



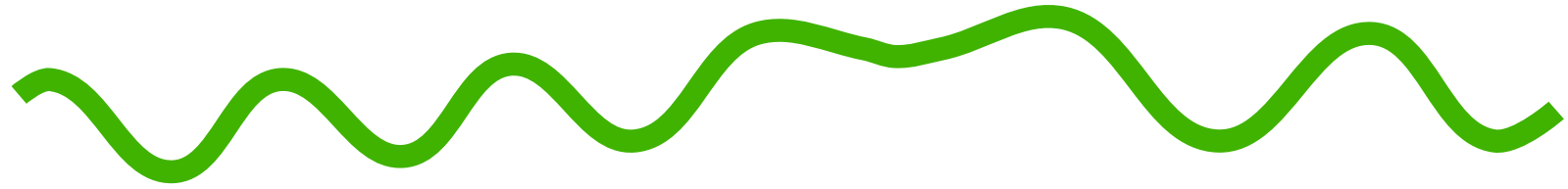
Introduction au multithreading

Maîtrise d'informatique

Philippe MARQUET

`Philippe.Marquet@lifl.fr`

Laboratoire d'informatique fondamentale de Lille
Université des sciences et technologies de Lille



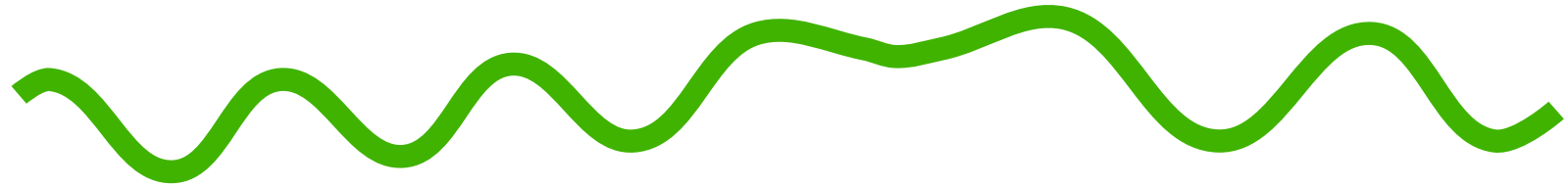
~ Création janvier 1999

~ Ce cours est diffusé sous la licence GNU Free Documentation License,
<http://www.gnu.org/copyleft/fdl.html>

~ La dernière version de ce cours est accessible à partir de
<http://www.lifl.fr/~marquet/ens/pp/>

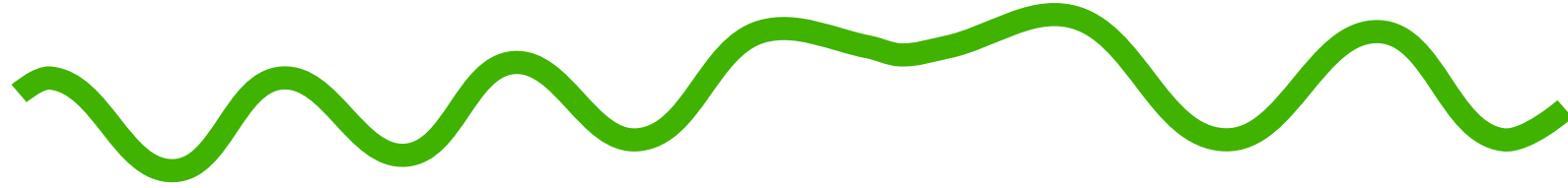
~ \$Id: mt.tex,v 1.7 2002/03/18 09:12:01 marquet Exp \$

Table des matières

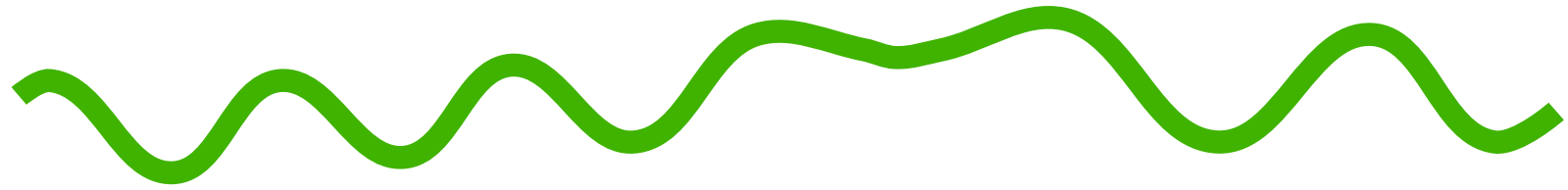


- Processus légers
- Critique des processus légers
- Interface `pthread`
- Programmer avec les threads
- Implantations des threads
- Compléments sur l'interface `pthread`
- Utilisation de threads et de bibliothèques
- Ce qui n'a pas été vu

Références et remerciements

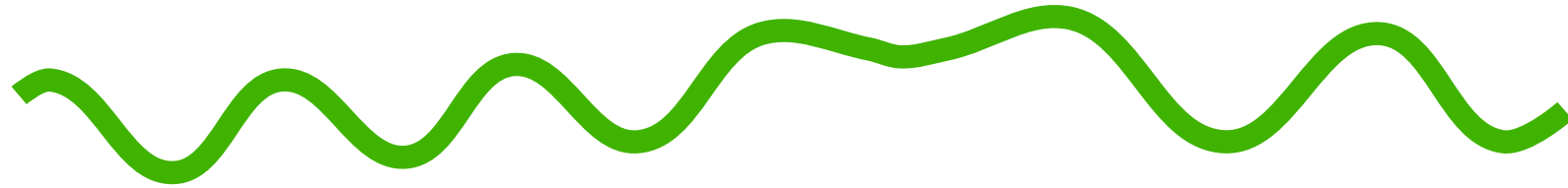


- ✓ *Programming with threads*
Steve KLEIMAN, Devang SHAH, Bart SMAALDERS
SunSoft Press, Prentice Hall, 1996
<http://www.sun.com/smi/ssoftpress/threads/>
- ✓ *Multithreaded Programming — Concepts and Practice*
Bil LEWIS
1999, <http://www.lambdaCS.com/>
- ✓ *PM² : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières*
Raymond NAMYST
Thèse, LIFL, Université de Lille 1, 1997
- ✓ *Programmer avec les threads Unix*
Charles NORTHRUP
John Willey & Sons, 1996, traduction française chez International Thomson
Publishing France, 1997
- ✓ Manuel en ligne : `man pthreads`



Processus légers

Les processus légers



~ Threads ou processus légers

~ Initialement dans les systèmes d'exploitation

~ recouvrement des opérations d'I/O (serveurs multithreadés)

~ exploitation des architectures multiprocesseurs

~ Thread =

séquence d'exécution du code d'un programme,
au sein d'un processus

~ Processus habituel : un seul thread

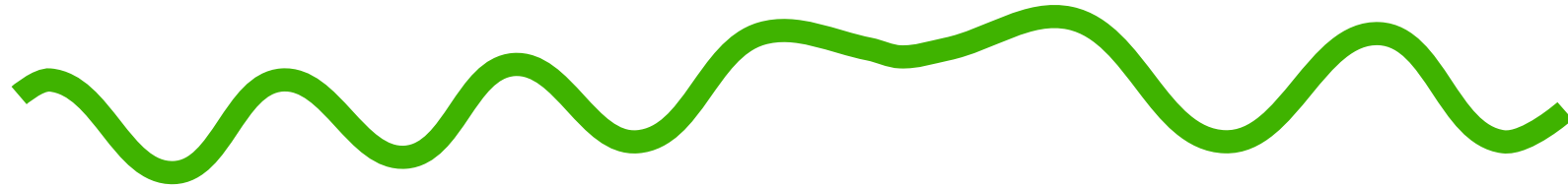
~ Processus multithreadé : plusieurs flots d'exécution simultanés

~ Les différents threads partagent les ressources systèmes

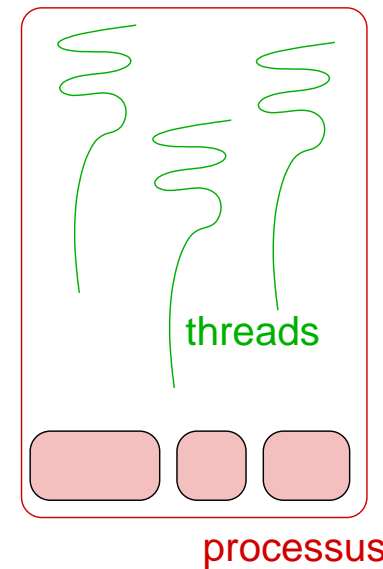
~ Principe

détacher les flots d'exécution des ressources systèmes

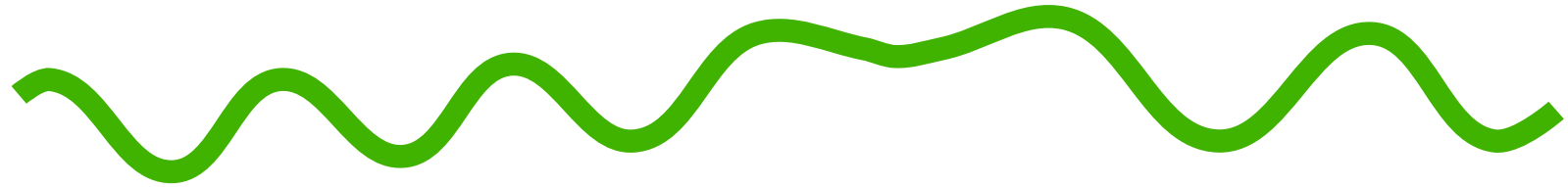
Les processus légers (cont'd)



- ✓ Un processus léger ne peut exister qu'au sein d'un processus lourd
- ✓ Les ressources d'un processus lourd sont partagées par tous les processus légers qu'il contient :
 - ✓ code
 - ✓ environnement de shell
 - ✓ mémoire
 - ✓ répertoire de travail courant
 - ✓ fichiers
 - ✓ droits Unix
- ✓ Un processus léger possède
 - ✓ sa propre pile d'exécution
 - ✓ indentificateur de thread
 - ✓ pointeur d'instruction
- ✓ Un processus (Unix) peut contenir plusieurs centaines de threads



Les processus légers (cont'd)



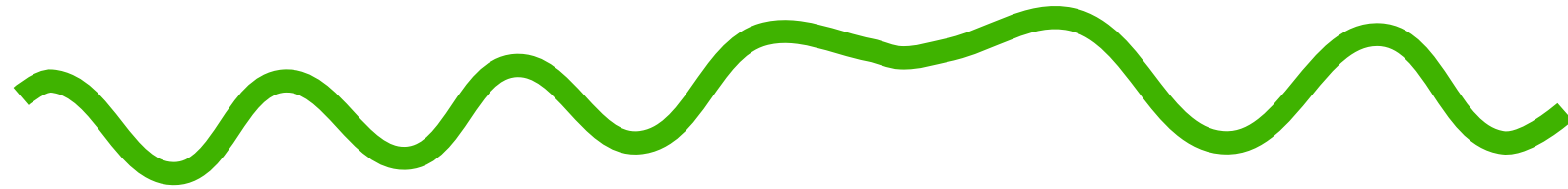
✓ Opérations relatives aux processus légers performantes

- ✓ création
- ✓ changement de contexte
- ✓ synchronisation
- ✓ ...

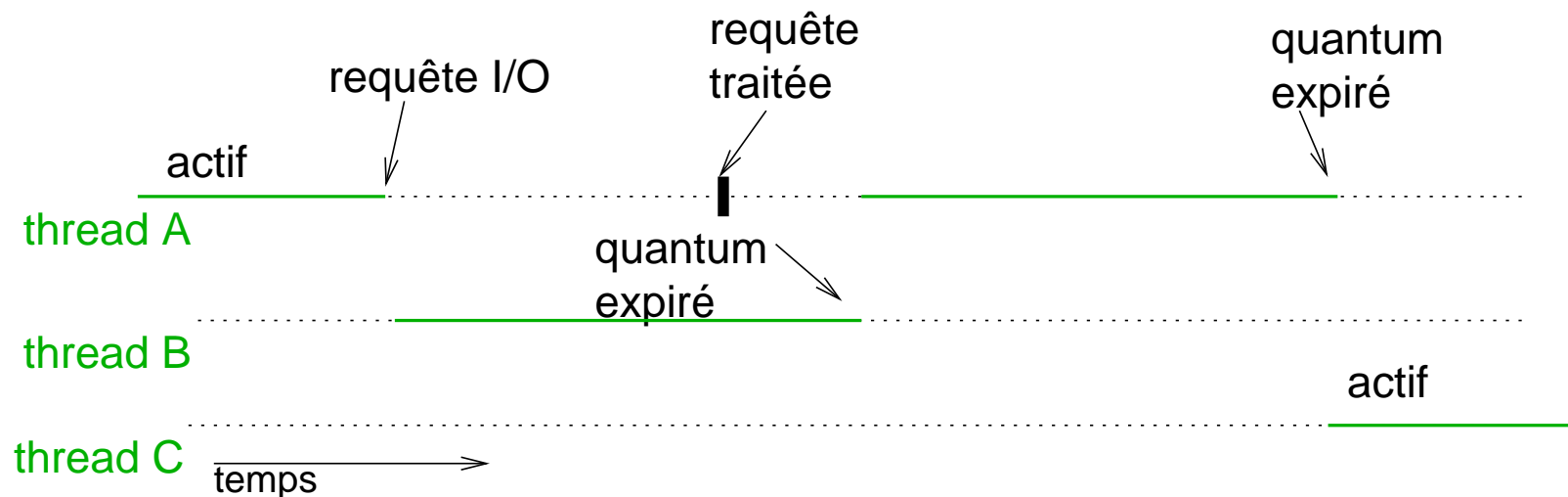
✓ Création d'un processus léger

- ✓ un processus léger exécute du code
 - adresse d'une fonction (+ paramètres)
- ✓ un processus léger possède une pile d'exécution
 - taille de la pile (valeur par défaut)
- ✓ les processus légers se partagent le temps processeur alloué au processus
 - priorité

