# DOCUMENTATION TECHNIQUE

## Simulateur d'Ordonnancement de Processus Linux

### avec Interface Web Interactive

Projet : Systèmes d'Exploitation Avancés

Décembre 2025

---


### Équipe de Développement

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

---

**Institut Supérieur d'Informatique - Ariana**
**1ING3**


**Dépôt GitHub:** arijsebai/Projet-Ordonnancement-Linux

# Table des Matières

# 1. Introduction et Vue d'Ensemble

## 1.1 Contexte et Objectifs

Ce projet implémente un **simulateur complet d'ordonnancement de processus sous Linux** combinant :

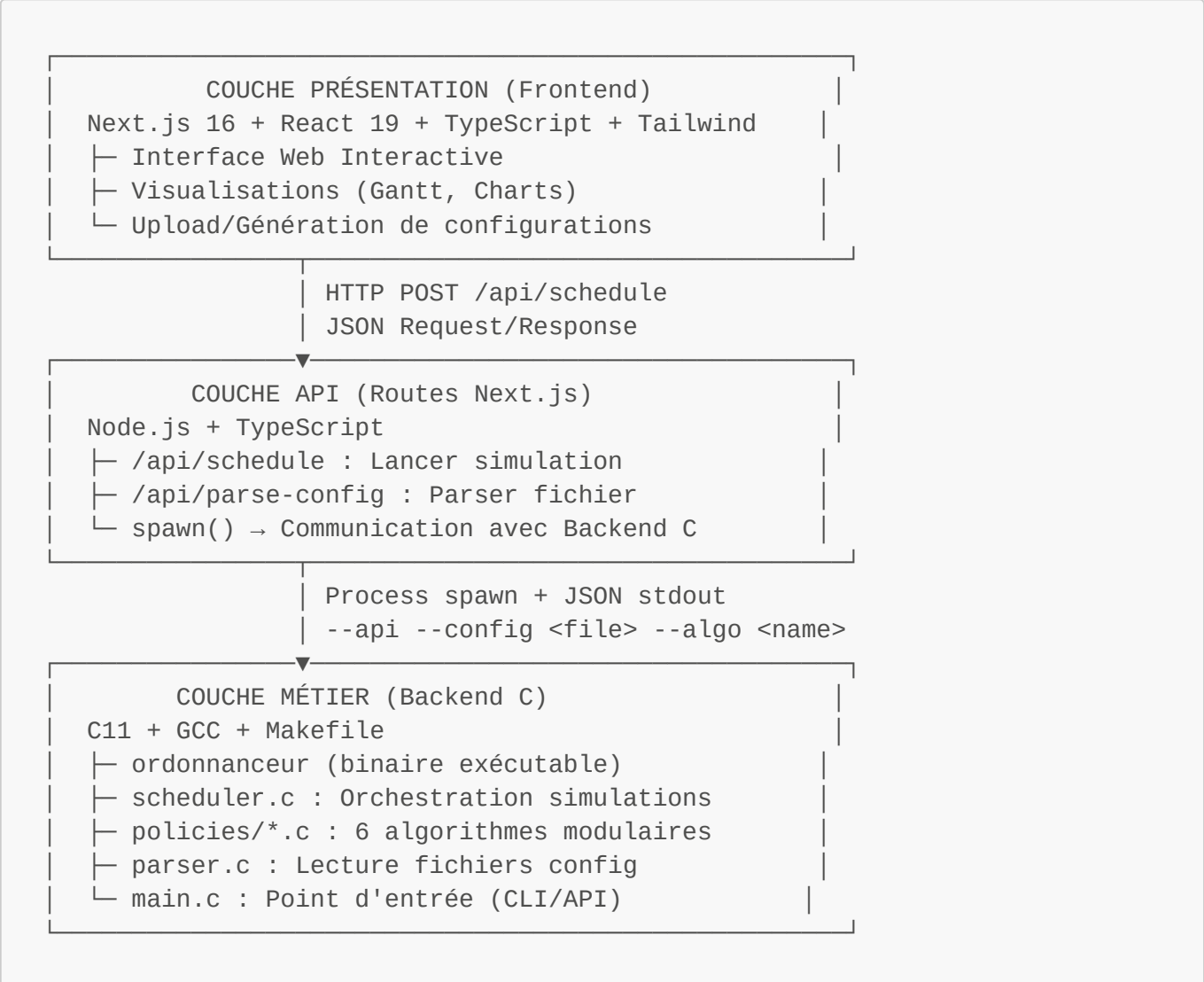
- Un **backend en C** (C11/GCC) simulant 6 algorithmes d'ordonnancement
- Un **frontend moderne en Next.js 16** avec interface web interactive
- Une **architecture hybride** permettant l'utilisation en CLI ou via navigateur web

**Objectifs Pédagogiques**

✅ **Comprendre les algorithmes d'ordonnancement** : FIFO, Priority, Round-Robin, SRT, Multilevel (statique/dynamique)
✅ **Visualiser en temps réel** : Diagramme de Gantt interactif, graphiques de performance
✅ **Expérimenter avec des paramètres** : quantum, ordre de priorité, fichiers de configuration
✅ **Analyser les métriques** : temps d'attente, makespan, utilisation CPU

## 1.2 Architecture Globale du Projet

Le projet suit une **architecture en 3 couches** :



**Avantages de cette architecture** :

- 🔄 **Séparation des responsabilités** : UI, API, Business Logic
- 🧩 **Modularité** : Chaque algorithme est un module indépendant
- 🚀 **Performance** : Backend C optimisé, Frontend React moderne
- 🎯 **Flexibilité** : Utilisable en CLI direct ou via Web UI

### 1.3 Technologies Utilisées

**Backend (Couche Métier)**

| Technologie | Version | Rôle |
|---|---|---|
| **C** | C11 standard | Langage principal du simulateur |
| **GCC** | 9.4.0+ | Compilateur C |
| **Make** | 4.2.1+ | Build automation |

**Frontend (Couche Présentation)**

| Technologie | Version | Rôle |
|---|---|---|
| **Next.js** | 16.0.3 | Framework React full-stack |
| **React** | 19.2.0 | UI library avec Server Components |
| **TypeScript** | 5.x | Type safety et meilleur DX |
| **Tailwind CSS** | 4.1.9 | Styling utility-first |
| **Radix UI** | Multiple | Composants UI accessibles |
| **Recharts** | Latest | Bibliothèque de graphiques |

