# Compétition, indéterminisme et synchronisation : Sémaphores

Aloÿs Dufour

ATER, LIPN équipe LoCal
Université Paris-Nord XIII

24 mars 2026

# Motivation : nanosleep

- On utilise souvent `sleep` ou `usleep` pour faire dormir un *thread* et simuler un travail (en général pour une durée aléatoire).
- Problème, vu en TP :

  ```
  erreur: déclaration implicite de la fonction « usleep »;
  ```

  Alors que le fichier `unistd.h` était bien inclus.
- Raison : `usleep` était dépréciée (*deprecated*) et est maintenant supprimée (avec l'option `std=c99` ou supérieur).
- On va devoir utiliser `nanosleep`.

```c
int nanosleep(const struct timespec *durée, struct timespec *rem);
struct timespec {
    time_t tv_sec;  /* Seconds */
    long tv_nsec;   /* Nanoseconds [0, 999'999'999] */
};
```

# Structures : initialisations entre accolades

- Pour que ce ne soit pas trop lourd, on va utiliser des éléments du langage C99 pour initialiser ou créer à la volée des objets de types structurés.
- En C90, ces éléments de langages étaient inexistants ou très restreints.

On considère par la suite :

```
struct foo {
    char bar[256];
    int baz;
};
void print_foo(const struct foo *sf)
{
    printf(".bar = %s, .baz = %d\n", sf->bar, sf->baz);
}
```

# Initialisation totale avec ou sans désignation de champ

```c
// initialisation entre accolades à l'ancienne (C90)
struct foo good_old_foo = { "good_old_foo", 93 };
print_foo(&good_old_foo);

// initialisation avec désignations de champs (C99+)
struct foo plop = { .baz = 42, .bar = "lol" };
print_foo(&plop);
```

Donne bien ce qu'on attend :

```
.bar = good_old_foo, .baz = 93
.bar = lol, .baz = 42
```

# Initialisation entre accolades, encore

- Une initialisation entre accolades peut être partielle (au moins un champ initialisé) : dans ce cas les autres champs sont tous initialisés à 0 (des octets nuls), y compris les tableaux.
- Depuis C99, les champs initialisés peuvent contenir des éléments non connus à la compilation (donc vraiment variables).

```
int n;
printf("Que mettre dans hop.n ? ");
scanf("%d", &n);
struct foo hop = { .baz = n };
print_foo(&hop);
```

Affiche, par exemple (et il est garanti que le champ `bar` ne contient que des octets nuls) :

```
Que mettre dans hop.n ? 23
.bar = , .baz = 23
```

# Littéraux composés, exemples

- Parfois, on veut juste passer une structure à une fonction sans vouloir créer de nouvelle variable (comme quand on passe un `int` littéral, comme 42).
- Ceci est possible, avec des *littéraux composés* (*compound litterals*).

Par exemple,

```
print_foo(&(struct foo) { "hello", 42 });
print_foo(&(struct foo) { .baz = 93 });
print_foo(&(struct foo) { .bar = "lol" });
```

affiche bien

```
.bar = hello, .baz = 42
.bar = , .baz = 93
.bar = lol, .baz = 0
```

# Littéraux composés

- Syntaxe :
  `(struct <nom_struct>) <initialisation_accolades>`
- Ce n'est pas un *cast*, le type entre parenthèses indique au compilateur le nom du type composé.
- Tout ce qui a été dit précédemment sur l'initialisation entre accolades s'applique à la partie `<initialisation_accolades>`
- Équivalent à :

  ```
  struct <nom_struct> _anonyme_ = <initialisation_accolades>;
  ```

  puis remplacer le littéral composé par `_anonyme_`.

# Littéraux composés

- Syntaxe :
  `(struct <nom_struct>) <initialisation_accolades>`
- Ce n'est pas un *cast*, le type entre parenthèses indique au compilateur le nom du type composé.
- Tout ce qui a été dit précédemment sur l'initialisation entre accolades s'applique à la partie `<initialisation_accolades>`
- Équivalent à :

  `struct <nom_struct> _anonyme_ = <initialisation_accolades>;`

  puis remplacer le littéral composé par `_anonyme_`.

Par exemple,

```
print_foo(&(struct foo) { .baz = 93 });
```

équivaut à

```
struct foo _anonyme1_ = { .baz = 93 };
print_foo(&_anonyme1_);
```

# Autre exemple avec initialisations imbriquées

Avec les deux structures suivantes :

```
struct foo {
    char bar[256];
    int baz;
};
struct nestedfoo {
    struct foo f;
    int tab[3];
};
```

On peut considérer des initialisations imbriquées.

```
struct nestedfoo nf = { .f = { "lol", 12 }, .tab = { 1, 2, 3 } };
struct nestedfoo nf2 = { .f = { .baz = 1 }, .tab = { [1] = 42 } };
```

- Dans le deuxième cas, `nf2.f.bar` est composé entièrement d'octets nuls,
- et le tableau `nf2.f.tab` contient `{0, 42, 0}` (remarquer l'initialisation partielle de tableau avec indice).

# Retour à nanosleep

```
int nanosleep(const struct timespec *durée, struct timespec *rem);
struct timespec {
    time_t tv_sec;   /* Seconds */
    long tv_nsec;  /* Nanoseconds [0, 999'999'999] */
};
```

nanosleep() suspend l'exécution du thread appelant jusqu'à ce que le temps indiqué dans *durée ait expiré, ou que la réception d'un signal ait déclenché l'invocation d'un gestionnaire dans le thread appelant ou ait terminé le processus.

