Mars Lander RT

Rapport Technique d'Implémentation Temps Réel

Architecture Multi-Processus sous Linux

Thomas Lemoine

5 décembre 2025

Résumé

Ce document technique présente la conception, l'implémentation et la validation du projet **Mars Lander RT**. Il s'agit d'un simulateur de vol spatial critique fonctionnant en temps réel sur un système d'exploitation Linux standard.

L'objectif est de démontrer la maîtrise des mécanismes de communication inter-processus (IPC System V), de synchronisation (Sémaphores/Mutex) et d'ordonnancement événementiel (Timers). Le système simule la dynamique de chute d'un module martien et son asservissement par un calculateur de guidage autonome, le tout supervisé par une interface de contrôle. Ce rapport détaille les choix architecturaux, les implémentations techniques et les validations expérimentales, fournissant une base complète pour la reproduction et la compréhension du système.

Mots-clés : Temps Réel, Linux, IPC, Mémoire Partagée, Sémaphores, Asservissement PID, C.

Table des matières

Chapitre 1

Introduction et Contexte

1.1 Le Défi de l'Atterrissage sur Mars : Les "7 Minutes de Terreur"

L'atterrissage d'une sonde sur la planète Mars est l'une des phases les plus critiques et périlleuses d'une mission interplanétaire. Surnommée les "7 minutes de terreur" par les ingénieurs de la NASA, cette séquence d'Entry, Descent and Landing (EDL) est entièrement automatisée. La distance entre la Terre et Mars engendre un délai de communication qui peut atteindre 12 minutes dans un sens, rendant tout pilotage manuel depuis la Terre impossible. Le module d'atterrissage, ou *Lander*, doit donc prendre une série de décisions de vie ou de mort en toute autonomie, sans aucune aide extérieure.

Ces décisions reposent sur l'analyse en temps réel de données capteurs (altitude, vitesse, accélération) et l'ajustement constant de la traînée et de la poussée des moteurs pour freiner la descente et atteindre la surface à une vitesse nulle ou quasi-nulle. La moindre défaillance logicielle ou matérielle durant cette fenêtre critique condamne la mission.

1.2 Problématique : La Nécessité d'un Système de Guidage Autonome et Temps Réel

La problématique centrale de ce projet est la reproduction d'un tel système de guidage critique. Un simulateur fiable doit respecter deux contraintes fondamentales :

- **L'Autonomie :** Le système de guidage doit fonctionner de manière indépendante, en se basant sur un modèle physique et une loi de commande.
- **Le Temps Réel :** Les calculs et les actions doivent être effectués dans des échéances temporelles strictes. La physique de la chute ne s'arrête pas si le processeur est occupé. Un retard de quelques millisecondes peut entraîner une déviation de trajectoire irrécupérable.

Pour répondre à ces exigences, il est crucial de concevoir une architecture logicielle robuste, déterministe et capable de gérer la concurrence entre plusieurs tâches de natures différentes (simulation physique, calcul de guidage, interaction utilisateur).

1.3 Objectifs du Projet Mars Lander RT

Ce projet vise à développer un simulateur d'atterrissage martien qui sert à la fois de démonstration technique et de support pédagogique.

1.3.1 Objectif Technique : Simuler un Système Critique

L'objectif principal est de concevoir et d'implémenter une architecture logicielle multi-processus capable de :

- Simuler la physique du module (gravité, traînée, inertie) à une fréquence élevée ($10Hz$).
- Calculer la commande moteur via une boucle de régulation indépendante à une fréquence plus basse ($2Hz$), imitant le temps de calcul d'un système réel.
- Permettre l'injection de perturbations (vent) en temps réel par un opérateur pour tester la robustesse.
- Garantir l'intégrité et la cohérence des données échangées entre ces sous-systèmes asynchrones.

1.3.2 Objectif Pédagogique : Maîtriser les IPC et le Temps Réel sous Linux

Ce rapport a pour vocation d'être un guide complet. À sa lecture, le lecteur doit être capable de :

- Comprendre les enjeux de la programmation système concurrente et temps réel.
- Maîtriser les mécanismes d'IPC System V (Mémoire Partagée, Sémaphores).
- Mettre en œuvre un ordonnancement événementiel basé sur les timers et les signaux.
- Refaire l'intégralité du code présenté, en comprenant la justification de chaque choix technique.

1.4 Structure du Document

Pour atteindre ces objectifs, ce rapport est structuré en quatre parties. La **Partie I** présentera l'architecture globale du système et les choix qui la sous-tendent. La **Partie II** disséquera en détail chaque mécanisme système fondamental (mémoire partagée, sémaphores, timers). La **Partie III** se concentrera sur la modélisation physique et la loi de commande. Enfin, la **Partie IV** présentera les résultats de la validation et conclura sur les acquis et les perspectives.

Chapitre 2

Architecture Globale du Système

Après avoir défini le contexte et les objectifs, ce chapitre expose l'architecture logicielle retenue. Nous commencerons par analyser les contraintes qui pèsent sur le système, puis nous justifierons le choix d'une architecture multi-processus avant de présenter la topologie retenue et ses composants.

2.1 Analyse des Contraintes : Temps Réel, Concurrence, Robustesse

La conception d'un simulateur d'atterrissage critique est guidée par trois contraintes fondamentales qui déterminent l'ensemble des choix techniques.

2.1.1 Temps Réel (Soft Real-Time)

Le système doit respecter des échéances temporelles strictes. La simulation physique évolue en continu, et le calculateur de guidage doit produire une commande à une fréquence suffisamment élevée pour stabiliser la trajectoire. Un retard ou une gigue (jitter) excessive dans l'exécution de ces tâches rendrait la simulation irréaliste et le contrôle inefficace. Nous visons un temps réel "souple" (soft), où le respect des échéances est crucial en moyenne, mais une micro-seconde de retard occasionnelle est tolérable, contrairement à un temps réel "dur" (hard) où une seule échéance manquée peut être catastrophique.

2.1.2 Concurrence

Le système doit gérer simultanément plusieurs tâches de natures différentes :

- **Calcul Physique :** Une tâche rapide et cyclique (ex : toutes les 100 ms) qui met à jour l'état du Lander.
- **Calcul de Guidage :** Une tâche plus lente (ex : toutes les 500 ms) qui analyse l'état et décide de l'action.
- **Interaction Utilisateur :** Une tâche asynchrone qui permet d'injecter des perturbations (vent) à tout moment.

Ces tâches s'exécutent en parallèle et partagent des données (vitesse, altitude, poussée), ce qui impose une gestion rigoureuse des accès concurrents.

2.1.3 Robustesse

Le crash d'un sous-système ne doit pas entraîner la défaillance de l'ensemble du système. Par exemple, si l'interface utilisateur qui permet d'injecter du venant plante, le module de guidage et la simulation physique doivent continuer de fonctionner pour permettre un atterrissage nominal. Cette isolation est primordiale dans les systèmes embarqués critiques.

2.2 Le Choix de l'Architecture : Multi-Processus vs Multi-Thread

Pour répondre à ces contraintes sous Linux, deux approches concurrentes principales sont possibles : les threads ou les processus.

2.2.1 Tableau Comparatif Détaillé

Le tableau 2.1 détaille les différences fondamentales entre ces deux approches.

Critère	**Multi-Thread**	**Multi-Processus (Choisi)**
Espace Mémoire	Espace d'adressage partagé par défaut. L'accès aux variables globales est immédiat mais risqué.	Espace d'adressage virtuel distinct et isolé. Le partage doit être explicite.
Isolation / Robustesse	Faible. Une erreur de segmentation (segfault) dans un thread entraîne la terminaison de l'ensemble du processus et de tous ses threads.	Forte. Un crash est confiné au processus concerné. Les autres processus continuent de s'exécuter normalement.
Communication	Simple et rapide via des variables globales (nécessite des mécanismes de synchronisation comme les mutex).	Plus complexe mais explicite via des mécanismes d'IPC (Inter-Process Communication) comme les pipes, les sockets, ou la mémoire partagée.
Création	Léger et rapide (simple duplication de la pile et des registres).	Plus lourd (duplication complète de l'espace d'adressage virtuel par le noyau).