**Structure des Fichiers**

```
Projet-Ordonnancement-Linux/
├── src/                    # Backend C sources
│   ├── main.c              # Point d'entrée (CLI + API)
│   ├── scheduler.c         # Orchestrateur simulations
│   ├── parser.c            # Parser de fichiers config
│   ├── generate_config.c   # Générateur automatique
│   └── utils.c             # Utilitaires
├── include/                # Headers C
│   ├── process.h           # Structure process
│   ├── scheduler.h         # API scheduler
│   ├── parser.h            # API parser
│   └── generate_config.h   # API generator
├── policies/               # Algorithmes d'ordonnancement
│   ├── fifo.c
│   ├── priority_preemptive.c
```

```
│   ├── roundrobin.c
│   ├── srt.c
│   ├── multilevel.c
│   └── multilevel_dynamic.c
├── tests/                  # Tests unitaires C
│   ├── test_fifo.c
│   ├── test_priority.c
│   ├── test_roundrobin.c
│   ├── test_multilevel.c
│   └── test_multilevel_dynamic.c
├── app/                    # Frontend Next.js
│   ├── page.tsx           # Page principale
│   ├── layout.tsx         # Layout global
│   ├── globals.css        # Styles globaux
│   └── api/               # Routes API
│       ├── schedule/route.ts      # POST /api/schedule
│       └── parse-config/route.ts  # POST /api/parse-config
├── components/             # Composants React
│   ├── algorithm-selector.tsx
│   ├── results-display.tsx
│   ├── file-generation-dialog.tsx
│   └── ui/                # Composants Radix UI
├── lib/                    # Utilitaires TypeScript
│   ├── types.ts          # Types TypeScript
│   └── utils.ts          # Helpers
├── config/               # Fichiers de configuration
│   └── sample_config.txt
├── Makefile             # Build automation C
├── package.json         # Dependencies Node.js
├── tsconfig.json        # Configuration TypeScript
└── README.md            # Documentation utilisateur
```

**Total** : ~30 fichiers C/H, ~15 composants React/TypeScript

# 2. Structures de Données

## 2.1 Structure `process` (include/process.h)

La structure `process` est le **cœur du système**, définie dans `include/process.h` :

```c
#define NAME_LEN 64

struct process {
    char name[NAME_LEN];        // Identifiant unique du processus
    int arrival_time;           // Temps d'arrivée dans le système
    int exec_time;              // Durée totale d'exécution (immutable)
    int priority;               // Priorité (PETITE valeur = HAUTE
priorité, Unix)
    int remaining_time;         // Temps restant à exécuter (mutable)
    int waiting_time;           // Temps d'attente cumulé
    int status;                 // État : non utilisé (gestion implicite)
```

```
    int end_time;               // Temps de fin (pour calcul métriques)
    int wait_time;              // Utilisé pour aging (Multilevel Dynamic)
};
```

**Champs clés** :

| Champ | Type | Description | Usage |
|---|---|---|---|
| `name` | char[64] | Identifiant (P1, P2...) | Affichage, tri |
| `arrival_time` | int | Moment d'arrivée | Critère FIFO, eligibilité |
| `exec_time` | int | Durée totale CPU | Référence immuable |
| `priority` | int | Priorité statique | Priority, Multilevel algos |
| `remaining_time` | int | Temps restant | Décrémenté chaque cycle, critère SRT |
| `waiting_time` | int | Temps attente total | Statistiques finales |
| `end_time` | int | Temps de terminaison | Calcul turnaround/wait times |

## 2.2 Structures de Simulation (include/scheduler.h)

**Structure `gantt_segment`**

Représente un **segment d'exécution** pour le diagramme de Gantt :

```
struct gantt_segment {
    char process[NAME_LEN];     // Nom du processus exécuté
    int start;                  // Temps de début
    int end;                    // Temps de fin
};
```

**Utilisé pour** : Construire le timeline du Gantt chart (frontend visualisation)

**Structure `process_stat`**

Statistiques **finales par processus** :

```
struct process_stat {
    char id[NAME_LEN];          // Identifiant processus
    int arrival_time;           // Temps d'arrivée
    int exec_time;              // Durée exécution
    int finish_time;            // Temps de fin
    int wait_time;              // Temps d'attente
    int priority;               // Priorité initiale
    int final_priority;         // Priorité finale (Multilevel Dynamic)
};
```

**Structure `simulation_result`**

Résultat **complet d'une simulation** (retourné en JSON) :

```
#define MAX_SEGMENTS 2048

struct simulation_result {
    char algorithm[64];                     // Nom algorithme ("fifo",
"roundrobin"...)
    struct gantt_segment segments[MAX_SEGMENTS]; // Timeline Gantt
    int segment_count;                      // Nombre de segments
    struct process_stat stats[256];         // Stats par processus
    int stat_count;                         // Nombre de processus
    double average_wait;                    // Moyenne temps d'attente
    int makespan;                           // Temps total simulation
};
```

**Flux :** `scheduler.c` remplit cette structure → `print_json_result()` → stdout JSON → route Next.js parse → frontend affiche

### 2.3 Convention de Priorité Unix

⚠️ **IMPORTANT** : Le projet utilise la **convention Unix standard** :

```
PETITE valeur = HAUTE priorité
```

**Exemples :**

- `priority = 0` → Priorité **maximale**
- `priority = 5` → Priorité **faible**
- `priority = 10` → Priorité **très faible**

**Modes de tri :**

| Mode | Flag CLI | Comportement |
|---|---|---|
| **Ascending** (défaut API) | `--prio-order asc` | Processus avec petite valeur sélectionné en premier |
| **Descending** (défaut CLI) | `--prio-order desc` | Processus avec grande valeur sélectionné en premier |

**Défauts système :**

- **CLI interactif :** `prio_mode = 1` (descending) défini dans `main.c`
- **API Next.js :** Envoie toujours `--prio-order asc` (ascending)

**Représentation dans le code :**

```
// Ascending (petite = haute prio)
if (priority < best_priority) {
    best = i;
}

// Descending (grande = haute prio)
if (priority > best_priority) {
    best = i;
}
```

## 3. Algorithmes d'Ordonnancement

Le projet implémente **6 algorithmes d'ordonnancement** modulaires dans le dossier `policies/`. Chaque algorithme expose une **fonction de sélection** appelée par `scheduler.c`.

