Структури от Данни и Обектно-Ориентирано Програмиране
спец. Компютърни Науки
Упражнение № 7
07.04.2009г.

ТЕМА: Струкутура от данни ОПАШКА

Задача: Да се създаде клас на езика Java, който описва обекти – процеси, характеризиращи се със следните атрибути:
- идентификатор на процес – цяло положително число;
- име на процеса;
- приоритет на процеса – цяло число в интервала [0,15];
- състояние, в което се намира процеса.

Всеки един процес се намира в едно от следните състояния – *„спящ*", *„готов*" за изпълнение и *„работещ"*.
Във всеки момент има най-много един *„работещ*" процес и произволен брой *„готови"* и *„спящи"* процеси.

Класът **Process** да съдържа:
- конструктор, който създава нов процес с определен приоритет и име и уникален идентификатор и привежда процеса в състояние *„готов;*.

Класът **Condition**, който описва условие, трябва да съдържа:
- опашка, в която се поместват свързаните с условието процеси;
- метод **wait**(), който добавя процес в опашката като го привежда от състояние *„работещ"* в състояние *„спящ"*;
- метод **signal**(), който отстранява процеса от началото на опашката и сменя състоянието му от *„спящ"* на *„готов"*.

Класът **ProcessManager** , който създава необходимите структури за пазене на информацията за процесите и има следните методи:
- метод **create**(String name, int priority), който създава нов процес с указаното име и приоритет;
- метод **pick**() - привежда процес, който е в състояние *„готов"* и има най-висок приоритет в състояние *„работещ";*
- метод **preempt**(boolean flag) – привежда процес от състояние *„работещ"* в състояние *„готов"* , като намалява приоритета му с единица, ако това е указано.

```
import java.util.NoSuchElementException;

enum State{READY, SLEEPING, RUNNING}

/*
 * Class Queue
 * Objects of this class are used as ordinary or priority queues
 */

class Queue {

        private int capacity;
        private int size;
        private int head, tail;
        private ProcessManager.Process[] queue;

        Queue(int n) {
                capacity = n;
                size = 0;
                head = 0;
                tail = -1;
                queue = new ProcessManager.Process[n];
        }

        void enqueue(ProcessManager.Process p) {
                if(p == null)
                        throw new NullPointerException();
                if(size == capacity)
                        throw new IllegalStateException();
                tail = (tail+1) % capacity;
                queue[tail] = p;
                size++;
        }

        void enqueueSorted(ProcessManager.Process p) {
                if(p == null)
                        throw new NullPointerException();
                if(size == capacity)
                        throw new IllegalStateException();
                int i = 0;
                while(i<size &&  queue[(head + i) % capacity].compareTo(p)>= 0)
                        i++;
                int index = (head + i) % capacity;
                if(i == size) {// add to the end of queue
```

```
                queue[index] = p;
                tail++;
        }
        else if(i < size/2) { // insert in first half in place (head+i-1)
                head = (head - 1 + capacity) % capacity; // new head
                int j = head;
                while((j+1)%capacity != index){
                        queue[j] = queue[(j+1)%capacity];
                        j = (j+1)%capacity;
                }
                queue[j] = p;
        }
        else { // insert in second half
                tail = (tail + 1) % capacity;
                int j = tail;
                while(j != index) {
                        queue[j] = queue[(j-1+capacity)%capacity];
                        j = (j-1+capacity)%capacity;
                }
                queue[index] = p;
        }
        size++;
}

ProcessManager.Process dequeue() {
        if(size == 0)
                throw new NoSuchElementException();
        ProcessManager.Process p = queue[head];
        head =( head+1) % capacity;
        size--;
        return p;
}

boolean isEmpty() {
        return size == 0;
}

int size() {
        return size;
}

public String toString() {
        System.out.println(" !!! Size = " + size + "hEAD = " + head);
        StringBuffer stb = new StringBuffer();
```

```
            int index = head;
            for(int i=0; i<size; i++){
                stb.append(queue[index].toString() + "\n");
                index = (index + 1) % capacity;
            }
            return stb.toString();
        }
}

/*
 * Class Process Manager
 * One object of this class creates and controls a set of processes
 */

public class ProcessManager {

        /*
         * Inner class Process
         * Objects of the class represent processes running under control of PM
         */

        class Process implements Comparable<Process> {
