U.B.A. FACULTAD DE INGENIERÍA

Departamento de Computación

Técnicas de programación concurrentes I 75-59

TRABAJO PRÁCTICO 1

Primavera concurrente

Curso: 2019 - 2do Cuatrimestre

Turno: Miércoles

GRUPO N° 5	
Integrantes	**Padrón**
Gamarra Silva, Cynthia Marlene	92702
Cuneo, Paulo	84840
Fecha de entrega:	25-09-2019
Fecha de aprobación:	
Calificación:	
Firma de aprobación:	

Observaciones:

Índice

1. Enunciado del trabajo práctico

Primer Proyecto: Primavera Concurrente

75.59 - Técnicas de Programación Concurrente I

Objetivo

Se debe implementar un sistema de software que simule el movimiento del mercado de floricultura.

Requerimientos Funcionales

La proximidad de la llegada de la primavera pone nuevamente en acción a los productores de flores.

La cadena productiva del mercado de flores está conformada por: productores del campo (son los que recolectan las flores), centros de distribución y puntos de venta (como el Mercado de flores de Barracas).

Los productores trabajan en el campo y cosechan *rosas* o *tulipanes*, las colocan en cajones de 10 ramos y las transportan al centro de distribución. Cada ramo está identificado con el nombre del productor en una etiqueta que cuelga del tallo. Los productores llevan un cajón al centro de distribución y continúan luego con el trabajo de campo, repitiendo cada uno el proceso sucesivamente.

En los Centros de Distribución se clasifican, separándolas entre rosas y tulipanes y se empaquetan de a 100 ramos para ahorrar costos de Logística, y se transportan a los puntos de venta.

En los puntos de venta, los comerciantes:

1. reciben pedidos por Internet con cantidades de rosas y de tulipanes. Los comerciantes arman los pedidos y emiten el remito que contiene la lista de las flores que componen el pedido (con su correspondiente nombre del productor), y lo entregan al sistema de repartos con bicicletas.

2. atienden al público en general: arman el ramo en el momento, lo decoran y se lo entregan al cliente en el momento. Los clientes ingresan de forma aleatoria al punto de venta y son atendidos de a uno.

Se deberá implementar lo siguiente:

1. Se deberá poder consultar en todo momento: quién es el productor que más flores vendió y cuál es el tipo de flores más comprado.

2. Se deberá poder configurar los pedidos que se reciben por Internet.

Como condiciones adicionales a las planteadas por el ejercicio, se deberán cumplir las siguientes:

1. La simulación debe poder pausarse para ser reanudada su ejecución más adelante. De forma que el stock de los centros de distribución y de venta sea persistido entre ejecuciones.

2. Se deberá poder configurar la cantidad de Centros de Distribución y de puntos de venta.

Requerimientos no Funcionales

Los siguientes son los requerimientos no funcionales de la aplicación:

1. El proyecto deberá ser desarrollado en lenguaje C o C++, siendo este último el lenguaje de preferencia.

2. La simulación puede no tener interfaz gráfica y ejecutarse en una o varias consolas de línea de comandos.
3. El proyecto deberá funcionar en ambiente Unix / Linux.
4. Todas las configuraciones deberán poder realizarse sin recompilar la aplicación.
5. El programa deberá poder ejecutarse en "modo debug", lo cual dejará registro de la actividad que realiza en un único archivo de texto para su revisión posterior. Se deberá poder seguir el recorrido de cada una las flores.
6. La aplicación deberá funcionar en una única computadora.
7. Las facilidades de IPC que se podrán utilizar para la comunicación entre procesos son:
   a) Pipes, FIFOs
   b) Memoria compartida

Tareas a Realizar

A continuación se listan las tareas a realizar para completar el desarrollo del proyecto:

1. Dividir el proyecto en procesos.
2. Una vez obtenida la división en procesos, establecer un esquema de comunicación entre ellos teniendo en cuenta los requerimientos de la aplicación. ¿Qué procesos se comunican entre sí? ¿Qué datos necesitan compartir para poder trabajar?
3. Diseñar y documentar el protocolo de comunicación.
4. Realizar la codificación de la aplicación. El código fuente debe estar documentado.

Entrega

La entrega del proyecto comprende lo siguiente:

1. Informe, se deberá presentar impreso en una carpeta o folio y en forma digital (PDF) a través del campus
2. El código fuente de la aplicación, que se entregará únicamente mediante el campus

La entrega en el campus estará habilitada hasta las 19 hs de la fecha indicada oportunamente.

El informe a entregar junto con el proyecto debe contener los siguientes ítems:

1. Detalle de resolución de la lista de tareas anterior.
2. Diagramas de clases.
3. Listado y detalle de las *user-stories*.

2. Objetivos

Aprender sobre el funcionamiento de pipes, procesos y fork.

2.1. User Stories

* Producir Cajar de Ramos
Como productor, quiero generar cajas de 10 ramos, con una cantidad de rosas aleatorias. Los ramos deben contenter el id del producto y un id propio para poder hacer el tracking.

* Empaquetar Ramos en el distribuidor
Como distribuidor, quiero recibir cajas de 10 ramos y almacenarlas en un stock interno, cada vez que el stock alcance los 100 ramos de un mismo tipo de flor, quiero armar un paquete con ese tipo de flor para enviarse a algún punto de venta.

