# Les Mutex et le pROJET

Naïm Qachri, Stéphane Fernandes Medeiros, Eythan Levy

Université Libre de Bruxelles
2010/2011

# Les Mutexes

- Le but de ces structures est de permettre d'installer une espèce de sémaphore sur certaines ressources partagées entre deux threads pour éviter les problèmes d'accès concurrentiels

# Les Mutexes

- Il y a 2 appels pour initialiser un mutex :

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Pour obtenir les attributs par défaut d'un mutex à l'initialisation, il suffit de mettre la valeur constante de la fonction *init* à NULL.

# Les Mutexes

- Voici l'appel pour en détruire un mutex

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

# Les Mutexes

- Il existe deux appels pour verrouiller une zone critique d'exécution

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

• Le premier verrouiller le mutex, si ce dernier l'est déjà alors il va attendre jusqu'à ce que ce dernier soit déverrouillé.

• Le second fait la même chose mais de manière non bloquante.

• Tous les deux retournent zéro en cas de réussite.

# Les Mutexes

- L'appel pour déverrouiller un mutex est le suivant

```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Il est possible d'être plus fin dans la gestion des mutexes pour obtenir des comportements plus particuliers dans certains cas.

# Introduction

- Deux phases de codage (phases 2 et 3).
- Phase 2: codage d'un premier protoype *fonctionnel*.
- Phase 3: codage du projet complet, avec des modifications éventuelles de l'énoncé de la phase 2.
- Méthode de design par *processus itératif* (cf. AMI).
- Remise du code source et d'un rapport détaillé à la fin de chaque phase.
- Défense après chaque phase du projet.

