Process Scheduling Simulator

운영체제 (Operating System)

Team So-So

Contents

1 **소개**

- So-So 조원 소개

2 **구현 방법**

- 구현 방법
- 코드 설명
- 자체 알고리즘

3 **시연**

- 시뮬레이터 설명
- 시뮬레이터 시연

4 **마무리**

-개인별 성찰 및 소감
- Q&A

1

소개

소개 > So - So 조원 소개

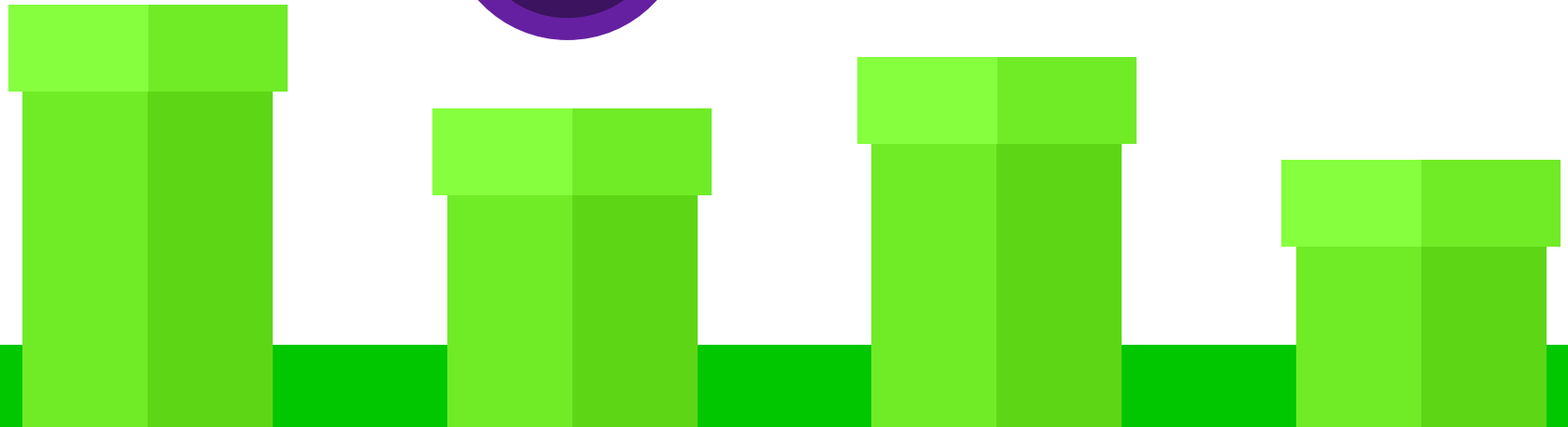
1	이해민	조 장 : 프로젝트 총괄, 보고서 작성
2	박상준	조원1 : 알고리즘, UI, 멀티코어 구현
3	김형구	조원2 : UI, 알고리즘, 오류 메시지, 도움말 구현
4	박금도	조원3 : PPT & 발표, 알고리즘, 소비전력 구현

1

소개

2

구현
방법



구현 > 구현 방법

개발 언어

개 발 환 경 : Eclipse IDE
개 발 언 어 : Java 11
프레임 워크 : Javafx
GUI Tool : Scene Builder

주요 기능

입력 : 알고리즘 선택, 코어 제어, 프로세스 입력
출력 : 실행, 결과 출력

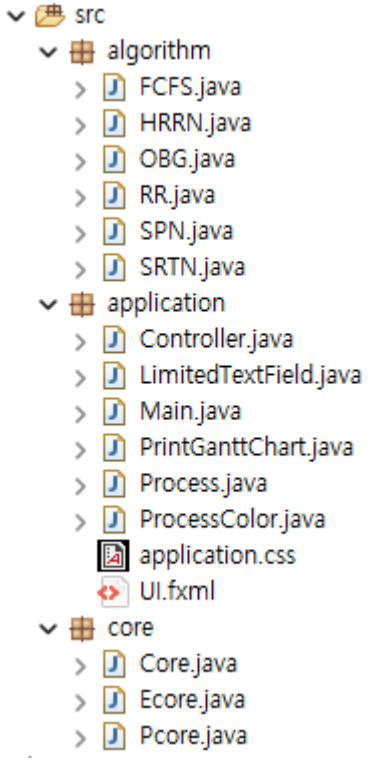
부가 기능

경고창 : 잘못된 값을 입력 받을 시 사용자에게 알림
도움말 : 프로세스 간단 설명

구현 > 코드 설명 > 패키지 구성

패키지

1. algorithm, application, core 3개의 패키지로 구성
2. algorithm : 스케줄링 알고리즘 패키지
3. application : 가시화와 프로세스 패키지
4. core : 프로세서 패키지



