



PROGRAMATION SYSTÈME

D.U.T. INFORMATIQUE ANNÉE SPÉCIALE

SÉBASTIEN DRANS



PARALLÉLISME – THREADS I

COURS 3 : QU'EST-CE QU'UN THREAD, COMMENT LE LANCER, LA MÉMOIRE DANS TOUT ÇA, CONCURRENCE D'ACCÈS

PARALLÉLISME – THREADS I

- Les threads
 - C'est quoi un thread ?
 - Les threads et la mémoire
- Les threads en C
 - Comment lancer un thread
 - Comment l'arrêter
 - Comment l'attendre
- La concurrence d'accès
 - Le problème
 - L'exclusion mutuelle

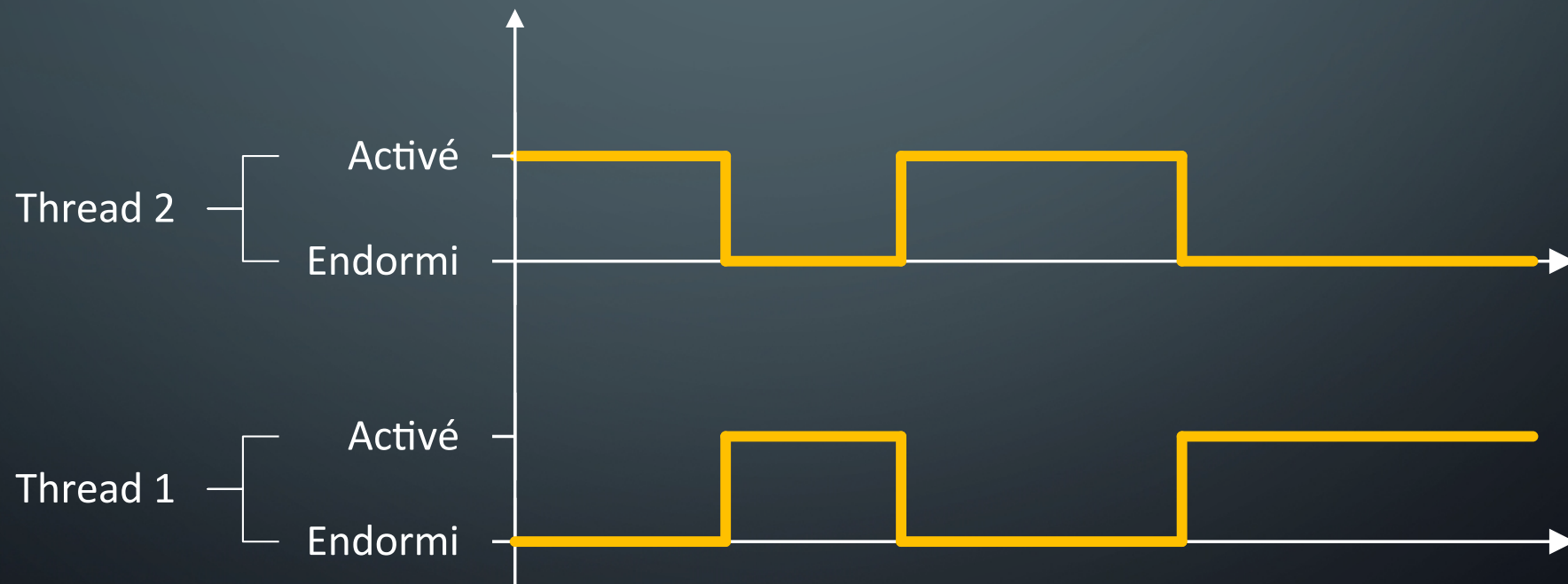
DIS PÈRE CASTOR, C'EST QUOI UN THREAD ?

- Processus léger
 - Plus rapide que la création d'un processus
- Parallélisme à l'intérieur d'un programme
 - Fonction exécutée en parallèle
 - Il existe toujours au moins un thread : celui qui lance le main
- L'ordre d'exécution des threads n'est pas définie à l'avance
 - C'est le processeur qui décide !

DIS PÈRE CASTOR, C'EST QUOI UN THREAD ?

- *L'ORDRE D'EXÉCUTION*

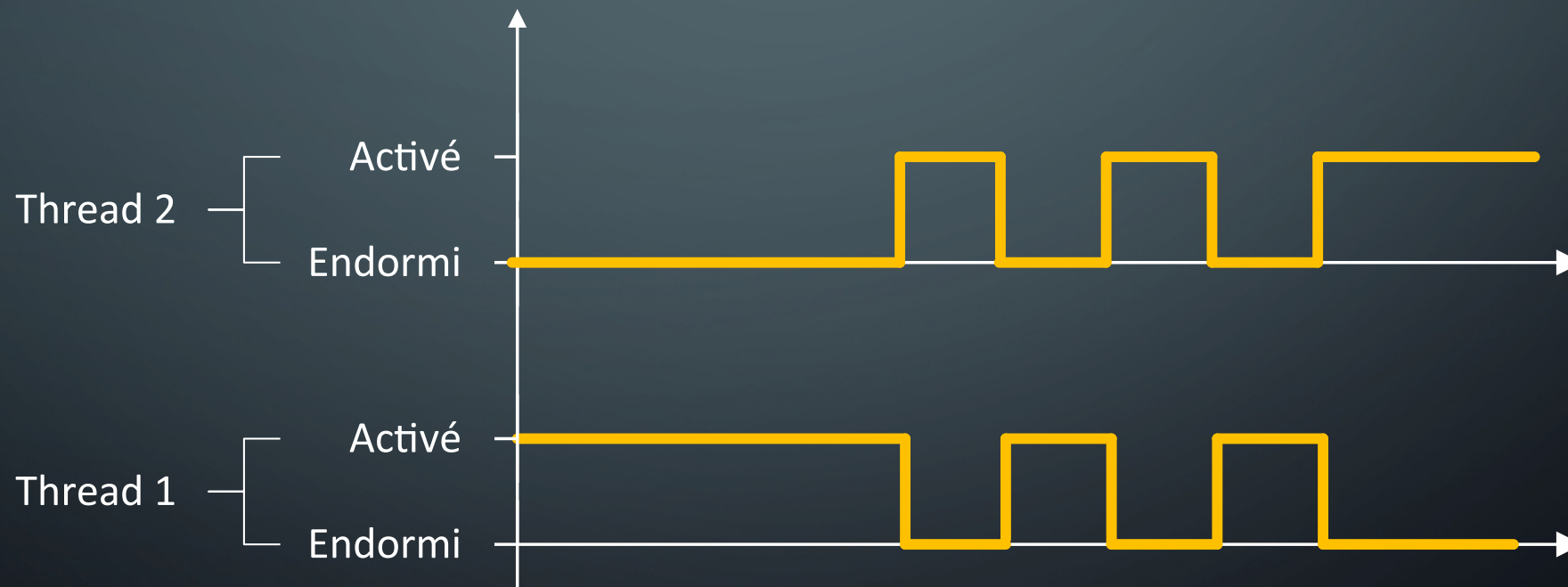
- Première exécution



DIS PÈRE CASTOR, C'EST QUOI UN THREAD ?

- *L'ORDRE D'EXÉCUTION*

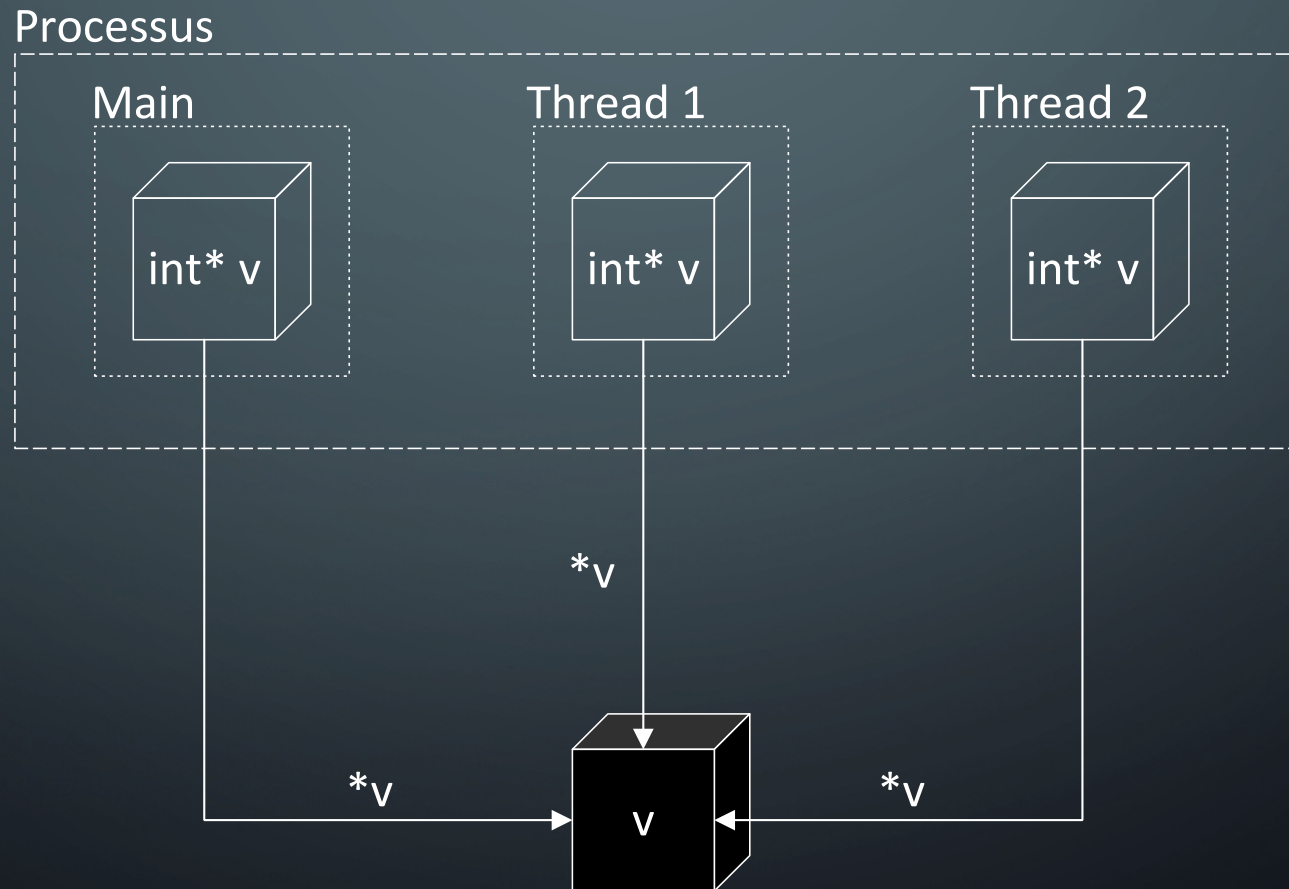
- Seconde exécution



LES THREADS ET LA MÉMOIRE

- Les threads partagent le même espace d'adressage
 - Les pointeurs indiquent la même zone mémoire
- Il est facile de partager des données entre threads
 - Passage de variable par référence !
- Cela peut entraîner des conflits, des problèmes de concurrence
 - Plusieurs threads peuvent accéder en même temps à la même chose

LES THREADS ET LA MÉMOIRE



LES THREADS EN C

- *LANCER UN THREAD*

```
int pthread_create (pthread_t * thread, pthread_attr_t * attr, void * (* fonction_parallele) (void *), void * arg)
```

Où stocker le thread ?