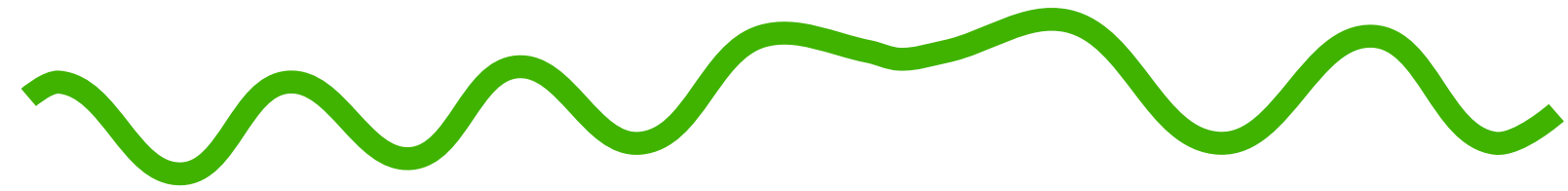
Les processus légers (cont'd)



- ✓ Exécution des processus légers au sein du processus
→ ordonnancement au sein du processus
- ✓ Typiquement, un autre thread prend la main quand :
 - ✓ le thread courant se bloque (I/O), ou
 - ✓ le thread courant a épuisé le quantum de temps qui lui été alloué

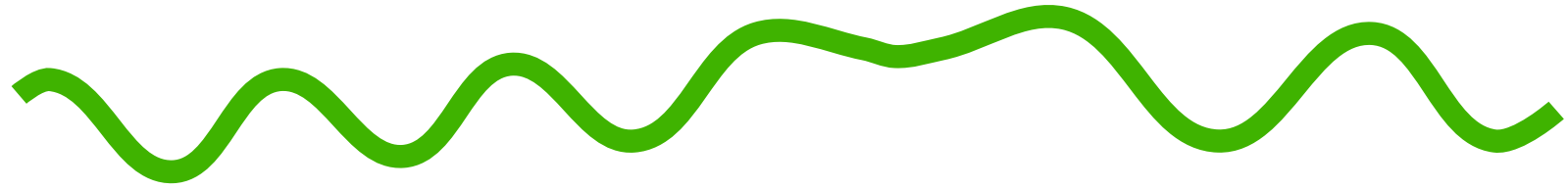


- ✓ Multiprocesseurs : possibilité de plusieurs threads actifs



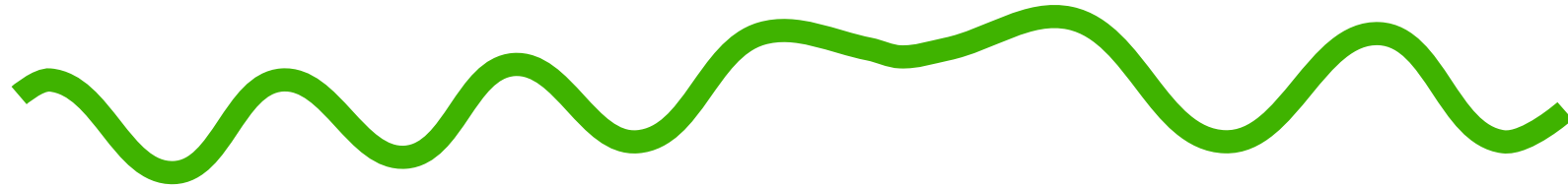
Critique des processus légers

Avantages des threads



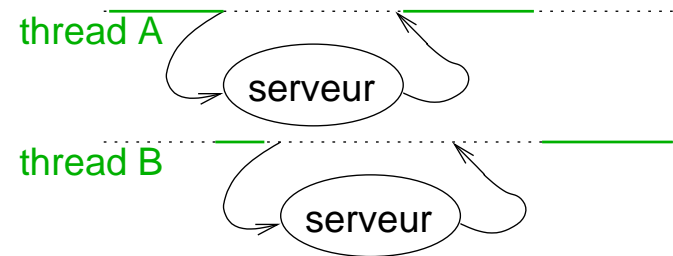
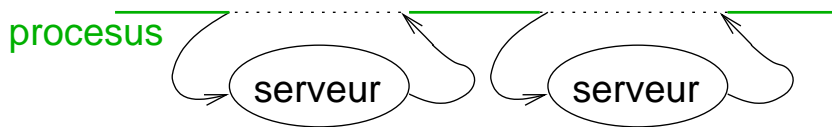
- ✓ Deux aspects :
 - ✓ systèmes d'exploitation
 - ✓ applications concurrentes
- ✓ Quelques exemples
 - ✓ débit
 - ✓ multiprocesseurs
 - ✓ réactivité (interface utilisateur)
 - ✓ réactivité d'un serveur
 - ✓ robustesse/fiabilité d'un serveur
 - ✓ structure d'un programme

Avantages des threads (cont'd)



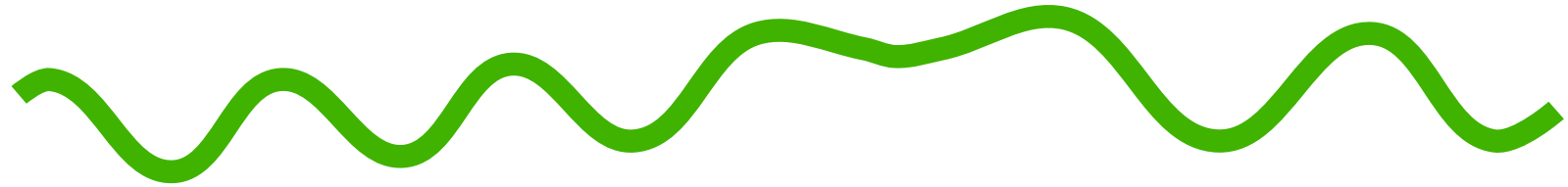
~ Débit

- ~ un seul thread
 - attente à chaque requête au système
- ~ plusieurs threads
 - possible superposition d'exécution et d'attente de retour de requête
- ~ le thread qui a fait la requête attend
- ~ un autre thread peut poursuivre son exécution
- ~ exemple : un programme fait plusieurs requêtes



- ~ attentes parallèles

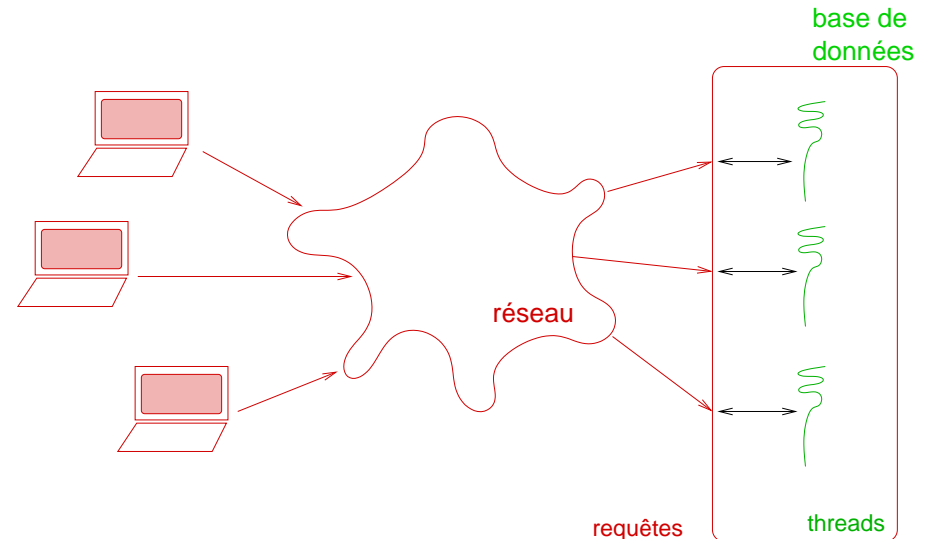
Avantages des threads (cont'd)



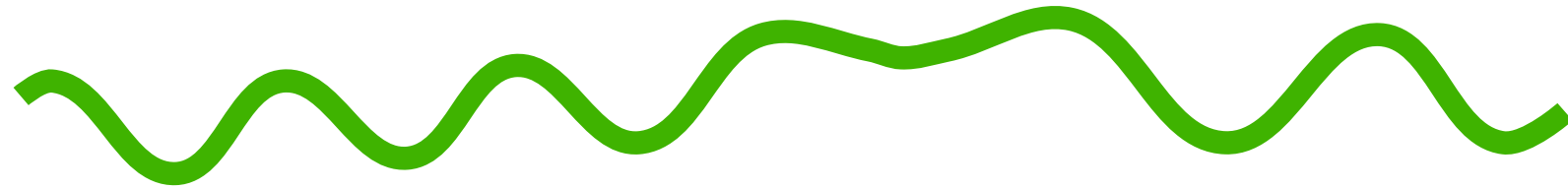
- ✓ Multiprocesseurs
 - ✓ machine multiprocesseurs
 - ✓ plusieurs threads au sein d'une application
 - utilisation effective du parallélisme
 - ✓ pas aussi simple !

Avantages des threads (cont'd)

- ✓ Réactivité d'un serveur
 - ✓ un serveur dans une application client/serveur
 - ✓ possibilité de plusieurs requêtes simultanées depuis différents clients
 - ✓ un unique thread dans le serveur → une requête bloque le serveur si une autre requête plus prioritaire ?
 - ✓ serveur multithreadé
 - ✓ une requête = un thread
 - ✓ priorités de requêtes : ordonnancement des threads



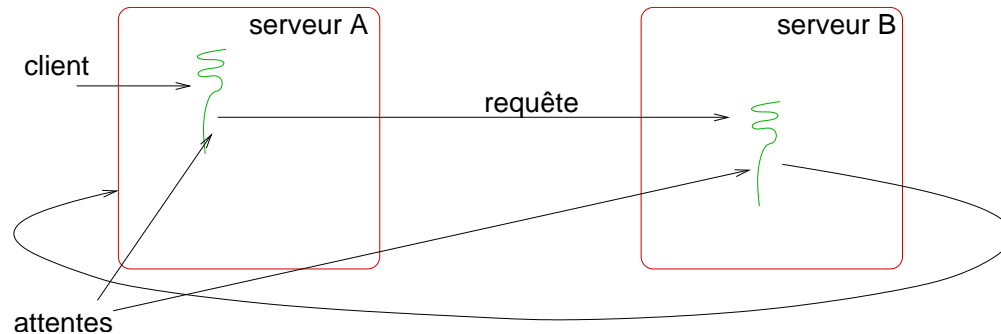
Avantages des threads (cont'd)



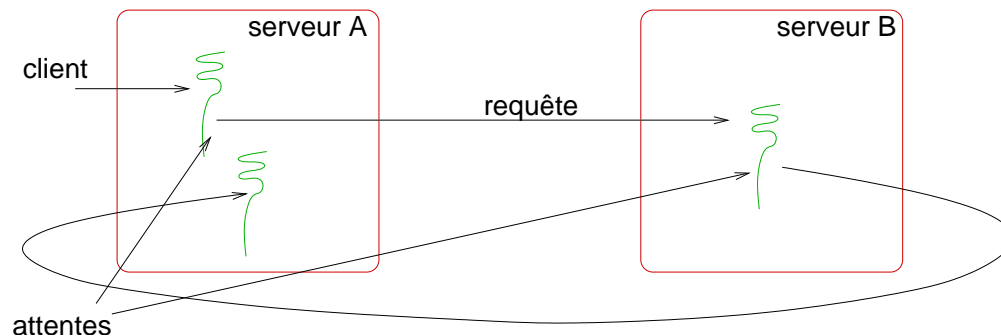
~ Serveur sans dead-lock

- ~ application client/serveur
- ~ serveur est aussi client
- ~ d'autres serveurs
- ~ de lui-même

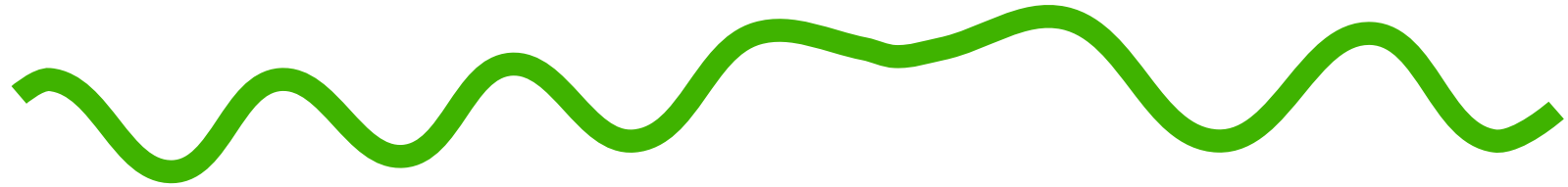
- ~ exemple : serveur « base de données »
qui utilise un serveur de noms
qui utilise le serveur de BD !