구현 > 코드 설명 > Process.java

멤버 변수

1. processName : 프로세스의 이름
2. arrivalTime : AT
3. burstTime : BT
4. waitingTime : WT
5. turnaroundTime : TT
6. NormalizedTT : NTT
7. remainBurstTime : 실시간 BT
8. TimeQ : 타임 퀀텀
9. remainTimeQ : 실시간 타임 퀀텀

```
public class Process {
    private SimpleStringProperty processName;
    private SimpleIntegerProperty arrivalTime;
    private SimpleIntegerProperty burstTime;
    private SimpleIntegerProperty waitingTime;
    private SimpleIntegerProperty turnaroundTime;
    private SimpleDoubleProperty NormalizedTT;
    private SimpleIntegerProperty remainBurstTime;
    private SimpleIntegerProperty TimeQ;
    private SimpleIntegerProperty remainTimeQ;
```

구현 > 코드 설명 > Core.java

멤버 변수

1. Process
: 코어에 들어오는 프로세스
2. visited
: 코어 내 프로세스 유무
3. prevUsed
: 코어가 쉬고 있었는지 유무
4. powerConsumption
: 소비 전력량

```
public class Core {
    private Process process;
    private boolean visited = false;
    private boolean prevUsed = false; //
    private double powerConsumption = 0;
```

구현 > 코드 설명 > Core.java

메소드

1. setComponent()
: TT, NTT, WT를 계산하여 설정(_p는 선점용)

2. emptyCore()
: 코어에 빈칸 할당

3.setPower()
: 전력 소모량 갱신

4. work()
: 1초 동안 프로세스 작업. 종료 여부를 반환
(RRwork, SRTNwork는 work를 수정한 전용 work 메소드)

```
public void setComponent(int time) {}
public void setComponent_p(int time) {}
public void emptyCore() {} //
public void setPower() {} //
public boolean work(int time) { return false; }
public boolean RRwork(int time, LinkedList<Process> waitingList) { return false; }
public boolean SRTNwork(int time, LinkedList<Process> waitingList) { return false; }
```

구현 > 코드 설명 > FCFS.java

중요 코드

1. 6가지 알고리즘은 algorithm패키지에 각각 존재.
2. Run 메소드를 통해 동작.
3. 6가지 알고리즘은 2번째 for문에 할당 방식 부분이 수정되어 코딩.
4. Core에 있는 work 메소드를 통해 작업. (RR과 SRTN는 전용 work 메소드 사용)

```
public void run(Core[] coreList, LinkedList<Process> waitingList, LinkedList<Process> endList,
        ObservableList<Process> obser, ObservableList<Process> observableList, ScrollPane GanttChart,
        Text[] corePower, Text Avgresponsetime, TableView<Process> tableView2,Button[] Buttons,Choice

    timeline = new Timeline(new KeyFrame(Duration.seconds(0.1), event -> {
        // 입력받은 프로세스의 AT가 현재 시간이 되면 waitingList에 삽입
        for (Process p : obser) {
            if (p.getArrivalTime() == time)
                waitingList.add(p);
        }

        // 남은 코어 자리에 waitingList에 있는 프로세스들을 번호 순서대로 할당함
        for (int index = 0; index < 4; ++index) {
            if (coreList[index] != null && !coreList[index].isVisited() && !waitingList.isEmpty()) {


                coreList[index].setProcess(waitingList.poll());
                coreList[index].setVisited(true);
            }
        }
```

```
// 코어별로 1초만큼의 일을 함.
// BT만큼의 작업을 다하면 작업이 완료되고, 작업이 완료되면 end큐에 넣음
for (int index = 0; index < 4; ++index) {
    if (coreList[index] != null && coreList[index].isVisited()) {
        boolean done = coreList[index].work(time);
        if (done)
            endList.add(coreList[index].getProcess());
    } else if (coreList[index] != null)
        coreList[index].setprevUsed(false); // 프로세스가 할당되지 않은 코어는 prevUsed를 false로 설정
}
```