Avec quoi le lancer ?

Avec quel argument ?

- Retourne 0 s'il n'y a pas d'erreur
- L'argument « fonction_parallele » est un pointeur de fonction
 - La fonction prend un paramètre du type void*
 - void* : un pointeur sur quelque chose, mais on ne sait pas encore quoi...

LES THREADS EN C

- *ARRÊTER UN THREAD*

- `void pthread_exit(void* valeur_de_retour)`
- `valeur_de_retour` : permet au thread de « retourner » une valeur
 - Il ne sera pas utilisé dans le cadre du cours
 - `pthread_exit(NULL)`

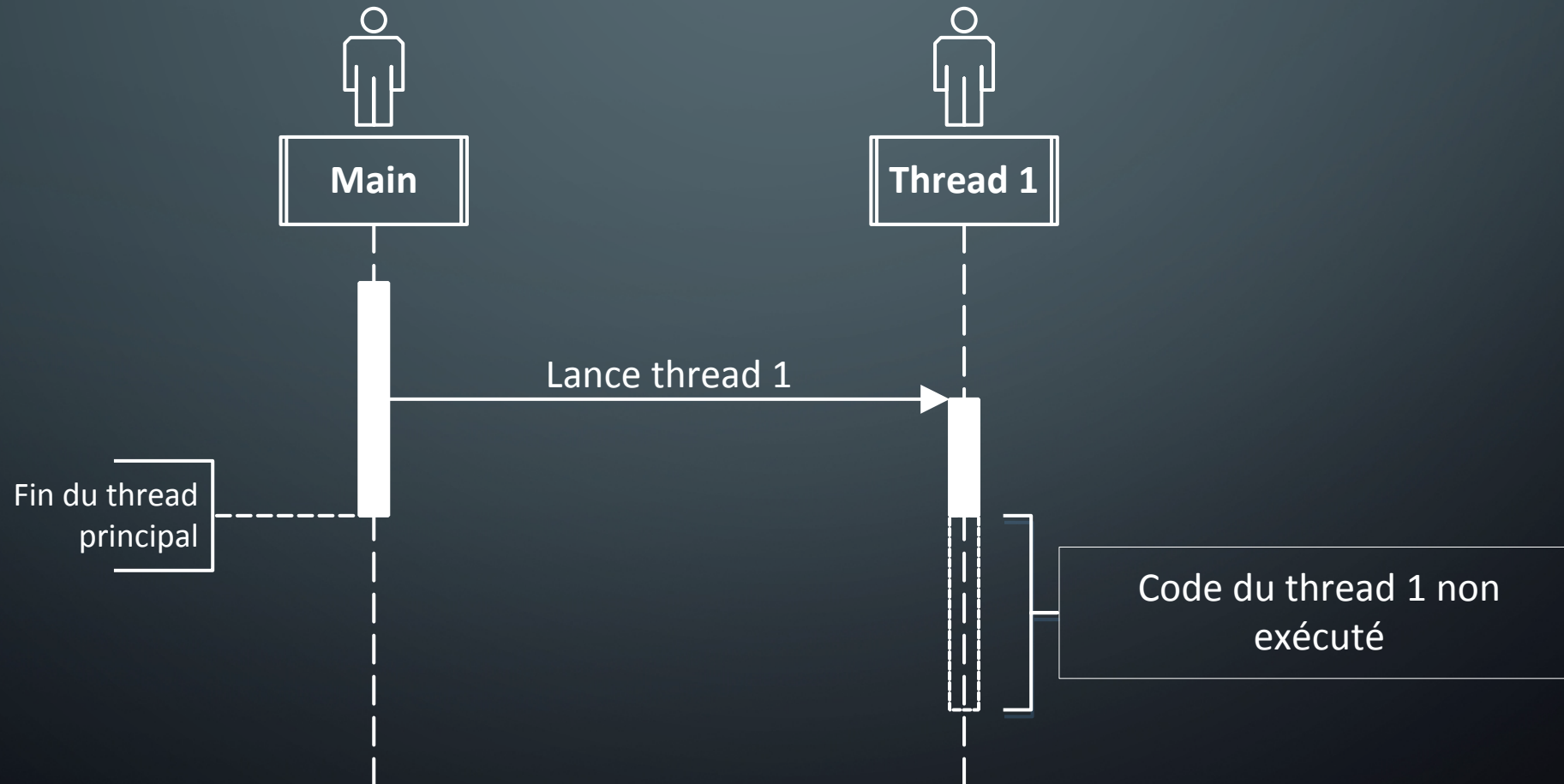
LES THREADS EN C

- *ATTENDRE UN THREAD*

- Pourquoi attendre ?
 - Un thread peut dépendre d'un autre
 - Ex. : le thread principal peut se terminer avant qu'un autre ai fini ce qu'il avait à faire

LES THREADS EN C

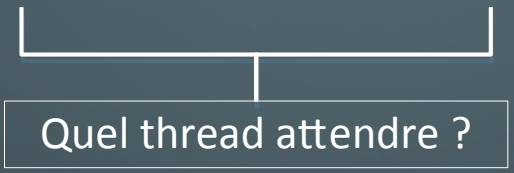
- *ATTENDRE UN THREAD*



LES THREADS EN C

- *ATTENDRE UN THREAD*

```
int pthread_join (pthread_t * thread, void** valeur_de_retour)
```



Quel thread attendre ?

- `valeur_de_retour` : permet de récupérer la valeur retournée par le thread
 - Ne sera pas utilisé dans le cadre du cours
- Retourne 0 si tout c'est bien passé

LES THREADS EN C

- L'EXEMPLE

```
graph TD; subgraph "ma_fonction_parallèle"; direction TB; A["void* ma_fonction_parallèle(void* argument) {  
    int* mon_nombre = (int*) argument;  
  
    printf(\"Mon nombre est %d\\n\", *mon_nombre);  
  
    pthread_exit(NULL);  
}"]; end; subgraph "main"; direction TB; B["int main(int argc, char** argv) {  
    int mon_nombre = 17;  
  
    pthread_t mon_thread;  
    int statut_creation = pthread_create(&mon_thread, NULL, ma_fonction_parallele, (void*) &mon_nombre);  
    if (statut_creation != 0) {  
        printf(\"Erreur de création de mon thread\\n\");  
        exit(-1);  
    }  
  
    pthread_join(mon_thread, NULL);  
    exit(0);  
}"]; end; A --> B; B --> A;
```

```
void* ma_fonction_parallèle(void* argument) {  
    int* mon_nombre = (int*) argument;  
  
    printf("Mon nombre est %d\\n", *mon_nombre);  
  
    pthread_exit(NULL);  
}  
  
int main(int argc, char** argv) {  
    int mon_nombre = 17;  
  
    pthread_t mon_thread;  
    int statut_creation = pthread_create(&mon_thread, NULL, ma_fonction_parallele, (void*) &mon_nombre);  
    if (statut_creation != 0) {  
        printf("Erreur de création de mon thread\\n");  
        exit(-1);  
    }  
  
    pthread_join(mon_thread, NULL);  
    exit(0);  
}
```

LES THREADS EN C

- *PLUSIEURS ARGUMENTS*

- Comment passer plusieurs arguments ?

LES THREADS EN C

- *PLUSIEURS ARGUMENTS*

- Comment passer plusieurs arguments ?
 - En « trichant »
 - Utilisation des structures

LES THREADS EN C

- *PLUSIEURS ARGUMENTS*

```
typedef struct {
    int numerateur;
    int denominateur;
} fraction;

void* ma_fonction_parallèle(void* argument) {
    fraction* ma_fraction = (fraction*) argument;
    ...
}

int main(int argc, char** argv) {
    fraction ma_fraction;
    ma_fraction.numerateur = 17;
    ma_fraction.denominateur = 6;

    pthread_t mon_thread;
    int statut_creation = pthread_create(&mon_thread, NULL, ma_fonction_parallele, (void*) &ma_fraction);
    ...
}
```

LA CONCURRENCE D'ACCÈS

- *LE PROBLÈME*

- Mémoire partagée
 - 😊 Plusieurs threads peuvent accéder à une valeur commune
 - ☹️ Ils peuvent la modifier en même temps

LA CONCURRENCE D'ACCÈS

- *LE PROBLÈME*

Banque	Vous		Votre conjoint(e)
Solde = 1 000 €			
	Lire solde		
	Ajoute 500 €		
Solde = 1 500 €	Écrire solde		
			Lire solde
			Retire 500 €
Solde = 1000 €			Écrire solde

LA CONCURRENCE D'ACCÈS

- *LE PROBLÈME*

Banque	Vous		Votre conjoint(e)
Solde = 1 000 €			
	Lire solde		
			Lire solde
	Ajoute 500 €		
Solde = 1 500 €	Écrire solde		
			Retire 500 €
Solde = 500 €			Écrire solde

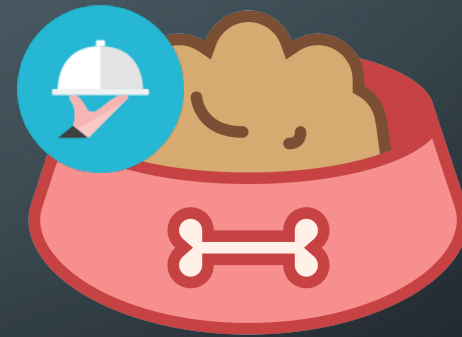
LA CONCURRENCE D'ACCÈS

- *LA SOLUTION*

- Les exclusions mutuelles
 - L'idée est de bloquer l'accès à une variable en commun
 - Protéger une section critique (ex. : le solde du compte)
- Mutex \approx jeton
 - Si le thread n'a pas le jeton, il n'accède pas à la section critique

