Max Link 48511338

API – not part of this rubric requirement. Summarized with ChatGPT for future referral

api.h

Class: MyJobSystemAPI

Private Members:

1. **JobSystem* jsForAPI**: An instance of a job system which manages job processing.
2. **std::map<std::string, Job*> jobsForAPI**: A map associating job names to job pointers.
3. **std::vector<std::string> jobOrder**: A vector to maintain the order in which jobs should be executed.
4. **std::mutex jobMutex**: A mutex to ensure thread safety while accessing the job system.
5. **JobFactory* jobFactory**: An instance of a factory to create custom jobs.

Public Methods:

1. **Constructor** **MyJobSystemAPI()**: Initializes the job system and job factory.
2. **Destructor** **~MyJobSystemAPI()**: Deletes the job factory instance.
3. **void getJobType()**: Prints out the possible job types.
4. **void createJob(const std::string& jobType)**: Creates a job based on the given job type.
5. **void destroyJob(const std::string& jobType)**: Destroys a specific job.
6. **void completeJob(const std::string& jobType)**: Completes a specific job.
7. **void getJobStatus(const std::string& jobType)**: Gets the status of a specific job.
8. **void startJobSystem()**: Initializes the job system and its worker threads.
9. **void stopJobSystem()**: Stops the job system.
10. **void destroyJobSystem()**: Destroys the job system.
11. **void dependency(const std::string& jobType1, const std::string& jobType2)**: Sets a dependency between two jobs.
12. **void createCustomJob(const std::string& jobType)**: Creates a custom job using the job factory.
13. **void registerCustomJob(JobFactoryFunction factoryFunction, const std::string& jobTypeName)**: Registers a custom job with the job factory.

api.cpp

1. **Constructor** **MyJobSystemAPI::MyJobSystemAPI()**: Initializes the job system and job factory instances.

2. **Destructor** `MyJobSystemAPI::~MyJobSystemAPI()`: Cleans up memory allocated to the job factory.
3. `void getJobType()`: Prints available job types.
4. `void createJob(const std::string& jobType)`: Creates a job of the specified type and adds it to the internal map and order list.
5. `void destroyJob(const std::string& jobType)`: Deletes the specified job from memory and removes it from the internal map.
6. `void getJobStatus(const std::string& jobType)`: Prints the status of a specified job.
7. `void completeJob(const std::string& jobType)`: Completes a specified job in the job system.
8. `void startJobSystem()`: Sets up the job system, including creating worker threads and queuing jobs in the specified order.
9. `void destroyJobSystem()`: Cleans up the job system.
10. `void stopJobSystem()`: Stops the job system without destroying it.
11. `void dependency(const std::string& jobType1, const std::string& jobType2)`: Modifies the job order based on the dependency between two jobs.
12. `void createCustomJob(const std::string& jobType)`: Creates a custom job and adds it to the internal map and order list.

This code defines an API for a job system, allowing users to create, manage, and destroy jobs. The API also supports the creation of custom jobs through a factory pattern.

FLOWSCRIPT PROCESSES:

EXAMPLE CASE ONE – Basic Job Flow:

Flowscript:

```
digraph G {

  subgraph cluster_0 {
    style=filled;
    color=lightgrey;
    node [style=filled,color=white];
    label = "Basic Job Flow";
  }

  a1[label="A
  ID
  Type"];
  a2[label="B
  ID
  Type"]
  a3[label="C
```

```
 ID
 Type"];

 a1->a2[label="Dependencies"]
 a2->a3
}
```

Elements used:

Containers hold:

- Job name. I.E. Job A, Job B, Job C etc.
- Job ID
- Job Type
- These are variables

Dependencies are used to show that one job relies on another job.

Why they are needed:

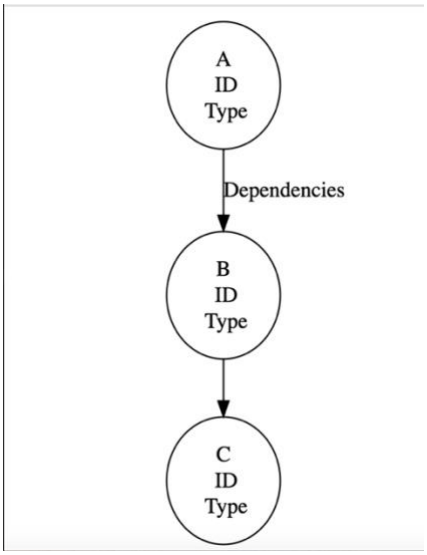
Containers are needed to hold jobs and job attributes. The dependencies are needed to ensure that jobs complete in the order they are supposed to run I.E. job A before job B.

Representation:

Flowscript represents these elements as follows:

- Nodes represent the containers that hold the job name, Job ID, and Job type. These elements (Job name, Job ID, and job Type) are variables that are contained inside a container.
- The dependencies are represented as edges that connect each job container. This representation makes it easy to see which jobs rely on each other.

