GNU Guix: the functional GNU/Linux distro that's a Scheme library

Ludovic Courtès

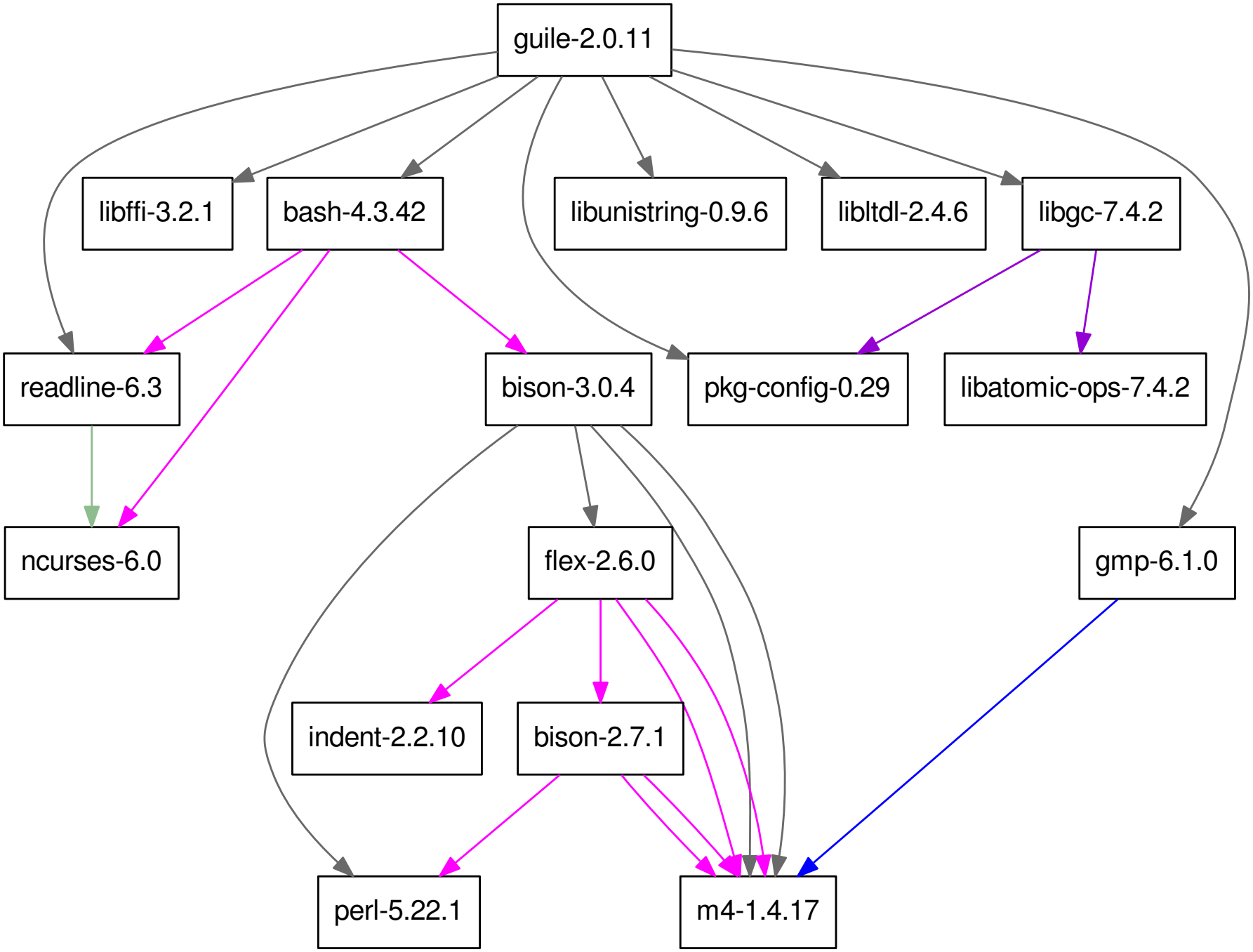
Scheme Workshop
18 September 2016, Nara, Japan

GuixSD

Functional package management.

```
$ guix package -i gcc-toolchain coreutils sed grep
…

$ eval `guix package --search-paths`
…

$ guix package --manifest=my-software.scm
…
```

Want to hack on Guile?

