Scheduled Tasks and Log Management

COSC301 Laboratory Manual

Today we learn about how to schedule programs to run in the background at a certain time. We also learn about logging on a UNIX-like system, how to manage the system logs, and cover a couple of very useful logging related tools for rotating and reporting on the various log entries. We'll also learn about simple regular expressions, which gives us a powerful tool for matching text against a pattern.

Note

In this lab, we'll just be doing all our work on your Server1 unless specified otherwise.

1. Cron

Being able to run jobs at scheduled time, and in the background, is a great assistance to a system administrator (and to the user as well). Cron is able to assist you to run jobs at scheduled times. Here is just a sample of the types of things you might use cron to do:

- Fetch any mail from your ISP every five minutes. You could then automatically have it sorted and processed.
- Download package updates at night, when traffic costs may be lower, or the network less busy.
- Check the system periodically, and page the administrator when something is wrong.
- Initiate system maintainance tasks, such as rotating logs or performing backups.
- Run commands once only, at a certain time (using **at**).
- Run commands when the system is not heavily loaded (using **batch**)

To use Cron, you need to know how to use **crontab**, which is the command for editing your own Cron Table (crontab). You also need to know the format of a crontab file. See crontab(1) for information about the command, and crontab(5) for more information on the format of a crontab.

For most system administration tasks, we don't use personal crontabs. Instead, we use some standard directories into which we can place jobs. These directories are `/etc/cron.d/` , `cron.hourly/`, `cron.daily/`, `cron.weekly/` and `cron.monthly/`. With the exception of `cron.d/`, these contain executable scripts, which are run every day, week and month. `cron.d/` stores normal crontab entries, including a time and specification. This modularity helps immensely with package management, due to its drop-in nature. You do not use **crontab** to edit these files, just a regular text-editor.

There is also the file `/etc/crontab` which is the crontab for the system (ie. the root user). The main items it contains are entries to start all the daily, weekly and monthly jobs. You do not generally need to edit `/etc/crontab`; use the directories instead. Shown below is what you would see in `/etc/crontab`. Note that it differs from a normal users crontab, in that the `user` field does not appear in a users crontab.

```
# /etc/crontab: system-wide crontab
# Unlike any other crontab you don't have to run the `crontab'
# command to install the new version when you edit this file
# and files in /etc/cron.d. These files also have username fields,
# that none of the other crontabs do.

SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

# m h dom mon dow user command
Note, we have edited this for clarity.
17 * * * * root run-parts --report /etc/cron.hourly
25 6 * * * root run-parts --report /etc/cron.daily
47 6 * * 7 root run-parts --report /etc/cron.weekly
52 6 1 * * root run-parts --report /etc/cron.monthly
```

run-parts is a helper program that runs all the executable scripts in a directory, such as `cron.daily/`.

The scripts in `cron.when/` etc. are run with the privileges of the root user, and run one after another (sequentially, based on its filename). The files in `cron.d/`, which can be seen of as fragments of `/etc/crontab`, have a user field in them, commonly root, but can easily be changed.

Each user can have a crontab file under `/var/spool/cron/crontabs/`[1]. This directory will most likely be empty on your system, because users don't most often will not have created any crontab entries. A user modifies or creates a crontab using the **crontab -e** command. This will start the editor specified by the `VISUAL` or `EDITOR` environment variable, or the powerful-but-not-very-newbie-friendly **vi**[2]. The format for the file is fairly simple, but easy to forget, so you might like to put a comment at the start of the file that reminds you of the format.

Here is an example of the crontab you would be editing using **crontab -e**. Remember, as this is a users crontab, it does not allow you to specify a user field. Consider carefully the difference between each time specification. Note that you are not required to make any modifications at this stage.

```

* 8 * * *     command  Will run every minute from 8:00am to 8:59am
0 8 * * *     command  Will run at 8:00am
0 */8 * * *   command  Will run every 8th hour
* */8 * * *   command  Will run every minute of every 8th hour
```

All entries can be thought of as being checked every minute. The * is a wildcard which always matches. `*/2` means every 2 hours (when in the hourly column). Likewise, `9,17,21` matches 9am, 5pm, and 9pm in 24-hour format. If you want something to run once in a particular hour(s), then the minute field must be absolutely specified. Consider the difference beween the first two entries above.

