ECE 459: Programming for Performance
Lab 1—Package Management[1]

Created by Patrick Lam & Jeff Zarnett
Due: January 27, 2023 at 23:59 Eastern Time

In this lab, you will write a program that computes dependencies for software packages, and also performs network requests using non-blocking I/O.

Learning Objectives:

- Become familiar with Rust and its idioms.
- Learn how to use non-blocking I/O to increase performance.

Random tips. My implementation is slow enough and the input is big enough that I have to test with `cargo run --release`; the debug build is too slow. Also, be aware of the Rust version on the server vs on your machine and be sure to make sure your code works on the version on the server.

Package Management Background. I [PL] was recently trying to record a video for a class and I was having a really bad IT day. The first solution to having a bad IT day is to reboot. That didn't help. So, instead, I reinstalled everything on one of my old laptops. (That did help). I was inspired, in my dealings with the package manager, to have you familiarize yourself with Rust by writing some parts of a package manager.

Package managers are everywhere: your phones and computers all have multiple package managers. A package manager tracks packages. A package is a bunch of files (the data) along with a name, version number, and a set of dependencies (metadata). Rust itself has the cargo package manager. For this assignment, we'll be working with a Debian-style package manager.

When installing a package, a package manager needs to make sure that the package's dependencies also get installed. For instance, Unix program `xeyes`[2] belongs to package `x11-apps`, which can't be installed without package `libxft2` (and many others) also being installed. In fact, the package manager will construct a dependency graph.

There are two complications here. 1) The package manager may have a choice of packages when satisfying a dependency. This is indicated by a vertical bar `|` between alternatives; for instance, Debian package `abcde` depends on `cdparanoia | icedax`, along with other dependencies. Any of the choices is valid. 2) A dependency may be versioned; back to `x11-apps`, it requires that `libxft2` have version strictly greater[3] than 2.1.1.

[1] v9, 2023-01-12

[2] Try it!

[3] I've implemented version comparison for you. You're welcome!

Clarification: non-available packages. We will not have any test cases where you have to (potentially transitively) install any package that has no available version.

Part 1: Manage That Package

Your task is to implement three groups of functions: parsing, direct queries, and transitive queries.

Parsing: `parsers.rs` (15)

The first order of business is to parse package data.

```
$ load-packages data/mirror.csclub.uwaterloo.ca_debian_dists_sid_main_binary-amd64_Packages
$ load-installed data/installed-packages
```

You can also use abbreviations `lp` and `li`, or convenience command `load-defaults (ld)` to load the supplied files listed above, once you have everything working.

This package data comes in text files. For our purposes, it'll work if we consider each line of a text file to be of colon-separated form `Key: Value`. I've provided a regular expression `KEYVAL_REGEX` that you can use to parse the line, as follows.

```
let kv_regexp = Regex::new(KEYVAL_REGEX).unwrap();
match kv_regexp.captures(&line) {
    None => (),
    Some(caps) => {
        let (key, value) = (caps.name("key").unwrap().as_str(),
                            caps.name("value").unwrap().as_str());
        // ...
```

There are two files to parse: the list of available packages, and the list of installed packages.

Installed packages (5) Installed packages is easier. Implement function `parse_installed()`. You need to recognize two keys: `Package` and `Version`. For a `Package` line, call the function `self.get_package_num_inserting(&value)` and remember the returned value for later. For a `Version` line, parse the version number with

```
let debver = value.trim().parse::<debversion::DebianVersionNum>().unwrap();
```

and call `self.installed_debvers.insert()` with the memorized package number and the `debver` you just computed.
I should point out that method `get_package_num` converts a package name to an `i32` while `get_package_name` goes the other way.

Available packages (10) This is pretty similar to installed packages, but there are more keys. The `MD5sum` and `Version` keys are nothing new, but insert into maps `self.md5sums` and `self.available_debvers` instead.

However, you also have to parse the dependencies, which are at key `Depends`. Your goal is to produce a `Vec<Dependency>`. Each `Dependency` is itself a `Vec<RelVersionedPackageNum>`. The separator between `Dependency`s is a comma (`,`), while the separator between alternatives within a `Dependency` is the vertical bar (`|`). I recommend the use of `split` to get individual alternatives.

