

# RAPPORT TECHNIQUE

## Politique d'ordonnancement FIFO (First In, First Out)

L'algorithme FIFO, également appelé FCFS (First Come, First Served), est une politique d'ordonnancement **non préemptive** qui exécute les processus **dans l'ordre de leur arrivée**. Chaque processus s'exécute entièrement avant que le suivant ne soit lancé, sans interruption.

Les processus sont d'abord triés selon leur temps d'arrivée afin de respecter le principe FIFO :

```
void schedule_fifo(Process processes[], int num_processes, int quantum) {
    // Trier les processus par temps d'arrivée
    qsort(processes, num_processes, sizeof(Process), compare_arrival_time)
```

Lors de l'exécution, si le processeur est libre avant l'arrivée d'un processus, le système attend jusqu'à son arrivée. Le processus démarre ensuite à l'instant courant :

```
if (current_time < p->arrival_time) {
    current_time = p->arrival_time;
}

p->start_time = current_time;
```

Étant donné que FIFO est non préemptif, chaque processus s'exécute en **une seule tranche** correspondant à son temps d'exécution (burst time) :

```
if (p->num_slices < MAX_SLICES) {
    p->slices[p->num_slices].start = current_time;
    p->slices[p->num_slices].duration = p->burst_time;
    p->num_slices++;
}
```

Enfin, le temps courant est mis à jour, le processus se termine et son temps restant devient nul :

```
current_time += p->burst_time;
p->finish_time = current_time;
p->remaining_time = 0;
```

Cette politique est simple à implémenter, mais elle peut provoquer un **effet de convoi**, où les processus courts subissent un temps d'attente important lorsqu'un processus long est exécuté en premier.

## Politique d'ordonnancement par priorité préemptive

L'algorithme de **priorité préemptive** exécute en priorité le processus ayant la **plus grande priorité** (valeur numérique la plus élevée). Contrairement à FIFO, un processus peut être **interrompu (préempté)** si un autre processus de priorité supérieure arrive. Les processus sont d'abord triés par **temps d'arrivée** pour gérer correctement l'ordre d'arrivée.

### Sélection du processus

À chaque instant, le CPU choisit le processus **prêt avec la priorité la plus haute** :

```
| Process *highest_priority_process = NULL;
| int max_priority = INT_MIN; // Chercher le maximum au lieu de min
|
// Parcourir tous les processus pour trouver celui de plus haute
for (int i = 0; i < num_processes; i++) {
    Process *p = &processes[i];
|
    // Le processus doit être arrivé et ne doit pas être terminé
    if (p->arrival_time <= current_time && p->remaining_time > 0)
        // Priorité statique: la valeur la plus grande est la plus élevée
        if (p->priority > max_priority) {
            max_priority = p->priority;
            highest_priority_process = p;
        }
        // Règle de départage : si priorités égales, FIFO (géré par l'ordre d'arrivée)
}
}
```

### Gestion du CPU inactif

Si aucun processus n'est prêt, le CPU **attend l'arrivée du prochain processus** :

```
if (highest_priority_process == NULL) {
    int next_arrival_time = INT_MAX;
    int next_arrival_index = -1;
```

### Exécution et préemption

Le processus courant s'exécute pour une durée déterminée par **son temps restant** ou l'arrivée d'un **processus plus prioritaire** :

```
int time_slice = current_process->remaining_time;
|
// Trouver l'événement de temps suivant (arrivée d'un processus)
int next_event_time = INT_MAX;
for (int i = 0; i < num_processes; i++) {
    Process *p_next = &processes[i];
|
    // Si un autre processus arrive plus tard mais avec une meilleure priorité
    if (p_next->arrival_time > current_time &&
        p_next->arrival_time < next_event_time &&
        p_next->priority > current_process->priority)
    {
        next_event_time = p_next->arrival_time;
    }
}
|
// Le temps d'exécution est limité soit par le temps restant, soit par l'heure de l'événement
if (next_event_time != INT_MAX) {
    time_slice = (next_event_time - current_time) < time_slice
        ? (next_event_time - current_time) : time_slice
}
```

## Fin ou préemption

- Si `remaining_time == 0` → processus terminé
- Sinon, il est préempté par un processus de priorité supérieure

```
// 2e. Gérer la terminaison ou la préemption
if (current_process->remaining_time == 0) {
    current_process->finish_time = current_time;
    completed_processes++;
    printf("Temps %d: Le processus '%s' termine.\n", current_time,
} else if (current_time == next_event_time) {
    // Préemption par un nouvel arrivant
    printf("Temps %d: Le processus '%s' est preempté par un nouve
} else {
    // Le processus continue son exécution jusqu'à la prochaine it
}
```

## Politique d'ordonnancement Round-Robin (RR)

L'algorithme **Round-Robin** (RR) est une politique **préemptive** qui partage le CPU entre les processus en utilisant un **quantum de temps fixe**. Chaque processus s'exécute pendant **au maximum un quantum**, puis est remplacé à la fin de la file d'attente s'il n'est pas terminé, permettant ainsi un partage équitable du CPU.