~ solution : serveur multithreadé



Avantages des threads (cont'd)

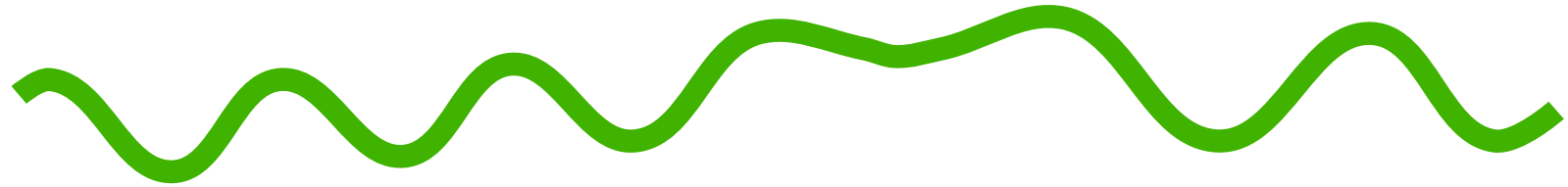


~ Structure de programmes

- ~ exemple : une feuille de calcul
 - ~ nouveau calcul des cellules à chaque modification
 - ~ continue de gérer l'interface utilisateur
- ~ code mono-thread

```
for ( ; ; ) {  
    while ( nouvelle_maj_necessaire()  
           && ! entree_utilisateur_disponible() ) {  
        recalculer_une_formule();  
    }  
    cmd = get_entree_utilisateur();  
    traiter_entree_utilisateur(cmd);  
}
```

Avantages des threads (cont'd)



~ Structure de programmes

~ exemple : une feuille de calcul

~ nouveau calcul des cellules à chaque modification

~ continue de gérer l'interface utilisateur

~ code multithreadé : 2 threads indépendants = 2 codes **indépendants**

~ premier thread

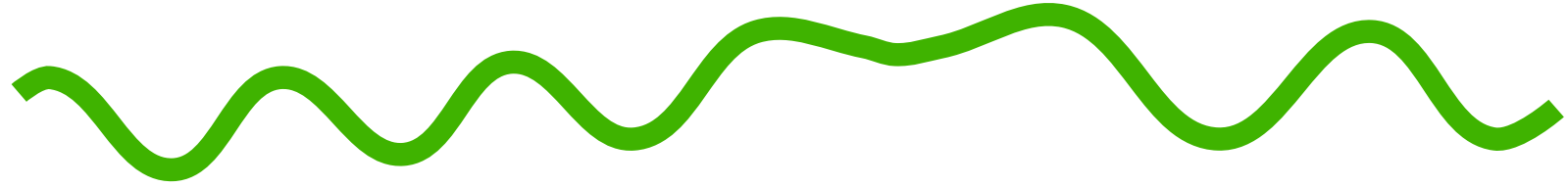
```
for (;;) {  
    cmd = get_entree_utilisateur()  
    traiter_entree_utilisateur(cmd);  
}
```

~ second thread

```
for (;;) {  
    attendre_un_changement()  
    for (toutes_les-formules)  
        recalculer_une_formule();  
}
```

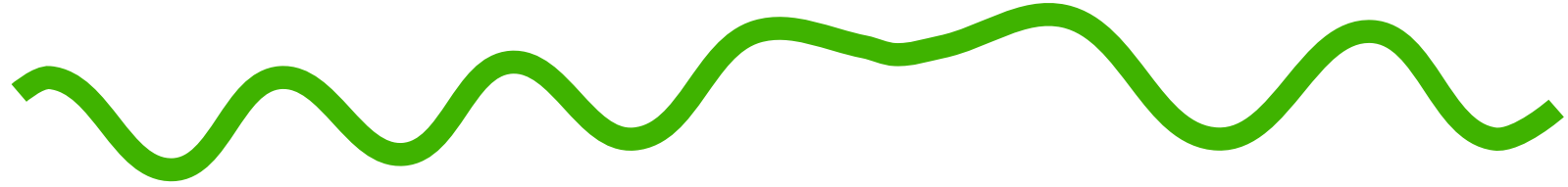
~ expression plus naturelle !

Inconvénients des threads ?



- ~ Cas limites
- ~ Contrôle de la concurrence
- ~ Contraintes techniques
- ~ Savoir faire

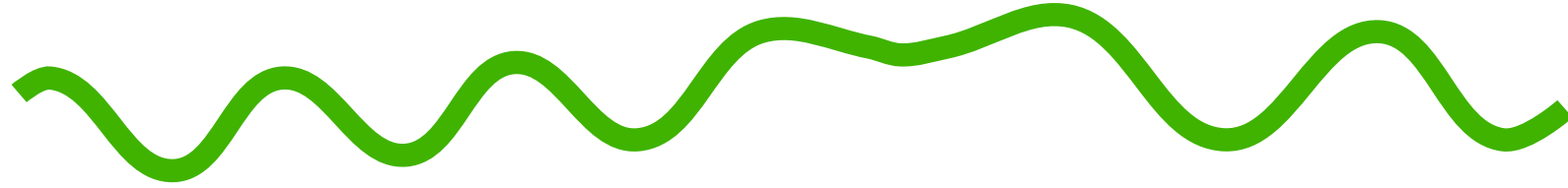
Inconvénients des threads ?



- ✓ Cas limites
- ✓ Contrôle de la concurrence
- ✓ Contraintes techniques
- ✓ Savoir faire

- ✓ Cas limites
 - ✓ application = un flot unique de calcul à accélérer sur un monoprocesseur
 - ✓ chaque séquence qui pourrait s'exécuter en parallèle → un thread

Contrôle de la concurrence



~ Toute action ne peut être faite concurremment

~ Exemple

```
counter++;
```

```
LOAD @"COUNTER" R1
INCR R1
STOR R1 @"COUNTER"
```

~ deux threads, un processeur

```
counter++;
```

```
counter++;
```

```
LOAD @"COUNTER" R1
```

```
LOAD @"COUNTER" R2
INCR R2
STOR R2 @"COUNTER"
```

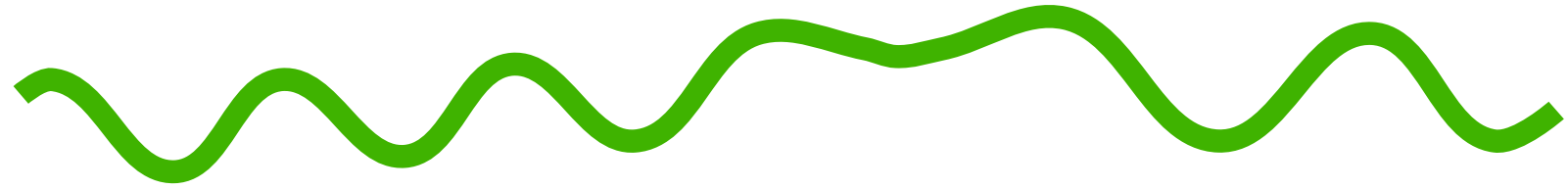
```
INCR R1
```

```
STOR R1 @"COUNTER"
```

on a perdu une incrémentation

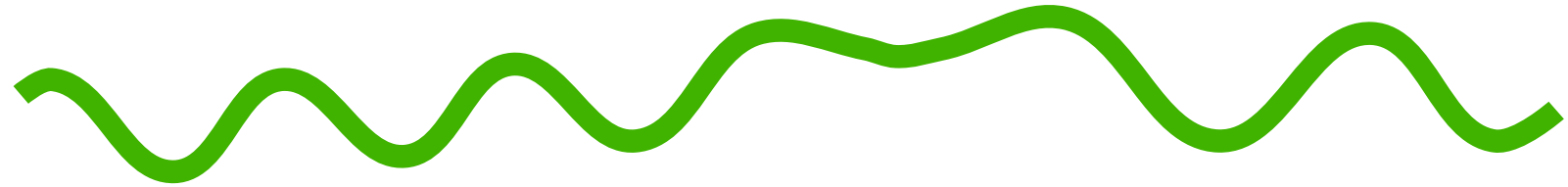
~ Il faut contrôler la concurrence : mécanismes ad hoc

Contraintes techniques



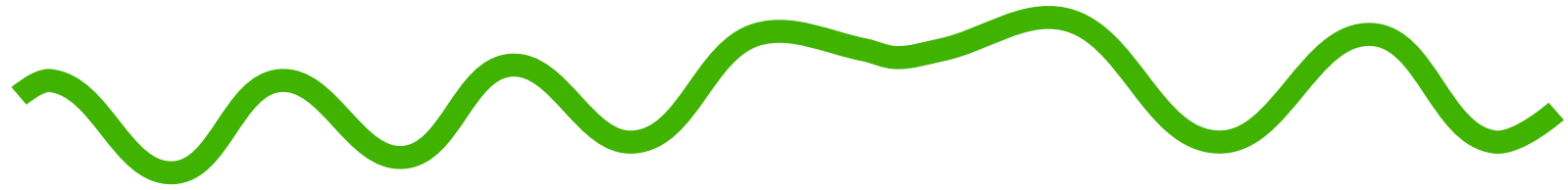
- ~ Dimensionnement des piles d'exécution
 - ~ processus légers : propre pile d'exécution
 - ~ variables locales
 - ~ pile d'exécution des fonctions appelées
 - ~ appels récursifs profonds...
 - ~ allouer systématiquement une « très grande pile »
 - ~ coût mémoire
 - ~ « très grande pile » ?
 - nombre d'appels récursifs ?
 - architecture cible
 - compilateur utilisé
 - ~ stratégie essai/erreur
 - ~ erreur ?
 - ~ agrandir la pile d'exécution ?

Contraintes techniques (cont'd)



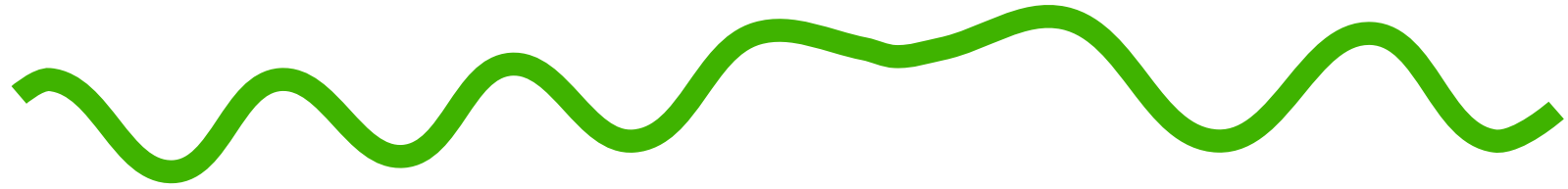
~ Réentrance du code

- ~ threads d'un même processus
 - ~ partage efficace de la mémoire
 - ~ avantage utilisateur
- ~ problème pour les variables globales gérées par les bibliothèques (exemple : `malloc()`)
 - ~ idée : contrôler la concurrence de l'accès à ces variables
 - ~ quid ? si code source des bibliothèques non disponible ?
 - ~ idée : contrôler la concurrence de l'accès à la bibliothèque
 - ~ quid ? si appel à la bibliothèque depuis une autre bibliothèque ?



Interface `pthread`

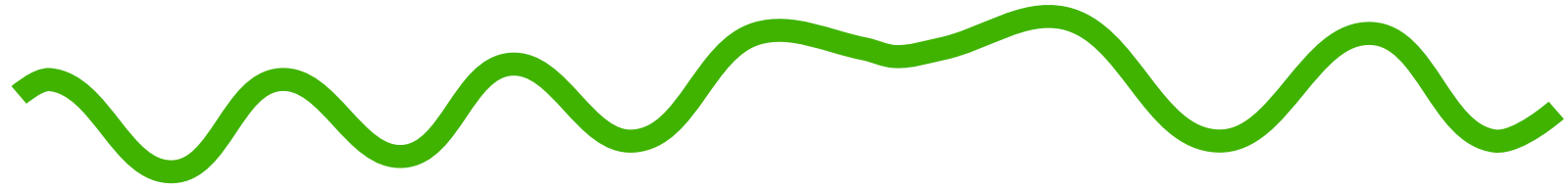
Interface *pthread*



- ✓ `pthread` : threads POSIX
- ✓ POSIX : IEEE Portable Operating System Interface, X as in UniX
- ✓ Unix threads interface P1003.1c
- ✓ Implantations diverses (et différentes) fournissant cette même API

- ✓ Illustration de l'API sur un exemple :
 - ✓ applications affichant
 - ✓ des images compressées lues
 - ✓ depuis un ensemble de fichiers
- ✓ Idée :
 - ✓ décompresser l'image suivante
 - ✓ pendant visualisation de la courante

Exemple : `get_next_picture()`

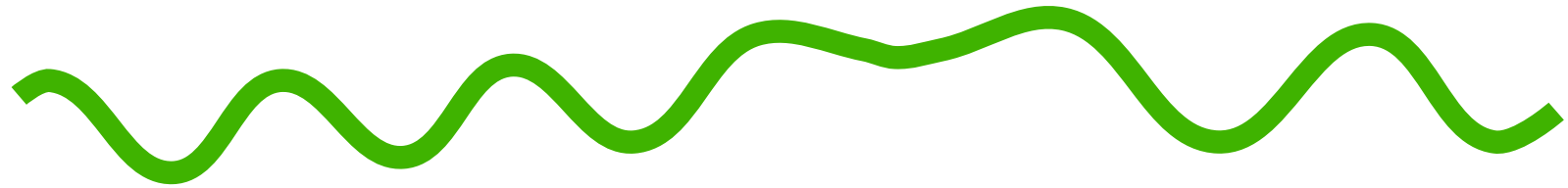


~ Fonction `get_next_picture()`

- ~ ouvre le fichier
- ~ alloue un buffer
- ~ décompresse l'image du fichier dans le buffer

```
void *
get_next_picture (void *arg)
{
    int fd;
    pic_buf_t *buf;
    fd = open((char*) arg, O_RDONLY);
    buf = (pic_buf_t*) malloc(...);
    while (nbytes = read(fd,...)) {
        /* decompress data into buf */
    }
    close(fd);
    return(buf);
}
```

Exemple : main()



