Sorbonne Université – Polytech Sorbonne

Filière : Electrique et Informatique – Systèmes Embarqués 4

Module : OS USER

Sherlock Holmes Deduction Game

Encadrer par :

- M. Thibaut HILAIRE

Réalisé par :

- M. Wassim GHACHI

Groupe B

Table des matières

Introduction

L'objectif de ce projet est de se familiariser avec les fonctionnalités clés du système d'exploitation (processus, sockets, threads, mutex, pipes) en développant un jeu de déduction réseau en C/SDL2 pour 4 joueurs. Chaque joueur est un client interagissant avec un serveur central pour mener des enquêtes et identifier un coupable parmi plusieurs personnages.

Ce rapport décrit :

- La mise en œuvre de chaque mécanisme OS dans le contexte du projet.
- Les raisons de nos choix techniques et leurs alternatives.
- Les extraits de code significatifs et les bonnes pratiques adoptées.

Processus

1. Définition et cycle de vie général

- **Instance d'exécution :** un processus est un programme chargé en mémoire avec son propre espace d'adressage, piles, tas et table de descripteurs.
- **Création :** généralement via fork() (duplique l'espace mémoire) suivi éventuellement de execve() (remplace l'exécutable et l'image mémoire).
- **Terminaison :** un processus se termine par exit() ou un signal ; son parent récupère son pid et son code de retour via wait().

Etape	Appel système	Effet
Création	fork()	Duplique le processus parent
Remplacement	execve()	Charge un nouveau programme en mémoire
Attente	wait()/waitpid()	Parent récupère le code retour de l'enfant

2. Usages typiques dans les systèmes d'exploitation

- **Isolation :** chaque processus est isolé, évite la corruption mémoire mutuelle.
- **Concurrence :** exécution parallèle sur plusieurs cœurs ou machines.
- **Sécurité :** séparation des privilèges (chroot, containers).
- **Modularité :** découpage d'une application en plusieurs services indépendants.

3. Implémentation dans ce projet

3.1. Lancement manuel des processus

Approche choisie : lancement manuel (ou via script) de 1 serveur et 4 clients

```
# Serveur unique
./server 5000
# Quatre clients indépendants
./sh13 127.0.0.1 5000 127.0.0.1 6000 joueur1
./sh13 127.0.0.1 5000 127.0.0.1 6001 joueur2
./sh13 127.0.0.1 5000 127.0.0.1 6002 joueur3
./sh13 127.0.0.1 5000 127.0.0.1 6003 joueur4
```

Pourquoi pas fork() + exec() ?

- **Complexité :** rediriger les sockets et gérer les signaux SIGCHLD pour reap.
- **Flexibilité :** lancement sur plusieurs machines et ports sans modifier le code.

3.2. Avantages de cette approche

- **Isolation complète** : crash d'un client n'affecte pas les autres.
- **Conformité pédagogique :** chaque exécutable se concentre sur son rôle (serveur vs client).
- **Facilité de débogage :** logs séparés et crash dumps isolés.

3.3. Gestion des ressources OS

- **Espace mémoire :** chaque processus a ses propres variables globales (deck[], tableCartes[][]) et ne partage rien en mémoire directe.
- **Table de descripteurs :** chaque client ouvre son propre socket d'écoute (port client) et ses connexions, isolant les flux réseau.

4. Alternatives et cas d'usage avancés

APPROCHE	DESCRIPTION	AVANTAGES	INCONVENIENTS
MONO-PROCESSUS + THREADS	Un seul binaire, plusieurs threads gérant UI et connexions	Moindre overhead mémoirePartage mémoire direct	Synchronisation massiveRisques de race
FORK() (DUPLIQUER)****	Serveur fork() un client par connexion	Héritage automatique des descripteurs	Gestion des zombies, complexité signal/signaux
CONTAINERS LEGER (DOCKER)	Isolation processus + namespace, déploiement reproductible	Sécurité, ports mappés indépendants	Setup plus lourd, dépendances de l'hôte

D'où, le lancement manuel de processus répond aux objectifs pédagogiques et techniques du projet, en mettant l'accent sur l'IPC via sockets et en évitant la surcharge de la gestion de forks ou de threads pour la logique de séparation.

Sockets TCP

1. Concepts généraux

Socket : point de terminaison d'une communication réseau.

Familles : AF_INET (IPv4), AF_INET6 (IPv6), AF_UNIX (IPC local).

Types :

- SOCK_STREAM (TCP) : connexion orientée flux, garanti et ordonné.
- SOCK_DGRAM (UDP) : datagrammes, non connectés, tolérance à la perte.

Propriétés clés de TCP

- **3-way handshake** : SYN → SYN-ACK → ACK, ~1 RTT (aller-retour).
- **Fiabilité** : retransmissions, contrôle de flux (window).
- **Stream** : pas de démarcation de messages, lecture par octets continus.