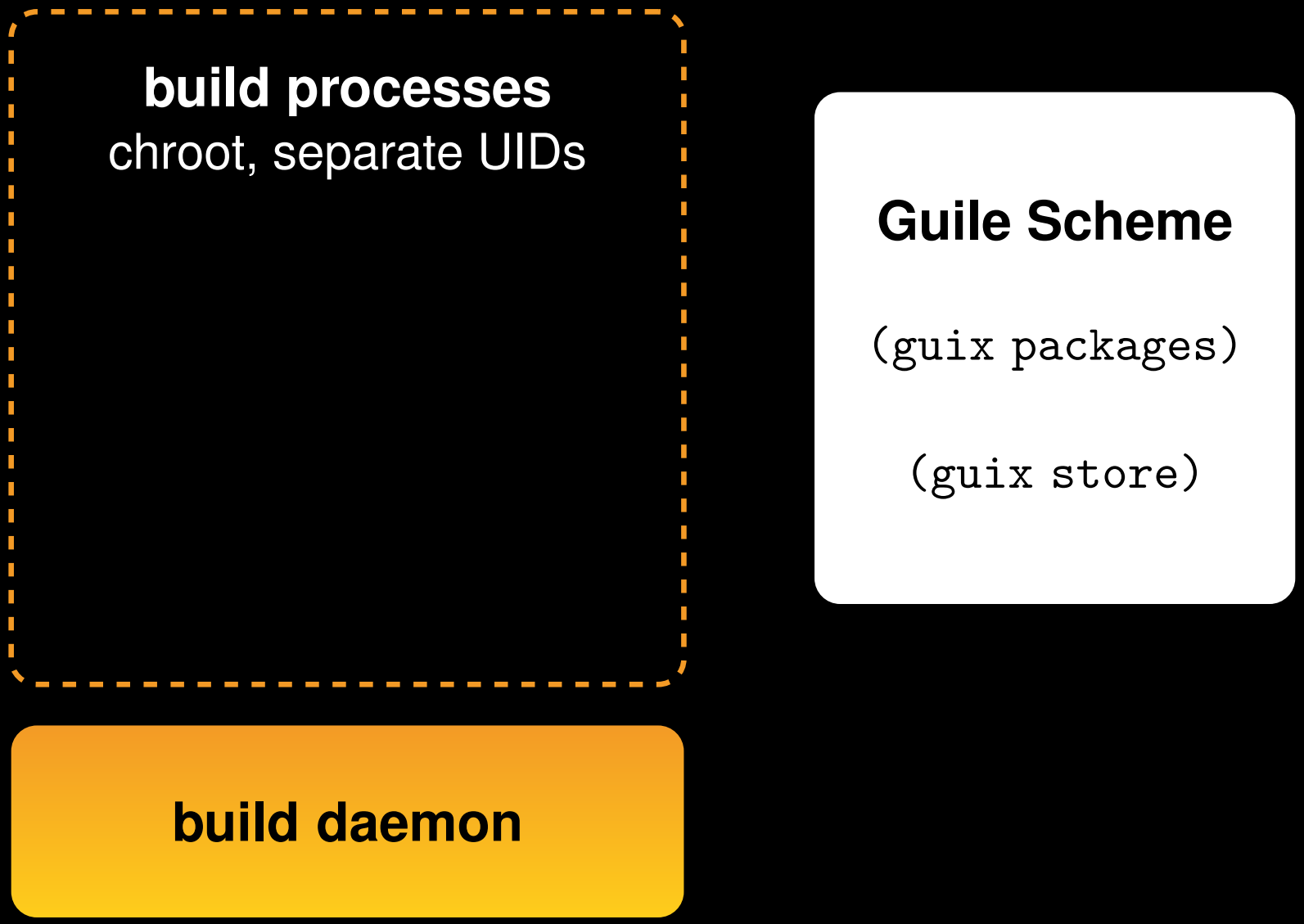

guile-2.0.11
libffi-3.2.1
bash-4.3.42
libunistring-0.9.6
libltdl-2.4.6
libgc-7.4.2
readline-6.3
bison-3.0.4
pkg-config-0.29
libatomic-ops-7.4.2
ncurses-6.0
flex-2.6.0
gmp-6.1.0
indent-2.2.10
bison-2.7.1
perl-5.22.1
m4-1.4.17

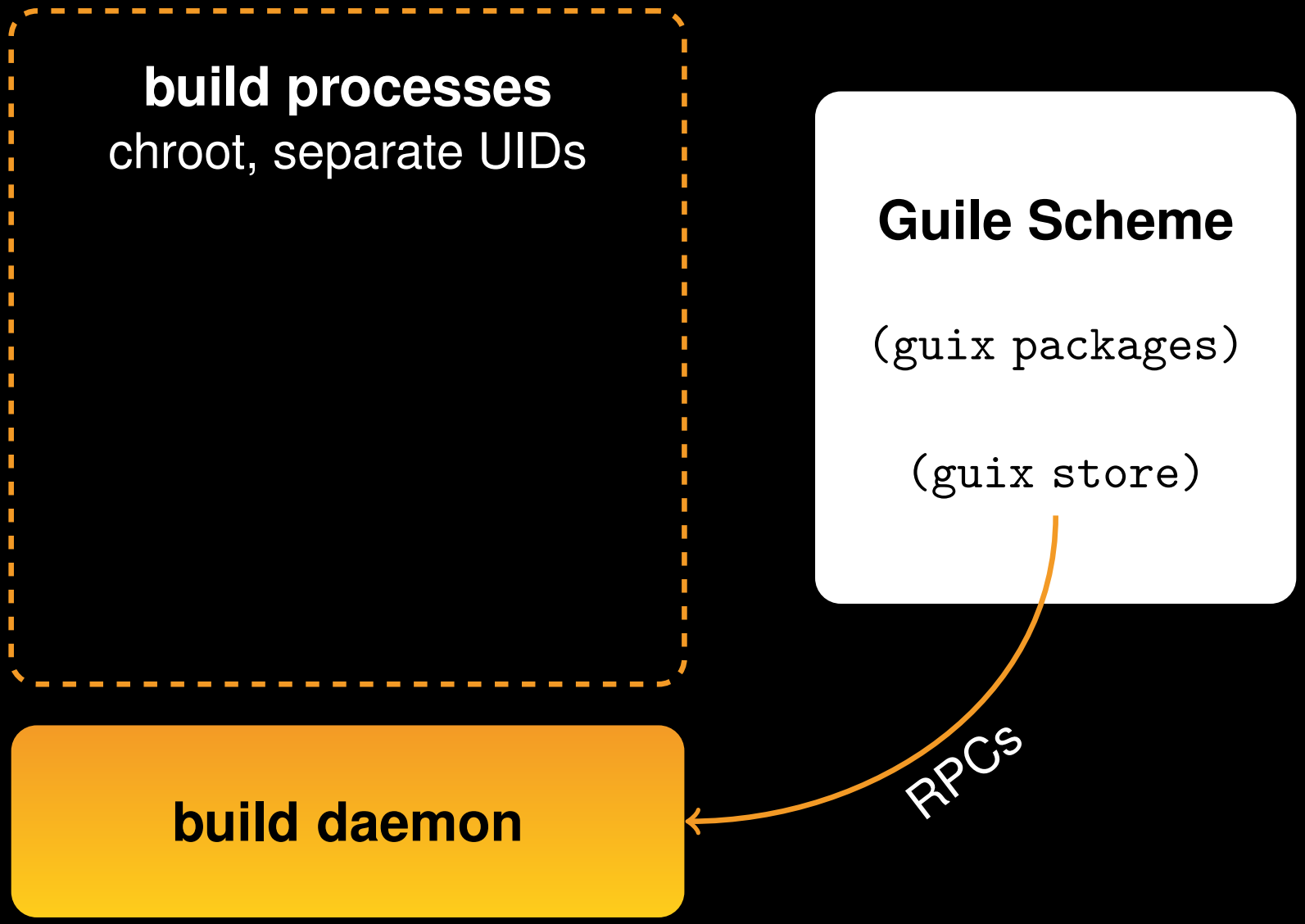
```
$ guix environment --container guile
…

$ guix environment --container guile \
     --ad-hoc git autoconf automake gdb
…
```

