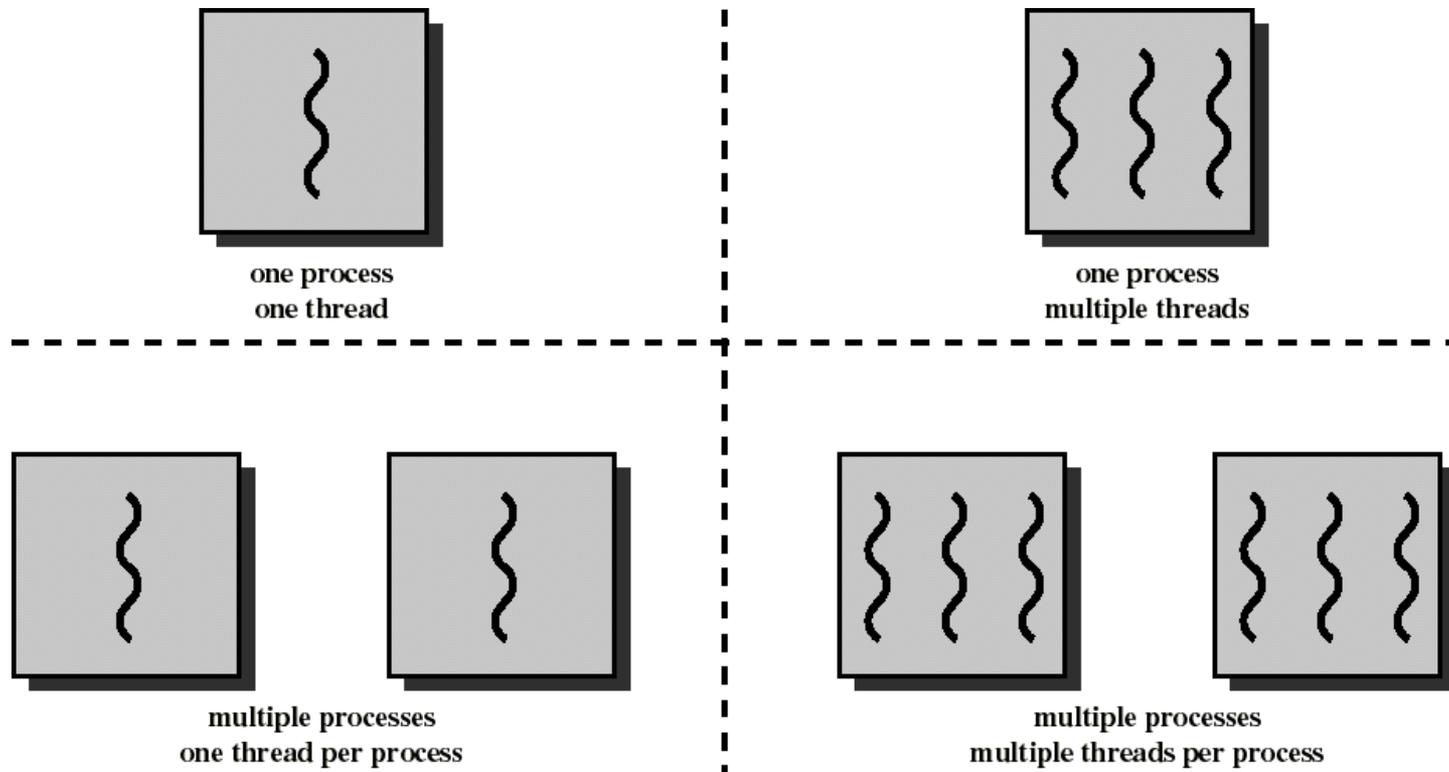


# Fils d'exécution

- Qu'est ce qu'un fil d'exécution?
- Usage des fils d'exécution
- Threads POSIX
  - Création
  - Attente de la fin d'un fil d'exécution
  - Terminaison
  - Nettoyage à la terminaison