1.1. Self-assessment

1. When will the following crontab entry run?

[1]This can be disabled or allowed only to particular users.
[2]**vi** is a common Unix default, but other systems can differ. Debian-based systems in particular, use its “alternatives” subsystem to run whatever is configured as **sensible-editor**, which is **nano** by default.

```
* * 13 jan * command
```

2. When will the following crontab entry run? Notice that both day of week and day of month are specified; check crontab(5) for how this edge-case is treated on your system.

```
30 4 1,15 * 5 command
```

3. Imagine you have a virtualisation host with a lot of identically configured virtual machines (guests). You notice that you get massive load spikes on a regular basis, and upon investigation during one of these load spikes, you determine that they are caused by cron jobs all starting at the same time (and thus every machine simultaneously wants to wake up and run something, causing the host to become very busy, even though each virtual machine has very little to do). How could you improve this situation? (You do not need to implement anything for this question, just brainstorm some solutions.)

2. Syslog

The venerable BSD Syslog remote logging protocol is implemented in Ubuntu using the **rsyslogd** program. Other systems use similar pieces of software, but conceptually they are quite similar. The remote aspects of Syslog have a number of limitations and problems related to reliability and security, and so there have been improvements made in a series of RFC documents to come up with the Reliability Enhanced Logging Protocol (RELP), which is what sets **rsyslogd** apart from the rest. However, few devices support that yet, and most of the time all we need is the usual Syslog protocol. Many Linux distributions today use the newer **rsyslogd** over its pre-decessor **sysklogd**[3]; Ubuntu is no exception here.

rsyslogd sits in the background, waiting for any program to give it a log entry. When it does, that program will specify a facility and a priority.

Examples of facilities are `auth`, `cron`, `daemon`, `kern`, and `mail`. You can find more detail in the syslog.conf(5) manual page. In addition to the standard keywords, there are also `local0` through `local7`, which you can use for any local purpose you desire. For example, you could configure all your active network elements (routers, switches, access-points etc.) to send to `local0`, phone systems to log to `local1`, building management services (security, air conditioning etc.) to `local2`, a SQL database might be configured to send to `local3` and your own software you've developed might use `local4`… you can see that it wouldn't be too hard to run out of facilities if you really embrace remote logging, but most of the time you only accept local log messages, and if you're using that many log sources, you should probably have multiple log servers.

A priority says how important the log entry is. Priorities range from `debug` (the lowest), through `warning` and `err`, to `emerg` (system about to crash). There are 8 different priorities you can assign, though most software should not get more severe than `err`.

rsyslogd is configured with the rsyslog.conf(5) configuration file. However, unlike other versions of **syslogd**, **rsyslogd** uses this file for overall settings, but the actual message routing elements are split out into indivitual files dropped into the directory `/etc/rsyslog.d/` ; have a look at the files that are there.

Whenever you change something in here, or manage any files in `/var/log/`, you should reload **syslogd** (**systemctl restart rsyslog.service** or **service rsyslog restart**). You may be interested to find out the subtle differences between **systemctl** and **service** yourself.

[3]**sysklogd** is comprised of two parts, a portable userland Syslog server called **syslogd** and a Linux-specific kernel logging proxy called **klogd**.

The format of each line in the file is `facility.priority target`. The `facility.priority` part is often called the SELECTOR, and the `target` is often called the ACTION. You will see these terms used in the documentation for **rsyslogd**.

You can specify multiple facilities in a line by separating them with a comma. For priority, you can (these are for reference, don't actually make any edits) say things like the following:

warn
: warn priority and up.

=warn
: Only warn priority.

!=warn
: Anything but warn priority.

!warn
: Only priorities above warn (not including warn).

none
: Used for excluding facilities.

For the target, you can specify either a log file, a remote host, a usercode(s) to who the log entry will be written on their terminal, or a terminal device, such as the system console.

/var/log/messages
: A fully qualified path to a log file. If a `-` prefixes the log file, the log will be written synchronously, meaning the changes will be written to disk immediately[4].

@valhalla
: Send the log message to a machine called valhalla. **rsyslogd** can receive log messages over the network, but like any Syslog implementation, needs to be configured to allow this first.

theauthor,root
: Write the message to these users terminal. You can use * to mean everyone logged in (you can use the **wall** command to send messages like this in an ad-hoc manner).

/dev/console
: A terminal device. The message will be written to the appropriate terminal.