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

**Fichier** : `policies/fifo.c`
**Type** : Non-préemptif
**Critère** : Plus petit `arrival_time`

#### Fonction de Sélection

```
int fifo_scheduler(struct process *procs, int n, int time, int current, int prio_mode)
```

**Algorithme** :

1. Parcourir tous les processus
2. Sélectionner ceux arrivés (`arrival_time <= time`) ET non terminés (`remaining_time > 0`)
3. Retourner l'index de celui avec le **plus petit** `arrival_time`
4. Retourner -1 si aucun processus prêt (CPU IDLE)

**Avantages** :

- ✅ Simple, déterministe
- ✅ Zéro overhead de context switch
- ✅ Bon pour batch jobs

**Inconvénients** :

- ❌ **Convoy effect** : Un processus long bloque tous les autres
- ❌ Temps d'attente élevé pour processus courts
- ❌ Injuste pour interactif

### 3.2 Priority Preemptive

**Fichier** : `policies/priority_preemptive.c`
**Type** : Préemptif
**Critère** : Meilleure priorité selon mode (asc/desc)

**Fonction de Sélection**

```
int priority_preemptive(struct process *procs, int n, int time, int
current, int prio_mode)
```

**Algorithme** :

1. Initialiser `best_prio` selon mode :
   - Ascending (0) : `best_prio = INT_MAX` (chercher minimum)
   - Descending (1) : `best_prio = INT_MIN` (chercher maximum)
2. Parcourir processus arrivés et non terminés
3. Selon `prio_mode` :
   - **Ascending** : Si `priority < best_prio` → nouveau meilleur
   - **Descending** : Si `priority > best_prio` → nouveau meilleur
4. Retourner index du processus avec meilleure priorité

**Convention** :

- **Ascending** (défaut API) : priority=0 > priority=5
- **Descending** (défaut CLI) : priority=5 > priority=0

**Avantages** :

- ✅ Priorité stricte respectée
- ✅ Bon pour systèmes temps-réel

**Inconvénients** :

- ❌ **Famine** : Basses priorités peuvent ne jamais s'exécuter
- ❌ Beaucoup de préemptions

### 3.3 Round Robin (RR)

**Fichier** : `policies/roundrobin.c`
**Type** : Préemptif avec quantum
**Critère** : File circulaire FIFO

**Implémentation**

**Particularité** : Round Robin **n'utilise pas** la fonction de sélection séparée. Il **implémente sa propre boucle complète** avec gestion de file.

```
void round_robin(struct process *procs, int n, int quantum)
```

**Algorithme** :

1. Initialiser file circulaire (`ready[100]`, `head`, `tail`)
2. **Boucle principale** :
   - Ajouter nouveaux arrivés à la file
   - Exécuter processus courant pour `quantum` unités de temps
   - Si quantum expiré ET processus non terminé → réinsérer en fin de file
   - Si processus terminé → ne pas réinsérer
3. Afficher Gantt textuel et statistiques

**Avantages** :

- ✅ Équitable entre processus
- ✅ Bon pour systèmes interactifs
- ✅ Pas de famine

**Inconvénients** :

- ❌ Overhead context switches si quantum trop petit
- ❌ Temps d'attente moyen augmente avec nombre processus

**Paramètre quantum** : Configurable via `--quantum <valeur>` (typiquement 2-4)

### 3.4 SRT (Shortest Remaining Time First)

**Fichier** : `policies/srt.c`
**Type** : Préemptif
**Critère** : Plus petit `remaining_time`

**Implémentation**

**Particularité** : SRT implémente également sa **propre boucle** (fonction `srt_simulation`).

```
void srt_simulation(struct process *p, int n)
```

**Algorithme** :

1. À chaque unité de temps :
2. Trouver processus avec **minimum** `remaining_time` parmi processus arrivés
3. Si égalité → départager par `arrival_time` (FIFO)
4. Exécuter ce processus pendant 1 unité
5. Décrémenter `remaining_time`

**Avantages** :

- ✅ **Temps d'attente moyen optimal** (prouvé mathématiquement)
- ✅ Processus courts favorisés

**Inconvénients** :

- ❌ **Famine des longs processus** ⚠️
- ❌ Requiert connaissance durée future (irréaliste)
- ❌ Beaucoup de préemptions

### 3.5 Multilevel Queue (Statique)

**Fichier** : `policies/multilevel.c`
**Type** : Préemptif avec quantum
**Critère** : Priorité stricte + Round-Robin par niveau

**Fonction de Sélection**

```
int select_multilevel(struct process *procs, int n, int time, int current,
int quantum_expired)
```

**Algorithme** :

1. **Trouver priorité minimum** (haute priorité Unix) parmi processus prêts
2. **Si processus courant** a même priorité ET quantum non expiré → continuer
3. **Sinon** : Round-Robin circulaire parmi processus de même priorité
    - Commencer à `(current + 1) % n`
    - Parcourir circulairement
    - Sélectionner premier processus avec priorité minimum

**Principe** :

- Processus regroupés par niveaux de priorité (valeur `priority`)
- **Priorité stricte** : Niveau supérieur s'exécute avant inférieur
- **Round-Robin intra-niveau** : Équité entre processus de même priorité

**Avantages** :

- ✅ Hiérarchie claire
- ✅ Déterministe
- ✅ Bon pour systèmes mixtes (batch + interactif)

**Inconvénients** :

- ❌ **Famine** : Basses priorités bloquées si hautes priorités actives
- ❌ Priorités fixes, pas d'adaptation

### 3.6 Multilevel Feedback Queue (Dynamique) ⭐

**Fichier** : `policies/multilevel_dynamic.c`
**Type** : Préemptif avec quantum + aging
**Critère** : Priorité dynamique + Round-Robin

**Fonction de Sélection**

```
int select_multilevel_dynamic(struct process *procs, int n, int time, int
current, int quantum_expired)
```

**Même logique que Multilevel Static** pour la sélection.

**Mécanisme d'Aging (Anti-Famine)**

**Implémenté dans `multilevel_dynamic_simulation()` (scheduler.c)** :

```
// À chaque cycle, pour TOUS les processus en attente
for (int i = 0; i < n; i++) {
    if (i != idx && procs[i].arrival_time <= time &&
procs[i].remaining_time > 0) {
        procs[i].priority--;    // Augmentation priorité (valeur diminue)
        procs[i].waiting_time++;
    }
}
```

**Principe** :

- Processus en attente voit sa priorité **augmenter continuellement**
- Plus un processus attend, plus sa priorité diminue (Unix : petite valeur = haute prio)
- **Garantit** qu'aucun processus n'attend indéfiniment

**Avantages** :

- ✅ **Pas de famine** (aging garantit exécution)
- ✅ Adaptatif aux conditions système
- ✅ Équitable long terme

