UNIVERSITY OF MALTA

FACULTY OF INFORMATION & COMMUNICATION TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

CPS2008 Assignment : `rex`

Keith Bugeja, Joshua Ellul, Alessio Magro, Kevin Vella

Instructions:

1. This is an **individual** assignment and carries **50**% of the final CPS2008 grade.
2. The report and all related files (including code) must be uploaded to the VLE by the indicated deadline. Before uploading, archive all files into a compressed format such as ZIP. It is your responsibility to ensure that the uploaded file and all contents are valid.
3. Everything you submit must be your original work. All source code will be run through plagiarism detection software. Please read carefully the plagiarism guidelines on the ICT website.
4. Reports (and code) that are difficult to follow due to poor writing-style, organisation or presentation will be penalised.
5. Always test function return values for errors; report errors to the standard error stream.
6. The Unix **man** command is your best friend, **Google search** your second best.
7. Each individual might be asked to present their implementation, at which time the program will be executed and the design explained. The presentation outcome may affect the final marking.

Overview

The goal of this assignment is to implement **`rex`**, a command-line tool for **r**emote job **ex**ecution on Linux-based operating systems. The command-line usage for the tool is:

```
rex command arg0 arg1 ...
```

where `command` is used to interact with the `rex` job scheduler, to submit, launch or manage jobs amongst others. The `rex` environment, including the scheduler, is governed by a Linux-based service (**`rexd`**); the service should run on each node in the pool of network machines that would provide remote execution. The implementation of `rexd` is also part of your assignment, as is any interprocess communication between `rex` and `rexd`, and between instances of the latter.

There are three sets of tasks to this assignment: (i) *scheduling and execution*, (ii) *job control* and (iii) *environment*; each set finds a corresponding section in this document where further implementation details are given, together with example scenarios and expected output.

Deliverables

Upload your source code (C files together with any accompanying headers), including any unit tests and additional utilities, through VLE by the specified deadline. Include makefiles with your submission which can compile your system. Make sure that your code is properly commented and easily readable. Be careful with naming conventions and visual formatting. Every system call output should be validated for errors and appropriate error messages should be reported. Include a report describing:

- the design of your system, including any structure, behaviour and interaction diagrams which might help;
- any implemented mechanisms which you think deserve special mention;
- your approach to testing the system, listing test cases and results;
- a list of bugs and issues of your implementation.

Do **not** include any code listing in your report; only snippets are allowed.

Background

A job is a unit of work or execution which can be started interactively, such as from the command-line, or scheduled for non-interactive execution (batch) by a job scheduler. In this assignment we concretely identify a job with a single process.

Interactive Jobs

An interactive job in `rex` is characterised by its treatment of input and output streams; in particular, an interactive job is launched instantaneously and performs redirection of input and output between the local machine (where `rex` is executed) and the remote machine (where `rexd` executes the job in question). It is safe to assume interactive jobs that are terminal-based (no graphical user interfaces are employed).

Batch Jobs

Batch jobs are assumed to be non-interactive; they perform no network redirection of streams but rather redirect their output to a local file (local with respect to their execution target), for later retrieval and analysis. Multiple jobs may be scheduled for execution on a stipulated date and time.

Job Description Format

Your implementation should follow a simple nomenclature for specifying a job, where the command – or path to an executable program or script – is prefixed by the network name or IP address of the target machine, with an infix colon (`:`) acting as a separator: `host:command arg0 arg1 ...`. The path may be specified either in relative or absolute terms; consider that the resource should exist in the filesystem namespace of the target machine. Below are some example job descriptions:

`localhost:/bin/ls`
: Denotes execution of `/bin/ls` on `localhost`.

`127.0.0.1:/bin/ls`
: Denotes execution of `/bin/ls` on `localhost`.

`jojo:/bin/cp /home/joestar/mask.txt /home/brando`
: Denotes copying the file `mask.txt` from `/home/joestar` on network node `jojo` to directory `/home/brando` on the same machine. Note that the actual copy is effected by `jojo`.

`natsu:g++ -o collection fairy.cpp rave.cpp`
: Denotes compiling files `fairy.cpp` and `rave.cpp` using the GNU C++ compiler into a binary executable `collection`; the files are assumed to be in the current working directory of `rexd` on host `natsu`, where the actual compilation takes place. Note that the arguments here are examples of relative paths.

`zolo:ls -la`
: Denotes performing a directory listing for the current working directory of `rexd` on `zolo`. It is assumed that `ls` is in `zolo`'s system path.