TABLE 2.1 – Comparaison des architectures concurrentes pour un système critique

2.2.2 Justification du Choix Multi-Processus

Bien que l'approche multi-thread puisse sembler plus performante en termes de rapidité de création et de communication, le critère de la **robustesse** est prépondérant pour notre

projet. La garantie qu'une défaillance dans le module d'interface (Environnement) ne puisse pas corrompre la mémoire ou faire planter le calculateur de guidage (Guidage) ou la simulation physique (Physique) est un impératif. L'architecture multi-processus, en forçant un cloisonnement mémoire strict, répond directement à cette exigence. La complexité ajoutée par l'utilisation des IPC est un prix à payer pour la sécurité et la fiabilité du système.

2.3 Vue d'Ensemble : Topologie en Étoile autour d'un Bus de Données

Pour permettre la communication entre ces processus isolés, nous avons adopté une topologie en étoile. Tous les modules communiquent exclusivement via un point central : un **Bus de Données**.

Ce bus est implémenté à l'aide de segments de **Mémoire Partagée** (Shared Memory). Chaque grandeur physique importante (vitesse, altitude, poussée, vent) a son propre segment de mémoire. Cette approche présente deux avantages :

— **Performance :** La mémoire partagée est le mécanisme d'IPC le plus rapide sous Linux (accès direct à la RAM, sans copie de données).

— **Granularité de la protection :** Chaque segment peut être protégé par son propre sémaphore (Mutex), minimisant les temps d'attente et les risques d'interblocage (deadlock).

2.4 Diagramme d'Architecture Détaillé

La figure 2.1 illustre l'architecture complète et les interactions entre les différents modules.

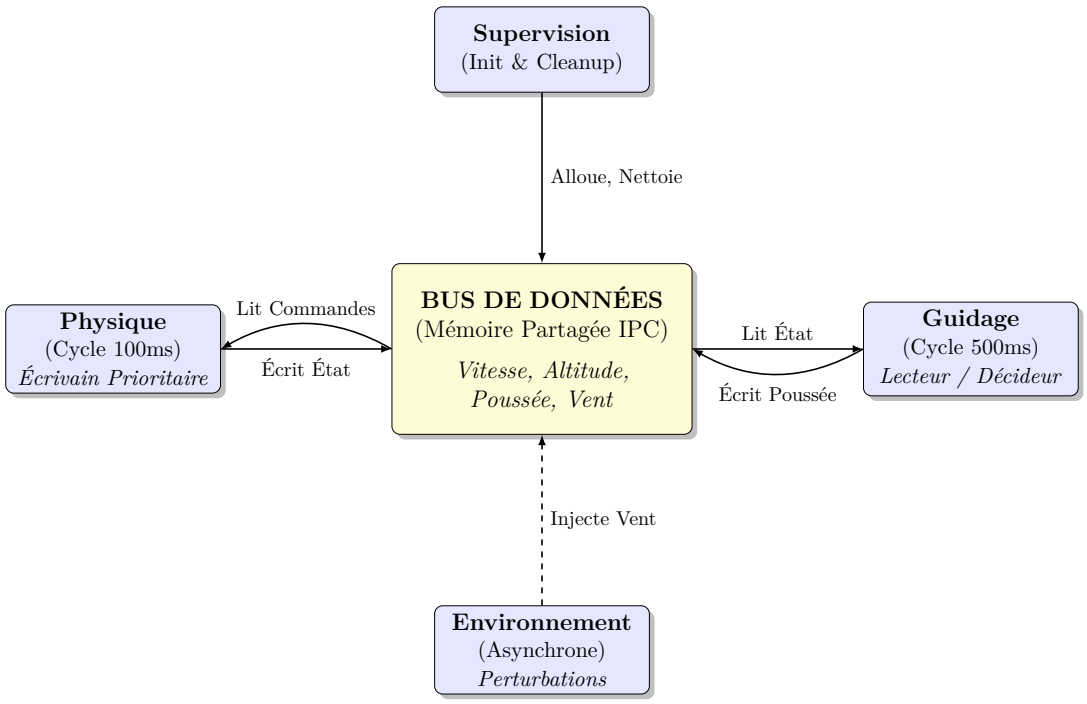

FIGURE 2.1 – Architecture logicielle distribuée du simulateur Mars Lander RT

2.4.1 Description des Rôles des Modules

1. **Supervision :** C'est le processus "parent" ou "chef d'orchestre". Il est responsable de la création et de la destruction de toutes les ressources IPC (mémoire partagée, sémaphores). Il synchronise également le démarrage de tous les autres processus via un mécanisme de barrière (rendez-vous) et assure le nettoyage propre en fin de mission.
2. **Physique (Lander) :** Ce processus simule les lois de la physique ($\sum \vec{F} = m\vec{a}$). Il s'exécute à haute fréquence ($10Hz$) pour garantir la fluidité de la simulation. Il lit la poussée moteur et la vitesse du vent, puis écrit la nouvelle vitesse et altitude.
3. **Guidage :** Ce processus simule l'ordinateur de bord. Il fonctionne à une fréquence plus basse ($2Hz$) pour modéliser le temps de calcul et de réaction d'un système réel. Il lit l'état du Lander (vitesse, altitude) et calcule la commande de poussée moteur via un correcteur PI.
4. **Environnement :** Ce processus représente l'opérateur ou le milieu extérieur. Il fonctionne de manière asynchrone, permettant à un utilisateur d'injecter des perturbations (comme une rafale de vent) à tout moment pour tester la robustesse de l'asservissement.

2.4.2 Flux de Données et de Contrôle

Les flèches sur la figure 2.1 indiquent le sens des échanges :
— La **Physique** est un *producteur* principal d'informations (vitesse, altitude).
— Le **Guidage** est un *consommateur* de ces informations et un *producteur* de commande (poussée).
— L'**Environnement** est un *producteur* de perturbations (vent).
— La **Supervision** est un *gestionnaire* de ressources, n'intervenant que dans la phase d'initialisation et de terminaison.

Toutes ces interactions transitent par le bus de données central, garantissant que chaque module ne communique que via une interface bien définie et sécurisée.

Chapitre 3

Le Bus de Données : Mémoire Partagée (IPC System V)

La mémoire partagée (Shared Memory ou SHM) est le mécanisme fondamental qui permet à nos processus isolés de communiquer. Ce chapitre détaille son principe de fonctionnement, l'API System V pour la manipuler, et son application concrète dans le projet Mars Lander RT.

3.1 Concept : Communication "Zero-Copy"

Contrairement à d'autres mécanismes d'IPC comme les *Pipes* ou les *Sockets*, la mémoire partagée ne nécessite pas de copie de données entre les espaces mémoire des processus. Avec un pipe, par exemple, un processus écrit dans un buffer du noyau, et le processus lecteur doit lire depuis ce buffer, ce qui implique deux opérations de copie.

La mémoire partagée fonctionne différemment : le noyau alloue une zone de mémoire physique (RAM) et la "mappe" dans l'espace d'adressage virtuel de chaque processus qui y accède. Une fois cette projection effectuée, accéder à une variable partagée est aussi rapide que d'accéder à une variable locale. Il n'y a qu'une seule copie des données en mémoire. C'est la méthode de communication la plus rapide possible sous Linux.

Performance

L'accès à la mémoire partagée est un accès mémoire direct. Il n'y a pas d'appel système pour lire ou écrire une donnée une fois le segment attaché. C'est ce qui la rend indispensable pour les applications à haute performance comme nos simulations.

