

# DOCUMENTATION TECHNIQUE

---

## Ordonnanceur Multitâche de Processus sous Linux

Mini-Projet : Systèmes d'Exploitation

Octobre - Décembre 2025

---

### Équipe de Développement

Arij Sebai • Aya Sakroufi • Balkis Hanafi  
Hadil Hasni • Wiem Ayari

---

L'Institut supérieur d'informatique à Ariana  
1ING3

# Table des Matières

<b>1. Introduction .....</b>	<b>Page 4</b>
<b>2. Choix des Structures de Données .....</b>	<b>Page 4</b>
• 2.1 Structure <b>process</b> : Le Cœur du Système	
• 2.2 Représentation des Données : Tableau Dynamique	
• 2.3 Représentation Implicite de la Ready Queue	
<b>3. Choix des Algorithmes d'Ordonnement .....</b>	<b>Page 6</b>
• 3.1 FIFO (First-In First-Out)	
• 3.2 Priority Preemptive	
• 3.3 Round Robin (RR)	
• 3.4 SRT (Shortest Remaining Time First)	
• 3.5 Multilevel Queue (Statique)	
• 3.6 Multilevel Feedback Queue (Dynamique)	
<b>4. Technologies et Architecture .....</b>	<b>Page 17</b>
• 4.1 Choix des Technologies	
• 4.2 Architecture du Projet	
• 4.3 Backend C : Mode Interactif vs Mode API	
• 4.4 Intégration complète : Frontend Next.js + Backend C	
• 4.5 Flow d'Exécution Complet	
<b>5. Déroulement du Développement SCRUM .....</b>	<b>Page 19</b>
• 5.1 Organisation Équipe	
• 5.2 Paramètres Scrum	
• 5.3 Product Backlog	
• 5.4 Réunion de Lancement (Sprint 0)	
• 5.5 Sprint Backlog 1	
• 5.6 Sprint Backlog 2	
• 5.7 Métriques SCRUM	
<b>6. Spécifications Techniques .....</b>	<b>Page 22</b>
• 6.1 Point d'Entrée (main.c) : Modes Interactif et API	
• 6.2 Format Fichier Configuration	
• 6.3 Générateur Configuration Automatique	
<b>7. Makefile et Compilation .....</b>	<b>Page 27</b>
• 7.1 Objectif du Makefile	
• 7.2 Variables Principales	
• 7.3 Règles Principales	
• 7.4 Déclaration PHONY	
• 7.5 Flags Compiler Expliqués	

- 7.6 Principes et Avantages
- 7.7 Utilisation Pratique

**8. Conclusion** ..... [Page 31](#)

# 1. Introduction

## Objectif du Projet

Ce projet est un **simulateur d'ordonnancement de processus sous Linux** avec une architecture hybride moderne : **Frontend Next.js 16/React 19 + Backend C**.

### Capacités principales :

- **✓ 6 algorithmes d'ordonnancement** : FIFO, Round-Robin, Priorité préemptive, SRT, Multilevel statique, Multilevel Dynamic (anti-famine)
- **✓ Interface Web interactive** : sélection algorithme, paramètres dynamiques (quantum, ordre priorité), visualisations temps réel
- **✓ Backend C performant** : moteur de simulation compilé, modes CLI interactif et API JSON
- **✓ Visualisations riches** : Gantt chart interactif (play/pause/zoom), graphiques (pie/bar charts), tableau détaillé avec métriques
- **✓ Générateur automatique** : création fichiers configuration aléatoires
- **✓ Métriques complètes** : temps d'attente, temps total, finish time, makespan, CPU utilization
- **✓ Architecture modulaire** : ajout nouvel algorithme = 1 fichier dans `policies/`, sans modifier `scheduler.c`

**Technologies** : Next.js 16, React 19, TypeScript, Tailwind CSS, Radix UI, Recharts (frontend) + C11/GCC, Make (backend)

**Dépôt Git** : [github.com/arijsebai/Projet-Ordonnancement-Linux](https://github.com/arijsebai/Projet-Ordonnancement-Linux) (branch: `main`)

## 2. Choix des Structures de Données

### 2.1 Justification des Structures Principales

#### Structure `process` : Le Cœur du Système

```
#define NAME_LEN 64
#define READY 0
#define RUNNING 1
#define BLOCKED 2
#define ZOMBIE 3

struct process {
    char name[NAME_LEN];           // Identification unique
    int arrival_time;              // Moment d'arrivée en système
    int exec_time;                 // Durée totale CPU requise (immuable)
    int priority;                  // Priorité statique (PETITE VALEUR = HAUTE
PRIORITÉ, convention Unix)
    int remaining_time;            // Temps restant à exécuter (modifiable)
    int waiting_time;              // Temps d'attente cumulé
    int status;                   // État : READY(0), RUNNING(1), BLOCKED(2),
ZOMBIE(3)
    int end_time;                 // Temps de fin d'exécution (pour
```

```

    métriques)
        int wait_time;           // Pour aging dynamique (Multilevel)
};

```

## 2.2 Représentation des Données : Tableau Dynamique

### Structure Générale

```

struct process *processes; // Pointeur vers tableau dynamique
int num_processes;        // Nombre de processus chargés

```

### Allocation au runtime :

```

processes = malloc(num_processes * sizeof(struct process));
if (!processes) { /* erreur allocation */ }

```

## 2.3 Représentation Implicite de la Ready Queue

### On utilise une représentation implicite :

```

for (int i = 0; i < n; i++) {
    if (procs[i].arrival_time <= time &&
        procs[i].remaining_time > 0 &&
        procs[i].status == READY) {
        selected = i;
        break;
    }
}

```

### Avantages de l'Approche Implicite

- ✓ **Zéro surcharge mémoire** supplémentaire
- ✓ **Code plus lisible** et directement mappable à l'OS réel
- ✓ **Flexibilité** : chaque politique définit son critère de "ready"
- ✓ **Pas de synchronisation** complexe entre structures

## 3. Choix des Algorithmes d'Ordonnancement

### 3.1 FIFO (First-In First-Out)

#### Principe

C'est une politique **non-préemptive**. Le processus arrivé le premier (**arrival\_time** le plus bas) est sélectionné et s'exécute jusqu'à sa fin complète sans interruption.

## Algorithme de Sélection et Simulation

### Étape 1 : Définir une fonction de sélection FIFO (`fifo_scheduler`)

Cette fonction est responsable de trouver quel processus exécuter.

À l'intérieur de cette fonction :

#### 1.1. Préparer la recherche du processus éligible

- Initialiser un indice "meilleur processus" à -1 (vide)
- Initialiser une variable "temps d'arrivée le plus tôt" à une valeur très élevée (`INT_MAX`)

#### 1.2. Parcourir tous les processus du système

- **1.2.1.** Sélectionner uniquement les processus "prêts" : ceux qui sont déjà arrivés (`arrival_time <= time`) ET qui n'ont pas encore terminé (`remaining_time > 0`)
- **1.2.2.** Parmi ces processus prêts, comparer leur `arrival_time` avec le "temps d'arrivée le plus tôt" trouvé jusqu'à présent
- **1.2.3.** Si un processus est trouvé avec un `arrival_time` plus bas, le marquer comme le nouveau "meilleur processus"

#### 1.3. Retourner l'indice du processus le plus anciennement arrivé

- Retourner l'index du "meilleur processus" (ou -1 si aucun processus n'est prêt)

### Étape 2 : Intégrer cette sélection dans la boucle principale de simulation

À chaque unité de temps (`time`) :

#### 2.1. Gérer les nouvelles arrivées

- Vérifier si de nouveaux processus arrivent à cet instant (`arrival_time == time`) et les marquer comme "prêts" (`status = READY`)

#### 2.2. Appeler la fonction de sélection (`fifo_scheduler`)

- La fonction `fifo_scheduler` est appelée pour déterminer quel processus doit s'exécuter

#### 2.3. Logique Non-Préemptive

- Tant que le processus en cours d'exécution n'est pas terminé, il restera celui avec le `arrival_time` le plus bas parmi tous les processus prêts
- Par conséquent, `fifo_scheduler` le re-sélectionnera à chaque tour, assurant qu'il n'est pas préempté par d'autres

#### 2.4. Exécuter le processus sélectionné

- **2.4.1.** Si un processus est sélectionné (`next != -1`) :
  - Exécuter ce processus pendant une unité de temps (décrémenter `remaining_time`)
  - Si le processus termine (`remaining_time == 0`), le marquer comme "terminé" (`status = ZOMBIE`) et incrémenter le compteur global `completed`
- **2.4.2.** Sinon (si `next == -1`) :

- Le processeur reste inactif (IDLE)

## 2.5. Avancer le temps

- Incrémenter `time` et répéter la boucle jusqu'à ce que tous les processus soient terminés (`completed == n`)

### Étape 3 : Générer les statistiques finales

À la fin de la simulation, utiliser l'historique d'exécution (notamment `end_time` pour chaque processus) pour calculer et afficher :

- Le tableau de Gantt
- Le temps de fin
- Le temps d'attente : `end_time - arrival_time - exec_time`
- Le temps d'attente moyen

### Avantages et Inconvénients

Aspect	Évaluation
✓ Très simple à implémenter	Parfait pour comprendre le concept
✓ Zéro préemption	Pas d'overhead context switch
✓ Déterministe	Toujours même résultat
✓ Bon pour batch	Tâches longues acceptées
✗ Très injuste	Processus court doit attendre les longs
✗ Temps d'attente élevé	Mauvais pour interactif
✗ Convoy effect	Un processus long bloque tout le système

### Cas d'Usage Réel

**Linux/Unix** : Utilisé pour batch jobs, scripts de maintenance (quand démarrage en background via cron).

## 3.2 Priority Preemptive

### Principe

À chaque instant, le processus le **plus prioritaire préempte immédiatement** tout processus en cours d'exécution.

#### ⚠ Convention de Priorité :

- **Mode Ascending** (petite valeur = haute priorité)
- **Mode Descending** (grande valeur = haute priorité)

### Algorithme de Sélection et Simulation

## Étape 0 : Initialiser le mode de priorité

- Récupérer le paramètre `prio_mode` (passé en CLI / API) :
  - `prio_mode = 0` → Ascending (petite valeur = haute prio)
  - `prio_mode = 1` → Descending (grande valeur = haute prio)
- **Défauts réels :**
  - CLI (interactif) : `prio_mode` démarre à **1 (descending)** dans `main.c`
  - API Next.js : la route passe `--prio-order asc` → `prio_mode = 0`

## Étape 1 : Définir une fonction de sélection préemptive

Cette fonction permet de choisir le prochain processus à exécuter à chaque unité de temps.