Example: parse the dependency $A, B|C, D$ as the `Vec<Dependency>` containing $[[A], [B, C], [D]]$. At the top level, you have the list of dependencies, all of which must be satisfied. Each dependency in the list of dependencies consists of a list of alternatives.

Once you have an alternative (e.g. `libxft (» 2.1.1)`), I have provided you with the regex `PKGNAME_AND_VERSION_REGEX` to parse it. You can request `pkg`, `op`, and `ver` from a match of the regex, and you can futher parse `op` and `ver` with
`(op.as_str().parse::<debversion::VersionRelation>().unwrap(), ver.as_str().to_string())`
All you have to do is to put it in a `RelVersionedPackageNum` struct.

Simple dependency calculations: `deps_available.rs` (25)

Now that you've implemented parsing for dependencies, you can use the `info` command to print out all loaded information about a package.

Let's now combine the available and installed data. You are going to implement `deps_available`, which checks the declared dependencies of a package and prints out whether they are satisfied or not. (Please stick to the provided output format for the TAs' sake; I've provided the necessary `println!` statements).

```
$ deps-available 4g8
Package 4g8:
- dependency "libc6 (>= 2.4)"
+ libc6 satisfied by installed version 2.33-1
- dependency "libnet1 (>= 1.1.2.1)"
-> not satisfied
- dependency "libpcap0.8 (>= 0.9.8)"
+ libpcap0.8 satisfied by installed version 1.10.1-4
```

I'd recommend iterating over the `dependencies` and using a helper function `dep_is_satisfied` for each `Dependency`. A dependency is satisfied if the relevant package is installed (i.e. installed-version is set) and meets the stated version constraints.

I've provided helper functions: you can call
`v.parse::<debversion::DebianVersionNum>().unwrap();`
to convert string `v` into a `DebianVersionNum`, and
`debversion::cmp_debversion_with_op(op, iv, &v)`
to see whether `iv` and `v` satisfy relation `op`.

I also recommend that you implement method `dep_satisfied_by_wrong_version` for use below. It returns the `Vec` of dependencies that would be satisfied, but have the wrong version installed.

Solvers: `solvers.rs` (40)

Finally, we'll wrap up this part with two commands, `transitive-dep-solution` and `how-to-install`, which do more sophisticated reasoning on collections of packages.

I recommend the use of *worklist* algorithms here, which you should have seen previously. The idea is that the main loop of the algorithm picks an item from the worklist, does work on that item, and puts new work that results from that item back onto the worklist. It iterates until there is no more work to be done.

Transitive dep solution (10)

```
$ transitive-dep-solution libzydis-dev
"libzydis-dev" transitive dependency solution: "libzycore-dev, libzydis3.2,
libzycore1.1, libc6, libgcc-s1, gcc-11-base"
```

This command computes one transitive solution to a package's dependencies, i.e. it includes all dependencies of dependencies. It's implemented by function

```
pub fn transitive_dep_solution(&self, package_name: &str) -> Vec<i32>
```

The key here is whenever this algorithm has a choice (alternatives), it chooses the first alternative. You may assume that this alternative is installable.

To implement this solver, use a dependency set `Vec`. Populate it with the first alternatives of `package_name`. Iterate through the dependency set and add all new dependencies of packages already in the dependency set (again, resolving alternatives by picking the first one). Stop if an iteration didn't add any new dependencies.

Computing how to install (30) The first part of the lab culminates in this method.

```
$ how-to-install 3depict
Package 3depict:
"3depict" to install: "libftgl2, libgsl27, libmgl7.6.0, libgslcblas0, libhdf4-0, libhpdf-2.3.0, libmgl-data"
```

This time, you implement method

```
pub fn compute_how_to_install(&self, package_name: &str) -> Vec<i32>
```

The differences here are:

- filter out dependencies that are already satisfied (you have `dep_is_satisfied`);
- when there is a choice (dependency `A|B`), and one of the choices is already installed, but the wrong version, then install that choice;
- when choosing between options (either because none is installed, or because multiple wrong versions are installed), go ahead and compare apples and oranges: install the higher version number.

Again, you may assume that all packages that are mentioned are also installable (though it doesn't hurt to check).

Clarification: I meant that choices here would optimistically assume that the thing that you pick will satisfy the dependency requirement, which is not necessarily true. You can either make

that assumption, or you can check that a choice satisfies the requirement before considering it, and panic if there are no valid choices.