Si l'appel est interrompu par un gestionnaire de signal nanosleep() renvoie -1, renseigne errno avec la valeur EINTR, et inscrit le temps restant dans la structure pointée par rem à moins que rem soit NULL. La valeur de *rem peut être utilisée pour rappeler à nouveau nanosleep() afin de terminer la pause (mais voir la section NOTES plus loin).

# FAIRE_DES_CHOSES

- Ancienne version (dépréciée) :

```
void faire_des_choses(void) {
    usleep(rand() % 10 * 50000);
}
```

- Avec nanosleep et initialisation par accolades (champ tv_sec à 0) :

```
void faire_des_choses(void) {
    struct timespec ts = { .tv_nsec = rand() % 10 * 50000000 };
    nanosleep(&ts, NULL);
}
```

- Avec nanosleep et littéral composé

```
void faire_des_choses(void) {
    nanosleep(&(struct timespec){ .tv_nsec = rand() % 10
      * 50000000 }, NULL);
}
```

# Compétition : accès au terminal

Note : pour gagner de la place, à partir de maintenant on omet les inclusions de fichiers d'en-tête et certains tests de succès des appels systèmes (à faire tout de même !).

```
void *afficher_100_fois(void *arg);
int main(void)
{
    pthread_t th1, th2;
    pthread_create(&th1, NULL, afficher_100_fois, "A");
    pthread_create(&th2, NULL, afficher_100_fois, "B");
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    printf("\n");
}
void *afficher_100_fois(void *arg)
{
    const char *ch = arg;
    for (int i = 0; i < 100; ++i)
        printf("%s", ch);
    return NULL;
}
```

# Sortie ?

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAABAABABAAAABABBBBBBABBAAAAAABAAAAA
AAAAAAAAAAAAABABBBBBABAAAAAAAAAAAABABBBBBABAAAAAAAAAAAABABBBBBA
AAABAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBB
```

- le résultat est imprévisible ;
- et change si l'on lance de nouveau programme...
- On parle *d'indéterminisme* lorsque le résultat d'une fonction ou d'un programme ne dépend pas uniquement de son entrée (arguments, fichiers ouverts en lecture, etc).

# Situation de compétition

- On parle de situation de *compétition* (en anglais *race condition*)
- lorsque le résultat d'un programme
- ou d'un travail organisé (par exemple dans une usine, une cuisine, etc)
- diffère suivant l'ordre, arbitraire, dans lequel les acteurs (processus, *threads*, ouvriers, ...) accomplissent leurs tâches.

# Situation de compétition

- On parle de situation de *compétition* (en anglais *race condition*)
- lorsque le résultat d'un programme
- ou d'un travail organisé (par exemple dans une usine, une cuisine, etc)
- diffère suivant l'ordre, arbitraire, dans lequel les acteurs (processus, *threads*, ouvriers, ...) accomplissent leurs tâches.

Exemple : yaourt à la fraise

Alice tant qu'il y a des pots de yaourt sans fraise, mettre de la confiture de fraise dans un pot de yaourt

Bob tant qu'il y a des pots de yaourt sans yaourt, mettre du yaourt dans un pot de yaourt

# Situation de compétition

- On parle de situation de *compétition* (en anglais *race condition*)
- lorsque le résultat d'un programme
- ou d'un travail organisé (par exemple dans une usine, une cuisine, etc)
- diffère suivant l'ordre, arbitraire, dans lequel les acteurs (processus, *threads*, ouvriers, ...) accomplissent leurs tâches.

Exemple : yaourt à la fraise

Alice tant qu'il y a des pots de yaourt sans fraise, mettre de la confiture de fraise dans un pot de yaourt

Bob tant qu'il y a des pots de yaourt sans yaourt, mettre du yaourt dans un pot de yaourt

Rien n'impose que la confiture de fraise soit en-dessous.

# Compétition et processus

Des processus entrent en compétition lorsqu'ils font des entrées/sorties sur les mêmes fichiers, plus précisément, si

- plusieurs processus écrivent dans le même fichier ou bien
- au moins un processus écrit dans un fichier et d'autres le lisent.

# Compétition et processus

Des processus entrent en compétition lorsqu'ils font des entrées/sorties sur les mêmes fichiers, plus précisément, si

- plusieurs processus écrivent dans le même fichier ou bien
- au moins un processus écrit dans un fichier et d'autres le lisent.

Exemple de compétition en écriture dans le même fichier (trois shells écrivent dans le terminal) :

```
$ echo A & echo B &
[9] 32285
A
[10] 32286
B
$
```

# Compétition et processus

Des processus entrent en compétition lorsqu'ils font des entrées/sorties sur les mêmes fichiers, plus précisément, si

- plusieurs processus écrivent dans le même fichier ou bien
- au moins un processus écrit dans un fichier et d'autres le lisent.

Exemple de compétition en écriture dans le même fichier (trois shells écrivent dans le terminal) :

```
$ echo A & echo B &
[9] 32285
A
[10] 32286
B
$
```

mais aussi :