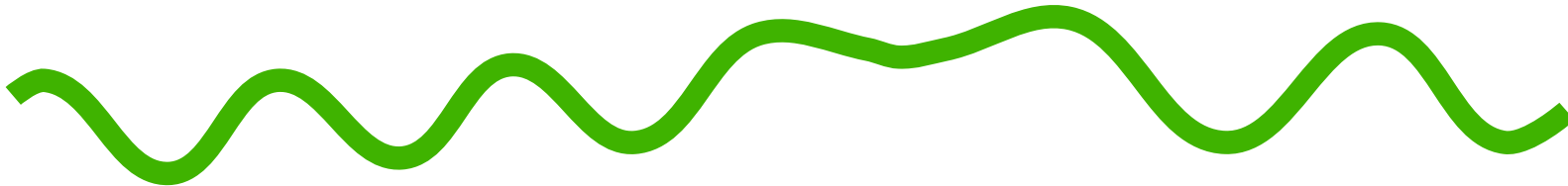
Programme principal

- ✓ appel de `get_next_picture()` pour lire la première image
- ✓ affiche l'image et lance la lecture suivante
- ✓ attend continuation de l'utilisateur

```
int main(argc, char *argv[])
{
    pthread_t pic_thread;
    int fileno;
    pic_buf_t *buf;

    for (fileno=1 ; fileno<argc ; fileno++) {
        if (fileno==1) { /* premiere lecture, sans thread */
            buf = get_next_picture((void*)argv[fileno]);
        } else { /* attend la fin de la decompression */
            pthread_join(pic_thread, (void**)&buf);
        }
        if (fileno<argc-1) /* lance un thread pour l'image suivante */
            pthread_create(&pic_thread, NULL, get_next_picture,
                (void*) argv[fileno+1]);
    }
}
```

Exemple : main() (cont'd)



```
int main(argc, char *argv[])
{
    pthread_t pic_thread;
    int fileno;
    pic_buf_t *buf;

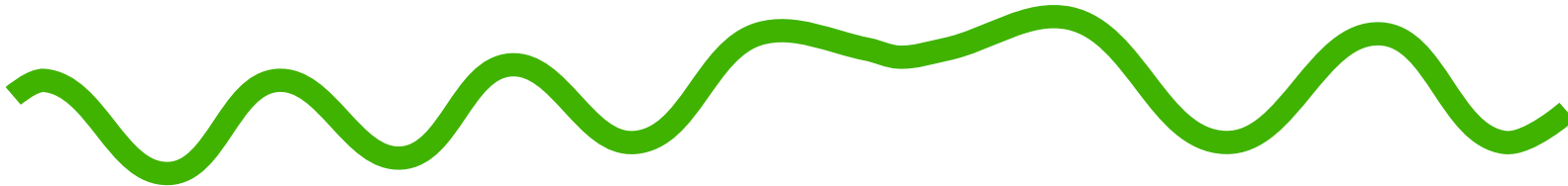
    for (fileno=1 ; fileno<argc ; fileno++) {
        if (fileno==1) {                /* premiere lecture, sans thread */
            buf = get_next_picture((void*)argv[fileno]);
        } else {                        /* attend la fin de la decompression */
            pthread_join(pic_thread, (void**)&buf);
        }
        if (fileno<argc-1) /* lance un thread pour l'image suivante */
            pthread_create(&pic_thread, NULL, get_next_picture,
                (void*) argv[fileno+1]);
        display_buf(buf);                /* affiche l'image */
        free(buf);                        /* libere le buffer */
        if (getchar() == EOF) /* attend un caractere utilisateur */
            break;
    }
}
```

Création d'un thread

```
int pthread_create(  
    pthread_t *threadid,  
    const pthread_attr_t *attr,  
    void *(*threadfunction)(void*),  
    void *arg);
```

- ✓ Crée un nouveau thread
 - ✓ qui exécute la fonction `threadfunction()` à un argument `arg`
 - ✓ (attributs du thread `attr`, défaut : `NULL`)
 - ✓ retourne un identifiant de thread `threadid`
 - ✓ retourne une valeur nulle ou erreur (`errno.h`)
- ✓ Au plus `PTHREAD_THREADS_MAX` (`limit.h`) threads
- ✓ Argument unique `void*` de la fonction
plusieurs arguments \Rightarrow structure...

Création d'un thread (cont'd)



```
struct two_args {
    int arg1;
    int arg2;
}
void *needs_two_args(void *ap)
{
    struct two_args *args = (void*)ap;
    int a1, a2;

    a1 = args-> arg1;
    a2 = args-> arg2;
    ...
}
```

Création d'un thread (cont'd)

```
pthread_t a()  
{  
    pthread_t t;  
    struct two_args args;  
    int error;  
  
    args.arg1 = ...  
    args.arg2 = ...  
    error = pthread_create(&t, NULL,  
                           needs_two_args, (void*)&args);  
    if (error)  
        /* traitement de l'erreur */  
    return(t);  
}
```

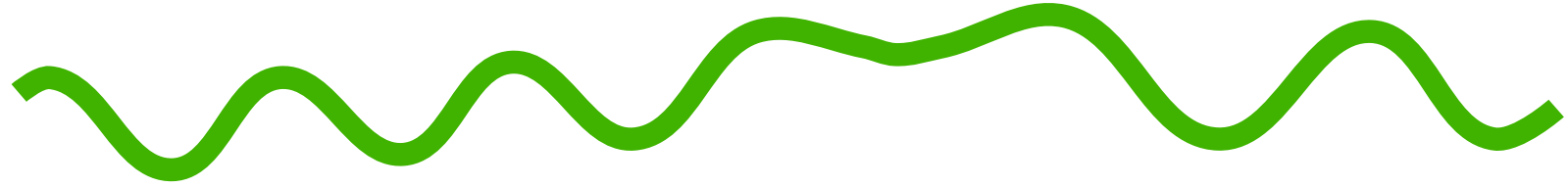
~ Bug !!

Création d'un thread (cont'd)

```
pthread_t a()  
{  
    pthread_t t;  
    struct two_args *ap = malloc(sizeof(struct two_args));  
    int error;  
  
    ap->arg1 = ...  
    ap->arg2 = ...  
    error = pthread_create(&t, NULL,  
                           needs_two_args, (void*)ap);  
    if (error)  
        /* traitement de l'erreur */  
    return t;  
}
```

~ Libération de la mémoire dans `needs_two_args()`

Terminaison d'un thread

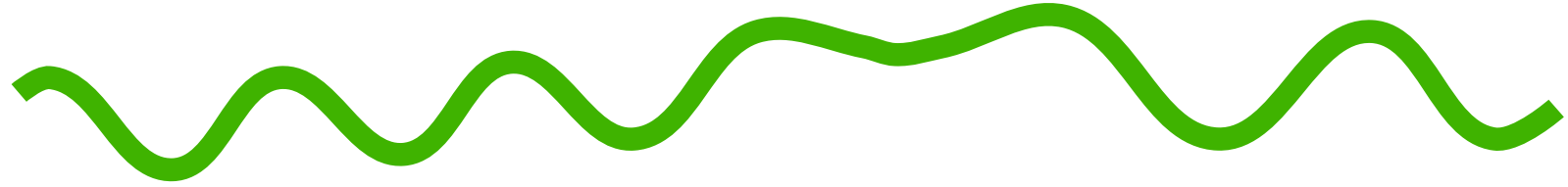


- ~ Au retour de la fonction, le thread se termine normalement
- ~ Se faire terminer un thread :

```
int pthread_exit(  
    void *status);
```

- ~ La valeur `status` est associée à l'identifiant du thread
- ~ Consultable par un autre thread

Attente de la terminaison d'un thread



~ Tel thread a-t-il terminé ?

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

~ Retourne le status du thread

- ~ à la terminaison du thread
- ~ ou si il a déjà terminé

~ Une fois joint un thread n'a plus d'existence

~ Attention à l'allocation du status du thread
pas dans la pile du thread qui a terminé

Identification d'un thread



Qui suis-je ?

- à la création d'un thread, `threadid` → père
- mon propre identifiant ?

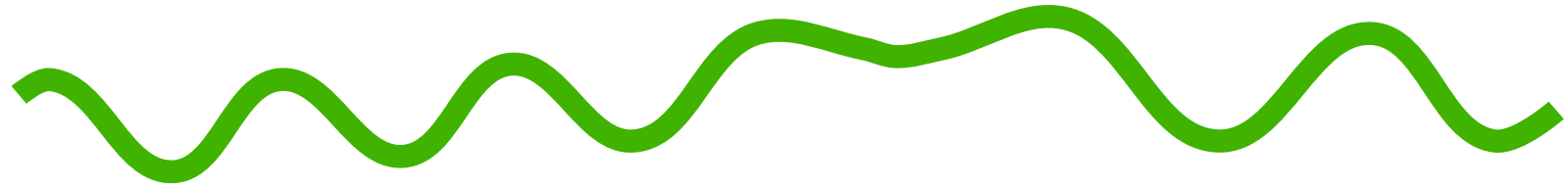
```
pthread_t pthread_self();
```

Égalité

- le type `pthread_t` : identificateur de thread
- comparaison

```
int pthread_equal(  
    pthread_t t1,  
    pthread_t t2);
```

Verrous mutex



- ✓ Accès en exclusion mutuelle à une section critique
- ✓ Protection de la section critique par un mutex
- ✓ Utilisation

```
lock(mutex)
... /* section critique */
unlock(mutex)
```

- ✓ Interface pthread
 - ✓ pthread_mutex_t
 - ✓ pthread_mutex_init()
 - ✓ pthread_mutex_destroy()
 - ✓ pthread_mutex_lock()
 - ✓ pthread_mutex_unlock()
 - ✓ pthread_mutex_trylock()

Création/destruction de mutex



Allocation

```
pthread_mutex_t lock;
```

ou

```
pthread_mutex_t *mp;
```

```
mp = malloc(sizeof(pthread_mutex_t));
```

Initialisation obligatoire

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

ou

```
int pthread_mutex_init(  
    pthread_mutex_t *mp,  
    const pthread_mutexattr_t *attr);
```

valeur par défaut des attributs → NULL

Destruction

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

Création/destruction de mutex (cont'd)



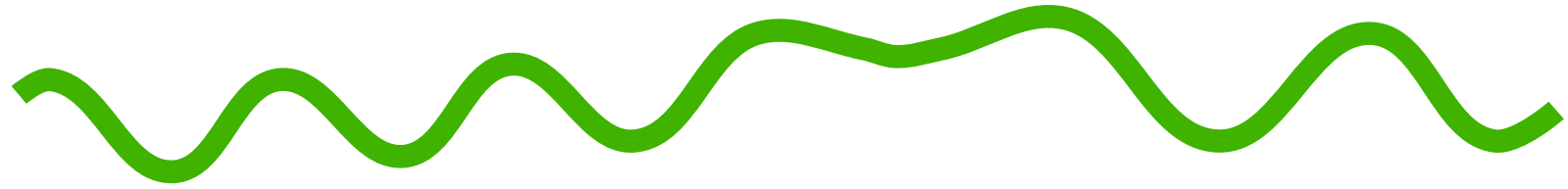
```
struct data_record {
    pthread_mutex_t data_lock;
    char data[128];
}

/* Create and initialize a new data record */
struct data_record *
new_record()
{
    struct_data_record *recp;

    recp = malloc(sizeof(struct data_record));
    pthread_mutex_init(&recp->data_lock, NULL);
    return(recp);
}

/* Delete a data record. The caller must ensure that no threads are
   waiting to acquire this record's data_lock */
void
delete_record (struct data_record *recp)
{
    pthread_mutex_destroy(&recp->data_lock);
    free((void*) recp);
}
```

Verrouillage/déverrouillage d'un mutex



Lock

```
int pthread_mutex_lock(  
    pthread_mutex_t *mp);
```

Unlock

```
int pthread_mutex_unlock(  
    pthread_mutex_t *mp);
```

Propriétaire du mutex

- le thread qui réalise le verrouillage

Déverrouillage

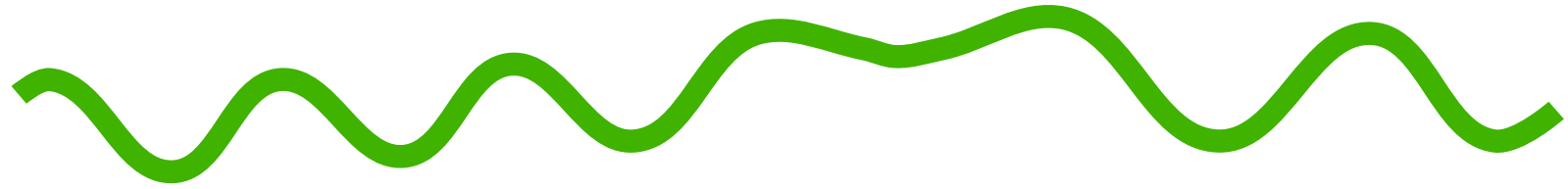
- par le propriétaire uniquement
- doit avoir été verrouillé

Tentative de verrouillage

```
int pthread_mutex_trylock(  
    pthread_mutex_t *mp);
```

- succès si le mutex est libre (retourne 0), devient propriétaire
- retour de EBUSY sinon

Verrouillage/déverrouillage d'un mutex (cont'd)

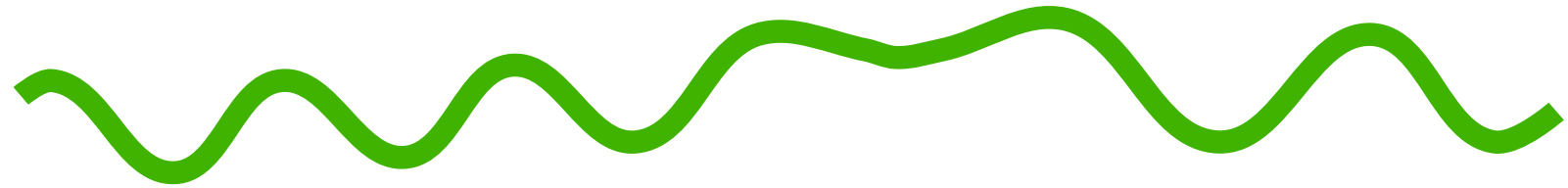


~ Exemple : synchronisation des accès à un fichier
ne pas mélanger les chaînes affichées

```
void
write_strings(char *strings[], int nstring)
{
    int i;
    static pthread_mutex_t file_lock=PTHREAD_MUTEX_INITIALIZER;

    pthread_mutex_lock(&file_lock);
    for (i=0 ; i<nstring ; i++) {
        write(1, strings[i], strlen(strings[i]));
    }
    pthread_mutex_unlock(&file_lock);
}
```


Conditions



- ✓ Mutex : séquentialisation des accès

- ✓ Attente d'un état

 - ✓ buffer non vide,

 - ✓ remplissage du buffer par un autre thread...