            int id;
            int priority;
            String name;
            State state;


            private Process(String name, int pri) {
                id = ++sNumber;
                this.name = name;
                priority = pri;
                state = State.READY;

            }

            public int compareTo(Process other) {
                if(this.priority < other.priority)
                    return -1;
                else if(this.priority > other.priority)
                    return 1;
                else
                    return 0;
            }
```

```
        public int getIdent() {
                return id;
        }

        public int getPriority() {
                return priority;
        }

        public String getName() {
                return name;
        }

        public State getState() {
                return state;
        }

        private void setState(State s) {
                state = s;
        }

        private  void setPriority(int p) {
                priority = p;
        }

        public String toString() {
                StringBuffer stb = new StringBuffer();
                stb.append("Process " + name + ": \n");
                stb.append("Process id = " + id + " ; ");
                stb.append("Process priority = " + priority + " ; ");
                stb.append("Process state = " + state);
                return stb.toString();
        }

        public void run() {
                System.out.println("Process " + name + " running");
        }

        public void runToSleep() {
                System.out.println("Process " + name + " running, going to
wait");
                cv.waitCond();
        }
    }
```

```
/*
 * Objects of the class are places, where a process waits until some other
process issues a signal
 */

class Condition {
	private Queue condQueue;

	private Condition(int n) {
		condQueue = new Queue(n);
	}

	private void waitCond() {
		if(running != null) {
			running.setState(State.SLEEPING);
			condQueue.enqueue(running);
			running = null;
		}
	}

	private void signal() {
		if(!condQueue.isEmpty()) {
			Process p = condQueue.dequeue();
			p.setState(State.READY);
			readyQueue.enqueueSorted(p);
		}
	}
}

static int sNumber = 0;
Queue readyQueue;
Process running;
Condition cv;

public ProcessManager() {
	readyQueue = new Queue(100);
	running = null;
	cv = new Condition(100);
}

public Process create(String name, int pri) {
	Process p = new Process(name, pri);
	System.out.println(p.toString());
	readyQueue.enqueueSorted(p);
```

```
        return p;
}

private void pick() {
        //System.out.print("   Pick");
        if(running == null) {
                running = readyQueue.dequeue();
                running.setState(State.RUNNING);
        }
        //System.out.println(" Ready processes: " + readyQueue.size());
}

private void preempt(boolean flag) {
        //System.out.println("   Preempt");
        if(running != null) {
                if(flag)
                        running.setPriority(running.getPriority() - 1);
                readyQueue.enqueueSorted(running);
                running.setState(State.READY);
                running = null;
        }
}

public String toString() {
        StringBuffer stb = new StringBuffer();
        stb.append("Running proces: ");
        if(running != null)
                stb.append(running.toString() + "\n");
        else
                stb.append("null " + "\n");
        stb.append("Ready Processes: \n" + readyQueue.toString());
        return stb.toString();
}

public void run() {
        System.out.println();
        System.out.println("  Process Manager running");
        for(int i=0; i<10; i++) {
                pick();
                running.run();
                preempt(false);
        }
        System.out.println();
        pick();
```

```
                running.run();
                preempt(false);
                System.out.println();
                for(int i=0; i<5; i++) {
                        pick();
                        running.runToSleep();
                        //preempt(false);
                }
                System.out.println("    Number of ready processes = " +
readyQueue.size());
                for(int i=0; i<5; i++) {
                        cv.signal();
                }
                System.out.println("    Number of ready processes = " +
readyQueue.size());
                for(int i=0; i<3; i++) {
                        pick();
                        running.run();
                        preempt(false);
                }
                System.out.println();
                for(int i=0; i<10; i++) {
                        pick();
                        running.run();
                        preempt(true);
                }
        }
}
```