**Inconvénients** :

- ❌ Overhead aging
- ❌ Comportement moins prévisible

### 3.7 Tableau Comparatif des Algorithmes

| Algorithme | Préemptif | Paramètres | Complexité Sélection | Famine | Cas d'Usage |
|---|---|---|---|---|---|
| **FIFO** | ❌ Non | Aucun | O(n) | ❌ Non | Batch jobs |
| **Priority** | ✅ Oui | prio_mode | O(n) | ⚠️ Oui | Temps-réel |
| **Round Robin** | ✅ Oui | quantum | O(1) | ❌ Non | Interactif |
| **SRT** | ✅ Oui | Aucun | O(n) | ⚠️ Oui | Théorique |
| **Multilevel** | ✅ Oui | quantum | O(n) | ⚠️ Oui | Multi-classe |

| Algorithme | Préemptif | Paramètres | Complexité Sélection | Famine | Cas d'Usage |
|---|---|---|---|---|---|
| **Multilevel Dynamic** | ✅ Oui | quantum | O(n) | ❌ Non | Production moderne |

**Légende** :

- **Famine** ⚠️ Oui : Certains processus peuvent ne jamais s'exécuter
- **Famine** ❌ Non : Tous les processus s'exécutent éventuellement

**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 |
|---|---|
| ✅ **Processus critiques prioritaires** | Parfait pour temps réel |
| ✅ **Flexible** | Modes ascendant/descendant |
| ✅ **Simple à implémenter** | Pas de structure complexe |
| ❌ **Processus faible priorité peuvent starver** | Risque famine |
| ❌ **Overhead context switches** | Dégradation performance si trop préemptions |
| ❌ **Pas équitable** | 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 déjà arrivé)
  - `remaining_time > 0` (processus non terminé)
  - `end_time == -1` (processus pas encore complété)
  - **Processus pas déjà présent dans la queue** (vérification explicite)
- **Mécanisme de détection de duplication** :
  - Pour chaque candidat, parcourir la queue actuelle `[head, tail)`
  - Vérifier si l'indice du processus est déjà dans `ready[j]`
  - Si trouvé → `in_queue = 1`, ne pas ajouter
  - Si non trouvé → ajouter `ready[tail++] = i`

#### 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. Mettre à jour le waiting_time**

- Pour tous les processus **encore en queue** (de `head` à `tail`) :
  - `waiting_time += run` (ils attendent pendant que `curr` s'exécute pendant `run` unités)

**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 | Equité | Notes |
|---|---|---|---|---|
| **1-2** | Très élevé | Excellente | Parfaite | Overhead inacceptable |
| **4** ✅ | Modéré | Bonne | Très bonne | **OPTIMAL TROUVÉ** |
| **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 |
|---|---|
| ✅ Temps attente très bon | Résultats excellents |
| ✅ Peu de préemptions | Comparé à Priority |
| ❌ **FAMINE des longs processus** ⚠️ | 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** : Petite valeur = Haute Priorité (ex: 1 > 10, conforme Unix)

**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 MINIMUM des Processus Prêts** (convention Unix : petite = haute)

Initialiser :

- `best_prio = INT_MAX` (très grande valeur, on cherche le minimum)
- `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 PETITE 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 petite)
- `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 minimum) :

- **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 |
| ❌ **FAMINE des basses priorités** ⚠️ | 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 MINIMUM parmi les processus prêts** (convention Unix : petite = haute)

- Initialiser `best_prio = INT_MAX` (valeur très grande)
- Parcourir tous les processus
- Si `arrival_time <= time` ET `remaining_time > 0` :
  - Si `priority < best_prio` → mettre à jour `best_prio` (on cherche la PETITE 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 petite)
  - `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 minimum)
- 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 |
|---|---|
| ✅ **Anti-famine** | Aging garantit personne n'attend indéfiniment |
| ✅ **Adaptation dynamique** | S'ajuste au comportement processus |
| ✅ **Équitable** | Meilleur que multilevel statique |
| ✅ **Moderne** | Inspiré Linux CFS réel |

| Aspect | Évaluation |
|---|---|
| ⚠️ Complexité accrue | Plus de compteurs et conditions |
| ⚠️ Moins déterministe | Feedback rend résultats moins prévisibles |

# 4. Architecture et Communication

## 4.1 Backend C : Modes d'Exécution

Le backend C supporte **3 modes d'opération** via `main.c` :

**Mode 1 : CLI Interactif (Menu)**

```
./ordonnanceur
```

**Fonctionnement** :

1. Affiche menu : "Générer config" ou "Charger fichier existant"
2. Charge/parse le fichier de configuration
3. Affiche liste des algorithmes disponibles
4. Demande paramètres (quantum, prio-order si applicable)
5. Exécute simulation
6. Affiche résultats **textuels** : Gantt ASCII, statistiques

**Usage** : Utilisation CLI directe, démos en console

**Mode 2 : CLI Direct File**

```
./ordonnanceur config/sample_config.txt
```

**Fonctionnement** :

- Charge directement le fichier fourni en argument
- Saute menu initial
- Affiche menu sélection algorithme
- Exécute et affiche résultats textuels

**Usage** : Scripts automatisés, tests rapides

**Mode 3 : API Mode (JSON)**

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

**Fonctionnement** :

1. Parse arguments CLI (flags)
2. Charge configuration via `parser.c`
3. Exécute algorithme spécifié
4. Génère **sortie JSON structurée** sur stdout
5. Aucun affichage interactif

**Sortie JSON** :

```
{
  "algorithm": "roundrobin",
  "ganttData": [
    { "process": "P1", "start": 0, "end": 4, "duration": 4 },
    { "process": "P2", "start": 4, "end": 7, "duration": 3 }
  ],
  "processStats": [
    {
      "id": "P1",
      "arrivalTime": 0,
      "executionTime": 10,
      "finishTime": 15,
      "waitTime": 5,
      "priority": 1
    }
  ],
  "averageWait": 5.2,
  "makespan": 25
}
```

**Usage** : Appelé programmatiquement par routes Next.js API

**Mode Spécial : Parse-Only**

```
./ordonnanceur --parse-config config/sample_config.txt
```

**Fonctionnement** :

- Parse fichier uniquement (pas de simulation)
- Retourne JSON array des processus
- Utilisé par `/api/parse-config` pour upload fichier

**Sortie JSON** :

```
[
  {"id":"P1","arrivalTime":0,"executionTime":5,"priority":1},
```

```
  {"id":"P2","arrivalTime":2,"executionTime":3,"priority":2}
]
```

### 4.2 Frontend Next.js : Composants React

**Page Principale (`app/page.tsx`)**

**Responsabilités** :

- Gestion fichiers : Upload `.txt` ou génération manuelle
- Affichage tableau processus (editable)
- Sélection algorithme via `<AlgorithmSelector />`
- Déclenchement simulation (button "Lancer")
- Affichage résultats via `<ResultsDisplay />`

**State management** :

```
const [processes, setProcesses] = useState<Process[]>([])
const [config, setConfig] = useState<AlgorithmConfig>({ algorithm: 'fifo'
})
const [result, setResult] = useState<SchedulingResult | null>(null)
```

