Université des Sciences et des Technologies d'Oran
Département d'Informatique

TOPKCAJ
Simulation de Roulette Américaine

Rapport de Projet - Systèmes d'Exploitation Avancés

Communication inter-processus (IPC) :
mémoire partagée et échange de messages

• Étude des mécanismes d'IPC permettant aux processus de communiquer et se synchroniser.
• Projet : réaliser une application utilisant mémoire partagée synchronisée (mutex, sémaphores) ou communication par messages/threads.

Réalisé par :
Oussam SALAH
Mouhcine ABDELMOUMENE

Spécialité :
4ème Année Ingénieur - Sécurité Informatique

12 décembre 2025

Table des matières

Introduction

Dans le cadre du module **Système d'exploitation avancé**, il nous a été demandé de concevoir une application mettant en œuvre les mécanismes de Communication Inter-Processus (IPC). L'objectif pédagogique est de manipuler concrètement la mémoire partagée et les primitives de synchronisation dans un environnement UNIX/Linux.

Notre projet, intitulé **TOPCKAJ**, simule une table de **Roulette Américaine** virtuelle. Ce choix s'explique par la nature intrinsèquement concurrente d'un jeu de casino : plusieurs joueurs (bots) doivent interagir simultanément avec une table de jeu (ressource critique) gérée par un croupier (serveur), le tout sous le regard d'un observateur (interface graphique).

Ce système met en jeu plusieurs concepts fondamentaux des systèmes d'exploitation : la création de processus lourds via `fork`, le partage d'un segment mémoire via `shmget`, et la synchronisation des accès concurrents (problème de la section critique) via des sémaphores POSIX utilisés en tant que Mutex.

Ce rapport détaille notre démarche en trois parties. Le premier chapitre expose les concepts théoriques relatifs aux processus et à la synchronisation. Le second chapitre présente les règles de la roulette américaine et leur modélisation mathématique. Enfin, le troisième chapitre décrit l'architecture technique de notre solution, son implémentation en langage C et l'intégration de la librairie graphique Raylib.

Chapitre 1

Concepts Théoriques et Mécanismes IPC

1.1 Le Processus : Entité Dynamique

Un processus est défini comme une entité dynamique représentant un programme en cours d'exécution. Contrairement au programme qui est passif (stocké sur disque), le processus est actif : il possède un compteur ordinal, des registres, une pile et un espace d'adressage propre .

Dans notre projet, nous utilisons l'appel système `fork()` pour créer une hiérarchie de processus. Le processus père (Launcher) engendre des processus fils (Serveur et Interface), et le processus Interface engendre à son tour les Bots. Chaque processus possède son propre PID (Process IDentifier) unique.

Les processus passent par différents états au cours de leur cycle de vie, notamment *Running* (en exécution), *Ready* (prêt) et *Waiting* (en attente d'un événement ou d'une ressource) . Cette gestion des états est cruciale pour l'ordonnancement effectué par le noyau Linux.

1.2 La Communication Inter-Processus (IPC)

Par défaut, les espaces d'adressage des processus sont isolés pour des raisons de sécurité et de stabilité. Pour permettre à nos bots (joueurs) de communiquer avec le croupier (serveur), nous devons utiliser des mécanismes d'IPC (*Inter-Process Communication*).

Nous avons opté pour la **Mémoire Partagée (Shared Memory)** System V. Elle permet à plusieurs processus d'attacher un même segment de mémoire physique à leur espace d'adressage virtuel. C'est la méthode de communication la plus rapide, car les données ne sont pas copiées d'un processus à l'autre, mais accessibles directement.

1.3 La Synchronisation et l'Exclusion Mutuelle

L'accès concurrent à une mémoire partagée introduit le problème de la **Section Critique**. Si deux bots tentent de modifier la banque ou le tableau des mises simultanément, une *Race Condition* (situation de compétition) peut survenir, menant à des données incohérentes.

Pour garantir l'intégrité des données, nous devons assurer l'**Exclusion Mutuelle** : un seul processus peut se trouver en section critique à un instant donné.

1.3.1 Le Sémaphore POSIX

Pour résoudre ce problème, nous utilisons un sémaphore, concept introduit par Dijkstra. Un sémaphore est une variable protégée utilisée pour restreindre l'accès aux ressources partagées.