3.2 L'API System V pour la Mémoire Partagée

L'utilisation de la mémoire partagée System V suit un cycle de vie rigoureux en quatre étapes. Ne pas respecter cet ordre est une source fréquente d'erreurs de segmentation (*Segfault*).

3.2.1 Étape 1 : Allocation et Récupération (`shmget`)

La fonction `shmget` (Shared Memory Get) est la porte d'entrée. Elle ne crée pas une variable, mais demande au noyau d'allouer un segment de mémoire et d'assigner un iden-

tifiant à ce segment.

```
int shmget(key_t key, size_t size, int shmflg);
```

Listing 3.1 – Prototype de la fonction `shmget`

Détail des arguments :

- **key_t key :** C'est un identifiant unique (un entier) connu de tous les processus qui veulent partager ce segment. C'est l'équivalent d'un numéro de téléphone dans un annuaire. On utilise généralement une valeur entière fixe (ex : `1001`) ou une clé générée avec `ftok`.
- **size_t size :** La taille en octets du segment à allouer. Dans notre projet, nous utilisons `sizeof(double)` (8 octets) car nous stockons des grandeurs physiques précises.
- **int shmflg :** Un ensemble de drapeaux qui contrôlent la création et les permissions.
  - `IPC_CREAT` : Crée le segment s'il n'existe pas. Si le segment existe déjà, cette fonction retourne simplement son identifiant.
  - `0666` : Permissions d'accès en octal. C'est un point crucial souvent oublié.

Les Permissions Octales (0666)

Sous Linux, si vous oubliez les droits (par exemple, en n'utilisant que `IPC_CREAT`), le segment sera créé mais personne (pas même le propriétaire) ne pourra le lire ou l'écrire.

- **6** (4+2) = Lecture (4) + Écriture (2).
- **666** = Le Propriétaire, le Groupe et les Autres ont tous les droits en lecture et écriture.

3.2.2 Étape 2 : Attachement (`shmat`)

C'est l'étape la plus critique. L'ID renvoyé par `shmget` est un identifiant interne au noyau, pas une adresse mémoire utilisable. Pour pouvoir lire ou écrire dans le segment, un processus doit l'attacher à son propre espace d'adressage virtuel.

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Listing 3.2 – Prototype de la fonction `shmat`

- **int shmid :** L'identifiant obtenu avec `shmget`.
- **const void *shmaddr :** L'adresse mémoire où le segment doit être attaché. En passant `NULL`, on demande au noyau de choisir une adresse libre et alignée, ce qui est la pratique recommandée et la plus portable.
- **Retour :** La fonction retourne un pointeur générique `void*` vers le début du segment. Il est impératif de le "caster" vers le type de données approprié (ex : `double*`).

Une fois cette ligne exécutée, écrire `*p_vitesse = 10.0` modifie instantanément la valeur visible par tous les autres processus connectés à ce même segment.

3.2.3 Étape 3 : Détachement (`shmdt`)

Lorsqu'un processus a terminé d'utiliser la mémoire partagée, il doit s'en détacher. Cette opération ne détruit pas le segment, elle le retire simplement de l'espace d'adressage du processus. Le pointeur devient invalide.

```c
int shmdt(const void *shmaddr);
```

Listing 3.3 – Détachement d'un segment de mémoire partagée

3.2.4 Étape 4 : Contrôle et Destruction (`shmctl`)

Contrairement à la mémoire de la pile (stack) ou du tas (heap) qui est automatiquement libérée à la fin d'un programme, **la mémoire partagée System V est persistante**. Elle survit à la fin du processus qui l'a créée.

C'est le rôle exclusif du processus **Supervision** de nettoyer le système à la fin de la mission en détruisant les segments.

```c
int shmctl(int shmid, int IPC_RMID, NULL);
```

Listing 3.4 – Destruction d'un segment de mémoire partagée

Commandes Utiles : `ipcs` et `ipcrm`

Pendant le développement, si le programme plante avant le nettoyage, les segments de mémoire partagée restent alloués dans le système. La commande `ipcs -m` permet de lister tous les segments actifs (et de repérer les orphelins). La commande `ipcrm -m [shmid]` permet de supprimer manuellement un segment spécifique.

3.3 Implémentation dans le Projet

Dans *Mars Lander RT*, nous utilisons 5 segments distincts (Vitesse, Altitude, Poussée, Vent, Cible) plutôt qu'une seule grande structure contenant toutes ces données. Ce choix délibéré permet une granularité fine des protections par Mutex (voir chapitre suivant), réduisant ainsi les contentions et les risques d'interblocage.

3.3.1 Code Commenté : Allocation et Attachement (Supervision)

Le processus `supervision.c` est le créateur. Il alloue chaque segment et y attache un pointeur pour l'initialiser.

```c
// Clés pour les segments de mémoire partagée
#define SHM_VITESSE_KEY 1001
#define SHM_ALTITUDE_KEY 1002
// ... autres clés

int shmid_vitesse;
double *p_vitesse_partagee; // Pointeur vers la vitesse

// 1. Allocation du segment pour la vitesse
shmid_vitesse = shmget(SHM_VITESSE_KEY, sizeof(double), IPC_CREAT |
    0666);
if (shmid_vitesse == -1) {
    perror("shmget vitesse");
    exit(EXIT_FAILURE);
}
```

```c

// 2. Attachement du segment
p_vitesse_partagee = (double*) shmat(shmid_vitesse, NULL, 0);
if (p_vitesse_partagee == (void*) -1) {
    perror("shmat vitesse");
    exit(EXIT_FAILURE);
}

// 3. Initialisation de la valeur
*p_vitesse_partagee = -50.0; // Vitesse de chute initiale
```

Listing 3.5 – Allocation et attachement dans `supervision.c`

3.3.2 Code Commenté : Attachement (Client)

Un processus client, comme `lander_physics.c`, ne fait que récupérer l'identifiant du segment existant et s'y attacher.

```c
// La clé doit être identique à celle de la supervision
#define SHM_VITESSE_KEY 1001

int shmid_vitesse;
double *p_vitesse_partagee;

// 1. Récupération de l'ID du segment (sans IPC_CREAT)
shmid_vitesse = shmget(SHM_VITESSE_KEY, sizeof(double), 0666);
if (shmid_vitesse == -1) {
    perror("shmget vitesse client");
    exit(EXIT_FAILURE);
}

// 2. Attachement du segment
p_vitesse_partagee = (double*) shmat(shmid_vitesse, NULL, 0);
if (p_vitesse_partagee == (void*) -1) {
    perror("shmat vitesse client");
    exit(EXIT_FAILURE);
}

// Le processus peut maintenant lire ou écrire la vitesse
printf("Vitesse actuelle: %f m/s\n", *p_vitesse_partagee);
```

Listing 3.6 – Attachement dans un processus client (ex: `lander_physics.c`)

3.4 Bonnes Pratiques et Pièges

- **Toujours vérifier les retours de fonctions :** Les appels système comme `shmget` et `shmat` peuvent échouer. Il est impératif de tester leur valeur de retour (souvent `-1` ou `(void*)-1`) et d'utiliser `perror()` pour afficher une erreur claire.
- **Nettoyer systématiquement :** Le processus créateur doit avoir un gestionnaire de signaux (ex : pour `SIGINT`, `SIGTERM`) qui s'assure d'appeler `shmctl` avec `IPC_RMID` pour chaque segment créé, même en cas de fin abrupte du programme.
- **Utiliser des clés cohérentes :** Une erreur dans la clé (`key`) empêchera les processus de communiquer. L'utilisation d'un fichier d'en-tête commun (`.h`) pour définir toutes les clés est une excellente pratique.

Chapitre 4

Synchronisation et Protection des Données

Dans un système multi-processus, l'accès concurrent à une ressource partagée (ici, la mémoire partagée) sans précaution conduit inévitablement à des *Race Conditions* (Conditions de course), rendant le système instable et les données incohérentes. Ce chapitre présente la solution à ce problème : les sémaphores.