**Composant AlgorithmSelector**

**Fichier** : `components/algorithm-selector.tsx`

**Fonctionnalités** :

- Dropdown avec 6 algorithmes : fifo, priority, roundrobin, srt, multilevel, multilevel-dynamic
- **Paramètres conditionnels** :
  - `quantum` : visible si roundrobin ou multilevel-dynamic
  - `priorityOrder` : visible si priority
- Validation saisie utilisateur

**Props** :

```
interface AlgorithmConfig {
  algorithm: 'fifo' | 'priority' | 'roundrobin' | 'srt' | 'multilevel' |
'multilevel-dynamic'
  quantum?: number
  priorityOrder?: 'asc' | 'desc'
}
```

**Composant ResultsDisplay**

**Fichier** : `components/results-display.tsx`

**Fonctionnalités** :

- **Gantt Chart** : Timeline interactif avec Recharts
  - Play/Pause simulation
  - Zoom/Pan
  - Tooltip processus
- **Pie Chart** : Répartition temps total par processus
- **Bar Chart** : Temps d'attente vs Temps total
- **Tableau détaillé** :
  - Colonnes : ID, Arrival, Execution, Wait, Total, Finish
  - **Priority initiale** : toujours affiché
  - **Priority finale** : uniquement pour multilevel-dynamic (après aging)
- **Métriques globales** : Average wait time, Makespan

**Palette couleurs** :

- 20 couleurs prédéfinies + fallback HSL
- Déterministe par process ID (hash)

## 4.3 Communication Frontend ↔ Backend

**Routes API Next.js**

**1. POST /api/parse-config**

**Fichier** : `app/api/parse-config/route.ts`

**Flow** :

```
Client Upload .txt
    │
    ▼
Sauver fichier temp
    │
    ▼
spawn('./ordonnanceur', ['--parse-config', tmpFile])
    │
    ▼
Parse stdout JSON
    │
    ▼
Retourner Process[] au client
    │
    ▼
Cleanup fichier temp
```

**2. POST /api/schedule**

**Fichier** : `app/api/schedule/route.ts`

**Payload** :

```
{
  processes: Process[],
  config: AlgorithmConfig
}
```

**Flow** :

```
Recevoir processes + config
    │
    ▼
Écrire sched_<timestamp>.txt
    │
    ▼
Mapper algorithm name (frontend → backend)
  fifo → fifo
  priority → priority_preemptive
  roundrobin → roundrobin
  multilevel → multilevel
  multilevel-dynamic → multilevel_dynamic
  srt → srt
    │
    ▼
Construire args CLI:
  ['--api', '--config', tmpFile, '--algo', mappedAlgo]
  + ['--quantum', quantum] si applicable
  + ['--prio-order', order] si priority
    │
    ▼
spawn('./ordonnanceur', args)
    │
    ▼
Collecter stdout (JSON)
    │
    ▼
Parse JSON → SchedulingResult
    │
    ▼
Retourner au client
    │
    ▼
Cleanup fichier temp
```

## 4.4 Flow d'Exécution Complet

**Scénario : Utilisateur lance simulation Round-Robin avec quantum=3**


1. Client (React)
- Sélectionne "Round Robin"
- Entre quantum = 3
- Clique "Lancer"
POST /api/schedule
{ processes: [...], config: { algorithm: 'roundrobin', quantum: 3 } }
2. API Route (route.ts)
- Écrit sched_20251213_143022.txt
- Map 'roundrobin' → 'roundrobin'
- Construit args:
['--api', '--config', 'sched_...txt',
'--algo', 'roundrobin', '--quantum', '3']
spawn('./ordonnanceur', args)
stdout pipe
3. Backend C (main.c)
- Parse flags: api_mode=1, algo='roundrobin'
- Charge config via parser.c
- Appelle run_scheduler_api()
4. Scheduler (scheduler.c)
- Identifie algo = RR
- Appelle round_robin(procs, n, 3)
5. Politique RR (policies/roundrobin.c)
- Exécute simulation avec quantum=3
- Remplit simulation_result:
* gantt_segments[]
* process_stats[]
* average_wait, makespan
return à scheduler.c
6. Scheduler (scheduler.c)
- Appelle print_json_result(&result)
- Écrit JSON structuré sur stdout
stdout JSON

```
┌─────────────────────────────────────────────────────┐
│  7. API Route (route.ts)                            │
│     - Collecte stdout complet                       │
│     - Parse JSON                                    │
│     - Cleanup sched_...txt                          │
│     - Return 200 + JSON au client                   │
└──────────────────┬──────────────────────────────────┘
                   │ HTTP Response
                   ▼
┌─────────────────────────────────────────────────────┐
│  8. Client (React)                                  │
│     - Reçoit SchedulingResult                       │
│     - ResultsDisplay affiche:                       │
│       * Gantt chart animé                           │
│       * Pie chart                                   │
│       * Bar chart                                   │
│       * Tableau statistiques                        │
└─────────────────────────────────────────────────────┘
```

**Temps total** : ~500ms (parsing + simulation + render)

## 4.1 Choix des Technologies

| Technologie | Justification | Bénéfices |
|---|---|---|
| **Langage C** | Requis ; bas niveau ; standard académique | Proximité système, performance |
| **Git + GitHub** | Contrôle version ; collaboration ; historique | Traçabilité modifications |
| **Scrum/Agile** | Gestion itérative ; sprints ; équipe | Planification adaptable |
| **Trello** | Tableau Kanban ; visualisation tâches | Suivi avancement temps réel |
| **Microsoft Teams** | Communication équipe ; réunions | Coordination synchrone |
| **VS Code** | IDE léger ; plugins C ; compilation intégrée | Productivité développement |

## 4.2 Architecture du Projet

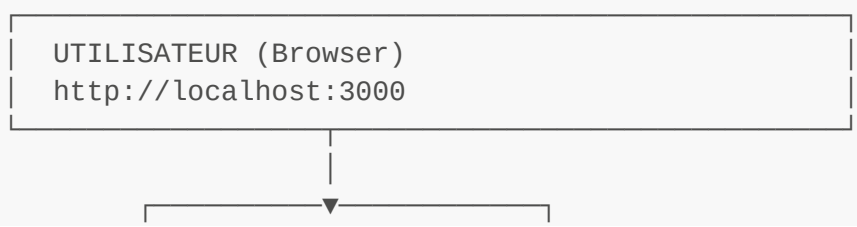
**Architecture Hybride : Next.js (Frontend) + C (Backend)**