# Défense du projet

- Défense orale et présentation du programme après chaque phase.
- Présence obligatoire de tous les membres à la défense (tous les membres seront intérrogés).
- Le programme doit compiler dans les salles PC !

# Le rapport: documents à remettre

- Votre rapport de la phase 1 corrigé selon nos remarques (l'ancien *et* le nouveau).
- Un diagramme de classes illustrant votre découpe en classes C++:
  - Montrer de manière simple les différentes composantes du code (client/serveur, code métier).
  - Montrer les intéractions statiques entre elles (qui utilise qui).
  - Pas rentrer dans tous les détails (classe `List`...).

# Le rapport: documents à remettre

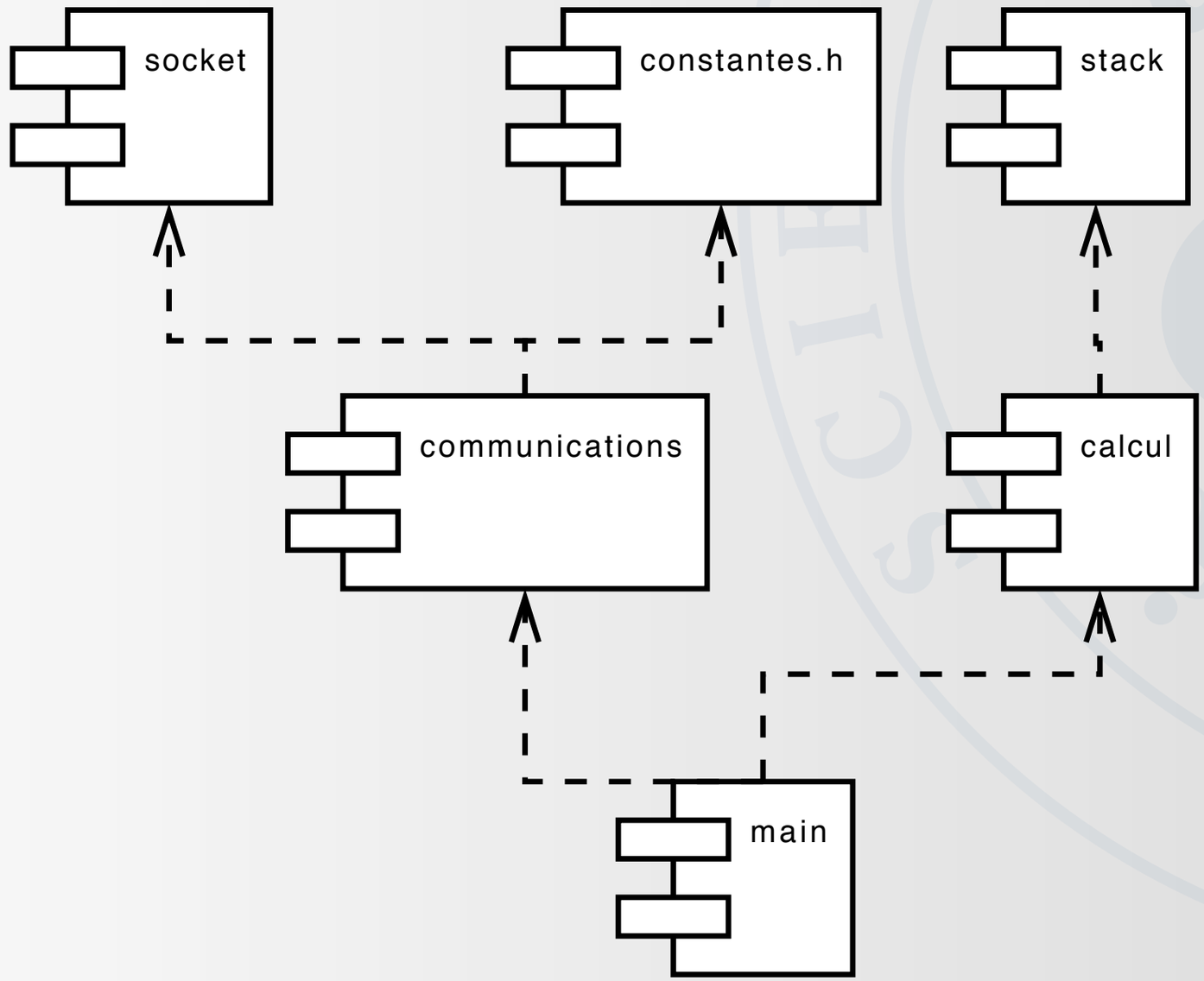
- Un diagramme des composants reprenant votre découpe en modules (fichiers).