```
$ echo A & echo B &
[1] 32290
[2] 32291
A
$ B
```

(et tous les autres cas sont possibles en théorie).

# Compétition entre *threads*

Entre *threads* c'est encore pire car la mémoire est partagée ! En plus des entrées sorties, on a aussi compétition lorsque plusieurs threads accèdent à une même case de la mémoire, avec au moins l'un d'entre eux qui modifie son contenu.
Exemple :

- Alice fait des choses puis écrit (produit) dans une variable globale `resultat` un caractère ;
- Bob fait des choses puis lit (utilise, consomme) dans `resultat` le caractère et l'affiche.

Deux cas peuvent survenir :

- Alice produit avant que Bob consomme, ok.
- Bob consomme avant qu'Alice produise, ko.

# Alice et Bob

```c
char res;
int main(void)
{
    pthread_t a, b;
    srand(time(NULL));
    pthread_create(&a, NULL, alice, NULL);
    pthread_create(&b, NULL, bob, NULL);

    pthread_join(a, NULL);
    pthread_join(b, NULL);
    return 0;
}
void faire_des_choses(void) { /* ... */ }
void *alice(void *arg) { faire_des_choses(); res = 'A'; }
void *bob(void *arg) { faire_des_choses(); printf("%c\n", res); }
```

Sortie (possible)

# Alice et Bob

```c
char res;
int main(void)
{
    pthread_t a, b;
    srand(time(NULL));
    pthread_create(&a, NULL, alice, NULL);
    pthread_create(&b, NULL, bob, NULL);

    pthread_join(a, NULL);
    pthread_join(b, NULL);
    return 0;
}
void faire_des_choses(void) { /* ... */ }
void *alice(void *arg) { faire_des_choses(); res = 'A'; }
void *bob(void *arg) { faire_des_choses(); printf("%c\n", res); }
```

Sortie (possible)

```
$ gcc -Wall alice_bob1.c -pthread
$ ./a.out

$ ./a.out
A
```

# Bob a une idée

```c
int alice_a_fini = 0;
int main(void)
{
    /* comme avant */
}
void faire_des_choses() { /* ... */ }
void *alice(void *arg) {
    faire_des_choses();
    res = 'A';
    alice_a_fini = 1;
}
void *bob(void *arg) {
    faire_des_choses();
    while (!alice_a_fini)
        ;
    printf("%c\n", res);
}
```

# Problème résolu ?

Ce n'est pas idiot (et vraiment le début de quelque chose).

# Problème résolu ?

Ce n'est pas idiot (et vraiment le début de quelque chose). Mais regardons le code en assembleur produit :

```
.L6:
    movl    alice_a_fini(%rip), %eax
    testl    %eax, %eax
    je       .L6
```

# Problème résolu ?

Ce n'est pas idiot (et vraiment le début de quelque chose). Mais regardons le code en assembleur produit :

```
.L6:
    movl    alice_a_fini(%rip), %eax
    testl    %eax, %eax
    je      .L6
```

Pendant que Bob attend qu'Alice ait fini, le processeur :

- charge en mémoire le contenu de `alice_a_fini` ;
- compare son contenu à 0
- si oui saute deux lignes plus haut
- sinon continue l'exécution

# Problème résolu ?

Ce n'est pas idiot (et vraiment le début de quelque chose). Mais regardons le code en assembleur produit :

```
.L6:
    movl    alice_a_fini(%rip), %eax
    testl    %eax, %eax
    je       .L6
```

Pendant que Bob attend qu'Alice ait fini, le processeur :

- charge en mémoire le contenu de `alice_a_fini` ;
- compare son contenu à 0
- si oui saute deux lignes plus haut
- sinon continue l'exécution

Si Alice met un millième de seconde de plus que Bob à faire ses choses, un processeur qui a une fréquence d'horloge d'1 GHz fait un million d'opérations, 333333 fois ces trois opérations.

# Problème résolu ?

Ce n'est pas idiot (et vraiment le début de quelque chose). Mais regardons le code en assembleur produit :

```
.L6:
    movl    alice_a_fini(%rip), %eax
    testl    %eax, %eax
    je       .L6
```

Pendant que Bob attend qu'Alice ait fini, le processeur :

- charge en mémoire le contenu de `alice_a_fini` ;
- compare son contenu à 0
- si oui saute deux lignes plus haut
- sinon continue l'exécution

Si Alice met un millième de seconde de plus que Bob à faire ses choses, un processeur qui a une fréquence d'horloge d'1 GHz fait un million d'opérations, 333333 fois ces trois opérations.
Ça réchauffe la planète... et empêche le processeur de faire autre chose de plus utile. Ce procédé, appelé *attente active* est à éviter !

# Appels système bloquant

C'est là qu'intervient notre meilleur ami, le noyau.

# Appels système bloquant

C'est là qu'intervient notre meilleur ami, le noyau. Un appel système est dit bloquant lorsqu'il est associé à une condition et

- retourne immédiatement lorsque la condition est remplie ; ou bien
- *met en sommeil* la tâche qui fait cet appel si la condition n'est pas réalisée ...
- puis *réveille* cette tâche lorsque la condition est satisfaite.

# Appels système bloquant

C'est là qu'intervient notre meilleur ami, le noyau. Un appel système est dit bloquant lorsqu'il est associé à une condition et

- retourne immédiatement lorsque la condition est remplie ; ou bien
- *met en sommeil* la tâche qui fait cet appel si la condition n'est pas réalisée ...
- puis *réveille* cette tâche lorsque la condition est satisfaite.

Exemples (certains déjà rencontrés) :

- `sleep`, `usleep`, `nanosleep` : dormir en attendant qu'une certaine durée soit écoulée
- `wait` : si un enfant a déjà fini, retourne immédiatement, sinon attend
- `read` dans un `pipe` ou une `socket` : s'il y a des octets à lire, les lire et retourner le nombre d'octets lus, sinon attendre qu'il y en ait
- `pthread_join` : attendre la terminaison d'un thread

# JOIN

Ici, la solution est toute trouvée :

- le thread initial transmet à bob l'identifiant de *thread* d'`alice` ;
- bob attend qu'`alice` ait fini avec `pthread_join`.