# Qu'est ce qu'un fil d'exécution?

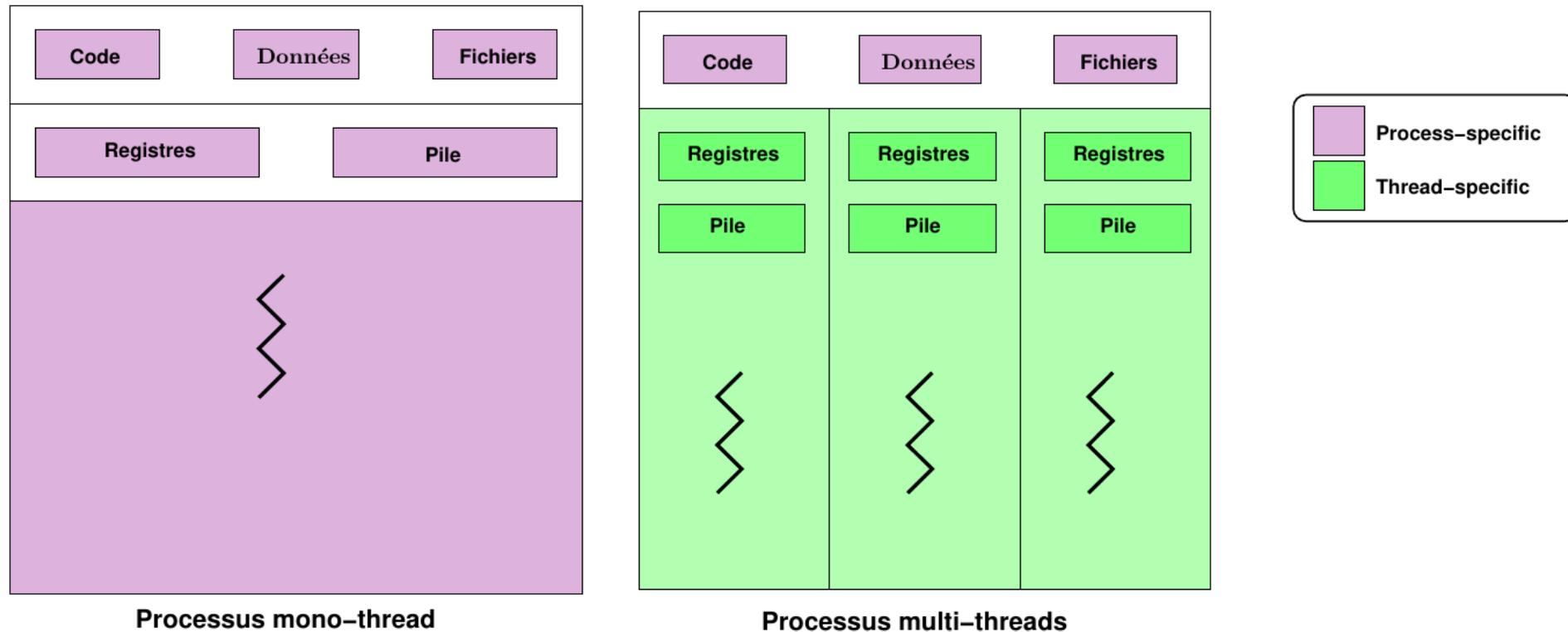
- Le modèle processus décrit précédemment est un programme qui s'exécute selon un chemin unique (compteur ordinal). On dit qu'il a un fil d'exécution ou flot de contrôle unique (single thread).
- De nombreux systèmes d'exploitation modernes offrent la possibilité d'associer à un même processus plusieurs chemins d'exécution (multithreading).



# Qu'est ce qu'un fil d'exécution?

- Un processus est vu comme étant un ensemble de ressources (espace d'adressage, fichiers, périphériques...) que ses fils d'exécution (threads) partagent.
- Lorsqu'un processus est créé, un seul fil d'exécution (thread) est associé au processus. Ce fil peut en créer d'autres.
- Chaque fil a:
  - un identificateur unique
  - une pile d'exécution
  - des registres (un compteur ordinal)
  - un état...

# Qu'est ce qu'un thread?



- Le multithreading permet l'exécution simultanée ou en pseudo-parallèle de plusieurs parties d'un même processus.

