OS LAB 5

U18CO021: SAHIL BONDRE

Assignment-5

Write a c/Java program for simulation of (1). Shortest Job First (SJF) (2). Shortest Remaining Time First (SRTF) CPU scheduler.

Program should maintain Ready_Q using process pointers. Each Process should have cpu_time and arrival_time . Cpu_time and arrival_time should be generated randomly. Demonstrate processes context switch according to SRTF Scheduling.

process.hpp

```
#if !defined(PROCESS_BLOC)
#define PROCESS_BLOC

typedef struct Process {
  Process(int pid, int cpu_time, int arrival_time);
  bool operator<(const Process &rhs) const;
  int pid;
  int cpu_time;
  int arrival_time;
} Process;

void print_process(const Process &p);

#endif // PROCESS_BLOC
```

process.cpp

```
#include "process.hpp"
#include <iostream>

using namespace std;

Process::Process(int pid, int cpu_time, int arrival_time) {
  this->pid = pid;
  this->cpu_time = cpu_time;
  this->arrival_time = arrival_time;
}
```

```
bool Process::operator<(const Process &rhs) const {
  if (this->arrival_time == rhs.arrival_time)
    return this->cpu_time < rhs.cpu_time;
  return this->arrival_time < rhs.arrival_time;
}

void print_process(const Process &p) {
  cout << "P" << p.pid << ": { AT = " << p.arrival_time
       << ", BT = " << p.cpu_time << "}\n";
}
```