* Atender pedidos

- Como puntos de venta,quiero atender los pedidos, y en base al stock disponible generar un remito con la resolución del pedido para enviar al inventario.

* Registrar Ventas en Inventario
El inventario debera recibir los remitos, y generar un reporte bajo demanda que indique el productor que mas vende y el tipo de flor que mas vende.

3. Diseño e implementación

La siguiente figura ilustra un diagrama de dominio del problema.

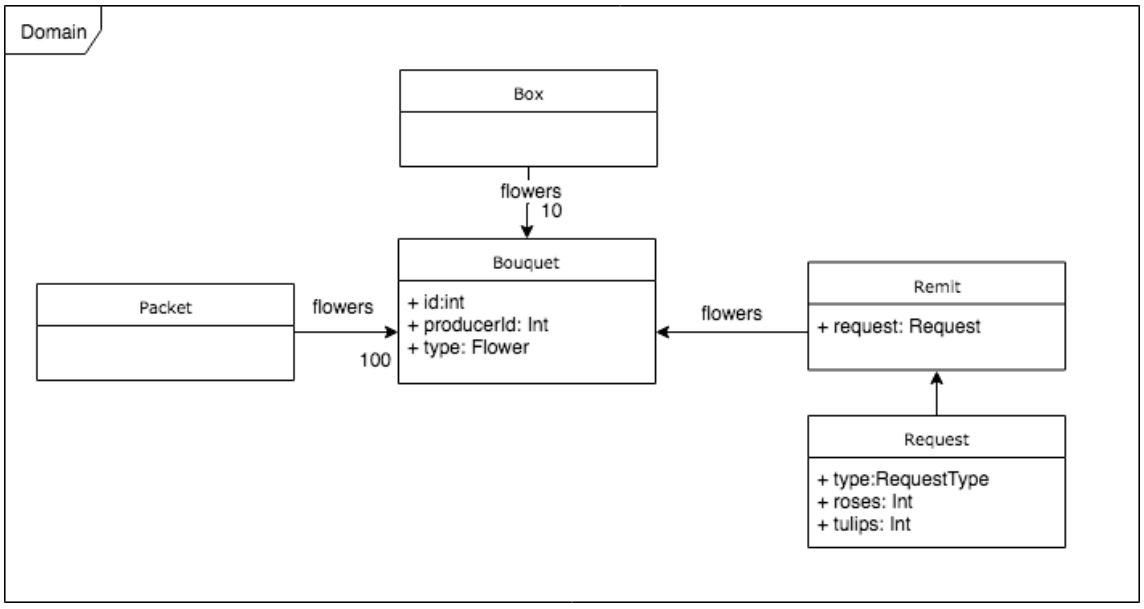

Figura 1: Diagrama de clases

La siguiente figura ilustra el diagrama de clases del programa.

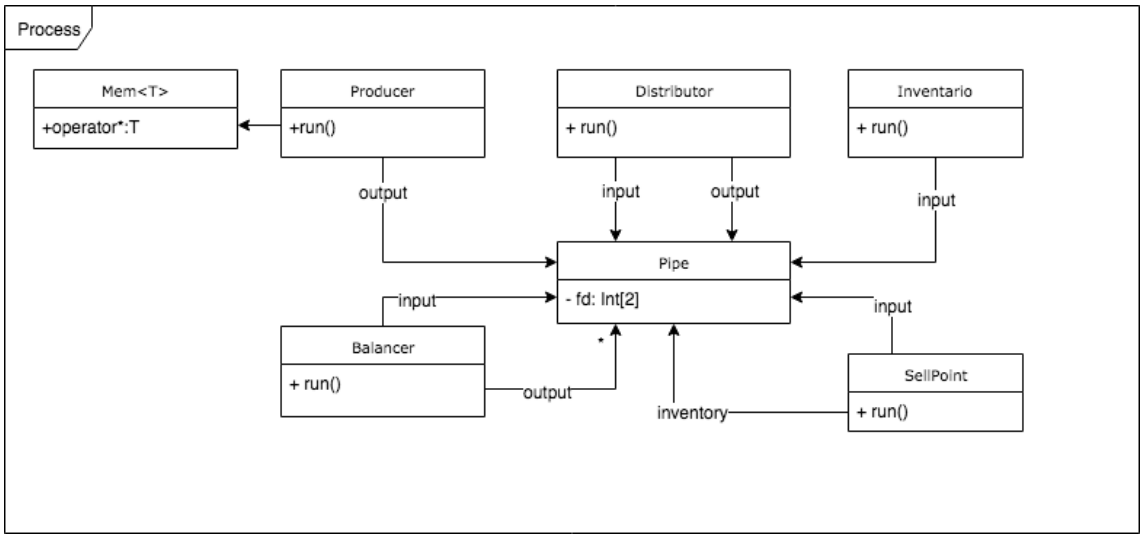

Figura 2: Clases de Procesos

La siguiente figura ilustra los procesos del programa.