```c
int main(void)
{
    pthread_t a, b;
    pthread_create(&a, NULL, alice, NULL);
    pthread_create(&b, NULL, bob, &a);

    pthread_join(b, NULL);
    return 0;
}
void *bob(void *thread)
{
    pthread_t *a = thread;
    faire_des_choses();
    pthread_join(*a, NULL);
    printf("%c\n", res);
    return NULL;
}
```

# Résultat

```
bob:
.LFB9:
    ; ...
    call    pthread_join
    ; ...
```

Version avec attente active de Bob : voir champ `user`.

```
$ time ./a.out
A

real  0m0,302s
user  0m0,051s
sys   0m0,001s
```

Même chose avec `join` (bloquant) :

```
$ time ./a.out
A

real  0m0,303s
user  0m0,000s
sys   0m0,003s
```

# Nouveau problème

Si `alice` a encore des choses à faire après avoir écrit ?

- Besoin d'autres appels bloquants.
- Dans ce cours : mutex et sémaphores.

# $1000000 = 1000 \times 1000$

```c
#define N 1000
void *bosser(void *);
int incremente_moi = 0;
int main(void)
{
    pthread_t th[N];
    int i;
    for (i = 0; i < N; ++i)
        pthread_create(&th[i], NULL, bosser, NULL);
    for (i = 0; i < N; ++i)
        pthread_join(th[i], NULL);

    printf("incremente_moi = %d\n", incremente_moi);
    return 0;
}
void *bosser(void *arg)
{
    int i;
    for (i = 0; i < N; ++i)
        ++incremente_moi;
    return NULL;
}
```

# Exécution

```
$ gcc -Wall million.c -pthread
$ ./a.out
incremente_moi = 997592
$ ./a.out
incremente_moi = 996979
$ ./a.out
incremente_moi = 997493
```

Angoissant...

# Incrémenter

Code assembleur (x86-64) :

```
movl    incremente_moi(%rip), %eax
addl    $1, %eax
movl    %eax, incremente_moi(%rip)
```

# Incrémenter

Code assembleur (x86-64) :

```
movl    incremente_moi(%rip), %eax
addl    $1, %eax
movl    %eax, incremente_moi(%rip)
```

Une ligne de code en C, mais 3 instructions :

- Copier la valeur de la globale `incremente_moi` dans le registre `%eax` ;
- ajouter 1 au contenu du registre `%eax` ;
- copier le contenu de `%eax` dans la globale `incremente_moi`.

On dit que l'incrémentation n'est pas une instruction atomique. En général l'atomicité dépend

- du compilateur (voire du niveau d'optimisation de celui-ci) ;
- de l'architecture du processeur.

# Problème de non-atomicité

Où est le problème avec ?

```
movl    incremente_moi(%rip), %eax
addl    $1, %eax
movl    %eax, incremente_moi(%rip)
```

Une possibilité, avec deux *threads* `t1` et `t2` sur un seul processeur qui incrémentent tous les deux `incremente_moi` :

1. `t1` charge `incremente_moi` qui vaut, disons, 100, dans le registre `eax`, puis incrémente le contenu d'`eax` qui contient 101.
2. PAF ! Commutation de contexte, le système donne la main à `t2`. Tous les registres associés à `t1` sont sauvegardés.
3. `t2` charge `incremente_moi` qui vaut toujours 100, dans le registre `eax`, puis incrémente le contenu d'`eax` qui contient 101 et met le contenu de `eax` (101) dans `incremente_moi`.
4. Le système redonne la main à `t1`, restore l'état dans lequel était le processeur au moment de la commutation de contexte : contenu de `eax` et compteur de programme.
5. `t1` reprend où il en était, il charge la valeur contenue dans `eax` (101) dans `incremente_moi`...
6. ... qui n'a donc été incrémenté qu'une seule fois au lieu de 2.

# Vu en TP, à ne pas faire

Parfois des *threads* ont besoin d'un indice pour fonctionner.

```
#define N 100
void *print_i(void *addr_i);
int main(void)
{
    pthread_t th[N];
    int i;
    for (i = 0; i < N; ++i)
        pthread_create(th + i, NULL, print_i, &i);
    for (i = 0; i < N; ++i)
        pthread_join(th[i], NULL);
    return 0;
}

void *print_i(void *addr_i)
{
    int i = *((int *) addr_i);
    printf("i = %d\n", i);
    return NULL;
```

# Compteur local à main et *threads*

```
$ gcc -Wall threads_compteur.c -pthread
$ ./a.out | sort -u | wc -l
95
$ ./a.out | sort -u | wc -l
94
$ ./a.out | sort -u | wc -l
96
```

La valeur de *i* peut changer entre l'appel à `pthread_create` et le déréférencement dans `print_i` !

# Une solution possible