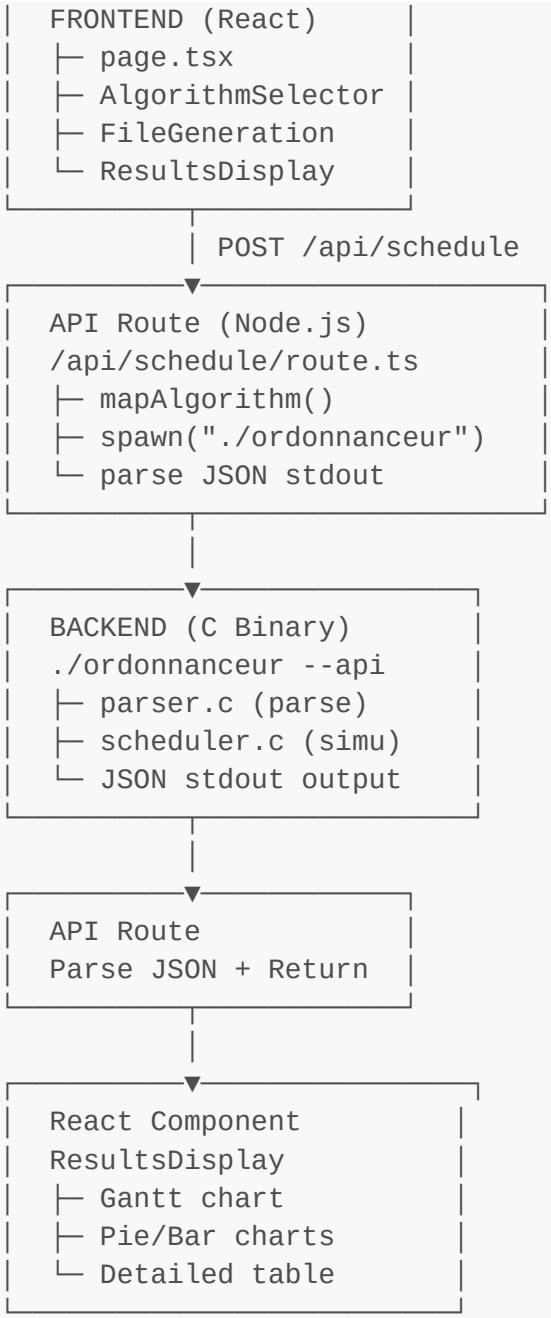
```
Projet-Ordonnancement-Linux-arij-dev/
│
├── FRONTEND (Next.js 16 + React 19)
│   │
│   ├── app/                          # Next.js App Router
│   │   ├── page.tsx                  # Page principale (UI)
│   │   ├── layout.tsx                # Layout racine
│   │   ├── globals.css               # Styles globaux
│   │   └── api/                      # API Routes (Node.js backend)
│   │       ├── schedule/route.ts     # Endpoint: POST /api/schedule
│   │       └── parse-config/route.ts  # Endpoint: POST /api/parse-config
```

```
│   │
│   ├── components/                    # React Components
│   │   ├── algorithm-selector.tsx     # Sélecteur algorithme (dropdown)
│   │   ├── file-generation-dialog.tsx # Dialog génération fichier
│   │   ├── results-display.tsx        # Affichage Gantt + Charts + Table
│   │   ├── theme-provider.tsx         # Thème UI (dark/light)
│   │   └── ui/                        # Components Radix UI
(réutilisables)
│   │       ├── button.tsx, card.tsx, dialog.tsx, input.tsx, etc.
│   │
│   ├── lib/                           # Utilitaires Frontend
│   │   ├── types.ts                   # Interfaces TypeScript (Process,
AlgorithmConfig, SchedulingResult)
│   │   └── utils.ts                   # Fonctions utilitaires
│   │
│   ├── hooks/                         # Hooks React personnalisés
│   │   ├── use-toast.ts               # Notifications
│   │   └── use-mobile.ts              # Détection responsive
│   │
│   ├── styles/                        # Feuilles de styles
│   │
│   ├── public/                        # Assets statiques
│   │
│   ├── tsconfig.json                  # Configuration TypeScript
│   ├── next.config.mjs                # Configuration Next.js
│   ├── postcss.config.mjs             # Configuration PostCSS
│   ├── package.json                   # Dépendances Node.js
│   └── pnpm-lock.yaml                 # Lock file dépendances
│
├── BACKEND (C + Binaire compilé)
│   │
│   ├── src/                           # Code source C
│   │   ├── main.c                     # Point d'entrée, modes (--api, --
parse-config, --config)
│   │   ├── parser.c                   # Parsing fichier configuration
│   │   ├── scheduler.c                # Moteur simulation + JSON output
│   │   ├── generate_config.c          # Générateur configs aléatoires
│   │   └── utils.c                    # Utilitaires C (logs, JSON,
affichage)
│   │
│   ├── include/                       # Headers C
│   │   ├── process.h                  # Structure process, constantes
│   │   ├── scheduler.h                # Prototypes moteur, statistiques
│   │   ├── parser.h                   # Prototypes parsing
│   │   ├── utils.h                    # Prototypes utilitaires
│   │   └── generate_config.h          # Prototypes générateur
│   │
│   ├── policies/                      # Implémentations algorithmes
│   │   ├── fifo.c                     # FIFO
│   │   ├── priority_preemptive.c      # Priority (préemptif)
│   │   ├── roundrobin.c               # Round Robin
│   │   ├── srt.c                      # SRT (Shortest Remaining Time)
│   │   ├── multilevel.c               # Multilevel (statique)
│   │   └── multilevel_dynamic.c       # Multilevel Dynamic (avec aging)
```

```
│   │
│   ├── 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 (généré par make)
│   │   └── *.o
│   │
│   ├── ordonnanceur                    # Binaire compilé (exécutable C)
│   ├── ordonnanceur.exe                # Binaire Windows
│   ├── Makefile                        # Compilation & tests
│   └── test_*                          # Exécutables tests
│
├── CONFIGURATION & DONNÉES
│   │
│   ├── config/                         # Fichiers configuration
│   │   ├── sample_config.txt           # Exemple configuration
│   │   └── config_*.txt                # Configs générées
│   │
│   └── components.json                 # Configuration Shadcn UI
│
├── DOCUMENTATION
│   │
│   ├── Documentation.md                # Documentation technique (cette
doc)
│   ├── README.md                       # Guide utilisateur + prérequis
│   ├── INDEX.md                        # Index navigation
│   ├── COMPLETION_SUMMARY.md           # Résumé changements
│   ├── FINAL_REPORT.md                 # Rapport validation
│   ├── CHANGELOG_CONFORMANCE.md        # Changelog détaillé
│   └── UPDATE_SUMMARY.md               # Résumé mises à jour
│
└── CONFIGURATION RACINE
    │
    ├── .gitignore                      # Git ignore
    ├── .next/                          # Cache Next.js
    ├── .vscode/                        # Configuration VS Code
    ├── node_modules/                   # Dépendances npm
    ├── LICENSE                         # MIT License
    ├── package.json                    # Dépendances Node.js + scripts
    └── tsconfig.json                   # TypeScript config
```