2. Implémentation serveur (server.c)

2.1. Création et options

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0) error("ERROR opening socket");
int opt = 1;
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
// désactive le Nagle pour réduire la latence si besoin
//    setsockopt(sockfd,    IPPROTO_TCP,    TCP_NODELAY,    &opt,
sizeof(opt));
```

- **SO_REUSEADDR** : permet de binder rapidement après un kill, évite le TIME_WAIT prolongé.
- **TCP_NODELAY** : désactive Nagle si on veut envoyer de petits messages sans attendre le buffer.

2.2. Liaison et écoute

```
struct sockaddr_in serv_addr = {0};
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;  // toute interface
```

```
serv_addr.sin_port = htons(portno);
bind(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
listen(sockfd, 4 + 1);  // 4 clients + backlog
```

- **INADDR_ANY** : simplifie le déploiement (écoute sur toutes les IP locales).
- **Backlog** fixé à nbClients + marge pour s'assurer qu'on ne perd pas de connexions entrantes.

2.3. Acceptation et gestion des connexions

```
while (1) {
  struct sockaddr_in cli_addr;
  socklen_t clilen = sizeof(cli_addr);
  int newsockfd = accept(sockfd, (struct sockaddr*)&cli_addr, &clilen);
  if (newsockfd < 0) error("ERROR on accept");
  // lecture synchrone : read(newsockfd, buffer, len);
  // stocker client IP/port et newsockfd pour reply
  close(newsockfd);
}
```

- **Mode bloquant** : simple, le serveur fait du pipeline séquentiel.
- **Scalabilité limitée** : jusqu'à 4 clients, suffisant pour un TP. Pour 1000+, envisager multi-thread ou select()/epoll().

3. Implémentation client (sh13.c)

3.1. Création et connexion rapide

```
void sendMessageToServer(const char* ip, int port, const char* mess) {
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in serv_addr;
    serv_addr.sin_family = AF_INET;
    inet_pton(AF_INET, ip, &serv_addr.sin_addr);
    serv_addr.sin_port = htons(port);
```

```
    connect(sockfd,        (struct        sockaddr*)&serv_addr,
sizeof(serv_addr));
    write(sockfd, mess, strlen(mess));
    close(sockfd);
}
```

- **Éphémère** : chaque clic utilisateur crée/ferme le socket, évite la gestion d'état.
- **Temps de connexion** : ~1 à 3 ms en local, acceptable pour une action de jeu.

3.2. Lecture des messages

Le client écoute sur un port client en parallèle (thread), avec accept() et read().

Non-bloquant UI : le thread réseau stocke le message avant que la boucle SDL traite l'événement.

En résumé, Serveur crée un socket écoutant (AF_INET, SOCK_STREAM), bind(), listen(), accept(). Les clients utilisent socket(), connect() pour envoyer des commandes au serveur. Les messages textuels prefixés (C, D, V, M, O, S, G) définissent le protocole de jeu.

Ce choix est expliqué par le fait que les sockets **bloquants** pour ont plus de simplicité, car le serveur gère les connexions dans un seul thread.

Threads

1. Concepts généraux

Définition : un thread (ou fil d'exécution) est la plus petite unité de traitement gérée par le système d'exploitation, partageant le même espace mémoire que son processus.

Cycle de vie :

1. **Création** via pthread_create() : alloue une pile pour le thread.
2. **Exécution** concurrente dans le même address space.
3. **Synchronisation** correspond à pthread_join() ou pthread_detach().
4. **Terminaison** par retour de la fonction ou pthread_exit().

Usages typiques :

- **Réactivité UI** : traiter I/O et affichage en parallèle.
- **Travail en parallèle** : découper un calcul intensif en sous-parties.
- **Serveur multi-threads** : gérer plusieurs connexions simultanément.

2. Implémentation dans le projet

1. Création du thread réseau

```
pthread_t thread_serveur_tcp_id;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

// Dans main() du client
pthread_create(&thread_serveur_tcp_id,   NULL,   fn_serveur_tcp,
NULL);
```

Détachement implicite : on ne fait pas pthread_join(), le thread fonctionne tant que le processus client est vivant.

2. Fonction fn_serveur_tcp

```
void *fn_serveur_tcp(void *arg) {
    int sockfd = socket(...);
    bind(sockfd, ...);
```

```
    listen(sockfd, 1);
    while (!quit) {
        int newsockfd = accept(sockfd, ...);
        char buffer[256];
        int n = read(newsockfd, buffer, sizeof(buffer));
        pthread_mutex_lock(&mutex);
        strcpy(gbuffer, buffer);
        synchro = 1;
        pthread_mutex_unlock(&mutex);
        close(newsockfd);
    }
    return NULL;
}
```

- **Rôle** : réception des messages du serveur sans bloquer l'UI.
- **Communication** : met à jour gbuffer protégé par un mutex, signale synchro.

3. Traitement en boucle principale SDL