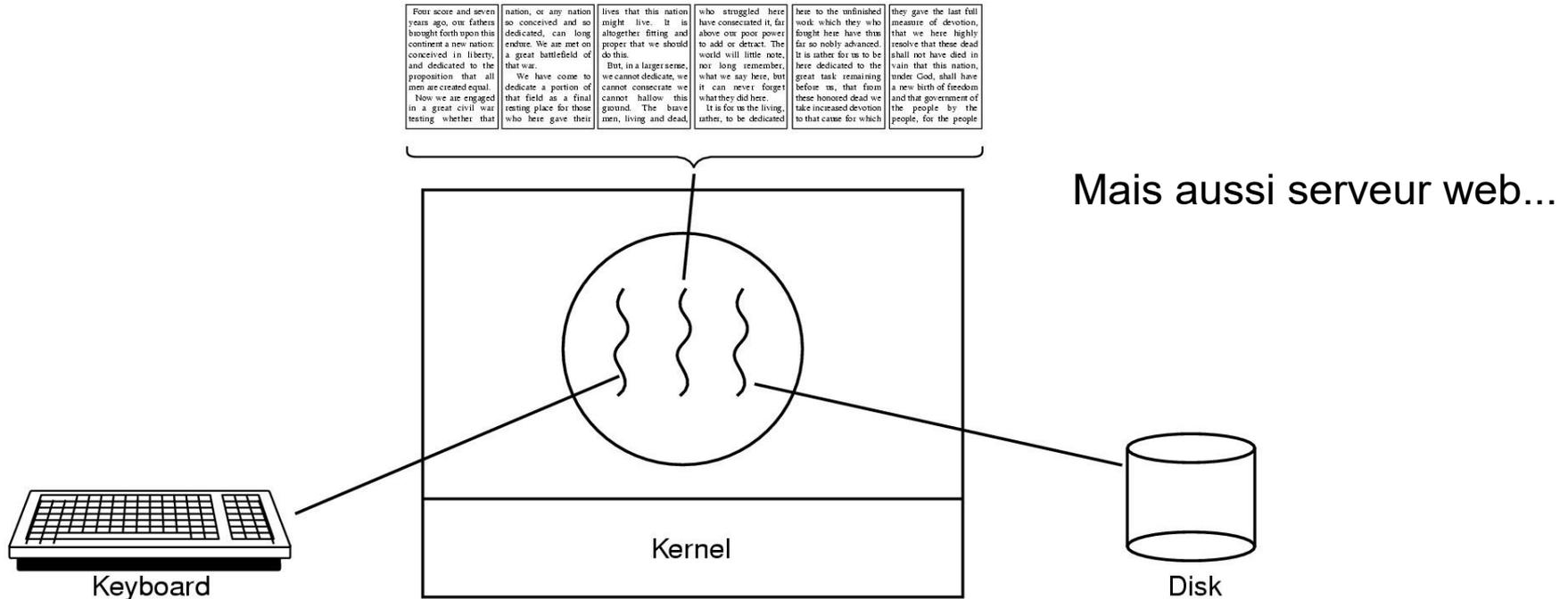
# Avantages des fils / processus

- Réactivité (le processus peut continuer à s'exécuter même si certaines de ses parties sont bloquées, exemple E/S),
- Partage de ressources (facilite la coopération, améliore les performances),
- Économie d'espace mémoire et de temps. Il faut moins de temps pour :
  - créer, terminer un fil (sous Solaris, la création d'un processus est 30 fois plus lente que celle d'un thread),
  - Changer de contexte entre deux fils d'un même processus.

- Recouvrement calcul/communication
- différentes tâches de priorité différentes
  - ordonnancement "temps réel mou"
  -
- Gestion des événements asynchrones
- Tâches indépendantes activées par des événements de fréquence irrégulière
  - Exemple : Serveur web peut répondre à plusieurs requêtes en parallèle
- Pour tirer profit des systèmes multi-cœurs

Mais plus risqué que les processus car moins de contrôle de l'OS (partage mémoire)

# Usage des fils: traitement de texte



- Un thread pour interagir avec l'utilisateur,
- Un thread pour reformater le texte en arrière plan,
- Un thread pour sauvegarder périodiquement le document

# PThreads (POSIX Threads)

- L'objectif premier des Pthreads est la portabilité (disponibles sous Linux, Windows ... ).