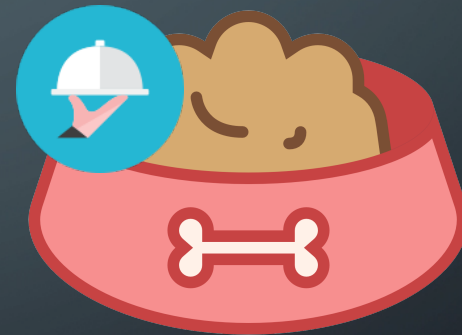
LA CONCURRENCE D'ACCÈS

- *LA SOLUTION*



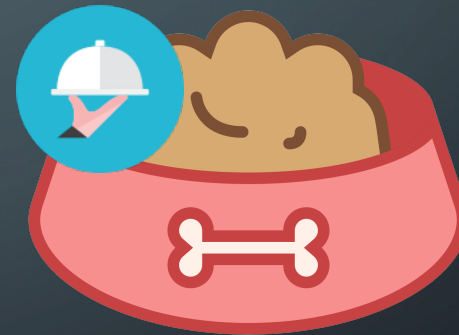
LA CONCURRENCE D'ACCÈS

- *LA SOLUTION*



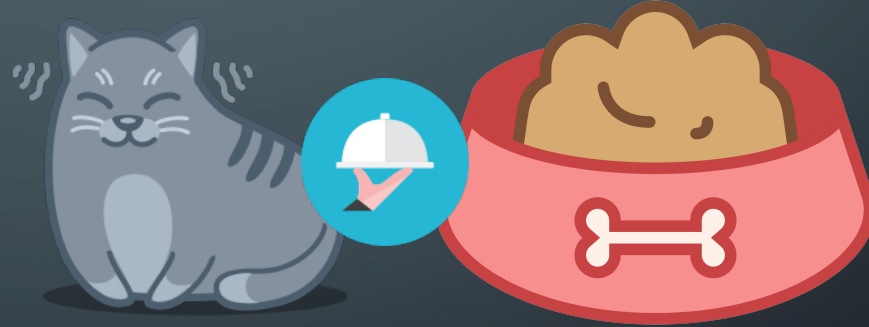
LA CONCURRENCE D'ACCÈS

- *LA SOLUTION*



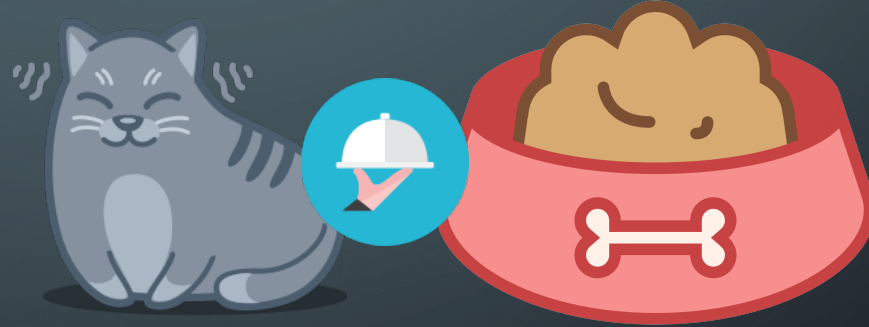
LA CONCURRENCE D'ACCÈS

- *LA SOLUTION*



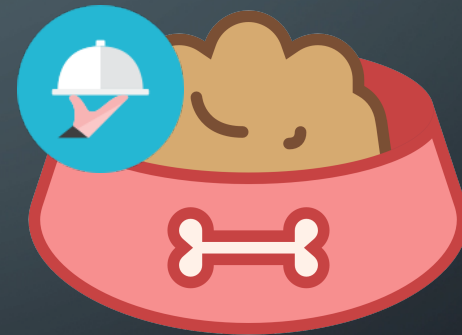
LA CONCURRENCE D'ACCÈS

- *LA SOLUTION*



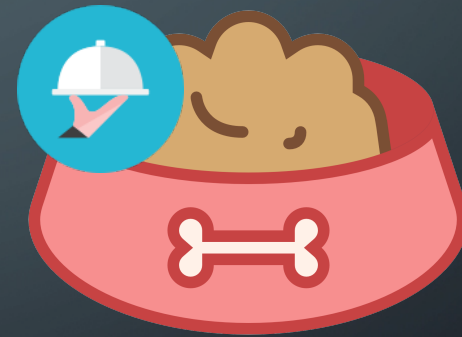
LA CONCURRENCE D'ACCÈS

- *LA SOLUTION*



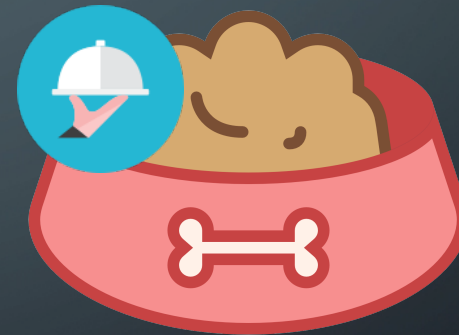
LA CONCURRENCE D'ACCÈS

- *LA SOLUTION*



LA CONCURRENCE D'ACCÈS

- *LA SOLUTION*



LA CONCURRENCE D'ACCÈS

- *LA SOLUTION*



LA CONCURRENCE D'ACCÈS

- *EN C : INITIALISATION*

- `int pthread_mutex_init(pthread_mutex_t* mutex, pthread_mutex_attr_t* attributs)`
- Nous n'utiliserons pas l'argument « attributs »
- Retourne 0 si tout c'est bien passé
- Exemple

```
pthread_mutex_t mon_mutex;  
int resultat = pthread_mutex_init(&mon_mutex, NULL);  
if(resultat != 0) {  
    printf("Quelque chose c'est mal passé\n");  
}
```

LA CONCURRENCE D'ACCÈS

- *EN C : VERROUILLAGE*

- `int pthread_mutex_lock(pthread_mutex_t* mutex)`
- Fonction bloquante
 - Tant que le thread n'obtient pas le verrou, il est en attente
- Retourne 0 si tout c'est bien passé

LA CONCURRENCE D'ACCÈS

- *EN C : DÉVERROUILLAGE*

- `int pthread_mutex_unlock(pthread_mutex_t* mutex)`
- Retourne 0 si tout c'est bien passé

LA CONCURRENCE D'ACCÈS

- *EN C : EXEMPLE*

```
pthread_mutex_t mon_mutex;
```

```
void* ma_fonction_parallèle(void* argument) {  
    int mon_numero;  
    int* numero = (int*) argument;  
    pthread_mutex_lock(&mon_mutex);  
    mon_numero = numero;  
    numero = numero + 1;  
    pthread_mutex_unlock(&mon_mutex);  
    ...  
}
```

Section critique

```
int main(int argc, char** argv) {  
    int numero_thread = 0;  
  
    pthread_mutex_init(&mon_mutex, NULL);  
  
    pthread_t premier_thread;  
    pthread_t second_thread;  
    pthread_create(&premier_thread, NULL, ma_fonction_parallele, (void*) &numero_thread);  
    pthread_create(&second_thread, NULL, ma_fonction_parallele, (void*) &numero_thread);  
    ...  
}
```

LA CONCURRENCE D'ACCÈS

- *EN C : QUELQUES CONSEILS*

- Un thread qui verrouille deux fois de suite le même mutex bloque le programme
- Déverrouiller deux fois le même mutex poser problème
- Un et un seul mutex par section critique !
 - Indiquez clairement le rôle de votre mutex (ahhh les commentaires... 😊)
- La bibliothèque à inclure est pthread.h
 - `#include <pthread.h>`
- Il est nécessaire de compiler avec l'option « `lpthread` »
 - `gcc mon_programme_en_parallele.c -o mon_executable_avec_des_threads -lpthread`



PARALLÉLISME – THREADS II

COURS 4 : LES CONDITIONS

PARALLÉLISME – THREADS II

- Les conditions
 - Pourquoi les conditions ?
 - Principe
- Les conditions en C
 - Déclarer une conditions
 - L'initialiser
 - Attendre qu'une condition soit remplie
 - Déclarer qu'une condition est remplie

LES CONDITIONS

- *POURQUOI ?*

- L'exclusion mutuelle
 - Permet de résoudre le problème de concurrence d'accès
 - De sécuriser une portion de code critique
- Mais
 - Comment bloquer un thread en attendant qu'une certaine opération soit terminée ?
 - L'exclusion mutuelle seule ne répond pas à cette question

LES CONDITIONS

- *POURQUOI ?*

- Les conditions
 - Permettent de bloquer un thread en attendant qu'une certaine condition soit remplie 😊
 - Sont complémentaires de l'exclusion mutuelle

LES CONDITIONS

- *LE PRINCIPE*

- Deux fonctions essentielles
 - Attendre qu'une condition soit remplie
 - Signaler qu'une condition est remplie
- Typiquement
 - Un thread effectue l'opération d'attente
 - Un second lui signalera qu'il peut reprendre son exécution

LES CONDITIONS

- *LE PRINCIPE : L'ATTENTE*

- Un thread obtient un jeton (entre dans la section critique)
 - Les conditions sont toujours associées à un jeton !
- Se met en pause
 - Attend qu'un autre thread le réveille
- Reprend son exécution lorsqu'il reçoit le signal
 - La condition est remplie !
 - Lorsqu'il se réveille, il a de nouveau le jeton
- Libère le jeton

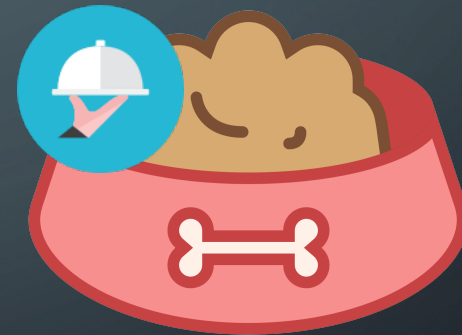
LES CONDITIONS

- *LE PRINCIPE : ENVOI DU SIGNAL*

- C'est relativement basique 😊
- Le thread signale que la condition est remplie pour « réveiller » un autre thread