^command
: Starts a program and sends the log message to it. Because the program is started each time, the program should be very fast to run and should not run too frequently.

|fifo
: Sends the log message to a program that is already running. The message is sent using a form of Inter-Process Communication (IPC) called a "named fifo" (First-In First-Out) which can be found in the filesystem. Because the program is already running, it is not bad if a lot of messages are sent this way, or if the program takes a while to run or start up.

Read the first few paragraphs of the "SELECTORS" and "ACTIONS" sections of the rsyslog.conf(5) manual page. We won't be altering any of the other rules today, but just looking

[4]Actually, guaranteeing this is very difficult on modern hardware. Additionally, a lot of synchronous writes can slow the system, so use it only when something awful might be about to happen.

at the defaults should give you a reasonable idea of how it works. The man page has an examples section, should you be interested.

Logs are generally kept in `/var/log/`. As a brief guide, the important ones are as follows, the most important to keep an eye on when debugging is `syslog`. How the logging is organised is particular to the distribution in use.

messages
: By default, everything, if not put anywhere else, will fall into here. Typically uninteresting.

syslog
: System messages, such as from kernel and daemons. Generally the most interesting log file.

secure
: Security related messages regarding authentication go in here.

utmp, wtmp, faillog
: Binary database of current logins, login history and login failures. Used with programs such as **who**, **last**, **lastlog** and **faillog**.

Some services often have their own logging, such as web servers and proxy caches, as the volume of data they log is typically quite high, and more flexibility may be desirable.

2.1. Self-assessment

1. In this question, we are going to route a copy of all of our non-sensitive logs to a spare virtual console, which can make it easy to keep an eye on what the system is doing.

   We are going to use the virtual console tty2[5].

   A virtual console is a type of terminal, much like Gnome Terminal or **xterm** is a type of terminal. Terminals are often referred to as “TTY” devices[6]. Linux systems often have a number of virtual consoles configured, and you can switch between them using **Alt+F1** through **Alt+F6**, depending on how many have been configured (note that on a Mac keyboard you need to use **Alt+fn+F6**). You can also use **Alt+←** and **Alt+→** to go to the next and previous virtual console respectively.

   Comment out the `PrivDrop` lines in `/etc/rsyslog.conf`,[7] then add the following to a new file `/etc/rsyslog.d/99-tty2.conf`:

```
*.notice;auth,authpriv.none    /dev/tty2
```

Exercise

As an exercise restart Rsyslogd using **sudo service rsyslog restart**. Use the **logger** program to submit a log message, and have a look on the virtual console tty2, where we send a copy of our log messages. Make sure you see the log message displayed on virtual console tty2.

[5]You may find that you open the volumn controls instead of switching to the virtual terminal. There are two ways to fix this, either use the **Fn** button (beside the **home** button above the arrow keys), or change the keyboard settings so that you "Use F1, F2, etc. keys as standard function keys".
[6]The term “TTY” comes from TeleTYpe terminal; a historical artifact.
[7]This leaves us open to security isues, but we shant worry about that now.

You can use the **logger** command to help your understanding of how different facilities and priorities are routed by the Syslog server. To do this, simply submit a unique log message, and then use **grep** to see which file(s) the log message appears in:

```
$ logger -p daemon.warning "syslog test N"
# grep -r "syslog test N" /var/log/
```

2. **[Optional challenge]** How might you send a log message to a cell-phone as an SMS message? You don't need to set this up for this lab. Assume you can send a SMS message using the Gnokii software, which interfaces well with Nokia phones: even very cheap phones can send text messages this way, so long as you have the right cable.

```
$ echo "Hello, World!" | gnokii --sendsms phone-number
```

Assuming you had **gnokii** configured to use a particular phone, you *could* hook this into **rsyslog** using an entry such as:

```
SELECTOR  ^gnokii --sendsms phone-number
```

And the command will be run *every* time a log message matches the selector. However, while this will work, there is a possibly very expensive and very annoying problem with this. What is this problem? How might you deal with this problem?

As an alternative mechanism that doesn't require any physical setup, sms gateways can be accessible using an API over the internet. Why might this mechanism be indesirable?

3. Rotating Logs

It's a fact of life, logs grow. Just like your lawn, the logs need processed every so often to keep them manageable, and to help you find any snakes in the grass. What you do with your old logs will reflect on your local policies, and possibly even laws eg. logs may need to be kept for several years, or on the other hand, logs perhaps may not be kept for longer than a few years due to privacy laws; you should seek advice from your lawyer or industry.

