
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

École Nationale d'Ingénieurs de Tunis (ENIT)

Département Informatique • 1ère Année Cycle Ingénieur

Décembre 2025

Table des Matières

1. Introduction [Page 3](#)

- 1.1 Objectif du Projet
- 1.2 Fonctionnalités Principales
- 1.3 Portée du Projet

2. Choix des Structures de Données [Page 5](#)

- 2.1 Structure **process** : Le Cœur du Système
- 2.2 Représentation des Données : Tableau Dynamique
- 2.3 Représentation Implicite de la Ready Queue
- 2.4 État des Processus : Machine d'État

3. Choix des Algorithmes d'Ordonnancement [Page 10](#)

- 3.1 FIFO (First-In First-Out)
- 3.2 Priority Preemptive
- 3.3 Round Robin
- 3.4 SRT (Shortest Remaining Time)
- 3.5 Multilevel Queue (Statique)
- 3.6 Multilevel Dynamic (avec Aging)
- 3.7 Tableau Comparatif des Algorithmes

4. Technologies et Architecture [Page 35](#)

- 4.1 Choix des Technologies
- 4.2 Architecture du Projet
- 4.3 Flow d'Exécution
- 4.4 Diagramme de Gantt et Métriques

5. Déroulement du Développement SCRUM [Page 42](#)

- 5.1 Sprint 0 : Initialisation (3 jours)
- 5.2 Sprint 1 : Algorithmes de Base (1 semaine)
- 5.3 Sprint 2 : Algorithmes Avancés (1 semaine)
- 5.4 Product Backlog Complet
- 5.5 Détail des Tâches Sprint 0
- 5.6 Détail des Tâches Sprint 1
- 5.7 Détail des Tâches Sprint 2
- 5.8 Métriques SCRUM

6. Spécifications Techniques [Page 58](#)

- 6.1 Format Fichier Configuration
- 6.2 Générateur Configuration Automatique
- 6.3 Règles de Parsing Détaillées

- 6.4 Algorithme de Parsing
- 6.5 Algorithme de Génération

7. Tests et Validation [Page 70](#)

- 7.1 Tests Unitaires
- 7.2 Tests d'Intégration
- 7.3 Benchmarks de Performance

8. Makefile et Compilation [Page 75](#)

- 8.1 Objectif du Makefile
- 8.2 Variables Principales
- 8.3 Règles Principales
- 8.4 Déclaration PHONY
- 8.5 Flags Compiler Expliqués
- 8.6 Principes et Avantages
- 8.7 Utilisation Pratique

9. Conclusion et Perspectives [Page 82](#)

- 9.1 Résultats Obtenus
 - 9.2 Améliorations Futures
 - 9.3 Apprentissages Clés
-

1. Introduction

Objectif du Projet

Ce projet vise à concevoir et réaliser un **simulateur d'ordonnancement de processus sous Linux** en langage C. L'objectif est d'offrir un outil pédagogique permettant de :

- ☒ **Générer ou lire** un jeu de processus (fichier texte configurable)
- ☒ **Appliquer plusieurs politiques** d'ordonnancement :
 - FIFO (First-In First-Out)
 - Round-Robin avec quantum configurable
 - Priorité préemptive (modes croissant/décroissant)
 - SRT (Shortest Remaining Time)
 - Multilevel Queue (statique)
 - Multilevel Dynamic (avec aging anti-famine)
- ☒ **Collecter des métriques** : temps d'attente, temps de réponse, temps de tour, utilisation CPU
- ☒ **Afficher les résultats** : console et diagramme de Gantt
- ☒ **Être modulaire, configurable et extensible**