Note: these job descriptions assume the following entries in the local machine's `\etc\hosts` file.

```
127.0.0.1       localhost
192.168.0.32    zolo
192.168.0.66    jojo
144.1.1.69      natsu
```

Scheduling and Execution

Your implementation of `rex` should discriminate between interactive and batch jobs, and provide distinct commands for handling each case. Interactive jobs are meant to start immediately and provide real-time feedback to the user. The example below shows the expected output from executing the list utility (`ls`) as an interactive job on a network host identified as `kaermorhen` using the `run` command; the output is redirected to the calling host (`cintra`):

```
cintra@cirilla:~$ rex run "kaermorhen:ls -la"
total 0
drwxr-xr-x   4 geralt  witchers  128 Mar  8 16:01 .
drwxr-xr-x+ 31 geralt  witchers  992 Mar 20 09:05 ..
drwxr-xr-x   3 geralt  witchers   96 Mar  8 16:02 quests
drwxr-xr-x   5 geralt  witchers  160 Feb 13 18:29 contracts
cintra@cirilla:~$
```

It should be possible to schedule non-interactive jobs for execution at a specified time in the future using the `submit` command; date and time may be passed as additional arguments on the command-line. In the example below, a non-interactive job is scheduled that attempts to delete the root folder in the filesystem namespace on the host identified as `nilfgard`:

```
cintra@cirilla:~$ rex submit "nilfgard:rm -rf /" 01/04/2019 21:00:00
Job "nilfgard:rm -rf /" submitted; unique ID is 3.
host@user:~$
```

If the user desires the non-interactive job to launch immediately, specifying `now` or omitting the arguments for date and time altogether schedules the job using the caller's current system time:

```
cintra@cirilla:~$ rex submit "nilfgard:rm -rf /" now
Job "nilfgard:rm -rf /" submitted; unique ID is 4.
host@user:~$
```

Whether interactive or not, each job is assigned an identifier (ID), typically an integer, that uniquely identifies it within the system at a given point in time. Think of a process identifier (*pid*) that is unique across the network of hosts running `rexd`. Furthermore, the system creates an output log for each job; for interactive processes, this will contain information about the job such as its ID, and any `rex`-related errors. For batch processes, this output log will also be the recipient of the job's standard output and error streams. The output log naming convention should make it easy to retrieve the file once the job ID is known (e.g. `jobID.output`).

Task Set I

(a) Implement the `run` command, to launch interactive jobs on any host running the `rexd` service. The usage is shown below:

```
rex run job-description
```

(i) The job's standard output and standard error (on the remote host) should be redirected to the output and error streams of the `rex` launcher on the local host.

(ii) The job's standard input should be wired to the input stream of the `rex` launcher. This allows for the behaviour shown below, where the `sort` command is executed remotely, but the input and output are transacted between the local and remote host.

```
$ rex run "remote-host:sort"
$ Sengoku <Enter>
$ Edo <Enter>
$ Meiji <Ctrl+d>
$ Edo
$ Meiji
$ Sengoku
$
```

(b) Implement the `submit` command to schedule non-interactive jobs for execution on any host running the `rexd` service. The usage is:

```
rex submit job-description [(date time) | now]
```

(i) Create a log file for each individual job and store its output and error streams.

(ii) Non-interactive jobs submitted with the `now` option are scheduled to execute at the current date and time on the local host; however, `rexd` actually executes jobs when their scheduled time matches the date and time of the remote host, the one, the particular `rexd` is running on.

(c) Implement the `rexd` service, which runs on each and every network host where remote execution is allowed.

(i) `rex` and `rexd` should communicate over the network via TCP/IP.

(ii) `rexd` should support concurrency and be able to launch multiple simultaneous jobs on a single host.

(iii) `rexd` should look for the program binary in the system path, unless specified in absolute terms (see also Environment Section).

(iv) `rexd` should assign a unique identifier to each newly created job.

(v) `rexd` should maintain a single consistent job registry across all running instances; to avoid consensus problems, all `rexd` instances should acknowledge a leader, elected by the user on start-up. All amendments to the job registry would then go through the leader.

Hint: If `rexd` is implemented as a daemon, remember to create a new session and make the forked process leader of the new process group, detaching it from the controlling terminal.