**Structure Logique par Rôle**






## 4.3 Backend C : Mode Interactif

**Mode Interactif (CLI)**

Le backend C supporte deux modes opérationnels :

**Mode 1: CLI Interactif (Menu)**

```
./ordonnanceur
# OU
./ordonnanceur [chemin_fichier_config.txt]
```

**Fonctionnement** :

- Affiche un menu interactif à l'utilisateur
- Permet de générer automatiquement des processus
- Permet de charger un fichier de configuration existant
- Affiche les résultats en texte sur stdout (Gantt textuel, statistiques)
- **Mode principal pour utilisation en ligne de commande**

### 4.3 Intégration complète : Frontend Next.js + Backend C

**Composants Frontend (React)**

**1. Page principale** (`app/page.tsx`)

- Gestion fichiers (créer processus ou charger fichier `.txt`)
- Sélecteur algorithme avec paramètres dynamiques
- Tableau de processus (preview, "Afficher les détails")
- Bouton "Lancer l'Ordonnancement"

**2. AlgorithmSelector** (`components/algorithm-selector.tsx`)

- **Options disponibles** : fifo, priority_preemptive, round-robin, multilevel, multilevel-dynamic, srt
- **Paramètres dynamiques** :
  - `quantum` : visible si round-robin ou multilevel-dynamic
  - `priorityOrder` : visible si priority_preemtive
- Validation saisie utilisateur

**3. ResultsDisplay** (`components/results-display.tsx`)

- **Gantt chart** : timeline interactif (play/pause, zoom)
- **Pie chart** : répartition temps total par processus
- **Bar chart** : temps d'attente vs temps total
- **Tableau détaillé** :
  - Colonnes : id, arrival, execution, waitTime, totalTime, finishTime
  - **Priorité Initiale** : toujours visible
  - **Priorité Finale** : visible **uniquement** pour multilevel_dynamic (après aging)
- **Palette de couleurs** : 20 couleurs distinctes + fallback HSL, déterministe par process ID

**APIs Routes Next.js**

**`POST /api/parse-config`**

- Upload fichier `.txt`
- Appelle `ordonnanceur --parse-config <tmpfile>`
- Renvoie array JSON : `[{ id, arrivalTime, executionTime, priority }, ...]`
- Utilisé pour charger un fichier existant

**`POST /api/schedule`**

- **Payload** : `{ processes: Process[], config: AlgorithmConfig }`
- **Étapes internes** :
  1. Écrit fichier temp (`sched_${timestamp}.txt`)

2. Construit CLI args : `["--api", "--config", tmpPath, "--algo", mappedAlgo, ...]`
3. Appelle `spawn("./ordonnanceur", args)`
4. Parse stdout JSON
5. Cleanup fichier temp

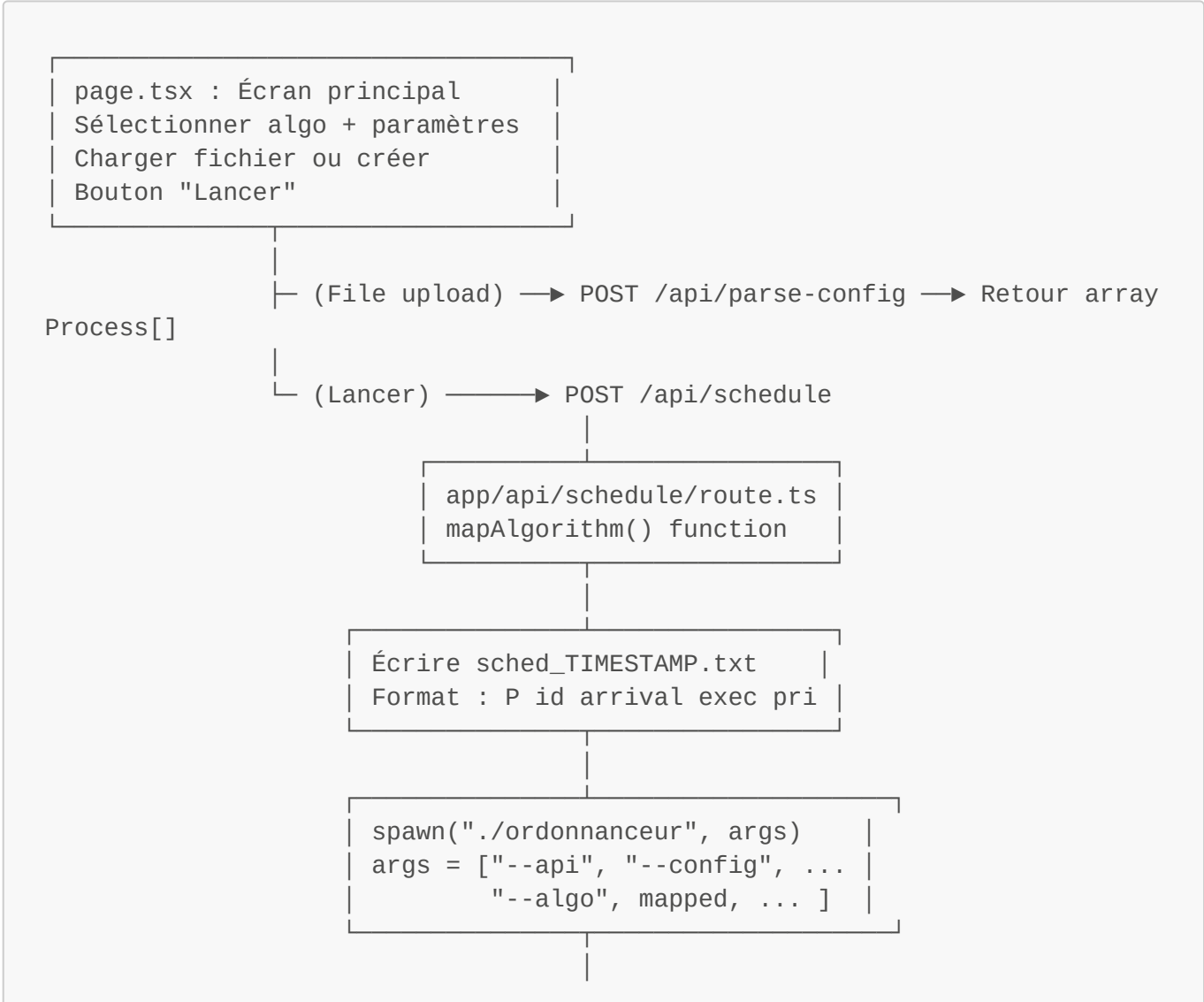
- **Réponse** : `SchedulingResult` : `{ algorithm, ganttData[], processStats[], averageWait, makespan }`