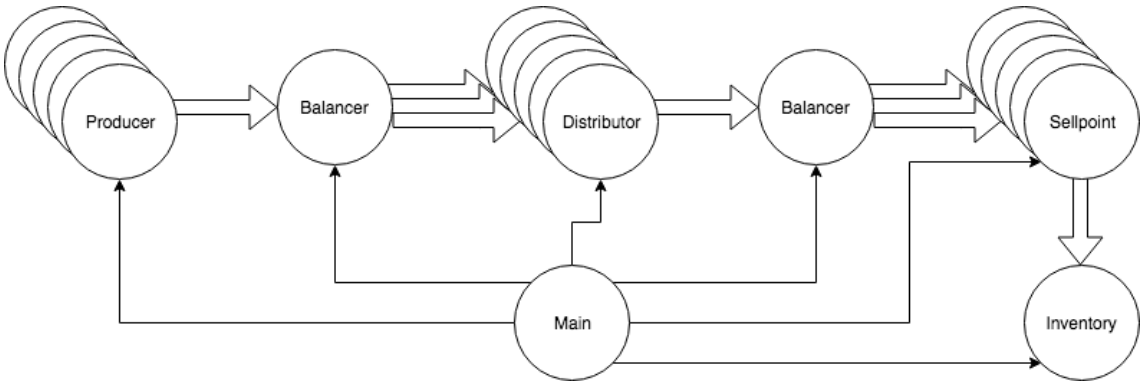

Figura 3: Procesos del programa

4. Parámetros del programa

El programa toma por línea de consola un único parámetro, que refiere a un archivo a utilizar como configuración.

El archivo debe tener el siguiente formato.

./main pathAlArchivoConfiguracion

Formato archivo configuración. productores
distribuidores
puntosDeVenta
almacenamientoDistribuidor1
almacenamientoDistribuidorN
almacenamientoPuntoVenta1
pedidosPuntoVenta1
almacenamientoPuntoVentaN
pedidosPuntoVentaN
almacenammientoInventario

5. Conclusiones

Combinar fork() y pipes, para desarrollar un programa concurrente requiere mantener un orden en la apertura y cerrado de los pipes. De forma de no mantener pipes abiertos en procesos que no consumen esos pipes.

Simplemente encapsular los pipes y forks en objetos no resuelve la problematica de establecer pre y post protocolos al inicializar o finalizar cada subproceso.

A. Código fuente

B. Balancer

```
#ifndef _BALANCER_HPP_
#define _BALANCER_HPP_

#include <iostream>
#include <vector>
#include "log.hpp"
#include "pipe.hpp"
#include <string>

class Balancer {
private:
  Pipe & inputPipe;
  std::vector<Pipe> & outputPipes;
  Logger logger;
public:
  Balancer(Pipe & in,
           std::vector<Pipe> & out,
           const std::string & name):
    inputPipe(in),
    outputPipes(out),
    logger(name) {
  }

  void close() {
    inputPipe.close();
    for(auto &o:outputPipes) {
      o.close();
    }
  }

  void run() {
    inputPipe.in().asStdIn();
    std::vector<Out> outs;
    for(auto&o: outputPipes) {
      outs.push_back(o.out());
    }
    while(true) {
      for(auto &o : outs) {
        if(std::cin.peek() == -1) {
          logger.debug("finalizing");
          close();
          return;
        }
        std::string lineIn;
        logger.debug("reading.");
        std::getline(std::cin, lineIn);
        logger.debug("read:" + lineIn);
        logger.debug("writing.");
        std::string lineOut = lineIn + "\n";
        o.write(lineOut.c_str(), lineOut.size());
        logger.debug("write:" + lineOut);
      }
    }
  }
};

#endif
```

C. Flower

```
#ifndef _FLOWER_HPP_
#define _FLOWER_HPP_

#include <iostream>
#include <vector>

template <char c>
std::istream& isChar(std::istream& in) {
  int chr = in.get();
  if(chr  == -1) {
    throw std::string("EOF no expected");
  }
  if(chr != c) {
    throw std::string("Expected ") + c;
  }
  return in;
};

template<class T>
T deserialize(std::istream & in);

enum Flower { ROSE=0, TULIP=1 };

// BOUQUETS
struct Bouquet {
  int id;
  int producer;
  Flower type;
  Bouquet(){}

  Bouquet(int id, int p, Flower f) : id(id), producer(p), type(f) { }

  Bouquet(const Bouquet&other) {
    this->id = other.id;
    this->producer = other.producer;
    this->type = other.type;
  }

  Bouquet& operator=(const Bouquet&other) {
    this->id = other.id;
    this->producer = other.producer;
    this->type = other.type;
    return *this;
  }

};

template<>
Bouquet deserialize<Bouquet>(std::istream & in) {
  Bouquet result;
  int a;
  in >> result.id >> isChar<','> >> result.producer >> isChar<','> >> a >> isChar<';'>;
  result.type = static_cast<Flower>(a);
  return result;
};

std::ostream&  operator<<(std::ostream & out, const Bouquet& b) {
  out << b.id << "," << b.producer << "," << b.type << ";";
  return out;
};

// BOXES
struct Box{
  Bouquet flowers[10];
};