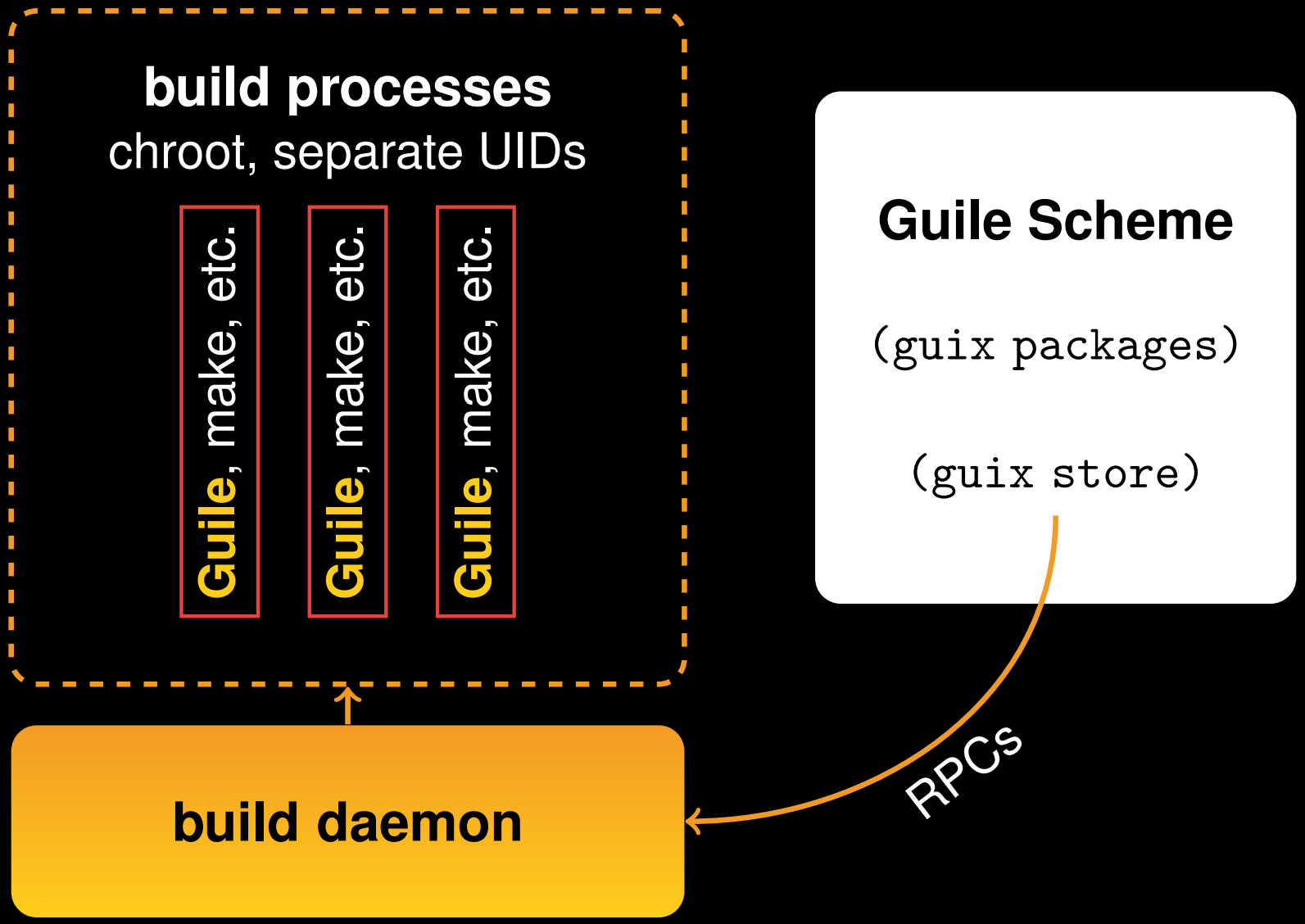
Functional package management paradigm:

1. build process = **pure function**
2. built software = **persistent graph**

Imposing a Memory Management Discipline on Software Deployment, Dolstra et al., 2004 (Nix package manager)


build processes
chroot, separate UIDs
Guile Scheme
(guix packages)
(guix store)
build daemon


build processes
chroot, separate UIDs
Guile Scheme
(guix packages)
(guix store)
RPCs
build daemon


build processes
chroot, separate UIDs
Guile, make, etc.
Guile, make, etc.
Guile, make, etc.
Guile Scheme
(guix packages)
(guix store)
RPCs
build daemon

```
$ guix build chibi-scheme
```

```
$ guix build chibi-scheme
/gnu/store/ h2g4sc09h4... -chibi-scheme-0.7.3
```