Also, during this computation, don't make any changes to the set of installed packages. For instance, if you ask `how-to-install libscim-dev`, it has a dependency on `libgtk2.0-dev` and another one on `libgtk-3-dev | libgtk2.0-dev (<< 2.21)`. Consider the first one, then consider the second one based on the initial state—don't assume that you've installed `libgtk2.0-dev` before considering the second dependency.

Part 2: Nonblocking I/O

In this part, you will implement handling for the `verify` command in the program. Every package that is published has an associated md5 hash. The hash is computed by looking at the package itself. In principle, this hash can be used to check whether the package has been tampered with, or if there was an error in transferring it somehow, or if the dodgy 10 year old USB key you stored your package on isn't to be trusted[4].

We have provided a little web server that can be used to retrieve the correct md5 hash, given a package name and version. This web server will be running at `ece459.patricklam.ca` on port `4590`. If the server is down, overloaded, or otherwise unavailable, you can run the server yourself for the purposes of your own development/testing: https://github.com/jzarnett/package-verifier. Please note that because you can test with your own server, if the "official" server is offline, you're not blocked. Downtime of the official server is not going to be a valid cause for an extension.

You can use the `set-server` command on your local solution to set the provided `server` state element in `AsyncState`; it points to `ece459.patricklam.ca:4590` by default, but you can point it to `localhost:4590` to run locally, for instance, which would work provided that you are running your own copy of the package verifier.

The server has one REST endpoint:

```
GET /rest/v1/checksums/{package_name}/{version}
```

For clarity, this endpoint takes two path parameters: `package_name` and `version`; replace those with the actual package name and its version that you would like to query.

The server will check its registry and if it finds a matching combination of name and version, it returns HTTP 200 with the body of the response being the hash for that package. If the server cannot find a corresponding combination of name and version, it will return a HTTP 404. Other invalid requests may result in HTTP 400 or other error codes.

Usage example:

```
curl 127.0.0.1:4590/rest/v1/checksums/example/1.0
```

[4]For the purposes of this assignment, we aren't actually going to compute the local md5 hash for packages, so that you don't have to have a bunch of packages around to test your assignment. Use your imagination.

```
0e0e8016d5f8b57eb13777efbb2893c9
```

(Not sure what status code you're getting back? Try `curl -v`.) You then compare the md5 hash received from the server against the md5 hash you have locally for the package. They should all be a match; if not, something is wrong.

Implementing the Verifier

The verifier will use a library called "curl" . The curl library uses *callbacks*. If you are not familiar with callbacks, the idea is pretty simple. You call curl's `perform()` function to do some work (i.e. interact with a web service), and it calls functions (which you provide) while carrying out that work. When the work is complete, curl returns from your call to `perform()`. Basically you are saying to curl "perform this task, and let me know when you need me to provide some data to you or if you have some data to give to me" .

To get a recap on curl, check the lecture notes; for a deeper dive into the topic, consult the ECE 252 material (even if it is in C).

Your job is to implement the verification logic so that it uses non-blocking I/O, that is, the curl "multi" interface. Your solution should *not* use threading.

The way the program will work is as follows. At the user's leisure, as they are using the package manager, they can request a verification of a package. This causes the package manager to enqueue a non-blocking curl request to the server, storing a handle for this request. Later, the user can execute the queue. At that point, the package manager does a blocking wait until all the answers come in, and prints out the requested verification results. When the user quits the package manager, the package manager will also execute any remaining requests.

```
$ load-csv data/packages.csv
Packages available: 63846
$ enq-verify bash
queueing request http://ece459.patricklam.ca:4590/rest/v1/checksums/bash/5.1-6
$ enq-verify 0ad
queueing request http://ece459.patricklam.ca:4590/rest/v1/checksums/0ad/0.0.25b-1.1
$ enq-verify libc6 28
queueing request http://ece459.patricklam.ca:4590/rest/v1/checksums/libc6/28
$ quit
verifying bash, matches: true
verifying 0ad, matches: true
got error 404 on request for package libc6 version 28
```

For this part of the lab, you need to implement 2 functions in file `src/packages/async_fns.rs`: `enq_verify_with_version` and `execute`. You can also add any necessary fields to the struct `AsyncState`, and initialize them in `new`.

```
pub fn enq_verify_with_version(&mut self, pkg:&str, version:&str)
```

This function prepares a request for pkg's md5sum (passing its name and version number) and adds a corresponding handle to the queue (actually a `Vec`). It must return right away.