In Linux distributions, logs are usually rotated using a tool called **logrotate**. Have a look at the default Ubuntu configuration of **logrotate** by looking in the file `/etc/logrotate.conf`. Note that it includes all the files in `/etc/logrotate.d/`, to enable package-maintainers to easily install rules for having **logrotate** rotate its logs, simply by dropping a file into the `logrotate.d/` directory. Here is what the `/etc/logrotate.conf` file looks like. The file has been edited to save space.

```
Specifying global defaults
weekly              Rotate logs weekly
rotate 4            Keep 4 weeks worth of backlogs
create              Create new (empty) log files after rotating old ones
#compress           Uncomment to compress rotated files
A directory so packages can install log rotation policies.
include /etc/logrotate.d/
/var/log/wtmp {     Stores login history.
    missingok       Don't complain if wtmp file is missing.
    monthly
    create 0664 root utmp   perm user group
    rotate 1
}
```

The above is a simple example; we could do a lot more. We've selected the condition to rotate based on time (`weekly` and `monthly`), but we could also choose to rotate based on file size. You can also specify commands to be run before and after rotating the log file. Here is a fictitious entry which shows some of the other useful entries. This could be a file in `/etc/logrotate.d/` Remember that the following is an *example*, so I don't expect you to input it.

```
You can specify multiple files
/var/log/foo/access /var/log/foo/errors {
  size=100k           Rotate when it reaches a certain size
  sharedscripts       Run post and prerotate only once for all files in this set
  postrotate          You can run a list of commands after rotation
    killall -HUP food food would be the daemon for the foo service
  endscript           End the list using endscript
}
```

logrotate gets run by **cron**, using `/etc/cron.daily/logrotate`. The scripts in `cron.daily` get run early each morning, typically about 6am. You can alter this by editing `/etc/crontab`.

You can force **logrotate** to rotate (ie. ignoring the selection tags such as `weekly` or `size`) the files by using the **-f** argument to **logrotate**. Have a look, using **ls -l**, in the `/var/log` directory, and then run the command **logrotate -f /etc/logrotate.conf**. Have a look to see what changed. Repeat a few times, and describe what happens.

Although Ubuntu doesn't compress them by default, **logrotate** is able to compress logs using **gzip**, and thus get an extension of `.gz`. Compressed logs can be viewed using the **zless** or **zcat** program, and grepped using **zgrep** (we'll cover **grep** later in the lab.

3.1. Self-assessment

1. **On paper**, write logrotate entry that rotates the file `/var/log/auth.log` on a weekly basis. This file is where you would find any log messages relating to failed login attempts etc. Any further rotation options are up to you. Rationalise why you chose particular options, particularly the number of logs to keep.

2. This question is designed to help you understand the concept of log rotation. Before we continue, we must first disable (temporarily comment-out) the `notifempty` directive in `/etc/logrotate.d/rsyslog`. This directive tells **rsyslogd** not to bother rotating a file if that file is empty. Normally, that is useful, but it does get in the way of understanding the concept we are trying to illustrate.

   After disabling `notifempty` and restarting **rsyslogd**, **cd** into `/var/log`. From there, run the following sequence of commands three or so times, until you see a pattern emerge. If you don't see a pattern after doing this, go on with the next question, and start democall to ask a demonstrator to explain.

```
$ ls auth.log* | while read file; do
> echo -en "$file\t"
> (zcat "$file" 2>/dev/null || cat "$file"; echo) \
>   | head -1 | cut -b-60
> done
# logrotate -f /etc/logrotate.conf
```

   This somewhat lengthy command, besides being another example of how useful scripting can be, will print out all the files matching `auth.log*`, with their name, and the first 60 characters of the first line. Pay attention to the association between the file name and the contents. The output from a single run will look something like this:

```
auth.log        May 11 12:25:36 client1 sudo: mal : TTY=unknown ; PWD=/
auth.log.1      Mar 30 08:09:01 client1 CRON[3513]: pam_unix(cron:session):
auth.log.2.gz   Mar 26 08:09:02 client1 CRON[13480]: pam_unix(cron:session):
auth.log.3.gz   Mar 15 22:00:39 client1 gnome-keyring-daemon[1504]: removing
auth.log.4.gz   Jan 12 15:49:48 client1 gdm-session-worker[1341]: pam_unix(g
```

Run the lengthy command three or so times (you may like to put the commands into a script) and describe what happens to the files in `/var/log/` when they are rotated, based on the pattern you find from the outputs.