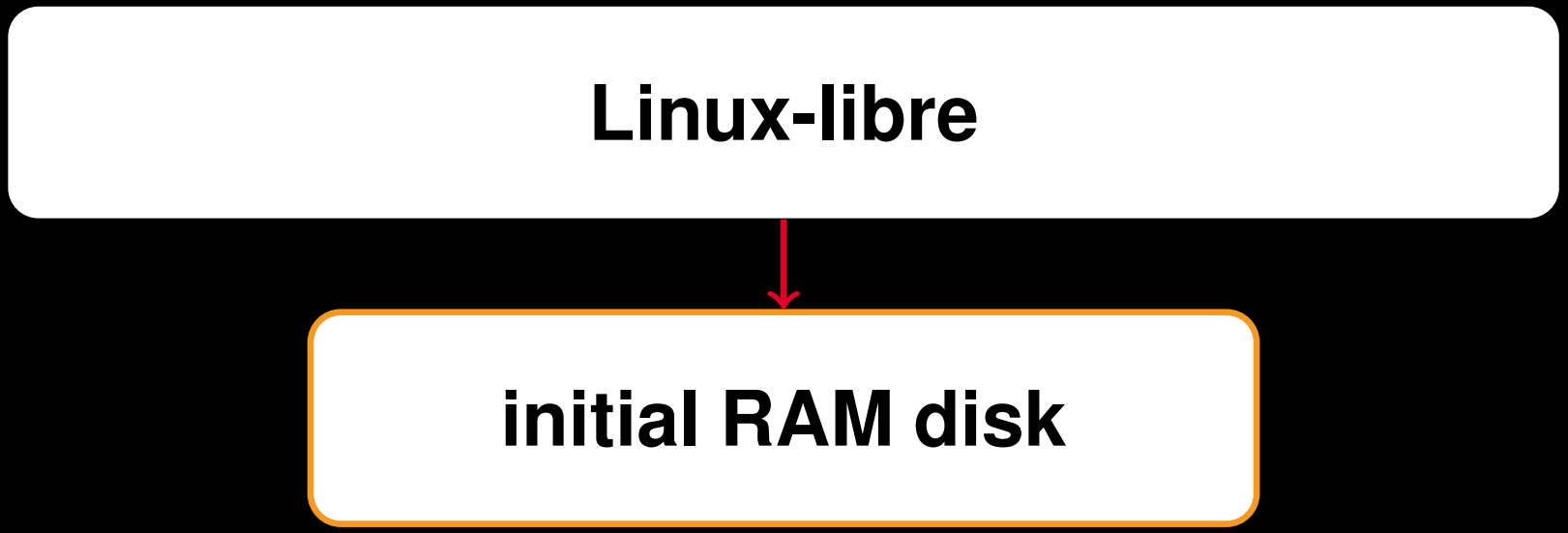
hash of *all* the dependencies

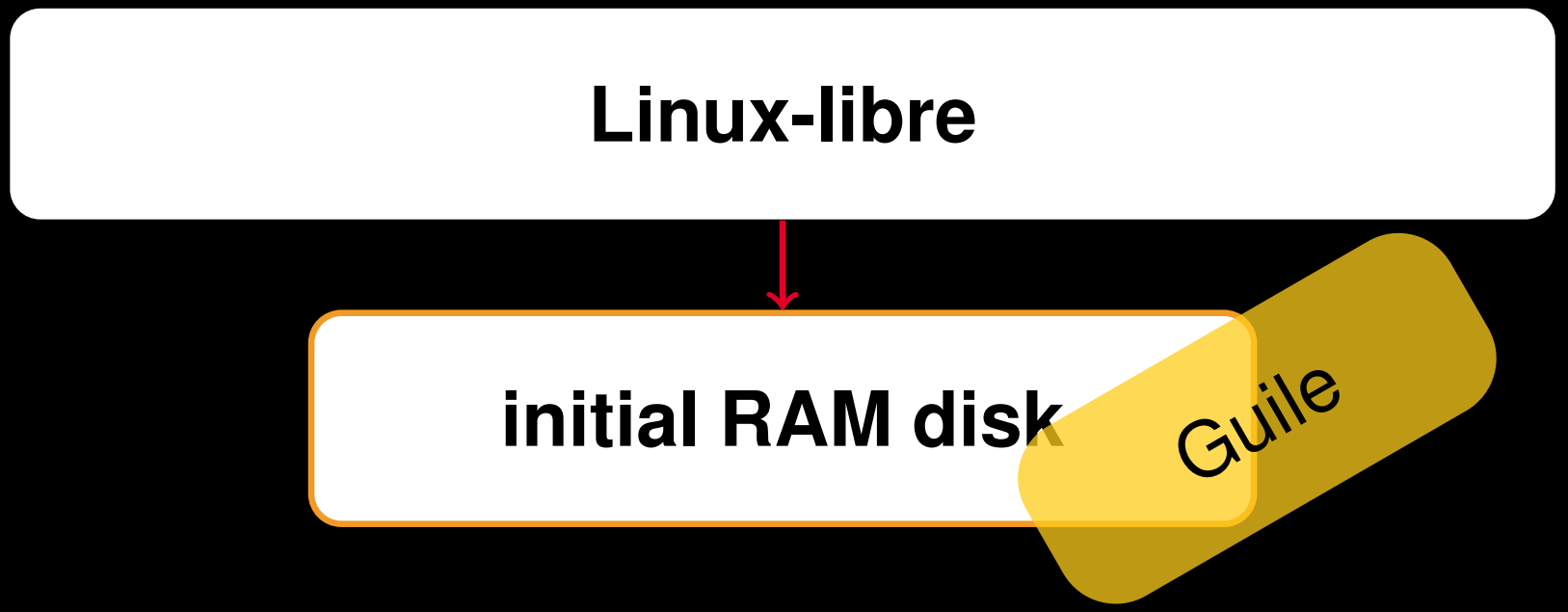
```
(define hello
  (package
    (name "hello")
    (version "2.8")
    (source (origin
              (method url-fetch)
              (uri (string-append
                    "http://ftp.gnu.org/…/hello-" version
                    ".tar.gz"))
              (sha256 (base32 "0wqd…dz6"))))
    (build-system gnu-build-system)
    (synopsis "An example GNU package")
    (description "Produce a friendly greeting.")
    (home-page "https://gnu.org/software/hello/")
    (license gpl3+)))

;; Yields: /gnu/store/…-hello-2.8
```