구현 > 코드 설명 > Algorithm

SPN

```
// 남은 코어 자리에 waitingList에 있는 프로세스들을 BT가 낮은 순서대로 할당함
for (int index = 0; index < 4; ++index) {
    if (coreList[index] != null && !coreList[index].isVisited() && !waitingList.isEmpty())


        int minidx = -1;
        int minbursttime = 10000;
        int i = 0;

        for (Process p : waitingList) { // waitingList에서 BT가 가장 낮은 인덱스를 찾음
            if (minbursttime > p.getBurstTime()) {
                minbursttime = p.getBurstTime();
                minidx = i;
            }
            ++i;
        }

        Process deleteP = waitingList.get(minidx);
        waitingList.remove(deleteP);
        coreList[index].setProcess(deleteP);
        coreList[index].setVisited(true);
    }
}
```

SRTN

```
// 남은 코어 자리에 waitingList에 있는 프로세스들을 남은 BT가 낮은 순서대로 할당함
for (int index = 0; index < 4; ++index) {
    if (coreList[index] != null && !coreList[index].isVisited() && !waitingList.isEmpty()) {


        int minidx = -1;
        int minremainbursttime = 10000;
        int i = 0;

        for (Process p : waitingList) { // waitingList에서 남은 BT가 가장 낮은 인덱스를 찾음
            if (minremainbursttime > p.getremainBurstTime()) {
                minremainbursttime = p.getremainBurstTime();
                minidx = i;
            }
            ++i;
        }

        Process deleteP = waitingList.get(minidx); // minidx에 위치하는 프로세스를 리스트에서 삭제할거임
        waitingList.remove(deleteP);
        coreList[index].setProcess(deleteP);
        coreList[index].setVisited(true);
    }
}
```

구현 > 코드 설명 > Package > Algorithm

HRRN

```
// 남은 코어 자리에 waitingList에 있는 프로세스들을 responseRatio가 높은 순서대로 할당함
for (int index = 0; index < 4; ++index) {
    if (coreList[index] != null && !coreList[index].isVisited() && !waitingList.isEmpty()) { // 코어리스트
                                                                                              // OFF가
                                                                                              // 미사용
        int minidx = -1;
        double maxresponseRatio = 0;
        int i = 0;

        for (Process p : waitingList) { // waitingList에서 BT가 가장 낮은 인덱스를 찾음
            double responseRatio = (double) (p.getWaitingTime() + p.getBurstTime()) / p.getBurstTime();
            if (maxresponseRatio < responseRatio) {
                maxresponseRatio = responseRatio;
                minidx = i;
            }
            ++i;
        }

        Process deleteP = waitingList.get(minidx); // minidx에 위치하는 프로세스를 리스트에서 삭제할거임.
        waitingList.remove(deleteP);
        coreList[index].setProcess(deleteP);
        coreList[index].setVisited(true);
    }
}
```

구현 > 코드 설명 > PrintGanttChart.java

시각화

1. css코드를 사용하여 코딩.
2. 알고리즘이 실행 될 때, print메소드를 이용하여 간트 차트 출력.

```
package application;

import core.Core;

public class PrintGanttChart {
    ProcessColor processcoler = new ProcessColor();
    VBox v = new VBox();

    public void print(Core[] Corelist, ScrollPane GanttChart, HBox hBox, int time) {
        Label sec = new Label(time + 1 + "");
        sec.setStyle("-fx-font-size: 15px; -fx-font-family: \"Arial\"; -fx-pref-width: 60px; -fx-pref-h
        v.getChildren().add(sec);
        for(int i = 0; i < 4; i++) {
            if(Corelist[i] != null) {
                if(Corelist[i].getProcess() != null) {
                    String PID = Corelist[i].getProcess().getProcessName();
                    int num = Integer.parseInt(PID.replaceAll("[^0-9]", "")) - 1; // 프로세스 이름에서 숫자만
                    String Pcolor = processcoler.getColor(num);
                    Label pid = new Label(PID + " ");
                    pid.setStyle("-fx-font-size: 20px; -fx-border-color: black; "
                            + "-fx-pref-width: 60px; -fx-pref-height: 60px; "
                            + "-fx-alignment: CENTER; -fx-background-color: " + Pcolor + ";");
                    v.getChildren().add(pid);
                }
                else {
                    Label idle = new Label(" ");
                    idle.setStyle("-fx-font-size: 20px; -fx-border-color: black; "
                            + "-fx-pref-width: 60px; -fx-pref-height: 60px; -fx-alignment: CENTER;");
                    v.getChildren().add(idle);
                }
            }
            else {
                Label idle = new Label(" "); // 코어 Off시 빈칸
                idle.setStyle("-fx-font-size: 20px; -fx-font-color: white;"
                        + "-fx-pref-width: 60px; -fx-pref-height: 60px; -fx-alignment: CENTER;");
                v.getChildren().add(idle);
            }
        }

        hBox.getChildren().add(v);
        GanttChart.hvalueProperty().bind(hBox.widthProperty()); // 수평 스크롤바 갱신
        GanttChart.setContent(hBox);
    }
}
```