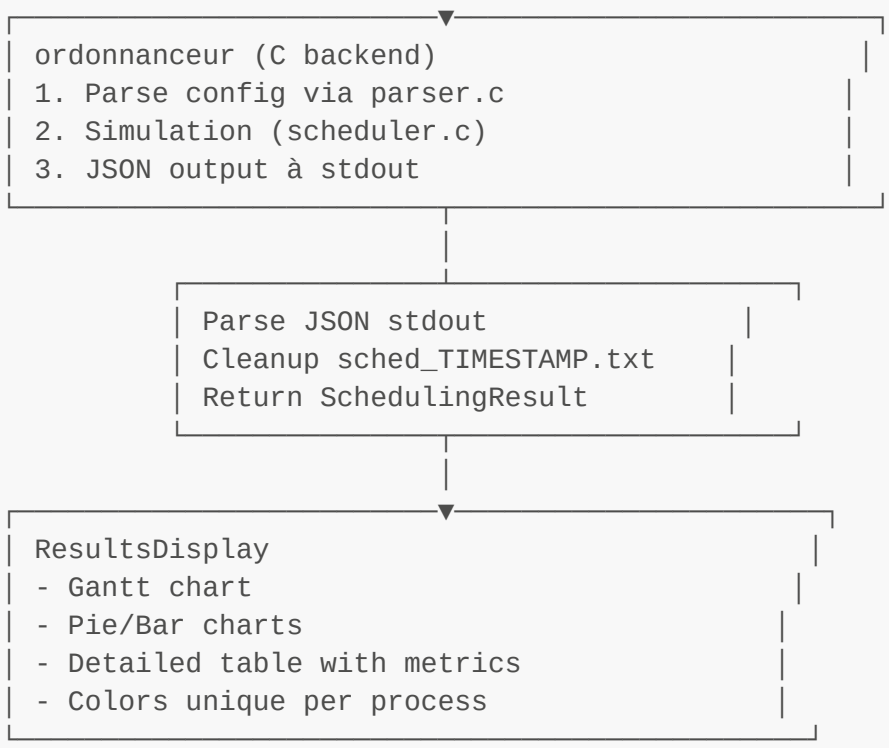
**Backend C (mode `--api`)**

- Lit fichier config via `--config <path>`
- Simule l'algorithme spécifié via `--algo <name>`
- Collecte métriques dans `process_stat` (waitTime, totalTime, finishTime, **finalPriority** pour multilevel_dynamic)
- Génère `ganttData` (timeline des allocations CPU)
- Sortie JSON structurée sur stdout
- Parsée immédiatement par route Next.js → envoyée au client React

### 4.5 Flow d'Exécution Complet

**Frontend → Backend → Frontend :**






# 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 |
| **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 |

| ID | User Story | Priorité |
|---|---|---|
| **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 |
|---|---|---|---|---|
| **Multilevel + Aging** | Ajouter un ordonnancement multi-files avec mécanisme d'aging | 8 pts | Haute | 12 h |
| **SRT** | Version préemptive de SJF (gestion du temps restant) | 8 pts | Haute | 12 h |
| **Génération Config** | Script/programme produisant un fichier valide automatiquement | 4 pts | Moyenne | 6 h |
| **IHM + Gantt** | 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** | `./ordonnanceur` | Humain | Texte + Gantt textuel | CLI local |
| **Direct File** | `./ordonnanceur [fichier]` | Humain | Texte + Gantt textuel | Script shell rapide |
| **API** | `./ordonnanceur --api --config ... --algo ...` | Programme/Script | JSON structuré | Routes Next.js |
| **Parse Only** | `./ordonnanceur --parse-config [fichier]` | 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

**Exemple de sortie JSON (Mode API)** :

```
{
  "algorithm": "roundrobin",
  "ganttData": [
    { "process": "P1", "start": 0, "end": 4 },
    { "process": "P2", "start": 4, "end": 8 },
    { "process": "P1", "start": 8, "end": 10 }
  ],
  "processStats": [
    { "id": "P1", "arrivalTime": 0, "executionTime": 10, "finishTime": 10,
"waitTime": 0 },
    { "id": "P2", "arrivalTime": 2, "executionTime": 6, "finishTime": 8,
"waitTime": 0 }
  ],
  "averageWait": 0,
  "makespan": 10
}
```

## 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 | `name` | Chaîne | Sans espaces (ex: P1, processA) |
| 2 | `arrival_time` | Entier | >= 0 |

| Position | Champ | Type | Contraintes |
|---|---|---|---|
| 3 | `exec_time` | Entier | > 0 (strictement positif) |
| 4 | `priority` | 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 ≥ 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 ≥ 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 |
|---|---|---|---|
| `nb_processes` | Entier | Nombre de processus à générer | 20 |
| `max_arrival_time` | Entier | Temps d'arrivée maximal (min=0) | 100 |
| `min_priority` | Entier | Priorité minimale | 0 |
| `max_priority` | Entier | Priorité maximale | 5 |
| `max_exec_time` | 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 ≥ 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 | `ordonnanceur` | Nom du binaire |
| **SRC_DIR** | Répertoire source | `src` | Où chercher .c principaux |
| **INC_DIR** | Répertoire headers | `include` | Où chercher .h |
| **POL_DIR** | Répertoire politiques | `policies` | Où chercher algorithmes .c |
| **BUILD_DIR** | Répertoire objets | `build` | Où générer .o |

| Variable | Signification | Valeur | Utilité |
|---|---|---|---|
| **SRC** | Liste source | `$(wildcard src/*.c)` | Tous .c dans src/ |
| **POLICIES** | Liste politiques | `$(wildcard policies/*.c)` | Tous .c dans policies/ |
| **OBJ** | Liste objets | Substitution → build/*.o | Fichiers intermédiaires |
| **CC** | Compilateur C | `gcc` | Exécutable compilation |
| **CFLAGS** | Options compilation | `-Wall -Wextra -std=c11 -I$(INC_DIR)` | 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 `policies/`.

**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 src/*.c ou policies/*.c 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'ordonnancement de processus** avec les résultats suivants :

#### Objectifs Techniques Atteints

**✅ 6 algorithmes d'ordonnancement implémentés et fonctionnels**

✅ **Architecture modulaire et extensible**

✅ **Générateur automatique de configurations**

✅ **Parser robuste**

✅ **Compilation automatisée**