- Un diagramme d'activités **commenté** détaillant la découpe en processus.
- Un diagramme de séquence reprenant toutes les communications entre le serveur et le client.
- Le code C++ commenté.
- Un makefile.

# Le diagramme de composants

- Un composant = un module.
- Module:
  - Paire de fichiers (.cpp, .hpp)
  - Fichier .hpp seul
  - Fichier main
- Un module est lié à un autre module si il fait appel à une fonction, une constante, ou une autre déclaration présente dans ce module.
- Le diagramme des composants montre les modules et leurs liens.
- Les flèches sont orientées selon la relation "utilise".
- Veiller à l’absence de cycles dans le module.

# Le diagramme de composants

- Exemple:



# Programmation en C++

- Le projet sera implémenté en C++, mais en intégrant les appels systèmes UNIX (conçus pour le C à la base).
- On s'attend à un bon usage de votre part du concept d'encapsulation !
  - Cacher les détails d'initialisation et de paramétrisation des appels-systèmes.
  - Avoir des classes claires et transparentes pour les opérations systèmes.
  - Exemple: accès au réseau. Il convient de créer une belle interface d'accès au système un peu complexe et maladroit de création de connexions réseau sous UNIX. L'exercice n'est pas trivial et doit se faire soigneusement (intervient dans la cotation).

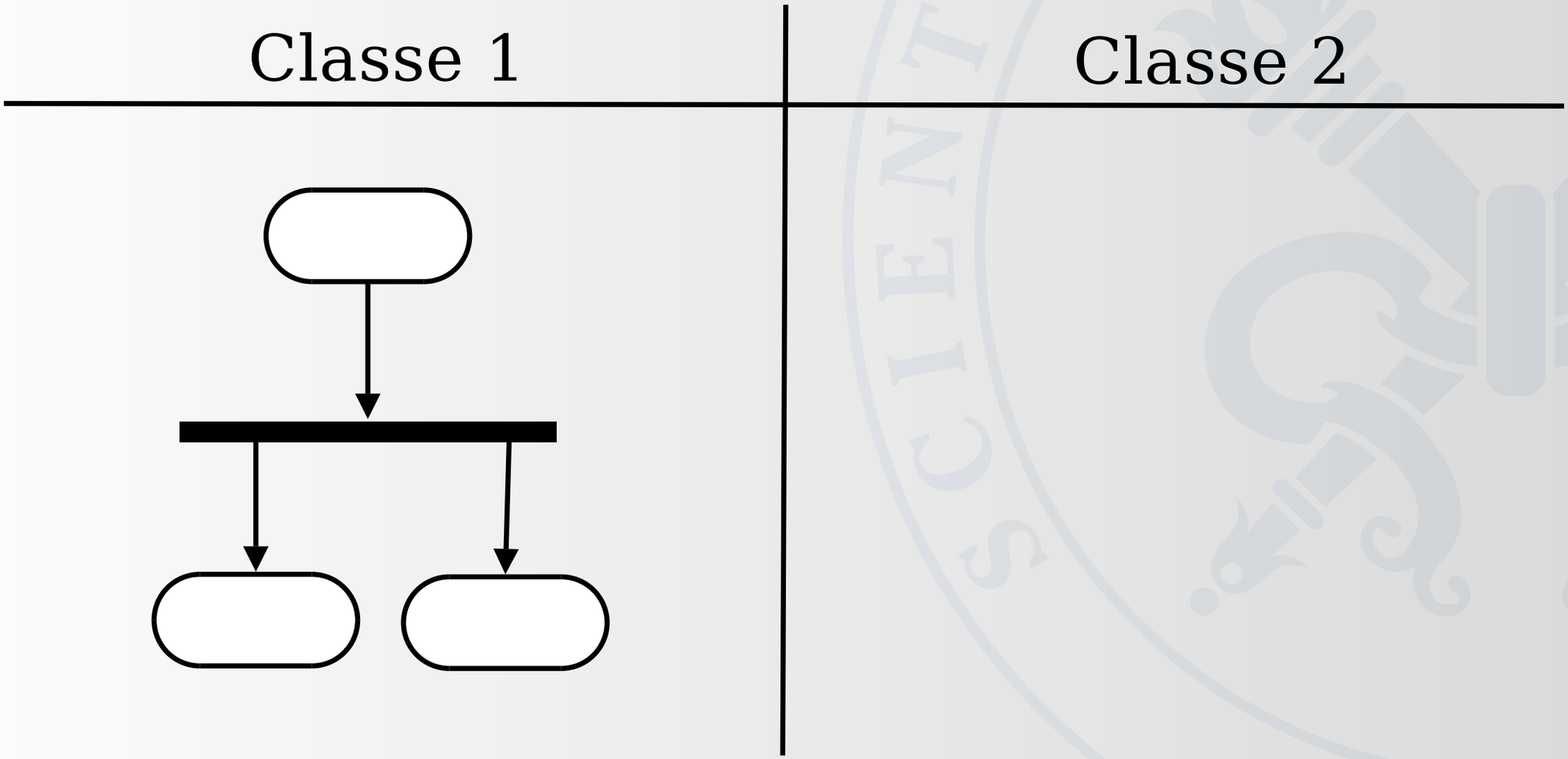
# Le diagramme d'activités

- Lancement de processus concurrents


- Mort d'un processus

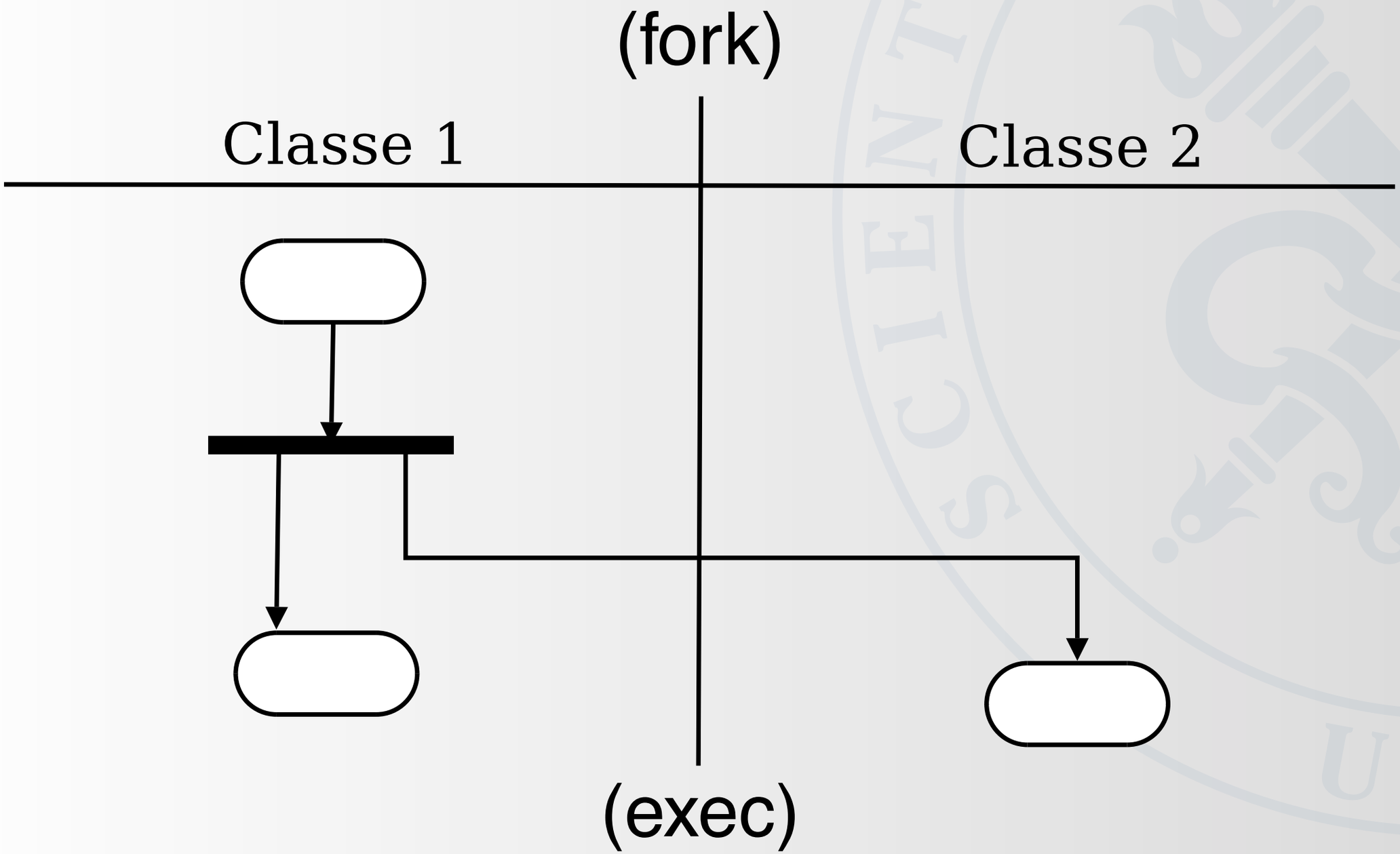
# Le diagramme d'activités

- Appel système fork() ou pthread_create()



# Le diagramme d'activités

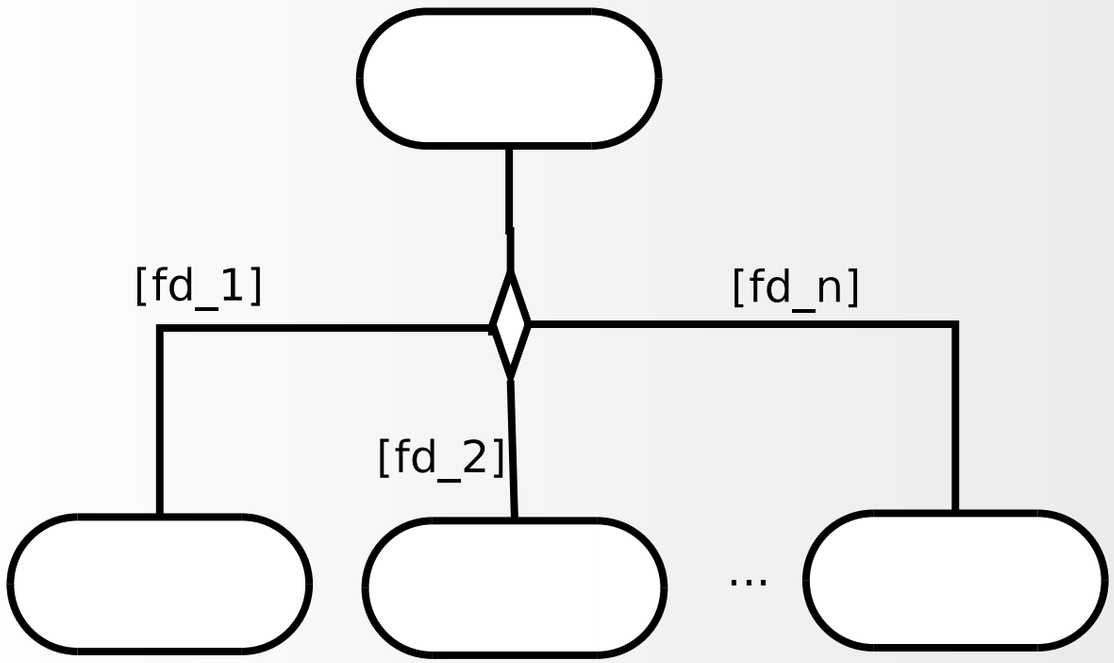
- Appel système fork() + exec()



# Le diagramme d'activités

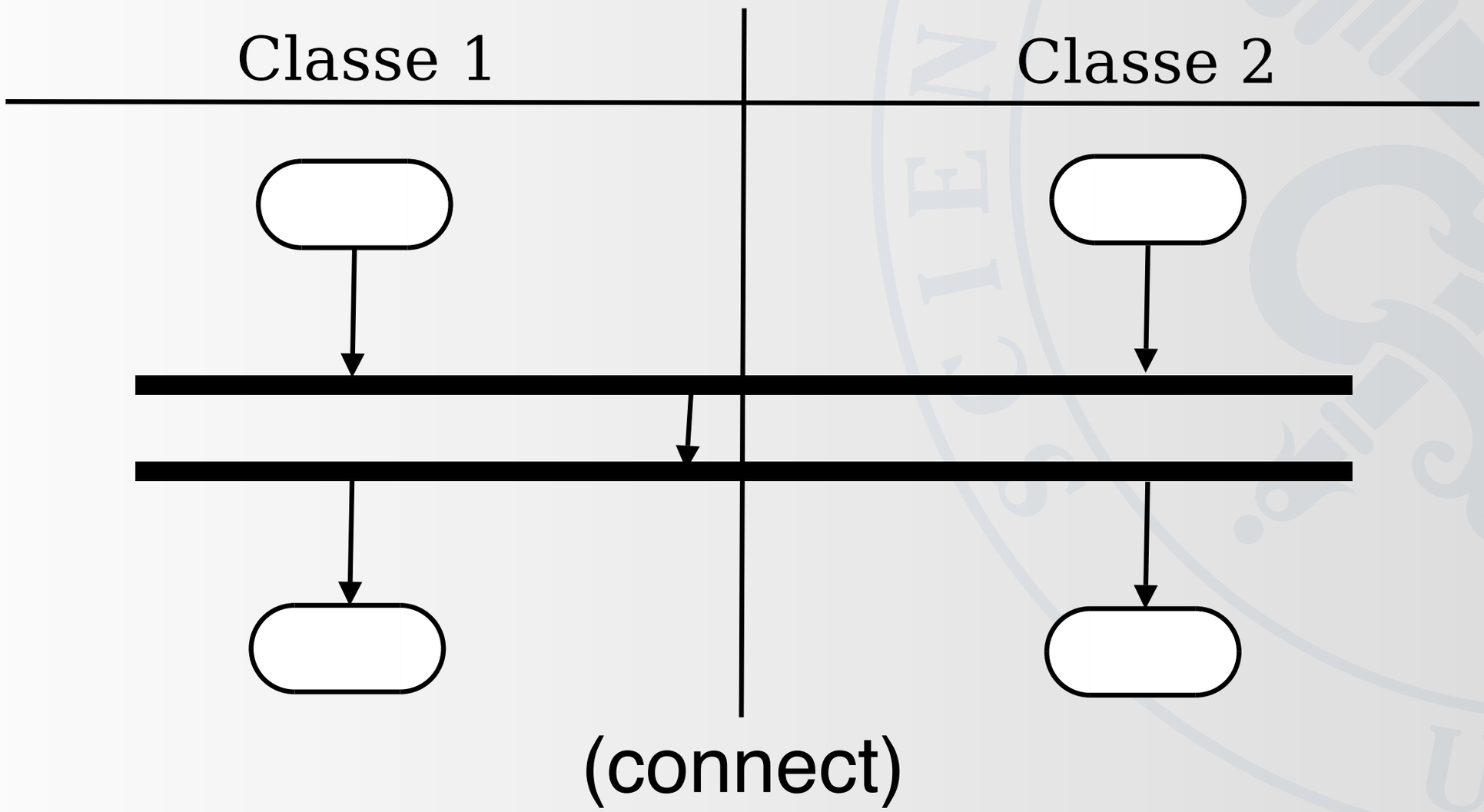