3. What would be a good thing to do with archived logs? What factors might you need to take into consideration?

4. Filtering Logs

Now that we have some self-maintenance in our logging, we can now work on keeping the administrator informed on what is going on by bringing attention to new and possibly undesirable events. This generally requires the administrator to check their email regularly. Serious messages might be delivered via a Pager, SMS, audio alarm etc.[8]

There are two fundamental methods of log scanning, the first is looking for various patterns that may constitute something interesting. The second method involves filtering out the mundane entries from the logs, so we're left with only interesting or new types of log information. The latter method is better in the general case, as most unexpected events would not be configured to be picked up in the first case (that's why they're unexpected). It does mean though, that the admin needs to tailor the configuration to filter out the uninteresting messages they would normally get, which can involve a lot of work on a new system, but it does mean the administrator gets a better feel for the day-to-day behaviour of the system.

We're going to use the latter method, using a tool called Logcheck. Install **logcheck** using the following procedure.

Procedure 1. Install Logcheck and Perform Initial Testing

1. Ensure you have a fresh record of what packages are available in the configured APT repositories.

```
# apt-get update
…
Reading package lists... Done
```

2. Install and configure an email server package called exim4, which will be used by logcheck.

```
# apt install exim4
…
# dpkg-reconfigure exim4-config
```

When configuring exim4, choose `Internet site`, use `server1.localdomain` as the mail name, leave empty the field of `IP-adresses to listen`. For the field of `other destinations`, use `server1.localdomain` as the only destination for which mail is accepted, leave `Domains to relay` and `Machines to relay` empty, and more importantly,

[8]See the **swatch** program if you want to run commands in response to certain log entries as they happen.

choose *mal* as the destination for mails sent to *root* and *postmaster*. We will reconfigure exim4 in more detail in the lab for email server.

After the configuration, make the changes effective and start the email server.

```
# update-exim4.conf
# systemctl start exim4.service
$ pstree
…
   -exim4
…
```

You should see exim4 in the output of **pstree** if everything went well.

3. Install mail utilities.

```
# apt install mailutils
```

4. Install the Logcheck package and its dependencies.

```
# apt-get install logcheck
```

5. Find the cron entry that was installed when you installed logcheck. This file will be in one of the `/etc/cron*` files or directories (**ls /etc/cron***). Write down this entry; there is a question in the assessment relating to this.

6. Test whether logcheck works by running **logcheck** by hand. In Ubuntu, when we installed LogCheck, an email server (Exim) was installed as well. Exim, per Ubuntu's default policy for that package, does not accept mail from users other than 'root' and 'postmaster' (a special mailbox name for the email administrator). In addition, **logcheck** should be run as the user "logcheck", so run it using **sudo** as follows:

```
$ sudo -u logcheck logcheck
This could take up to a minute, but usually much less
You have new mail in /var/mail/mal
```

A word of explanation about the final line regarding new mail: this line is output by your shell. After a command is completed, your shell will check your mail spool file to see if the file timestamp has changed, and if it has, it will output the message you see above.

One of the more primitive tools we can use for checking our mail on a Unix-like machine is the **mail** command, which is sufficient for looking at the messages in your mail spool file. There are, of course, much nicer programs, including **pine** and **mutt**, but we shall have a look at those when we take a closer look at email in a later lab.

```
$ mail
```

To work in the primitive **mail** application, type a number of a message to view the message (**q** to exit the viewer). Type **q** to exit the mail prompt (&) prompt and return to your shell.

Your first message will be very large, as it contains all the logs it hasn't already seen (and most of them will not be filtered as they are bootup messages that typically contain a lot of hardware-specific kernel messages). Subsequent runs will be much smaller, and if there is no output–which is very common–no mail will be sent.

You may be wondering why the email got sent to `mal`, and not `root`. It's because when you installed the operating-system for you, the installer automatically adds the user created during install to the file `/etc/aliases`, so any mail to root gets sent instead to that user. Have a quick look at it now, it's very small. We'll revisit it when we come to look closer at email servers.

If you didn't receive the message about the new mail, then the problem is in the `/etc/aliases`, add `root: mal` to the end of the file, then run **newaliases**. You will need to regenerate the log message.

Procedure 2. Instructing Logcheck to Ignore Particular Log Entries

One of the tasks that you will commonly be doing, quite repetitively, on a new server is adjusting the logcheck rules, ignoring messages that are not interesting.

