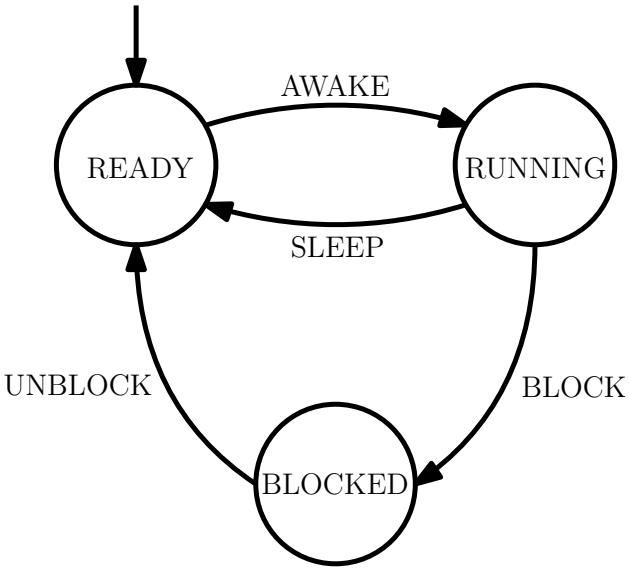
amU Faculté des sciences Aix Marseille Université

Année 2024-25

Domaine Sciences et Technologies
Licence Informatique : math-info
Initiation au Génie logiciel : TD 2
Codes UE : SIN5U34
Gestionnaire de processus

1 Gestionnaire de processus

On veut simuler (de façon très simplifiée) un gestionnaire de processus d'un système d'exploitation. Chaque processus possède un nom, un temps d'exécution prédéfini (en nombre de quantums de temps) et un état. Le gestionnaire de processus possède une liste de processus parmi lesquels il peut choisir le prochain à exécuter pendant un quantum de temps. Un processus peut avoir trois états qui dépendent des signals reçus par le processus.



2 Classe `Process`

Les instances de `Process` représentent des processus avec un état et une durée d'exécution en quantum.

```
public class Process {
    enum State { READY, RUNNING, BLOCKED}

  private final String name;
    private int timeLeft;
    private State state = State.READY;
    private int lastExecDate = 0;

    public Process(String name, int duration) {
        this.name = name;
        this.timeLeft = duration;
    }
```

```
    public void execQuantum(int date) {
        System.out.println(date + ":" + name);
        timeLeft--;
        lastExecDate = date;
    }
    public boolean isReady() {
        return state == State.READY;
    }
    public boolean isRunning() {
        return state == State.RUNNING;
    }
    public int getLastExecDate() {
        return lastExecDate;
    }
    public String getName() {
        return name;
    }
    public int getTimeLeft() {
        return timeLeft;
    }

    public void handleSignal(Signal signal) {
    switch (signal) {
      case BLOCK -> {
        if (state == State.RUNNING) {
          System.out.println(name + " blocked");
          state = State.BLOCKED;
        }
      }
      case UNBLOCK -> {
        if (state == State.BLOCKED) {
          System.out.println(name + " unblocked");
          state = State.READY;
        }
      }
      case SLEEP -> {
        if (state == State.RUNNING)
          state = State.READY;
      }
      case AWAKE -> {
        if (state == State.READY)
          state = State.RUNNING;
      }
    }
  }
}
```

3 Enum `Signal`

Cette énumération permet de représenter les différents types de signal qu'on peut envoyer à un processus.

```
public enum Signal { BLOCK, UNBLOCK, SLEEP, AWAKE}
```

4 Classe `ProcessManager`

Les instances de `ProcessManager` possède une liste de processus parmi lesquels il peut choisir le prochain à exécuter pendant un quantum de temps.

```
public class ProcessManager {
    private List<Process> processes = new ArrayList<>();
    private int date = 0;
    private Process running = null;

    public void add(Process process) {
        processes.add(process);
        System.out.println(process.getName() + " new");
    }

    private Process selectNext() { // Selection par round robin (tourniquet)
        Process candidate = null;
        int lastDate = Integer.MAX_VALUE;
        for(Process process : processes) {
            int lastExecDate = process.getLastExecDate();
            if(process.isReady() && lastExecDate < lastDate) {
                candidate = process;
                lastDate = lastExecDate;
            }
        }
        return candidate;
    }

    public void exec(int nbQuantum) {
        for(int i = 0; i < nbQuantum ; i++) {
            Process candidate = selectNext();
            switchRunningProcess(candidate);
            execRunningProcessQuantum();
            date++;
        }
    }

    private boolean canExecRunningProcess() {
        return running != null && running.isRunning();
    }
```

```
    private void switchRunningProcess(Process candidate) {
        if (candidate == null) return;
        if (running != null) running.handleSignal("sleep");
        candidate.handleSignal("awake");
        running = candidate;

    }
    private void execRunningProcessQuantum () {
        if (!canExecRunningProcess())
            return;
        running.execQuantum(date);
        if (running.getTimeLeft() <= 0) {
            processes.remove(running);
            running = null;
        }
    }
}
```