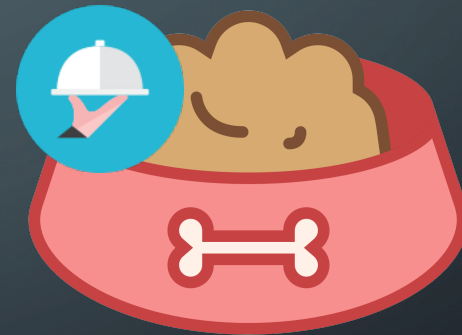
LES CONDITIONS

- *LE PRINCIPE : LE RETOUR DES CHATS*



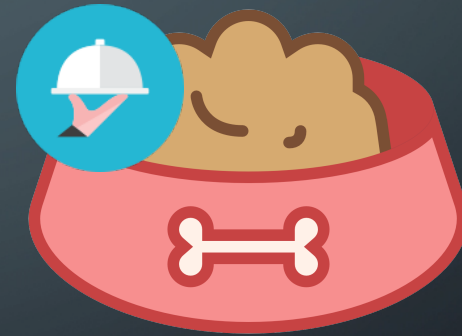
LES CONDITIONS

- *LE PRINCIPE : LE RETOUR DES CHATS*



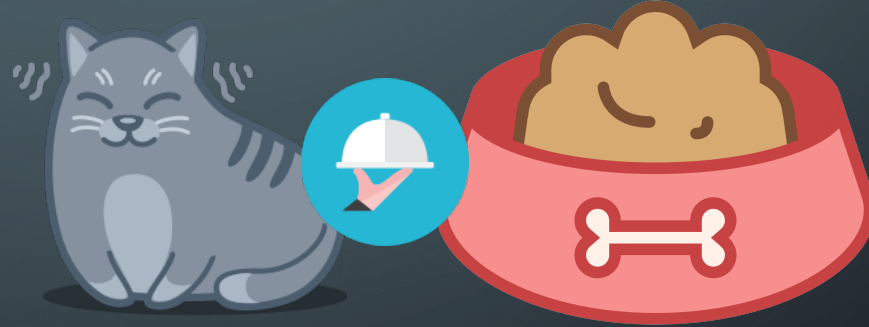
LES CONDITIONS

- *LE PRINCIPE : LE RETOUR DES CHATS*



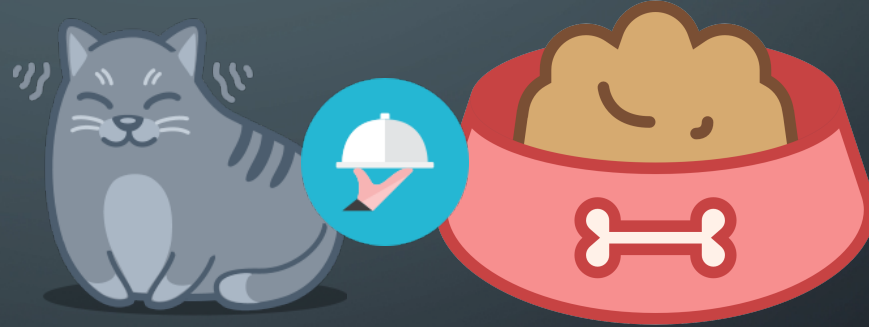
LES CONDITIONS

- *LE PRINCIPE : LE RETOUR DES CHATS*



LES CONDITIONS

- *LE PRINCIPE : LE RETOUR DES CHATS*



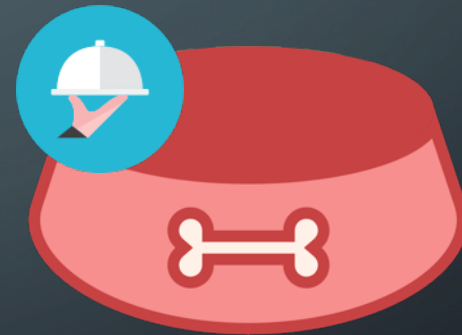
LES CONDITIONS

- *LE PRINCIPE : LE RETOUR DES CHATS*



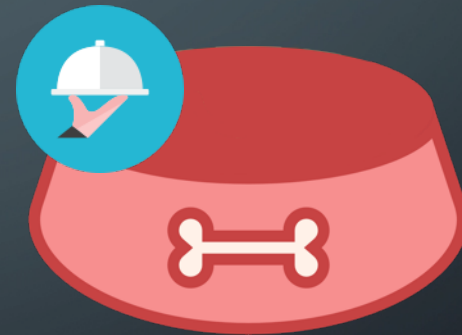
LES CONDITIONS

- *LE PRINCIPE : LE RETOUR DES CHATS*



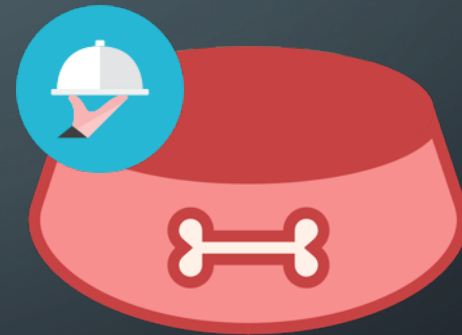