구현 > 코드 설명 > 소비전력

중요 코드

1. Core 클래스에 저장된 변수 사용.
2. prevUsed를 확인하여 시동 전력을 추가로 더함.
3. Core를 상속한 Pcore, Ecore에서 각각 계산.

```
public class Core {
    private Process process;
    private boolean visited = false;
    private boolean prevUsed = false; //
    private double powerConsumption = 0;
```

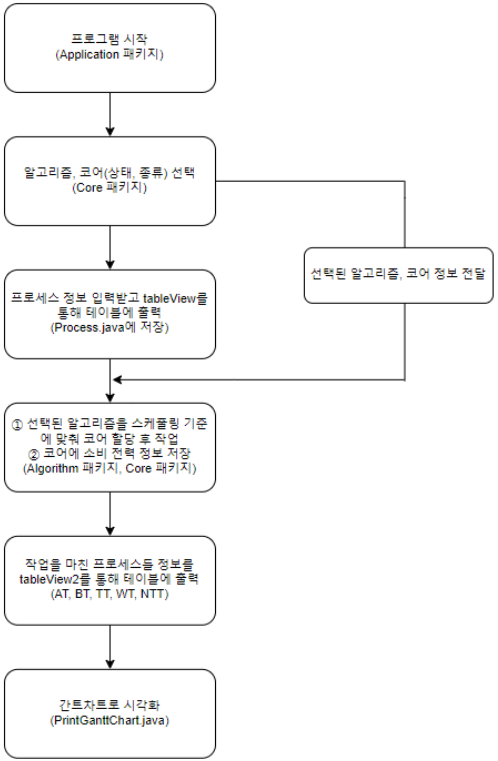
```
// 전력 소모량 갱신
@Override
public void setPower() {
    if (!this.isprevUsed()) {
            setPowerConsumption(getPowerConsumption() + 0.1);
            this.setprevUsed(true);
        }
    setPowerConsumption(getPowerConsumption() + 1);
}

@Override
public void setPower() {
    if (!this.isprevUsed()) {
            setPowerConsumption(getPowerConsumption() + 0.5);
            this.setprevUsed(true);
        }
    setPowerConsumption(getPowerConsumption() + 3);
}
```

구현 > 코드 설명 > 동작 과정

동작 과정

1. 프로그램 시작
2. 알고리즘, 코어 선택
3. 프로세스 입력, tableView로 출력
4. 2에서 선택한 알고리즘을 기준으로 코어 할당 및 작업 -> 소비 전력 실시간 저장
5. 작업이 종료 된 프로세스를 tableView를 통해 출력
6. 간트차트로 시각화



구현 > 자체 알고리즘(One-By-Group)

목적

동시 실행이 문제가 되는 프로세스를 관리하기 위해

내용

프로세스에 group 속성 추가, group이 같은 프로세스가 이미 프로세서에 있을 경우 그 프로세스를 실행되지 않는다.