 - condition

- ✓ Condition :

 - ✓ utilisée avec un mutex

 - ✓ attente d'une condition, d'un état

- ✓ Opérations de base

 - ✓ attendre, wait

 - ✓ signaler, signal

- ✓ Interface pthread

 - ✓ pthread_cond_t

 - ✓ pthread_cond_init()

 - pthread_cond_destroy()

 - ✓ pthread_cond_wait()

 - pthread_cond_signal()

 - pthread_cond_timedwait()

 - pthread_cond_broadcast()

Création/destruction de conditions



Allocation

```
pthread_cond_t lock;
```

ou

```
pthread_cond_t *mp;
```

```
cp = malloc(sizeof(pthread_cond_t));
```

Initialisation obligatoire

```
pthread_mutex_t cond = PTHREAD_COND_INITIALIZER;
```

ou

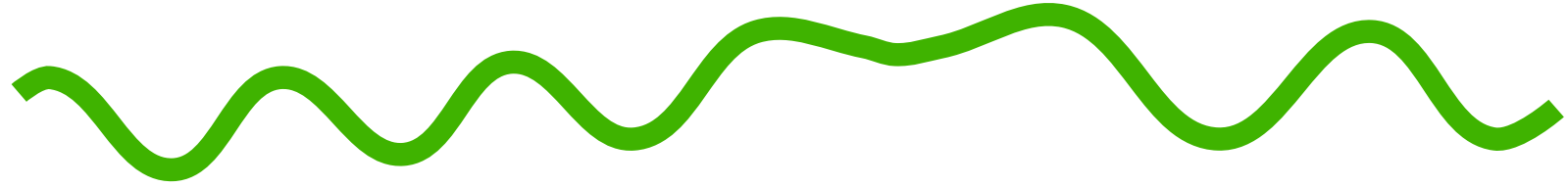
```
int pthread_cond_init(  
    pthread_cond_t *cp,  
    const pthread_condattr_t *attr);
```

valeur par défaut des attributs → NULL

Destruction

```
int pthread_cond_destroy(  
    pthread_cond_t *cp);
```

Attente et signal sur une condition

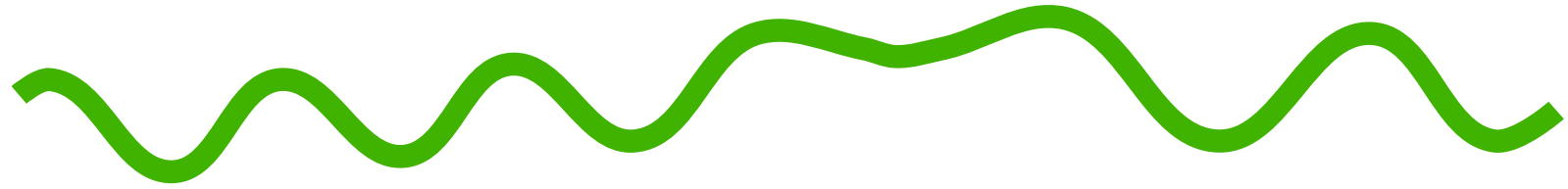


```
int pthread_cond_wait(  
    pthread_cond_t *cp,  
    pthread_mutex_t *mp);
```

~ Bloque le thread

- ~ jusqu'à ce que la condition soit signalée
- ~ libère le mutex associé avant de bloquer le thread
 - ~ seul le propriétaire du mutex peut appeler `pthread_cond_wait()`
- ~ verrouille le mutex avant de retourner suite à un signal sur la condition

Attente et signal sur une condition (cont'd)



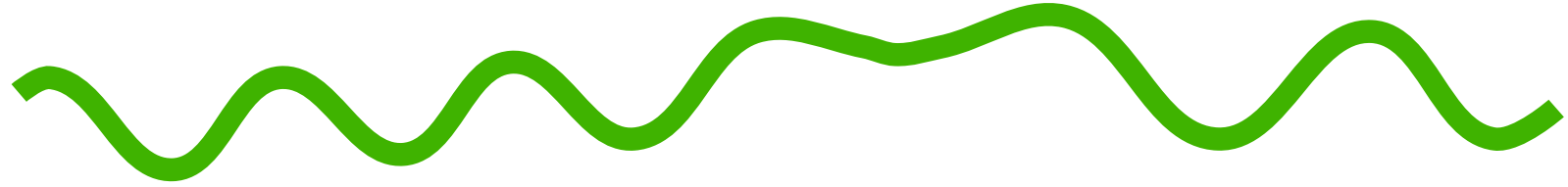
```
int pthread_cond_signal(  
    pthread_cond_t *cp);
```

- ✓ Réveille (au moins) un thread en attente sur la condition
- ✓ une implémentation donnée peut réveiller plus de un thread bloqué
- ✓ particulièrement sur multiprocesseurs

```
int pthread_cond_broadcast(  
    pthread_cond_t *cp);
```

- ✓ Réveille tous les threads en attente sur la condition

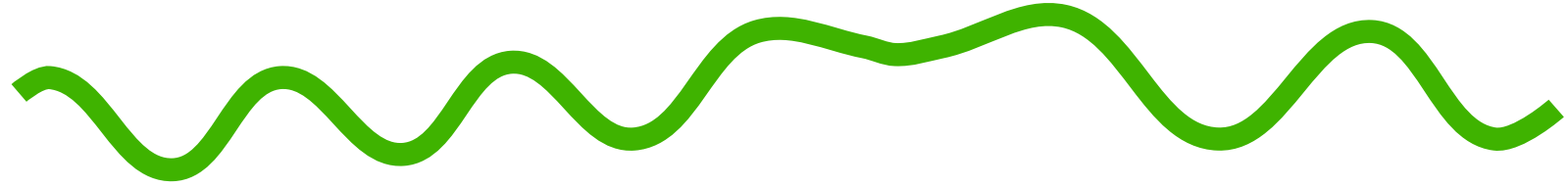
Signaler peut réveiller plusieurs threads



- Utilisation typique de l'attente
- Boucle sur la condition

```
pthread_mutex_lock(&mutex);  
/* la condition ne peut plus changer une fois le verrou acquis  
while (condition expression) { /* boucle pour re-tester la con  
    /* lache le mutex lors du wait */  
    pthread_cond_wait(&cond, &mutex);  
    /* ici on a repris le verrou */  
}  
... /* fait ce qui est necessaire, la condition etant vraie */  
pthread_mutex_unlock(&mutex);
```

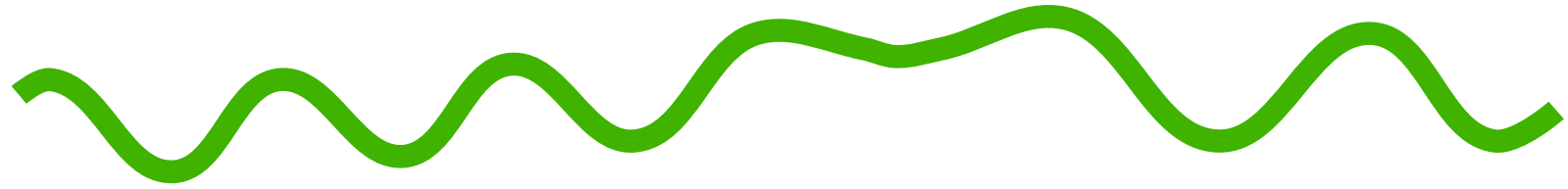
Exemple du producteur/consommateur



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

#define QSIZE 10                                /* number of pointers in the queue */
typedef struct {
    pthread_mutex_t buf_lock;                  /* lock the structure */
    int start_idx;                             /* start of valid data */
    int num_full;                              /* # of full locations */
    pthread_cond_t notfull;                   /* full -> notfull condition */
    pthread_cond_t notempty;                 /* empty -> notempty condition */
    void *data[QSIZE];                        /* circular buffer of pointers */
} circ_buf_t;
```

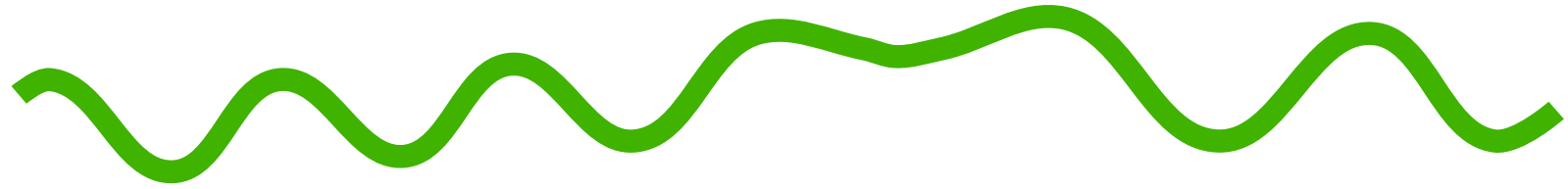
Exemple du producteur/consommateur (cont'd)



```
/* new_cb() creates and initializes a new circular buffer */
circ_buf_t *
new_cb()
{
    circ_buf_t *cbp;

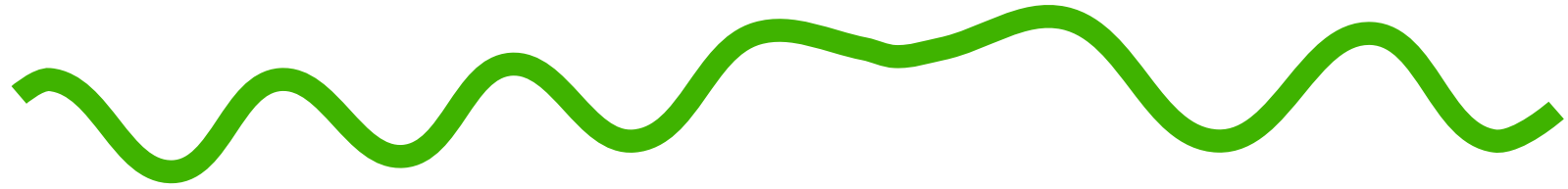
    cbp = malloc(sizeof(circ_buf_t));
    if (cbp == NULL)
        return NULL;
    pthread_mutex_init(&cbp->buf_lock, NULL);
    pthread_cond_init(&cbp->notfull, NULL);
    pthread_cond_init(&cbp->notempty, NULL);
    cbp->start_idx = 0;
    cbp->num_full = 0 ;
    return(cbp);
}
```

Exemple du producteur/consommateur (cont'd)



```
/* delete_cb() frees a circular buffer */  
void  
delete_cb(circ_buf_t *cbp)  
{  
    pthread_mutex_destroy(&cbp->buf_lock);  
    pthread_cond_destroy(&cbp->notfull);  
    pthread_cond_destroy(&cbp->notempty);  
    free(cbp);  
}
```


Exemple du producteur/consommateur (cont'd)



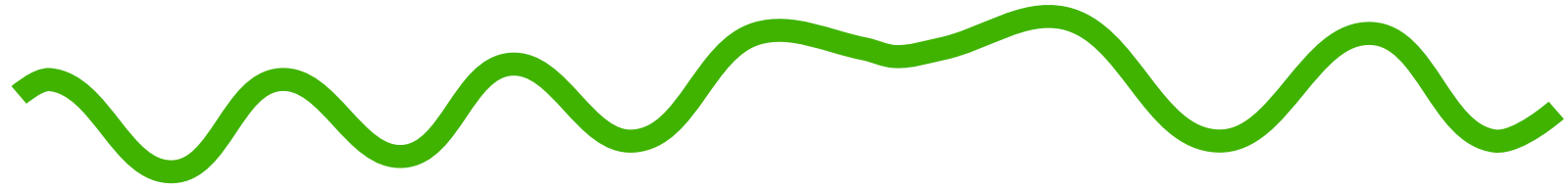
~ Producteur

```
si tampon_plein
    attendre(non_plein)
remplir_tampon()
signaler(non_vide)
```

~ Consommateur

```
si tampon_vide
    attendre(non_vide)
vider_tampon()
signaler(non_plein)
```

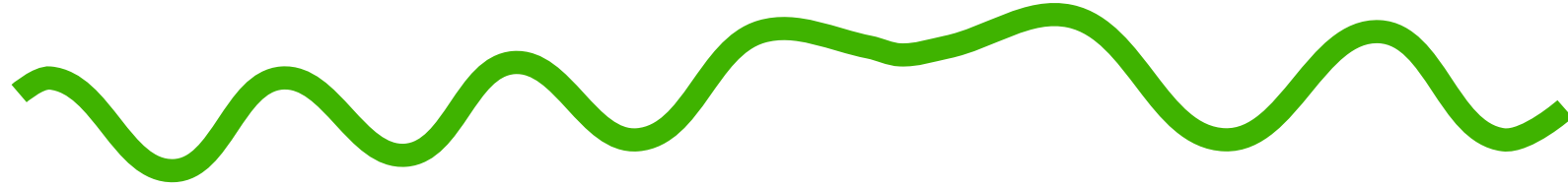
Exemple du producteur/consommateur (cont'd)