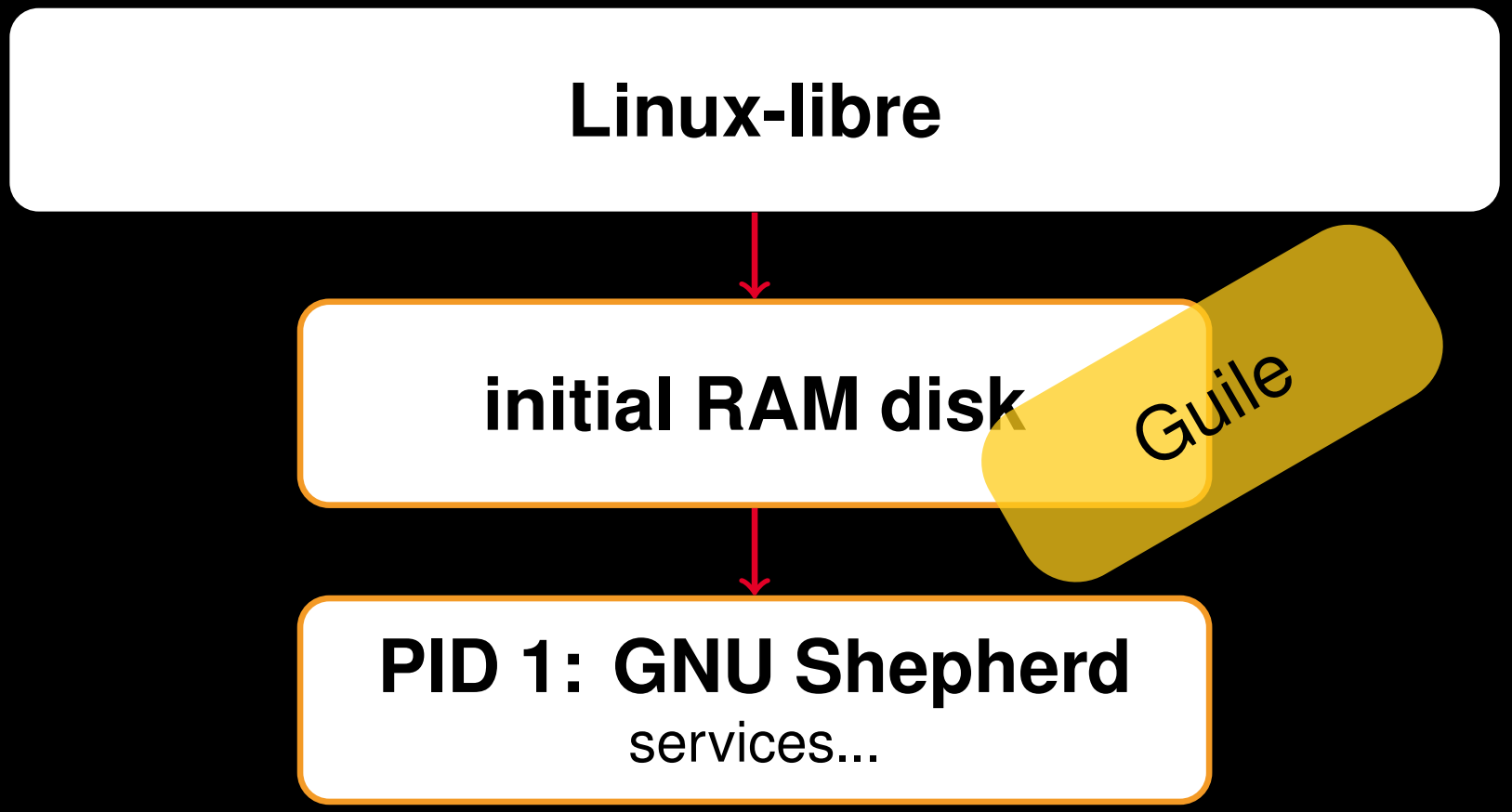
Scheme all the way down.

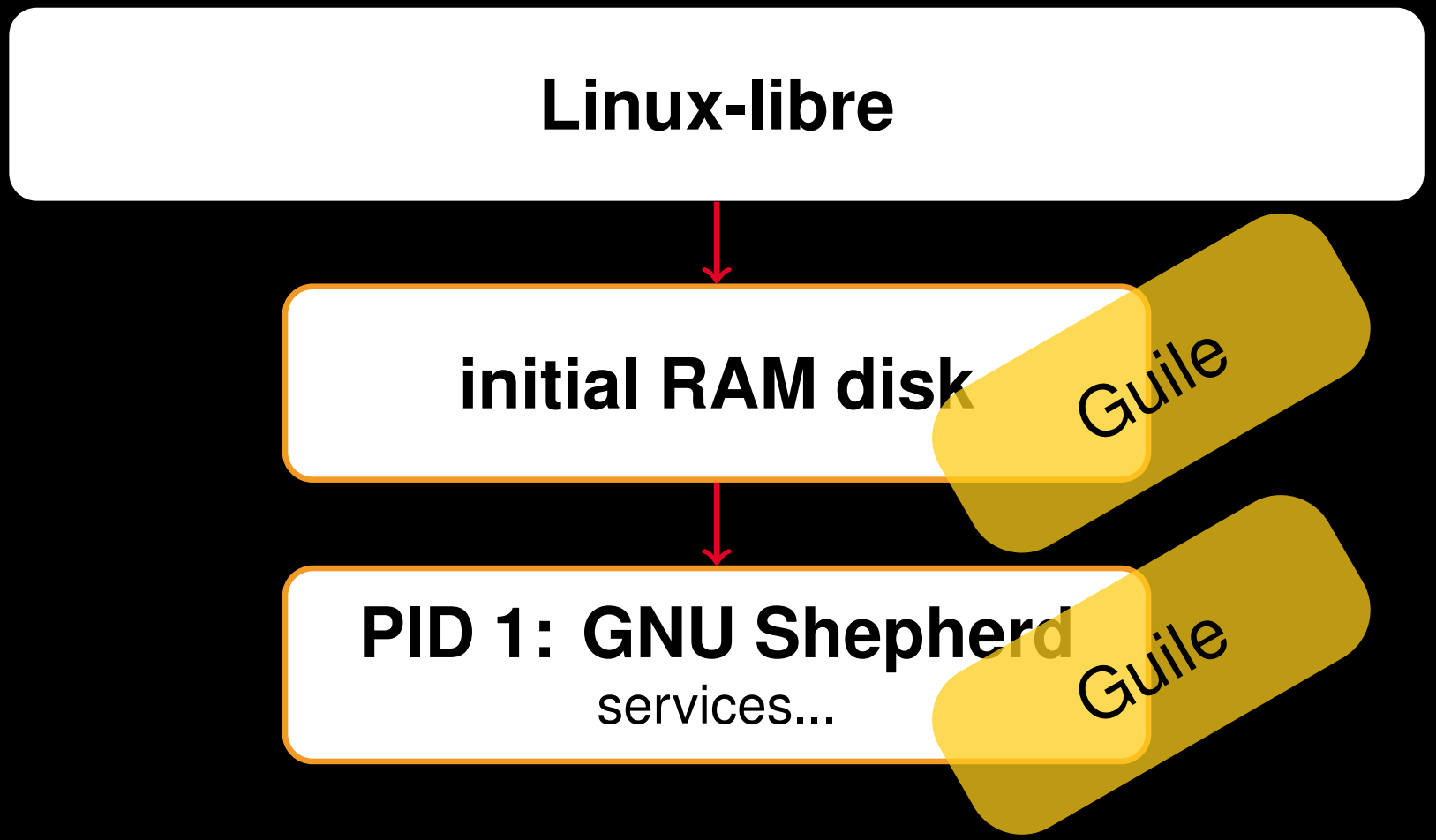
```
(operating-system
  (host-name "schememachine")
  (timezone "Japan")
  (locale "ja_JP.utf8")
  (bootloader (grub-configuration (device "/dev/sda")))
  (file-systems (cons (file-system
                        (device "my-root")
                        (title 'label)
                        (mount-point "/")
                        (type "ext4"))
                      %base-file-systems))
  (users (cons (user-account
                (name "alice")
                (group "users")
                (home-directory "/home/alice"))
               %base-user-accounts))
  (services (cons* (dhcp-client-service)
                   (lsh-service #:port-number 2222)
                   %base-services)))
```