<b>Appel</b>	<b>Description</b>
pthread_create	Créer un nouveau fil d'exécution
pthread_exit	Terminer le fil appelant
pthread_join	Attendre la fin d'un autre fil
pthread_mutex_init	Créer un mutex
pthread_mutex_destroy	Détruire un mutex
pthread_mutex_lock	Verrouiller un mutex
pthread_mutex_unlock	Relâcher un mutex
pthread_cond_init	Créer une condition
pthread_cond_destroy	Détruire une condition
pthread_cond_wait	Attendre après une condition
pthread_cond_signal	Signaler une condition

# Fonction pthread\_create()

- **int pthread\_create(**

**pthread\_t \*tid,**

// sert à récupérer le TID du pthread créé

**const pthread\_attr\_t \*attr,**

// sert à préciser les attributs du pthread ` (taille de la pile, priorité....)

//attr = NULL pour les attributs par défaut

**void \* (\*func) (void\*),**

// est la fonction à exécuter par le pthread

**void \*arg);**

//le paramètre de la fonction.

- L'appel renvoie 0 s'il réussit, sinon il renvoie une valeur non nulle identifiant l'erreur qui s'est produite

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

exemple-pthread-create-1.c  
(C) 2000-2010 - Christophe BLAESS

```
void * fonction_thread(void * arg);
```

```
int main (void)
{
    pthread_t thr;
    if (pthread_create(& thr, NULL, fonction_thread, NULL) != 0) {
        fprintf(stderr, "Erreur dans pthread_create\n");
        exit(EXIT_FAILURE);
    }
    while (1) {
        fprintf(stderr, "Thread Main\n");
        sleep(1);
    }
}
```



- Remarque :
- si on fait un `ps aux` sur un autre terminal seul `./exemple-pthread-create-1` est visible
- Pour voir les threads il faut utiliser `ps maux`

```

Imascari 4498 0.0 0.0 14652 392 pts/3 - 10:42
0:00 ./exemple-pthread-create-1
Imascari - 0.0 - - -- SI+ 10:42 0:00 -
Imascari - 0.0 - - -- SI+ 10:42 0:00 -

```

# Passage de paramètres

void \* : cast (éventuellement d'une structure si plusieurs paramètres)

```
void * fonction_thread(void * arg);
```

```
#define NB_THREADS 5
```

```
int main (void) {
```

```
    pthread_t thr[NB_THREADS];
```

```
    long i; // attention à la taille sizeof(int) != sizeof (void *)
```

```
    for (i = 0; i < NB_THREADS; i ++) {
```

```
        if (pthread_create(& thr[i], NULL, fonction_thread,  
            (void *) i) != 0) {
```

```
            fprintf(stderr, "Erreur dans pthread_create\n");
```

```
            exit(EXIT_FAILURE);
```

```
        }
```

```
    }
```

```
    while (1) {
```

```
        fprintf(stderr, "Thread Main\n");
```

```
        sleep(1);
```

```
    }
```

```
}
```

exemple-pthread-create-2.c

(C) 2000-2010 - Christophe BLAESS

```
void * fonction_thread(void * arg)
{
long num = (long) arg;
while (1) {
    fprintf(stderr, "Thread numero %d\n", num+1);
    sleep(1);
}
}
```

```
$ ./exemple-pthread-create-2
Thread numero 4
Thread numero 3
Thread Main
Thread numero 2
Thread numero 1
Thread numero 5
Thread numero 3
Thread numero 4
Thread numero 2
Thread numero 1
Thread Main
```

6 messages par seconde.

```
typedef struct {  
    int X;  
    int Y;  
} coordonnee_t;
```

exemple-pthread-create-3.c  
(C) 2000-2010 - Christophe BLAESS

```
int main (void) {  
pthread_t thr;  
coordonnee_t * coord;  
coord = malloc(sizeof(coordonnee_t));  
if (coord == NULL) {  
    perror("malloc");  
    exit(EXIT_FAILURE);  
}  
coord->X=10; coord->Y=20;  
if (pthread_create(& thr, NULL, fonction_thread,  
                  coord) != 0) {  
fprintf(stderr, "Erreur dans pthread_create\n");  
exit(EXIT_FAILURE);  
}  
}
```

```
while (1) {  
    fprintf(stderr, "Thread Main\n");  
    sleep(1);  
}  
}
```

```
void * fonction_thread(void * arg) {  
    coordonnee_t * coord = (coordonnee_t *) arg;  
    int X = coord->X;  
    int Y = coord->Y;  
    free(coord);  
    while (1) {  
        fprintf(stderr, "Thread X=%d, Y=%d\n", X, Y);  
        sleep(1);  
    }  
}
```

```
$ ./exemple-pthread-create-3  
Thread Main  
Thread X=10, Y=20  
Thread Main  
Thread X=10, Y=20
```