```c
#define N 100
void *print_i(void *addr_i);
int main(void)
{
    pthread_t th[N];
    int cpt[N];
    int i;
    for (i = 0; i < N; ++i) {
        cpt[i] = i;
        pthread_create(th + i, NULL, print_i, cpt + i);
    }
    for (i = 0; i < N; ++i)
        pthread_join(th[i], NULL);
    return 0;
}
void *print_i(void *addr_i)
{
    int i = *((int *) addr_i);
    printf("i = %d\n", i);
    return NULL;
}
```

# Compteur et *threads*, solution

- Le contenu de `cpt[i]` n'est pas modifié après son initialisation.
- ```
  $ gcc -Wall threads_compteur2.c -pthread
  $ ./a.out | sort -u | wc -l
  100
  $ ./a.out | sort -u | wc -l
  100
  $ ./a.out | sort -u | wc -l
  100
  ```
- technique courante en calcul distribué : la promotion d'un scalaire (une variable numérique) en un vecteur (un tableau).
  ```

# Un autre problème

Considérer aussi :

- Dans un premier thread :

```
n = 3;
/* ... */
n = 0;
```

- Dans un second thread :

```
if (n != 0) {
    x = 8 / n;
    /* ... */
}
```

- Possible que :
  1. le premier thread met 3 dans `n` ;
  2. le second fait le test `n != 0` puis saute à l'instruction suivante ;
  3. le premier thread met 0 dans `n` ;
  4. le second thread calcule `8 / 0` : arrêt du programme avec `Floating Point Exception`

# Sémaphores

- Du grec *sema* signe et *phoros*, porter.
- Inventés par Dijkstra.
- Objet opaque contenant un compteur $\geq 0$.
- Un compteur à $0$ peut s'interpréter comme une absence de ressource.
- Un compteur à $n > 0$ peut s'interpréter comme la présence de $n > 0$ ressources qui vont pouvoir être consommées.
- Ce compteur n'est jamais lu ni modifié directement.

# Sémaphores

Seulement $4$ opérations (dans ce cours et dans presque tous les cours sur les sémaphores).

```
#include <semaphore.h>
sem_init(sem_t *s, 0, val);
sem_wait(sem_t *s);
sem_post(sem_t *s);
sem_destroy(sem_t *s);
```

# Initialisation et destruction

```
int sem_init(sem_t *s, 0, unsigned val);
```

- Initialise le sémaphore d'adresse `s`.
- Comme d'habitude, valeur de retour $0$ si succès $\neq 0$ si échec.
- Deuxième argument : pour utiliser le sémaphore avec des processus, on oublie, toujours $0$.
- Troisième argument : valeur initiale du sémaphore :
  - $n > 0$ si on considère qu'il doit être initialement « ouvert », ou encore qu'on commence avec $n > 0$ ressource disponible ;
  - $0$ sinon (fermé, bloqué, pas de ressource, ...)
- Le choix de cette valeur initiale est critique.

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

pour détruire le sémaphore lorsqu'on n'en a plus besoin (libération de ressources).

# Attente d'une condition/Consommation d'une ressource

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

Un *thread* dont l'exécution arrive à cette instruction :

- est *endormi* si le compteur associé au séminaire d'adresse `s` est $0$
- ou bien décrémente ce compteur puis passe à la prochaine instruction s'il est $> 0$.
- Opération atomique.

# Signaler une production ou la réalisation d'une condition

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

- Incrémente le compteur associé au sémaphore d'adresse `s` ;
- Si des *threads* sont endormis en attente sur le sémaphore d'adresse `s`, réveille un et un seul d'entre eux (qui décrémentera alors immédiatement le compteur).
- Opération atomique.

# Pour les heureux (?) possesseurs de Mac

Les sémaphores anonymes (ceux décrits plus haut) ne sont pas implémentés dans MacOS (en fait l'appel `sem_init` se contente de toujours retourner une erreur...)
Heureusement, il y a tout de même les sémaphores nommés qui s'utilisent presque de la même manière :

```
sem_t *mon_sem;
if ((mon_sem = sem_open("/mon_sem", O_CREAT | O_EXCL, 0600, 1)
        == SEM_FAILED) {
    perror("creation mon_sem");
    exit(1);
}

/* utiliser le sémaphore comme d'habitude */
sem_close(mon_sem); /* fermer le  fichier */
sem_unlink("/mon_sem"); /* supprimer le fichier */
```

# Alice et Bob au pays des sémaphores

```c
void *alice(void *);
void *bob(void *);
void faire_des_choses();
char res;
int main(void)
{
    pthread_t a, b;
    sem_t s;
    sem_init(&s, 0, 0); /* initialement, 0 ressource */
    srand(time(NULL));
    pthread_create(&a, NULL, alice, &s);
    pthread_create(&b, NULL, bob, &s);

    pthread_join(b, NULL);
    pthread_join(a, NULL);

    sem_destroy(&s);
    return 0;
}
void faire_des_choses() { /* ... */ }
```

# Alice et Bob au pays des sémaphores (2)

```c
void *alice(void *sem)
{
    sem_t *s = sem;
    faire_des_choses();
    res = 'A';
    sem_post(s); /* dire à Bob qu'il peut continuer */
    faire_des_choses();
    printf("Alice : j'ai fait pas mal de choses !\n");
    return NULL;
}
void *bob(void *sem)
{
    sem_t *s = sem;
    faire_des_choses();
    /* bloque si besoin jusqu'à ce qu'Alice ait fini */
    sem_wait(s);
    printf("Bob : Alice a fini et a écrit : %c\n", res);
    return NULL;
}
```

# Plus compliqué

- Alice fait un certain nombre de calculs (elle produit des valeurs).
- Lorsqu'un tel résultat est prêt, Bob l'utilise (ici, l'affiche).
- Lorsque tous les résultats ont été produit, Alice doit communiquer à Bob qu'il n'y a plus rien à produire.
- Bob doit attendre qu'Alice produise `res` avant de l'utiliser.
- Alice doit attendre avant de mettre à jour `res` que Bob l'ait utilisé.