1. So now let's try telling logcheck to ignore a certain entry. Let's assume you are regularly seeing a particular log message that you don't care about, and the log entry contains `-- HELLO WORLD --`.

   Create the file `/etc/logcheck/ignore.d.server/local`, and insert a single line with `-- HELLO WORLD --`. Spaces matter, and there must be no blank lines in the file.

2. Run **sudo -u logcheck logcheck**, just so that we can easily see (or rather, not see) our log message that we're about to submit.

   Check your mail, just so you can tell the old messages from the one that might appear on the next run of **logcheck**.

3. Use **logger -- '-- HELLO WORLD --'** to log the entry we want to ignore.

4. **Exercise**

   As an exercise run Logcheck, then check your mail again. You shouldn't get a message, but if do, check that it doesn't contain the log message we are wanting to ignore. If you didn't get a message, that's excellent: all new log messages (including the one we just logged) were ignored, and so a report wasn't sent. Make sure your logcheck configuration file (where you added the entry) is correct.

4.1. Self-assessment

1. At what times is **logcheck** run by **crond**? Explain how **logcheck** is being run; you need to explain the use of **nice** as well as `-R`.

2. **[Optional]** Why should **logcheck** be run as the "logcheck" user, and not "root" or "mal"? What system administration principle does this illustrate?

3. Make sure you have done all the exercises.

5. Regular Expressions

In order to meaningfully filter log files, we need to learn how to write good regular expressions. A regular expression (*regexp* for short), is a pattern that matches text. It's similar to, but more advanced than wildcards[9], such as `*.pdf`. Wildcards are generally only used for matching filenames etc, whereas regular expressions are used for most other matching tasks, and are much more powerful.

[9]Referred to as *glob* patterns.

Regular expressions consist of various meta-characters which have special meaning. There are a handful of regexp flavours, and so there are some differences between them. Since learning regular expressions can be a bit of work, we'll stick to reasonably easy expressions, enough to get you through Logcheck effectively.

Regular expressions can be immensely powerful, being represented to it's highest degree in Perl Compatible Regular Expressions, which other languages often have some support for as well. It is largely the power of regular expressions that draws many administrators to Perl.

I'll show you the POSIX Extended regular expression syntax, since that's what **logcheck** uses, and it should be preferred compared to the older Basic syntax. **egrep** (or **grep -E**) is a command that uses the POSIX Extended syntax, whereas **grep** uses POSIX Basic syntax. Logcheck uses **egrep -i** internally, the `-i` makes the matching case-insensitive.

The following table explains the meta-characters available for use with POSIX Extended regular expression syntax.

In the following examples, a custom tool called **regex_test** has been used. It works much the same as **egrep**, but its output is more useful for learning how regular expressions work. In particular, it will only show the *first* match. This tool should be available in the `Lab Resources/Regular Expressions` folder. On your workstation, copy the file `regex_test.c` from the Lab Resources into your virtual machines shared folder, and from inside your virtual machine compile it:

```
$ cd /media/host/
$ make regex_test
cc regex_test.c -o regex_test
$ sudo install -o root -g root -m 0755 regex_test /usr/local/bin/
```

Table 1. Regular Expression Cheatsheet

Syntax	Description

`.`	Matches any single character, but not a new-line character.
	`$ echo "hello" | regex_test '.'`
	`hello`

`*`	Matches *zero* or more occurences of the previous item.
	`$ echo "hello" | regex_test '.*'`
	`hello`
	`$ echo "abba" | regex_test 'a*'`
	`abba`
	`$ echo "ac" | regex_test 'ab*'`
	`ac`
	`$ echo "abba" | regex_test 'b*'`
	`abba`
	`  (successfully matched 0 characters from index 0 to 0)`
	`BEWARE: Leftmost longest match, especially with *`
	`$ echo "abba" | regex_test 'ab*'`
	`abba`

`+`	Matches *one* or more occurences of the previous item. `a+` is much like `aa*`, but much more convenient when you have larger things to repeat, as we see in the grouping operator below.
	`$ echo "ac" | regex_test 'ab+c'`
	`<NO MATCH>`
	`$ echo "abc" | regex_test 'ab+c'`
	`abc`