Hints. You can use the `format!` macro and `urlencoding::encode` to URL-encode the version number. You probably want to do something like in `mod.rs` with the packages, but this time using your own `EASYKEY_COUNTER` and using your index to find the parts of the state you need to keep (you can't index on an Easy2Handle). Think about what data you'll need to do the verification.

```
pub fn execute(&mut self)
```

This function asks curl to perform work until there's no more work to be done. You can ignore any errors that curl may raise while it's performing work. Once all the work is done, drain the set of keys to handles. For each handle: if you got a response code of 200, then compare the md5sums that came back with the md5sums that you have stored locally, and print the result of the comparison. For response codes ≥ 400, print out an error message. **Please match the provided format to enable TAs to automate some of the grading.**

Note. You can do this async I/O part independently from the first part, by using command `load-csv data/packages.csv`. Once you've implemented the parsers, you can also use the normal way of loading data to get the md5sums.

Benchmarking

You can measure how long your program takes to execute by using the hyperfine command on Linux. When you build your program, the executable is placed in target/release/rpkg, so here's a typical sequence:

```
> cargo build --release
> hyperfine -i "target/release/rpkg < [commands]"
```

The -i option to hyperfine tells it to ignore any error status returned from your program. The starter code does not explicitly return an error status from main(), so without the -i option you will find that hyperfine aborts without giving you useful output.

The < operator redirects input from file `commands` to `rpkg`. You can put your test scripts in files and run `rpkg` with those files using redirection.

Rubric

The general principle is that correct solutions earn full marks. However, it is your responsibility to demonstrate to the TA that your solution is correct. Well-designed, clean solutions are more likely to be recognized as correct. Solutions that do not compile will earn at most 39% of the available marks for that part. Solutions that compile but crash earn at most 49%. Grading is done by building your code, running it, checking the output, and inspecting the code. For this lab, we are not grading performance.

Part 1: Managing the Packages (80 marks)

Your code needs to correctly implement the package management functionality as described above. 15 marks for parsing, 25 for simple dependency calculations, 40 marks for solvers.

Part 2: Nonblocking I/O (20 marks)

Your code must properly use curl's "multi" features.

Clarifications

- *What about compilation warnings?* We encourage your code to be free of warnings but we won't dock marks if there is no underlying issue. You can also fix linter errors in the provided code.
- *What about error handling?* You can pretend that failures never happen (famous last words), i.e. you need not implement code to retry after failures. If you do notice an error, print a message and exit. You can also assume that the inputs are correctly-formatted and solveable.
- *What can we change in the provided code?* Go ahead and change it as you like. You can add extra optional command-line arguments and commands if you'd like, but don't make them mandatory (i.e. the TA should be able to plug-and-play your code just like all the other solutions.)
- Tell me more about the `how-to-install` algorithm.

  Here's an example (thanks to Alexander Kursell). A depends on $[B, C]$; B depends on $D \mid E$; and C depends on $E \mid F$. Let's say the rules resolve B's dependency to D and C's dependency to E.

  Proceed using an operational worklist-based definition.

  1. Put A in the worklist.
  2. Pull out A and get its dependency on $[B, C]$. Push B and C onto the worklist.
  3. Pull out B and get $D \mid E$. Put D on the worklist.
  4. Pull out C and get $E \mid F$. Put E on the worklist.
  5. Pull D and E from the worklist.

  You end up with install set $[A, B, C, D, E]$.

version 3 update: fixed incorrect output for 3depict in v2 of this writeup, add note about redirection.

version 4 update: change font size for how-to-install output.

version 5 update: clarify in `how-to-install` about not having to check that alternatives satisfy requirements (though it's also allowed to do that check).

version 6 update: clarify that `how-to-install` doesn't do incremental updates to state while calculating; add small comment about `curl -v`.

version 7 update: actually cycles in the dependency graph are fine; clarify that `transitive-dep-solutio` only adds new dependencies to the worklist.

version 8 update: We will not have any test cases where you have to (potentially transitively) install any package that has no available version.

version 9 update: 2023.