LES CONDITIONS

- *LE PRINCIPE : LE RETOUR DES CHATS*



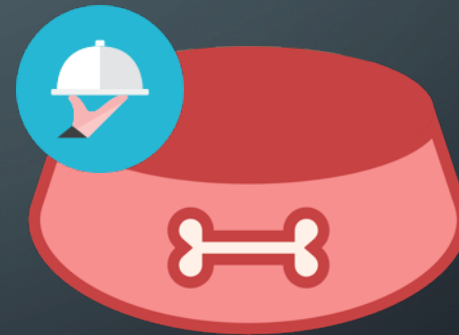
LES CONDITIONS

- *LE PRINCIPE : LE RETOUR DES CHATS*



LES CONDITIONS

- *LE PRINCIPE : LE RETOUR DES CHATS*



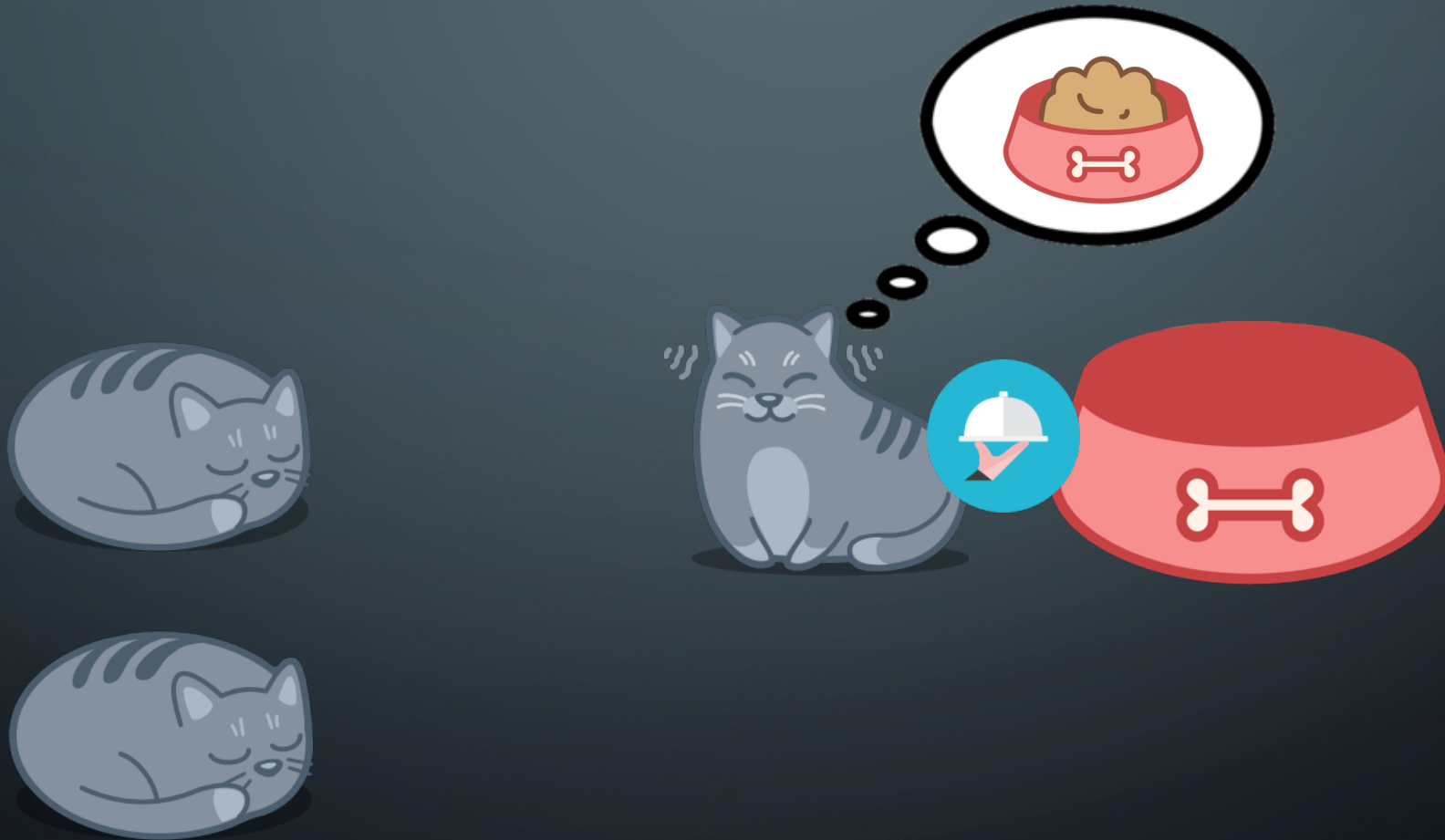
LES CONDITIONS

- *LE PRINCIPE : LE RETOUR DES CHATS*



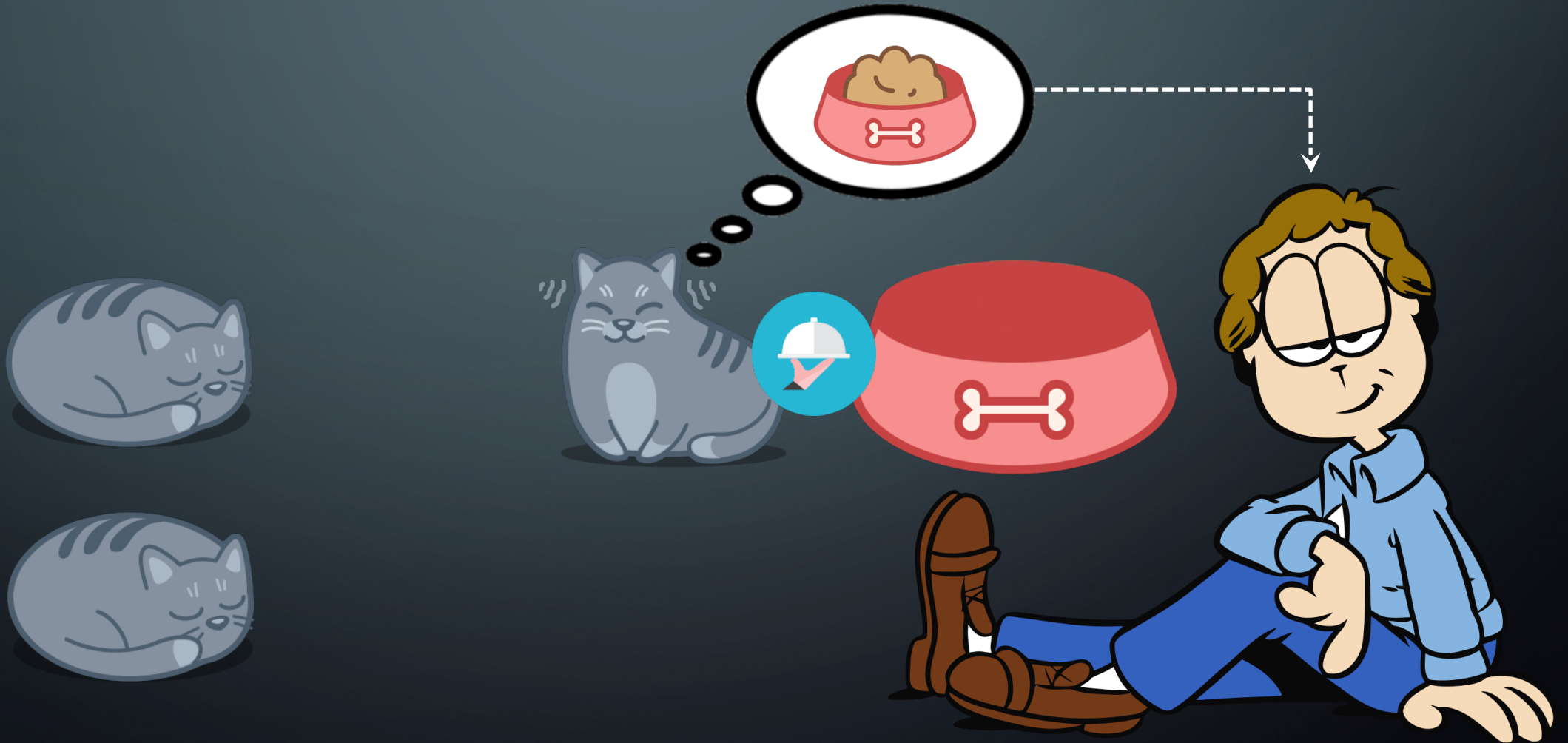
LES CONDITIONS

- *LE PRINCIPE : LE RETOUR DES CHATS*



LES CONDITIONS

- *LE PRINCIPE : LE RETOUR DES CHATS*



LES CONDITIONS

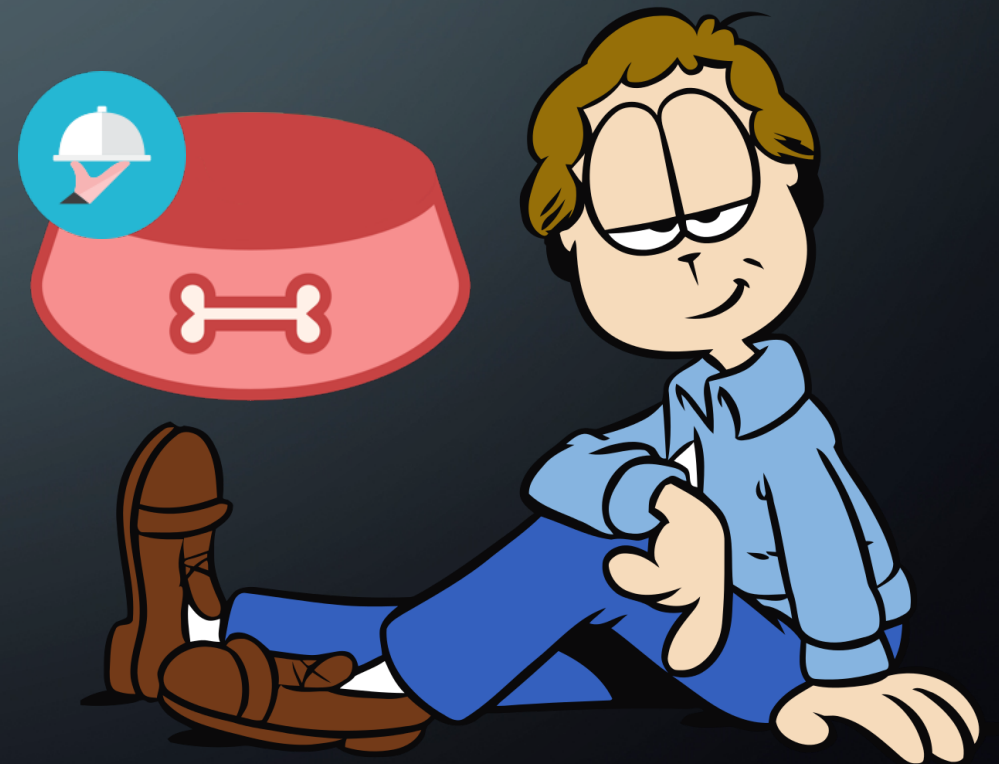
- *LE PRINCIPE : LE RETOUR DES CHATS*



LES CONDITIONS



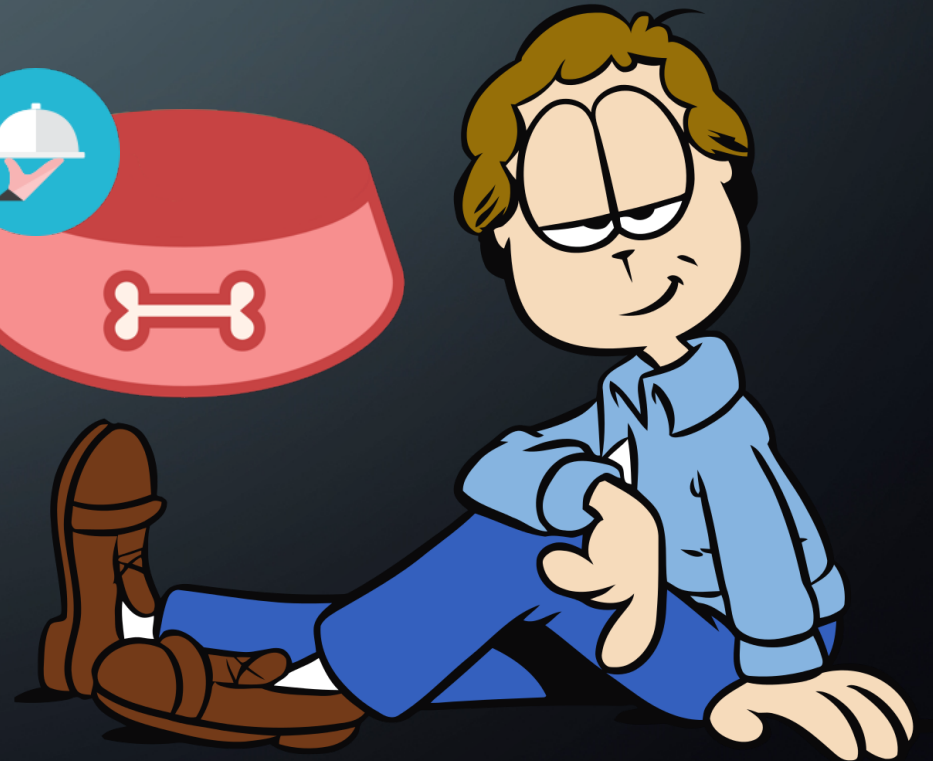
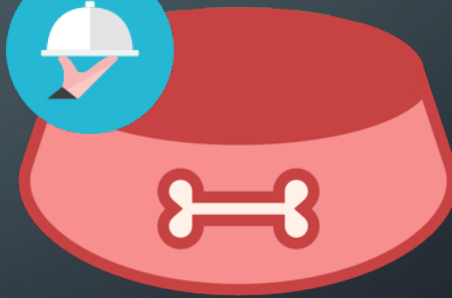
LE PRINCIPE : LE RETOUR DES CHATS



