Rešenja prvog kolokvijuma iz Operativnih sistema 2
Oktobar 2017.

1. (10 poena)

```
class Scheduler {
public:
  Scheduler ();
  PCB* get ();
  void put (PCB*);
protected:
  inline PCB* remove (int pri);
private:
  static const int N;
  static const Time MaxWaitingTime;
  PCB* head[N];
  PCB* tail[N];
};

Scheduler::Scheduler () {
  for (int i=0; i<N; i++)
    head[i]=tail[i]=0;
}

void Scheduler::put (PCB* pcb) {
  if (pcb==0) return; // Exception!
  pcb->deadline = Time::now() + MaxWaitingTime;
  pcb->next = 0;
  int pri=pcb->priority;
  if (tail[pri]==0)
    tail[pri] = head[pri] = pcb;
  else
    tail[pri] = tail[pri]->next = pcb;
}

inline PCB* Scheduler::remove (int pri) {
   PCB* ret = head[pri];
   head[pri] = ret->next;
   if (head[pri]==0) tail[pri]=0;
   ret->next = 0;
   return ret;
}

PCB* Scheduler::get () {
  Time now = Time::now();
  for (int i=0; i<N; i++)
    if (head[i] && head[i]->deadline<=now)
       return remove(i);
  for (int i=0; i<N; i++)
    if (head[i])
       return remove(i);
  return 0;
}
```

2. (10 poena)

```
monitor ResourceAllocator;
export alloc, free;

  var
    count : integer;
    cond : condition;

procedure alloc;
begin
  while count>=N do cond.wait;
  count := count+1;
end;

procedure free;
begin
  count := count-1;
  cond.signal;
end;

begin
  count := 0;
end;
```

3. (10 poena)

```
public class ClientRequest {
    private final String ip;
    private final int port;
    private int requestedTickets;

    public ClientRequest(String ip, int port, int requestedTickets) {
        this.ip = ip;
        this.port = port;
        this.requestedTickets = requestedTickets;
    }

    public String getIp() {
        return ip;
    }

    public int getPort() {
        return port;
    }

    public int getRequestedTickets() {
        return requestedTickets;
    }

    public void removeRequestedTickets(int requestedTickets) {
        this.requestedTickets -= requestedTickets;
    }
}

public class RequestHandler extends Thread {
    private TicketCounter counter;
    private Service service;

    public RequestHandler(TicketCounter counter, Socket socket)
                                                                    throws IOException {
        this.counter = counter;
        service = new Service(socket);
    }

    public void run() {
        String ip = null;
        try {
            ip = service.receiveMessage();
            int port = Integer.parseInt(service.receiveMessage());
            String operation = service.receiveMessage();

            if ("reserve".equals(operation)) {
                int tickets = Integer.parseInt(service.receiveMessage());
```

```
                int ticketsAvailable = counter.reserveTickets(
                            new ClientRequest(new ClientRequest(ip, port, tickets));

                service.sendMessage(Integer.toString(ticketsAvailable));
            } else {
                int tickets = Integer.parseInt(service.receiveMessage());
                List<ClientRequest> ret =
                                    counter.returnTickets(tickets);
                giveTickets(ret);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

    }

    private void giveTickets(List<ClientRequest> ret) throws IOException {
        for (ClientRequest client : ret) {
            Service clientConn = new Service(
                            new Socket(client.getIp(), client.getPort()));
            clientConn.sendMessage("reserved");
            clientConn.sendMessage(
                            Integer.toString(client.getRequestedTickets()));
        }
    }
}

public class TicketCounter {
    private int availableTickets;

    private List<ClientRequest> waitingList =
                                    new ArrayList<ClientRequest>();

    public TicketCounter(int availableTickets) {
        this.availableTickets = availableTickets;
    }

    public synchronized int reserveTickets(ClientRequest client) {
        if (client.getRequestedTickets() <= availableTickets) {
            availableTickets -= client.getRequestedTickets();
            return client.getRequestedTickets();
        }

        client.removeRequestedTickets(availableTickets);
        waitingList.add(client);
        int ret = availableTickets;
        availableTickets = 0;
        return ret;
    }

    public synchronized List<ClientRequest> returnTickets(int tickets) {
        availableTickets +=tickets;

        List<ClientRequest> ret = new ArrayList<ClientRequest>();
        while (availableTickets > 0 && waitingList.size() > 0) {
            ClientRequest first = waitingList.get(0);
            ClientRequest send;
            if (first.getRequestedTickets() > availableTickets) {
                send = new ClientRequest(
                                        first.getClient(), availableTickets);
                first.removeRequestedTickets(availableTickets);
                availableTickets = 0;
            } else {
                send = new ClientRequest(
                            first.getClient(), first.getRequestedTickets());
                availableTickets -= first.getRequestedTickets();
                waitingList.remove(0);
            }
            ret.add(send);
        }

        return ret;
    }

    public synchronized List<ClientRequest> getAllWaitingClients() {
        return Collections.unmodifiableList(waitingList);
    }
}

public class Server extends Thread {
    private boolean work;
    private int port;
```

```
    private TicketCounter counter;

    public Server(int port) {
        this.port = port;
    }

    public void run() {
        ServerSocket serverSocket = null;
        List<Thread> threads = new ArrayList<Thread>();
        try {
            serverSocket = new ServerSocket(port);

            while(work) {
                Socket clientSocket = null;
                try {
                    clientSocket = serverSocket.accept();
                    RequestHandler handler =
                              new RequestHandler(counter, clientSocket);
                    threads.add(handler);
                    handler.start();

                }
                catch (IOException e) {
                    e.printStackTrace();
                }
            }

            waitThreads(threads);

            cleanUp();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void startServer(int n) {
        work = true;
        init(n);
        start();
    }

    private void init(int n) {
        counter = new TicketCounter(n);
    }

    public synchronized void stopServer() {
        work = false;
        interrupt();
    }

    private void cleanUp() throws IOException {
        for (ClientRequest client : counter.getAllWaitingClients()) {
            Service clientConn = new Service(
                  new Socket(client.getIp(), client.getPort()));
            clientConn.sendMessage("end");
        }
    }

    private void waitThreads(List<Thread> threads) {
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
            }
        }
    }
}
```

Klasa Service je klasa data na vežbama.