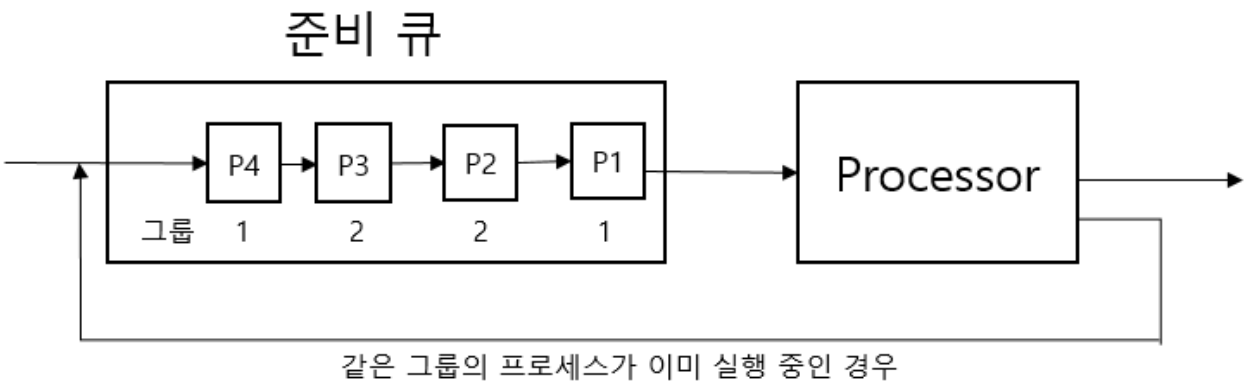
특징

선점 여부 : Non-preemptive scheduling
기준 : 도착 시간, 먼저 도착한 프로세스를 먼저 처리

장단점

장점 : 충돌 방지
단점 : Convoy effect, 긴 평균 응답시간, 멀티 프로세서에서만 유효

구현 > 자체 알고리즘(One-By-Group)