Dans notre implémentation, nous utilisons un sémaphore binaire (initialisé à 1), fonctionnant comme un **Mutex** (Verrou d'exclusion mutuelle). Nous utilisons l'API POSIX (`<semaphore.h>`) :

- `sem_init` : Initialise le sémaphore.
- `sem_wait` (opération P) : Décrémente le sémaphore. Si la valeur devient négative, le processus est bloqué (mis en file d'attente). C'est l'entrée en section critique.
- `sem_post` (opération V) : Incrémente le sémaphore et réveille un processus en attente si nécessaire. C'est la sortie de section critique.
- `sem_destroy` : Libère les ressources du sémaphore.

Chapitre 2

La Roulette Américaine : Règles et Probabilités

2.1 Présentation et Règles

La roulette est un jeu de hasard emblématique des casinos. Notre projet simule spécifiquement la variante **Américaine**. La différence majeure avec la roulette européenne réside dans la présence d'une case supplémentaire : le double zéro (**00**).

La roue comporte donc 38 numéros :

— Les numéros de 1 à 36 (alternance Rouge/Noir).

— Le 0 et le 00 (Verts).

2.2 Types de Paris Implémentés

Notre simulation gère une large variété de mises, divisées en deux catégories :

2.2.1 Mises Intérieures (Inside Bets)

Ces mises portent sur des numéros spécifiques. Elles offrent des gains élevés mais une probabilité de victoire faible.

— **Plein (Single) :** Mise sur un seul numéro. Gain 35 :1.

— **Cheval (Split) :** Mise sur deux numéros adjacents. Gain 17 :1.

— **Transversale (Street) :** Mise sur une rangée de 3 numéros. Gain 11 :1.

— **Carré (Square) :** Mise sur 4 numéros formant un carré. Gain 8 :1.

— **Sixain (Double Street) :** Mise sur 6 numéros (2 rangées). Gain 5 :1.

2.2.2 Mises Extérieures (Outside Bets)

Ces mises portent sur des propriétés des numéros. Elles offrent une meilleure probabilité de gain mais des rapports plus faibles.

— **Chances Simples (Gain 1 :1) :** Rouge/Noir, Pair/Impair, Manque (1-18)/Passe (19-36).

— **Douzaines et Colonnes (Gain 2 :1) :** Première, deuxième ou troisième douzaine/colonne.

2.3 Modélisation Probabiliste des Bots

Pour rendre la simulation vivante, nos "Bots" ne jouent pas totalement au hasard. Nous avons implémenté une logique de décision probabiliste dans le fichier `players.c`. Chaque bot

génère un nombre aléatoire (`roll`) pour déterminer son type de pari via la formule :

$$\text{score} = 100 - |N - 4| \times 4 + N \times 3.5$$

Cette répartition assure une diversité visuelle sur la table de jeu et permet d'observer différentes stratégies de gains et pertes au fil des tours.

Chapitre 3

Architecture et Implémentation

3.1 Architecture Globale

L'application repose sur une architecture multi-processus centrée autour d'un segment de mémoire partagée.

- **Shared Memory (SHM) :** Contient l'état du jeu, la banque, les paris, et le Mutex.
- **Server (Croupier) :** Écrit l'état (Open/Closed/Results), tire le numéro gagnant, calcule les gains.
- **Bots (Joueurs) :** Lisent l'état, écrivent des paris (`Bet`) quand l'état est `BETS_OPEN`.
- **App (GUI) :** Lit l'état en lecture seule pour l'affichage (Raylib).
- **Launcher :** Processus père qui initialise et orchestre le lancement des modules.

FIGURE 3.1 – Architecture du système TOPCKAJ

3.2 Structure de Données Partagées

Le cœur du système est la structure `SharedResource` définie dans `shared.h`. C'est cette structure qui est mappée en mémoire.

```
typedef struct {
    sem_t mutex;          // S maphore POSIX pour la synchronisation
    int state;            //  tat  du jeu (0=OPEN,1=CLOSED,2=RESULTS)
    int winning_number;   // Num ro gagnant
    Bet bets[MAX_BETS];   // Tableau des paris
    int total_bets;       // Compteur de paris
    int bank;             // Banque commune
} SharedResource;
```

Listing 3.1 – Structure partagée principale

3.3 Synchronisation et Section Critique

L'accès au tableau `bets[]` et à la variable `bank` constitue notre section critique.

