ```

Exo : bien ou bien ?

- ```
  void *f(void *arg) {
      printf("hop\n");
      char *ch = malloc(256);
      strcpy(ch, "plop\n");
      return ch;
  }
  int main(void) {
      pthread_t p;
      void *ret;
      pthread_create(&p, NULL, f, NULL);
      pthread_join(p, &ret);
      printf(ret);
      free(ret);
      return 0;
  }
  ```
- L'affichage est `hop` suivi de `plop`. Le *thread* créé a écrit dans le tas la chaîne `plop`. Le *thread* initial y a accédé puis a libéré la mémoire.

- ```
  void *f(void *arg) {
      printf("hop\n");
      char ch[256];
      strcpy(ch, "plop\n");
      return ch;
  }
  int main(void) {
      pthread_t p;
      void *ret;
      pthread_create(&p, NULL, f, NULL);
      pthread_join(p, &ret);
      printf(ret);
      return 0;
  }
  ```
- Comportement imprévisible. Erreur courante ! Le *thread* créé a écrit dans sa pile, mais lorsqu'il se termine, sa pile est libérée. Le *thread* initial cherche à lire une chaîne dans une partie de la mémoire qui est libérée, ce qui provoquera sans doute une erreur de segmentation.

Exo classique : parallélisation du produit matriciel

Nombre N de lignes fixé à la compilation
Nombre K de threads fixé à la compilation.