# Projet C avancé – Master, Workers et Clients

Juan Jose Giraldo Ramos, Sebastian Constain

Licence Informatique L3 – TD2 – TP3

01/12/2025

## Table des matières

1.1 Organisation du code ..... 1
1.1.1Fichiers source principaux ..... 1
1.1.1.1 master.c ..... 1
1.1.1.2 client.c ..... 2
1.1.1.3 client_bis.c ..... 2
1.1.1.4 worker.c ..... 3
1.1.2 Fichiers auxiliaires ..... 3
1.1.2.1 Makefile ..... 3
1.2 Protocoles de communications ..... 4
1.2.1 Protocole client-master ..... 4
1.2.2 Protocole master-client ..... 4
1.2.3 Protocole master-worker ..... 4
1.2.4 Protocole worker-master ..... 5
1.2.5 Protocole worker-worker ..... 5
1.2.6 Protocole semaphores IPC ..... 6
1.2.6.1 Semaphore 1 – Mutex ..... 6
1.2.6.2 Semaphore 2 – Syncronisation ..... 6
1.3 Spécifications importantes ..... 6
1.3.1 Fonctionnalité du projet ..... 6
1.3.2 Précisions concernant le contenu et la structure du projet (par rapport au code fourni) ..... 6

## 1.1 Organisation du code

### 1.1.1Fichiers source principaux

#### *1.1.1.1 master.c*

Rôle Principal : Coordinateur central du système

Fonctionnalités implémentées :

- Création des tubes nommés (mkfifo)
- Création des sémaphores IPC (semget)
- Lancement du premier worker (fork/exec)
- Dans la boucle principale:
  1. Attente d'un client (sémaphore)
  2. Lecture de la commande (tube nommé)
  3. Traitement selon le type de commande:
  - ORDER_STOP: arrêt propre du système
  - ORDER_TEST_PRIME: utilisation du pipeline de Hoare
  - ORDER_COUNT_PRIMES: consultation directe
  - ORDER_MAX_PRIME: consultation directe
  4. Envoi de la réponse
  5. Libération des ressources client

Structures de données importantes :

typedef struct {

char type;  (PRIME (1) ou NOT_PRIME (2))

int value;  (nombre testé)

} Message;

Variables globales pour l'état du système :

static int dernier_envoye = 2;

static int nombre_premiers_trouves = 1;

static int plus_grand_premier = 2;

Points forts à mentionner : Implémentation correcte du crible de Hoare, gestion propre des erreurs (tous les appels système vérifiés), synchronisation client/master via sémaphores IPC, libération complète des ressources à la fin.

#### *1.1.1.2 client.c*

Rôle: Interface utilisateur avec le système master/worker

Protocole d'utilisation:

Usage: ./client [number]

Orders: 1=stop, 2=test_prime, 3=count_primes, 4=max_prime

Fonctionnement interne:

1. Validation des arguments
2. Connexion au master (tubes nommés)
3. Section critique (sémaphore mutex)
4. Envoi de la commande
5. Réception de la réponse
6. Libération (sémaphore de synchronisation)

Points forts à mentionner: Validation robuste des entrées utilisateur, gestion des erreurs de communication, respect de la section critique entre clients concurrents, Interface simple mais complète.

#### *1.1.1.3 client_bis.c*

Rôle : Implémentation locale du crible d'Ératosthène avec threads POSIX

Algorithme implémenté :

1. Création du tableau de booléens (2 à N)
2. Lancement de $\sqrt{N}$ threads (un par diviseur potentiel)
3. Chaque thread marque les multiples de son diviseur
4. Synchronisation via mutex
5. Affichage des résultats

Points forts à mentionner : Implémentation parallèle du crible d'Ératosthène, pas de variables globales (bonne pratique), section critique protégée par mutex, gestion mémoire complète (malloc/free)

#### *1.1.1.4 worker.c*

Rôle : Filtre dans le pipeline de Hoare, responsable d'un nombre premier

Création dynamique de workers : Quand un worker reçoit un nombre non divisible par son prime et qu'il n'a pas de successeur :

1. Créer un nouveau pipe (pipe())

2. Forker un nouveau processus (fork())

3. Exécuter worker avec les nouveaux paramètres (execvp())

Points forts à mentionner : Architecture en cascade conforme au crible de Hoare, création à la demande des workers (économie de ressources), communication unidirectionnelle simple mais efficace, propagation propre du signal d'arrêt.

### 1.1.2 Fichiers auxiliaires

#### *1.1.2.1 Makefile*

Rôle : Automatise la compilation des 4 exécutables avec des Options strictes de compilation (-Wall -Wextra -pedantic) et qui supporte du multi-threading (-pthread). Règle clean pour le nettoyage des fichiers temporaires.