LES CONDITIONS



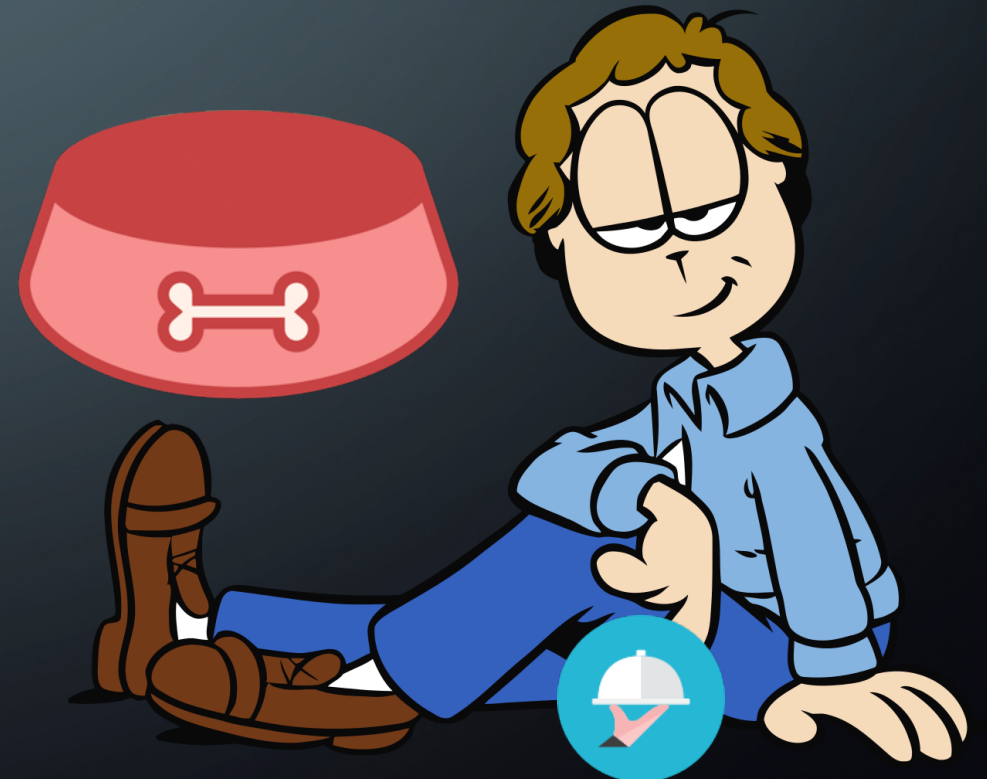
LE PRINCIPE : LE RETOUR DES CHATS



LES CONDITIONS



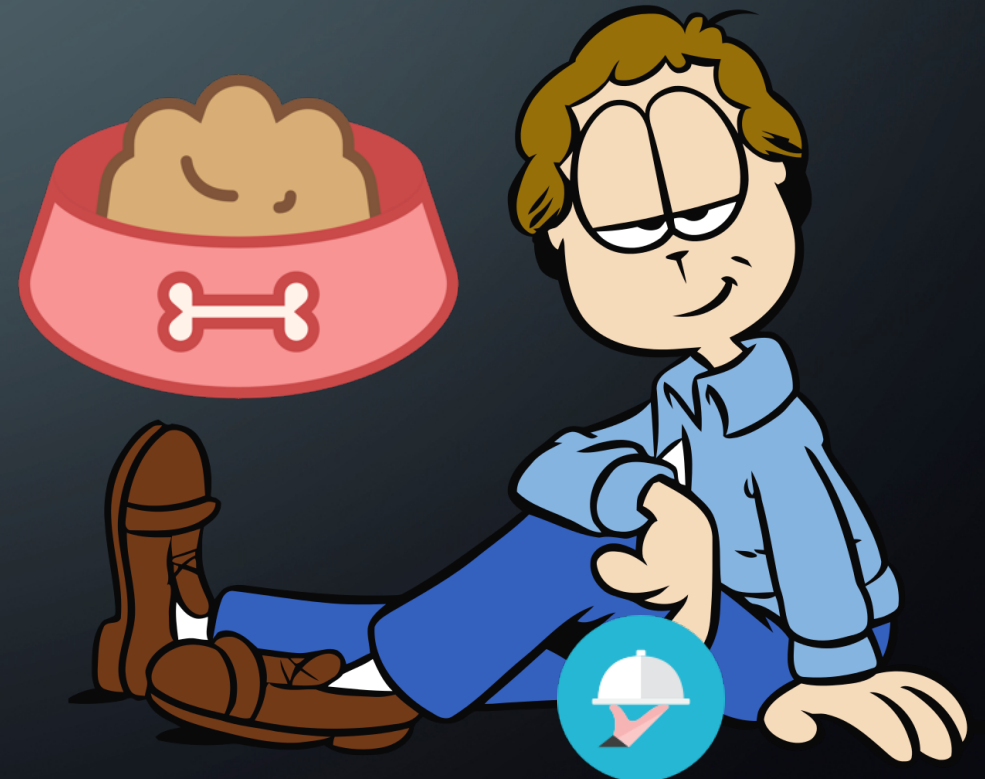
LE PRINCIPE : LE RETOUR DES CHATS



LES CONDITIONS

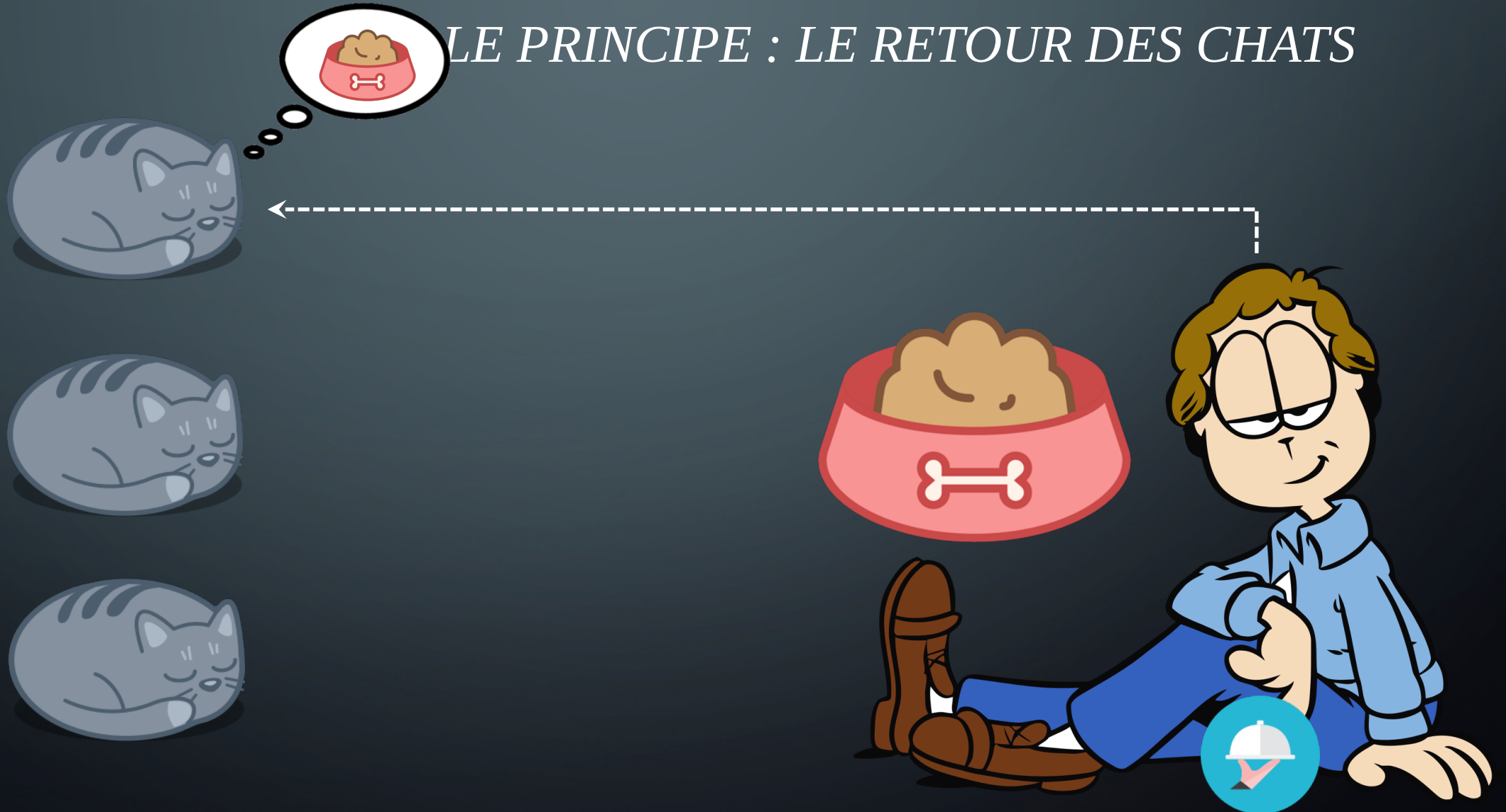


LE PRINCIPE : LE RETOUR DES CHATS



LES CONDITIONS

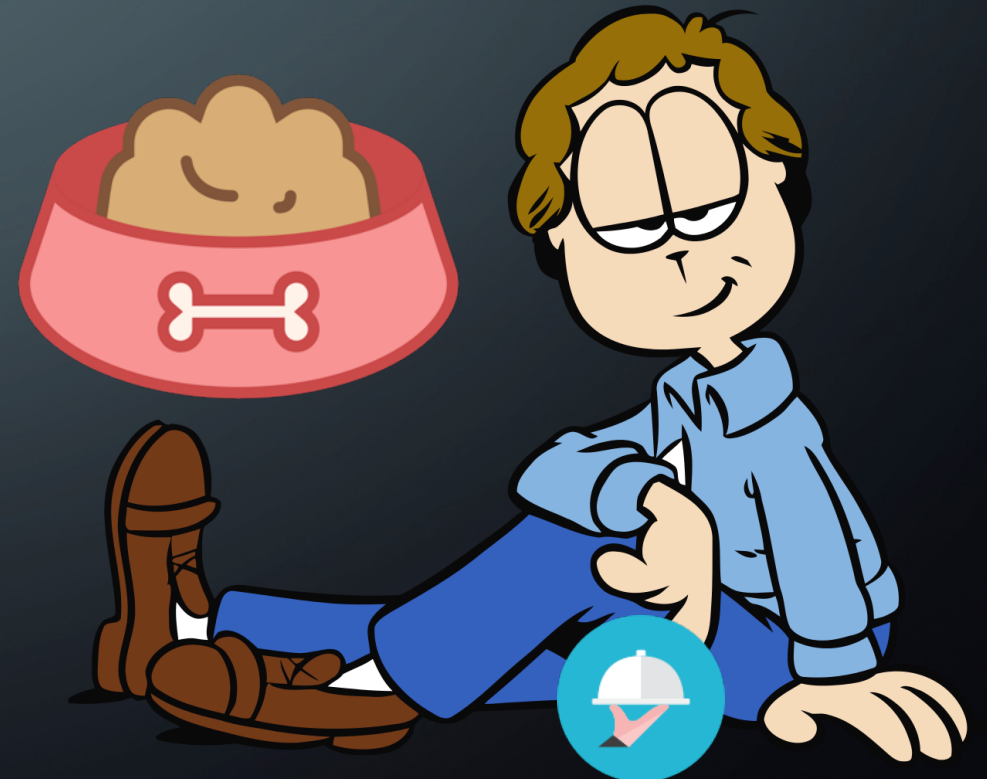
LE PRINCIPE : LE RETOUR DES CHATS



LES CONDITIONS



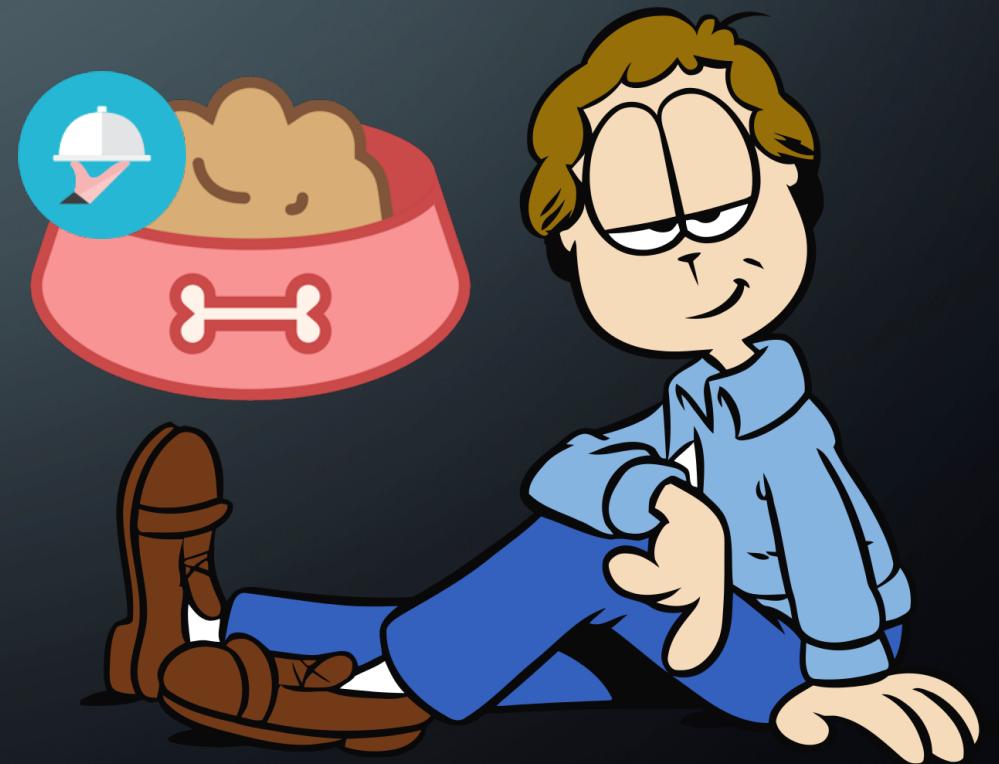
LE PRINCIPE : LE RETOUR DES CHATS



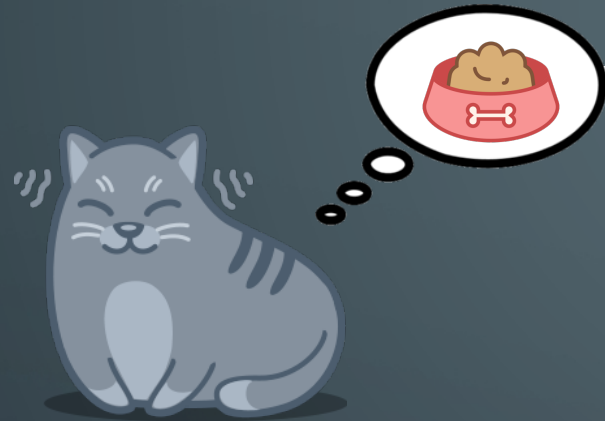
LES CONDITIONS



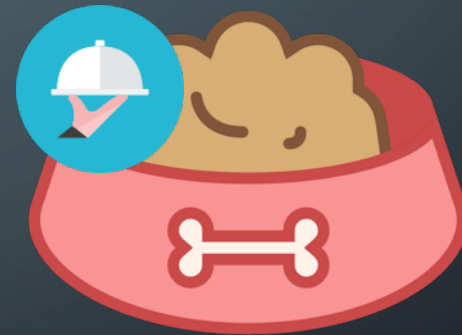
LE PRINCIPE : LE RETOUR DES CHATS



LES CONDITIONS

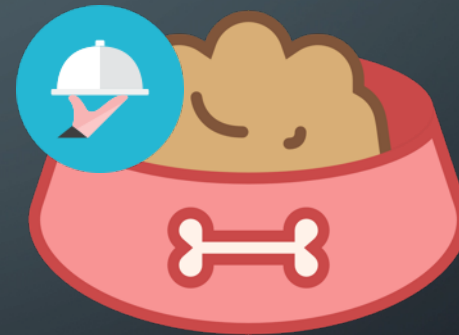


LE PRINCIPE : LE RETOUR DES CHATS



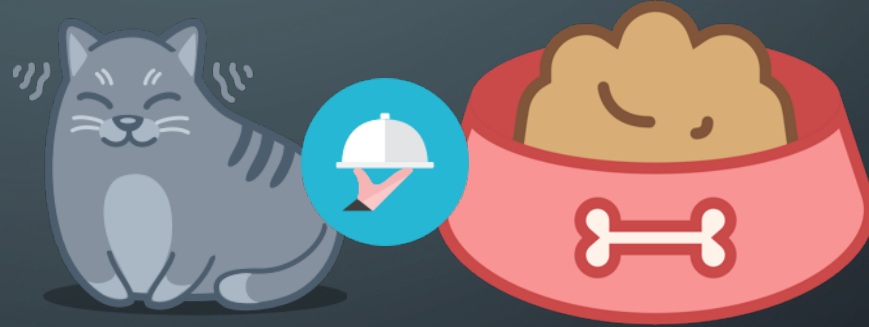
LES CONDITIONS

- *LE PRINCIPE : LE RETOUR DES CHATS*



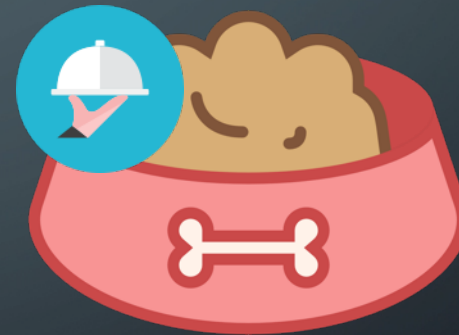
LES CONDITIONS

- *LE PRINCIPE : LE RETOUR DES CHATS*



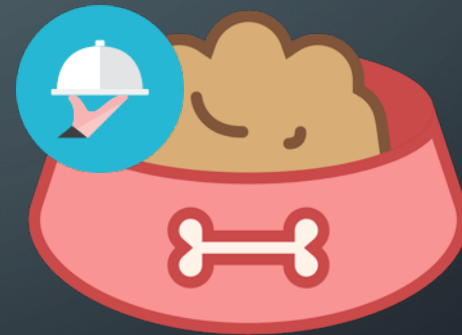
LES CONDITIONS

- *LE PRINCIPE : LE RETOUR DES CHATS*



LES CONDITIONS

- *LE PRINCIPE : LE RETOUR DES CHATS*



LES CONDITIONS EN C

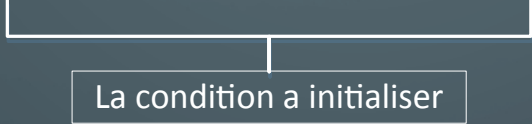
- *DÉCLARER UNE CONDITION*

- `pthread_cond_t ma_condition;`
- Déclare une condition
 - Non initialisée !

LES CONDITIONS EN C

- *INITIALISER UNE CONDITION*

```
int pthread_cond_init(pthread_cond_t* condition, pthread_condattr_t* attr)
```



La condition a initialiser

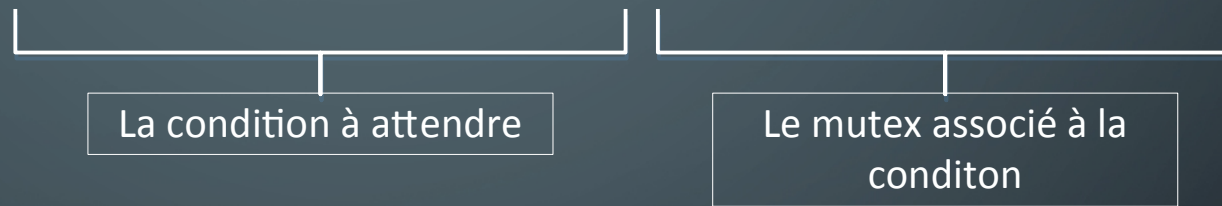
- Nous n'utiliserons pas l'argument « attr »