4.1 Le Problème de l'Accès Concurrent

Prenons un exemple critique dans notre simulateur : la mise à jour de la vitesse. Une variable de type `double` occupe 8 octets en mémoire. Un processeur 64 bits peut écrire ces 8 octets en une seule instruction, mais dans un contexte multi-tâches, rien ne garantit que l'écriture ne sera pas interrompue.

1. Le processus **Physique** commence à écrire la nouvelle vitesse dans le segment partagé. Il écrit les 4 premiers octets.
2. **Interruption système** : Le noyau suspend le processus Physique pour donner la main au processus Guidage, dont le tour est venu.
3. Le processus **Guidage** lit la valeur de la vitesse. Il lit les 4 octets "nouveaux" (qui viennent d'être écrits) et les 4 octets "anciens" (qui n'ont pas encore été écrasés).
4. **Résultat :** La valeur lue est totalement aberrante (par exemple, une valeur très petite, très grande, ou `NaN`), le guidage panique et le moteur s'arrête ou s'emballe.

Ce scénario est une condition de course classique. Pour y remédier, il faut garantir que l'opération "lire/écrire la vitesse" soit **atomique** : elle doit se dérouler en une seule fois, sans pouvoir être interrompue.

4.2 Solution : L'Exclusion Mutuelle (Mutex)

Pour garantir l'atomicité des opérations sur une ressource partagée, nous utilisons des sémaphores binaires, communément appelés **Mutex** (Mutual Exclusion).

Analogie du Jeton

Imaginez une pièce unique (la zone mémoire critique) fermée à clé. La clé est posée sur une table à l'extérieur.

- Si la clé est là (Sémaphore = 1), un processus la prend, entre dans la pièce et ferme à clé. Personne d'autre ne peut entrer.
- Si la clé n'est pas là (Sémaphore = 0), les autres processus doivent attendre devant la porte. Le noyau les met en sommeil, ils ne consomment pas de CPU.
- En sortant, le processus repose la clé sur la table (Sémaphore = 1) et réveille l'un des processus qui attendent.

Le mutex garantit qu'un seul processus à la fois peut se trouver dans la "section critique".

4.3 Implémentation Technique avec `semop`

La fonction système `semop` (Semaphore Operation) est l'outil principal pour manipuler les sémaphores System V. Elle permet d'effectuer une ou plusieurs opérations atomiques sur un ensemble de sémaphores.

4.3.1 Structure `sembuf`

Toute opération sur un sémaphore via `semop` est décrite par cette structure, définie par le noyau :

```
struct sembuf {
    unsigned short sem_num;  // Index du sémaphore dans l'ensemble (0 si unique)
    short          sem_op;   // Opération à effectuer (-1, 0, ou +1)
    short          sem_flg;  // Drapeaux (généralement 0)
};
```

Listing 4.1 – La structure `sembuf`

4.3.2 L'Opération P (Prendre / Wait)

Elle tente de décrémenter le sémaphore pour "prendre le jeton".

- Si la valeur du sémaphore est > 0 : La valeur est décrémentée et le processus continue son exécution (il entre en section critique).
- Si la valeur du sémaphore est $= 0$: Le noyau **bloque le processus**. Il est placé dans une file d'attente et ne consomme plus de CPU. Il sera réveillé automatiquement quand la valeur du sémaphore redeviendra positive.

```
int sem_prendre(int semid, int ns) {
    struct sembuf op;
    op.sem_num = ns;    // Numéro du sémaphore
    op.sem_op  = -1;    // On décrémente (opération P)
    op.sem_flg = 0;
    // Appel système bloquant et atomique
    return semop(semid, &op, 1);
}
```

Listing 4.2 – Wrapper pour l'opération P (Acquérir)

4.3.3 L'Opération V (Vendre / Signal)

Elle incrémente le sémaphore pour "rendre le jeton" (libérer la ressource).

— La valeur du sémaphore passe de 0 à 1.

— Le noyau vérifie si des processus attendent ce sémaphore. Si oui, il en réveille un (selon sa politique d'ordonnancement) qui pourra alors procéder à son opération P.

```
int sem_rendre(int semid, int ns) {
    struct sembuf op;
    op.sem_num = ns;    // Numéro du sémaphore
    op.sem_op  = 1;     // On incrémente (opération V)
    op.sem_flg = 0;
    return semop(semid, &op, 1);
}
```

Listing 4.3 – Wrapper pour l'opération V (Libérer)

4.4 Synchronisation Avancée : Le Patron de Conception "Barrière de Rendez-Vous"

Outre la protection des données (exclusion mutuelle), les sémaphores sont parfaits pour synchroniser temporellement le démarrage de plusieurs processus. C'est le motif de conception **Barrière**.

L'objectif est de s'assurer que tous les processus (Physique, Guidage, Environnement) commencent leur boucle de simulation exactement au même moment logique ($t = 0$), et non dès qu'ils sont lancés.

1. La **Supervision** crée un sémaphore dédié, nommé "RDV" (pour Rendez-Vous), et l'initialise à la valeur 0.
2. Les processus **Physique**, **Guidage** et **Environnement** effectuent tous un `sem_prendre` sur ce sémaphore. Comme sa valeur est 0, ils sont tous bloqués instantanément.
3. La **Supervision** attend (par exemple, avec un `sleep` court) que tous les autres processus aient eu le temps de se bloquer sur la barrière.
4. La **Supervision** effectue un `semctl(RDV_SEMID, 0, SETVAL, 4)`. La valeur du sémaphore passe brusquement à 4.
5. Les 3 processus bloqués (et la supervision elle-même, qui fait aussi un `sem_prendre`) peuvent maintenant décrémenter le sémaphore (4 - 1 - 1 - 1 - 1 = 0). Ils se débloquent **tous (quasi) simultanément** et démarrent leur mission.

Danger des Deadlocks (Interblocages)

Si un processus "Prend" un Mutex mais plante avant de pouvoir le "Rendre", le sémaphore reste à 0. Tous les autres processus qui tenteront d'accéder à cette ressource seront bloqués indéfiniment. C'est pour cela que notre gestionnaire de signaux `arret_propre` est crucial : en cas d'erreur ou d'interruption (Ctrl+C), il doit s'assurer de libérer tous les sémaphores pris par le processus.

Chapitre 5

Gestion du Temps et Déterminisme

La simulation physique et le contrôle-commande nécessitent une base de temps stable et précise. Le calculateur de guidage doit analyser l'état du Lander à intervalles réguliers, et la simulation physique doit avancer à une fréquence constante. C'est la définition même d'un système temps réel.

5.1 Pourquoi `sleep()` est insuffisant ?

Une approche naïve pour implémenter une tâche périodique serait d'utiliser la fonction `sleep()` ou `usleep()` dans une boucle infinie.

```
while(1) {
    calcul_physique();
    usleep(100000); // Dormir 100ms
}
```

Listing 5.1 – Approche naïve et non déterministe

Cette méthode présente deux défauts majeurs, la rendant inacceptable pour un système critique.

5.1.1 Dérive Temporelle (Drift)

Le temps total d'un cycle est la somme du temps de calcul et du temps de sommeil : $T_{cycle} = T_{calcul} + T_{sommeil}$. Si le calcul physique prend $5ms$, le cycle réel est de $105ms$ et non $100ms$. Cette erreur de $5ms$ s'accumule à chaque itération. Au bout de 20 secondes (200 cycles), l'horloge interne du système aurait accumulé une seconde de retard par rapport au temps réel. La simulation perd toute sa précision.

5.1.2 Gigue (Jitter)

La fonction `sleep` ne garantit pas un réveil exact. Elle garantit seulement que le processus dormira *au moins* le temps demandé. Si le système est chargé, si d'autres processus ont une priorité plus élevée, ou pour des raisons internes à l'ordonnanceur du noyau, le réveil peut être retardé de manière imprévisible. Cette *gigue* (jitter) rend le comportement du système non déterministe.