```

```
67 template<>
68 Box deserialize<Box>(std::istream & in) {
69   Box result;
70   for(int i = 0; i< 10; ++i) {
71     result.flowers[i] = deserialize<Bouquet>(in);
72   }
73   int chr = in.get();
74   if(chr != '\n') {
75     throw std::string("Broken Box");
76   }
77   return result;
78 };
79
80 std::ostream& operator<<(std::ostream & out, const Box & box) {
81   for(int i=0; i< 10; ++i) {
82     out << box.flowers[i];
83   }
84   out << std::endl;
85   return out;
86 };
87
88 // PACKETS
89 #define FLOWER_PACKET_SIZE 20
90 struct Packet{
91   std::vector<Bouquet> flowers;
92   Packet(const std::vector<Bouquet> & flowers) : flowers(flowers) {}
93 };
94
95 template<>
96 Packet deserialize<Packet>(std::istream & in) {
97   std::vector<Bouquet> flowers;
98   for(int i = 0; i< FLOWER_PACKET_SIZE; ++i) {
99     flowers.push_back(deserialize<Bouquet>(in));
100  }
101  int chr = in.get();
102  if(chr != '\n') {
103    throw std::string("Broken Packet");
104  }
105  return Packet(flowers);
106 };
107 std::ostream& operator<<(std::ostream & out, const Packet & packet) {
108   for(int i=0; i< FLOWER_PACKET_SIZE; ++i) {
109     out << packet.flowers[i];
110   }
111   out << std::endl;
112   return out;
113 };
114
115 // REQUEST
116 enum RequestType { INTERNET = 0, FRONTDESK = 1};
117
118 struct Request {
119   RequestType type;
120   int roses;
121   int tulips;
122 };
123
124 template<>
125 Request deserialize<Request>(std::istream & in) {
126   Request result;
127   int a;
128   in >> a >> isChar<','> >> result.roses >> isChar<','> >> result.tulips >> isChar<';'>;
129   result.type = static_cast<RequestType>(a);
130   return result;
131 };
132
133 std::ostream& operator<<(std::ostream & out, const Request & request) {
```

```
134   out << request.type << "," << request.roses << "," << request.tulips << ";";
135   return out;
136 };
137
138 struct Remit {
139   std::vector<Bouquet> flowers;
140   Request request;
141   Remit(const std::vector<Bouquet> & flowers,
142         const Request & request):
143     flowers(flowers),
144     request(request) {
145   }
146 };
147
148
149 template<>
150 Remit deserialize<Remit>(std::istream & in) {
151   Request request = deserialize<Request>(in);
152   int size;
153   in >> size >> isChar<':'>;
154   std::vector<Bouquet> flowers;
155   for(int i = 0; i< size; ++i) {
156     flowers.push_back(deserialize<Bouquet>(in));
157   }
158   return Remit(flowers, request);
159 };
160
161 std::ostream& operator<<(std::ostream & out, const Remit & remit) {
162   out << remit.request
163       << remit.flowers.size()
164       << ":";
165   for(int i=0; i< remit.flowers.size(); ++i) {
166     out << remit.flowers.at(i);
167   }
168   return out;
169 };
170
171 #endif
```

D. Inventory

```
#ifndef _INVENTORY_HPP_
#define _INVENTORY_HPP_

#include <signal.h>
#include <stddef.h>
#include <unistd.h>

#include <string>
#include <unordered_map>

#include "log.hpp"
#include "flower.hpp"

class Inventory {
private:
  Pipe& in;
  Logger logger;
  const std::string fileName;
  static std::vector<Remit> remits;

  static void printReport(int signo) {
    std::unordered_map<int,int> producerTotal;
    std::unordered_map<Flower,int> flowerTotal;
    for(auto& r:remits) {
      for(auto& b:r.flowers) {
        auto produceCount = producerTotal.find(b.producer);
        if (produceCount != producerTotal.end()){
          ++(produceCount->second);
        } else {
          producerTotal[b.producer] = 1;
        }

        auto flowerCount = flowerTotal.find(b.type);
        if (flowerCount != flowerTotal.end()){
          ++(flowerCount->second);
        } else {
          flowerTotal[b.type] = 1;
        }
      }
    }

    Str reportProducer;
    reportProducer << "Producers:";
    int max = producerTotal.begin()->second;
    int maxProducer = producerTotal.begin()->first;
    for (auto & pair:producerTotal) {
      reportProducer << " " << pair.first << " <- " << pair.second << ";";
      if(pair.second > max) {
        maxProducer = pair.first;
        max = pair.second;
      }
    }
    root.info(reportProducer << "Max: " << maxProducer << " total " << max << ".");

    Str reportFlower;
    reportFlower << "Flowers:";
    max = flowerTotal.begin()->second;
    Flower maxFlower = flowerTotal.begin()->first;
    for (auto & pair:flowerTotal) {
      reportFlower << " " << pair.first << " <- " << pair.second << ";";
      if(pair.second > max) {
        maxFlower = pair.first;
        max = pair.second;
      }
    }

```

```
67      root.info(reportFlower << "Max: " << maxFlower << " total " << max << ".");
68    }
69
70  public:
71    Inventory(Pipe&in, const std::string& fileName) :
72      in(in),
73      fileName(fileName),
74      logger("Inventory") {
75      signal(SIGQUIT, Inventory::printReport);
76    }
77
78    void run() {
79      in.in().asStdIn();
80      while(std::cin.peek() != -1) {
81        Remit remit = deserialize<Remit>(std::cin);
82        logger.debug("ACK Remit");
83        Inventory::remits.push_back(remit);
84      }
85      logger.debug("finalizing");
86    }
87  };
88
89  std::vector<Remit> Inventory::remits;
90
91  #endif
```