```
/* put_cb_data() puts new data on the queue.  
   If the queue is full, it waits until there is room. */  
void  
put_cb_data(circ_buf_t *cbp, void *data)  
{  
    pthread_mutex_lock(&cbp->buf_lock);  
    /* wait while the buffer is full */  
    while (cbp->num_full == QSIZE)  
        pthread_cond_wait(&cbp->notfull, &cbp->buf_lock);  
    cbp->data[(cbp->start_idx + cbp->num_full) % QSIZE] = data;  
    cbp->num_full += 1;  
    /* let a waiting reader know there's data */  
    pthread_cond_signal(&cbp->notempty);  
    pthread_mutex_unlock(&cbp->buf_lock);  
}
```

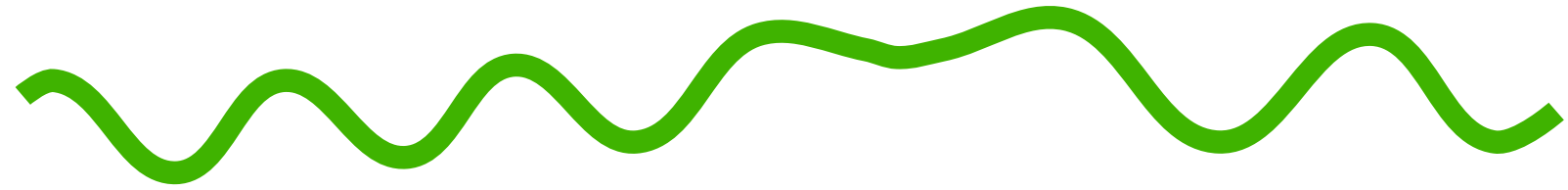
Exemple du

producteur/consommateur (cont'd)



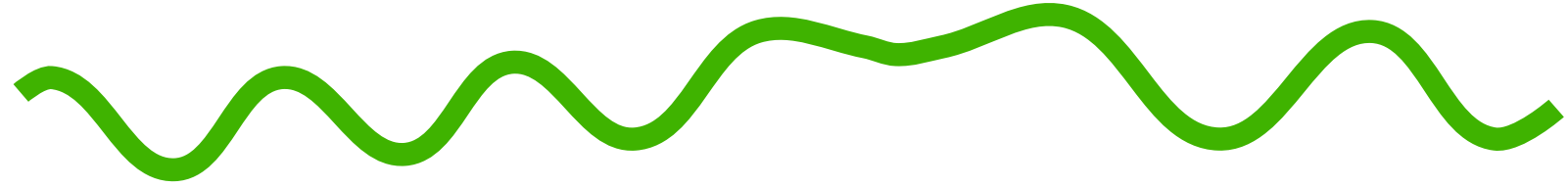
```
/* get_cb_data() get the oldest data in the circular buffer.
   If there is non, it waits until new data appears. */
void *
get_cb_data(circ_buf_t *cbp)
{
    void *data;

    pthread_mutex_lock(&cbp->buf_lock);
    /* wait while there's nothing in the buffer */
    while (cbp->num_full == 0)
        pthread_cond_wait(&cbp->notempty, &cbp->buf_lock);
    data = cbp->data[cbp->start_idx];
    cbp->start_idx = (cbp->start_idx + 1) % QSIZE;
    cbp->num_full -= 1;
    /* let a waiting writer know there's room */
    pthread_cond_signal(&cbp->notfull);
    pthread_mutex_unlock(&cbp->buf_lock);
    return data;
}
```



Programmer avec les threads

Granularité des données



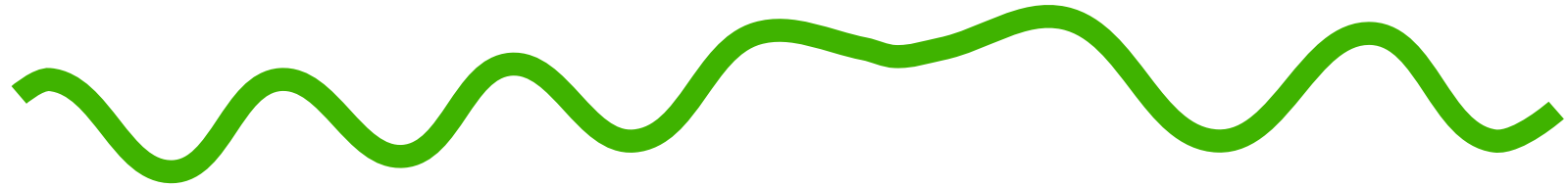
- La synchronisation sur des données suppose que les données sont indépendantes (un changement sur une donnée n'affecte pas d'autres données).
- Ce n'est pas toujours le cas !
- Les processeurs actuels ne savent pas adresser (load and store) moins qu'un mot (32 ou 64 bits par exemple).
- Ne pas utiliser deux parties d'un même mot :

```
char array [8] ;          /* Bad ! */
/* Thread 1 */           /* Thread 2 */
array[0]++ ;             array[1]++ ;
```

- Normalement les variables sont alignées sur des mots :

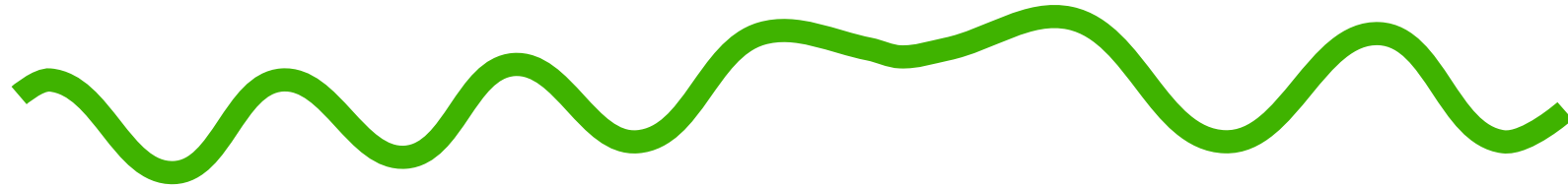
```
char a, b ;              /* Devrait etre OK */
/* Thread 1 */           /* Thread 2 */
a++ ;                    b++ ;
```

Verrou « récursif »



- ✓ N'a de récursif que la terminologie habituelle
- ✓ Une fonction verrouille un mutex
 - ✓ appelle une autre fonction qui verrouille aussi le mutex
 - ✓ situation « naturelle » : les deux fonctions accèdent une même variable...
- ✓ Mutex POSIX : deadlock !
 - ✓ il existe des mutex récursifs (Java...)
 - ✓ on peut les construire au dessus des verrous POSIX (cf TD...)
- ✓ Utilisation de mutex récursifs ?
 - ✓ emploi plus coûteux
 - ✓ bien souvent : pas d'intérêt
 - ✓ plutôt restructurer le code

Verrou « récursif » (cont'd)



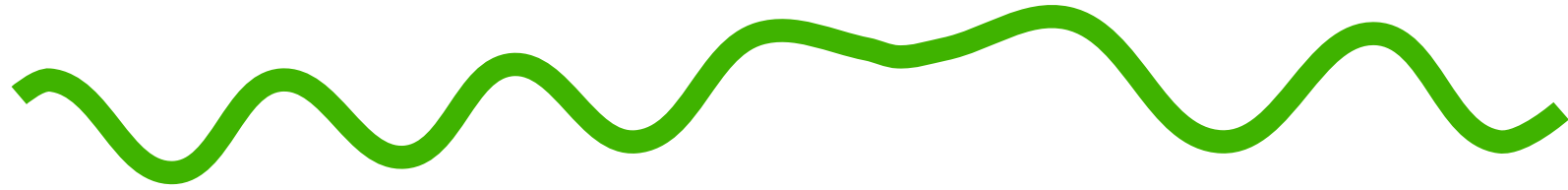
Exemple restructuration du code de la fonction `foo()`

```
data_t data ;          bar()          gee()
mutex_t m ;           {              {
                        lock(&m) ;      lock(&m) ;
                        ... data ...    gee'() ;
foo()                  unlock(&m) ;    unlock(&m) ;
{                      bar() ;        }
  lock(&m) ;           lock(&m) ;
  ... data ...       ... data ...
  foo() ;             unlock(&m) ;
  ... data ...
  unlock(&m) ;        }
}

                        }
                        ... data ...
                        gee'() ;
                        ... data ...
                        }
```

- bar() peut-être invalide : un autre thread prend la main avant l'appel récursif
- bar() coûteux (vs. coût d'un mutex récursif ?)
- gee()/gee'() la solution ?

Race conditions



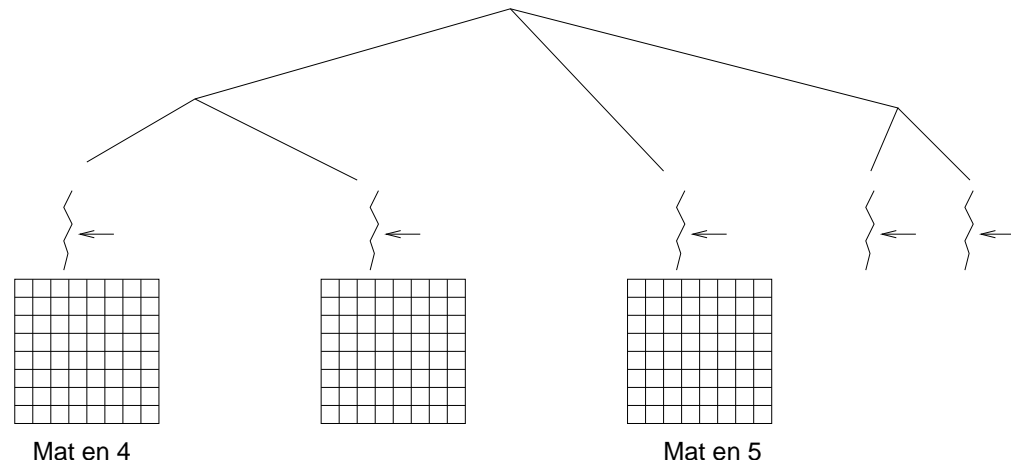
- Exécution concurrente en français ?
- Le résultat du programme dépend de l'ordonnancement des flots d'exécution

Peut être un bug de conception du programme !

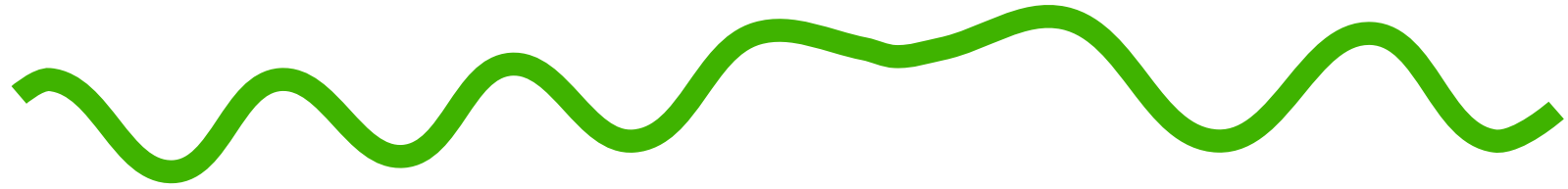
```
/* Thread 1 */  
mutex_lock(&m);  
v = v-1;  
mutex_unlock(&m);
```

```
/* Thread 2 */  
mutex_lock(&m);  
v = v*2;  
mutex_unlock(&m);
```

- Pas nécessairement un bug
 - plusieurs résultats corrects du programme



Race conditions (cont'd)



~ La plupart des « race conditions » sont à l'origine de deadlocks !

```
/* Thread 1 */  
mutex_lock(&m1);  
mutex_lock(&m2);
```

```
/* Thread 2 */
```

```
mutex_lock(&m2);  
mutex_lock(&m1);
```

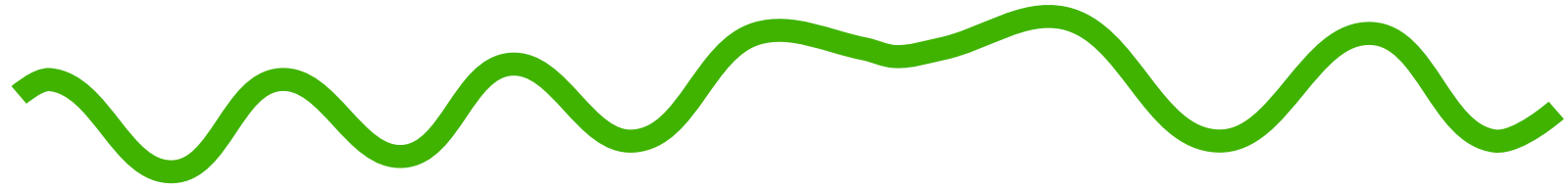
Le code va fonctionner 5 000 000 de fois ; puis...

