HATICE YAREN BULUT-20191710013

Project Report

1. Introduction

This project aims to simulate an automotive parts production line, detailing various production stages (loading, machining, assembly, inspection, packaging). It provides a platform to analyze and optimize the performance of the production line, considering machine breakdowns, maintenance times, workforce management, and product variety.

2. Technologies and Tools Used

The project is implemented in C++ programming language, utilizing standard libraries (iostream, vector, string, random, chrono, thread, iomanip) to support data structures, timing operations, random number generation, and output formatting.

3. Constants and Parameters

Several constants and parameters are defined to manage different aspects of the production process:

- PROCESSING_TIMES[]: Represents processing times for each production stage in minutes.
- MAINTENANCE_TIME: Specifies the machine maintenance time.
- BREAKDOWN_PROB: Indicates the probability of machine breakdown during a stage.
- WORKER_COUNTS[]: Defines the required number of workers for each stage.
- SHIFT_LENGTH: Specifies the shift length in minutes.
- SIMULATION_DURATION: Sets the total simulation duration in minutes.
- NUM_PRODUCT_TYPES: Represents the number of supported product types.

These parameters provide the foundational elements for simulating and evaluating the production line's performance.

4. ProductionLine Class

The ProductionLine class defines the core structure of the production line. It simulates the production process for each product part, managing workforce, machine breakdowns, and logging production data.

- process_part(int part_id, int product_type): Simulates the production stages for each part. It handles processing times, workforce management, machine breakdowns, and logs production data.

- print_log_data(): Outputs the logged production data in a formatted table.
- LogEntry: A structure used to store detailed production data, including part ID, stage, start and finish times, duration, and product type.

5. Main Function and Simulation Initialization

The setup(int num_parts_per_type) function initializes the simulation for a specified number of parts and product types. Each part's production process is initiated through the create_part function.

The main() function serves as the entry point, calling the setup function to start the simulation and printing the results to the console.

6. Project Scalability and Potential Enhancements

While initially supporting a single product type, the project is designed to easily expand to handle multiple product types. This scalability is facilitated by the NUM_PRODUCT_TYPES constant and the product type parameter within the process_part function.

7. Conclusions and Recommendations

The project provides a valuable tool for analyzing and improving operational efficiency within the production line simulation. Data collection, analysis, and visual presentation of simulation results can identify optimization opportunities and enhance operational strategies.

This report comprehensively describes the project activities and findings. Future enhancements could include deeper analysis of managing multiple product types and exploring additional operational scenarios.

DETAILED DESCRIPTION OF THE CODE

1. Libraries and Initialization

At the beginning of the code, standard C++ libraries are included. These libraries are essential for various data structures, timing functions, and output operations.

```
#include <iostream>
#include <vector>
#include <string>
#include <random>
#include <chrono>
#include <thread>
```

```
#include <iomanip>
```

iostream: Used for basic input/output operations.
vector: Represents dynamically sized arrays and is used as a data structure in this project.
string: Essential for text manipulations, specifically for storing stage names.
random: Used to generate random numbers, crucial for simulating breakdown probabilities.
chrono: Provides timing and duration calculations.
thread: Supports multi-threading, particularly useful for timed waiting operations.
iomanip: Used for formatting output, such as for printing tables.

2. Constants and Parameters

```
const int PROCESSING_TIMES[] = {7, 12, 9, 7, 5};
const int MAINTENANCE_TIME = 4;
const double BREAKDOWN_PROB = 0.15;
const int WORKER_COUNTS[] = {3, 4, 5, 3, 4};
const int SHIFT_LENGTH = 8 * 60;
const int SIMULATION_DURATION = 100;
const int NUM_PRODUCT_TYPES = 2;
```

PROCESSING_TIMES: Represents the processing times (Loading, Machining, Assembling, Inspecting, Packaging) in minutes for each stage.
MAINTENANCE_TIME: Specifies the maintenance time in minutes.
BREAKDOWN_PROB: Indicates the probability of a machine breakdown occurring during any stage.
WORKER_COUNTS: Specifies the number of workers required for each stage.
SHIFT_LENGTH: Represents the total length of a shift in minutes.
SIMULATION_DURATION: Specifies the total duration of the simulation in minutes.
NUM_PRODUCT_TYPES: Represents the number of product types managed in the project.