E. Distributor

```
#ifndef _DISTRIBUTOR_HPP_
#define _DISTRIBUTOR_HPP_

#include <vector>
#include <string>
#include <algorithm>


#include "log.hpp"
#include "flower.hpp"
#include "pipe.hpp"
#include "storage.hpp"

class Distributor  {
private:
  Pipe& packets;
  Pipe& boxes;
  Logger logger;
  Storage storage;

public:
  Distributor(
              Pipe& boxes,
              Pipe& packets,
              const std::string & storageFile):

    boxes(boxes),
    packets(packets),
    storage(storageFile),
    logger("Distributor") {
  }

  void run() {
    boxes.in().asStdIn();
    packets.out().asStdOut();

    storage.loadStock();

    while(std::cin.peek() != -1) {
      Box box = deserialize<Box>(std::cin);
      logger.info(Str() << " in: " << box);
      for(auto& flower: box.flowers) {
        if(flower.type == ROSE) {
          storage.roses.push_back(flower);
        } else if(flower.type == TULIP) {
          storage.tulips.push_back(flower);
        } else {
          throw std::string("Unhandler flower type");
        }
      }
      attemptToDispatch(storage.roses, "roses");
      attemptToDispatch(storage.tulips, "tulips");
    }
    logger.debug("finalizing");

    storage.storeStock();

    packets.close();
    boxes.close();
  }

  Packet pack(std::vector<Bouquet> & flowers) {
    std::vector<Bouquet> result(FLOWER_PACKET_SIZE);
    std::copy_n(flowers.begin(), FLOWER_PACKET_SIZE, result.begin());
    flowers.erase(flowers.begin(), flowers.begin() + FLOWER_PACKET_SIZE);
    return Packet(result);
```

```
67    }
68
69    void attemptToDispatch(std::vector<Bouquet> & flowers, const std::string& type) {
70      if(flowers.size() >= FLOWER_PACKET_SIZE) {
71        Packet packet = pack(flowers);
72        logger.info(Str() << " out: " << packet);
73        std::cout << packet;
74      } else {
75        logger.info(Str() << " no ready to send " << type << ":" << flowers.size());
76      }
77    }
78
79  };
80
81  #endif
```

F. Log

```
#ifndef _LOG_HPP_
#define _LOG_HPP_

#include <iostream>
#include <sstream>
#include <unistd.h>
#include <sys/time.h>
#include <algorithm>

class Str {
private:
  std::stringstream ss;
public:

  template<typename T>
  Str& operator<<(const T & t) {
    ss << t;
    return *this;
  }

  Str& operator<<(std::ostream& (*fun)(std::ostream&)) {
    ss << fun;
    return *this;
  }

  std::string str() const {
    return ss.str();
  }
};

std::ostream &  operator<<(std::ostream & o, const Str & s) {
  o << s.str();
  return o;
}

class Logger {
private:
  std::string name;
  std::ostream& out;
  int FLAGS = 0xFFFF;
public:
  Logger(const std::string& name,
         std::ostream& out = std::cerr):
    name(name),
    out(out) {}

  std::ostream& log(const std::string& level, const std::string& msg) {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    std::string str(msg);
    str.erase(std::remove(str.begin(), str.end(), '\n'), str.end());

    std::stringstream stream;
    stream <<"[" << tv.tv_sec << " sec " << tv.tv_usec << " usec] "
           << "(" << getpid() << ") "
           << level << " "
           << name << " "
           << str
           << std::endl;


    out << stream.str();
    return out;
  }

  void info(const std::string& msg) {
```

```
67      log("INFO", msg);
68    }
69
70    void debug(const std::string& msg) {
71      if(FLAGS & 0x1)
72        log("DEBUG", msg);
73    }
74
75    void error(const std::string& msg) {
76      log("ERROR", msg);
77    }
78
79    void info(const Str& msg) {
80      log("INFO", msg.str());
81    }
82
83    void debug(const Str& msg) {
84      if(FLAGS & 0x1)
85        log("DEBUG", msg.str());
86    }
87
88    void error(const Str& msg) {
89      log("ERROR", msg.str());
90    }
91  };
92
93  Logger root("ROOT");
94
95  #endif
```