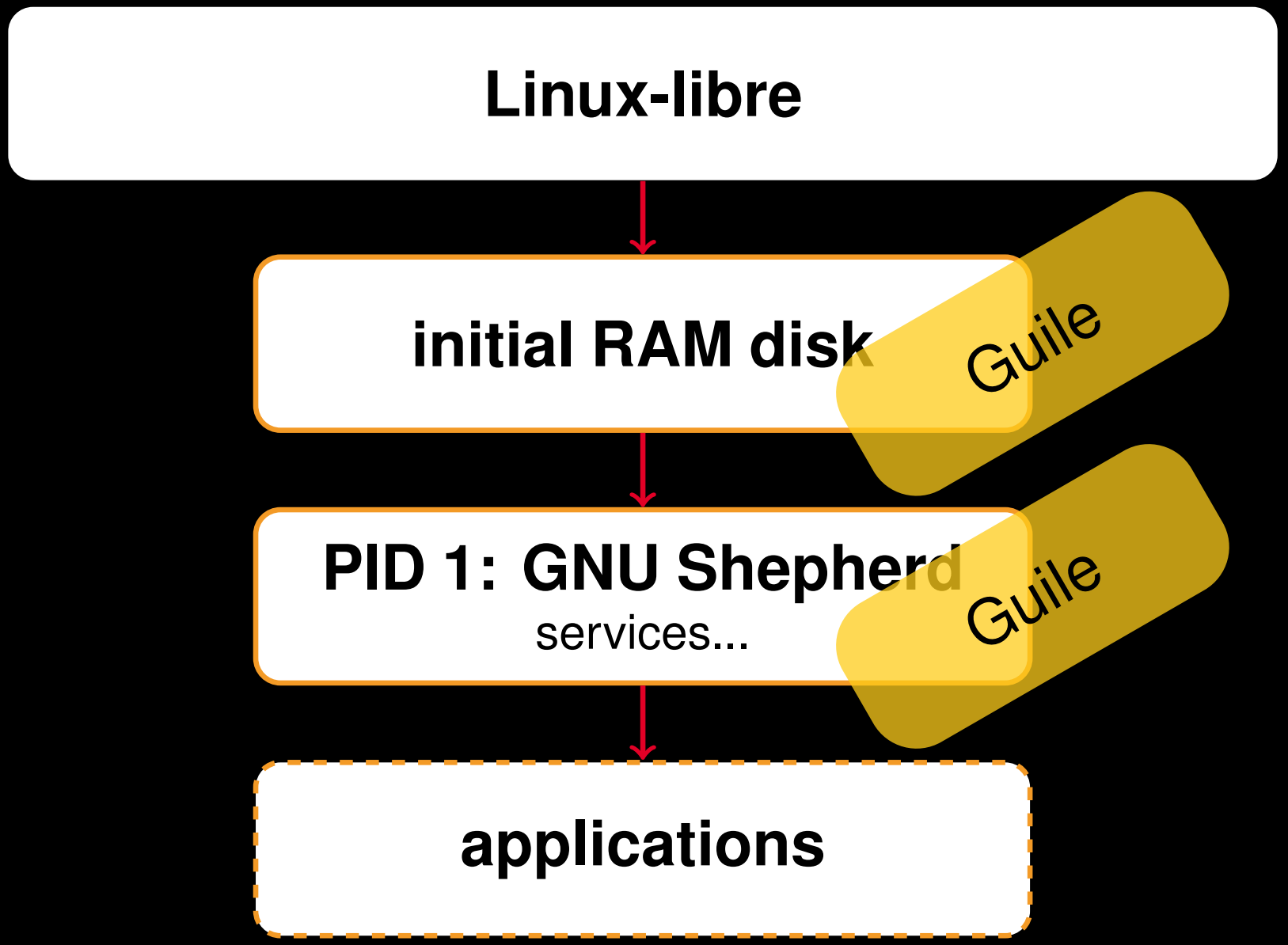
Linux-libre


Linux-libre
initial RAM disk

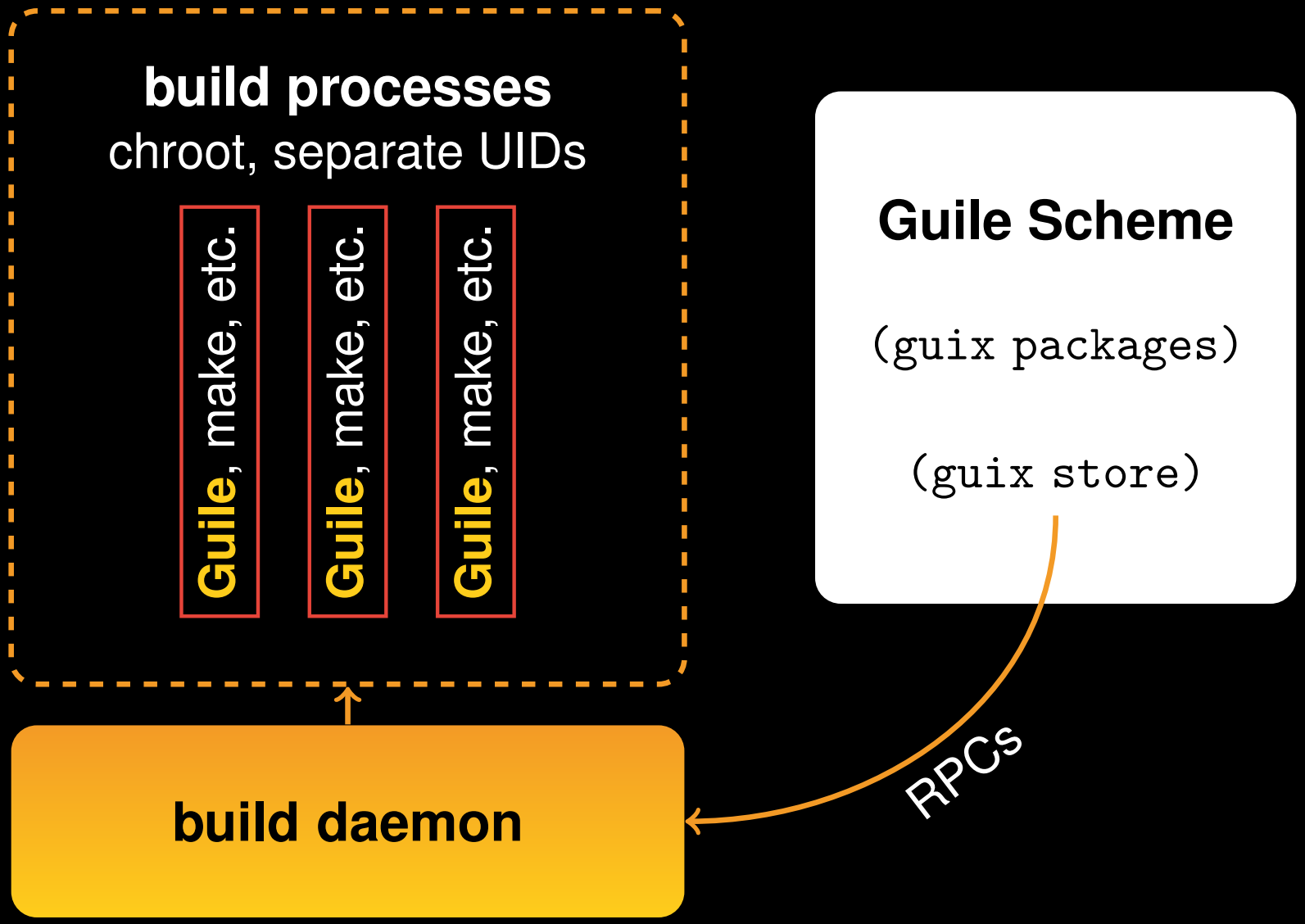

Linux-libre
initial RAM disk
Guile


Linux-libre
initial RAM disk
Guile
PID 1: GNU Shepherd
services...


Linux-libre
initial RAM disk
Guile
PID 1: GNU Shepherd
services...
Guile


Linux-libre
initial RAM disk
Guile
PID 1: GNU Shepherd
services...
Guile
applications

Code staging.


build processes
chroot, separate UIDs
Guile, make, etc.
Guile, make, etc.
Guile, make, etc.
Guile Scheme
(guix packages)
(guix store)
RPCs
build daemon

Staging: take #1

```
(define build-exp
  ;; Build-side code.
  '(symlink "/gnu/store/123...-coreutils-8.25"
            "/gnu/store/abc...-result"))
```

Staging: take #1

```
(define build-exp
  ;; Build-side code.
  '(symlink (assoc-ref %build-inputs "coreutils")
            %output))

;; ... with unhygienic global variable:
;; (define %build-inputs
;;   '(("coreutils" . "/gnu/store/...-coreutils-8.25")))

(define inputs
  ;; What goes into the chroot.
  `(("coreutils" ,coreutils)))

(build-expression->derivation store
                              "symlink-to-coreutils"
                              build-exp
                              #:inputs inputs)
```

Staging: take #1

```
(define build-exp
  ;; Build-side code.
  '(symlink (assoc-ref %build-inputs "coreutils")
            %output))

;; ... with unhygienic global variable:
;; (define %build-inputs
;;   '())
```

```
(build-expression->derivation store
                              "symlink-to-coreutils"
                              build-exp