À l'intérieur de cette fonction :

### 1.1. Ignorer le processus actuellement en cours d'exécution

- Cela permet une préemption immédiate si un meilleur processus devient disponible

### 1.2. Préparer la recherche du processus le plus prioritaire

- Initialiser un indice "meilleur processus" à -1 (vide)
- Initialiser une valeur extrême selon le mode :
  - Mode Ascending : `best_priority = INT_MAX` (on cherche le minimum)
  - Mode Descending : `best_priority = INT_MIN` (on cherche le maximum)

### 1.3. Parcourir tous les processus

- **1.3.1.** Sélectionner uniquement ceux qui sont arrivés (`arrival_time <= time`) ET n'ont pas encore terminé (`remaining_time > 0`)
- **1.3.2.** Comparer leur priorité selon le mode :
  - Mode Ascending : si `priority < best_priority` → nouveau meilleur candidat
  - Mode Descending : si `priority > best_priority` → nouveau meilleur candidat
- **1.3.3.** Mettre à jour `best_priority` et l'indice du meilleur processus

### 1.4. Retourner l'indice du processus le plus prioritaire

- Retourner -1 ("aucun processus prêt") si aucun n'est prêt

## Étape 2 : Intégrer cette sélection dans la boucle principale de simulation

À chaque unité de temps :

### 2.1. Appeler la fonction de sélection préemptive pour déterminer quel processus exécuter

### 2.2. Si un processus est sélectionné :

- Exécuter ce processus pendant une unité de temps et décrémenter son temps restant

### 2.3. Sinon :

- Le processeur reste inactif (CPU IDLE)

### 2.4. Incrémenter le temps et répéter jusqu'à ce que tous les processus soient terminés



### Étape 3 : Générer les résultats finaux

À la fin de la simulation, générer le diagramme de Gantt et les statistiques à partir de l'historique d'exécution.

#### Avantages et Inconvénients

Aspect	Évaluation
✓ <b>Processus critiques prioritaires</b>	Parfait pour temps réel
✓ <b>Flexible</b>	Modes ascendant/descendant
✓ <b>Simple à implémenter</b>	Pas de structure complexe
✗ <b>Processus faible priorité peuvent starver</b>	Risque famine
✗ <b>Overhead context switches</b>	Dégradation performance si trop préemptions
✗ <b>Pas équitable</b>	Processus longs = toujours peu servis

#### Cas d'Usage Réel

**Systèmes temps réel dur** : Avionique, médical, contrôle industriel (processus critiques d'abord).

### 3.3 Round Robin (RR)

#### Principe

Chaque processus reçoit un **quantum** de temps fixe (configurable par l'utilisateur). Si le processus ne se termine pas après avoir consommé son quantum, il retourne en **fin de ready queue** et attend son prochain tour.

#### Algorithme de Sélection et Simulation

##### Étape 1 : Initialisation

- Créer une copie des processus pour ne pas modifier l'original
- Pour chaque processus :
  - `remaining_time = exec_time` (temps restant à exécuter)
  - `waiting_time = 0` (temps d'attente cumulé)
  - `end_time = -1` (marqueur de non-terminé)
- Initialiser le temps global à 0
- Initialiser `completed = 0` (compteur de processus terminés)
- Créer une **file d'attente linéaire** (ready queue) avec indices `head` et `tail` initialisés à 0

##### Étape 2 : Gestion de la Ready Queue

À chaque itération de la boucle principale :

#### 2.1. Ajouter les nouveaux arrivants à la ready queue

Parcourir tous les processus :

- **Critères d'ajout :**

- `arrival_time == time` (processus qui arrive exactement à cet instant)
- `remaining_time > 0` (processus non terminé)

- **Mécanisme d'ajout :**

- Si les critères sont remplis → ajouter `ready[tail++] = i`
- **⚠ Note d'implémentation :** Le code actuel n'implémente pas de vérification de duplication explicite, ce qui peut théoriquement causer des ajouts multiples si la logique de gestion de queue n'est pas rigoureuse

## 2.2. Vérifier si la queue est vide

- Si `head == tail` (queue vide, aucun processus prêt) :
  - Chercher le prochain `arrival_time` futur parmi les processus non terminés
  - Sauter directement à ce temps : `time = next_arrival`
  - Afficher : `"%4d [IDLE] []"`
  - Continuer à l'itération suivante

## Étape 3 : Sélection et Exécution du Processus

### 3.1. Extraire le processus en tête de file

- `curr = ready[head]` (premier processus dans la queue, index dans le tableau)
- Incrémenter `head++` (retirer de la queue)

### 3.2. Calculer le temps d'exécution effectif

- `run = min(remaining_time, quantum)`
  - Si `remaining_time < quantum` → exécuter seulement le temps restant
  - Sinon → exécuter exactement le quantum complet

### 3.3. Afficher l'état actuel

- Format : `"%4d %-8s [ready_queue_content]"`
  - Temps actuel
  - Nom du processus en cours d'exécution
  - Contenu de la ready queue : `"name:remaining_time"` séparés par virgules

### 3.4. Calcul du waiting\_time

- **⚠ Note d'implémentation :** Le temps d'attente est calculé à la fin de la simulation avec la formule :
  - `wait_time = finish - arrival_time - exec_time`
- Cette approche utilise le calcul final plutôt qu'une accumulation incrémentale pendant l'exécution

### 3.5. Exécuter le processus

- `remaining_time -= run` (décrémenter le temps restant)
- `time += run` (avancer le temps global de `run` unités)

## Étape 4 : Gestion des Nouveaux Arrivants Pendant le Quantum

- Vérifier si de nouveaux processus arrivent pendant l'exécution du quantum
- **Condition** : `arrival_time > (time - run)` ET `arrival_time <= time`
  - C'est-à-dire arrivés entre le début et la fin de ce quantum
- **Critères supplémentaires** :
  - `remaining_time > 0` (non terminé)
  - `end_time == -1` (pas complété)
  - Pas déjà présent dans la queue (même vérification que 2.1)
- Si toutes les conditions sont remplies : ajouter à `ready[tail++]`

## Étape 5 : Replacer ou Terminer le Processus

### 5.1. Si le processus n'est pas terminé (`remaining_time > 0`) :

- Le remettre **en fin de queue** : `ready[tail++] = curr`
- Il attendra son prochain tour (équité garantie)

### 5.2. Si le processus est terminé (`remaining_time == 0`) :

- Marquer `end_time = time` (temps de fin d'exécution)
- Incrémenter `completed++`
- **Ne pas remettre en queue**

## Étape 6 : Répéter jusqu'à Terminaison


- Répéter les étapes 2 à 5 tant que `completed < n`

## Étape 7 : Calcul des Statistiques Finales

Pour chaque processus (après terminaison de tous) :

- `finish = end_time` (temps de fin)
- `wait_time = finish - arrival_time - exec_time` (**formule exacte du temps d'attente**)
- Afficher : "Name Arrival Exec Finish Wait"
- Calculer `total_wait` (somme de tous les `wait_time`)
- Calculer `makespan = max(end_time)` (temps total de simulation)
- Afficher `Average Wait Time = total_wait / n`
- Afficher `Makespan`

## Choix Optimal du Quantum

Quantum	Impact CPU	Réactivité	Équité	Notes
1-2	Très élevé	Excellente	Parfaite	Overhead inacceptable
4 	Modéré	Bonne	Très bonne	<b>OPTIMAL TROUVÉ</b>
8	Bas	Moyenne	Bonne	Bon compromis aussi
16+	Minimal	Mauvaise	Basse	Devient comme FIFO

## Avantages et Inconvénients

Aspect	Évaluation
✓ ÉQUITÉ MAXIMALE ✓	Aucun processus attend indéfiniment
✓ Pas de famine	Tous progressent
✓ Idéal pour interactif	Bonne expérience utilisateur
✗ Overhead modéré	Context switches nombreux
✗ Quantum à tuner	Pas optimal pour tout workload

## Cas d'Usage Réel

**Linux** : CFS (Completely Fair Scheduler) basé sur ce principe. **Windows** : 20-100ms par processus selon priorité.

## 3.4 SRT (Shortest Remaining Time First - SRTF)

### Principe

Ordonnancement **préemptif** basé sur le **temps restant le plus court**. À chaque unité de temps, le processus avec le `remaining_time` minimum s'exécute. Si un processus plus court arrive, il **préempte immédiatement** le processus en cours.

## Algorithme de Sélection et Simulation

### Étape 1 : Initialisation

- Créer une copie des processus pour ne pas modifier l'original
- Pour chaque processus :
  - `remaining_time` = `exec_time` (temps restant à exécuter)
  - `end_time` = `-1` (marqueur de non-terminé)
- Initialiser le temps global à `0`
- Initialiser `completed` = `0` (nombre de processus terminés)

### Étape 2 : Boucle Principale de Simulation

À chaque unité de temps (`time`) :

#### 2.1. Rechercher le Processus avec le Temps Restant Minimum

Initialiser :

- `best` = `-1` (indice du meilleur processus)
- `min_rem` = `999999` (temps restant minimum trouvé)

Parcourir tous les processus :

- **Critères de sélection :**

- `arrival_time <= time` (processus déjà arrivé)
- `remaining_time > 0` (processus non terminé)

- **Logique de sélection :**

- Si `remaining_time < min_rem` → nouveau meilleur processus
- Si `remaining_time == min_rem` → départager par `arrival_time` (FIFO pour égalité)
  - Sélectionner celui avec `arrival_time` le plus petit
- Mettre à jour `min_rem` et `best`

## 2.2. Gestion de l'État IDLE

- Si `best == -1` (aucun processus prêt) :
  - CPU reste inactif (IDLE)
  - Afficher `[IDLE]`
  - Incrémenter `time` et continuer

## Étape 3 : Exécution du Processus Sélectionné

### 3.1. Affichage de l'état actuel

- Afficher le processus en cours d'exécution
- Afficher la ready queue avec les `remaining_time` de chaque processus en attente

### 3.2. Exécuter une unité de temps

- `remaining_time--` (décrémenter d'1 unité)
- `time++` (avancer le temps global)

## Étape 4 : Vérification de la Terminaison

- Si `remaining_time == 0` (processus vient de se terminer) :
  - Marquer `end_time = time` (temps de fin)
  - Incrémenter `completed`

## Étape 5 : Répéter

- Répéter les étapes 2 à 4 tant que `completed < n`

## Étape 6 : Calcul des Statistiques Finales

Pour chaque processus :

- `turnaround_time = end_time - arrival_time` (temps de rotation)
- `wait_time = turnaround_time - exec_time` (temps d'attente exact)
- Calculer la moyenne des temps d'attente
- Calculer le makespan (temps total de simulation)

## Avantages et Inconvénients

Aspect	Évaluation
--------	------------

Aspect	Évaluation
✓ Temps attente très bon	Résultats excellents
✓ Peu de préemptions	Comparé à Priority
✗ <b>FAMINE des longs processus</b> ⚠	Processus long jamais sélectionné
✗ Irréaliste en pratique	Pas possible en vrai système d'exploitation

### Cas d'Usage Réel

**Aucun en production** (requiert avenir). **Théorique uniquement.**

## 3.5 Multilevel Queue (Statique)

### Principe

Cet algorithme gère les processus en respectant une **hiérarchie stricte de priorité**, tout en assurant une équité entre les processus de même rang grâce au tourniquet (**Round-Robin**).

