Assignment Answer

Code:

```
const cluster = require('cluster');
const http = require('http');
const os = require('os');

if (cluster.isMaster) {
   // Master process
   const numCPUs = os.cpus().length;

   // Fork workers
   for (let i = 0; i < 2; i++) {  // Creating two replicas
      cluster.fork();
   }

   cluster.on('exit', (worker, code, signal) => {
      console.log(`Worker ${worker.process.pid} died. Starting a new
one.`);
      cluster.fork();
   });
} else {
   // Worker process
   const express = require('express');
   const rateLimit = require('express-rate-limit');
   const app = express();

   // Middleware to handle rate limiting
   const taskLimiter = rateLimit({
      windowMs: 60 * 1000, // 1 minute window
      max: 20, // Limit each user to 20 requests per windowMs
      keyGenerator: (req) => req.query.userId, // Identify user by userId
      handler: (req, res) => res.status(429).json({ message: 'Too many
requests' })
   });

   app.use(express.json());
   app.use(taskLimiter);

   const taskQueue = {};
```

```
    // Function to process the task
    const processTask = (userId) => {
        console.log(`Processing task for user ${userId} at ${new
Date().toISOString()}`);
        // Simulate task processing
        setTimeout(() => {
            taskQueue[userId].shift();
            if (taskQueue[userId].length > 0) {
                processTask(userId); // Process next task
            }
        }, 1000);
    };

    // Task route
    app.post('/task', (req, res) => {
        const userId = req.query.userId;
        if (!userId) {
            return res.status(400).json({ message: 'User ID is required' });
        }

        // Initialize user queue if not exists
        if (!taskQueue[userId]) {
            taskQueue[userId] = [];
        }

        taskQueue[userId].push(req.body.task);

        if (taskQueue[userId].length === 1) {
            processTask(userId); // Start processing if not already processing
        }

        res.status(202).json({ message: 'Task accepted and queued' });
    });

    http.createServer(app).listen(3000, () => {
        console.log(`Worker ${process.pid} started on port 3000`);
    });
}
```

Explanation

Cluster Setup: We use Node.js's cluster module to fork two worker processes that will handle incoming HTTP requests. If a worker dies, it is automatically restarted.

Rate Limiting: We use the express-rate-limit middleware to enforce rate limits. We define a rate limit of 1 request per second and 20 requests per minute per user. The rate limit is enforced based on the userId passed in the request query.

Task Queue: We maintain a task queue for each user. When a new task is submitted, it is added to the queue. If no other task is currently being processed for that user, the task is processed immediately; otherwise, it is queued.

Task Processing: The processTask function simulates task processing. After completing a task, it checks if there are more tasks in the queue and processes the next one if available.

Considerations:

Persistence: In a production scenario, the task queue and rate limiting would likely be backed by a persistent storage solution like Redis to handle server restarts and distributed systems.

Scalability: The use of Redis or another distributed rate limiter would allow scaling across multiple servers beyond just clustering on a single machine.