- Retourne 0 si tout c'est bien passé
- Exemple

```
pthread_cond_t ma_condition;  
int resultat = pthread_cond_init(&ma_condition, NULL);  
if(resultat != 0) {  
    printf("Quelque chose c'est mal passé\n");  
}
```

LES CONDITIONS EN C

- *ATTENDRE UNE CONDITION*

```
int pthread_cond_wait(pthread_cond_t* condition, pthread_mutex_t* mutex)
```



- Un condition est toujours associée à un mutex !
- Retourne 0 si tout c'est bien passé

LES CONDITIONS EN C

- *ATTENDRE UNE CONDITION*

```
pthread_mutex_t mon_mutex;  
pthread_cond_t ma_condition;
```

```
void* ma_fonction_parallèle(void* argument) {
```

```
    ...
```

```
    pthread_mutex_lock(&mon_mutex);
```

```
    if(« J ai besoin de quelque chose ») {
```

```
        pthread_cond_wait(&ma_condition, &mon_mutex);
```

```
    }
```

```
    ...
```

```
    pthread_mutex_unlock(&mon_mutex);
```

```
    ...
```

```
}
```

```
int main(int argc, char** argv) {
```

```
    pthread_mutex_init(&mon_mutex, NULL);
```

```
    pthread_cond_init(&ma_condition, NULL);
```

```
    pthread_t mon_thread;
```

```
    pthread_create(&mon_thread, NULL, ma_fonction_parallele, NULL);
```

```
    ...
```


```
}
```

```
pthread_mutex_unlock(&mutex)  
attendre_en_bloquant(cond)  
pthread_mutex_lock(&mutex)
```


LES CONDITIONS EN C

- *SIGNALER QU'UNE CONDITION EST REMPLIE*

```
int pthread_cond_signal(pthread_cond_t* condition)
```



La condition à envoyer, à
marquer comme
« remplie »