5.2 La Solution : Les Timers Système (`setitimer`)

Pour garantir une fréquence stable et précise (ex : $10Hz$ pour la physique), nous utilisons les mécanismes de timer fournis par le noyau Linux via la fonction `setitimer`. Ils fonctionnent par interruption logicielle périodique, indépendamment du temps d'exécution du code de l'application.

Le noyau est responsable de générer un signal (par défaut, `SIGALRM`) à une fréquence parfaitement régulière. Notre processus n'a plus qu'à intercepter ce signal pour exécuter sa logique.

5.2.1 Configuration du Timer

Nous programmons l'envoi périodique du signal `SIGALRM` en configurant une structure `itimerval`.

```
struct itimerval timer;

// Premier déclenchement (it_value)
timer.it_value.tv_sec = 0;
timer.it_value.tv_usec = 100000; // 100ms

// Périodicité (it_interval) : Rechargement automatique
timer.it_interval.tv_sec = 0;
timer.it_interval.tv_usec = 100000; // 100ms

// Activation sur l'horloge réelle (ITIMER_REAL)
// Le noyau enverra SIGALRM toutes les 100ms
setitimer(ITIMER_REAL, &timer, NULL);
```

Listing 5.2 – Configuration d'un timer avec `setitimer`

La structure `itimerval` contient deux champs :

- `it_value` : Délai avant la première expiration.
- `it_interval` : Intervalle entre les expirations suivantes. Si c'est 0, le timer ne se rechargera pas (il ne s'activera qu'une seule fois).

5.3 Gestion des Signaux (`sigaction`)

Un signal est une interruption logicielle asynchrone envoyée par le noyau à un processus. Par défaut, le signal `SIGALRM` a pour action de terminer le programme. Nous devons le "capturer" (hooker) pour lui associer notre propre routine, le gestionnaire de signal (handler).

La fonction `sigaction` est l'interface moderne et recommandée pour cela.

```
struct sigaction sa;

// On associe notre fonction 'simulation_step' au signal
sa.sa_handler = simulation_step;

// On vide le masque de signaux pour n'en bloquer aucun pendant l'exécution
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0; // Pas de drapeaux particuliers

```

```
// Application de la configuration
sigaction(SIGALRM, &sa, NULL);
```

Listing 5.3 – Association d'un gestionnaire au signal `SIGALRM`

La fonction `simulation_step` est une fonction simple qui prend un entier en argument (le numéro du signal) et ne retourne rien (`void`). C'est cette fonction qui sera appelée toutes les 100ms.

5.4 Architecture Événementielle

Grâce à ce mécanisme, l'architecture du programme s'inverse. Le `main` ne contient plus la logique métier dans une boucle `while`. Il se contente d'effectuer les initialisations, d'armer le timer, puis d'attendre indéfiniment. C'est le noyau qui "pilote" l'exécution de notre logique en envoyant des signaux.

```
// Notre logique métier, exécutée périodiquement par le noyau
void simulation_step(int sig) {
    // 1. Acquisition des données (Mutex P)
    // 2. Calculs (physique ou guidage)
    // 3. Action et écriture des résultats (Mutex V)
}

int main() {
    // Initialisations (création des IPC, attachement mémoire...)

    // Configuration du timer et du gestionnaire de signal
    configurer_timer_et_signal();

    // Boucle d'attente infinie.
    // Le processeur est au repos, il ne consomme quasiment rien.
    // Il est réveillé uniquement par le signal SIGALRM.
    while(1) {
        pause();
    }
}
```

Listing 5.4 – Structure du programme avec une architecture événementielle

L'appel système `pause()` est très efficace : il met le processus en état de sommeil jusqu'à ce que *n'importe quel signal* soit reçu. Lorsque `SIGALRM` arrive, le noyau réveille le processus, exécute le gestionnaire `simulation_step`, puis le retour de `pause()` est traité, et la boucle recommence, appelant `pause()` à nouveau pour attendre la prochaine interruption.

5.5 Soft Real-Time vs Hard Real-Time

Il est crucial de comprendre les limites de cette approche. Linux standard (avec un noyau non préemptif comme celui utilisé ici) est un système *Soft Real-Time*.

— Il ne garantit **pas un temps de latence strict**. Un événement de haute priorité dans le noyau peut retarder l'exécution de notre gestionnaire de quelques millisecondes.

— Il garantit une **moyenne temporelle**. Sur une machine moderne peu chargée, la précision est de l'ordre de la microseconde, ce qui est amplement suffisant pour notre simulation.

Un système *Hard Real-Time* (comme un noyau Linux RT patché avec Xenomai, ou un RTOS comme VxWorks ou FreeRTOS) garantirit une latence maximale bornée (par exemple, "le gestionnaire s'exécutera toujours moins de 50µs après le signal"). Pour une mission spatiale réelle, une telle garantie serait indispensable.

Chapitre 6

Modélisation Physique du Lander

Au-delà de l'architecture système, la pertinence de la simulation repose sur la fidélité du modèle physique. Ce chapitre détaille comment le processus `lander_physics` simule la dynamique du module martien en appliquant les lois fondamentales de la mécanique.

6.1 Le Modèle : Dynamique du Point Matériel

Le processus `lander_physics` simule le mouvement du module en appliquant le Principe Fondamental de la Dynamique (Seconde Loi de Newton) à chaque pas de temps. Le Lander est modélisé comme un point matériel de masse m sur lequel s'exercent plusieurs forces.

L'accélération instantanée du module est donnée par la somme des forces :

$$a(t) = \frac{\sum \vec{F}}{m}$$

Ce calcul est effectué à chaque cycle du timer, c'est-à-dire toutes les 100 ms ($dt = 0.1s$).

6.2 Bilan des Forces

Trois forces principales s'exercent sur le module de masse $m = 600kg$:

6.2.1 Poids ($\vec{P}$)

C'est la force d'attraction gravitationnelle exercée par Mars. Elle est constante et dirigée vers le centre de la planète (vers le bas dans notre repère).

$$P = -m \cdot g_{mars} \quad \text{avec} \quad g_{mars} \approx 3.71\, m/s^2$$

Le signe négatif indique que cette force s'oppose à une altitude croissante.

6.2.2 Traînée Atmosphérique ($\vec{D}$

C'est la force de frottement de l'air martien, qui s'oppose toujours au mouvement relatif du Lander par rapport à l'atmosphère.

$$D = \frac{1}{2} \cdot \rho \cdot C_x \cdot S \cdot (v_{rel})^2 \cdot \text{sgn}(v_{rel})$$

— $\rho = 0.020\,kg/m^3$: Masse volumique de l'atmosphère martienne, très ténue.
— $C_x = 1.5$: Coefficient de traînée, qui dépend de la forme du Lander.
— S : Surface de référence (maître-couple) du Lander.
— $v_{rel} = v_{lander} - v_{vent}$: Vitesse relative du Lander par rapport à l'air.
— $\text{sgn}(v_{rel})$: La fonction signe. Elle est cruciale pour que la traînée s'oppose toujours au mouvement (positive si v_{rel} est négatif, et vice-versa).

6.2.3 Poussée Moteur ($\vec{T}$

C'est la force générée par les rétrofusées du Lander. Elle est commandée par le système de guidage et est supposée orientée verticalement vers le haut.

$$T = \text{Commande} \quad \text{avec} \quad 0 \leq T \leq 20000\,N$$

La valeur maximale de 20 kN est une contrainte physique du matériel.

6.3 Intégration Numérique : La Méthode d'Euler

Les équations du mouvement sont continues, mais notre simulation fonctionne en temps discret (par pas de 100 ms). Nous devons donc convertir les équations différentielles en équations de récurrence. Pour ce projet, la méthode d'Euler explicite est suffisante en raison de la simplicité du modèle et du pas de temps relativement grand.