Syntax	**Description**

```
$ echo "abbc" | regex_test 'ab+c'
abc
```

?	Matches zero or one of the previous item. In other words, the previous item is optional.

```
$ echo "ac" | regex_test 'ab?c'
ac
$ echo "abc" | regex_test 'ab?c'
abc
$ echo "abbc" | regex_test 'ab?c'
<NO MATCH>
```

^	*Anchors* the match to begin at the start of the line.

```
$ echo "spline" | regex_test 'line'
spline
$ echo "spline" | regex_test '^line'
<NO MATCH>
$ echo "line #1" | regex_test '^line'
line #1
```

$	Anchors the match to finish matching at the start of the line.

```
$ echo "linear" | regex_test 'line'
linear
$ echo "linear" | regex_test 'line$'
<NO MATCH>
$ echo "spline" | regex_test 'line$'
spline
$ echo -e "... in Section 5, where ... two lines of input
> Section 5" | regex_test '^Section 5$'
<NO MATCH>
Section 5
```

[…]	Matches a *single* input character, which must be one of the characters listed between the square brackets. Most characters inside the square brackets lose any special significance they usually have, though some gain special significance, to allow things like ranges and negation.

```
$ echo "square" | regex_test '[aeiou]+'
square
$ echo "–123.45" | regex_test '[0-9]+'
- introduces a range
-123.45
$ echo "-123.45" | regex_test '[0-9.-]+'
A literal - must be last. . is no longer a meta-character.
-123.45
$ echo "0x0800C4DF" | regex_test '0[xX][0-9a-fA-F]+'
0x0800C4DF
$ echo 'some "quoted" text' | regex_test '"[^"]*"'
^ at the start negates the match: any but a "
some "quoted" text
$ echo 'but "there are \"limits\" to regexps" so watch out' \
>   | regex_test '"[^"]*"'
but "there are \"limits\" to regexps"
```

{n} and {m,n}	Bounded repitition. Matches the previous item m times exactly, or between m and n times. The upper or lower bound may be omitted, but leave the comma if you want an the range unbounded high or low.
\	Removes (escapes) special meaning from a meta-character.

Syntax	**Description**

```
$ echo '123 1.2 123' | regex_test '[0-9].[0-9]'
123 1.2 123
$ echo '123 1.2 123' | regex_test '[0-9]\.[0-9]'
123 1.2 123
```

`(foo)`	Grouping construct. You could use it with the repetition qualifiers above.

```
$ echo '192.168.1.2' | regex_test '([0-9]+\.){3}[0-9]+
192.168.1.2
$ echo '1.2 1.2.3.4.5 192.168.1.2' | regex_test '([0-9]+\.){3}[0-9]+
1.2 1.2.3.4.5 192.168.1.2  Trap!: beware what comes after
```

`(foo|bar)`	Alternation inside a group. For example, the pattern `(foo|bar)` would match all of `foo` *or* `bar`.

```
$ echo '... Figure 1.2 ...
>  ...
>  ... Table 1.1 ...' | regex_test -i '(figure|table) [0-9.]+'
... Figure 1.2 ...
<NO MATCH>
... Table 1.1 ...
```

`[[:alpha:]]`	Named character classes, can be used to specify things such as a “alphabetic” character, a “lowercase” character, a “digit”, etc. See re_format(7) for further details. Constructions such as `[a-zA-Z]` are not sufficient in a non-English locale. For example, in Spanish you would also consider ‘ñ’ as a letter. What is matched depends on the *locale*: in an English locale, ‘ñ’ would not be expected to matched, but unfortunately, it probably would be[a]. **You are not expected to try this.** Nor are you expected to know how to type in Spanish or Chinese, but it is important to be aware of the differences.

```
Match some alphabetic characters
$ echo '¡Español!' | LANG=es_MX.UTF-8 regex_test '[[:alpha:]]+'
¡Español!
Use LANG=C to ensure 7-bit ASCII
$ echo '¡Español!' | LANG=C regex_test '[[:alpha:]]+'
¡Español!
But matching is entirely too inclusive
$ echo '###' | LANG=es_MX.utf8 regex_test -i '[[:alpha:]]+'
###  Chinese characters are not valid Spanish alphabetic characters.
```

[a]Any Unicode character with a “Letter” property is actually what is matched. While explainable, is is likely unexpected in the context of a locale.

Here is a simple procedure to show you how to write a simple regular expression to match log file entries.