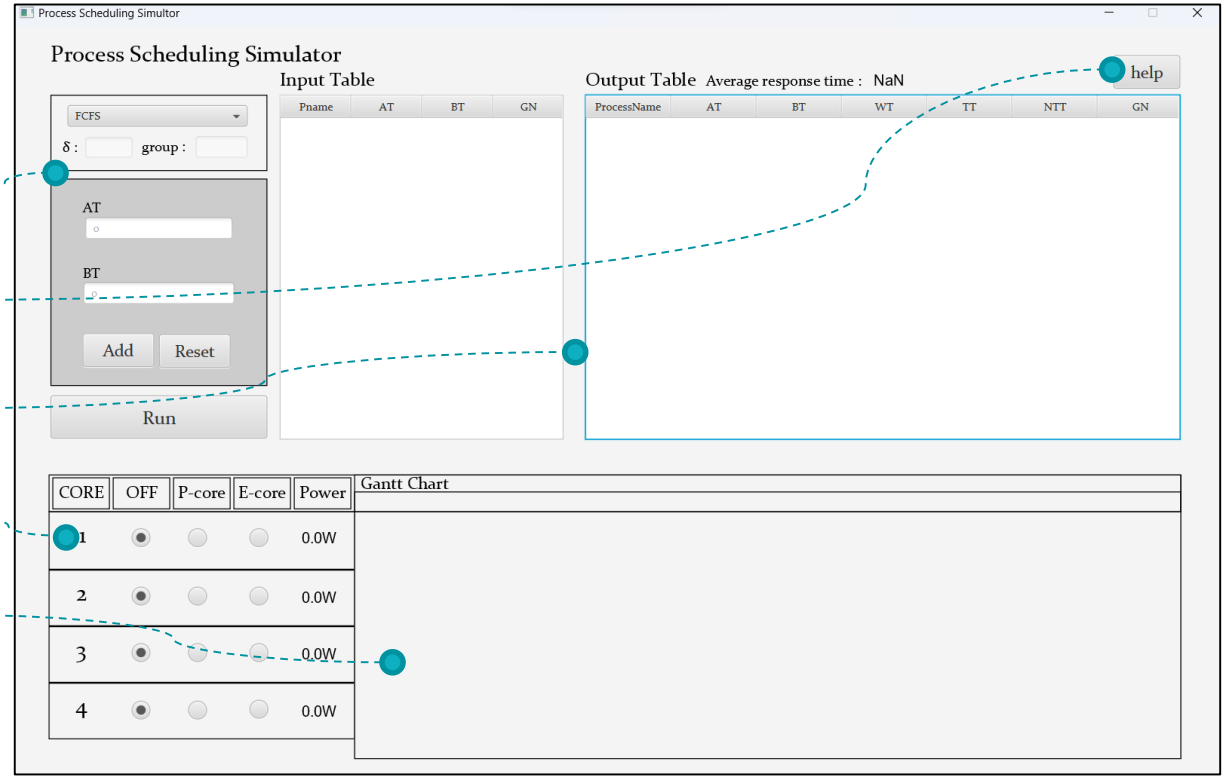



1
소개
2
구현
방법
3
시연

시연 > 시뮬레이터 설명

UI 상세

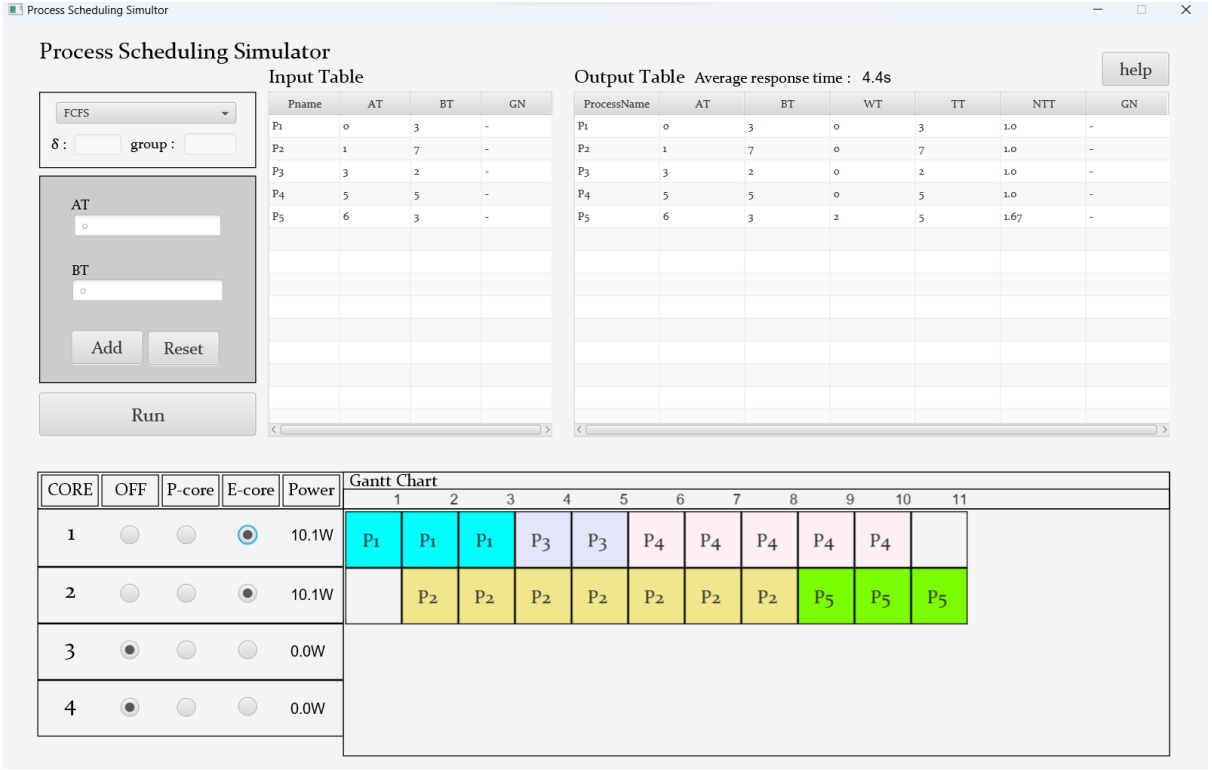
- 알고리즘 선택 & 프로세스 입력
- 도움말 버튼
- 입력 받은 프로세스 정보 출력
  실행이 끝난 프로세스 정보 출력
- 코어 on&off, 종류 선택
- 알고리즘이 실행되는 모습을
  시간대별로 간트 차트로 출력