G. Main

```cpp
#include <stddef.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>

#include <iostream>
#include <vector>
#include <string>
#include <sstream>

#include "log.hpp"
#include "flower.hpp"
#include "shm.hpp"
#include "pipe.hpp"
#include "balancer.hpp"
#include "producer.hpp"
#include "distributor.hpp"
#include "sellpoint.hpp"
#include "inventory.hpp"


Mem<bool> stopFlag("/dev/null", 0, false);

void setStopFlag (int signo) {
  root.debug(Str() << "Signal " << strsignal(signo) << ".");
  *stopFlag = true;
};
struct Args {
  int producers;
  int distributors;
  int sellpoints;
  std::vector<std::string> requestFiles;
  std::vector<std::string> distributorsStorageFiles;
  std::vector<std::string> sellpointsStorageFiles;
  std::string inventoryFileName;
  Args(const std::string & configFile) {
    std::fstream config;
    config.open(configFile);
    config >> producers >> distributors >> sellpoints;
    for(int i=0; i < distributors; ++i) {
      std::string fileName;
      config >> fileName;
      distributorsStorageFiles.push_back(fileName);
    }

    for(int i=0; i < sellpoints; ++i) {
      std::string storageFileName, requestFileName;
      config >> storageFileName >> requestFileName;
      sellpointsStorageFiles.push_back(storageFileName);
      requestFiles.push_back(requestFileName);
    }

    config >> inventoryFileName;
    config.close();
  }
};

int main(int argc, char** argv) {
  try{
    Args args(argv[1]);
    signal(SIGINT, setStopFlag);
    std::vector<pid_t> children;
    // Create Producers;
    Pipe producerOut;
    for(int i = 0; i< args.producers; i++) {
      pid_t pid = fork();
```

```
67        if(pid == 0) {
68          Producer p(i,
69                     producerOut,
70                     stopFlag);
71          p.run();
72          producerOut.close();
73          return 0;
74        } else {
75          children.push_back(pid);
76        }
77      }
78      // Create DistributionCenters
79      Pipe distributorOut;
80      std::vector<Pipe> distributorsInPipes;
81      for(int i = 0; i< args.distributors; i++) {
82        Pipe distributorIn;
83        pid_t pid = fork();
84        if(pid == 0) {
85          producerOut.close();
86          Distributor d(
87                        distributorIn,
88                        distributorOut,
89                        args.distributorsStorageFiles[i]);
90          d.run();
91          distributorOut.close();
92          distributorIn.close();
93          return 0;
94        } else {
95          distributorsInPipes.push_back(distributorIn);
96          children.push_back(pid);
97        }
98      }
99      // Producers to distributors
100     pid_t pid = fork();
101     if(pid == 0) {
102       distributorOut.close();
103       Balancer distributorBalancer(producerOut, distributorsInPipes, "ProdToDistro");
104       distributorBalancer.run();
105       closeAll(distributorsInPipes);
106       producerOut.close();
107       return 0;
108     }
109     children.push_back(pid);
110     closeAll(distributorsInPipes);
111     producerOut.close();
112
113     // Create Sellpoints
114     Pipe inventoryPipe;
115     std::vector<Pipe> sellpointInPipes;
116     for(int i = 0; i < args.sellpoints; i++) {
117       Pipe sellpointIn;
118       pid_t pid = fork();
119       if(pid == 0) {
120         distributorOut.close();
121         SellPoint s(
122                     sellpointIn,
123                     inventoryPipe,
124                     args.requestFiles[i],
125                     args.sellpointsStorageFiles[i]);
126         s.run();
127         sellpointIn.close();
128         return 0;
129       } else {
130         children.push_back(pid);
131         sellpointInPipes.push_back(sellpointIn);
132       }
133     }
```

```
134     // distributor to sellpoints
135     pid = fork();
136     if(pid == 0) {
137       inventoryPipe.close();
138       Balancer sellpointsBalancer(distributorOut, sellpointInPipes, "DistroToSellPoint");
139       sellpointsBalancer.run();
140       closeAll(sellpointInPipes);
141       distributorOut.close();
142       return 0;
143     }
144     children.push_back(pid);
145     closeAll(sellpointInPipes);
146     distributorOut.close();
147 
148     // Inventory
149     pid=fork();
150     if(pid == 0){
151       Inventory inventory(inventoryPipe,
152                           args.inventoryFileName);
153       inventory.run();
154       return 0;
155     }
156     inventoryPipe.close();
157 
158     // Finish
159     for(auto i: children) {
160       waitpid(i, NULL, 0);
161     }
162   }catch(const char* str){
163     root.error(str);
164   }catch(const std::string & str) {
165     root.error(str);
166   }
167   return 0;
168 }
```