Points à mentionner : Respect des contraintes académiques (options de compilation) imposées par le cahier de charges, production d'exécutables optimisés et sans warnings et gestion automatique des dépendances

## 1.2 Protocoles de communications

### 1.2.1 Protocole client-master

> Chaîne : tube_client_to_master

> Format du message : int ordre[2] = {commande, nombre};

>Taille: 2 * sizeof(int) (8 bytes)

> Commande (ordre[0]) :

1 = Arrêt du master

2 = Tester si un nombre est premier

3 = Nombre de nombres premiers calculés

4 = Plus grand nombre premier trouvé

> Nombre (ordre[1]) :

Pour commande 2 : nombre à tester

Pour autres commandes : 0

> Flux : Client → write(ordre[2]) → tube_client_to_master → Master read(ordre[2])

### 1.2.2 Protocole master-client

> Chaîne : tube_master_to_client

> Format du message : int reponse;

> Taille : sizeof(int) (4 bytes)

> Valeurs :

Pour commande 2 : 1 = premier, 0 = non premier

Pour commande 3 : nombre de primes trouvés

Pour commande 4 : plus grand prime trouvé

Pour commande 1 : 1 = accusé de réception

> Flux : Master → write(reponse) → tube_master_to_client → Client read(reponse)

### 1.2.3 Protocole master-worker

> Chaîne : Pipe anonyme pipe_to_worker[1] --> pipe_to_worker[0]

> Format du message : int valeur;

> Taille : sizeof(int) (4 bytes)

> Valeurs :

n > 0 = nombre à tester

-1 = ordre d'arrêt

> Comportement Hoare : Le master envoie les nombres séquentiellement

> Flux : Master write(valeur) → pipe_to_worker[1] → pipe_to_worker[0] → Worker read(valeur)

### 1.2.4 Protocole worker-master

> Chaîne : Pipe anonyme  pipe_from_worker[1] --> pipe_from_worker[0]

> Format du message :

typedef struct {

  char type;  // PRIME (1) ou NOT_PRIME (2)

  int value;  // nombre testé

} Message;

> Taille : sizeof(Message) (5 bytes)

> Type :

PRIME (1) = nombre est premier

NOT_PRIME (2) = nombre n'est pas premier

> Value : Le nombre qui a été testé

> Flux : Worker write(Message) → pipe_from_worker[1] → pipe_from_worker[0] → Master read(Message)

### 1.2.5 Protocole worker-worker

> Chaîne : Pipes anonymes créés dynamiquement

> Format du message : int n;

> Taille : sizeof(int) (4 bytes)

> Valeurs :

n > 0 = nombre à passer au worker suivant

-1 = propagation d'arrêt

> Creation dynamique : Chaque worker crée le suivant quand nécessaire

> Flux : Worker_i write(n) → newpipe[1] → newpipe[0] → Worker_{i+1} read(n)

### 1.2.6 Protocole semaphores IPC

#### *1.2.6.1 Semaphore 1 – Mutex*

> Clé : ftok(“.”, 1) ou clé fixe

> Valeur initiale : 1 (libre)

> Operations :

P(sem_mutex) = semop(-1) = entrer section critique

V(sem_mutex) = semop(+1) = sortir section critique

> But : Exclusion mutuelle entre clients

#### *1.2.6.2 Semaphore 2 – Syncronisation*

> Clé : ftok(“.”, 2) ou clé fixe

> Valeur initiale : 0 (bloqué)

> Operations :

P(sem_sync) = semop(-1) = master attend fin client

V(sem_sync) = semop(+1) = client signale fin au master

> But : Synchronisation master-client

## 1.3 Spécifications importantes

### 1.3.1 Fonctionnalité du projet

Nous tenons à préciser que le code que nous avons écrit respecte les directives de compilation et ne présente aucune erreur, aucun warning, aucun problème avec le valgrind, ni aucun débordement de mémoire.

Cependant, après la compilation avec « make » et appel au processus maître avec « ./master », lorsque l'on tente d'appeler le client avec « ./client 2 17 » par exemple, le processus se bloque et nous n'avons pas pu en déterminer la cause au niveau du code.

Nous soupçonnons que cela soit dû à un problème avec le sémaphore 2, puisque le master reste bloqué en attente du sémaphore de synchronisation que le client devrait incrémenter, mais nous avons pu déterminer pourquoi cela se produisait.

### 1.3.2 Précisions concernant le contenu et la structure du projet (par rapport au code fourni)

Du côté du code fourni, nous avions du mal à comprendre la logique à suivre. Malgré plusieurs tentatives, nous nous sommes toujours retrouvés dans une impasse. Après avoir examiné attentivement la fiche de projet, qui n'indiquait jamais explicitement que le code fourni devait être suivi à la lettre pour soumettre un projet valide (d’après notre compréhension), nous avons décidé de simplifier la structure de notre code afin de le

rendre plus compréhensible, tout en respectant les objectifs et les exigences définis dans le cahier des charges (structures, bibliothèques, etc).