**Convention de priorité** : Grande valeur = Haute Priorité (ex: 10 > 1)

### Algorithme de Sélection (fonction `select_multilevel`)

#### Entrées :

- `procs[]` : tableau des processus
- `n` : nombre de processus
- `time` : temps actuel
- `current` : indice du processus actuellement en cours (-1 si aucun)
- `quantum_expired` : booléen indiquant si le quantum est expiré

#### Étape 1 : Identifier la Priorité MAXIMUM des Processus Prêts (convention : grande = haute)

##### Initialiser :

- `best_prio = -1` (très petite valeur, on cherche le maximum)
- `processes_ready = 0` (flag indiquant si au moins un processus est prêt)

##### Parcourir tous les processus :

- **Critères "Processus Prêt" :**
  - `arrival_time <= time` (déjà arrivé)
  - `remaining_time > 0` (pas encore terminé)
- Si processus prêt :
  - Si `priority > best_prio` → mettre à jour `best_prio` (on cherche la GRANDE valeur)
  - Marquer `processes_ready = 1`

Si aucun processus prêt (`processes_ready == 0`) → **Retourner -1 (CPU IDLE)**

## Étape 2 : Logique Round-Robin pour la Priorité MINIMUM

### 2.1. Vérifier si le processus courant peut continuer

Si **toutes** les conditions suivantes sont vraies :

- `current != -1` (un processus est en cours)
- `procs[current].remaining_time > 0` (pas encore terminé)
- `procs[current].priority == best_prio` (a toujours la meilleure priorité = même valeur grande)
- `procs[current].arrival_time <= time` (toujours valide)
- `!quantum_expired` (quantum non expiré)

→ **Retourner** `current` (continuer le même processus = stabilité)

### 2.2. Sinon, chercher le prochain candidat (Round-Robin circulaire)

- Calculer `start_index = (current + 1) % n` (commencer juste après le processus courant)
- Parcourir circulairement tous les processus à partir de `start_index`

Pour `i = 0` à `n-1`:

- `idx = (start_index + i) % n` (parcours circulaire)
- Si processus `idx` est prêt ET a la priorité `best_prio` (même priorité maximum) :
  - **Retourner** `idx` (prochain processus à exécuter)

Si aucun candidat trouvé → **Retourner -1**

### Avantages et Inconvénients

Aspect	Évaluation
✓ Priorités fixes = déterministe	Comportement prévisible
✓ Bon pour systèmes mixtes	Interactif + batch
✗ <b>FAMINE des basses priorités</b> ⚠	Prio 2 peut attendre indéfiniment
✗ Rigide	Pas d'adaptation aux changements

### Cas d'Usage Réel

Unix v7, BSD, System V (historique). **Problème** : Famine bien connue.

## 3.6 Multilevel Feedback Queue (Dynamique) ★ MODERNE

### Principe

La politique **Multilevel Dynamic** utilise la même fonction de sélection que Multilevel Static (`select_multilevel_dynamic`), mais implémente un **mécanisme d'aging continu** dans la boucle de simulation pour éviter la famine.

**Différence clé avec Multilevel Static :**

- **Statique** : Les priorités restent fixes toute la simulation
- **Dynamique** : Les priorités augmentent automatiquement pour les processus en attente (anti-famine)

### Algorithme de Sélection (fonction `select_multilevel_dynamic`)

#### Entrées :

- `procs[]` : tableau des processus
- `n` : nombre de processus
- `time` : temps actuel
- `current` : indice du processus actuellement en cours (-1 si aucun)
- `quantum_expired` : booléen indiquant si le quantum est expiré (`quantum_counter >= quantum`)

#### Logique de sélection :

##### Étape 1 : Trouver la priorité MAXIMUM parmi les processus prêts (convention : grande = haute)

- Initialiser `best_prio = -1` (valeur très petite)
- Parcourir tous les processus
- Si `arrival_time <= time` ET `remaining_time > 0` :
  - Si `priority > best_prio` → mettre à jour `best_prio` (on cherche la GRANDE valeur)
- Si aucun processus prêt → retourner -1 (IDLE)

##### Étape 2 : Continuer le processus courant si possible

- Si **toutes** les conditions suivantes sont vraies :
  - `current != -1` (un processus est en cours)
  - `procs[current].remaining_time > 0` (pas encore terminé)
  - `procs[current].priority == best_prio` (a toujours la meilleure priorité = même valeur grande)
  - `procs[current].arrival_time <= time` (toujours valide)
  - `!quantum_expired` (quantum non expiré)
- → Retourner `current` (continuer le même processus)

##### Étape 3 : Sinon, Round-Robin circulaire

- `start_index = (current + 1) % n`
- Parcourir circulairement de `start_index`
- Trouver le premier processus avec `priority == best_prio` (même priorité maximum)
- Retourner son indice (ou -1 si aucun)

### Implémentation du Feedback Loop (boucle de simulation)

La logique d'aging est implémentée dans `multilevel_dynamic_simulation()` du fichier `scheduler.c`.

#### Étape 1 : Initialisation

- `current = -1` (aucun processus en cours)
- `quantum_counter = 0` (compteur de quantum)
- `time = 0, finished = 0`



## Étape 2 : Boucle principale (tant que `finished < n`)

### 2.1. Sélection du processus

- Appeler `select_multilevel_dynamic(procs, n, time, current, quantum_counter >= quantum)`
- Si retourne -1 → CPU IDLE, incrémenter `time`, reset `quantum_counter = 0, current = -1`

### 2.2. Affichage de l'état

- Afficher le processus en cours d'exécution
- Afficher la ready queue avec format `"name:remaining_time"`

### 2.3. Aging dynamique (Anti-Famine) ★ CLEF

Pour **tous les processus en attente** (ceux qui NE sont PAS en cours d'exécution) :

- **Critères** : `i != idx` ET `arrival_time <= time` ET `remaining_time > 0`
- **Action** :
  - `priority++` (augmentation de priorité à chaque cycle)
  - `waiting_time++` (compteur d'attente)

**Mécanisme anti-famine :**

- Un processus en attente voit sa priorité augmenter **continuellement**
- Après suffisamment de cycles, il finira par atteindre la priorité maximum
- Il sera alors sélectionné par la fonction de sélection
- **Garantie** : Aucun processus ne peut attendre indéfiniment

### 2.4. Exécution du processus sélectionné

- `remaining_time--` (décrémenter d'1 unité)
- `current = idx` (marquer comme processus courant)
- `quantum_counter++` (incrémenter compteur de quantum)

### 2.5. Vérification de terminaison

- Si `remaining_time == 0` :
  - `end_time = time + 1`
  - `finished++`
  - `quantum_counter = 0` (reset)

### 2.6. Gestion du quantum expiré

- Si `quantum_counter >= quantum` :
  - `quantum_counter = 0` (reset pour permettre round-robin)
  - Le prochain appel à `select_multilevel_dynamic` aura `quantum_expired = true`
  - Permettra de passer au processus suivant de même priorité

### 2.7. Avancer le temps

- `time++`

Étape 3 : Statistiques finales

Afficher pour chaque processus :

- Name, Arrival, Exec, Finish, Wait
- **Final\_Prio** (priorité finale après aging)

Calculer :

- Average Wait Time
- Makespan

Avantages et Inconvénients

Aspect	Évaluation
✓ <b>Anti-famine</b>	Aging garantit personne n'attend indéfiniment
✓ <b>Adaptation dynamique</b>	S'ajuste au comportement processus
✓ <b>Équitable</b>	Meilleur que multilevel statique
✓ <b>Moderne</b>	Inspiré Linux CFS réel
⚠ <b>Complexité accrue</b>	Plus de compteurs et conditions
⚠ <b>Moins déterministe</b>	Feedback rend résultats moins prévisibles

4. Technologies et Architecture

4.1 Stack Technologique Complet

A. Frontend (Web UI Moderne)

Technologie	Version	Rôle	Justification
<b>Next.js</b>	16.0.3	Framework React SSR + routing	Production-ready, performance optimale
<b>React</b>	19.2.0	UI library composants	Modern hooks, state management
<b>TypeScript</b>	^5	Typage static	Sécurité types, refactoring facile
<b>Tailwind CSS</b>	^4.1.9	Utility-first CSS	Styling rapide, responsive design
<b>Radix UI</b>	~1.x	Composants accessibles	Boutons, dialogs, tabs, forms
<b>Recharts</b>	latest	Visualisation données	Gantt, pie/bar charts interactifs
<b>Sonner</b>	^1.7.4	Toast notifications	Feedback utilisateur
<b>next-themes</b>	^0.4.6	Dark/Light theme	Thème UI persistant
<b>zod</b>	3.25.76	Validation schémas	Validation configs uploads

Technologie	Version	Rôle	Justification
React Hook Form	^7.60.0	Gestion formulaires	Forms performantes

B. Backend (Moteur Simulation C)

Technologie	Version	Rôle	Justification
C	C11	Langage implémentation	Requis, bas niveau, performance
GCC	9.4.0	Compilateur	Standard Linux, optimisé
Make	4.2.1	Build automation	Compilation incrémentale
POSIX	Standard	API système	Portabilité Linux/Unix

C. Build & Package Management (Frontend)

Technologie	Version	Rôle	Justification
pnpm	10.24.0	Package manager	Plus rapide/efficace que npm
Node.js	20.19.6	Runtime JavaScript	Exécution Next.js, APIs
npm/corepack	10.8.2	Package install	Gestion dépendances
Turbopack	Intégré	Bundler Next.js	Compilation ultra-rapide
PostCSS	^8.5	CSS processing	Tailwind compilation

D. Tooling & Développement

Technologie	Version	Rôle	Justification
VS Code	Latest	IDE principal	Léger, plugins C/TypeScript/Git
Git	2.x	Contrôle version	Collaboration, historique
GitHub	Cloud	Dépôt hébergé	Collaboration équipe, CI/CD
ESLint	8.x (optionnel)	Linting JS	Qualité code (voir README)