5 Classe `AppProcessManager`

Cette classe permet d'observer le comportement des classes précédentes en demandant l'exécution de la suite d'instructions suivante :

```
public class AppProcessManager {
    public static void main(String[] args) {
        Process p1 = new Process("p1", 5);
        Process p2 = new Process("p2", 7);
        Process p3 = new Process("p3", 4);
        ProcessManager processManager = new ProcessManager();
        processManager.add(p1); //affiche "p1 new"
        processManager.execQuantum(1); // affiche "0:p1"
        processManager.add(p2); // affiche "p2 new"
        processManager.execQuantum(2); // affiche "1:p2" puis "2:p1"
        processManager.add(p3); // affiche "p3 new"
        p1.handleSignal("block"); // affiche "p1 blocked"
        processManager.execQuantum(3); // affiche "3:p3", "4:p2", "5:p3"
        p1.handleSignal("unblock"); // affiche "p1 unblocked"
        processManager.execQuantum(10); // affiche "6:p1", "7:p2", "8:p3", "9:p1", etc
    }
}
```

6 Questions

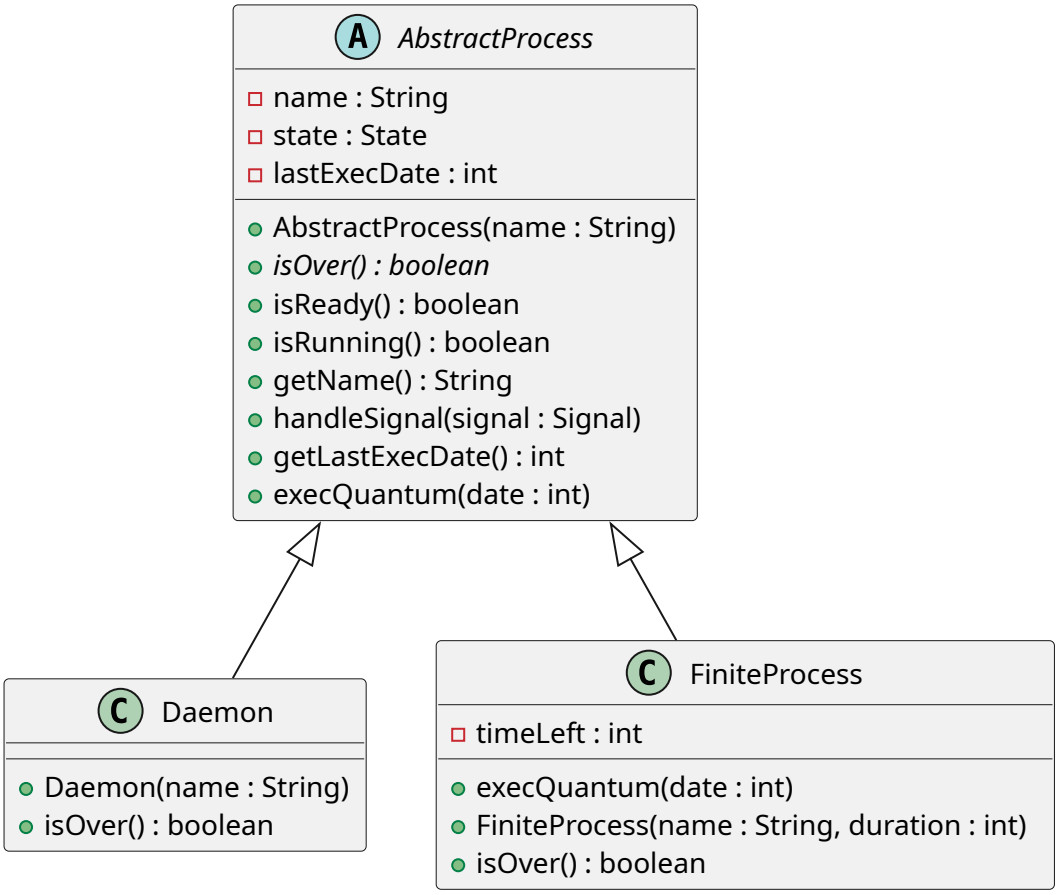
Question 1 : Il s'avère que d'autres types de processus peuvent exister (par exemple des processus avec différents niveaux de priorités). *Selon cette constatation précise, quel principe SOLID a été violé et pourquoi ?*

Question 2 : Par ailleurs, le gestionnaire de processus pourrait avoir une autre façon de sélectionner les prochains processus à exécuter et une autre façon d'exécuter un processus pendant un quantum de temps. *Selon cette constatation précise, quel principe SOLID a été violé et pourquoi ?*