H. Pipe

```
#ifndef _PIPE_HPP_
#define _PIPE_HPP_

#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <string>
#include <vector>
#include "log.hpp"

class Closeable {
public:
  virtual void close() = 0;
};

class In :public Closeable {
protected:
  int fd;
public:
  In(const int fd): fd(fd) {}
  In(const In & i): fd(i.fd) {}

  size_t read(void* buffer, const int size) {
    return ::read(fd, buffer, size);
  }

  void asStdIn() {
    if(-1 == ::dup2(fd, 0))
      throw std::string(" asStdIn: ") + std::string(strerror(errno)) + "\n";
  }

  void close() {
    ::close(fd);
  }
};

class Out : public Closeable {
protected:
  int fd;
public:
  Out(const int fd): fd(fd) {}
  Out(const Out & o): fd(o.fd) {}

  ssize_t write(const void* buffer, const int size) {
    return ::write(fd, buffer, size);
  }

  template<typename T>
  Out& operator<<(const T & t) {
    std::stringstream ss;
    ss << t;
    std::string s = ss.str();
    write(s.c_str(), s.size());
    return *this;
  }

  void asStdOut() {
    if(-1 == ::dup2(fd, 1))
      throw std::string("asStdOut: ") + std::string(strerror(errno)) + "\n";
  }

  void asStdErr() {
    if(-1 == ::dup2(fd, 2))
      throw std::string("asStdErr: ") + std::string(strerror(errno)) + "\n";
  }
```

```
67
68    void close() {
69      ::close(fd);
70    }
71  };
72
73  class Writeable : public Closeable {
74  public:
75    virtual Out out() = 0;
76  };
77
78  class Readable : public Closeable {
79  public:
80    virtual In in() = 0;
81  };
82
83  class Pipe : public Writeable, public Readable {
84  private:
85          int fds[2];
86
87  public:
88
89    Pipe(const Pipe&other) {
90      this->fds[0] = other.fds[0];
91      this->fds[1] = other.fds[1];
92    }
93
94    Pipe& operator=(const Pipe& other) {
95      this->fds[0] = other.fds[0];
96      this->fds[1] = other.fds[1];
97      return *this;
98    }
99
100         Pipe() {
101     ::pipe(fds);
102   }
103
104   ~Pipe() {
105   }
106
107   In in() {
108     ::close(fds[1]);
109     return In(fds[0]);
110   }
111
112   Out out() {
113     ::close(fds[0]);
114     return Out(fds[1]);
115   }
116
117   void close() {
118     ::close(fds[0]);
119     ::close(fds[1]);
120   }
121 };
122
123 template<class T>
124 void closeAll(std::vector<T>& v) {
125   for(auto& c: v) {
126     c.close();
127   }
128 };
129
130 #endif
```

I. Producer

```
#ifndef _PRODUCER_HPP_
#define _PRODUCER_HPP_

#include <vector>
#include <string>
#include <stdlib.h>

#include "shm.hpp"
#include "log.hpp"

#include "flower.hpp"
#include "pipe.hpp"


class Producer {
private:
  Mem<bool>& stopFlag;
  Pipe& output;
  Logger logger;
  int producerId;
  int flowerCounter;
public:
  Producer(int id,
           Pipe& output,
           Mem<bool>& stopFlag):
    producerId(id),
    output(output),
    stopFlag(stopFlag),
    logger("Producer") {
  }

  Box randomBox() {
    Box box;
    box.flowers[0] = Bouquet(++flowerCounter, producerId, static_cast<Flower>(rand()%2));
    box.flowers[1] = Bouquet(++flowerCounter, producerId, static_cast<Flower>(rand()%2));
    box.flowers[2] = Bouquet(++flowerCounter, producerId, static_cast<Flower>(rand()%2));
    box.flowers[3] = Bouquet(++flowerCounter, producerId, static_cast<Flower>(rand()%2));
    box.flowers[4] = Bouquet(++flowerCounter, producerId, static_cast<Flower>(rand()%2));
    box.flowers[5] = Bouquet(++flowerCounter, producerId, static_cast<Flower>(rand()%2));
    box.flowers[6] = Bouquet(++flowerCounter, producerId, static_cast<Flower>(rand()%2));
    box.flowers[7] = Bouquet(++flowerCounter, producerId, static_cast<Flower>(rand()%2));
    box.flowers[8] = Bouquet(++flowerCounter, producerId, static_cast<Flower>(rand()%2));
    box.flowers[9] = Bouquet(++flowerCounter, producerId, static_cast<Flower>(rand()%2));
    return box;
  }

  void run() {
    Out out = output.out();
    out.asStdOut();
    while(!(*stopFlag)) {
      Box box = randomBox();
      logger.info(Str() << box);
      std::cout << box;
      sleep(1);
    }
    logger.debug("finalizing");
    output.close();
  }
};

#endif
```

J. Sellpoint

```
#ifndef _SELLPOINT_HPP_
#define _SELLPOINT_HPP_
#include <vector>
#include <string>
#include <fstream>
#include <algorithm>
#include <iterator>

#include "log.hpp"
#include "flower.hpp"
#include "pipe.hpp"

class SellPoint  {
private:
  Pipe& packets;
  Pipe& inventory;
  Logger logger;
  std::string requestFileName;
  Storage storage;
public:
  SellPoint(
            Pipe& packets,
            Pipe& inventory,
            const std::string & requestFileName,
            const std::string & storageFile):

    packets(packets),
    inventory(inventory),
    requestFileName(requestFileName),
    storage(storageFile),
    logger("SellPoint") {
  }

  void run() {
    packets.in().asStdIn();

    storage.loadStock();
    Out remits = inventory.out();

    std::fstream requestFile;
    requestFile.open(requestFileName);
    while(std::cin.peek() != -1) {

      Packet packet = deserialize<Packet>(std::cin);
      logger.info(Str() << "in:" << packet);
      for(auto& f : packet.flowers) {
        if(f.type == ROSE) {
          storage.roses.push_back(f);
        } else if(f.type == TULIP) {
          storage.tulips.push_back(f);
        } else {
          throw std::string("Unhandled rose type");
        }
      }

      if(requestFile.peek() != -1) {
        Request req = nextRequest(requestFile);
        logger.debug(Str () << "Process Request: " << req);
        std::vector<Bouquet> remitFlowers;
        if(req.roses > 0) {
          transferNFlowers(storage.roses, req.roses, remitFlowers);
        }
        if(req.tulips > 0) {
          transferNFlowers(storage.tulips, req.tulips, remitFlowers);
        }