- Retourne 0 si tout c'est bien passé

LES CONDITIONS EN C

- *SIGNALER QU'UNE CONDITION EST REMPLIE*

```
pthread_mutex_t mon_mutex;  
pthread_cond_t ma_condition;
```

```
void* ma_fonction_qui_s_endort(void* argument) {  
    while(1) {  
        pthread_mutex_lock(&mon_mutex);  
        pthread_cond_wait(&ma_condition, &mon_mutex);  
        pthread_mutex_unlock(&mon_mutex);  
    }  
}
```

```
void* ma_fonction_qui_reveille(void* argument) {  
    while(1) {  
        sleep(rand() % 10);  
        pthread_cond_signal(&ma_condition);  
    }  
}
```

```
int main(int argc, char** argv) {  
    pthread_mutex_init(&mon_mutex, NULL);  
    pthread_cond_init(&ma_condition, NULL);  
  
    pthread_t premier_thread, second_thread;  
    pthread_create(&premier_thread, NULL, ma_fonction_qui_s_endort, NULL);  
    pthread_create(&second_thread, NULL, ma_fonction_qui_reveille, NULL);  
    ...  
}
```

Débloque un thread qui attend la condition « ma_condition »

À VOUS DE JOUER

- TD 4

- Emplacement du sujet

<https://goo.gl/0aG9Nn>

- Objectif : exercices 1 à 3



PARALLÉLISME – THREADS III

COURS 5 : LES SÉMAPHORES

PARALLÉLISME – THREADS III

- Les sémaphores
 - Pourquoi les sémaphores ?
 - Qu'est-ce qu'un sémaphore ?
- Les sémaphores en C
 - Déclarer un sémaphore
 - L'initialiser
 - Réserver une place
 - Libérer une place

LES SÉMAPHORES

- *POURQUOI ?*

- L'exclusion mutuelle
 - Permet de synchroniser les thread
 - Un thread à la fois dans la section critique
- Mais
 - Si plusieurs thread peuvent accéder en même temps à la ressource critique ?
 - Pool de connexions à une base de données
 - Limiter des tâches « gourmandes » qui s'exécutent en même temps
 - N-1 processus (N étant le nombre de cœurs de la machines)

LES SÉMAPHORES

- *QU'EST-CE QUE C'EST ?*

- Goulot d'étranglement
 - Compteur avec un certain nombre de places
 - Prendre un jeton décrémente le compteur
 - On ne peut pas prendre un jeton s'il n'y en a plus !
 - Libérer un jeton incrémente le compteur
- \approx mutex avec plusieurs jetons
 - Le mutex peut être vu comme un cas particulier du sémaphore
 - Si j'enlève le jeton vert, j'obtiens bien un mutex, non ?



LES SÉMAPHORES

- *LES CHATS, SUITE ET FIN*



LES SÉMAPHORES

- *LES CHATS, SUITE ET FIN*



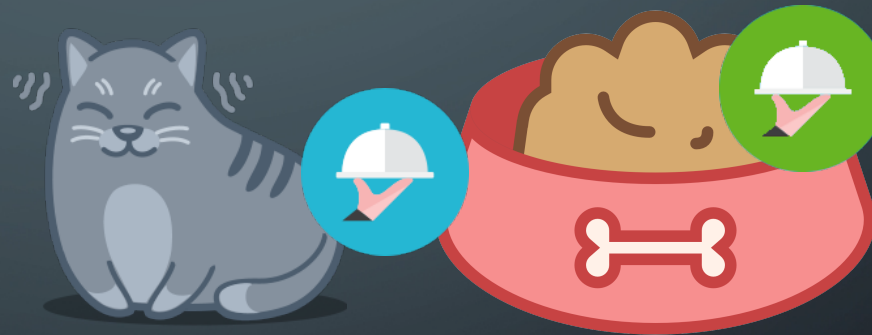
LES SÉMAPHORES

- *LES CHATS, SUITE ET FIN*



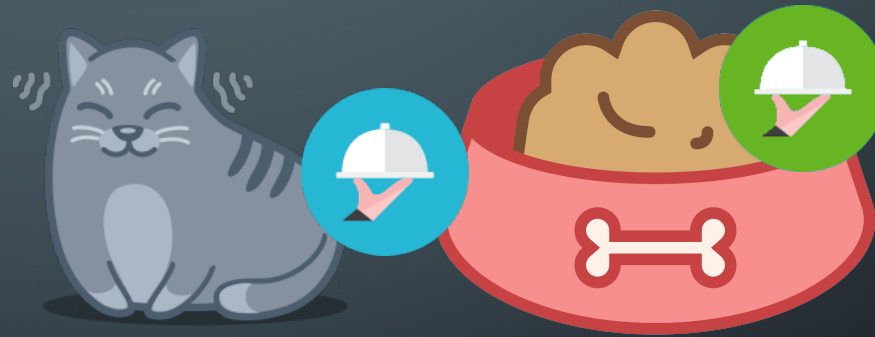
LES SÉMAPHORES

- *LES CHATS, SUITE ET FIN*



LES SÉMAPHORES

- *LES CHATS, SUITE ET FIN*



LES SÉMAPHORES

- *LES CHATS, SUITE ET FIN*



LES SÉMAPHORES

- *LES CHATS, SUITE ET FIN*



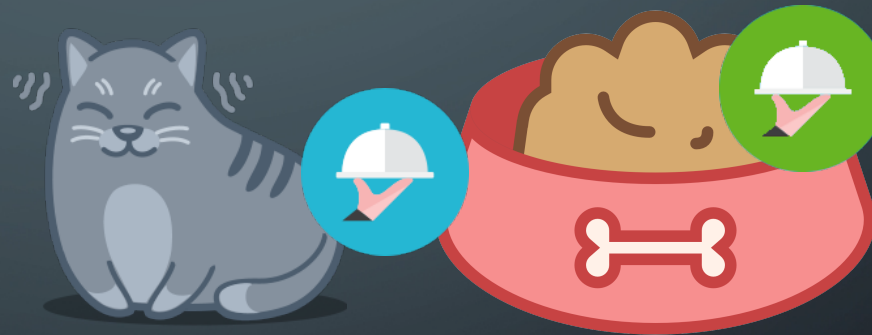
LES SÉMAPHORES

- *LES CHATS, SUITE ET FIN*



LES SÉMAPHORES

- *LES CHATS, SUITE ET FIN*



LES SÉMAPHORES

- *AVARICE*

- Comment faire en sorte qu'un seul thread occupe seul un sémaphore à plusieurs places ?

LES SÉMAPHORES

- *AVARICE*

- Comment faire en sorte qu'un seul thread occupe seul un sémaaphore à plusieurs places ?
 - Il peut réserver plusieurs places pour « affamer » les autres



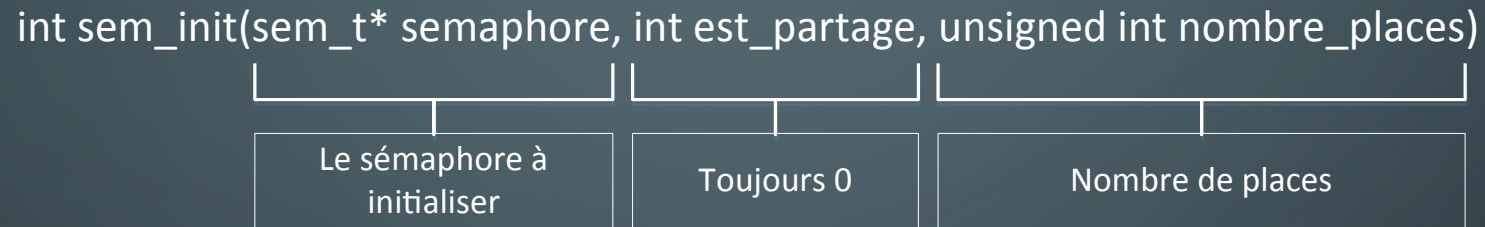
LES SÉMAPHORES EN C

- *DÉCLARER UN SÉMAPHORE*

- `Sem_t mon_semaphore;`
- Déclare un sémaphore
 - Non initialisé !

LES SÉMAPHORES EN C

- *INITIALISER UN SÉMAPHORE*



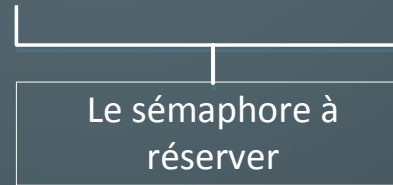
- Nous n'utiliserons pas l'argument « estPartage »
 - Toujours 0 (partagé entre les threads)
- Retourne 0 si tout c'est bien passé
- Exemple

```
sem_t mon_semaphore;  
int resultat = sem_init(&mon_semaphore, 0, 17);  
if(resultat != 0) {  
    printf("Quelque chose c'est mal passé\n");  
}
```

LES SÉMAPHORES EN C

- *RÉSERVER UNE PLACE*

```
int sem_wait(sem_t* semaphore)
```



- Réserve une place si le compteur est strictement positif et le décrémente
- Sinon, attend qu'une place se libère 😊
- Retourne 0 si tout c'est bien passé

LES SÉMAPHORES EN C

- *LIBÉRER UNE PLACE*

```
int sem_post(sem_t* semaphore)
```



Le sémaphore à libérer

- Libère une place dans le sémaphore
 - Incrément le compteur d'une place
- Retourne 0 si tout c'est bien passé

LES SÉMAPHORES EN C

- *EXEMPLE*

```
#define NB_VISITEURS 100
#define NB_SIEGES 20
sem_t semaphore_attraction;
```

```
void* visiter_attraction(void* argument) {
```

```
...
```

```
sem_wait(&mon_semaphore);
```

```
s_amuser();
```

```
sem_post(&mon_semaphore);
```

```
...
```

```
}
```

Seuls 20 visiteurs peuvent
s'amuser en même temps dans
l'attraction alors qu'ils sont 100

```
int main(int argc, char** argv) {
```

```
sem_init(&semaphore_attraction, 0, 20);
```

```
pthread_t mes_visiteurs[NB_SIEGES];
```

```
int i;
```

```
for(i = 0; i < NB_VISITEURS; i++) pthread_create(&mes_visiteurs[i], NULL, visiter_attraction, NULL);
```

```
...
```

```
}
```

À VOUS DE JOUER

- TD 6

- Il est nécessaire d'inclure la bibliothèque « semaphore.h »
 - `#include <semaphore.h>`
- Emplacement du sujet

<https://goo.gl/1dNg2W>

- Objectif : exercices 1 à 5