# Deux sémaphores et une globale

- On va utiliser un tableau `s` de deux sémaphores :
  - le premier élément (`s[0]`) est utilisé par Alice pour dire qu'elle a fini de produire un résultat
  - le second (`s[1]`) par Bob pour dire qu'il a fini d'utiliser le résultat précédent
- une variable globale `fini` est mise à 1 lorsqu'Alice a fini toutes ses productions.

# Nouvelle fonction main

```c
char res = 'A' - 1;
int fini = 0;
int main(void)
{
    pthread_t a, b;
    sem_t s[2];
    /* initialement alice n'a pas encore produit */
    sem_init(s, 0, 0);
    /* bob est prêt à afficher une nouvelle valeur */
    sem_init(s + 1, 0, 1);
    srand(time(NULL));
    pthread_create(&a, NULL, alice, s);
    pthread_create(&b, NULL, bob, s);

    pthread_join(b, NULL);
    pthread_join(a, NULL);

    sem_destroy(s);
    sem_destroy(s + 1);
    return 0;
}
```

# Nouveau code d'Alice

```
void *alice(void *sem)
{
    sem_t *s = sem;
    while (res < 'Z') {
        faire_des_choses();
        sem_wait(s + 1);
        ++res;
        if (res == 'Z')
            fini = 1;
        sem_post(s);
    }
    printf("Alice: j'ai fait pas mal de choses !\n");
    return NULL;
}
```

# Nouveau code de Bob

```c
void *bob(void *sem)
{
    sem_t *s = sem;
    for (;;) {
        faire_des_choses();
        sem_wait(s);
        printf("Bob: Alice a fini et a écrit: %c\n", res);
        if (fini)
            break;
        sem_post(s + 1);
    }
    return NULL;
}
```

# TD, Exo 1 — Exemples de sémaphores I

Dans toutes les questions qui suivent, on suppose donnée la fonction `main`, les globales suivantes, et les déclarations suivantes :

```
void *hop(void *);
void *plop(void *);
void travailler(void);
sem_t sem1;
sem_t sem2;
int res;
int main(void)
{
   pthread_t th_hop, th_plop;
   sem_init(&sem1, 0, X);
   sem_init(&sem2, 0, Y);
   pthread_create(&th_hop, NULL, hop, NULL);
   pthread_create(&th_plop, NULL, plop, NULL);
   pthread_join(th_hop, NULL);
   pthread_join(th_plop, NULL);
   sem_destroy(&sem1);
   sem_destroy(&sem2);
   return 0;
}
```

Les valeurs initiales `X` et `Y` des sémaphores seront données pour chaque question. La fonction `travailler` représente un travail, arbitraire, qui ne modifie ni les sémaphores, ni la variable `res`. Donner le comportement du programme ou, si une situation de compétition ou une étreinte mortelle (*deadlock*, le programme est bloqué) sont possibles.

# TD, Exo 1 — Exemples de sémaphores II

1. `X = 0`, `Y = 1` et les fonctions `hop` et `plop` sont les suivantes :

```
void *hop(void *arg) {
   travailler();
   sem_wait(&sem2);
   res = 42;
   sem_post(&sem1);
   travailler();
   sem_wait(&sem2);
   res = 93;
   sem_post(&sem1);
   return NULL;
}
void *plop(void *arg) {
   travailler();
   sem_wait(&sem1);
   printf("%d\n", res);
   sem_post(&sem2);
   travailler();
   sem_wait(&sem1);
   printf("%d\n", res);
   sem_post(&sem2);
   return NULL;
}
```

2. Mêmes fonctions avec `X = 1` et `Y = 1`.
3. Mêmes fonctions avec `X = 0` et `Y = 0`.
4. Mêmes fonctions avec `X = 0` et `Y = 2`.
5. Les fonctions `hop` et `plop` sont les suivantes, `X = 1` et `sem2` n'est pas utilisé. La fonction `main` affiche le contenu de `res` après avoir `join` les deux *threads*. Attention, l'incrémentation n'est pas supposée atomique.

```
void *hop(void *arg) {
   for (int i = 0; i < 3; i++) {
      sem_wait(&sem1);
      res++;
      sem_post(&sem1);
   }
   return NULL;
}
void *plop(void *arg) {
   return hop(NULL);
}
```

6. Même chose pour `X = 2`.
7. Même chose pour `X = 0`.

# TD, Exo 1 — Corr

1. Après avoir travaillé, `plop` est bloqué sur l'attente de `sem1`, tandis que `hop` n'attend pas sur `sem2` (car il est initialisé à 1), qui passe à 0, puis met 42 dans `res`, incrémente `sem1`, ce qui réveille `plop`. Ensuite `plop` affiche le résultat (42) et incrémente `sem2`, ce qui permet à `hop` de mettre 93 dans `res` puis d'incrémenter `sem1`. `plop` peut maintenant décrémenter `sem1` (qui repasse à 0) afficher le résultat (93) et incrémenter `sem2` qui repasse à 1.