### Gestion de la file d'attente

Les processus sont triés par **temps d'arrivée** et ajoutés à une **file prête** (`ready_queue`) dès qu'ils arrivent :

```
// Ajouter les processus arrivés
while (next_arrival_index < num_processes &&
      processes[next_arrival_index].arrival_time <= current_time) {
    enqueue(&ready_queue, &processes[next_arrival_index]);
    next_arrival_index++;
}
```

### Exécution par quantum

Le processus en tête de la file s'exécute pour **le quantum ou le temps restant**, puis est soit terminé, soit préempté et remis en fin de file :

```
int execution_time = (current_process->remaining_time < quantum) ?
                     current_process->remaining_time : quantum;

current_time += execution_time;
current_process->remaining_time -= execution_time;

else {
    enqueue(&ready_queue, current_process);
    printf("Temps %d: '%s' preempte (restant: %d).\n",
}
```

## Enregistrement des tranches (Gantt)

Chaque exécution partielle est enregistrée comme **slice** pour représenter le diagramme de Gantt :

```
if (current_process->num_slices < MAX_SLICES) {
    current_process->slices[current_process->num_slices].start = current_
    current_process->slices[current_process->num_slices].duration = execu
    current_process->num_slices++;
}
```

## Politique Multi-Level Queue (MLQ) – Priorité + Round-Robin

L'algorithme MLQ combine priorité préemptive et Round-Robin pour gérer plusieurs processus :

1. Le processus de priorité la plus élevée est exécuté en premier.
2. Si plusieurs processus ont la même priorité, ils s'exécutent selon Round-Robin avec un quantum choisi par l'utilisateur.
3. Si un processus de priorité supérieure arrive pendant l'exécution, il peut préempter le processus courant.

### Sélection du processus le plus prioritaire

Le CPU choisit la file non vide de priorité la plus haute et exécute le processus en tête de file :

```
for (int p = max_priority; p >= min_priority; p--) {
    if (p >= MAX_QUEUES) continue;
    if (!is_empty(&queues[p].queue)) {
        current = queues[p].queue.elements[queues[p].queue.front];
        selected_queue = p;
        break;
    }
}
```

### Round-Robin pour les processus de même priorité

Chaque file de priorité utilise un quantum fixe et les processus sont ré-enfilés s'ils ne sont pas terminés :

```
int time_slice = queues[selected_queue].quantum;
if (current->remaining_time < time_slice) {
    time_slice = current->remaining_time;
}
```

```
current_time += time_slice;
current->remaining_time -= time_slice;
```

```
// Ré-enfiler si pas terminé (seulement dans sa file actuelle)
if (current->remaining_time > 0) {
    dequeue(&queues[selected_queue].queue); // on l'enlève d'abord
    enqueue(&queues[current->priority].queue, current); // on le
```

## Gestion des tranches pour le Gantt

Chaque exécution partielle est enregistrée pour le diagramme de Gantt :

```
current->slices[current->num_slices].start = current_time;
current->slices[current->num_slices].duration = time_slice;
current->num_slices++;
```

## Politique Multi-Level Queue avec Aging

Cette variante du **MLQ** combine **priorité préemptive**, **Round-Robin par niveau** et un mécanisme d'**aging** pour éviter l'inversion de priorité.

1. Le **processus avec la priorité dynamique la plus élevée** s'exécute en premier.
2. Si plusieurs processus ont la **même priorité dynamique**, ils sont exécutés en **Round-Robin** avec un **quantum interne fixe**.
3. Chaque unité de temps exécutée **diminue la priorité dynamique** du processus (aging), ce qui favorise les processus moins prioritaires qui attendent depuis longtemps.

## Initialisation des processus

Chaque processus démarre avec sa **priorité statique**, un **quantum interne** pour le RR et le temps restant :

```
p->remaining_time    = p->burst_time;
p->dynamic_priority = p->priority;
p->quantum_left      = AGING_QUANTUM;
```

## Sélection et exécution

À chaque unité de temps, le CPU choisit **le niveau de priorité le plus élevé non vide** et exécute 1 ms du processus en tête :

```
// >>> Recuperer le premier processus de ce
Process *current = dequeue(&queues[lvl]);
```

```
current->remaining_time--;
current->quantum_left--;
```

## Application de l'aging

La priorité dynamique du processus diminue à chaque exécution pour favoriser l'avancement des processus moins prioritaires :

```
// sa priorité dynamique diminue de 1
if (current->dynamic_priority > 0) {
    current->dynamic_priority--;
}
```

## Réinsertion et Round-Robin

Si le processus **n'est pas terminé**, il est réinséré dans la file correspondant à sa **nouvelle priorité dynamique**. Le quantum interne est réinitialisé si nécessaire :

```
if (current->quantum_left == 0) {
    current->quantum_left = AGING_QUANTUM;
}

int new_pr = current->dynamic_priority;
if (new_pr < 0) new_pr = 0;
if (new_pr > MAX_LEVELS) new_pr = MAX_LEVELS;

enqueue(&queues[new_pr], current);
```

## Enregistrement des tranches pour le Gantt

Chaque unité de temps est enregistrée comme **slice** pour représenter l'exécution dans le diagramme de Gantt :

```
current->slices[s].start    = time;
current->slices[s].duration = 1;
current->num_slices++;
```

## Makefile – Gestion de la compilation et des bibliothèques dynamiques

Le **Makefile** est un composant essentiel du projet, permettant de **compiler automatiquement le code source**, de créer les **bibliothèques dynamiques pour les politiques d'ordonnancement**, et de **gérer l'exécution et l'installation**.