3.3.1 Le Serveur (Croupier)

Le serveur gère le cycle de jeu. Lorsqu'il doit changer l'état du jeu ou nettoyer la table, il acquiert le mutex pour garantir qu'aucun bot n'est en train d'écrire. Cycle : *Faites vos jeux* → *Rien ne va plus* (Lock Mutex) → Tirage → Calcul des gains → *Résultats.*

3.3.2 Les Bots (Joueurs)

Les bots sont des processus autonomes. Leur boucle principale vérifie si `state == BETS_OPEN`. Si oui, ils tentent de placer un pari. C'est ici que l'exclusion mutuelle est vitale pour éviter que deux bots n'écrivent dans la même case du tableau `bets` ou ne corrompent le montant de la banque.

```
// Attente du feu vert (P operation)
sem_wait(&shm->mutex);

// DEBUT SECTION CRITIQUE
if (shm->total_bets < MAX_BETS && shm->state == BETS_OPEN) {
    shm->bank -= bet_price;       // Modification banque
    shm->bets[shm->total_bets] = m; // Ajout du pari
    shm->total_bets++;
}
// FIN SECTION CRITIQUE

// Lib ration du verrou (V operation)
sem_post(&shm->mutex);
```

Listing 3.2 – Protection de l'écriture par Mutex (players.c)

3.4 Interface Graphique et Monitoring

L'interface utilisateur est réalisée avec **Raylib**. Elle agit comme un observateur passif. Elle affiche :

— La table de jeu et la position des jetons.
— L'animation de la roue et de la bille.
— Un **panneau de monitoring** en temps réel.

Ce panneau est particulièrement intéressant pour l'analyse système car il affiche l'état du Mutex (Verrouillé/Libre), le PID du processus qui détient le verrou, et un historique (Log) des prises et relâches du sémaphore. Cela permet de visualiser concrètement la "lutte" pour l'accès à la ressource critique.

Conclusion

Le projet **TOPCKAJ** nous a permis de matérialiser les concepts théoriques vus en cours de Systèmes d'Exploitation Avancés. La transition de la théorie (cours sur les sémaphores et la mémoire partagée) à la pratique nous a confrontés aux réalités de la programmation système concurrente.

Nous avons appris à structurer une application multi-processus robuste, où la survie du système ne dépend pas d'un seul processus. L'utilisation des **sémaphores POSIX** s'est révélée indispensable pour garantir la cohérence des données financières (la banque commune) et des données de jeu (les paris). Sans cette synchronisation, nous aurions observé des *Race Conditions* menant à des pertes de paris ou des corruptions de mémoire.

Enfin, l'intégration d'une interface graphique performante (Raylib) communiquant via la mémoire partagée démontre la puissance des IPC pour découpler la logique métier (Serveur/Bots) de la couche de présentation. Ce projet constitue une base solide pour appréhender des problématiques plus complexes comme les *Deadlocks* ou la famine dans les systèmes d'exploitation.

Chapitre 4

Annexes

4.1 Code Source Complet du Projet

Le code source complet de TOPKCAJ est disponible sur GitHub :
Répertoire GitHub :
— https://github.com/Harkaso/TOPKCAJ

4.2 Structure des Fichiers Projet

Fichier	Rôle Principal
launcher.c	Bootstrap + cleanup
app.c	Raylib + fork server/bots
server.c	3 phases
players.c	bots + algorithme paris pondéré
shared.h	SharedResource + enum BetType
Makefile	Compilation multi-fichiers
Doxyfile	Générateur de documentation Doxygen
README.md	Instructions compilation/exécution

TABLE 4.1 – Composition complète du projet GitHub

4.3 Commandes de Compilation et Exécution

```
# Compilation
make

# Exécution par défaut (8 bots, 2000$, 25$/pari)
./launcher

# Exécution personnalisée
./launcher --bots 16 --bank 5000 --bet-price 50
```

4.4 Dépendances Requises

— **Raylib**
— **GCC**
— **Linux x86_64**

Bibliographie

[1] **Cours Advanced Operating Systems (AOS)**. Université USTO-MB. Professeur B. DJELLALI, 2025-2026.

[2] Raylib Technologies. (2025). **Raylib - A simple and easy-to-use library**. https://www.raylib.com/.

[3] Doxygen. (2025). **Doxygen : Documentation automatique pour C/C++**. https://www.doxygen.nl/.

[4] Kerrisk, M. (2010). **The Linux Programming Interface**. No Starch Press. Chapitre 44 : Shared Memory et Sémaphores POSIX.

[5] Wikipedia. **Roulette**. https://en.wikipedia.org/wiki/Roulette.