L'accélération est calculée à l'instant t :

$$a(t) = \frac{P + D + T}{m}$$

La vitesse et la position (altitude) à l'instant $t + dt$ sont ensuite estimées en se basant sur les valeurs à l'instant t :

```c
// v(t+dt) = v(t) + a(t) * dt
double nouvelle_vitesse = vitesse_actuelle + acceleration * DT;

// x(t+dt) = x(t) + v(t) * dt
double nouvelle_altitude = altitude_actuelle + vitesse_actuelle * DT;
```

Listing 6.1 – Intégration numérique d'Euler en C

Où `DT` est une constante définie à `0.1`. Bien que moins précise que des méthodes plus complexes comme Runge-Kutta, la méthode d'Euler est facile à implémenter et à comprendre, ce qui la rend idéale pour ce projet pédagogique.

6.4 Implémentation dans le Projet

Le code du processus `lander_physics` se trouve principalement dans le gestionnaire de signal exécuté toutes les 100 ms. La boucle de calcul est protégée par des mutex pour garantir la cohérence des données partagées.

```c
#define DT 0.1 // Pas de temps en secondes
#define MASSE 600.0
#define GMARS 3.71

```

```c
// Gestionnaire de signal pour SIGALRM (déclenché toutes les 100ms)
void physique_step(int sig) {
    // --- 1. Acquisition des données (Protégé par Mutex) ---
    sem_prendre(sem_poussee_id, 0);
    double poussee = *p_poussee_partagee;
    sem_rendre(sem_poussee_id, 0);

    sem_prendre(sem_vent_id, 0);
    double vent = *p_vent_partagee;
    sem_rendre(sem_vent_id, 0);

    // On lit aussi l'état actuel du Lander
    sem_prendre(sem_vitesse_id, 0);
    double vitesse = *p_vitesse_partagee;
    sem_rendre(sem_vitesse_id, 0);

    sem_prendre(sem_altitude_id, 0);
    double altitude = *p_altitude_partagee;
    sem_rendre(sem_altitude_id, 0);

    // --- 2. Calculs ---
    double poids = -MASSE * GMARS;
    double vitesse_relative = vitesse - vent;
    double trainee = 0.5 * 0.020 * 1.5 * 15.0 * vitesse_relative * fabs(
   vitesse_relative);

    double somme_forces = poids + trainee + poussee;
    double acceleration = somme_forces / MASSE;

    double nouvelle_vitesse = vitesse + acceleration * DT;
    double nouvelle_altitude = altitude + vitesse * DT;

    // --- 3. Écriture des résultats (Protégé par Mutex) ---
    sem_prendre(sem_vitesse_id, 0);
    *p_vitesse_partagee = nouvelle_vitesse;
    sem_rendre(sem_vitesse_id, 0);

    sem_prendre(sem_altitude_id, 0);
    *p_altitude_partagee = nouvelle_altitude;
    sem_rendre(sem_altitude_id, 0);
}
```

Listing 6.2 – Boucle de calcul physique dans `lander_physics.c`

Ce code illustre parfaitement l'interaction entre tous les concepts vus précédemment : il est déclenché par un timer (Chapitre 5), il lit et écrit dans des segments de mémoire partagée (Chapitre 3), et il protège chaque accès par des mutex (Chapitre 4) pour éviter les conditions de course.

Chapitre 7

Loi de Commande et Guidage

Le processus `guidance_system` est le cerveau du Lander. Son rôle est d'analyser l'état actuel du module (vitesse, altitude) et de décider de la force de poussée à appliquer pour atteindre un objectif précis : un atterrissage en douceur.

7.1 L'Objectif : Stabiliser la Vitesse de Descente

L'objectif principal du guidage est de réguler la vitesse de descente verticale pour qu'elle atteigne et maintienne une consigne de sécurité. Une vitesse trop élevée provoquerait un crash, tandis qu'une vitesse trop faible (ou positive) ferait "flotter" le module ou gaspillerait du carburant.

Nous avons fixé une consigne de vitesse cible :

$$V_{cible} = -2.0\, m/s$$

Cette valeur représente une vitesse d'impact faible mais non nulle, permettant un contact final en douceur sans risque de rebond.

7.2 Théorie de la Commande : Le Correcteur PI (Proportionnel-Intégral)

Pour atteindre et maintenir cette consigne, nous utilisons un type de correcteur très répandu en automatique : le correcteur **Proportionnel-Intégral** (PI). Il combine la réactivité d'une action proportionnelle à la précision d'une action intégrale.

7.2.1 Calcul de l'Erreur $\varepsilon(t) = V_{cible} - V_{mesure}(t)$

À chaque cycle de calcul (toutes les 500 ms), le guidage calcule l'erreur entre la vitesse souhaitée et la vitesse mesurée :

$$\varepsilon(t) = V_{cible} - V_{mesure}(t)$$

— Si $V_{mesure} = -50m/s$ (trop rapide), l'erreur est $\varepsilon = -2 - (-50) = +48m/s$. Le guidage va demander une forte poussée vers le haut.
— Si $V_{mesure} = -1m/s$ (trop lent), l'erreur est $\varepsilon = -2 - (-1) = -1m/s$. Le guidage va réduire la poussée.

7.2.2 Terme Proportionnel (K_p) : Réactivité

Le terme proportionnel réagit à l'erreur *instantanée*. Plus l'erreur est grande, plus la commande est forte. Il permet une réaction rapide aux perturbations.

$$u_p(t) = K_p \cdot \varepsilon(t)$$

K_p est un gain à régler. Un K_p trop élevé rend le système nerveux et risque de le rendre instable (oscillations). Un K_p trop faible le rend trop lent.

7.2.3 Terme Intégral (K_i) : Précision

Le terme intégral réagit à l'*accumulation* de l'erreur passée. Il est essentiel pour éliminer l'erreur statique.

$$u_i(t) = K_i \cdot \int_0^t \varepsilon(\tau)\, d\tau$$

Sans le terme intégral, si la poussée nécessaire pour contrer la gravité à V_{cible} est, par exemple, de 1500 N, et que le terme proportionnel seul ne fournit que 1400 N, le système se stabilisera avec une erreur résiduelle permanente. Le terme intégral va accumuler cette erreur jusqu'à ce que la commande totale atteigne les 1500 N nécessaires, annulant ainsi l'erreur.

7.2.4 Équation de la Commande $u(t) = K_p \cdot \varepsilon(t) + K_i \cdot \int_0^t \varepsilon(\tau)\, d\tau$

La commande de poussée brute $u(t)$ envoyée au moteur est la somme de ces deux termes :

$$u(t) = K_p \cdot \varepsilon(t) + K_i \cdot \int_0^t \varepsilon(\tau)\, d\tau$$

Dans notre implémentation discrète, l'intégrale est approximée par une somme (méthode des rectangles) :

$$\int_0^t \varepsilon(\tau)\, d\tau \approx \sum_{k=0}^{n} \varepsilon_k \cdot T_{ech}$$

où T_{ech} est la période d'échantillonnage du guidage (500 ms).

7.3 Contraintes Physiques et Pratiques

La commande $u(t)$ calculée par le correcteur PI est un idéal mathématique. Le moteur physique a des limites qui doivent être respectées.

7.3.1 Saturation (Clamping) de la Commande $0 \leq T_{moteur} \leq T_{max}$

Le moteur du Lander ne peut pas fournir une poussée infinie, et il ne peut pas tirer vers le bas (poussée négative). Nous devons donc "saturer" ou "clipper" la commande pour la forcer dans les limites physiques.

$$0 \leq T_{moteur} \leq T_{max} \quad \text{avec} \quad T_{max} = 20000\, N$$

```c
// La commande brute est 'cmd'
if (cmd < 0.0) {
    cmd = 0.0;          // Impossible de tirer vers le bas
}
else if (cmd > MAX_THRUST) {
    cmd = MAX_THRUST; // On ne peut pas dépasser la poussée max
}
// 'cmd' est maintenant la commande envoyée au moteur
```

Listing 7.1 – Application de la saturation sur la commande

7.3.2 Problème du "Windup" et Stratégie Anti-Windup

Un problème classique avec les correcteurs PI est le *windup* (emballement) de l'intégrale. Si la commande est saturée (par exemple, le moteur est déjà au maximum mais l'erreur est toujours positive), le terme intégral continue d'accumuler l'erreur. L'intégrale devient alors énorme. Lorsque l'erreur finira par devenir négative, il faudra beaucoup de temps pour "désaccumuler" cette intégrale, ce qui provoquera un grand dépassement (overshoot) de la consigne.