시연 > 시뮬레이터 시연

시연 과정

- 알고리즘 선택 및 프로세스 입력
- 코어 선택
- 5가지 알고리즘 시연
- 자체 알고리즘 시연
- 시연 결과 확인



1

소개

2

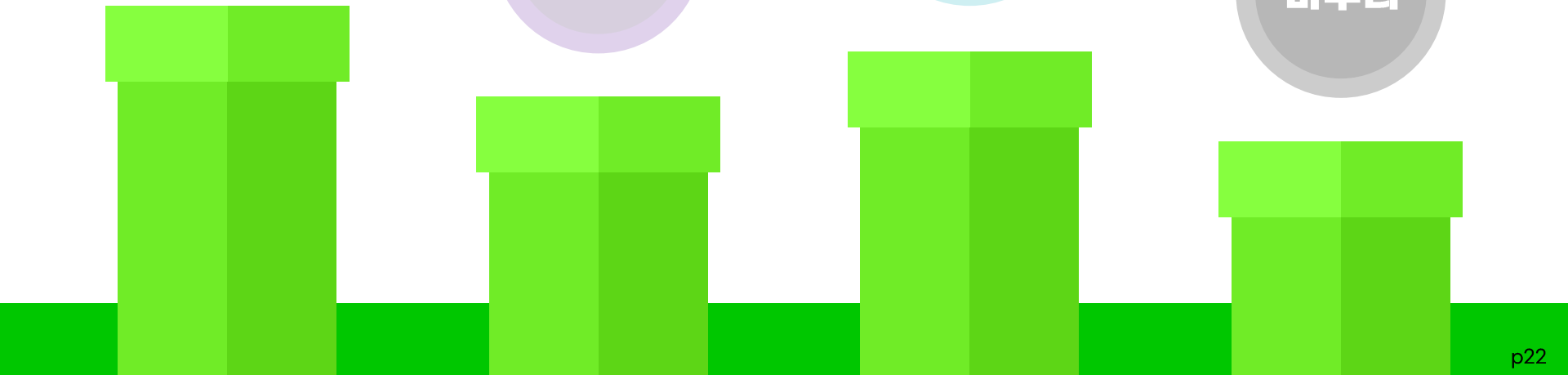
구현
방법

3

시연

4

마무리



마무리 > 성찰 & 소감

이해민

교수님께서 수업시간에 프로젝트에서 대해서 설명을 해주시고 이전 선배님들의 완성본을 보여주셨었는데 처음 봤을 때 프로젝트 제출은 할 수 있을지 걱정이 될 만큼 어렵게 다가왔다. 팀원들과 프로젝트를 시작하고 초반 부분에서는 스케줄링 알고리즘에 대해 어떻게 구현해야 할지에 대한 문제가 많았는데 박상준과 김형구의 첫 시작이 컸다고 생각한다. 어떻게 작성할지 막막하던 상황에서 기초 틀을 잡아주었고 그것에 맞춰 이후 다른 알고리즘들을 구현해 나가는 것은 처음보다는 진전이 있다고 느꼈다. 하지만 여러명이서 나눠서 진행하는 만큼 공유해야될 상황이 매우 많았는데 깃허브를 사용하는게 익숙하지 않아 코드가 바뀔 때 마다 하던 작업을 보내주고 팀원이 보내주면 하던 작업에 합치거나 수정해서 다시 보고 하는 부분이 매우 불편했다. 깃허브를 중요하게 생각하지 않고 있었는데 이번 팀프로젝트를 진행하면서 깃허브에 중요성을 크게 느낄 수 있었다. UI부분은 관여한 부분이 없다시피 하는데 알고리즘을 구현하는 것보다 사용자에게 보여질 UI가 더 중요하다는 생각이 많이 들었다. 알고리즘을 아무리 잘 구현하더라도 UI가 좋지 않다면 사용하기 힘들 것이고 사용자는 줄 것 이다. 프로그램을 만드는 부분에서 UI를 만드는 디자인 부분 일이 중요하다고 생각하게 되었다. 마지막으로 이번 프로젝트를 진행하면서 지금 내가 알고리즘이나 코드를 작성하는 부분에 있어서 어느 정도 수준인지 조금은 알 수 있게 된 것 같고 UI, 깃허브의 중요성 등 여러 부분에서 아직 알아가야 할 부분이 많다는 것을 느끼는 시간들이었다.