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 (GRANDE VALEUR = HAUTE
PRIORITÉ)
    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)
};
```

Justification de Chaque Champ

Champ	Justification	Algorithmes Utilisateurs
name[64]	Identification lisible (P1, ProcessA, etc.)	Tous (affichage)
arrival_time	Détermine quand le processus devient READY	FIFO, Priority, RR, Multilevel
exec_time	Durée immuable totale	Métriques (calcul temps attente)
priority	Support ordonnancement hiérarchique (Grande valeur = haute priorité)	Priority, Multilevel
remaining_time	Temps à exécuter (modifiable)	SRT, Priority, RR, Multilevel
waiting_time	Métrique cumulative d'attente	Métriques finales
status	Gestion des états (READY, RUNNING, ZOMBIE)	Tous les ordonnanceurs
end_time	Date de fin (pour turnaround time)	Métriques (calcul efficacité)
wait_time	Temps d'attente pour aging dynamique	Multilevel Dynamic (anti-famine)

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 (select_fifo)

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 (`select_fifo`)

- La fonction `select_fifo` 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, `select_fifo` 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

Aspect	Évaluation
✗ 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 selon le mode choisi par l'utilisateur (valeur la plus grande ou la plus petite, la valeur la plus grande étant le mode par défaut) **préempte immédiatement** tout processus en cours d'exécution.

Algorithme de Sélection et Simulation

Étape 0 : Initialiser une variable globale définissant le mode de priorité

- Par défaut, la valeur la plus grande correspond à la plus haute priorité

Étape 1 : Définir une fonction de sélection préemptive

Cette fonction permet de choisir le prochain processus à exécuter.

À l'intérieur de cette fonction :

1.1. Ignorer le processus actuellement en cours d'exécution

- Cela permet une préemption immédiate à chaque appel de la fonction

1.2. Mettre à jour le mode global de priorité selon le choix de l'utilisateur

- Le mode peut être basé sur la valeur numérique la plus grande ou la plus petite

1.3. Préparer la recherche du processus le plus prioritaire

- Initialiser un indice "meilleur processus" à vide et une valeur extrême de priorité adaptée au mode choisi

1.4. Parcourir tous les processus

- **1.4.1.** Sélectionner uniquement ceux qui sont arrivés et n'ont pas encore terminé leur exécution
- **1.4.2.** Comparer leur priorité avec la meilleure trouvée jusqu'à présent
- **1.4.3.** Mettre à jour le processus sélectionné si sa priorité est meilleure selon le mode choisi (croissant ou décroissant)

1.5. Retourner l'indice du processus le plus prioritaire

- Retourner "aucun processus" (-1) 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

2.4. Incrémenter le temps et répéter

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.

Modes de Priorité

- **Mode Descending (par défaut)** : Valeur numérique plus grande = priorité plus haute
 - Exemple : Priority 9 > Priority 0
- **Mode Ascending** : Valeur numérique plus petite = priorité plus haute
 - Exemple : Priority 0 > Priority 9 (conventionnel Unix)

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`
- Créer une **file d'attente circulaire** (ready queue) avec indices `head` et `tail`

Étape 2 : Gestion de la Ready Queue

À chaque unité de temps :

2.1. Ajouter les nouveaux arrivants

- Parcourir tous les processus
- Si `arrival_time <= time` ET `remaining_time > 0` ET processus pas déjà en file
 - Ajouter l'indice du processus à `ready[tail]`
 - Incrémenter `tail`

2.2. Vérifier si la queue est vide

- Si `head == tail` (queue vide) :
 - Chercher le prochain temps d'arrivée futur
 - Sauter directement à ce temps (CPU IDLE)
 - Afficher `[IDLE]`

Étape 3 : Sélection et Exécution du Processus

3.1. Extraire le processus en tête de file

- `curr = ready[head]` (premier processus dans la queue)
- Incrémenter `head` (retirer de la queue)

3.2. Calculer le temps d'exécution

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

3.3. 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)

3.4. Exécuter le processus

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

Étape 4 : Gestion des Nouveaux Arrivants Pendant le Quantum

- Pendant l'exécution du quantum, vérifier si de nouveaux processus arrivent
- Pour chaque nouveau processus arrivé entre `(time - run)` et `time` :
 - Ajouter à la ready queue s'il n'y est pas déjà

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

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

- Marquer `end_time = time` (temps de fin)
- 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 :

- `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)

Choix Optimal du Quantum

Quantum	Impact CPU	Réactivité	Équité	Notes
1-2	Très élevé	Excellente	Parfaite	Overhead inacceptable
4 <input checked="" type="checkbox"/>	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
<input checked="" type="checkbox"/> ÉQUITÉ MAXIMALE <input checked="" type="checkbox"/>	Aucun processus attend indéfiniment

Aspect	Évaluation
☑ Pas de famine	Tous progressent
☑ Très réactif	Pas de monopole CPU
☑ Standard moderne	Utilisé partout (Linux, Windows)
☑ 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.

Théoriquement optimal pour minimiser le temps d'attente moyen.

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
☑ OPTIMAL mathématiquement ☑	Meilleur temps d'attente théorique
☑ Temps attente très bon	Résultats excellents
☑ Peu de préemptions	Comparé à Priority
☑ Utile comme benchmark	Référence de comparaison
✗ FAMINE des longs processus ⚠	Processus long jamais sélectionné
✗ REQUIERT FUTUR	exec_time doit être connu à l'avance
✗ Irréaliste en pratique	Pas possible en vrai système d'exploitation
⚠ Utilisé pour benchmark	Comparer autres algos contre SRT

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

Initialiser :

- `best_prio` = -1 (priorité maximum trouvée)
- `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`
 - Marquer `processes_ready = 1`

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

Étape 2 : Logique Round-Robin pour la Priorité Maximum

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)
- `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` :
 - **Retourner `idx`** (prochain processus à exécuter)

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

Avantages et Inconvénients

Aspect	Évaluation
<input checked="" type="checkbox"/> Différencie types processus	Traitement adapté
<input checked="" type="checkbox"/> Priorités fixes = déterministe	Comportement prévisible
<input checked="" type="checkbox"/> Bon pour systèmes mixtes	Interactif + batch
<input checked="" type="checkbox"/> FAMINE des basses priorités ⚠	Prio 2 peut attendre indéfiniment
<input checked="" type="checkbox"/> Rigide	Pas d'adaptation aux changements
<input checked="" type="checkbox"/> Pas équitable	Basse prio jamais s'exécute si haute prio active

Cas d'Usage Réel

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

3.6 Multilevel Feedback Queue (Dynamique) ★ MODERNE

Principe

IMPORTANT : Dans le code actuel (`multilevel_dynamic.c`), la fonction `select_multilevel_dynamic` est **identique** à `select_multilevel` (statique).

La logique de **feedback loop** (pénalité dynamique) doit être implémentée dans la **boucle principale de simulation** (fichier de test ou scheduler principal), **pas** dans la fonction de sélection.

Algorithme de Sélection (fonction `select_multilevel_dynamic`)

Actuellement, cette fonction est identique à la version statique.

Entrées :

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

Logique : (voir Section 3.5 Multilevel Statique - identique)

1. Trouver la priorité maximum parmi processus prêts
2. Si processus courant a toujours la priorité max ET quantum non expiré → continuer
3. Sinon → Round-Robin circulaire pour trouver le prochain

Implémentation du Feedback Loop (dans la boucle de simulation)

Pour implémenter le comportement dynamique, la boucle principale doit :

Étape 1 : Après chaque exécution de quantum, analyser le comportement