                                          )
```

Take #2: G-expressions

```
(define build-exp
  ;; First-class object that carries info
  ;; about its dependencies.
  (gexp (symlink (ungexp coreutils)
                 (ungexp output))))

;; Leads to a build script like:
;; (symlink "/gnu/store/123...-coreutils-8.25"
;;          (getenv "out"))

(gexp->derivation "symlink-to-coreutils" build-exp)
```

Take #2: G-expressions

```
(define build-exp
  ;; First-class object that carries info
  ;; about its dependencies.
  #~(symlink #$coreutils #$output))

;; Leads to a build script like:
;; (symlink "/gnu/store/123...-coreutils-8.25"
;;          (getenv "out"))

(gexp->derivation "symlink-to-coreutils" build-exp)
```

Take #2: G-expressions

```
(define build-exp
  ;; First-class object that carries info
  ;; about its dependencies.
  #~(symlink #$coreutils #$output))

;; Leads to a build script like:
;; (symlink "/gnu/store/h8a…-coreutils-8.25"
;;          (getenv "out"))

(gexp->derivation "symlink-to-coreutils" build-exp
                  #:system "i686-linux")
```

Cross-Compilation

```
(gexp->derivation "vi"
   #~(begin
       (mkdir #$output)
       (system* (string-append #+coreutils "/bin/ln")
                "-s"
                (string-append #$emacs "/bin/emacs")
                (string-append #$output "/bin/vi")))
)

;; Yields:
;; (begin
;;   (mkdir (getenv "out"))
;;   (system* (string-append "/gnu/store/123..." "/bin/ln")
;;            "-s"
;;            (string-append "/gnu/store/345..." ...)
;;            (string-append "/gnu/store/567..." ...)))
```