# TD, Exo 1 — Corr

1. Après avoir travaillé, `plop` est bloqué sur l'attente de `sem1`, tandis que `hop` n'attend pas sur `sem2` (car il est initialisé à 1), qui passe à 0, puis met 42 dans `res`, incrémente `sem1`, ce qui réveille `plop`. Ensuite `plop` affiche le résultat (42) et incrémente `sem2`, ce qui permet à `hop` de mettre 93 dans `res` puis d'incrémenter `sem1`. `plop` peut maintenant décrémenter `sem1` (qui repasse à 0) afficher le résultat (93) et incrémenter `sem2` qui repasse à 1.
2. Avec `X = 1` et `Y = 1`, il y a compétition car `plop` n'attend pas que `hop` produise, il est donc possible qu'il affiche 0 dans certains cas, ou bien 42 si `hop` a fini de produire avant la fin du travail de `plop`.

# TD, Exo 1 — Corr

1. Après avoir travaillé, `plop` est bloqué sur l'attente de `sem1`, tandis que `hop` n'attend pas sur `sem2` (car il est initialisé à 1), qui passe à 0, puis met 42 dans `res`, incrémente `sem1`, ce qui réveille `plop`. Ensuite `plop` affiche le résultat (42) et incrémente `sem2`, ce qui permet à `hop` de mettre 93 dans `res` puis d'incrémenter `sem1`. `plop` peut maintenant décrémenter `sem1` (qui repasse à 0) afficher le résultat (93) et incrémenter `sem2` qui repasse à 1.
2. Avec `X = 1` et `Y = 1`, il y a compétition car `plop` n'attend pas que `hop` produise, il est donc possible qu'il affiche 0 dans certains cas, ou bien 42 si `hop` a fini de produire avant la fin du travail de `plop`.
3. Situation de *deadlock* (étreinte mortelle) dès le départ : chaque *thread* est en attente sur son sémaphore, et `main` attend avec `join` la fin des *thread*, qui n'arrive jamais.

# TD, Exo 1 — Corr

1. Après avoir travaillé, `plop` est bloqué sur l'attente de `sem1`, tandis que `hop` n'attend pas sur `sem2` (car il est initialisé à 1), qui passe à 0, puis met 42 dans `res`, incrémente `sem1`, ce qui réveille `plop`. Ensuite `plop` affiche le résultat (42) et incrémente `sem2`, ce qui permet à `hop` de mettre 93 dans `res` puis d'incrémenter `sem1`. `plop` peut maintenant décrémenter `sem1` (qui repasse à 0) afficher le résultat (93) et incrémenter `sem2` qui repasse à 1.
2. Avec `X = 1` et `Y = 1`, il y a compétition car `plop` n'attend pas que `hop` produise, il est donc possible qu'il affiche 0 dans certains cas, ou bien 42 si `hop` a fini de produire avant la fin du travail de `plop`.
3. Situation de *deadlock* (étreinte mortelle) dès le départ : chaque *thread* est en attente sur son sémaphore, et `main` attend avec `join` la fin des *thread*, qui n'arrive jamais.
4. Encore compétition, mais plus subtile. Comme `Y` vaut 2 initialement, il est possible que `hop` enchaîne les deux productions (42 puis 93) avant que `plop` n'affiche le résultat, et donc que `plop` affiche deux fois 93. Mais il est aussi possible que le programme se déroule « normalement » (42 puis 93).

# TD, Exo 1 — Corr

1. Après avoir travaillé, `plop` est bloqué sur l'attente de `sem1`, tandis que `hop` n'attend pas sur `sem2` (car il est initialisé à 1), qui passe à 0, puis met 42 dans `res`, incrémente `sem1`, ce qui réveille `plop`. Ensuite `plop` affiche le résultat (42) et incrémente `sem2`, ce qui permet à `hop` de mettre 93 dans `res` puis d'incrémenter `sem1`. `plop` peut maintenant décrémenter `sem1` (qui repasse à 0) afficher le résultat (93) et incrémenter `sem2` qui repasse à 1.
2. Avec `X = 1` et `Y = 1`, il y a compétition car `plop` n'attend pas que `hop` produise, il est donc possible qu'il affiche 0 dans certains cas, ou bien 42 si `hop` a fini de produire avant la fin du travail de `plop`.
3. Situation de *deadlock* (étreinte mortelle) dès le départ : chaque *thread* est en attente sur son sémaphore, et `main` attend avec `join` la fin des *thread*, qui n'arrive jamais.
4. Encore compétition, mais plus subtile. Comme `Y` vaut 2 initialement, il est possible que `hop` enchaîne les deux productions (42 puis 93) avant que `plop` n'affiche le résultat, et donc que `plop` affiche deux fois 93. Mais il est aussi possible que le programme se déroule « normalement » (42 puis 93).
5. Les deux *threads* font la même chose, l'un des deux décrémente `sem1` puis incrémente `res`, et incrémente `sem1`, ce qui réveille l'autre, etc. Cela permet de s'assurer qu'aucun des deux *thread* n'interrompt l'autre pendant l'incrémentation, donc le résultat est bien 6.