마무리 > 성찰 & 소감

박상준

이때까지는 알고리즘 문제를 풀거나 아무리 커봤자 부모클래스와 자식클래스 정도만 다뤘었는데 이렇게 큰 규모의 프로젝트를 해본 적은 처음이었다. 그래서 그때그때 필요한 코드를 작성해도 코드가 프로젝트에서 정상적으로 동작하도록 적용하는데 많은 시간이 필요했다. 작성하다보니 코드 중복이 많이 발생했는데 줄이는게 굉장히 힘들었고, 또 아직도 많이 줄이지 못한 점이 아쉽다. 코드 관리를 하는데에 있어서도 구조를 잘짜놓으면 유지보수하는데 얼마나 편리한지 느끼게 되었다. 깃허브로 프로젝트를 진행하려고 했는데 사용법을 숙지하는데 시간이 걸릴것 같아서 각자 작업을 한 코드를 export해서 공유했는데 생각보다 너무 불편했다. 깃허브와 같은 버전 관리 툴의 필요성을 절실히 느끼게 되었다. 앞으로 이와 같은 프로젝트를 진행하게 된다면 깃허브를 사용할 생각이다. 실제로 알고리즘 자체를 구현하는데는 오래 걸리지 않았는데 사용자에게 입력과 출력에 대한 UI를 지원하고 사용자 친화적으로 표현하는게 어려웠다. 결국 사용자에게 보여지는 것은 디자인적인 부분이지만 디자인에 대해서는 지식이 없어서 별다른 작업을 하지 못했는데 그 부분이 아쉽다. 규모있는 기업에 디자인부서가 왜 있는지 알 것 같았다. 그래도 직접 짠 코드가 동작하는 것을 보니 뭔가 뿌듯하다. 알고리즘 문제를 풀거나 작은 메소드를 구현하는 것보다 프로젝트 전반을 만들어보니 내가 무엇을 모르고 있었는지 알게 되었고 프로그래밍 능력이 크게 향상된 것 같아 좋았다.

마무리 > 성찰 & 소감

김형구

이번 프로젝트로 인한 여러가지 기술을 배울 수 있어서 좋았습니다. 특히 이번 프로젝트에서 사용한 언어인 java에 있는 javafx를 사용함으로써 어플리케이션을 어떻게 만드는지 알게 되었고 버튼과 마우스의 상호작용이 어떻게 일어나는지 알게 되었으며 다 만들고 나니 많은 버그들이 있었습니다. 예를 들어 at와 bt에 음수가 들어가면 안되는데 들어가게 끔 프로그램이 짜줘있는 버그 같은 것이 있었습니다. 그런 버그를 하나 하나 찾고 수정하면서 버그가 얼마나 중요하며 그 버그를 어떻게 수정 할 것인지 또한 버그가 발생할 때 유저에게 어떤 방식으로 전달해야하는지 깊은 고민을 하였습니다. 그리고 이번에 가장 아쉬웠던 점은 버전 관리 프로그램을 사용하지 않는 점이였습니다. 버전 관리 프로그램을 사용하지 않음으로써 여러가지 불편한 상황들이 나왔습니다. 다음에는 버전 관리 프로그램을 사용하는 방향으로 하고 싶습니다.

마무리 > 성찰 & 소감

박금도

이전까지 과제를 풀고 시험을 볼 때에는 우리가 배웠던 부분 내에서 정답이 있는 코딩만을 해왔었는데, 이렇게 큰 규모의 프로젝트를 해보니 지금까지 배워 온 것들을 활용해야 할 뿐만아니라 시뮬레이터 개발을 위해 더 신경 써야 할 부분이 많았고 한 문제, 한 문제 개별의 문제가 아닌 하나의 시뮬레이터를 만드는 과정이다 보니 기능 하나를 추가하는 것도 기능 하나를 그냥 덧붙이는 것이 아니라 더 신경 써야 할 부분이 많다는 것을 알게 되었습니다. 그렇다 보니 프로젝트를 진행하며 깃허브에 대한 이야기를 자주 했었습니다. 프로젝트를 시작할 때에는 큰 어려움을 느껴본 적이 없어 깃허브의 필요성을 그렇게 크게 느끼지 못했는데, 진행 도중 진행상황이 제대로 전달되지 않거나 불필요한 중복이 생기기도 하고 예상하지 못한 오류나 생기기도 하는 등 이전에는 경험해보지 못한 여러 문제들과 마주했습니다. 그러는 와중에 RR과 HRRN과 같은 비선점 알고리즘을 코딩하는 것은 생각만큼 쉽게 해결되지 않고 시간만 허비되어 중간부터는 이 프로젝트가 제대로 완성될 수 있을까 하는 걱정을 많이 했습니다. 고맙게도 다른 조원들이 이런 부분을 보충해주고 프로젝트에 더 신경써줬기에 지금은 처음 프로젝트를 받았을 때 생각했었던 결과물보다 더 좋은 결과물이 나올 수 있었습니다. 이번 프로젝트를 통해 제가 부족한 점을 확인할 수 있었고 앞으로 어떤 부분을 더 신경 쓰고 공부해야 하는지 알게 되었습니다.

Q&A