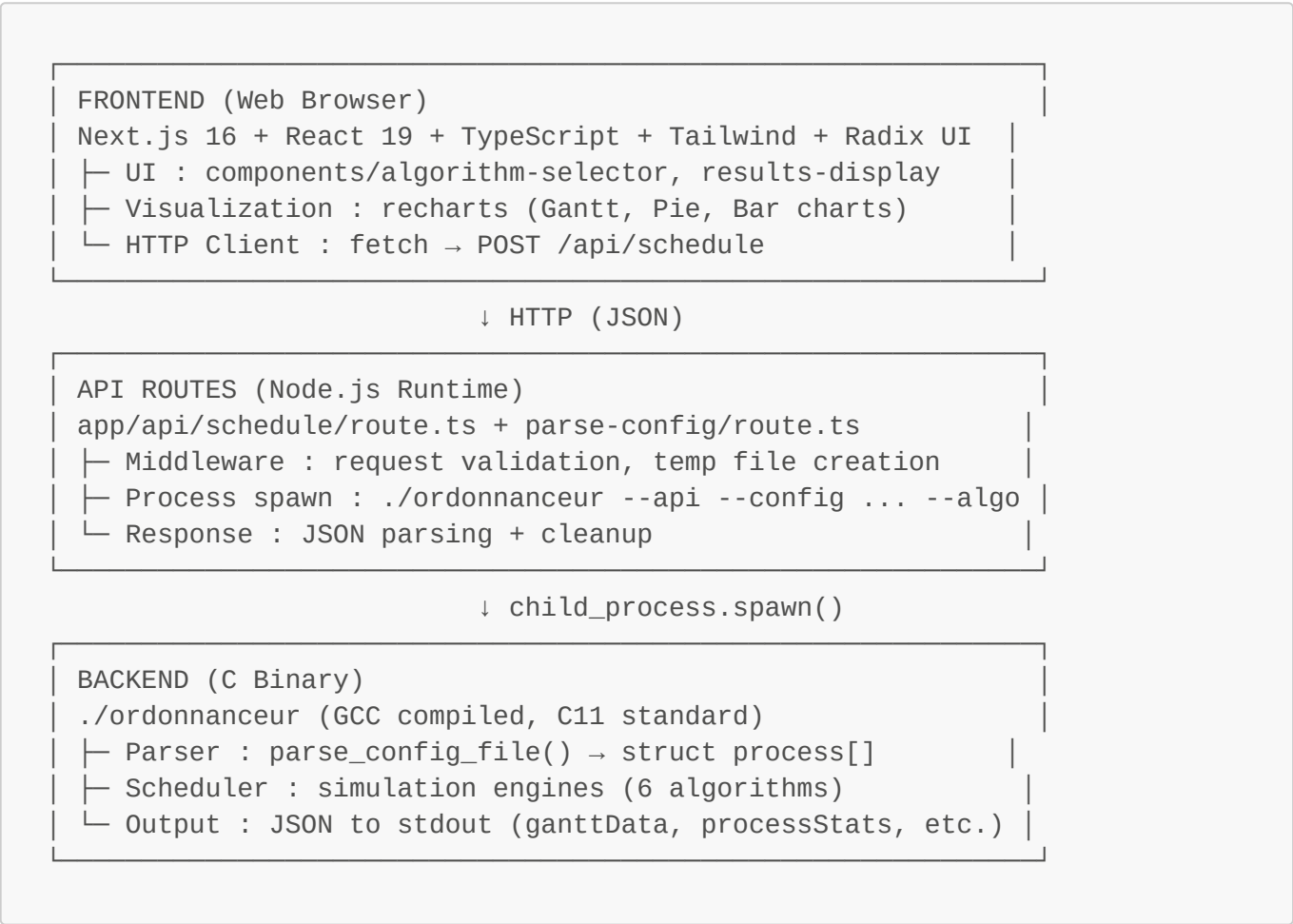
E. Architecture & Infrastructure

Technologie	Rôle	Détails
Docker (optionnel)	Containerisation	Reproductibilité environnement
Linux/WSL2	OS cible	Ubuntu 20.04 sur WSL/VM
Port 3000	Next.js dev server	http://localhost:3000
Child Process Spawn	Backend invocation	Node.js exécute <code>./ordonnanceur --api</code>

F. Interaction Frontend ↔ Backend

Composant	Technologie	Communication
Frontend (React)	Next.js + TypeScript	HTTP POST requests
API Route (Node.js)	app/api/*/route.ts	child_process.spawn()
Backend Binary (C)	./ordonnanceur --api	stdout JSON
Response	JSON over HTTP	JSON.parse() → React render

Tableau Récapitulatif - Stack par Couche



```
"node": "20.19.6"
}
```

### Backend Stack :

```
C Compiler : gcc 9.4.0
Build Tool : make 4.2.1
C Standard : C11
Target OS  : Linux (any POSIX-compliant)
```

### Development Environment :

```
OS      : Ubuntu 20.04 LTS (or WSL2)
IDE     : VS Code
VCS     : Git 2.x + GitHub
Runtime : Node.js 20.19.6 (frontend)
         C runtime (backend)
```

## 4.2 Architecture du Projet

### Architecture Hybride : Next.js (Frontend) + C (Backend)

```
Projet-Ordonnancement-Linux/
|
├── FRONTEND (Next.js 16 + React 19 + TypeScript)
|   |
|   ├── app/                                # Next.js App Router
|   |   ├── page.tsx                        # Page principale UI
|   |   |   (upload/generate config, select algo, launch)
|   |   ├── layout.tsx                     # Layout racine (metadata,
|   |   |   ThemeProvider)
|   |   ├── globals.css                     # Styles Tailwind globaux
|   |   └── api/                            # API Routes Next.js (Node.js
|   |       runtime)
|   |       ├── schedule/
|   |       |   └── route.ts                # POST /api/schedule → lance
|   |       |   ordonnanceur --api
|   |       |   └── parse-config/
|   |       |       └── route.ts            # POST /api/parse-config → parse
|   |       |   fichier config
|   |       └── components/                 # React Components
|   |           ├── algorithm-selector.tsx # Dropdown algorithmes + params
|   |           |   (quantum, priorityOrder)
|   |           ├── file-generation-dialog.tsx # Dialog génération config (nb
|   |           |   processus, ranges)
|   |           └── results-display.tsx     # Gantt interactif + Pie/Bar charts
```

```

+ Table stats
|   |   | theme-provider.tsx      # Dark/Light theme (next-themes)
|   |   | └─ ui/                  # Radix UI primitives (button,
card, dialog, tabs, etc.)
|   |   |
|   |   | └─ lib/                  # Utilitaires TypeScript
|   |   |   └─ types.ts            # Interfaces (Process,
AlgorithmConfig, SchedulingResult, etc.)
|   |   |     └─ utils.ts          # Helpers (cn pour classnames)
|   |   |
|   |   | └─ hooks/                # Hooks React personnalisés
|   |   |   └─ use-toast.ts        # Toast notifications (sonner)
|   |   |     └─ use-mobile.ts     # Responsive breakpoint detection
|   |   |
|   |   | └─ public/              # Assets statiques (images, fonts,
etc.)
|   |   |
|   |   | └─ next.config.mjs       # Config Next.js (typescript,
images)
|   |   |   └─ tsconfig.json       # Config TypeScript
|   |   |   └─ postcss.config.mjs  # PostCSS + Tailwind
|   |   |   └─ components.json     # Shadcn UI config
|   |   |   └─ package.json        # D  ps (next, react, recharts,
radix-ui, etc.) + scripts
|   |   |   └─ pnpm-lock.yaml      # Lock file pnpm
|   |   |     └─ node_modules/     # D  pendancies install  es
|   |   |
|   |   | └─ BACKEND C (Moteur simulation)
|   |   |
|   |   |   └─ src/                # Code source C
|   |   |     └─ main.c            # Point d'entr  e (modes:
interactif, --api, --parse-config)
|   |   |   └─ scheduler.c         # Moteur simulation + simulations
(fifo_simulation, etc.)
|   |   |     └─ parser.c          # Parse fichiers config
(parse_config_file)
|   |   |       └─ generate_config.c # G  n  re configs al  atoires
(generate_config)
|   |   |         └─ utils.c       # Utilitaires (display_config_file,
etc.)
|   |   |
|   |   |   └─ include/           # Headers C
|   |   |     └─ process.h        # struct process, NAME_LEN,
READY/RUNNING/BLOCKED/ZOMBIE
|   |   |       └─ scheduler.h     # Prototypes simulations, struct
simulation_result
|   |   |         └─ parser.h      # parse_config_file,
validate_config
|   |   |           └─ generate_config.h # generate_config
|   |   |             └─ utils.h    # display_config_file, etc.
|   |   |
|   |   |   └─ policies/          # Algorithmes ordonnancement
(fonctions s  lection)
|   |   |     └─ fifo.c           # int fifo_scheduler(process*, int,