Visualization:



EXAMPLE CASE TWO – Control Statement Flow

Flowscript:

```
digraph G {

  subgraph cluster_0 {
    style=filled;
    color=lightgrey;
    node [style=filled,color=white];
    label = "Control Statement Flow";
  }

  a1[label="A
  ID
  Type"];
  ControlStatement[shape=Mdiamond, label="a.outv1>a.outv2"];
  a2[label="B
  ID
  Type"];
  a3[label="C
  ID
  Type"];
  a4[label="D
```

```
 ID
 Type"]

 a1->ControlStatement[label="Dependencies"];
 ControlStatement->a2[label="True"];
 ControlStatement->a3[label="False"];
 a3->a4[label="Dependencies"];
}
```

Containers hold:

- Job name. I.E. Job A, Job B, Job C etc.
- Job ID
- Job Type
- These are variables

This flow includes a control statement. The control statement takes in two variables:

- A.outv1
- A.outv2
- Compares these two variables with a > operator.
- If variable a.outv1 is greater than a.outv2, then the control statement evaluates to true and branches to the left. This branch runs job B. If a.outv1 is < a.outv2, the expression evaluates to false and job C and D are ran in that order of dependency.

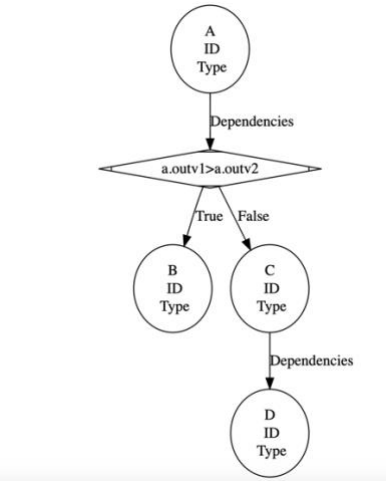
Why they are needed:

- This added control statement allows a program to perform different actions depending on the condition that is entered into the program. This gives the language more versatility.

Representation:

- The control statement is represented by a diamond.
- The variables and > operator are represented as text inside the diamond written as: a.outv1 > a.outv2

Visualization:



EXAMPLE CASE THREE – Loop Flow

Flowscript:

```
digraph G {

  subgraph cluster_0 {
    style=filled;
    color=lightgrey;
    node [style=filled,color=white];
    label = "Loop Flow";
  }

  a1[label="A
  ID
  Type"];
  a2[label = "B
  ID
  Type
b.outv"]
  Loop[shape=square, label="Loop
  b.outv == 0"]
  a3[label="C
  ID
  Type"];

  a1->a2[label="Dependencies"]
  a2->Loop
```

```
  Loop->a2
  Loop->a3
}
```

Elements used:

Containers hold:

- Job name. I.E. Job A, Job B, Job C etc.
- Job ID
- Job Type
- These are variables

This flow includes a loop that checks the output of job B. This loop takes in a Boolean variable called b.outv. b.outv starts as 0, indicating that Job B has not finished its task. Job B updates b.outv to 1 at the end of its process. If the flag fails to update, then its assumed that Job B did not run, which means the loop should initiate and re-run job B.

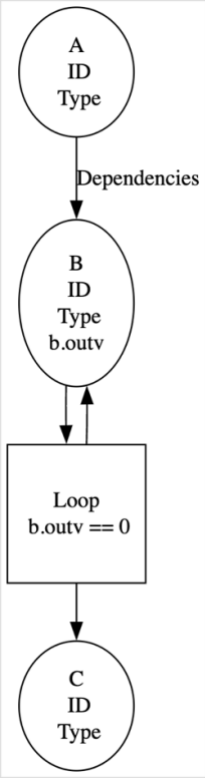
Why they are needed:

The Boolean flag is needed to ensure loop functionality. The loop allows the program to go back and re-run jobs or run jobs multiple times depending on the Boolean flag condition. Being able to run jobs multiple times or simply re-run them, expands program flexibility and security.

Representation:

The loop is represented by a square. The Boolean flag that the loop takes in is represented by b.outv. b.outv is contained inside Job B to communicate that Job B updates the bool at the end of its cycle.

Visualization:



These example cases give various scenarios of a programming language. Example case one provides basic job flow with jobs being dependent on one another. Example case two provides a control statement so that the language can situationally run. Example case three provides a loop flow so that multiple processes can be ran quickly. These control structures expand the language because they introduce the need for variables to hold values, both Boolean variables and integer variables. The a.outv1 and a.outv2 can be thought of as integer variables for the if statement conditional in case two. The b.outv can be thought of as either an integer or a Boolean variable for the loop in case three. Variables need to be defined like this in this language so that we can run situational programs like case two and case three.

Syntax and Semantics

Overall the language includes:

- Variable integers and Booleans
- Process containers → for running jobs or other processes
- Edges that represent process dependencies
- Variables for process ID, TYPE, and Name

    - Name variables are A-C in this case that hold a string for identifying the process
    - ID is an integer variable that holds an # for the process
    - TYPE is a string variable that identifies the type of process
- Loops are represented as squares
    - These loops perform a check on an integer/Boolean variable to determine if a loop is necessary or not
- if statements are defined as diamonds that take in two variable integers (in case two: a.outv1 and a.outv2) and compares them. If a.outv1 is greater than a.outv2 then the if statement evaluates to true and one process happens. On the other hand, if the statement evaluates to false, the other path is chose.
- This introduces Booleans. Booleans are important for this language because they allow us to make conditional code. We should have Boolean variables in this language defined as b.outv1,b.outv2,b.outv3…etc. You can think of case three's b.outv as either an integer or Boolean, either work, but I prefer Boolean for clearer language architecture.
- Object-Oriented Programming (OOP): The paradigm used for my language is OOP, which combines data and behavior. OOP emphasizes encapsulation, inheritance, polymorphism, and abstraction to generate very readable code that humans can understand.