อัลกอริทึม

1. First-Come, First-Served (FCSC)
   1. รับ process list มาจาก user
   2. เรียง process จาก arrival time น้อยไปมาก และหากเท่ากันให้เรียงจาก burst time จากน้อยไปมาก
   3. ทำงานตาม process ทีละตัวจนหมด
   4. สรุปผล
2. Shortest-Job-First (SJF) และ generalized ได้เป็น Shortest Remaining Time First
   1. รับ process list มาจาก user
   2. เรียง process จาก arrival time น้อยไปมาก และหากเท่ากันให้เรียงจาก burst time จากน้อยไปมาก
   3. หาผลต่าง arrival time ระหว่าง process ที่ 1 และ 2 และเนื่องจาก list ถูกเรียงจากน้อยไปมาก นี่คือผลต่างที่น้อยที่สุด
   4. ถ้าผลต่างเป็น 0 ให้ใช้ burst time จาก process ที่ 1 หากมากกว่า 0 ให้ใช้ผลต่างนี้เป็น burst time
   5. ทำงาน process ที่ 1 โดยใช้เวลา burst time
   6. หาก burst time ที่ใช้คือ burst time ของ process 1 ให้ลบมันออกจาก list เพราะทำงานครบแล้ว แต่หากเป็น burst time ที่ได้จากผลต่างข้างต้น ให้ลด burst time ของ process 1 ลงเท่าที่ใช้ทำงานไป
   7. ลด arrival time ของ process ทุกตัวตาม burst time ที่ใช้ไป
   8. หาก list ยังไม่ว่าง กลับไปทำข้อ b
   9. สรุปผล
3. Round Robin (RR)
   1. รับ process list มาจาก user
   2. ไล่ทำงานตาม time quantum ทุกๆ process
   3. หาก list ยังไม่ว่าง กลับไปทำข้อ b
   4. สรุปผล

โปรแกรม

|  |
| --- |
| #include <algorithm>  #include <iostream>  #include <vector>  struct Process {  int id;  int arrivalTime;  int waitedSince;  int burstTime;  int priority;  };  struct Report {  double avgWaitingTime;  double avgTurnAroundTime;  };  auto firstArriveAtZero(const std::vector<Process> &P) {  std::vector<Process> \_P(P);  int minArrivalTime =  std::min\_element(\_P.begin(), \_P.end(), [](const Process &a,  const Process &b) {  return a.arrivalTime < b.arrivalTime;  })->arrivalTime;  std::for\_each(\_P.begin(), \_P.end(), [minArrivalTime](Process &x) {  x.arrivalTime -= minArrivalTime;  });  return \_P;  }  auto sortByArrivalTime(const std::vector<Process> &P) {  std::vector<Process> \_P(P);  std::sort(\_P.begin(), \_P.end(), [](const Process &a, const Process &b) {  return a.arrivalTime < b.arrivalTime;  });  return \_P;  }  auto sortByArrivalTimeAndBurstTime(const std::vector<Process> &P) {  std::vector<Process> \_P(P);  std::sort(\_P.begin(), \_P.end(), [](const Process &a, const Process &b) {  return (a.arrivalTime < b.arrivalTime) ||  ((a.arrivalTime == b.arrivalTime) && (a.burstTime < b.burstTime));  });  return \_P;  }  void printProcess(const Process &p) {  std::cout << p.id << " { " << p.arrivalTime << ", " << p.burstTime << ", "  << p.priority << "}" << std::endl;  }  void printProcessList(const std::vector<Process> &P) {  for (auto &p : P) {  printProcess(p);  }  }  Report runFCFS(const std::vector<Process> &P) {  if (P.empty())  return {0, 0};  auto \_P = sortByArrivalTimeAndBurstTime(P);  int n = \_P.size();  int time = 0;  int waitingTime = 0;  int turnAroundTime = 0;  for (auto &p : \_P) {  waitingTime += time - p.waitedSince;  time += p.burstTime;  turnAroundTime += time - p.waitedSince;  }  double avgWaitingTime = (double)waitingTime / (double)n;  double avgTurnAroundTime = (double)turnAroundTime / (double)n;  return {avgWaitingTime, avgTurnAroundTime};  }  Report runSJF(const std::vector<Process> &P) {  if (P.empty())  return {0, 0};  auto \_P = sortByArrivalTimeAndBurstTime(P);  int n = \_P.size();  int time = 0;  int waitingTime = 0;  int turnAroundTime = 0;  while (!\_P.empty()) {  // std::cout << "time " << time << std::endl;  // printProcessList(\_P);  auto &p = \_P.at(0);  waitingTime += time - p.waitedSince;  // std::cout << "waitedSince " << p.waitedSince << std::endl;  if (\_P.size() == 1) {  time += p.burstTime;  turnAroundTime += time - p.waitedSince;  \_P.erase(\_P.begin());  } else {  auto &q = \_P.at(1);  int smallestArrivalDiff = q.arrivalTime - p.arrivalTime;  int burstTime = p.burstTime;  if (q.arrivalTime != p.arrivalTime && smallestArrivalDiff < burstTime) {  burstTime = smallestArrivalDiff;  p.burstTime -= burstTime;  p.waitedSince += burstTime;  } else {  turnAroundTime += time - p.waitedSince;  \_P.erase(\_P.begin());  }  std::for\_each(\_P.begin(), \_P.end(), [burstTime](Process &x) {  x.arrivalTime = std::max(0, x.arrivalTime - burstTime);  });  // std::cout << "burst " << burstTime << std::endl;  // std::cout << "-------------------------" << std::endl;  time += burstTime;  \_P = sortByArrivalTimeAndBurstTime(\_P);  }  }  double avgWaitingTime = (double)waitingTime / (double)n;  double avgTurnAroundTime = (double)turnAroundTime / (double)n;  return {avgWaitingTime, avgTurnAroundTime};  }  Report runRR(const std::vector<Process> &P, int quantum) {  if (P.empty())  return {0, 0};  auto \_P = std::vector<Process>(P);  int n = \_P.size();  int time = 0;  int waitingTime = 0;  int turnAroundTime = 0;  int prevId = -1;  while (!\_P.empty()) {  for (auto &p : \_P) {  // std::cout << "time " << time << std::endl;  // printProcessList(\_P);  if (prevId != p.id) {  waitingTime += time - p.waitedSince;  }  int burstTime = std::min(quantum, p.burstTime);  // std::cout << "burstTime " << burstTime << std::endl;  time += burstTime;  int bt = std::min(p.burstTime, burstTime);  p.burstTime = std::max(0, p.burstTime - burstTime);  // std::cout << "waitedSince = " << p.waitedSince << std::endl;  // if (prevId != p.id) {  if (p.burstTime <= 0) {  // std::cout << "turnAroundTime += " << (bt) << std::endl;  turnAroundTime += bt;  } else {  // std::cout << "turnAroundTime += " << (time - p.waitedSince)  // << std::endl;  turnAroundTime += time - p.waitedSince;  }  if (p.burstTime > 0) {  // p.waitedSince += burstTime;  p.waitedSince = time;  }  // std::cout << "-------------------------" << std::endl;  prevId = p.id;  }  \_P.erase(std::remove\_if(\_P.begin(), \_P.end(),  [](Process &x) { return x.burstTime <= 0; }),  \_P.end());  // std::cout << "+++++++" << std::endl;  }  double avgWaitingTime = (double)waitingTime / (double)n;  double avgTurnAroundTime = (double)turnAroundTime / (double)n;  return {avgWaitingTime, avgTurnAroundTime};  }  void printReport(std::string head, const Report &rep) {  std::cout << head << std::endl;  std::cout << "avgWaitingTime = " << rep.avgWaitingTime << std::endl;  std::cout << "avgTurnAroundTime = " << rep.avgTurnAroundTime << std::endl;  }  int main() {  std::cout << "Enter processes" << std::endl;  std::cout << "How many processes? ";  int n;  std::cin >> n;  std::vector<Process> P;  for (int i = 0; i < n; i++) {  int j = i + 1;  std::cout << "P" << j << ": " << std::endl;  std::cout << "Arrival Time? ";  int at;  std::cin >> at;  std::cout << "Burst Time? ";  int bt;  std::cin >> bt;  P.push\_back({j, at, at, bt, 0});  }  int qt;  std::cout << "Time Quantum? ";  std::cin >> qt;  Report repFCFS = runFCFS(P);  printReport("runFCFS", repFCFS);  std::cout << "=======================" << std::endl;  Report repSJF = runSJF(P);  printReport("runSJF", repSJF);  std::cout << "=======================" << std::endl;  Report repRR = runRR(P, qt);  printReport("runRR", repRR);  // std::cout << "=======================" << std::endl;  return 0;  } |