```

```

int, int, int)
|   |   |─ priority_preemptive.c    # int priority_preemptive(...)
|   |   |─ roundrobin.c            # void round_robin(process*, int,
int)
|   |   |─ srt.c                    # void srt_simulation(process*,
int)
|   |   |─ multilevel.c              # int select_multilevel(...)
|   |   |─ multilevel_dynamic.c     # int
select_multilevel_dynamic(...)
|   |   |─ tests/                    # Tests unitaires C
|   |   |   |─ test_fifo.c
|   |   |   |─ test_priority.c
|   |   |   |─ test_roundrobin.c
|   |   |   |─ test_multilevel.c
|   |   |   |─ test_multilevel_dynamic.c
|   |   |   |─ test_parser.c
|   |   |   |─ testfile.txt
|   |   |─ build/                    # Fichiers objets .o (généré par
make)
|   |   |─ ordonnanceur              # Binaire Linux compilé
|   |   |─ ordonnanceur.exe          # Binaire Windows cross-compilé
|   |   |─ Makefile                  # Build system (all, clean,
mrproper)
|   |   |─ tests/*.c                # Sources de tests unitaires
(binares test_* non trackés)
|   |   |─ CONFIGURATION & DONNÉES
|   |   |   |─ config/
|   |   |   |   |─ sample_config.txt  # Fichiers configuration processus
|   |   |   |   |─ config_*.txt      # Exemple par défaut (7 processus)
|   |   |   |   |─ sample_config_*.txt # Configs générées (timestamp)
|   |   |   |   |                     # Configs générées par web UI
|   |   |   |─ DOCUMENTATION
|   |   |   |   |─ Documentation.md   # Documentation technique complète
|   |   |   |   |                     (ce fichier)
|   |   |   |   |─ README.md          # Guide utilisateur + install +
usage
|   |   |   |   |─ API_REFERENCE.md    # Référence API Next.js
|   |   |   |   |─ ARCHITECTURE.md    # Architecture détaillée
|   |   |   |   |─ PROFESSIONAL_README.md # README professionnel
|   |   |   |   |─ PROJECT_SUMMARY.txt # Résumé projet
|   |   |   |   |─ INDEX.md           # Index navigation docs
|   |   |   |   |─ LICENSE            # MIT License
|   |   |   |─ CONFIGURATION RACINE
|   |   |   |   |─ .git/               # Dépôt Git local
|   |   |   |   |─ .gitignore          # Ignore (node_modules, build,
.next, etc.)

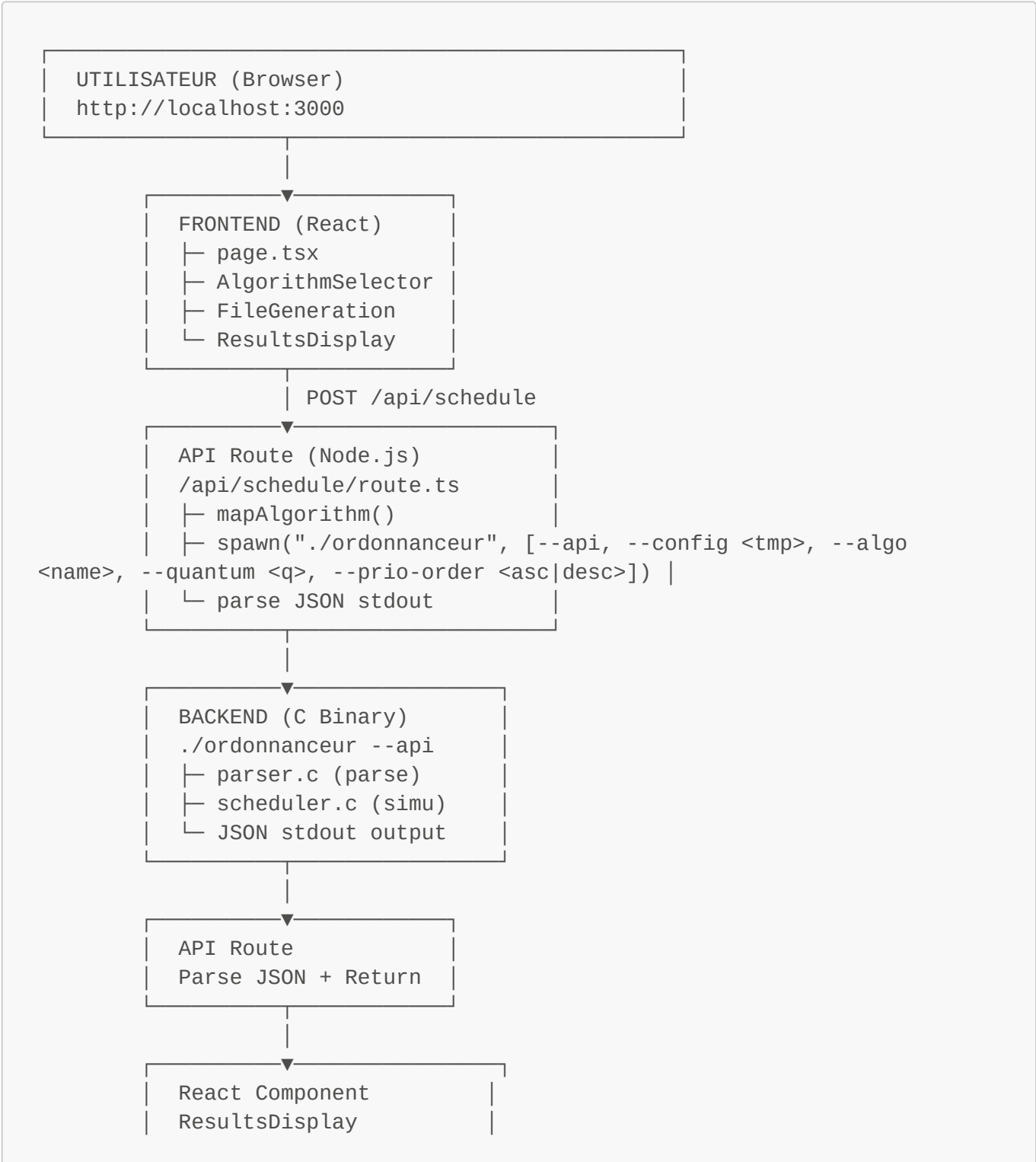
```

```
├── .next/                                # Cache Next.js (dev/production)
├── .vscode/                             # Settings VS Code
└── (aucun binaire de test tracké)      # Les binaires
test_* sont ignorés par .gitignore
```

Dépôt GitHub :

- URL : <https://github.com/arijsebai/Projet-Ordonnancement-Linux.git>
- Branch active : `dev`
- Remote : `origin`

Structure Logique par Rôle





- └─ Gantt chart
- └─ Pie/Bar charts
- └─ Detailed table

4.3 Backend C : Modes d'Opération

Le backend C (`ordonnanceur`) supporte **3 modes** :

Mode	Commande	Output	Cas d'Usage
Interactif	<code>./ordonnanceur</code>	Texte + Gantt textuel	CLI local, démo
Direct File	<code>./ordonnanceur config.txt</code>	Texte + Gantt textuel	Scripts shell
API	<code>./ordonnanceur --api --config ... -- algo ...</code>	JSON structuré	Routes Next.js
Parse Only	<code>./ordonnanceur --parse-config config.txt</code>	JSON array	Validation fichiers

4.4 Intégration Frontend ↔ Backend

**Flow complet : User → React → API Route → C Backend → Results**

```
User (Browser)
  ↓ Sélectionne algorithme + paramètres
React Component (page.tsx)
  ↓ POST /api/schedule avec { processes, config }
API Route (app/api/schedule/route.ts)
  ↓ Écrit fichier temp + spawn("./ordonnanceur --api ...")
C Backend (ordonnanceur)
  ↓ Parse → Simulate → JSON stdout
API Route
  ↓ Parse JSON + cleanup temp file
React Component (ResultsDisplay)
  ↓ Affiche Gantt + Charts + Table
```

**Routes API Next.js :**

- 1. **POST /api/parse-config** : Upload fichier `.txt` → retourne array processus
- 2. **POST /api/schedule** : Lance simulation → retourne résultats complets

4.5 Flow d'Exécution Complet

4.5 Visualisations Frontend

**Components React :**

### 1. AlgorithmSelector ([components/algorithm-selector.tsx](#))

- Dropdown 6 algorithmes
- Paramètres dynamiques : quantum (RR, Multilevel Dynamic), priorityOrder (Priority)

### 2. ResultsDisplay ([components/results-display.tsx](#))

- **Gantt chart** : timeline interactif (play/pause, step, zoom)
- **Pie chart** : répartition CPU par processus
- **Bar chart** : temps attente vs temps total
- **Table** : stats détaillées (arrival, execution, wait, finish, priority)
- **Couleurs** : 20 couleurs distinctes, déterministe par process ID

### 3. FileGenerationDialog ([components/file-generation-dialog.tsx](#))

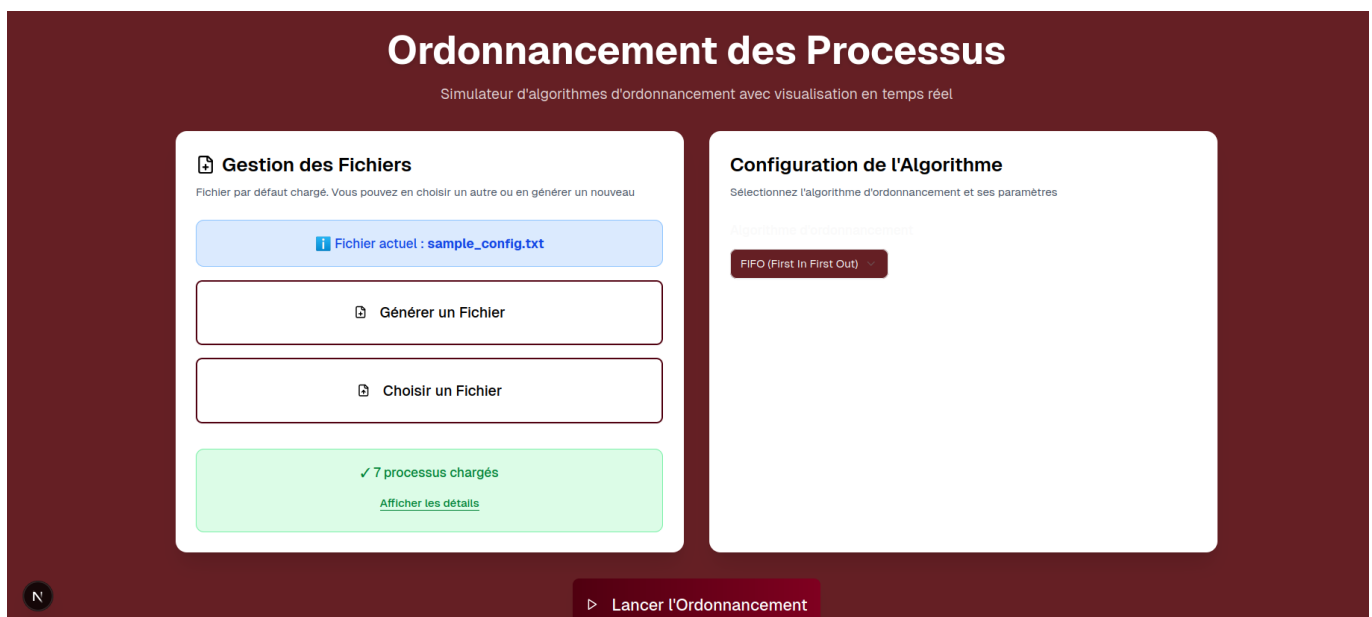
- Dialog création config (nb processus, ranges arrival/exec/priority)

## Captures d'écran (placeholders)

Placez les fichiers suivants dans [public/](#) et ils seront référencés dans la documentation et le README :

- [public/home.png](#) — Page d'accueil avec AlgorithmSelector et FileGenerationDialog
- [public/gantt.png](#) — Vue Résultats avec diagramme de Gantt
- [public/processus.png](#) — Vue Résultats avec diagramme d'état des processus
- [public/tableau.png](#) — Vue Résultats avec tableau
- [public/bar.png](#) — Vue Résultats avec Bar charts
- [public/cercle.png](#) — Vue Résultats avec Pie charts

Aperçu intégré :