```
/* Thread 1 */  
mutex_lock(&m1);  
  
mutex_lock(&m2);
```

```
/* Thread 2 */
```

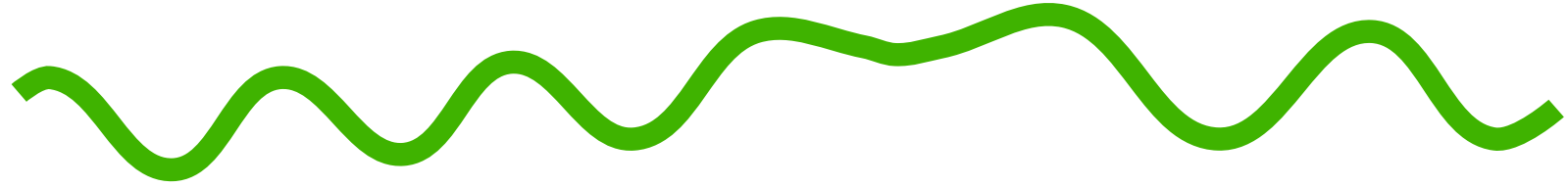
```
mutex_lock(&m2);  
  
mutex_lock(&m1);
```

Recouvrir un deadlock



- ✓ Le problème : un thread attend sur un verrou qu'un autre thread ne lachera jamais
 - ✓ (programme erroné)
 - ✓ erreur sur le thread...
- ✓ Recouvrir le deadlock : corriger, à l'exécution, cette situation
- ✓ Ne pas le faire ! Corriger le programme !
- ✓ Il ne suffit pas de lacher le verrou !

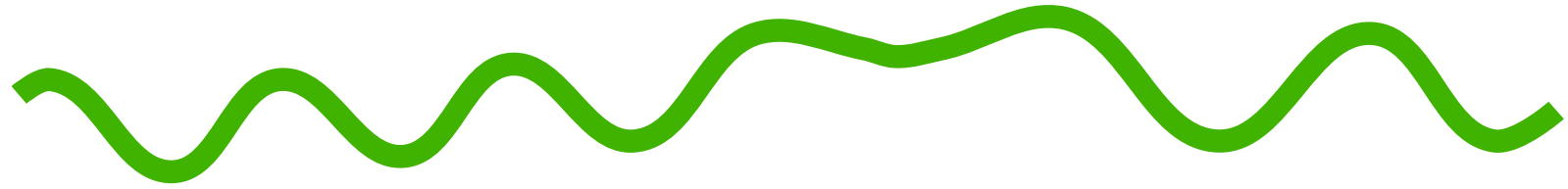
Fin d'un thread – thread détaché



```
void pthread_exit(  
    void *retval);
```

- ✓ Termine le thread courant
 - ✓ `retval` est la valeur retournée par le thread
 - ✓ elle pourra être consultée par un autre thread par un `pthread_join()`
 - ✓ suppose que le thread est en mode « *joinable* »
 - ✓ l'appel `pthread_join` n'est qu'une autre manière de synchroniser des threads
 - ✓ on confond l'attente sur la fin du thread avec l'attente sur le résultat de ce thread
- ✓ Explicitement attendre sur le résultat que doit produire ce thread

Fin d'un thread – thread détaché (cont'd)



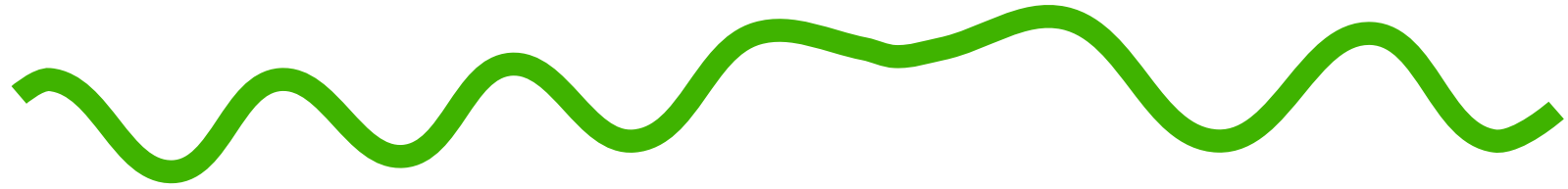
- ~ Lancer les threads en mode *détaché* par défaut :

```
pthread_attr_t attr;  
pthread_attr_init(&attr);  
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);  
  
pthread_create(&tid, &attr, func, arg);
```

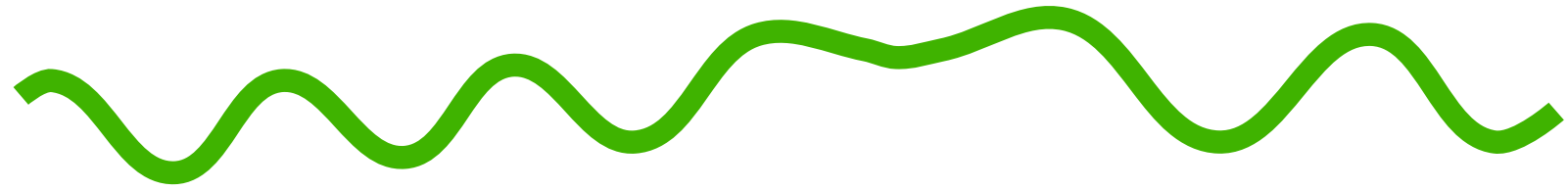
- ~ Passer un thread en mode détaché :

```
int pthread_detach(pthread_t t);
```

Debug d'un programme multithreadé

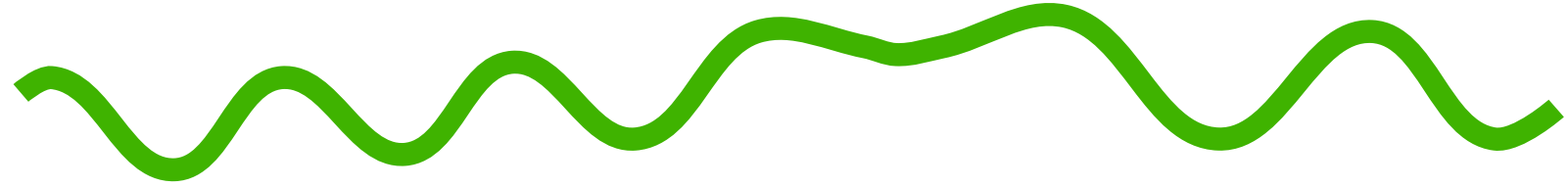


- ✓ Sources d'erreurs dans les programmes multithreadés :
 - ✓ passage en paramètre à un nouveau thread d'un pointeur dans la pile de l'appelant
 - ✓ accès à la mémoire globale sans protection (mutex...)
 - ✓ création de deadlock : deux threads tentent d'acquies deux ressources dans un ordre différent
 - ✓ tentative de ré-acquies un verrou déjà détenu
 - ✓ utilisation des signaux Unix et de threads
 - ✓ création de threads qui (par défaut!) ne sont pas détachés (doivent être réclamés par un `pthread_join()`)
 - ✓ prudence insuffisante lors de l'utilisation de cast `((void*))` pour les paramètres et retours des threads (`pthread_create()`)

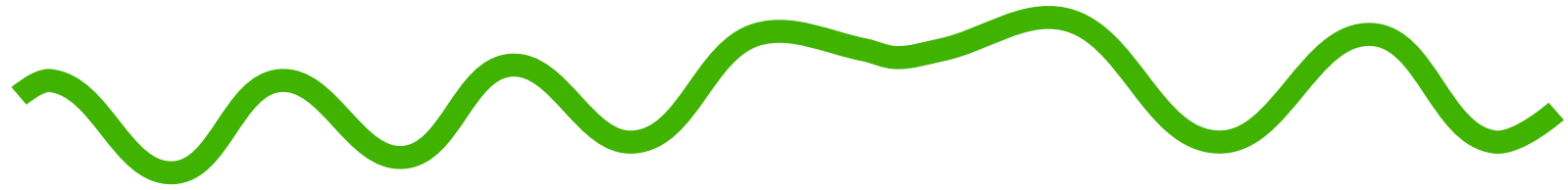


Implantations des threads

Deux niveaux de threads

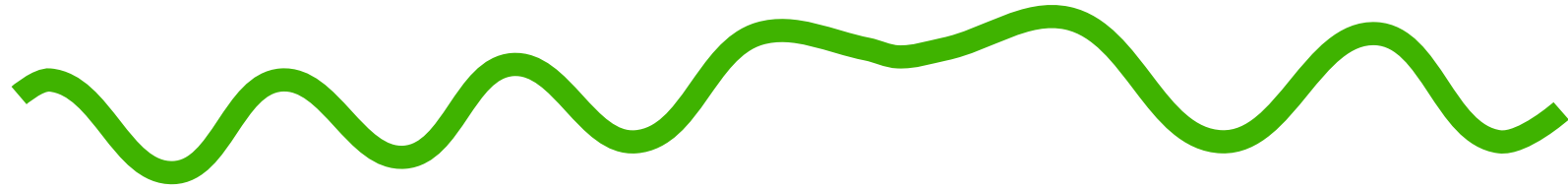


- ~ Deux grands types d'implantations de threads :
 1. threads utilisateurs — bibliothèque de threads
 2. threads noyaux — appels systèmes
- (Cf. cours de système ?)

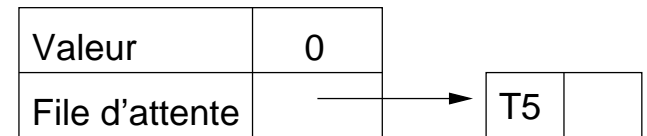
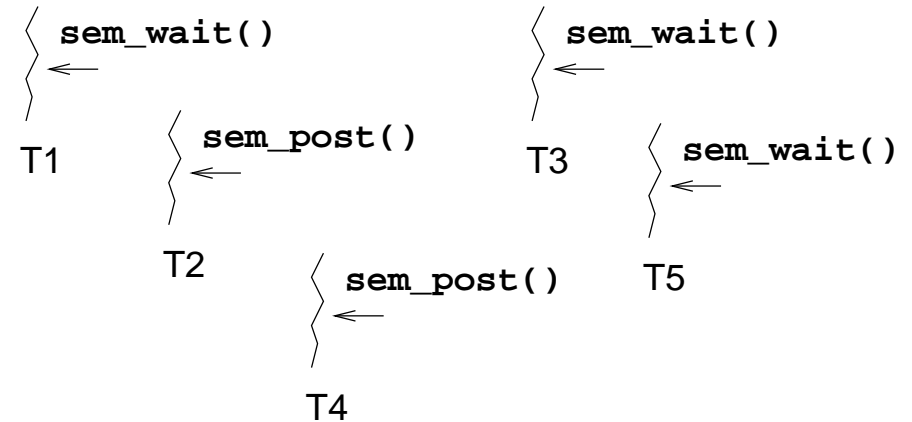


Compléments sur l'interface `pthread`

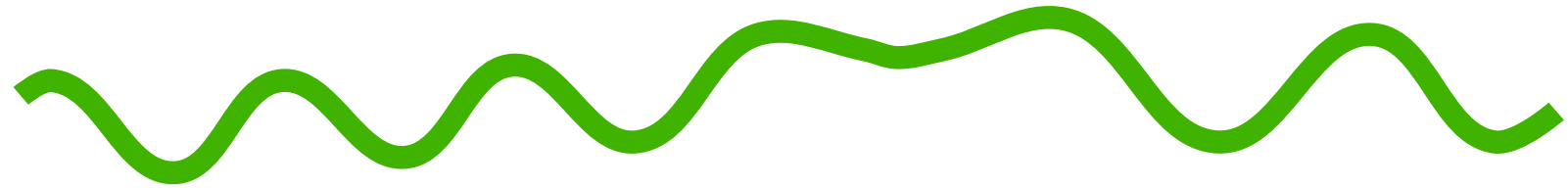
Sémaphores



- ✓ Synchronisation POSIX :
 - ✓ déjà vu : mutex, conditions, `exit()/join()`
 - ✓ existe aussi : sémaphores (à compteur)
- ✓ Sémaphore = une valeur et une file d'attente
 - ✓ `sem_wait()` bloquant si $\text{compteur} \leq 0$
 - ✓ `sem_post()` incrémente le compteur
 - ✓ pas de notion de propriétaire d'un sémaphore



Sémaphores (cont'd)



Interface POSIX :

```
#include <semaphore.h>
```

```
sem_t
```

```
sem_init();
```

```
sem_destroy();
```

```
sem_wait();
```

```
sem_post();
```

```
sem_trywait();
```

```
sem_getvalue();
```

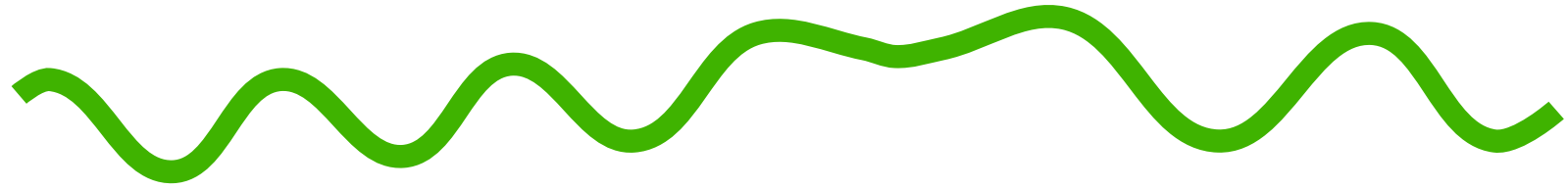
Création/destruction de sémaphores :

```
int sem_init(  
            sem_t *sem,  
            int pshared,  
            unsigned int value);
```

```
int sem_destroy(  
              sem_t * sem);
```

- ✓ initialisation obligatoire
- ✓ valeur initiale `value`
- ✓ `pshared` : sémaphore local au processus (valeur 0) ou partagé avec d'autres processus

Attente et notification sur un sémaphore



```
int sem_post(  
    sem_t *sem);
```

~ Incrmente le compteur de manière atomique