# TD, Exo 1 — Corr

1. Après avoir travaillé, `plop` est bloqué sur l'attente de `sem1`, tandis que `hop` n'attend pas sur `sem2` (car il est initialisé à 1), qui passe à 0, puis met 42 dans `res`, incrémente `sem1`, ce qui réveille `plop`. Ensuite `plop` affiche le résultat (42) et incrémente `sem2`, ce qui permet à `hop` de mettre 93 dans `res` puis d'incrémenter `sem1`. `plop` peut maintenant décrémenter `sem1` (qui repasse à 0) afficher le résultat (93) et incrémenter `sem2` qui repasse à 1.
2. Avec `X = 1` et `Y = 1`, il y a compétition car `plop` n'attend pas que `hop` produise, il est donc possible qu'il affiche 0 dans certains cas, ou bien 42 si `hop` a fini de produire avant la fin du travail de `plop`.
3. Situation de *deadlock* (étreinte mortelle) dès le départ : chaque *thread* est en attente sur son sémaphore, et `main` attend avec `join` la fin des *thread*, qui n'arrive jamais.
4. Encore compétition, mais plus subtile. Comme `Y` vaut 2 initialement, il est possible que `hop` enchaîne les deux productions (42 puis 93) avant que `plop` n'affiche le résultat, et donc que `plop` affiche deux fois 93. Mais il est aussi possible que le programme se déroule « normalement » (42 puis 93).
5. Les deux *threads* font la même chose, l'un des deux décrémente `sem1` puis incrémente `res`, et incrémente `sem1`, ce qui réveille l'autre, etc. Cela permet de s'assurer qu'aucun des deux *thread* n'interrompt l'autre pendant l'incrémentation, donc le résultat est bien 6.
6. Pour `X = 2`, les deux *threads* peuvent incrémenter « en même temps » la variable `res`. Comme l'incrémentation n'est pas atomique, il peut se produire la chose suivante (entre autres possibilités) : `hop` charge dans un registre la valeur de `res` (0) pendant que `plop` charge la même valeur (0), les deux *threads* écrivent 1 dans `res` qui n'est incrémentée qu'une fois. En lançant 10000 fois le programme, c'est arrivé quelques fois.

# TD, Exo 1 — Corr

1. Après avoir travaillé, `plop` est bloqué sur l'attente de `sem1`, tandis que `hop` n'attend pas sur `sem2` (car il est initialisé à 1), qui passe à 0, puis met 42 dans `res`, incrémente `sem1`, ce qui réveille `plop`. Ensuite `plop` affiche le résultat (42) et incrémente `sem2`, ce qui permet à `hop` de mettre 93 dans `res` puis d'incrémenter `sem1`. `plop` peut maintenant décrémenter `sem1` (qui repasse à 0) afficher le résultat (93) et incrémenter `sem2` qui repasse à 1.
2. Avec `X = 1` et `Y = 1`, il y a compétition car `plop` n'attend pas que `hop` produise, il est donc possible qu'il affiche 0 dans certains cas, ou bien 42 si `hop` a fini de produire avant la fin du travail de `plop`.
3. Situation de *deadlock* (étreinte mortelle) dès le départ : chaque *thread* est en attente sur son sémaphore, et `main` attend avec `join` la fin des *thread*, qui n'arrive jamais.
4. Encore compétition, mais plus subtile. Comme `Y` vaut 2 initialement, il est possible que `hop` enchaîne les deux productions (42 puis 93) avant que `plop` n'affiche le résultat, et donc que `plop` affiche deux fois 93. Mais il est aussi possible que le programme se déroule « normalement » (42 puis 93).
5. Les deux *threads* font la même chose, l'un des deux décrémente `sem1` puis incrémente `res`, et incrémente `sem1`, ce qui réveille l'autre, etc. Cela permet de s'assurer qu'aucun des deux *thread* n'interrompt l'autre pendant l'incrémentation, donc le résultat est bien 6.
6. Pour `X = 2`, les deux *threads* peuvent incrémenter « en même temps » la variable `res`. Comme l'incrémentation n'est pas atomique, il peut se produire la chose suivante (entre autres possibilités) : `hop` charge dans un registre la valeur de `res` (0) pendant que `plop` charge la même valeur (0), les deux *threads* écrivent 1 dans `res` qui n'est incrémentée qu'une fois. En lançant 10000 fois le programme, c'est arrivé quelques fois.
7. *Deadlock* (étreinte mortelle), comme dans la question **3.**.

# TD, Exo 2 — A, puis B !

On considère le programme suivant :

```c
void boulot(int n);
int main(void) {
    boulot(100000);
    printf("A\n");
    boulot(100000);
    printf("B\n");
}

void boulot(int n) {
    for (int i = 0; i < n; i++)
        ;
}
```

1. Faire faire le `boulot` à des *threads* :
   - un nouveau *thread* fait le boulot puis quand il a fini écrit `A` sur le terminal ;
   - un autre nouveau *thread* fait le boulot puis quand il a fini écrit `B` sur le terminal.
2. Quel est le risque si le *thread* initial n'attend pas les autres ?
3. L'ordre des affichage est-il fixé à l'avance ?
4. À l'aide d'un sémaphore, faire en sorte que `A` soit toujours affiché avant `B`. Attention, ces *threads* doivent pouvoir travailler en même temps !
5. Reprendre l'exercice avec trois `boulots` et l'écriture de `A`, puis `B`, puis `C`, dans cet ordre.