Cross-Compilation

```
(gexp->derivation "vi"
   #~(begin
       (mkdir #$output)
       (system* (string-append #+coreutils "/bin/ln")
                "-s"
                (string-append #$emacs "/bin/emacs")
                (string-append #$output "/bin/vi")))
   #:target "mips64el-linux-gnu")

;; Yields:
;; (begin
;;   (mkdir (getenv "out"))
;;   (system* (string-append "/gnu/store/123..." "/bin/ln")
;;            "-s"
;;            (string-append "/gnu/store/9ab..." ...)
;;            (string-append "/gnu/store/fc2..." ...)))
```

Modules

```
(define build-exp

    #~(begin
        (use-modules (guix build utils))
        (mkdir-p (string-append #$output "/bin"))

(gexp->derivation "empty-bin-dir" build-exp)
;; ERROR: (guix build utils) not found!
```

Modules

```
(define build-exp
  ;; Compile (guix build utils) and add it
  ;; to the chroot.
  (with-imported-modules '((guix build utils))
    #~(begin
        (use-modules (guix build utils))
        (mkdir-p (string-append #$output "/bin")))

(gexp->derivation "empty-bin-dir" build-exp)
```

Modules & Scripts

```
(define script
  (with-imported-modules (source-module-closure
                          '((guix build gremlin)))
    #~(begin
        (use-modules (guix build gremlin)
                     (ice-9 match))

        (match (command-line)
          ((command argument)
           (validate-needed-in-runpath argument))))))

(gexp->script "check-runpath" script)
```

Modules & Initial RAM Disk

```
(expression->initrd
 (with-imported-modules (source-module-closure
                         '((gnu build linux-boot)
                           (guix build utils)))
   #~(begin
       (use-modules (gnu build linux-boot)
                    (guix build utils))

       (boot-system #:mounts '#$file-systems
                    #:linux-modules '#$linux-modules
                    #:linux-module-directory '#$kodir)))
```

Defining “Compilers”

```
(define-gexp-compiler (package-compiler (package <package>)
                                        system target)
  ;; Return a derivation to build PACKAGE.
  (if target
      (package->cross-derivation package target system)
      (package->derivation package system)))
```

Defining “Compilers”

```
(define-gexp-compiler (package-compiler (package <package>)
                                        system target)
  ;; Return a derivation to build PACKAGE.
  (if target
      (package->cross-derivation package target system)
      (package->derivation package system)))


(define-record-type <plain-file>
  (plain-file name content)
  ...)

(define-gexp-compiler (plain-file-compiler (file <plain-file>)
                                           system target)
  ;; "Compile" FILE by adding it to the store.
  (match file
    (($ <plain-file> name content)
     (text-file name content))))
```

Compilers & “Expanders”

```
#~(string-append #$coreutils "/bin/ls")

;; Yields:
;; (string-append "/gnu/store/…" "/bin/ls")
```

Compilers & “Expanders”

```
#~(string-append #$coreutils "/bin/ls")

;; Yields:
;; (string-append "/gnu/store/…" "/bin/ls")

(file-append coreutils "/bin/ls")

;; Yields:
;; "/gnu/store/…/bin/ls"
```

Implementation

- `gexp` macro
- `<gexp>` record type
- `gexp->sexp` linear in the number of `ungexp`

Limitations

- **hygiene**, oh my!
- **modules** in scope?
- **serialization** of non-primitive data types?
- cross-stage **debugging info** à la Hop?

Related Work

syntax-case

- gexps similar in spirit to **syntax objects**
- ... but staging with gexps is not referentially transparent

Writing Hygienic Macros in Scheme with Syntax-Case, R. Kent Dybvig, 1992

MetaScheme

- referentially transparent (“hygienic”) staging
- ... but PoC is simplistic
  - modules in scope?
  - how to determine which forms introduce bindings?

MetaScheme, or untyped MetaOCaml,
`http://okmij.org/ftp/meta-programming/`, O. Kiselyov, 2008

Hop

```
(define-service (shello6 x)
  (<HTML>
    (<BODY>
      :onclick ~(with-hop ($(service ()
                                 (format "Bonjour ~a" x)))
                  (lambda (v) (alert v)))
      "Hello!")))
```

Hop

- staged code is **JavaScript**, not Scheme
- programmers can express **modules in scope** for staged code
- ~ and $ implemented as compiler magic
  - ~ expressions are not first-class objects

A Multi-Tier Semantics for Hop, Serrano and Queinnec, 2010

Nix language

```
derivation {
  name = "foo";
  system = "x86_64-linux";
  builder = "${./static-bash}";
  args = [ "-c" "echo hello > " $out " ];
}
```

Nix language

```
let dep =  derivation  {
  name = "foo";
  system = "x86_64-linux";
  builder = "${./static-bash}";
  args = [ "-c" "echo hello > " $out " ];
 } ; in derivation {
  name = "bar";
  system = "x86_64-linux";
  builder = "${./static-bash}";
  args = [ "-c"
    '' mkdir -p "$out"
       ln -s " ${dep} /some-result" "$out/my-result"
    '' ];
  PATH = "${coreutils}/bin";
}
```

expands to `/nix/store/...-foo`

Nix language

- has **string interpolation**
- strings retain info about their **dependencies**
- built into the interpreter

NixOS: A Purely Functional Linux Distribution, Dolstra and Löh, 2008

NixOS / nixpkgs

Code | Issues 1,215 | Pull requests 317 | Projects 0 | Pulse

lib: Make escapeShellArg more robust

Quoting various characters that the shell *may* interpret specially is a very fragile thing to do.

I've used something more robust all over the place in various Nix expression I've written just because I didn't trust escapeShellArg.

Here is a proof of concept showing that I was indeed right in distrusting escapeShellArg:

Wrap up.

Summary

- Guix provides **functional OS deployment**
- it's a **Scheme library and toolbox**
- it's a **multi-tier Scheme system**

Lots of other niceties!

- **system service** architecture
- ... and services written in Scheme (Shepherd, mcron)
- the **“store monad”**!
- **Emacs** integration (awesome!)
- **whole-system test suite** (staging!)
- **distributed deployment** with Guile-SSH (staging!)
- ...

The First No-Compromise LISP Machine

LMI LAMBDA

Join us now, share the parens!

- **install the distribution**
- **use it**, report bugs, add packages
- share your **ideas**!

GuixSD

ludo@gnu.org

https://gnu.org/software/guix/