```
int sem_wait(  
    sem_t *sem);
```

~ Bloque tant que compteur ≤ 0 . Décrmente de manière atomique

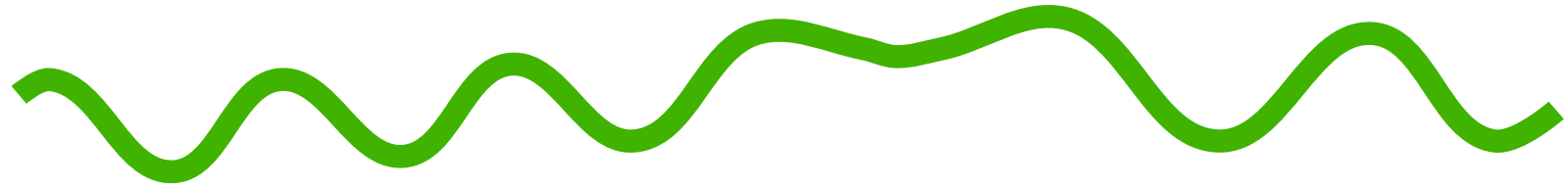
~ Interruption possible du thread dans un `sem_wait()`

~ (réception d'un signal, appel à `fork()`)

~ \Rightarrow retourne -1, positionne `errno` à `EINTR`

```
while (sem_wait(&s) == -1) { /* vraisemblablement rien */ }  
/* ici le semaphore est decremente */
```

Attente et notification sur un sémaphore (cont'd)



```
int sem_trywait(  
    sem_t *sem);
```

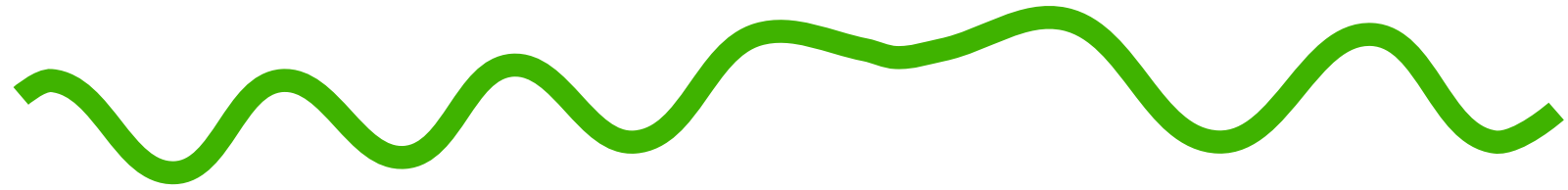
✓ Décrémenter un sémaphore

- ✓ variante non-bloquante de `sem_wait()`
- ✓ compteur > 0 : décrémente et retourne 0
- ✓ compteur à zéro : retourne immédiatement avec l'erreur `EAGAIN`

```
int sem_getvalue(  
    sem_t *sem,  
    int *sval);
```

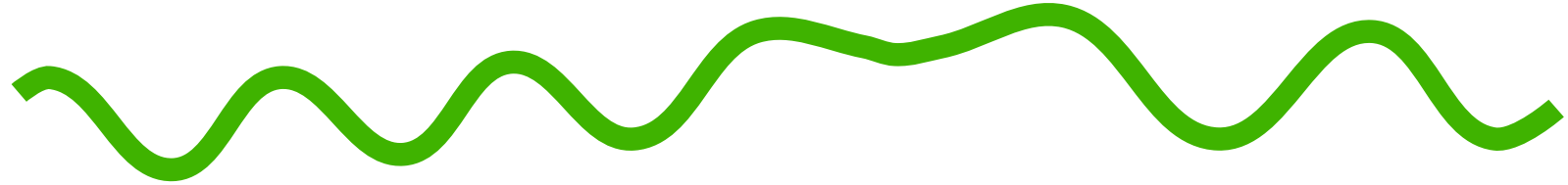
✓ Valeur d'un sémaphore

- ✓ retourne dans `sval` la valeur du compteur
- ✓ inutile : c'est toujours l'ancienne valeur !



Utilisation de threads et de bibliothèques

Bibliothèques « MT-safe »

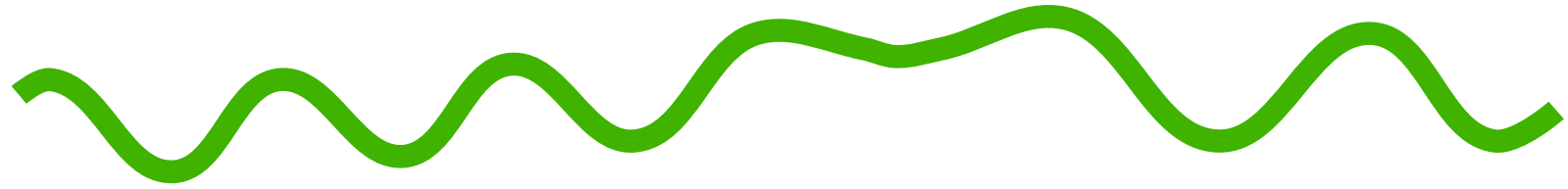


~ Triple exigence :

1. bibliothèque efficace
2. bibliothèque MT-safe
3. bibliothèque conforme à la sémantique et à l'API Unix

Deux parmi les trois : ok...

Bibliothèque non MT-safe ou changement d'API ?



~ Bibliothèque non MT-safe :

```
void
print_time(time_t time)
{
    char *s;
    s = ctime(time);
    printf("%s", s);
}
```

~ Version MT-safe de bibliothèque :

```
void
print_time(time_t time)
{
    char s[SIZE];
    ctime_r(time, s);
    printf("%s", s);
}
```

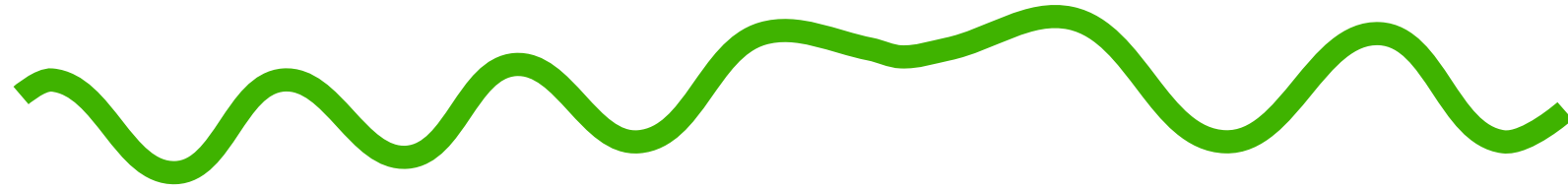
On trouve dans la libc :

```
char *
ctime(time_t time)
{
    static char s[SIZE];
    ...
    return(s);
}
```

Changement de l'API !

```
char *
ctime_r(time_t time, char *s)
{
    ...
    return(s);
}
```

Contrôle des accès à une bibliothèque non MT-safe



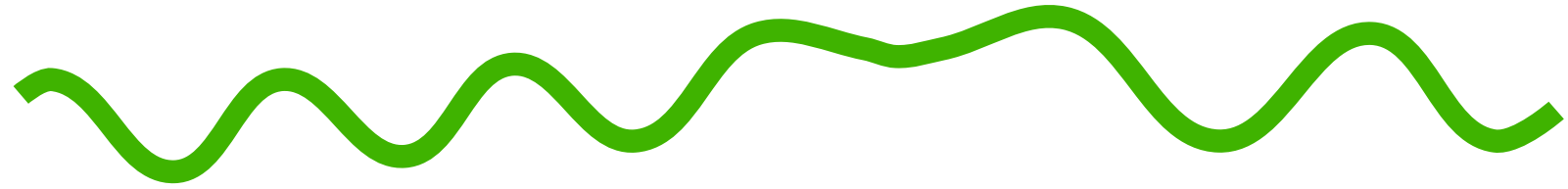
```
void
print_time(time_t time)
{
    char *s, str [SIZE];
    static pthread_mutex_t lock;

    pthread_mutex_lock(&m);
    s = ctime(time);
    strcpy(str, s) ;
    pthread_mutex_unlock(&m);

    printf("%s", str);
}
```

~ Quid des appels à la bibliothèque depuis une autre bibliothèque ?

Bibliothèque MT-safe avec une autre sémantique



Version MT-safe change la sémantique

- par exemple `pread()` et `pwrite()` comparées à `read()` et `write()`

- ```
ssize_t read(
 int fd, void *buf,
 size_t count);
```

- lit `count` octets dans le fichier associé au descripteur `fd` et les range dans `buf`

- lit ces octets à partir de la position courante dans le fichier

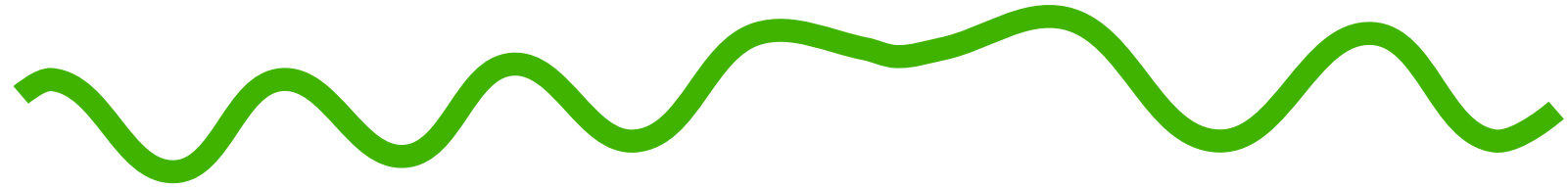
- effet de bord : incrémente la position courante de `count` : non thread-safe !

- ```
ssize_t pread(  
    int fd, void *buf,  
    size_t nbyte,  
    off_t offset);
```

- identique à `read()`

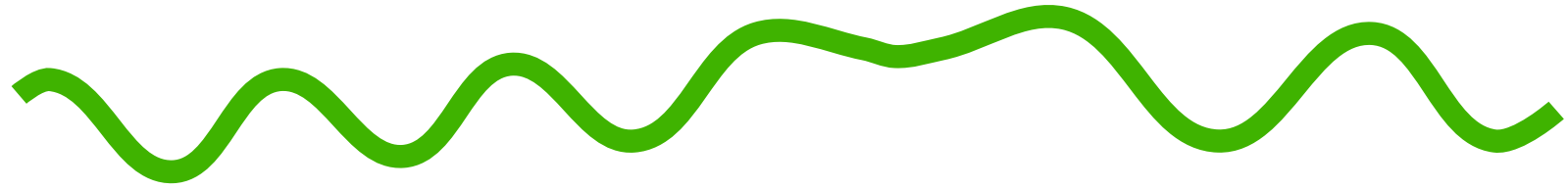
- mais : lit à partir de la position `offset` sans changer la position courante dans le fichier

Autres problèmes : `errno`, `getc()`

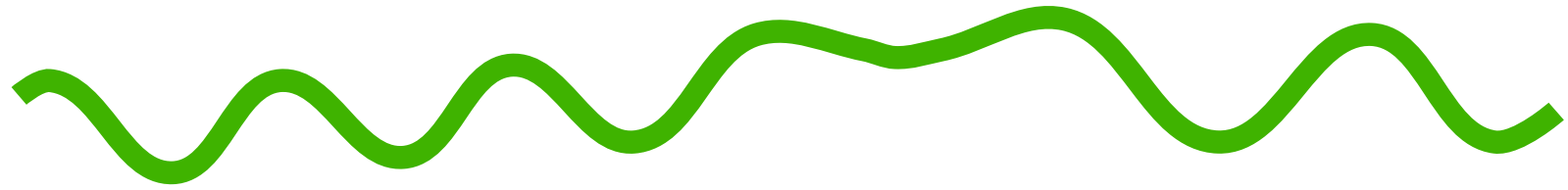


- ✓ La variable `errno`
 - ✓ positionnée par les appels systèmes (et quelques bibliothèques) pour préciser une erreur
 - ✓ cette valeur est significative quand un appel système a retourné une erreur (normalement -1)
- ✓ Ancienne implémentation : une variable globale
- ✓ On veut une valeur de erreur locale à chaque thread
 - ✓ c'est pour ce genre d'exigence qu'a été introduit le TSD
 - ✓ Thread Specific Data

Autres problèmes : *errno*, *getc()* (cont'd)

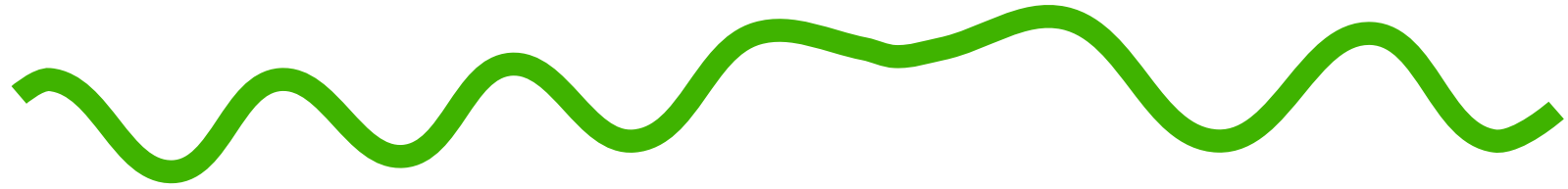


- ✓ `#include <stdio.h>`
- `int getc(FILE *fp);`
- ✓ macro définie dans `stdio.h`.
- ✓ retourne le caractère suivant du flux `fp`
- ✓ effet de bord : avance la position courante dans `fp`
- ✓ il existe des versions thread-safe de `getc()`
- ✓ pas de versions « `async-cancel-safe` »
- ✓ problème si le thread est interrompu pendant le `getc()`



Ce qui n'a pas été vu

Ce qui n'a pas été vu



- ~ Thread Specific Data
- ~ Cancellation
- ~ Threads et signaux Unix
- ~ Threads et... Java
- Threads et... Windows NT
- ~ Threads en environnement distribué
- ~ Threads communicants (« *talking threads* »)