### 1/ Variables et configuration

#### Compilateur et flags :

Le compilateur **gcc** est utilisé avec des flags pour activer les avertissements, le debug et le standard C99. Les flags GTK+ 3 sont automatiquement détectés pour l'interface graphique.

```
CC = gcc

# Détection automatique des flags GTK+ 3.0
GTK_CFLAGS = $(shell pkg-config --cflags gtk+-3.0)
GTK_LIBS   = $(shell pkg-config --libs gtk+-3.0)

# Flags de compilation
CFLAGS = -Wall -Wextra -g -std=c99 -fPIC -pthread $(GTK_CFLAGS)

# Flags de linkage
LDFLAGS = $(GTK_LIBS) -ldl -lpthread -lm    # -ldl pour dlopen,
```

### Répertoires :

Les **répertoires** servent à organiser le code et les fichiers du projet. Ils indiquent au compilateur où trouver les sources, les headers et les politiques.

```
# Répertoires
SRCDIR = src
INCLDIR = includes
POLDIR = policies
CONFIGDIR = config
```

### Sources et bibliothèques dynamiques :

#### Sources (.c) :

Ce sont les fichiers du code principal qui définissent le fonctionnement de l'ordonnanceur :

Ils sont compilés en fichiers objets (.o) avant de créer l'exécutable.

Contient la logique principale, les fonctions utilitaires et l'interface graphique.

```
# Fichiers sources principaux (gui.c est ton fichier avec l'IHM Gantt)
SRC_FILES = main.c parser.c simulation.c utils.c gui.c
SRC       = $(addprefix $(SRCDIR)/, $(SRC_FILES))
```

#### Bibliothèques dynamiques (.so) :

Ce sont les fichiers compilés à partir des politiques d'ordonnancement (dans policies/)

```
# Bibliothèques dynamiques (.so) des politiques
POL_SOURCES = $(wildcard $(POLDIR)/*.c)
POL_SO      = $(POL_SOURCES:.c=.so)
```

Les fichiers .c sont compilés en .o pour créer l'exécutable final, et en .so pour les bibliothèques dynamiques, permettant de charger les politiques à la volée sans recompiler le programme.

## 2/ Compilation des fichiers

#### Fichiers sources principaux → objets :

Chaque fichier .c dans src/ (ex : main.c, simulation.c, utils.c, gui.c, parser.c) est compilé en fichier objet .o grâce à la règle :

```
# Compilation des fichiers .c → .o
$(SRCDIR)/%.o: $(SRCDIR)/%.c
    $(CC) $(CFLAGS) -I$(INCLDIR) -c $< -o $@
```

Ces fichiers objets sont ensuite liés pour créer l'**exécutable final** ordonnanceur :

```
$(EXECUTABLE): $(OBJECTS)
    $(CC) $(CFLAGS) -rdynamic $(LDLIBS) -o $@
```

#### Politiques dynamiques (.so) :

Les fichiers .c dans policies/ sont compilés en **bibliothèques partagées .so**, chargées dynamiquement par le programme :

```
$(POLDIR)/%.so: $(POLDIR)/%.c
    @echo "Construction de la politique $@"
    $(CC) $(CFLAGS) -I$(INCLDIR) -c $< -o $(POLDIR)/*.o
    $(CC) -shared $(POLDIR)/*.o -o $@
    rm -f $(POLDIR)/*.o
```

### 3/ Règles pratiques

-make → Compile tout (exécutable + bibliothèques).

-make clean → Supprime tous les fichiers .o, .so et l'exécutable.

-make re → Nettoie puis recompile complètement.

```
clean:
    rm -f $(OBJECTS) $(EXECUTABLE)
    rm -f $(POLDIR)/*.o $(POLDIR)/*.so

# Recompiler tout proprement
re: clean all

# Règle pratique pour exécuter directement après compilation
run: all
    @echo "Lancement avec un fichier de test (exemple : processes.txt)"
    ./$(EXECUTABLE) $(CONFIGDIR)/processes.txt # change le fichier si tu
```

#### Installation système ou locale :

Le Makefile copie l'exécutable et les bibliothèques dans /usr/local/bin si possible, sinon dans un dossier local ./bin :

```
INSTALL_DIR = /usr/local/bin
install: $(EXECUTABLE) $(POL_SO)
    @echo "Installation dans $(INSTALL_DIR)..."
    @if [ -w $(INSTALL_DIR) ]; then \
        cp $(EXECUTABLE) $(INSTALL_DIR)/; \
        mkdir -p $(INSTALL_DIR)/policies; \
        cp $(POL_SO) $(INSTALL_DIR)/policies/; \
        echo "Installation réussie dans $(INSTALL_DIR)"; \
    fi
```