```
if (synchro) {
    pthread_mutex_lock(&mutex);
    processMessage(gbuffer);
    synchro = 0;
    pthread_mutex_unlock(&mutex);
}
```

- **Effet** : la boucle SDL (SDL_PollEvent) reste non bloquante.
- **Pattern Producteur/Consommateur** : thread prod les messages, UI thread consomme.

3. Justifications et alternatives

CHOIX	POURQUOI
1 THREAD RESEAU PAR CLIENT	Simplicité, UI fluide, découplage I/O ↔ affichage
MUTEX POUR GBUFFER	Garantir atomicité, éviter corruption pendant strcpy()
THREAD DETACHE (DETACH)	Pas de join, pas de supervision de la fin du thread

Synchronisation et mutex

1. Concepts généraux

Définition : un mutex (mutual exclusion) est un objet de synchronisation qui garantit qu'une seule section critique est exécutée à un moment donné parmi tous les threads d'un même processus.

Utilisations courantes :

Protéger l'accès à des données partagées (variables globales, structures) pour éviter la corruption.

Assurer l'exclusion mutuelle lors de l'entrée dans une section critique.

Construire des primitives plus complexes : sémaphores, conditions variables.

2. Mise en place dans le projet

Initialisation :

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Protection du buffer réseau : le thread fn_serveur_tcp et la boucle SDL principale partagent le buffer global gbuffer et le flag synchro.

- **Producteur (thread réseau) :**

```
pthread_mutex_lock(&mutex);
strcpy(gbuffer, buffer_from_socket);
synchro = 1;
pthread_mutex_unlock(&mutex);
```

- **Consommateur (UI thread)** :

```
if (synchro) {
    pthread_mutex_lock(&mutex);
    processMessage(gbuffer);
    synchro = 0;
    pthread_mutex_unlock(&mutex);
}
```

3. Pourquoi ce choix :

Atomicité : garantit que gbuffer n'est ni lu ni écrit simultanément par deux threads.

Prévention de corruption : évite les trames partielles ou corrompues lors du strcpy() concurrent.

Préparation à l'évolution : si on ajoute d'autres threads consommateurs ou producteurs, le mutex central permet de se prémunir des courses.

Comparatif approfondi des choix OS

Pour chaque besoin principal (isolation, communication, concurrence, synchronisation), nous avons comparé plusieurs mécanismes :

Besoin	**Mécanisme utilisé**	**Avantages dans ce projet**	**Alternatives potentielles**	**Justification du choix**
Isolation	Processus séparés	Crash d'un client n'impacte pas le serveur ni les autres clients. Logs et dumps isolés.	Mono-processus + threads	Simplicité de déploiement sur plusieurs machines, évite la gestion de fork/exec dans le code.
Communication	Sockets TCP blocants (connect/close à chaque message)	Fiabilité, protocole clair (C/D/V/M/O/S/G), simplicité d'implémentation.	Connexion persistante, UDP ou I/O multiplexé (select/epoll)	Peu de clients (<5), latence de connexion marginale, évite la complexité de gestion d'états de socket.
Concurrence UI/Réseau	Thread réseau par client	Interface SDL reste fluide (~60 FPS), découplage net de l'I/O réseau.	Boucle unique avec select(), event-driven (libuv)	SDL ne s'intègre pas bien avec select(), threads offrent une implémentation plus directe et pédagogique.
Protection mémoire	Mutex POSIX (lock/unlock)	Protège gbuffer et synchro contre les races, overhead négligeable.	Spinlock, rwlock, atomics C11	Mutex POSIX suffisant pour une seule section critique, API standard vue en TP.
Scalabilité I/O	Non utilisé (pipes)	Overhead réseau géré par TCP, pas de besoin d'IPC local.	Pipes + multiplexage I/O, shared memory, message queues	Pipes auraient ajouté de la complexité sans bénéfice significatif.

Synthèse : chaque mécanisme a été retenu pour son équilibre entre **simplicité, robustesse** et **adéquation pédagogique.** Ce comparatif sert de base pour envisager ultérieurement des évolutions (échelles supérieures, protocoles plus légers, architectures distribuées).

Conclusion

À travers ce projet, nous avons transposé des concepts fondamentaux du système d'exploitation dans un contexte applicatif complet : la création d'un jeu de déduction réseau.

- **Isolation et modularité** : la séparation en processus distincts pour le serveur et les clients garantit une robustesse à toute épreuve, chaque composant pouvant planter ou être redémarré indépendamment.
- **Communications fiables** : l'emploi de sockets TCP, configurées avec SO_REUSEADDR et optimisables via TCP_NODELAY, nous a permis de définir un protocole clair et résilient, assurant l'intégrité et la séquence des échanges.
- **Réactivité utilisateur** : en déléguant l'E/S réseau à un thread secondaire, la boucle SDL principale reste non bloquante, offrant une interface fluide et respectant les bonnes pratiques vues en TP.
- **Sécurisation des données partagées** : l'utilisation de mutex POSIX pour protéger le buffer global et le flag de synchronisation illustre l'importance de l'exclusion mutuelle pour prévenir les corruptions lors des accès concurrents.
- **Choix pédagogiques** : chaque décision (connexion éphémère, mode bloquant, lancement manuel des processus) a été guidée par le souci de clarté, de facilité de débogage et d'adéquation aux objectifs d'apprentissage.

En définitive, ce projet constitue un terrain d'expérimentation riche pour maîtriser les primitives OS essentielles, tout en offrant une base modulaire et extensible pour des développements ultérieurs en réseau distribué.