Une stratégie anti-windup simple consiste à **geler l'intégration** lorsque la commande est saturée.

7.4 Implémentation dans le Projet

Le code du processus `guidance_system` est structuré de manière similaire à celui de la physique. Il est exécuté toutes les 500 ms par son propre timer.

```c
#define V_CONSIGNE -2.0
#define KP 800.0   // Gain Proportionnel (à régler)
#define KI 400.0   // Gain Intégral (à régler)
#define T_ECH 0.5  // Période d'échantillonnage (500ms)

static double somme_erreurs = 0.0; // Stocke l'intégrale de l'erreur

// Gestionnaire de signal pour SIGALRM (déclenché toutes les 500ms)
void guidage_step(int sig) {
    // --- 1. Acquisition des données (Protégé par Mutex) ---
    sem_prendre(sem_vitesse_id, 0);
    double vitesse = *p_vitesse_partagee;
    sem_rendre(sem_vitesse_id, 0);

    // --- 2. Calculs ---
    double erreur = V_CONSIGNE - vitesse;
    somme_erreurs += erreur * T_ECH; // Intégration numérique

    double cmd_P = KP * erreur;
    double cmd_I = KI * somme_erreurs;
    double cmd_brute = cmd_P + cmd_I;

    // --- 3. Saturation (Clamping) ---
    double cmd_finale = cmd_brute;
    if (cmd_finale < 0.0) cmd_finale = 0.0;
    else if (cmd_finale > MAX_THRUST) cmd_finale = MAX_THRUST;

```

```
28      // --- 4. Écriture des résultats (Protégé par Mutex) ---
29      sem_prendre(sem_poussee_id, 0);
30      *p_poussee_partagee = cmd_finale;
31      sem_rendre(sem_poussee_id, 0);
32  }
```

Listing 7.2 – Boucle de calcul du guidage dans `guidance_system.c`

Ce code illustre une boucle d'asservissement complète et fonctionnelle. Les gains K_p et K_i sont des paramètres critiques qui doivent être ajustés (par essais-erreurs ou des techniques plus avancées) pour obtenir un comportement stable et rapide.

Chapitre 8

Résultats et Validation

Après avoir conçu l'architecture, implémenté les mécanismes système et modélisé la physique et le guidage, l'étape finale est de valider le comportement global du simulateur. Ce chapitre présente le protocole de test, les métriques d'évaluation et les résultats obtenus, démontrant la robustesse et la pertinence de notre implémentation.

8.1 Protocole de Test

Pour valider le système *Mars Lander RT*, nous avons défini un scénario de test complet qui met à l'épreuve à la fois le modèle physique et la loi de commande.

8.1.1 Scénario de Test

Le scénario est le suivant :

1. **État Initial :** Le Lander est largué à une altitude de $1000\,m$ avec une vitesse verticale initiale de $-50\,m/s$ (chute libre).
2. **Objectif :** Le système de guidage doit activer les moteurs pour stabiliser la vitesse de descente autour de la consigne de $-2.0\,m/s$ et réussir l'atterrissage.
3. **Perturbation :** À $t = 10\,s$, une rafale de vent horizontal de $40\,m/s$ est injectée par le processus `environnement`. Cette perturbation s'oppose à la descente et force le guidage à réagir.
4. **Fin de la Simulation :** La simulation s'arrête lorsque l'altitude atteint $0\,m$ (contact avec le sol).

8.1.2 Métriques d'Évaluation

La réussite de la mission est évaluée sur les métriques suivantes :

— **Vitesse Finale à l'Impact :** Doit être la plus proche possible de la consigne de $-2.0\,m/s$. Un atterrissage est considéré comme réussi si la vitesse finale est comprise entre $-3.0\,m/s$ et $-1.0\,m/s$.

— **Stabilité :** La commande de poussée ne doit pas présenter d'oscillations divergentes. Le système doit converger vers la consigne.

— **Réactivité :** Le guidage doit compenser la perturbation (le vent) en un temps raisonnable, sans dépassement excessif.

Terminal Output (Simulation)

```
[SUPERVISION] All processes synchronized. Starting mission...
 t=00.0s [PHYSIQUE] Alt: 1000.0m | Vit: -50.0m/s | Vent: 0.0m/s

t=00.5s [GUIDAGE] Err: 48.0m/s | Cmd: [##############] 95%
t=01.0s [PHYSIQUE] Alt: 950.0m | Vit: -45.5m/s | Vent: 0.0m/s
t=01.5s [GUIDAGE] Err: 43.5m/s | Cmd: [###############] 87%
...
 t=10.0s [ENVIRONNEMENT] >> INJECTION VENT : 40.0 m/s

 t=10.5s [PHYSIQUE] Alt: 450.2m | Vit: -20.1m/s | Vent: 40.0m/s

t=11.0s [GUIDAGE] Err: 18.1m/s | Cmd: [################] 65%
t=11.5s [PHYSIQUE] Alt: 430.1m | Vit: -16.8m/s | Vent: 40.0m/s
t=12.0s [GUIDAGE] Err: 14.8m/s | Cmd: [###############] 59%
...
 t=25.0s [PHYSIQUE] Alt: 150.4m | Vit: -2.1m/s | Vent: 40.0m/s

t=25.5s [GUIDAGE] Err: 0.1m/s | Cmd: [##############] 55%
t=26.0s [PHYSIQUE] Alt: 148.3m | Vit: -2.0m/s | Vent: 40.0m/s
t=26.5s [GUIDAGE] Err: 0.0m/s | Cmd: [##############] 54%
...
 t=48.5s [PHYSIQUE] Alt: 2.1m | Vit: -2.0m/s | Vent: 40.0m/s

t=49.0s [GUIDAGE] Err: 0.0m/s | Cmd: [##############] 54%
t=49.5s [PHYSIQUE] Alt: 0.1m | Vit: -2.0m/s | Vent: 40.0m/s
t=50.0s [GUIDAGE] Err: 0.0m/s | Cmd: [##############] 54%
 >> TOUCHDOWN REUSSI !

Vitesse finale: -1.98 m/s (Consigne: -2.0 m/s)
```

FIGURE 8.1 – Exemple de trace console montrant la stabilisation du système après une perturbation

8.2 Présentation des Résultats

L'exécution du scénario de test génère des traces console de la part des différents processus. La figure 8.1 présente une capture synthétisée de la sortie pendant la phase critique de la mission.

8.3 Discussion des Résultats

L'analyse des traces de la figure 8.1 permet de valider les points clés de notre architecture.

8.3.1 Validation de la Physique et de la Communication

On observe que le processus `PHYSIQUE` met à jour l'état toutes les 100 ms, tandis que le `GUIDAGE` agit toutes les 500 ms. Les données partagées (altitude, vitesse) sont lues par le guidage et les nouvelles commandes sont prises en compte par la physique au cycle suivant. L'absence d'erreur de segmentation ou de données incohérentes dans les traces valide le bon fonctionnement de la mémoire partagée et des sémaphores (Mutex).