# Ordonnancement des Processus

Simulateur d'algorithmes d'ordonnancement avec visualisation en temps réel

## Résultats de l'Ordonnancement

Algorithme: FIFO

Nouvelle Simulation

### Diagramme de Gantt

Occupation CPU & File d'attente

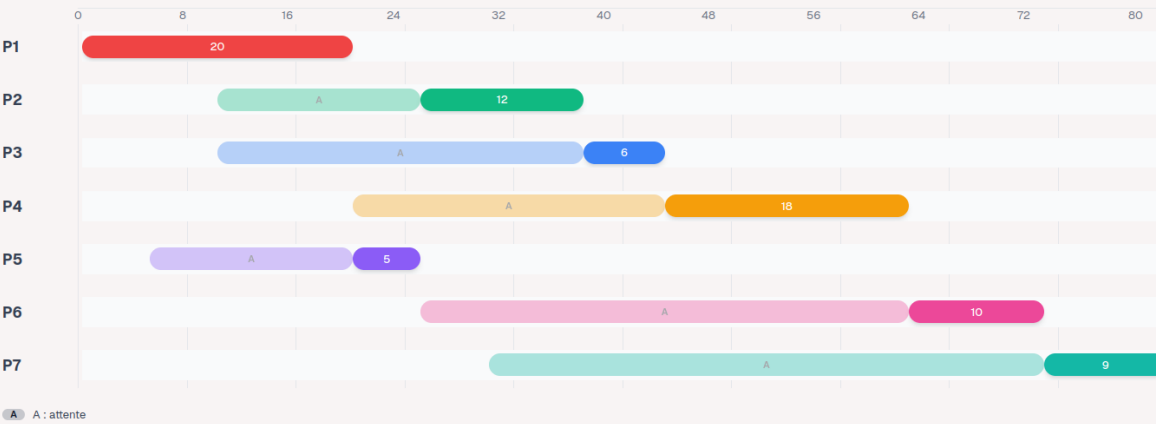
Navigation: Lecture, -1s, +1s, T = 80s



### Diagramme de l'état des processus

Visualisation temporelle de l'exécution des processus

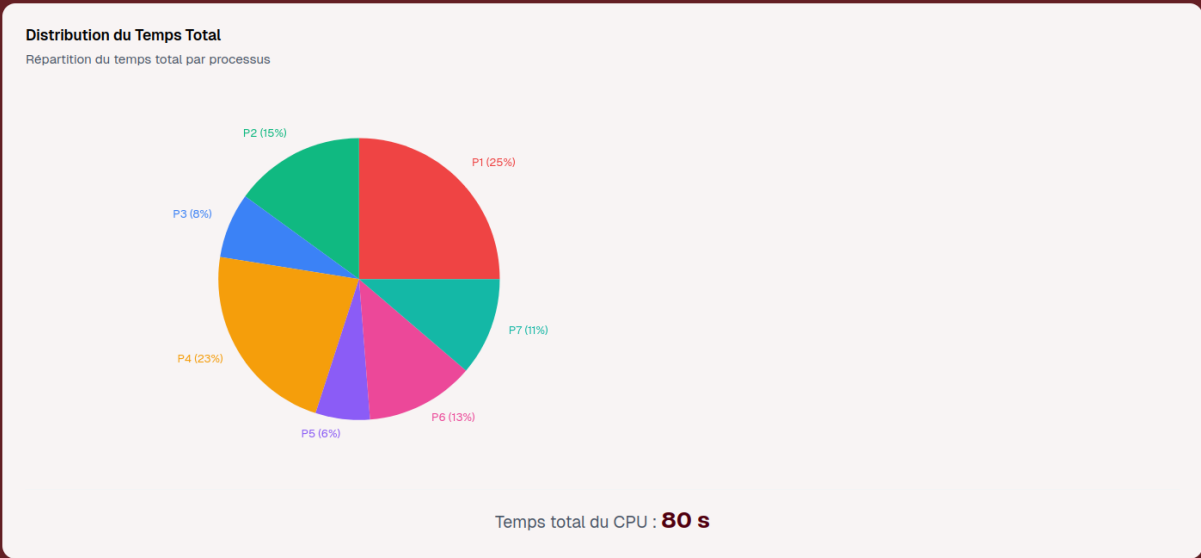
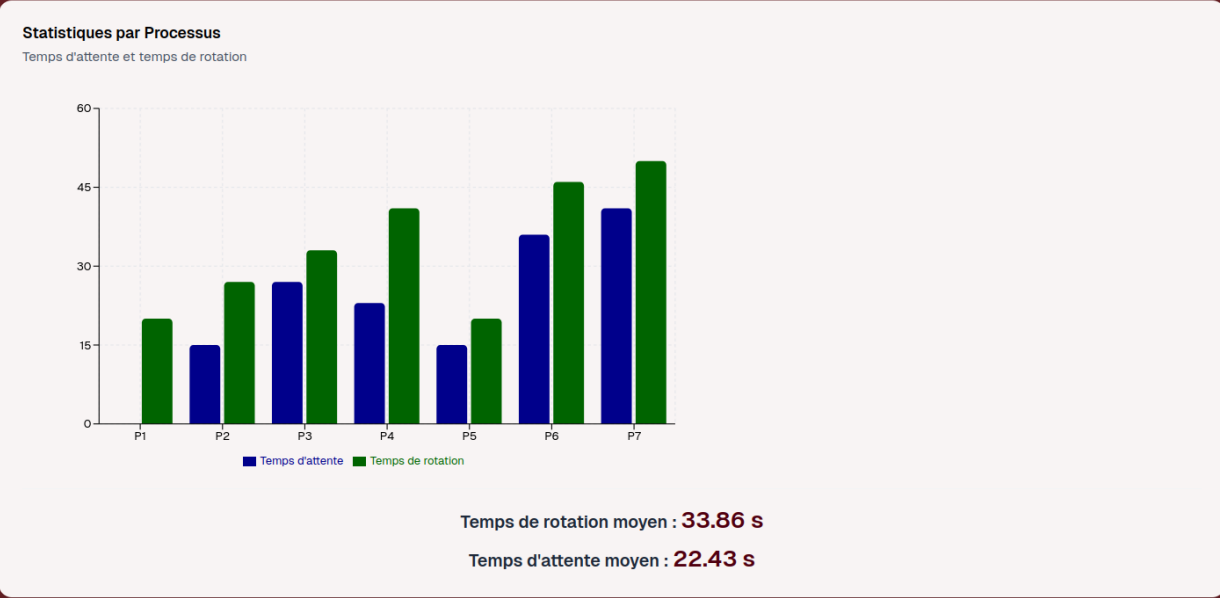
Navigation: Lecture, Précédent (-1s), Suivant (+1s), T = 80s



### Tableau Détaillé des Processus

Détails complets de l'exécution pour chaque processus

NOM DU PROCESSUS	TEMPS D'ARRIVÉE	TEMPS D'EXÉCUTION	PRIORITÉ INITIALE	TEMPS DE FIN	TEMPS D'ATTENTE
P1	0	20	3	20	0
P2	10	12	2	37	15
P3	10	6	4	43	27
P4	20	18	1	61	23
P5	5	5	5	25	15
P6	25	10	2	71	36
P7	30	9	3	80	41



4.6 Mapping Algorithmes Frontend → Backend

5. Déroulement du Développement SCRUM

5.1 Organisation Équipe

Rôle	Responsable(s)
Product Owner	Mme Yosra Najar
Scrum Master	Arij Sebai
Développeuses	Aya Sakroufi, Balkis Hanafi, Hadil Hasni, Wiem Ayari

5.2 Paramètres Scrum

Paramètre	Valeur
Durée totale	5 semaines

Paramètre	Valeur
Durée sprint	2 semaine (12 jours ouvrables)
Nombre sprints	2 sprints
Réunions	Planning (1h), Daily (15min), Review (1h), Retro (45min)
Total Story Points	~180 SP

5.3 Product Backlog

ID	User Story	Priorité
1	En tant qu'utilisateur, je veux lire un fichier de configuration contenant les processus (nom, arrivée, durée, priorité)	Moyenne
2	En tant que développeur, je veux un Makefile fonctionnel	Haute
3	En tant qu'utilisateur, je veux simuler un ordonnancement FIFO	Moyenne
4	En tant qu'utilisateur, je veux simuler un ordonnancement Round Robin	Haute
5	En tant qu'utilisateur, je veux simuler un ordonnancement à priorité préemptive	Haute
6	En tant qu'utilisateur, je veux voir les résultats sur la console (temps d'attente, temps de retour, Gantt textuel)	Moyenne
7	En tant qu'utilisateur, je veux choisir dynamiquement l'algorithme d'ordonnancement	Moyenne
8	En tant qu'utilisateur, je veux une politique multilevel avec aging	Haute
9	En tant qu'utilisateur, je veux une politique SRT (Shortest Remaining Time)	Haute
10	En tant qu'utilisateur, je veux automatiser la génération d'un fichier de configuration	Moyenne
11	En tant qu'utilisateur, je veux un affichage graphique (diagramme de Gantt)	Haute
12	En tant qu'utilisateur, je veux une interface graphique simple (IHM)	Haute

5.4 Réunion de Lancement (Sprint 0)

**Objectif :** Préparer le projet et établir les fondations

**Tâches essentielles :**

1. Lire et comprendre le sujet

- Analyser les spécifications du projet
- Identifier les cas d'usage
- Clarifier les ambiguïtés

2. Identifier les fonctionnalités minimales et avancées

- **Minimales** : FIFO, Priority, RR, affichage console
- **Avancées** : Multilevel, SRT, Gantt graphique, IHM

### 3. Créer le dépôt GitHub + choisir la licence (MIT)

- Initialiser git local
- Créer dépôt GitHub
- Ajouter LICENSE MIT
- Configurer .gitignore
- Premier commit

## 5.5 Sprint Backlog 1

**Objectif** : Implémenter ordonnanceurs de base et infrastructure

#	Tâche	Charge	Estimation
1	Conception du fichier de configuration des processus	3 pts	4.5 h
3	Développement de la politique FIFO	5 pts	7.5 h
4	Développement de Round Robin (gestion du quantum)	8 pts	12 h
5	Développement de la politique à priorité préemptive	8 pts	12 h
2	Création du Makefile (build / clean)	4 pts	6 h
6	Affichage textuel des résultats (temps d'attente, temps de retour, Gantt textuel)	3 pts	4.5 h
7	Initialisation du dépôt GitHub	1 pt	1.5 h
10	Ajout d'exemples de tests simples	2 pts	3 h

**Total Sprint 1 : 34 points (50.5 heures)**

## 5.6 Sprint Backlog 2

**Objectif** : Implémenter algorithmes avancés et interface utilisateur

Tâche	Description	Charge	Priorité	Estimation
<b>Multilevel + Aging</b>	Ajouter un ordonnancement multi-files avec mécanisme d'aging	8 pts	Haute	12 h
<b>SRT</b>	Version préemptive de SJF (gestion du temps restant)	8 pts	Haute	12 h
<b>Génération Config</b>	Script/programme produisant un fichier valide automatiquement	4 pts	Moyenne	6 h
<b>IHM + Gantt</b>	IHM basique + génération d'un diagramme de Gantt visuel	12 pts	Haute	18 h

**Total Sprint 2 : 32 points (48 heures)**

## 5.7 Métriques SCRUM - Sprints 0, 1, 2

Récapitulatif Charges

Sprint	Objectif	Points	Heures	Tâches
Sprint 0	Réunion de lancement	N/A	3	3
Sprint 1	FIFO + Foundation	34	50.5	8
Sprint 2	Algorithmes avancés	32	48	4
TOTAL		66	101.5	15

6. Spécifications Techniques : Point d'Entrée, Parser et Générateur

6.1 Point d'Entrée (main.c) : Modes Interactif et API

Vue d'ensemble des Modes d'Opération

Le backend C (`ordonnanceur`) supporte **3 modes d'opération** :

Mode	Commande	Utilisateur	Output	Cas d'Usage
Interactif	<code>./ordonnanceur</code>	Humain	Texte + Gantt textuel	CLI local
Direct File	<code>./ordonnanceur [fichier]</code>	Humain	Texte + Gantt textuel	Script shell rapide
API	<code>./ordonnanceur --api --config ... --algo ...</code>	Programme/Script	JSON structuré	Routes Next.js
Parse Only	<code>./ordonnanceur --parse-config [fichier]</code>	Programme/Script	JSON array	Validation fichiers