# Partage d'espace mémoire

- Automatique : même espace mémoire !
- Mais sans protection...

```
#define NB_THREADS 5
```

```
int compteur = 0;
```

```
int main (void)
```

```
{
```

```
pthread_t thr[NB_THREADS];
```

```
long i;
```

```
for (i = 0; i < NB_THREADS; i ++)
```

```
{  
if (pthread_create(& thr[i], NULL, fonction_thread, (void *) i)  
0) {
```

```
fprintf(stderr, "Erreur dans pthread_create\n");
```

```
exit(EXIT_FAILURE);
```

```
}
```

```
}
```

```
while (1) {
```

```
fprintf(stderr, "Thread Main, compteur = %d\n", compteur);
```

```
sleep(1);
```

```
}
```

```
}
```

exemple-pthread-create-4.c

(C) 2000-2010 - Christophe BLAESS

```
void * fonction_thread(void * arg)
{
    long num = (long) arg;
    while (1) {
        fprintf(stderr, "Thread numero %d, compteur = %d \n", num+1, compteur);
        compteur ++;
        sleep(1);
    }
}
```

```
$ ./exemple-pthread-create-4
```

```
Thread numero 2, compteur = 0
```

```
Thread numero 3, compteur = 0
```

```
Thread Main, compteur = 0
```

```
Thread numero 5, compteur = 0
```

```
Thread numero 4, compteur = 0
```

```
Thread numero 1, compteur = 0
```

```
Thread numero 2, compteur = 5
```

```
Thread numero 3, compteur = 5
```

```
Thread Main, compteur = 6
```

```
Thread numero 5, compteur = 6
```

```
Thread numero 4, compteur = 7
```

```
Thread numero 1, compteur = 8
```

# Fin d'un thread

- Lorsqu'une la fonction principale d'un thread se termine, celui-ci est éliminé. La fonction doit renvoyer une valeur (void\*) qui pourra être récupérée dans un autre thread.
- Autre possibilité :

`void pthread_exit( void * retour);`

- Termine l'exécution du pthread et renvoie retour (éventuellement NULL)
- **JAMAIS exit()** sinon terminaison du processus.

```
int main (void)
{
    pthread_t thr[3];
    long i;
```

exemple-pthread-exit-1.c  
(C) 2000-2010 - Christophe BLAESS

```
for (i = 0; i < 3; i ++) {
    if (pthread_create(& thr[i], NULL, fonction_thread, (void *) i)
        != 0) {
        fprintf(stderr, "Erreur dans pthread_create\n");
        exit(EXIT_FAILURE);
    }
}
while (1) {
    fprintf(stderr, "Thread Main\n");
    sleep(1);
}
}
```

```

void * fonction_thread(void * arg)
{
long num = (long) arg;
long i = 0;
while (1) {
    fprintf(stderr, "Thread %d, iteration %d\n", num, i);
    if ((num == 0) && (i == 1))
        pthread_exit(NULL);
    if ((num == 1) && (i == 3))
        return NULL;
    i ++;
    sleep(1);
}
}

```

3 threads :

- 1 terminé au bout de 2 secondes,
- un second au bout de 4 secondes

```

$./exemple-pthread-exit-1
Thread Main
Thread 0, iteration 0
Thread 1, iteration 0
Thread 2, iteration 0
Thread Main
Thread 0, iteration 1
Thread 2, iteration 1
Thread 1, iteration 1
Thread Main
Thread 2, iteration 2
Thread 1, iteration 2
Thread Main

```

```
int main (void) {  
pthread_t thr[3];  
long i;
```

exemple-pthread-exit-2.c  
(C) 2000-2010 - Christophe BLAESS

```
for (i = 0; i < 3; i ++) {  
    if (pthread_create(& thr[i], NULL, fonction_thread, (void *) i)  
        != 0) {  
        fprintf(stderr, "Erreur dans pthread_create\n");  
        exit(EXIT_FAILURE);  
    }  
}  
pthread_exit(NULL);  
}  
void * fonction_thread(void * arg) {  
    long num = (long) arg;  
    while (1) {  
        fprintf(stderr, "Thread %d\n", num);  
        sleep(1);  
    }  
}
```

```
$/exemple-pthread-exit-2
```

```
Thread 0
```

```
Thread 1
```

```
Thread 2
```

```
Thread 0
```

```
Thread 2
```

```
Thread 1
```

```
Thread 2
```