main.cpp

```
#include "process.hpp"
#include <algorithm>
#include <chrono>
#include <iomanip>
#include <iostream>
#include <list>
#include <queue>
#include <random>
#include <thread>
#include <vector>

#define PROCESS_COUNT 5
#define PROCESS_ARRIVAL_MIN 1
#define PROCESS_ARRIVAL_MAX 4
#define PROCESS_BURST_MIN 2
#define PROCESS_BURST_MAX 6
#define SLEEP_DELAY 10 // ms

using namespace std;

void sleep(int ms) {
  chrono::milliseconds timespan(ms); // or whatever
  this_thread::sleep_for(timespan);
}

int random(int min, int max) {
  static bool first = true;
  if (first) {
    srand(time(NULL));
    first = false;
```

```
  }
  return min + rand() % ((max + 1) - min);
}

void print_solution_table(const vector<vector<int>> &st) {
  cout << "\n===== Solution Table =====\n";
  vector<double> avg(7, 0);
  int width = 5;
  cout << left << setw(width) << "PID" << setw(width) << "AT" <<
setw(width)
       << "BT" << setw(width) << "RT" << setw(width) << "CT" <<
setw(width)
       << "TAT" << setw(width) << "WT"
       << "\n";
  for (int i = 1; i < st.size(); ++i) {
    auto row = st[i];
    for (int j = 0; j < 7; ++j) {
      avg[j] += row[j];
    }
    cout << left << setw(width) << row[0] << setw(width) << row[1]
         << setw(width) << row[2] << setw(width) << row[3] <<
setw(width)
         << row[4] << setw(width) << row[5] << setw(width) << row[6] <<
"\n";
  }

  for (int j = 0; j < 7; ++j) {
    avg[j] /= (double)st.size() - 1;
  }

  cout << left << setw(width) << "avg" << setw(width) << avg[1] <<
setw(width)
       << avg[2] << setw(width) << avg[3] << setw(width) << avg[4]
       << setw(width) << avg[5] << setw(width) << avg[6] << "\n";
}

void sjf(vector<Process> processes) {
  cout << "\n====== SJF ======";
  cout << "\n===== Gantt Chart =====";
  int time = 1;
  vector<vector<int>> solution_table(processes.size() +
                                     1); // pid, at, bt, rt, ct, tat, wt

  // fill solution table
  for (auto p : processes) {
    solution_table[p.pid].push_back(p.pid);          // pid
```

```
    solution_table[p.pid].push_back(p.arrival_time); // at
    solution_table[p.pid].push_back(p.cpu_time);     // bt
  }

  list<Process> ready_queue;
  Process current_process = Process(-1, -1, -1);
  bool is_process_running = false;
  vector<Process>::iterator next_process = processes.begin();
  while (next_process != processes.end() || !ready_queue.empty() ||
         is_process_running) {
    cout << "\nTime: " << time << "\n";

    // fill ready_queue
    while (next_process != processes.end() &&
           next_process->arrival_time <= time) {
      ready_queue.push_back(*next_process);
      next_process++;
    }
    cout << "Ready Queue Size " << ready_queue.size() << "\n";

    if (!is_process_running && ready_queue.size() != 0) {
      // pick shortest job if no job is being executed
      auto shortest_process = min_element(
          ready_queue.begin(), ready_queue.end(),
          [](Process &a, Process &b) { return a.cpu_time < b.cpu_time;
});

      current_process = *shortest_process;
      cout << "Starting Execution: ";
      print_process(current_process);
      // update solution table: response time
      solution_table[current_process.pid].push_back(
          time - current_process.arrival_time);
      is_process_running = true;
      ready_queue.erase(shortest_process);
    } else if (is_process_running) {
      // reduce time of current process
      --current_process.cpu_time;
      if (current_process.cpu_time == 1) {
        cout << "Finished Executing: ";
        print_process(current_process);
        is_process_running = false;

        // update solution table: completion time, tat, rt
        solution_table[current_process.pid].push_back(time + 1); // ct
        int tat = time - current_process.arrival_time + 1;
```

```
        solution_table[current_process.pid].push_back(tat); // tat = ct
- at
        int wt = tat - solution_table[current_process.pid][2];
        solution_table[current_process.pid].push_back(wt); // wt = tat -
bt
      } else {
        cout << "Now Executing: ";
        print_process(current_process);
      }
    }
    ++time;
    sleep(SLEEP_DELAY);
  }

  print_solution_table(solution_table);
}

void srtf(vector<Process> processes) {
  cout << "\n====== SRTF ======";
  cout << "\n===== Gantt Chart =====";
  int time = 1;
  vector<vector<int>> solution_table(processes.size() +
                                    1); // pid, at, bt, rt, ct, tat, wt

  vector<bool> visited(processes.size() + 1, false);
  // fill solution table
  for (auto p : processes) {
    solution_table[p.pid].push_back(p.pid);          // pid
    solution_table[p.pid].push_back(p.arrival_time); // at
    solution_table[p.pid].push_back(p.cpu_time);     // bt
  }

  list<Process> ready_queue;
  vector<Process>::iterator next_process = processes.begin();
  while (next_process != processes.end() || !ready_queue.empty()) {
    cout << "\nTime: " << time << "\n";

    // fill ready_queue
    while (next_process != processes.end() &&
           next_process->arrival_time <= time) {
      ready_queue.push_back(*next_process);
      next_process++;
    }
    cout << "Ready Queue Size " << ready_queue.size() << "\n";

    if (!ready_queue.empty()) {
```

```
      // pick shortest job if no job is being executed
