Planificador MLFQ

Explicación del Código en C++

David Bayona

3 de octubre de 2025

Índice

1. Código en C++ explicado por fragmentos

1.1. Definición de la clase Process

C++

```cpp
class Process {
public:
    string tag;         // Nombre del proceso
    int AT, BT, Q, Pr; // Arrival Time, Burst Time, Queue,
    Priority

    // Metricas
    int CT, WT, RT, TAT;
    bool firstRun;
    int remaining;
    int lastExec; // ultima vez que estuvo en CPU (para WT)

    Process(string t, int at, int bt, int q, int pr)
        : tag(t), AT(at), BT(bt), Q(q), Pr(pr),
          CT(0), WT(0), RT(0), TAT(0),
          firstRun(true), remaining(bt), lastExec(at) {}
};
```

Explicación

Este bloque define la clase `Process`, que encapsula la información de un proceso:

- `tag`: nombre del proceso.
- `AT, BT`: *Arrival Time* y *Burst Time*.
- `Q, Pr`: identificador de cola y prioridad (no usados aún, pero se guardan).
- `CT, WT, RT, TAT`: métricas calculadas (Completion, Waiting, Response, Turnaround).
- `firstRun`: marca si el proceso se ejecuta por primera vez (necesario para RT).
- `remaining`: tiempo restante por ejecutar.
- `lastExec`: último instante en que el proceso salió del CPU (usado para recalcular WT).

1.2. Función MLFQ

C++

```cpp
void mlfq(vector<Process> &processes, vector<int> quantums) {
    int time = 0;
    int n = processes.size();
    int completed = 0;

    // Orden lexicografico si AT es igual
    sort(processes.begin(), processes.end(),
         [](const Process &a, const Process &b) {
        if (a.AT == b.AT) return a.tag < b.tag;
        return a.AT < b.AT;
```

```
    });
```

Explicación

Aquí comienza la función principal:

- Se inicializa el reloj de CPU (`time`), el número de procesos `n` y un contador de completados.
- Antes de ejecutar, se ordenan los procesos primero por tiempo de llegada y en caso de empate, por orden lexicográfico de la etiqueta. Esto asegura justicia inicial.

C++

```
    int level = 0;
    while (completed < n) {
        bool avanzamos = false;

        for (int i = 0; i < n; i++) {
            Process &p = processes[i];
            if (p.remaining <= 0) continue;

            avanzamos = true;
            if (time < p.AT) time = p.AT;

            if (p.firstRun) {
                p.RT = time - p.AT;
                p.WT = p.RT;
                p.firstRun = false;
            } else {
                p.WT += (time - p.lastExec);
            }
```

Explicación

Este bucle recorre todos los procesos en cada nivel de quantum:

- Si el proceso aún no ha llegado, se avanza el reloj hasta su `AT`.
- Si es la primera ejecución, se calculan `RT` y el primer `WT`.
- Si no, se acumula el tiempo de espera usando la diferencia entre `time` y `lastExec`.

C++

```
            int exec = min(quantums[level], p.remaining);
            time += exec;
            p.remaining -= exec;
            p.lastExec = time;

            if (p.remaining == 0) {
                p.CT = time;
                p.TAT = p.CT - p.AT;
                completed++;
            }
        }

```

```
            if (level < (int)quantums.size() - 1) level++;
            else if (!avanzamos) break;
        }
    }
```

Explicación

Aquí ocurre la ejecución:

- Se ejecuta el proceso por el mínimo entre el quantum actual y su tiempo restante.
- Se avanza el reloj global y se actualiza `lastExec`.
- Si el proceso termina, se calculan `CT` y `TAT`.

Luego, se avanza al siguiente nivel de quantum (RR(3) → RR(5) → RR(6) → RR(20)). Si ya estamos en el último nivel, se mantiene hasta que todos los procesos acaben.