- Appel système select()

| Classe 1 | Classe 2 |
|---|---|



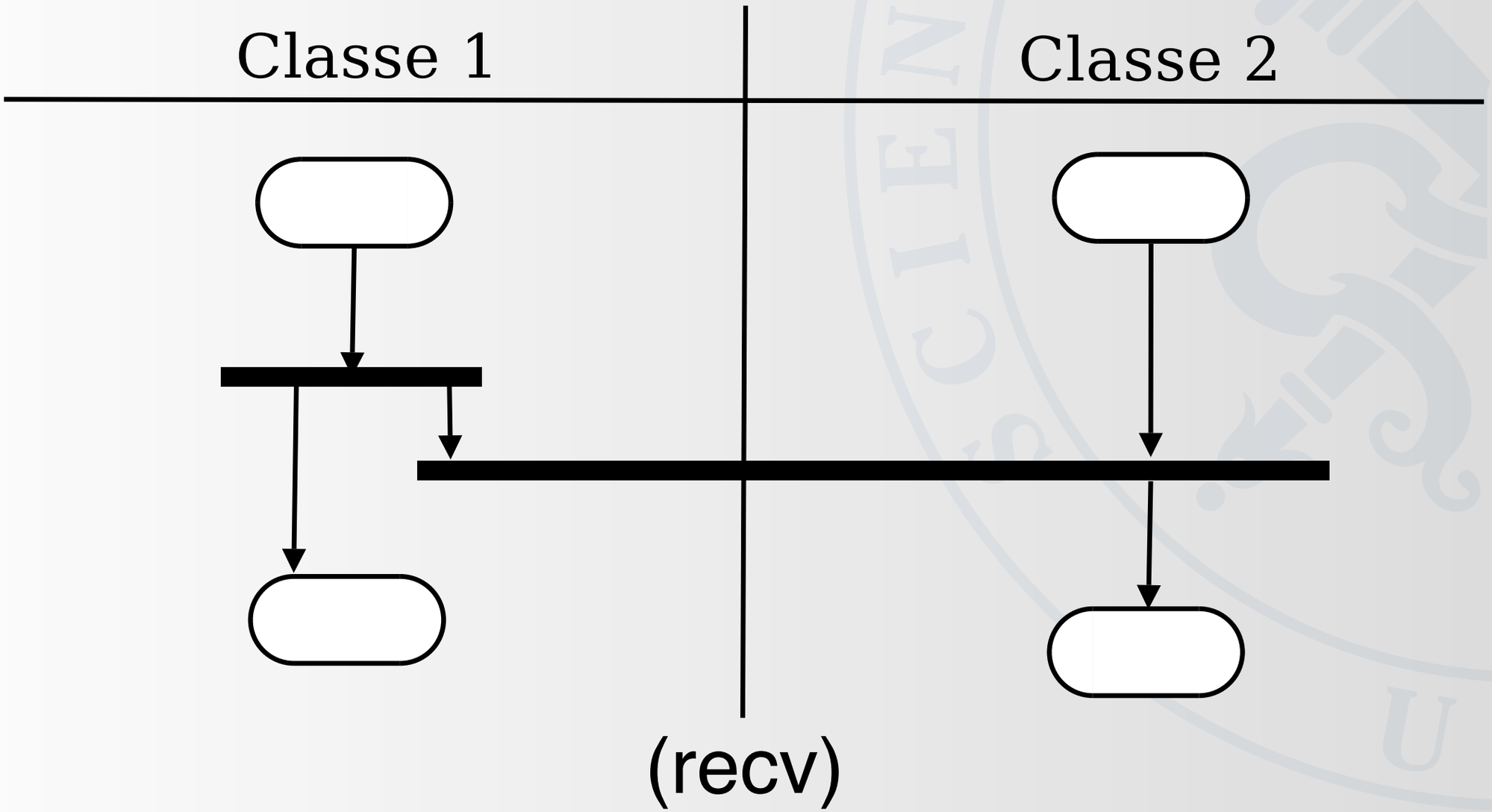
# Le diagramme d'activités

- Appels systèmes accept()-connect()



# Le diagramme d'activités

- Appels systèmes send()-receive() ou read()-write()



# Le diagramme d'activités

- Exercice: modéliser à l'aide d'un diagramme d'activités le modèle client-serveur vu au TP: boucle infinie avec la séquence select-accept-fork-recv-send.

# Le serveur SVN

- Un serveur SVN est mis à votre disposition afin d'archiver et de partager votre code.
- Principe:
  - Vos fichiers sources sont stockés sur le serveur.
  - Après chaque modification locale, vous mettez à jour la version présente sur le serveur (*commit*).
  - Avant de travailler en local, vous récupérez les dernières mises à jour sur le serveur (*update*).
  - SVN gère la fusion des différentes modifications et signale les éventuels conflits.
  - Toutes les versions des fichiers sont archivés, ce qui permet de revenir à une ancienne version

# Le serveur SVN

- La version du projet compilée à la défense sera celle présente sur le serveur.
- Avantage: permet de travailler de manière cohérente à plusieurs personnes sur le même projet sur des machines différentes.
- Chaque groupe dispose d'un mot de passe.

# Le serveur SVN

- Commandes:
  - Initialisation (à faire une seule fois sur chaque machine locale):
    - `svn co svn://lit7.ulb.ac.be/Projets/2007-2008/XXX --username developer`
  - Ajout d'un nouveau fichier:
    - `svn add fichier`
  - Envoi des mises à jour vers le serveur:
    - `svn commit`
  - Mise à jour de la version locale (à faire avant de modifier les fichiers):
    - `svn update`