      auto shortest_process = min_element(
          ready_queue.begin(), ready_queue.end(),
          [](Process &a, Process &b) { return a.cpu_time < b.cpu_time;
});
      Process current_process = *shortest_process;

      cout << "Now Executing: ";
      print_process(current_process);
      shortest_process->cpu_time -= 1;

      if (!visited[shortest_process->pid]) {
        // update solution table: response time
        solution_table[shortest_process->pid].push_back(
            time - shortest_process->arrival_time);
      }

      if (shortest_process->cpu_time == 0) {
        // process completed
        // update solution table: completion time, tat, rt
        cout << "Finished Executing: ";
        print_process(current_process);
        solution_table[current_process.pid].push_back(time + 1); // ct
        int tat = time - current_process.arrival_time + 1;
        solution_table[current_process.pid].push_back(tat); // tat = ct
- at
        int wt = tat - solution_table[current_process.pid][2];
        solution_table[current_process.pid].push_back(wt); // wt = tat -
bt
        ready_queue.erase(shortest_process);
      }
      visited[shortest_process->pid] = true;
    }
    ++time;
    sleep(SLEEP_DELAY);
  }

  print_solution_table(solution_table);
}

int main(int argc, char const *argv[]) {
  // create random processes
  vector<Process> processes;
  for (int i = 0; i < PROCESS_COUNT; ++i) {
    int at = random(PROCESS_ARRIVAL_MIN, PROCESS_ARRIVAL_MAX);
    int bt = random(PROCESS_BURST_MIN, PROCESS_BURST_MAX);
```

```
    Process temp = Process(i + 1, bt, at);
    processes.push_back(temp);
  }

  sort(processes.begin(), processes.end());

  for (auto p : processes) {
    print_process(p);
  }

  sjf(processes);
  srtf(processes);
  return 0;
}
```

```
→  lab-5 git:(master) ✗ ./a.out
P4: { AT = 1, BT = 5}
P5: { AT = 2, BT = 6}
P2: { AT = 3, BT = 2}
P3: { AT = 3, BT = 3}
P1: { AT = 4, BT = 3}
```

```
====== SJF ======
===== Gantt Chart =====
Time: 1
Ready Queue Size 1
Starting Execution: P4: { AT = 1, BT = 5}

Time: 2
Ready Queue Size 1
Now Executing: P4: { AT = 1, BT = 4}

Time: 3
Ready Queue Size 3
Now Executing: P4: { AT = 1, BT = 3}

Time: 4
Ready Queue Size 4
Now Executing: P4: { AT = 1, BT = 2}

Time: 5
Ready Queue Size 4
Finished Executing: P4: { AT = 1, BT = 1}

Time: 6
Ready Queue Size 4
Starting Execution: P2: { AT = 3, BT = 2}

Time: 7
Ready Queue Size 3
Finished Executing: P2: { AT = 3, BT = 1}

Time: 8
Ready Queue Size 3
Starting Execution: P3: { AT = 3, BT = 3}

Time: 9
Ready Queue Size 2
Now Executing: P3: { AT = 3, BT = 2}
```

```
Time: 10
Ready Queue Size 2
Finished Executing: P3: { AT = 3, BT = 1}

Time: 11
Ready Queue Size 2
Starting Execution: P1: { AT = 4, BT = 3}

Time: 12
Ready Queue Size 1
Now Executing: P1: { AT = 4, BT = 2}

Time: 13
Ready Queue Size 1
Finished Executing: P1: { AT = 4, BT = 1}

Time: 14
Ready Queue Size 1
Starting Execution: P5: { AT = 2, BT = 6}

Time: 15
Ready Queue Size 0
Now Executing: P5: { AT = 2, BT = 5}

Time: 16
Ready Queue Size 0
Now Executing: P5: { AT = 2, BT = 4}

Time: 17
Ready Queue Size 0
Now Executing: P5: { AT = 2, BT = 3}

Time: 18
Ready Queue Size 0
Now Executing: P5: { AT = 2, BT = 2}

Time: 19
Ready Queue Size 0
Finished Executing: P5: { AT = 2, BT = 1}
```

```
===== Solution Table =====
PID  AT   BT   RT   CT   TAT  WT
1    4    3    7    14   10   7
2    3    2    3    8    5    3
3    3    3    5    11   8    5
4    1    5    0    6    5    0
5    2    6    12   20   18   12
avg  2.6  3.8  5.4  11.8 9.2  5.4
```

```
====== SRTF ======
===== Gantt Chart =====
Time: 1
Ready Queue Size 1
Now Executing: P4: { AT = 1, BT = 5}

Time: 2
Ready Queue Size 2
Now Executing: P4: { AT = 1, BT = 4}

Time: 3
Ready Queue Size 4
Now Executing: P2: { AT = 3, BT = 2}

Time: 4
Ready Queue Size 5
Now Executing: P2: { AT = 3, BT = 1}
Finished Executing: P2: { AT = 3, BT = 1}

Time: 5
Ready Queue Size 4
Now Executing: P4: { AT = 1, BT = 3}

Time: 6
Ready Queue Size 4
Now Executing: P4: { AT = 1, BT = 2}

Time: 7
Ready Queue Size 4
Now Executing: P4: { AT = 1, BT = 1}
Finished Executing: P4: { AT = 1, BT = 1}

Time: 8
Ready Queue Size 3
Now Executing: P3: { AT = 3, BT = 3}

Time: 9
Ready Queue Size 3
Now Executing: P3: { AT = 3, BT = 2}
```

```
Time: 10
Ready Queue Size 3
Now Executing: P3: { AT = 3, BT = 1}
Finished Executing: P3: { AT = 3, BT = 1}

Time: 11
Ready Queue Size 2
Now Executing: P1: { AT = 4, BT = 3}

Time: 12
Ready Queue Size 2
Now Executing: P1: { AT = 4, BT = 2}

Time: 13
Ready Queue Size 2
Now Executing: P1: { AT = 4, BT = 1}
Finished Executing: P1: { AT = 4, BT = 1}

Time: 14
Ready Queue Size 1
Now Executing: P5: { AT = 2, BT = 6}

Time: 15
Ready Queue Size 1
Now Executing: P5: { AT = 2, BT = 5}

Time: 16
Ready Queue Size 1
Now Executing: P5: { AT = 2, BT = 4}
```

```
Time: 17
Ready Queue Size 1
Now Executing: P5: { AT = 2, BT = 3}

Time: 18
Ready Queue Size 1
Now Executing: P5: { AT = 2, BT = 2}

Time: 19
Ready Queue Size 1
Now Executing: P5: { AT = 2, BT = 1}
Finished Executing: P5: { AT = 2, BT = 1}

===== Solution Table =====
PID  AT   BT   RT   CT   TAT  WT
1    4    3    7    14   10   7
2    3    2    0    5    2    0
3    3    3    5    11   8    5
4    1    5    0    8    7    2
5    2    6    12   20   18   12
avg  2.6  3.8  4.8  11.6 9    5.2
```