3. ProductionLine Class

```
class ProductionLine {
public:
   ProductionLine() : current_time(0) {
      for (int i = 0; i < 5; ++i) {
         workers[i] = WORKER_COUNTS[i];
      }
   }

   void process_part(int part_id, int product_type) {
      vector<string> stages = {"Loading", "Machining", "Assembling", "Inspecting", "Packaging"};

      for (int i = 0; i < 5; ++i) {
         double duration = PROCESSING_TIMES[i] * (product_type == 1 ? 1.0 : 1.2);
         double start_time = current_time;

         while (workers[i] == 0) {
             std::this_thread::sleep_for(chrono::seconds(1));
             current_time++;
         }
         workers[i]--;

         std::this_thread::sleep_for(chrono::milliseconds(static_cast<int>(duration * 1000)));

         if (random_breakdown()) {
             repair_machine(i);
         }

         current_time += duration;
         workers[i]++;

         LogEntry entry;
         entry.part_id = part_id;
         entry.stage = stages[i];
         entry.start_time = start_time;
         entry.finish_time = current_time;
         entry.duration = duration;
         entry.product_type = product_type;
         log_data.push_back(entry);
      }
   }

   void print_log_data() {
      cout << left << setw(10) << "Part"
           << left << setw(12) << "Stage"
           << left << setw(15) << "Start Time"
           << left << setw(15) << "Finish Time"
           << left << setw(10) << "Duration"
           << left << setw(15) << "Product Type"
           << endl;

      for (const LogEntry& entry : log_data) {
         cout << left << setw(10) << entry.part_id
              << left << setw(12) << entry.stage
```

```
                << left << setw(15) << entry.start_time
                << left << setw(15) << entry.finish_time
                << left << setw(10) << entry.duration
                << left << setw(15) << entry.product_type
                << endl;
        }
    }

private:
    struct LogEntry {
        int part_id;
        string stage;
        double start_time;
        double finish_time;
        double duration;
        int product_type;
    };

    vector<LogEntry> log_data;
    int workers[5];
    double current_time;

    bool random_breakdown() {
        static random_device rd;
        static mt19937 gen(rd());
        static uniform_real_distribution<> dis(0.0, 1.0);
        return dis(gen) < BREAKDOWN_PROB;
    }

    void repair_machine(int stage) {
        while (repair_team == 0) {
            std::this_thread::sleep_for(chrono::seconds(1));
            current_time++;
        }
        repair_team--;
        std::this_thread::sleep_for(chrono::milliseconds(MAINTENANCE_TIME * 1000));
        current_time += MAINTENANCE_TIME;
        repair_team++;
    }

    int repair_team = 2;
};
```

ProductionLine Class: This class defines the basic structure and operation of the production line. It simulates the production process for each product part.

ProductionLine(): Constructor initializes workers array based on WORKER_COUNTS and sets current_time to zero.

process_part(int part_id, int product_type): Simulates the production process of each part. Manages processing times, worker availability, machine breakdowns, and logs production data using LogEntry.

print_log_data(): Prints the recorded production data in a formatted table.

LogEntry: Structure to store production data including part ID, stage, start and finish times, duration, and product type.

random_breakdown(): Determines if a random machine breakdown occurs based on BREAKDOWN_PROB.

repair_machine(int stage): Simulates repairing a breakdown machine at a specific stage.

4. Main Function and Simulation Initialization

```
void setup(int num_parts_per_type) {
   ProductionLine line;

   for (int product_type = 1; product_type <= NUM_PRODUCT_TYPES; ++product_type) {
      for (int i = 0; i < num_parts_per_type; ++i) {
         create_part(line, i + 1, product_type);
      }
   }

   line.print_log_data();
}

int main() {
   setup(5); // Starts a simulation with 5 parts
   return 0;
}
```
setup(int num_parts_per_type): Initializes the simulation for a specific number of parts per product type. Creates a line object and uses create_part to send each part through the production line.

main(): Entry point of the program. Calls setup to start the simulation