1. Use the following log entry as an example for the rest of procedure.

```
Feb 17 13:17:44 belgarath snmpd[2978]: Connection from 10.18.1.1
```

2. Identify the parts that will change, and the parts that will stay the same. Syslog entries have the date and time stamp, which will definately change. `belgarath` in this example is the name of the host that submitted the log. That will stay the same, and is worth including

for a point of reference[10]. snmpd is the process name the submitting agent gave. That is a key part to identifying the log message. The number isn't important, it's the PID and will change. The string `Connection from` is important. Together with the `snmpd` part it practically identifies the whole line.

We have an IP address in the line. If the service is used by many different IP addresses in the same subnet, you may elect not to match only part of it, say `10.18.`, which can be used for matching everything in the 10.18.0.0/16 subnet.

3. Your system's logcheck will likely start with a consistent header to match the timestamp and hostname part that Syslog prepends to the message. Thus, you can replace the timestamp and hostname with whatever other logcheck entries start with:

```
^\w{3} [ :0-9]{11} [._[:alnum:]-]+ snmpd[2978]: Connection from 10.18.1.1
```

The `\w` is not documented in regex(7), but is mentioned in the GNU "Info" page. It is equivalent to `[[:alnum:]]` and is a shorthand notation borrowed from Perl regular expressions. Consider it a GNU extension.

4. Escape every meta-character in the input you wish to match, by prefixing it with a \ in the regular expression. Note that not all punctuation characters are meta-characters. Ignore the header part which we've already fixed up.

```
^\w{3} [ :0-9]{11} [._[:alnum:]-]+ snmpd\[2978\]: Connection from 10\.18\.1\.1
```

5. Replace varying numerical sequences with `[0-9]+` This is useful for the process identifier.

```
^\w{3} [ :0-9]{11} [._[:alnum:]-]+ snmpd\[[0-9]+\]: ↩
Connection from 10\.18\.[0-9]+\.[0-9]+
```

6. Your completed regular expression should look like this.

```
^\w{3} [ :0-9]{11} [._[:alnum:]-]+ snmpd\[[0-9]+\]: ↩
Connection from 10\.18\.[0-9]+\.[0-9]+
```

7. Test the regexp using **egrep -i**. We suggest that you put the input text into a file, to save typing.

```
$ echo 'Feb 17 13:17:44 belgarath snmpd[2978]: Connection from 10.18.1.1' \
> > log_message.txt
$ egrep -i 'belgarath snmpd.[0-9]*.: Connection from 10.18.1.1' log_message.txt
Feb 17 13:17:44 belgarath snmpd[2978]: Connection from 10.18.1.1
The line was printed, so it matches.
```

5.1. Self-assessment

1. Transform the following two log messages into two suitable **egrep** (POSIX Extended) style regular expressions for use with Logcheck.

```
Feb 10 18:32:22 belgarath sshd[2947]: Accepted publickey for theauthor from ↩
10.18.2.11 port 34061 ssh2
```

```
Feb 17 06:25:02 belgarath su[20870]: + ??? root:nobody
```

[10]Syslog can be made to accept remote log messages.

I have `indicated` those parts of each message that would change each time *or* will appear differently in each. Some of it you may wish to keep. For example, you may wish to only match part of the IP address, if you routinely have people logging in from 10.18.2.`X`; or perhaps you have many users and don't want to match each user explicitly.

In the second example, the `+` indicates the beginning of a session. The `???` is for the terminal (typically something like `/dev/tty1`,) but in this case there is no controlling terminal associated with this command. The `?` is special to **egrep** (meaning the preceding is optional,) so you will need to escape it, turning each literal `?` into `\?`. `root:nobody` indicates that the root user transitioned to the nobody user. Since this is effectively dropping privileges, it's generally okay to ignore this event.

6. Final Words

There is, as always, a *lot* more we could have taught you in this lab, but limited time in a paper does not allow us to go that deeply.

One thing that you may want to look at is remote logging. Then you just need to point your devices at your syslog server... this includes such devices as routers, wireless access point, and network-enabled printers, as well as servers and even workstations.

But there are some security issues that you should at least consider when doing this: you generally want to at least limit who can submit logs, and implement some rate-limiting and file-system management so that your log server doesn't run out of disk space. Rsyslog, as part of the RELP logging improvements to Syslog, has added features for reliability and security, although most senders won't support them at present.

The value of logs coming from remote devices is generally *very* useful for diagnosing faults quickly, as it removes a lot of guesswork, and gives you a concrete message with which to start your diagnosis.