การทดลองที่ 1

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| # | 1 | | 2 | | 3 | | 4 | |
|  | AT | BT | AT | BT | AT | BT | AT | BT |
| P1 | 0 | 50 | 0 | 50 | 0 | 50 | 0 | 50 |
| P2 | 0 | 50 | 10 | 50 | 20 | 50 | 40 | 50 |
| P3 | 0 | 50 | 20 | 50 | 40 | 50 | 80 | 50 |
| P4 | 0 | 50 | 30 | 50 | 60 | 50 | 120 | 50 |
| P5 | 0 | 50 | 40 | 50 | 80 | 50 | 160 | 50 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | FCFS | | SJF | | RR | |
| # | AWT | ATAT | AWT | ATAT | AWT | ATAT |
| 1 | 100 | 150 | 100 | 110 | 196 | 238 |
| 2 | 80 | 130 | 80 | 90 | 176 | 218 |
| 3 | 60 | 110 | 60 | 70 | 156 | 198 |
| 4 | 20 | 70 | 32 | 30 | 116 | 158 |

AWT = Average Waiting Time, ATRT = Average Turn Around Time

การทดลองที่ 2

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| # | 1 | | 2 | | 3 | | 4 | |
|  | AT | BT | AT | BT | AT | BT | AT | BT |
| P1 | 0 | 50 | 0 | 50 | 0 | 50 | 0 | 50 |
| P2 | 50 | 50 | 50 | 100 | 50 | 100 | 50 | 200 |
| P3 | 100 | 50 | 100 | 150 | 100 | 200 | 100 | 800 |
| P4 | 150 | 50 | 150 | 200 | 150 | 400 | 150 | 3200 |
| P5 | 200 | 50 | 200 | 250 | 200 | 800 | 200 | 12800 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | FCFS | | SJF | | RR | |
| # | AWT | ATAT | AWT | ATAT | AWT | ATAT |
| 1 | 0 | 50 | 0 | 10 | 96 | 138 |
| 2 | 100 | 250 | 100 | 150 | 296 | 442 |
| 3 | 160 | 470 | 160 | 320 | 416 | 722 |
| 4 | 1020 | 4430 | 1020 | 3580 | 2136 | 5542 |

AWT = Average Waiting Time, ATRT = Average Turn Around Time

สรุป

จากการทดลอง SJF ทำงานได้มีประสิทธิภาพมากที่สุด โดยทำงานช้ากว่า FCFS เพียงแค่กรณีที่ Arrival Time มีค่าเป็น multiple ของ Burst Time ซึ่งเกิดขึ้นไม่บ่อย ส่วน RR ได้ทดลองใช้ Time Quantum = 2 ซึ่งทำงานได้แย่ที่สุดระหว่างทั้งสาม