Job Control

It should be possible to terminate a running job or remove a scheduled job from the execution queue before it starts running. This may be accomplished through the `kill` command, which takes two arguments, the job ID and the termination mode. There are three termination modes:

- `soft` - try to terminate job using `SIGTERM`;
- `hard` - terminate job using `SIGKILL`; and
- `nice` - first try `soft`; after grace period, if job still persists, use `hard`. This mode takes an optional third parameter, specifying the length of the grace period in seconds.

```
cintra@cirilla:~$ rex submit "novigrad:ggprogram" now
Job "novigrad:ggprogram" submitted; unique ID is 5.
host@user:~$ rex kill 5 nice
Job "novigrad:ggprogram" with ID 5 has terminated.
cintra@cirilla:~$ rex submit "novigrad:imatroll" now
Job "novigrad:imatroll" submitted; unique ID is 6.
host@user:~$ rex kill 6 nice 2
Job "novigrad:imatroll" with ID 6 was forcefully terminated.
```

Monitoring

Your implementation should allow for monitoring of `rex` jobs and display their status and other important parameters; this should be accomplished via the `status` command. An example of the expected output is shown below:

```
host@user:~$ rex status
Job     Host        Command  Type      Status        Date       Time
  1 novigrad    crusty.d -a     B  TERMINATED  20/02/2019   01:01:01
  2   cintra            top     I     RUNNING  21/03/2019   00:13:37
  3 novigrad  ./myscript.sh     B   SCHEDULED  01/05/2019   06:66:66
```

Task Set II

(a) Implement the `kill` command, to terminate any running jobs or stop currently scheduled jobs from running. The usage for this command is:

```
rex kill job-ID mode [grace-period]
```

where `mode` specifies the desired termination approach - this is either of `soft`, `hard` or `nice`; `grace-period` is only required when `nice` is specified and is expressed in seconds. If omitted, a grace period of one second is assumed.

(b) Implement the `status` command, to list the current job registry. The usage is as follows:

```
rex status
```

without any additional arguments. The status listing should display the following fields:

job ID - a network-wide identifier that uniquely identifies a job

host - the host name or IP address of the network node where the particular job is scheduled, running or has executed

command - the job description string excluding the host name; i.e. job description substring right of the (:) colon

type - job execution modality; **I** is for interactive jobs and **B** is for batch or non-interactive jobs

status - current job execution status which can either be SCHEDULED, RUNNING or TERMINATED

date - date when job is/was scheduled to run

time - time when job is/was scheduled to run

Environment

The rex launcher can do very little on its own; in fact, as previously outlined, you should also implement rexd, a Unix service (either as a normal program or a daemon), and run it on all eligible target machines in the network. The service should be responsible for executing commands issued through the rex command-line tool. In cases where a job refers to a command binary that is not available on a particular target, the rexd service should work with the command-line tool to provide file copy functionality, such that program files and their data may be copied to a target system, prior to execution.

Task Set III

(a) Your implementation should provide a copy command in rex with a syntax similar to the code Unix cp utility. The usage for the copy command should be as follows:

```
rex copy source-file target-file
```

or

```
rex copy source-file ... target-directory
```

where remote files are qualified in a similar way to jobs, with a prefix representing the host machine.

```
host@user:~$ rex copy "novigrad:~/my_program/my_file" my_file_local
Copying files...
host@user:~$ rex copy my_local_directory "novigrad:~/my_remote_env/"
Copying files...
host@user:~$
```

(b) Implement the chdir command to modify the current working directory for a given rexd service. The usage for the chdir command is as follows:

```
rex chdir remote-host:remote_directory
```

Considerations

Make sure you spend enough time designing your system before starting to code. Think of several scenarios which your tool might encounter and design appropriate handling mechanisms. Aim for a system that is free of race conditions and deadlocks, and is still considerably efficient. The following is a non-exhaustive list of considerations you might want to ponder about:

- How is shared state represented? What data structures are required?
- Are data structure instances unique to each service or shared across multiple ones? Which approach would be more efficient? What overheads are required to share data structures?
- What happens when two services attempt to concurrently modify the same data structure?
- How are operations from command-line tool sent to the services?
- Are services thread-safe?
- Is asynchronous functionality required?
- Are requests from the command-line tool queued and serialised, or is some form of parallelism implemented by the services? What are the benefits of the latter option, if any?
- Is some type of conflict resolution required?
- What happens when the connection between a service and the command-line tool is lost?
- Are proper byte-ordering mechanisms implemented?
- What happens if a service crashes, or the command-line tool disconnects?

---

end of paper