- le thread principal se termine
- les autres continuent

```
int main (void) {
pthread_t thr;
if (pthread_create(& thr, NULL, fonction_thread, NULL) != 0) {
    fprintf(stderr, "Erreur dans pthread_create\n");
    exit(EXIT_FAILURE);
}
while (1) {
    fprintf(stderr, "Thread MAIN en fonctionnement\n");
    sleep(1);
}
}

void * fonction_thread(void * arg) {
    char * ptr = NULL;
    sleep(4);
    ptr[0] = 'A';
return NULL;
}
```

exemple-pthread-exit-4.c

(C) 2000-2010 - Christophe BLAESS

```
$/exemple-pthread-exit-3  
Thread MAIN en fonctionnement  
Thread MAIN en fonctionnement  
Thread MAIN en fonctionnement  
Thread MAIN en fonctionnement  
Erreur de segmentation (core dumped)
```

- un thread se “plante”
- tout le processus se termine !

# Fonctions pthread\_join()

```
void pthread_join( pthread_t tid, void * *retour);
```

- Attend la fin d'un pthread. L'équivalent de waitpid des processus sauf qu'on doit spécifier le tid du pthread à attendre.
- retour sert à récupérer la valeur de retour et l'état de terminaison.
- Peut échouer si le tid n'existe pas, s'il est détaché (plus tard)

```
int main (int argc, char * argv[]) {
pthread_t * thr = NULL;
void * ptr;
int i;
long n;
thr = calloc(argc - 1, sizeof(pthread_t));
if (thr == NULL) {
    perror("calloc");
    exit(EXIT_FAILURE);
}
for (i = 1; i < argc; i ++) {
    n = atoi(argv[i]);
    if (pthread_create(& thr[i-1], NULL, fonction_thread,
        (void *) n) != 0) {
        fprintf(stderr, "Erreur dans pthread_create\n");
        exit(EXIT_FAILURE);
    }
}
...
}
```

exemple-pthread-join-1.c

(C) 2000-2010 - Christophe BLAESS

...

```
for (i = 1; i < argc; i++) {  
    pthread_join(thr[i - 1], & ptr);  
    fprintf(stderr, "%d -> %d\n", atoi(argv[i]), (int) ptr);  
}  
return EXIT_SUCCESS;  
}
```

```
void * fonction_thread(void * arg)  
{  
    long num;  
    num = (int) arg;  
    return (void *) (num * num);  
}
```

```
$ ./exemple-pthread-join-1 1 2 3
1 -> 1
2 -> 4
3 -> 9
$
```

- autant de thread que d'arguments
- la fonction de chaque thread calcule le carré et le renvoie.
- le thread principal récupère les résultats et affiche

# Détachement des threads

- Un pthread détaché par la fonction  
**int pthread\_detach(pthread\_t tid) ;**

a pour effet de le rendre indépendant de celui qui l'a créé (pas de valeur de retour attendue).

Echoue si le thread n'existe pas ou déjà détaché.

- Pourquoi cette fonction ? parce que sinon les codes de retour (et ses piles) sont conservées : risque de saturation !
- Avec detach la pile et le code de retour sont libérés dès la fin du thread.

```

int main (int argc, char * argv[])
{
    pthread_t thr; /* La meme variable sera ecrasee a chaque fois */
    int nb_lances = 0;
    while (pthread_create(& thr, NULL, fonction_thread, NULL) == 0)
        pthread_detach(thr);
        nb_lances ++;
    if ((nb_lances % 100) == 0)
        fprintf(stderr, "%d thread lances\n", nb_lances);
    usleep(10000); /* 10 ms */
}
fprintf(stderr, "Echec de creation apres %d threads\n", nb_lances);
return EXIT_SUCCESS;
}

void * fonction_thread(void * arg)
{
    return NULL;
}

```

exemple-pthread-detach.c  
(C) 2000-2010 - Christophe BLAESS

```
$ ./exemple-pthread-detach  
100 thread lances  
...  
3000 thread lances  
^C  
$
```

- avec le detach “pas de limite”

# Comparaison de threads

```
int pthread_equal(pthread_t tid1, pthread_t  
tid2) ;
```

renvoie une valeur non nulle si les threads sont égaux.