```
Si (processus a consommé quantum COMPLET) ET (remaining_time > 0) :  
    → Pénaliser : priority-- (descendre d'un niveau)  
    → Limite : priority ne descend pas sous niveau plancher (ex: 1)  
    → Reset quantum_counter = 0
```

Étape 2 : Avant chaque sélection, vérifier aging (optionnel anti-famine)

```
Pour chaque processus en attente :  
    Si (temps_sans_execution > seuil_aging) :  
        → Promotion : priority++ (monter d'un niveau)  
        → Limite : priority ne dépasse pas niveau max
```

Étape 3 : Appeler la fonction de sélection avec priorités mises à jour


```
next_process = select_multilevel_dynamic(procs, n, time, current, quantum_expired)
```

Avantage : Anti-Famine avec Aging

Conséquence : Aucun processus n'attendra **indéfiniment** → ☒ Anti-famine garanti

Avantages et Inconvénients

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

4. Technologies et Architecture

4.1 Choix des Technologies

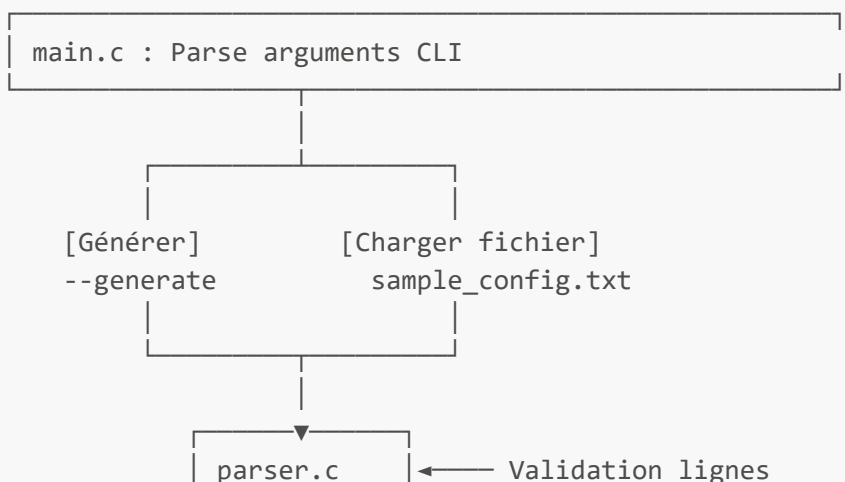
Technologie	Justification	Bénéfices
Langage C	Requis ; bas niveau ; standard académique	Proximité système, performance
Git + GitHub	Contrôle version ; collaboration ; historique	Traçabilité modifications
Scrum/Agile	Gestion itérative ; sprints ; équipe	Planification adaptable
Trello	Tableau Kanban ; 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