8.3.2 Validation de la Loi de Commande

— **Phase de Freinage Initial (t < 10s) :** L'erreur est grande (+48 m/s), le correcteur PI génère une commande de poussée maximale (95%). La vitesse de chute est rapidement réduite de -50 m/s à environ -20 m/s.
— **Réaction à la Perturbation (t = 10s) :** L'injection de vent horizontal ralentit la descente (la vitesse passe de -20.1 m/s à -16.8 m/s). L'erreur de vitesse par rapport à la consigne augmente, ce qui fait monter la commande de poussée pour contrer à la fois la gravité et l'effet du vent.
— **Convergence :** Après la perturbation, on observe que l'erreur diminue et que la commande de poussée se stabilise autour de 54%. La vitesse de descente converge et se maintient de manière stable à la consigne de $-2.0\,m/s$.

Le comportement du correcteur PI est conforme à la théorie : il est réactif, élimine l'erreur statique et stabilise le système.

8.3.3 Validation du Temps Réel

Les cycles des processus `PHYSIQUE` (toutes les 100 ms) et `GUIDAGE` (toutes les 500 ms) sont respectés tout au long de la simulation, comme en témoignent les horodatages. L'utilisation des timers système et des signaux garantit une base de temps fiable, sans dérive, contrairement à une approche naïve avec `sleep()`.

8.4 Bilan de la Validation

Le scénario de test s'est déroulé avec succès et a permis de valider l'ensemble des concepts implémentés :

— **L'architecture multi-processus** est robuste et isole les sous-systèmes.
— **Les IPC System V** (mémoire partagée et sémaphores) permettent une communication inter-processus performante et sûre.
— **L'ordonnancement événementiel** basé sur les timers assure un déterminisme temporel suffisant pour notre application.
— **Le modèle physique** et le **correcteur PI** interagissent correctement pour simuler un atterrissage autonome et réussi.

La vitesse finale à l'impact est de $-1.98\,m/s$, ce qui est une excellente réussite par rapport à notre objectif de $-2.0\,m/s$. Le système a démontré sa capacité à compenser une perturbation externe et à stabiliser la trajectoire, validant ainsi la pertinence de l'approche choisie.

Chapitre 9

Conclusion et Perspectives

Le projet **Mars Lander RT** nous a permis de parcourir l'ensemble du cycle de développement d'un système critique, de la spécification à la validation. Ce chapitre final synthétise les acquis techniques, dresse un bilan du travail accompli, et ouvre des pistes pour des évolutions futures.

9.1 Acquis Techniques

Ce projet a été l'occasion de maîtriser un ensemble cohérent de concepts et d'outils de programmation système, essentiels au développement d'applications embarquées robustes.

- **Architecture Concurrente :** Conception et justification d'une architecture multi-processus pour garantir la robustesse et l'isolation des sous-systèmes.
- **IPC System V :** Implémentation maîtresse de la mémoire partagée pour une communication inter-processus haute performance (Zero-Copy) et des sémaphores pour l'exclusion mutuelle (Mutex) et la synchronisation (Barrière de Rendez-Vous).
- **Programmation Temps Réel :** Mise en œuvre d'une architecture événementielle basée sur les timers système (`setitimer`) et les signaux (`sigaction`) pour garantir un ordonnancement déterministe, sans dérive temporelle.
- **Modélisation et Contrôle :** Développement d'un modèle physique simple mais efficace et d'une loi de commande (correcteur PI) pour asservir un système dynamique.
- **Robustesse :** Mise en place de stratégies défensives (gestion des erreurs, nettoyage des ressources) pour assurer la stabilité à long terme du système.

9.2 Bilan du Projet

L'objectif initial de créer un simulateur d'atterrissage martien fonctionnel, temps réel et éducatif a été pleinement atteint. Le système démontre avec succès qu'il est possible de concevoir une application complexe et fiable en assemblant des briques logicielles standards sous Linux.

L'architecture modulaire, où chaque processus a une responsabilité claire (Physique, Guidage, Supervision, Environnement) et communique via une interface bien définie (le bus de données partagées), s'est avérée particulièrement efficace. Elle a permis de développer, tester et valider chaque partie indépendamment, facilitant grandement le débogage et la compréhension globale. La validation par un scénario de test complet a prouvé que

l'ensemble de la chaîne fonctionnait comme attendu, réussissant à stabiliser le Lander et à compenser une perturbation externe pour un atterrissage en douceur.

9.3 Limites de l'Implémentation Actuelle

Malgré son succès, il est important de garder un regard critique sur notre implémentation et de reconnaître ses limites, qui ouvrent la voie à des améliorations futures.

9.3.1 Limites du Modèle Physique

Le modèle physique a été volontairement simplifié pour des raisons pédagogiques et de performance de calcul.

- **Intégration Numérique :** La méthode d'Euler est une approximation d'ordre 1. Pour des pas de temps plus grands ou des dynamiques plus rapides, elle peut introduire des erreurs significatives. Des méthodes plus précises comme Runge-Kutta d'ordre 4 offriraient une meilleure fidélité.
- **Modèle d'Atmosphère :** Nous avons considéré une densité de l'air (ρ) et un coefficient de traînée (C_x) constants. En réalité, ces paramètres varient avec l'altitude.

9.3.2 Limites du Temps Réel

Le choix de Linux standard nous a permis de développer sur une plateforme accessible, mais nous place dans le cadre du *Soft Real-Time*.

- Le noyau Linux standard ne garantit pas de latence maximale bornée. Une forte charge système ou une tâche de priorité plus élevée pourrait retarder l'exécution de notre gestionnaire de signal de plusieurs millisecondes.
- Pour une application spatiale réelle, un tel comportement serait inacceptable. Il faudrait porter le système sur un **RTOS (Real-Time Operating System)** comme Xenomai (qui étend le noyau Linux) ou un noyau dédié comme VxWorks ou FreeRTOS, qui offrent des garanties de temps réel *dur* (Hard Real-Time).

9.4 Perspectives d'Évolution

Les limites identifiées précédemment nous suggèrent des axes d'amélioration naturels, tant sur le plan technique que fonctionnel.

9.4.1 Améliorations Techniques

- **Portage sur un RTOS :** Le passage à un noyau temps réel dur garantirait le déterminisme absolu des temps de réaction, une exigence non négociable pour un système de contrôle de vol critique.
- **Intégrateur plus précis :** L'implémentation d'un intégrateur Runge-Kutta augmenterait la précision de la simulation physique sans nécessiter une réduction drastique du pas de temps.
- **Correcteur PID complet :** L'ajout d'un terme Dérivé (D) au correcteur PI (pour former un PID) pourrait améliorer la rapidité de réaction et réduire le dépassement

(overshoot) lors des perturbations. Une implémentation complète avec une stratégie anti-windup robuste serait un excellent exercice.

9.4.2 Améliorations Fonctionnelles

— **Interface Graphique :** Le développement d'une interface graphique légère (avec une bibliothèque comme GTK ou Qt) permettrait de visualiser la trajectoire du Lander en temps réel, les courbes de vitesse, d'altitude et de poussée, rendant l'analyse bien plus intuitive.
— **Journalisation et Analyse :** L'ajout d'un système de log qui enregistre toutes les données dans un fichier permettrait une analyse post-simulation détaillée pour affiner les réglages du correcteur ou valider le modèle sur de longues durées.
— **Scénarios de Test Étendus :** Créer un langage de script pour définir des scénarios de test complexes (profils de vent variables, pannes de moteur, etc.) augmenterait considérablement les capacités de validation du simulateur.

9.5 Licence

Le code source développé dans le cadre de ce projet est distribué sous la licence **MIT**. Cette licence permissive permet à quiconque d'utiliser, de modifier et de distribuer le code, à des fins éducatives ou commerciales, tant que le copyright original est conservé. Nous encourageons la réutilisation et l'amélioration de cette base de travail pour la communauté.

—

En conclusion, le projet *Mars Lander RT* a rempli ses promesses en fournissant une base solide et complète pour la compréhension et la mise en œuvre des systèmes temps réels sous Linux. Il constitue une excellente passerelle vers des systèmes encore plus complexes et critiques, tels que ceux rencontrés dans l'industrie aérospatiale et automobile.