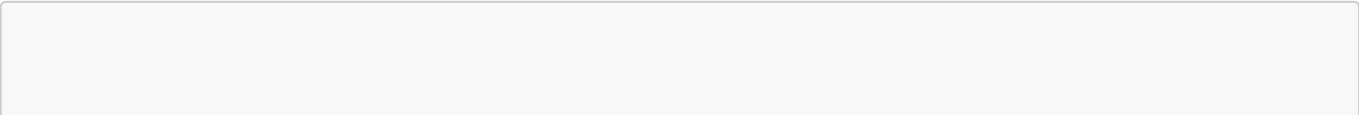
Mode 1 : CLI Interactif (Menu Principal)

Étapes du Programme Principal (Mode Interactif)

Étape 0 : Détection du Mode d'Opération

À la première ligne de `main()` :

- Si aucun argument (`argc == 1`) → Mode Interactif (menu)
- Si un argument non-flag (`argc == 2` et `argv[1][0] != '-'`) → Mode Direct File (fichier passé directement)
- Si flags détectés (`--api`, `--parse-config`, `--config`) → Mode API (sortie JSON)



```
if (argc == 2 && argv[1][0] != '-') {
    direct_file_mode = 1; // Mode: ./ordonnanceur config.txt
    strncpy(filename, argv[1], sizeof(filename) - 1);
}
```

Ensuite, parcourir tous les arguments pour capturer les flags API :

```
for (int i = 1; i < argc; i++) {
    if (strcmp(argv[i], "--api") == 0) { api_mode = 1; }
    else if (strcmp(argv[i], "--parse-config") == 0) { parse_only = 1; }
    else if (strcmp(argv[i], "--config") == 0) { /* read filename */ }
    else if (strcmp(argv[i], "--algo") == 0) { /* read algo name */ }
    else if (strcmp(argv[i], "--quantum") == 0) { /* read quantum */ }
    else if (strcmp(argv[i], "--prio-order") == 0) { /* read asc|desc */ }
}
```

### Étape 1 : Affichage du Menu Interactif

- Afficher le titre : `=== Scheduler Project ===`
- Afficher les deux options :
  - Option 1 : "Generate configuration file automatically (default)"
  - Option 2 : "Use an existing configuration file"
- Demander le choix de l'utilisateur : `Your choice (press ENTER for default):`
- Utiliser `fgets()` pour lire l'entrée (sûr contre débordement de buffer)
- **Validation :**
  - Si entrée vide (juste ENTER) → choix par défaut = 1
  - Si entrée = "1" ou "2" → utiliser ce choix
  - Sinon → avertissement et défaut = 1

### Étape 2 : Gestion du Choix 1 (Générer Configuration)

- Récupérer timestamp système :
  - Appeler `time(NULL)` pour obtenir temps actuel
  - Appeler `localtime()` pour convertir en structure `tm`
  - Utiliser `strftime(format, ...)` avec pattern `"%Y%m%d_%H%M%S"` (ex: `20251206_143022`)
- Construire le chemin complet :
  - Format : `"config/sample_config_TIMESTAMP.txt"`
  - Exemple : `config/sample_config_20251206_143022.txt`
  - Utiliser `snprintf()` pour formater de manière sûre
- Appeler `generate_config(filename)` :
  - Passe le chemin au générateur
  - Si retourne 0 → succès
  - Si retourne erreur → afficher message d'erreur et quitter (return 1)



### Étape 3 : Gestion du Choix 2 (Charger Fichier Existant)

- Demander : `Enter configuration file name (with path if needed):`
- Lire le nom du fichier avec `scanf("%255s", filename)` :
  - Limite : 255 caractères (sécurité buffer)
  - Accepte chemins avec sous-répertoires (ex: `config/sample_config.txt`)
- **Nettoyage du buffer stdin :**
  - Après `scanf()`, le caractère newline reste dans le buffer
  - Boucle de nettoyage : `while ((c = getchar()) != '\n' && c != EOF);`
  - Essentiel avant utilisation de `fgets()` ultérieurement

### Étape 4 : Affichage du Fichier de Configuration

- Afficher message : `Loading configuration file: <filename>`
- Appeler `display_config_file(filename)` pour afficher le contenu brut du fichier
- Permet à l'utilisateur de vérifier avant parsing

### Étape 5 : Parsing et Chargement des Processus

- Allouer un pointeur : `struct process *list = NULL`
- Initialiser compteur : `int n = 0`
- Appeler `parse_config_file(filename, &list, &n)` :
  - Remplit le tableau `list` avec les processus trouvés
  - Remplit `n` avec le nombre de processus chargés
  - Retourne 0 si succès, erreur sinon
- Si erreur (return != 0) :
  - Afficher message d'erreur
  - Quitter (return 1)
- Afficher succès : `✓ N processes loaded.`

### Étape 6 : Chargement des Politiques d'Ordonnancement

- Appeler `load_policies()` :
  - Initialise la liste des politiques disponibles
  - Enregistre les fonctions de sélection (FIFO, Priority, RR, SRT, Multilevel, etc.)

### Étape 7 : Menu de Sélection de Politique

- Appeler `choose_policy()` :
  - Affiche les politiques disponibles avec numéros
  - Demande à l'utilisateur de choisir
  - Retourne l'indice de la politique choisie

### Étape 8 : Lancer la Simulation

- Appeler `run_scheduler(list, n, policy)` :
  - Lance la simulation avec les processus et la politique choisis
  - Orchestre la boucle temps dans `scheduler.c`
  - Affiche les résultats (tableau, statistiques, Gantt)

## Étape 9 : Libération Mémoire et Terminaison

- Appeler `free(list)` pour libérer le tableau de processus
  - Retourner 0 (succès)
- 

## Mode 2 : Direct File Mode (Fichier en Arguments)

**Comportement :** `./ordonnanceur config/sample_config.txt`

### Différence avec Mode Interactif :

- Saute le menu initial
- Charge directement le fichier fourni en argument
- Affiche le contenu du fichier
- Affiche le menu de sélection de politique
- Exécute la simulation et affiche résultats (texte + Gantt)

**Avantage :** Utile pour scripts shell automatisés sans intervention utilisateur.

## Mode 3 : API Mode (Mode Programmable JSON)

**Comportement :** `./ordonnanceur --api --config <file> --algo <algo> [--quantum <q>] [--prio-order <asc|desc>]`

**Ou en cas de Parse Only :** `./ordonnanceur --parse-config <file>`

### Différence avec Modes Interactifs :

- Aucune interaction utilisateur
- Sortie **UNIQUEMENT** JSON structuré sur stdout
- Pas d'affichage de menu, pas de Gantt textuel
- Erreurs en JSON format (pour faciliter parsing)
- Conçu pour appels programmatiques

### Étapes Internes (Mode API) :

1. **Parsing des flags** (déjà fait à Étape 0)
2. **Vérification si parse\_only :**
  - Si oui : parser le fichier → retourner JSON array des processus → terminer
  - Si non : continuer au scheduler
3. **Vérification si api\_mode :**
  - Si non : mode interactif classique
  - Si oui : continuer mode API
4. **Chargement de la configuration :**
  - Appeler `parse_config_file(config_path, &list, &n)`

- Si erreur : `printf("{\"error\":\"Failed to parse config\"}\n")`

## 5. Création de la structure d'options :

```
struct scheduler_options opts = {
    .algorithm = algo,           // "fifo", "priority", "roundrobin",
    etc.
    .quantum = quantum,         // pour RR et multilevel_dynamic
    .prio_mode = prio_mode      // 0 = asc, 1 = desc
};
```

## 6. Appel au scheduler mode API :

- Appeler `run_scheduler_api(list, n, &opts, &result)`
- Cette fonction remplit `result` avec :
  - `gantt_segment[]` : allocation CPU par temps
  - `process_stat[]` : statistiques par processus
  - `average_wait, makespan` : métriques globales

## 7. Sortie JSON :

- Appeler `print_json_result(&result)`
- Affiche JSON structuré sur stdout
- API route Next.js parse ce JSON

## 6.2 Format Fichier Configuration

### Syntaxe Générale

Chaque ligne représente soit :

- Un **processus valide** : 4 champs séparés par espaces ou tabulations
- Une **ligne vide** : ignorée
- Un **commentaire** : ignoré

### Ordre des Champs (Obligatoire)

Position	Champ	Type	Contraintes
1	<code>name</code>	Chaîne	Sans espaces (ex: P1, processA)
2	<code>arrival_time</code>	Entier	$\geq 0$
3	<code>exec_time</code>	Entier	$> 0$ (strictement positif)
4	<code>priority</code>	Entier	Intervalle selon contexte

### Règles Commentaires

- **Commentaire entier** : Ligne commençant par `#` → ignorée complètement
- **Commentaire en fin de ligne** : Tout ce qui suit `#` → ignoré

### Exemple Complet

```
# Configuration exemple processus
P1 0 250 3      # Processus 1, arrive t=0, durée 250ms, prio 3
P2 10 100 1     # Processus 2, arrive t=10, durée 100ms, prio 1
P3 20 150 0     # Processus 3, arrive t=20, durée 150ms, prio 0

# Ligne vide ci-dessus = ignorée

P4 20 50 5      # Valide
# P5 25 75 2    # Commentaire entier → ignoré complètement
P6 30 200 2 # Commentaire fin ligne → ignoré

P7  40  100 1   # Tabulations acceptées
```

## Algorithme de Parsing Détaillé

### Étape 1 : Initialisation

- Ouvrir le fichier de configuration en mode lecture
- Allouer un tableau dynamique de processus (capacité initiale : 16 éléments)
- Initialiser compteur de processus à 0
- Initialiser numéro de ligne à 0

### Étape 2 : Lecture ligne par ligne

Pour chaque ligne du fichier :

#### 2.1. Pré-traitement de la ligne

- Supprimer le caractère de fin de ligne `\n` si présent
- Identifier le premier caractère non-blanc
- Si la ligne est entièrement vide → ignorer et passer à la suivante
- Si le premier caractère est `#` → ligne commentaire complète, ignorer

#### 2.2. Traitement des commentaires en fin de ligne

- Chercher le caractère `#` dans la ligne
- Si trouvé : tronquer la ligne à cette position (tout après `#` est ignoré)
- Résultat : seule la partie avant `#` est conservée

#### 2.3. Tokenisation (découpage)

- Utiliser la fonction de tokenisation pour découper la ligne selon délimiteurs : espace et tabulation
- Extraire jusqu'à 4 tokens maximum :
  - Token 0 : `name` (chaîne de caractères)
  - Token 1 : `arrival_time` (chaîne à convertir en entier)
  - Token 2 : `exec_time` (chaîne à convertir en entier)
  - Token 3 : `priority` (chaîne à convertir en entier)
- Si moins de 4 tokens trouvés → ligne mal formée, ignorer

#### 2.4. Conversion et validation numériques

Pour chaque champ numérique :