- Attention : le test sur les tid seuls ne marche pas !  
pthread\_t est vu comme un entier mais il existe une structure interne...

# Synchronisations entre threads : mutex

- Deux états : libre ou verrouillé
- Un thread qui verrouille un mutex *tient le mutex*
- Un seul thread peut tenir un mutex donné
- En général variables globales ou locales statiques.

- Initialisation statique :

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- Initialisation dynamique :

```
int pthread_mutex_init(pthread_mutex_t * mutex,  
const pthread_mutexattr_t *attributs);  
attributs peut être NULL (défauts)
```

# Synchronisations entre threads : mutex

- Libération :

```
int pthread_mutex_destroy(pthread_mutex_t *);
```

- verrouillage :

- ```
int pthread_mutex_lock(pthread_mutex_t * mutex);
```

- si mutex libre, verrouillé et attribué au thread appelant

- sinon blocage jusqu'à libération, puis verrouillé et attribué au thread appelant

- ```
int pthread_mutex_trylock(pthread_mutex_t * mutex);
```

- Idem mais échoue avec EBUSY si verrouillé (pas de blocage)

- Déconseillé

# Synchronisations entre threads : mutex

- déverrouillage :

- `int pthread_mutex_unlock(pthread_mutex_t * mutex);`

exemple-pthread-mutex.c  
(C) 2000-2010 - Christophe BLAESS

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
```

```
static void * routine_threads (void * argument);
static int aleatoire (int maximum);
```

```
pthread_mutex_t mutex_stdout = PTHREAD_MUTEX_INITIALIZER;
```

```
int main (void)
```

```
{
```

```
int i;
```

```
pthread_t thread;
```

```
for (i = 0; i < 5; i ++)
```

```
pthread_create(& thread, NULL, routine_threads, (void *) i);
```

```
pthread_exit(NULL);
```

```
}
```

```
static void * routine_threads (void * argument) {
    int numero = (int) argument;
    int nombre_iterations;
    int i;
    nombre_iterations = 1 + aleatoire(3);
    for (i = 0; i < nombre_iterations; i ++) {
        sleep(aleatoire(3));
        pthread_mutex_lock(& mutex_stdout);
        fprintf(stdout, "Le thread %d a obtenu le mutex\n", numero);
        sleep(aleatoire(3));
        fprintf(stdout, "Le thread %d relache le mutex\n", numero);
        pthread_mutex_unlock(& mutex_stdout);
    }
    return NULL;
}
```

```
static int aleatoire (int maximum) {
    double d;
    d = (double) maximum * rand();
    d = d / (RAND_MAX + 1.0);
    return ((int) d);
}
```

```
$ ./exemple-pthread-mutex-1
Le thread 0 a obtenu le mutex
Le thread 0 relache le mutex
Le thread 4 a obtenu le mutex
Le thread 4 relache le mutex
Le thread 1 a obtenu le mutex
Le thread 1 relache le mutex
Le thread 2 a obtenu le mutex
Le thread 2 relache le mutex
Le thread 3 a obtenu le mutex
Le thread 3 relache le mutex
Le thread 3 a obtenu le mutex
Le thread 3 relache le mutex
...
$
```

```
void handler(int sig) {}
```

```
int main(void) {  
    pid_t pid;  
    int i;  
  
    if (signal(SIGUSR1, handler) == SIG_ERR) {  
        perror("signal");  
        exit(1);  
    }
```

```
    switch (pid = fork()) {  
        case -1:  
            perror("fork");  
            exit(1);  
        case 0:  
            /* fils */  
            pid = getppid();  
            for (i = 1; i <= 100; i += 2) {  
                printf("%d\n", i);  
                kill(pid, SIGUSR1);  
                pause();
```

1) le fils est suspendu juste après le kill()  
5) il reçoit le signal : exécution du handler(), puis  
pause()

```
            }  
            break;  
        default:  
            /* père */  
            for (i = 2; i <= 100; i += 2) {  
                pause();  
                printf("%d\n", i);  
                kill(pid, SIGUSR1);  
            }
```

2) le noyau donne la main au père, qui est en pause()  
3) Il se réveille, affiche le nombre suivant,  
4) émet le signal puis pause()

```
    }  
}
```