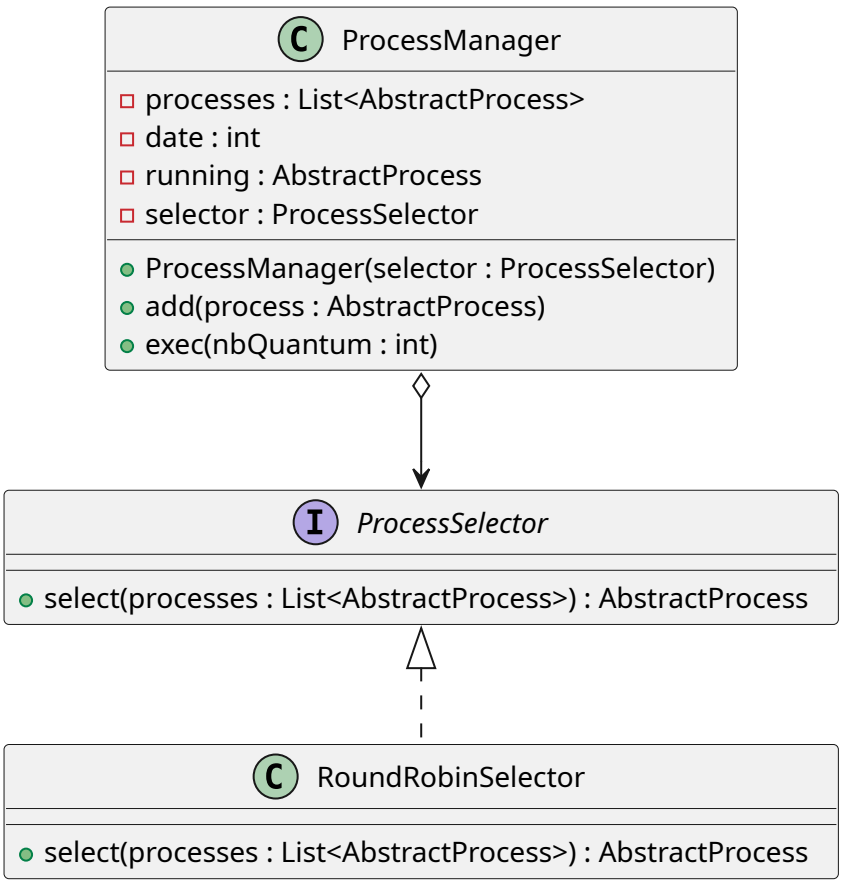
Question 3 : Si on veut proposer de nouveaux états pour les processus, il faut changer le contenu la classe `Process`. *Selon cette constatation précise, quel principe SOLID a été violé et pourquoi ?*

Question 4 : On souhaite rajouter des processus qui ne sont jamais censés s'arrêter (les processus "démons"). Or, la classe `Process` possède une méthode `int getTimeLeft()`. *Selon cette constatation précise, quel principe SOLID a été violé et pourquoi ?*

Question 5 : *Pour rajouter les processus démons de la question précédente, on vous demande de rajouter des classes `AbstractProcess`, `Daemon` et `FiniteProcess` de telle sorte à ce que `Daemon` et `FiniteProcess` étendent `AbstractProcess` (cf. diagramme de classes ci-après).* L'objectif est que `FiniteProcess` remplace la classe `Process` et que `Daemon` permette de créer des processus démons. Pour ceci `AbstractProcess` contiendra une méthode abstraite `boolean isOver()` qui indique si le processus est terminé.



Question 6 : *Réécrire la classe `ProcessManager` pour qu'elle délègue la sélection de processus à une instance d'une classe de l'interface `ProcessSelector` contenant l'unique méthode `Process selectNext(List<Process> processes)` (cf. le diagramme ci-dessous).*



Question 7 : Pour le moment, les processus définis ne peuvent être que dans trois états : *ready*, *running* ou *blocked*. Comme un processus peut habituellement être dans plus d'états (il existe neuf états de processus sous Unix), il faut prévoir l'ajout (ou le retrait) d'autres états dans des versions ultérieures du programme. Vous allez utiliser le patron de conception *State* pour définir les états possibles d'un processus ainsi que ses transitions. *Dessinez le diagramme de classe correspondant à cette modification.*

Question 8 : *Donnez les modifications à faire dans le code pour obtenir la réorganisation du code que vous avez décrite pour la question précédente.*

Question 9 : On souhaite rajouter de nouveaux processus avec différents niveaux de priorités allant de 1 à 10 et d'utiliser les priorités pour la sélection du processus à exécuter (par exemple en ne sélectionnant le prochain processus à exécuter uniquement parmi ceux de priorité maximale qui peuvent s'exécuter). *Décrivez à l'aide d'un diagramme de classes la manière de réorganiser le code pour prendre compte cette modification tout en gardant la possibilité d'avoir un système de processus sans priorité.*