- Utiliser `strtol()` pour convertir le token en entier long
- Vérifier que la conversion a réussi (pointeur de fin modifié)
- Appliquer les règles de validation :
  - `arrival_time` : doit être  $\geq 0$  (sinon ignorer la ligne)
  - `exec_time` : doit être  $> 0$  (sinon ignorer la ligne)
  - `priority` : toute valeur entière acceptée

## 2.5. Expansion dynamique du tableau

- Si le tableau est plein (nombre de processus  $\geq$  capacité) :
  - Doubler la capacité du tableau
  - Réallouer la mémoire avec `realloc()`
  - Vérifier succès allocation (sinon libérer et retourner erreur)

## 2.6. Ajout du processus au tableau

- Copier le nom dans `processes[n].name` (limite : `NAME_LEN` caractères)
- Assigner `arrival_time`, `exec_time`, `priority`
- Initialiser `remaining_time = exec_time`
- Initialiser `status = 0` (READY)
- Initialiser `end_time = 0`, `waiting_time = 0`
- Incrémenter le compteur de processus

## Étape 3 : Finalisation

### 3.1. Fermeture du fichier

- Fermer le descripteur de fichier

### 3.2. Vérification résultat

- Si aucun processus valide trouvé (`count = 0`) :
  - Libérer le tableau
  - Retourner succès avec 0 éléments

### 3.3. Optimisation mémoire (optionnel)

- Réduire la taille allouée à la taille exacte utilisée
- Utiliser `realloc()` pour ajuster à `count * sizeof(struct process)`

## Étape 4 : Tri par temps d'arrivée

- Appeler `qsort()` avec comparateur `cmp_arrival()`
- Comparateur : retourne `pa->arrival_time - pb->arrival_time`
- Résultat : tableau trié par ordre croissant d'arrivée

## Étape 5 : Retour

- Assigner le pointeur du tableau à `*out`
- Assigner le nombre de processus à `*out_n`

- Retourner 0 (succès)

### 6.3 Générateur Configuration Automatique

**But**

Créer automatiquement un fichier de configuration contenant des processus générés aléatoirement, sans intervention manuelle.

**Paramètres d'Entrée**

Le générateur accepte **5 paramètres** :

Paramètre	Type	Explication	Exemple
<code>nb_processes</code>	Entier	Nombre de processus à générer	20
<code>max_arrival_time</code>	Entier	Temps d'arrivée maximal (min=0)	100
<code>min_priority</code>	Entier	Priorité minimale	0
<code>max_priority</code>	Entier	Priorité maximale	5
<code>max_exec_time</code>	Entier	Durée d'exécution maximale (min=1)	500

**Algorithme de Génération Automatique**

**Étape 1 : Initialisation du générateur aléatoire**

- Appeler `srand(time(NULL))` pour initialiser le seed
- Utiliser le timestamp actuel comme source d'aléatoire
- Garantit génération différente à chaque exécution

**Étape 2 : Collecte des paramètres utilisateur**

Demander interactivement à l'utilisateur :

- **Nombre de processus** : `nb_processes` (doit être > 0)
- **Temps d'arrivée maximal** : `max_arrival_time` (doit être ≥ 0)
- **Priorité minimale** : `min_priority` (toute valeur entière)
- **Priorité maximale** : `max_priority` (doit être ≥ min\_priority)
- **Temps d'exécution maximal** : `max_exec_time` (doit être > 0)

Validation : vérifier que les contraintes sont respectées, sinon retourner erreur

**Étape 3 : Création du fichier de sortie**

- Ouvrir le fichier en mode écriture ("`w`")
- Nom du fichier : passé en paramètre ou généré avec timestamp
- Format timestamp : `sample_config_YYYYMMDD_HHMMSS.txt`
- Si échec ouverture : afficher erreur et retourner -1

**Étape 4 : Écriture de l'en-tête**

- Ligne 1 : `# Auto-generated file - N random processes`
- Ligne 2 : `# Params: arrival[0-MAX], priority[MIN-MAX], exec[1-MAX]`
- Ligne 3 : ligne vide pour séparation

## Étape 5 : Génération des processus

Pour chaque processus  $i$  de 1 à `nb_processes` :

### 5.1. Génération du nom

- Format : `P` suivi du numéro séquentiel
- Exemple : `P1, P2, P3, ..., P20`
- Utiliser `snprintf()` pour formater

### 5.2. Génération temps d'arrivée

- Formule : `arrival_time = rand() % (max_arrival_time + 1)`
- Plage résultante : `[0, max_arrival_time]` (inclusif)
- Distribution : uniforme

### 5.3. Génération temps d'exécution

- Formule : `exec_time = 1 + rand() % max_exec_time`
- Plage résultante : `[1, max_exec_time]` (jamais 0)
- Distribution : uniforme
- Garantie : processus toujours exécutables

### 5.4. Génération priorité

- Formule : `priority = min_priority + rand() % (max_priority - min_priority + 1)`
- Plage résultante : `[min_priority, max_priority]` (inclusif)
- Distribution : uniforme
- Exemple : si `min=0` et `max=5` → priorités possibles : 0, 1, 2, 3, 4, 5

### 5.5. Écriture de la ligne

- Format : `NAME ARRIVAL EXEC PRIORITY\n`
- Exemple : `P1 15 250 3\n`
- Utiliser `fprintf()` pour écrire dans le fichier

## Étape 6 : Finalisation

- Fermer le fichier avec `fclose()`
- Afficher message de confirmation : ✓ `File 'filename' generated successfully.`
- Afficher le chemin absolu ou relatif du fichier créé
- Retourner 0 (succès)

## Étape 7 : Vérification automatique

- Le fichier généré est **toujours valide** (respect des règles)
- Toutes les lignes ont exactement 4 champs
- Tous les `exec_time` sont  $> 0$

- Tous les `arrival_time` sont  $\geq 0$
- Pas besoin de validation manuelle

Fichier Résultat

- **Nommage** : `sample_config_TIMESTAMP.txt`
  - Format timestamp : `YYYYMMDD_HHMMSS` (ex: `sample_config_20251206_143052.txt`)
- **Validité** : Fichier généré est automatiquement **valide** (respecte toutes les règles)
- **Sortie** : Affichage confirmation + chemin fichier

7. Makefile et Compilation

7.1 Objectif du Makefile

Le Makefile permet de :

- **Compiler automatiquement** l'exécutable `ordonnanceur` à partir des fichiers source
- **Générer les fichiers objets** (.o) dans `build/`
- **Faciliter le nettoyage** du projet (remove objets, exécutables)
- **Éviter la compilation manuelle** (pas besoin de taper gcc à chaque fois)

7.2 Variables Principales

Variable	Signification	Valeur	Utilité
TARGET	Exécutable final	<code>ordonnanceur</code>	Nom du binaire
SRC_DIR	Répertoire source	<code>src</code>	Où chercher .c principaux
INC_DIR	Répertoire headers	<code>include</code>	Où chercher .h
POL_DIR	Répertoire politiques	<code>policies</code>	Où chercher algorithmes .c
BUILD_DIR	Répertoire objets	<code>build</code>	Où générer .o
SRC	Liste source	<code>\$(wildcard src/*.c)</code>	Tous .c dans src/
POLICIES	Liste politiques	<code>\$(wildcard policies/*.c)</code>	Tous .c dans policies/
OBJ	Liste objets	Substitution → <code>build/*.o</code>	Fichiers intermédiaires
CC	Compilateur C	<code>gcc</code>	Exécutable compilation
CFLAGS	Options compilation	<code>-Wall -Wextra -std=c11 -I\$(INC_DIR)</code>	Warnings + includes

7.3 Règles Principales

Règle par défaut : `all`



```
all: build $(TARGET)
```

### Dépendances :

1. Crée le répertoire **build/** (si nécessaire)
2. Construit l'exécutable **ordonnanceur**

### Usage :

```
make          # Compilation complète
make all      # Équivalent
```

### Construction de l'exécutable

```
$(TARGET): $(OBJ)
$(CC) -o $@ $^ $(CFLAGS)
```

- **\$@** : Cible (ordonnanceur)
- **\$^** : Toutes dépendances (fichiers .o)
- **Action** : Linker tous les objets en un exécutable unique

### Compilation fichiers source

```
$(BUILD_DIR)/%.o: $(SRC_DIR)/%.c
$(CC) $(CFLAGS) -c $< -o $@
```

- **%.o** : Règle pattern pour n'importe quel fichier objet
- **\$<** : Fichier source correspondant
- **-c** : Compiler uniquement (pas de linking)
- **-o \$@** : Output fichier objet
- **Note** : **-I\$(INC\_DIR)** déjà inclus dans CFLAGS

### Compilation fichiers politiques

```
$(BUILD_DIR)/%.o: $(POL_DIR)/%.c
$(CC) $(CFLAGS) -c $< -o $@
```

**Identique** à la précédente, mais pour fichiers dans **politiques/**.

### Création du dossier build/

```
build:
    @mkdir -p $(BUILD_DIR)
```

- **-p** : Crée le dossier uniquement si inexistant, pas d'erreur
- **@** : Supprime affichage de la commande dans terminal

### Nettoyage standard : **clean**

```
clean:
    rm -rf $(BUILD_DIR) $(TARGET)
```

#### Supprime :

- Répertoire **build/** et tous fichiers **.o**
- Exécutable **ordonnanceur**

#### Usage :

```
make clean          # Préparer recompilation propre
```

### Nettoyage complet : **mrproper**

```
mrproper: clean
```

**Action** : Appelle simplement **clean** (actuellement identique)

#### Usage :

```
make mrproper      # Nettoyage complet
```

## 7.4 Déclaration PHONY

```
.PHONY: all clean mrproper build
```

**Pourquoi** : Indique à **make** que ce ne sont pas des fichiers, mais des commandes. Évite conflits si un fichier s'appelle "clean".

## 7.6 Principes et Avantages

Principe	Avantage
Automatisation	Plus besoin de gcc manuel à chaque fois
Modularité	Ajouter <i>src/.c</i> ou <i>policies/.c</i> sans modifier Makefile
Compilation incrémentale	Recompile uniquement ce qui a changé
Répertoire dédié	build/ = propre, tous les .o centralisés
Nettoyage facile	make clean = repartir à zéro
Portabilité	Variables faciles à modifier pour autre compilateur

## 7.7 Utilisation Pratique

```
# Compilation complète
make
# Nettoyer objets uniquement (récompile changé)
make clean
# Nettoyage total (repartir zéro + config)
make mrproper
# Voir étapes compilation
make -d                # Mode debug
```

## 8. Conclusion

### 8.1 Résultats Obtenus

Ce projet a permis de réaliser un **simulateur complet d'ordonnement de processus** avec les résultats suivants :

#### Objectifs Techniques Atteints

- ✅ 6 algorithmes d'ordonnement implémentés et fonctionnels
- ✅ Architecture modulaire et extensible
- ✅ Générateur automatique de configurations
- ✅ Parser robuste
- ✅ Compilation automatisée