```

```
67          Remit remit(remitFlowers, req);
68
69          if(req.type == INTERNET) {
70            logger.info(Str() << "internet: " << remit);
71          } else if(req.type = FRONTDESK){
72            logger.info(Str() << "frontdesk: " << remit);
73          } else {
74            throw std::string("Unhandled rose type");
75          }
76          remits << remit;
77        } else {
78          logger.debug("No more requests.");
79        }
80      }
81      logger.debug("finalizing");
82
83      storage.storeStock();
84
85      requestFile.close();
86
87      packets.close();
88      requestFile.close();
89      inventory.close();
90    }
91
92    void transferNFlowers(std::vector<Bouquet>& in, int size, std::vector<Bouquet> & out) {
93      int actualSize = (in.size() <= size)? in.size() : size;
94      if(actualSize > 0) {
95        std::copy_n(in.begin(), actualSize, std::back_inserter(out));
96        in.erase(in.begin(), in.begin() + actualSize);
97      }
98    }
99
100   Request nextRequest(std::fstream & file) {
101     return deserialize<Request>(file);
102   }
103 };
104
105 #endif
```

K. Storage

```
#ifndef _STORAGE_HPP_
#define _STORAGE_HPP_

#include <unistd.h>
#include <string>
#include <fstream>
#include <vector>
#include "flower.hpp"
#include "log.hpp"

class Storage {
public:
  std::vector<Bouquet> roses;
  std::vector<Bouquet> tulips;
  const std::string & storageFile;

  Storage(const std::string & storageFile) :
    storageFile(storageFile) {
    root.debug(Str() << "StorageFile:" << storageFile << std::endl);
  }

  void loadStock() {
    std::fstream storage;
    storage.open(storageFile);
    while(storage.peek() != -1) {
      Bouquet bouquet = deserialize<Bouquet>(storage);
      if(bouquet.type == ROSE) {
        roses.push_back(bouquet);
      } else if(bouquet.type == TULIP) {
        tulips.push_back(bouquet);
      } else {
        throw std::string("Unhandler flower type");
      }
    }
    storage.close();
    ::truncate(storageFile.c_str(), 0);
  }

  void storeStock() {
    std::fstream storage;
    storage.open(storageFile);
    for(auto& r: roses) {
      storage << r;
    }
    for(auto& t: tulips) {
      storage << t;
    }
    storage.close();
  }
};

#endif
```

L. Shm

```
#ifndef _SHM_HPP_
#define _SHM_HPP_

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string>
#include <string.h>
#include <iostream>
#include <errno.h>

template <class T> class Mem {
private:
  int   shmId;
  T*    data;
  int   countAttachedProcesses() const {
    shmid_ds estado;
    shmctl (this->shmId, IPC_STAT, &estado);
    return estado.shm_nattch;
  }
public:
  Mem(const std::string& file,
      const char index,
      const T& init) {
    key_t clave = ftok (file.c_str(), index);
    if(-1 == clave) {
      std::string mensaje = std::string("Error en ftok(): ") + std::string(strerror(errno));
      throw mensaje;
    }

    shmId = shmget (clave, sizeof(T), 0644 | IPC_CREAT);
    if(-1 == shmId) {
      std::string mensaje = std::string("Error en shmget(): ") + std::string(strerror(errno));
      throw mensaje;
    }

    void* tmpPtr = shmat (this->shmId, NULL, 0);
    if ((void*) -1 == tmpPtr) {
      std::string mensaje = std::string("Error en shmat(): ") + std::string(strerror(errno));
      throw mensaje;
    }

    data = static_cast<T*> (tmpPtr);
    *data = init;
  }

  Mem (const Mem& origen)
    : shmId(origen.shmId) {
    void* tmpPtr = shmat(origen.shmId, NULL, 0);
    if ((void*) -1 == tmpPtr) {
      std::string mensaje = std::string("Error en shmat(): ") + std::string(strerror(errno));
      throw mensaje;
    }
    data = static_cast<T*> (tmpPtr);
  }

  ~Mem() {
    int errorDt = shmdt(static_cast<void*> (data));
    if (-1 == errorDt) {
      std::cerr << "Error en shmdt(): " << strerror(errno) << std::endl;
    }
    if (countAttachedProcesses() == 0) {
      shmctl(shmId, IPC_RMID, NULL);
    }
  }

```

```
67    Mem<T>& operator=(const Mem& origen){
68      shmId = origen.shmId;
69      void* tmpPtr = shmat(this->shmId, NULL, 0);
70      if ((void*) -1 == tmpPtr) {
71        std::string mensaje = std::string("Error en shmat(): ") + std::string(strerror(errno));
72        throw mensaje;
73      }
74      data = static_cast<T*> (tmpPtr);
75      return *this;
76    }
77
78    Mem<T>& operator*(const T& data) {
79      *(this->data) = data;
80      return *this;
81    }
82
83    T& operator*(void) {
84      return *(this->data);
85    }
86  };
87
88  void testMem() {
89    Mem<int> mem("/dev/null", 0, 0);
90    int val = *mem;
91    std::cout <<"*Mem : " << val;
92    *mem = 1;
93    val = *mem;
94    std::cout <<"*Mem : " << val;
95  };
96
97  #endif
```