```
projet-ordonnanceur-linux/
├── src/                                # Code source principal
│   ├── main.c                         # Point d'entrée, CLI, orchestration
│   ├── parser.c                       # Lecture et validation fichiers config
│   ├── scheduler.c                    # Moteur simulation et boucle temps
│   ├── generate_config.c              # Générateur configs aléatoires
│   └── utils.c                        # Fonctions utilitaires (logs, affichage)
└── include/                           # Headers (.h) publics
```

process.h	# Def struct process, états, constantes
scheduler.h	# Prototypes moteur, politiques
parser.h	# Prototypes parsing, validation
utils.h	# Prototypes utilitaires
generate_config.h	# Prototypes générateur
policies/	# Implémentation des ordonnanceurs
fifo.c	# FIFO non-préemptif
priority_preemptive.c	# Priority avec préemption
roundrobin.c	# Round Robin avec quantum
srt.c	# SRT (Shortest Remaining Time)
multilevel.c	# Multilevel statique
multilevel_dynamic.c	# Multilevel avec aging (MODERNE)
config/	# Fichiers de configuration
sample_config.txt	# Exemple d'utilisation
config_generated_*.txt	# Configs générées dynamiquement
tests/	# Tests unitaires et d'intégration
test_fifo.c	
test_priority.c	
test_roundrobin.c	
test_multilevel.c	
test_multilevel_dynamic.c	
test_parser.c	
testfile.txt	# Fichier test pour parser
build/	# Fichiers objets (généré par make)
*.o	# Fichiers compilés
Makefile	# Automatisation compilation/tests
README.md	# Guide utilisateur
Documentation.md	# Cette documentation
LICENSE	# MIT License
ANALYSE_LICENCES.md	# Analyse légale code ChatGPT/Grok

4.3 Flow d'Exécution





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)

Paramètre	Valeur
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
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	10 pts	Moyenne	15 h

Total Sprint 2 : 30 points (45 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	30	45	4
TOTAL		64	98.5	15

6. Spécifications Techniques

6.1 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
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
```

Règles de Parsing Détaillées

Cas	Détection	Action	Exemple
Ligne vide	Zéro caractères non-blancs	Ignorer	\n ou
Commentaire	1er caractère non-blanc = #	Ignorer	# Configuration...
Processus valide	4 tokens + conversions OK + valeurs acceptables	Parser struct	P1 0 250 3
Moins de 4 tokens	Split retourne < 4 éléments	Warning + ignorer	P1 0 250 (3 champs)
Champ non-numérique	atoi() échoue sur token[i]	Warning + ignorer	P1 zero 250 3
arrival_time < 0	atoi(token[1]) < 0	Erreur fatale	P1 -5 250 3
exec_time <= 0	atoi(token[2]) <= 0	Erreur fatale	P1 0 0 3 ou P1 0 -10 3
priority hors intervalle	atoi(token[3]) hors [0, MAX]	Warning (mode strict) ou ignorer	P1 0 250 99 (si MAX=10)

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 `strtok_r()` 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 \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)

6.2 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 $\geq \text{min_priority}$)
- **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 : `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: `config_generated_20251206_143052.txt`)
- **Validité** : Fichier généré est automatiquement **valide** (respecte toutes les règles)
- **Sortie** : Affichage confirmation + chemin fichier

8. Makefile et Compilation

8.1 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)
- **Lancer les tests** unitaires
- **Éviter la compilation manuelle** (pas besoin de taper gcc à chaque fois)

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	<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 → build/*.o	Fichiers intermédiaires
CC	Compilateur C	<code>gcc</code>	Exécutable compilation
CFLAGS	Options compilation	<code>-Wall -Wextra -std=c11 -I\$(INC_DIR)</code>	Warnings + includes

8.3 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
```

8.4 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".

8.5 Flags Compiler Expliqués

Flag	Signification	Utilité	Exemple
-Wall	"Warn All"	Affiche TOUS les warnings	Captures variables inutilisées
-Wextra	Warnings supplémentaires	Rigueur accrue	Détecte plus de problèmes
-std=c11	Standard C11	Assure compatibilité	Types bool, uint64_t, etc
-I(dir)	Include directory	Ajoute répertoire headers	-I\$(INC_DIR) cherche dans include/

Note : Le Makefile actuel n'utilise pas **-g** (debug) ni **-O2** (optimisation) par défaut.

8.6 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

8.7 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
```

Conclusion

Cette documentation couvre **intégralement** :

☒ **Choix Structures de Données**

☒ **Choix Algorithmes**

☒ **SCRUM Détaillé**

Statut : ☒ **Complet et Conforme aux Consignes**