1.3. Impresión de Resultados

C++

```
void printResults(const vector<Process> &processes) {
    cout << "# RR(3), RR(5), RR(6), RR(20).\n";
    cout << "# etiqueta; BT; AT; Q; Pr; WT; CT; RT; TAT\n";

    double avgCT=0, avgWT=0, avgRT=0, avgTAT=0;

    for (auto &p : processes) {
        cout << p.tag << ";" << p.BT << "; " << p.AT << "; "
             << p.Q << "; " << p.Pr << "; "
             << p.WT << "; " << p.CT << "; "
             << p.RT << "; " << p.TAT << "\n";

        avgCT += p.CT; avgWT += p.WT;
        avgRT += p.RT; avgTAT += p.TAT;
    }

    int n = processes.size();
    cout << "WT=" << fixed << setprecision(1) << avgWT / n
         << "; CT=" << avgCT / n
         << "; RT=" << avgRT / n
         << "; TAT=" << avgTAT / n << ";\n";
}
```

Explicación

Esta función imprime la tabla de resultados y los promedios:

- Cada proceso muestra: etiqueta, BT, AT, Q, Pr, WT, CT, RT, TAT.
- Al final se muestran los promedios de cada métrica, con un decimal de precisión.

1.4. Función Main

C++

```cpp
int main() {
    vector<Process> processes;
    string line;

    while (getline(cin, line)) {
        if (line.empty() || line[0] == '#') continue;
        stringstream ss(line);
        string seg;

        getline(ss, seg, ';'); string tag = seg;
        getline(ss, seg, ';'); int bt = stoi(seg);
        getline(ss, seg, ';'); int at = stoi(seg);
        getline(ss, seg, ';'); int q  = stoi(seg);
        getline(ss, seg, ';'); int pr = stoi(seg);

        Process p(tag, at, bt, q, pr);
        processes.push_back(p);
    }

    vector<int> quantums = {3, 5, 6, 20};
    mlfq(processes, quantums);
    printResults(processes);
    return 0;
}
```

Explicación

En `main`:

1. Se leen los procesos desde entrada estándar en el formato `Etiqueta;BT;AT;Q;Pr`.
2. Se almacenan como objetos `Process`.
3. Se define el conjunto de quantums de la MLFQ.
4. Se ejecuta el planificador y se imprimen los resultados.

2. Resultados de Ejecución

2.1. Input 3

```
A;40; 0; 1; 5; 23; 63; 0; 63
B;9; 0; 1; 4; 22; 31; 3; 31
C;14; 0; 2; 3; 23; 37; 6; 37
WT=22.7; CT=43.7; RT=3.0; TAT=43.7;
```

2.2. Input 4

```
B;4; 0; 5; 5; 6; 10; 0; 10
C;5; 0; 5; 5; 7; 12; 3; 12
A;10; 1; 5; 5; 8; 19; 5; 18
WT=7.0; CT=13.7; RT=2.7; TAT=13.3;
```

2.3. mlq007.txt

```
K;9; 0; 2; 5; 28; 37; 0; 37
L;13; 0; 3; 4; 29; 42; 3; 42
X;11; 0; 1; 3; 34; 45; 6; 45
Y;17; 0; 2; 2; 37; 54; 9; 54
Z;4; 0; 3; 1; 32; 36; 12; 36
WT=32.0; CT=42.8; RT=6.0; TAT=42.8;
```

2.4. mlq021.txt

```
P1;10; 0; 2; 5; 40; 50; 0; 50
P2;13; 0; 2; 4; 42; 55; 3; 55
P3;18; 1; 1; 3; 62; 81; 5; 80
P4;15; 2; 3; 3; 65; 82; 7; 80
P5;16; 3; 1; 2; 65; 84; 9; 81
P6;12; 4; 3; 1; 61; 77; 11; 73
WT=55.8; CT=71.5; RT=5.8; TAT=69.8;
```

3. Conclusiones

- El código implementa un planificador MLFQ correcto con distintos niveles de Round Robin.
- Los resultados muestran los tiempos promedio de **WT, CT, RT y TAT**.
- El sistema es flexible: basta con cambiar el vector de quantums para modificar el comportamiento.