Amail - a mail client for Acme

(version 0.10.0)

Alexander Sychev (santucco@gmail.com)

1. Introduction.

`Amail` is a mail client for `Acme` - an editor/window manager/shell. It is supposed to be a replacement for `Mail` - the classic mail client for `Acme`.

For years I was being a user of `Opera` - a web browser with a mail client. But a quality of the web browser of `Opera` was becoming low from a version to a version, so I decided to change the web brower to a `Chromium`, but I didn't find a mail client for my requirements.

Few years ago I saw `Acme` and found it is very simple, but powerful and extremely extensible. Yes, it is not perfect (nothing is perfect), but it is good enough, and I use it like a programming environment (instead of `Emacs`). I had known about `Mail` - a mail client for `Acme`, and a time to try it has come.

I have found `Mail` has some disadvantages (at least for me):

- it doesn't have a support of threads
- it doesn't have a support for read/unread messages
- it doesn't have a navigation through mailboxes
- it has a quite big loading time with big mailboxes.

I also prefer to view some messages in `html`-form (if any) with a possibility to open them in a web browser.

`Amail` is supposed to use with a conjunction with a `upas` - a mail filesystem supports `IMAP4` mail protocol. I'm going to save a compatibility with `Mail` by commands.

For the moment `Amail` is working with `Acme` from `Plan 9 from User Space` (http://swtch.com/plan9port/). I have some doubts `Amail` will work in `Plan9` without changes.

2. Implementation.

```
// This file is part of amail
//
// Copyright (c) 2013, 2014, 2020 Alexander Sychev. All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are
// met:
//
// * Redistributions of source code must retain the above copyright
// notice, this list of conditions and the following disclaimer.
// * Redistributions in binary form must reproduce the above
// copyright notice, this list of conditions and the following disclaimer
// in the documentation and/or other materials provided with the
// distribution.
// * The name of author may not be used to endorse or promote products derived from
// this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
package main
import(
  ⟨ Imports 11 ⟩
)
type(
  ⟨ Types 19 ⟩
)
⟨ Constants 42 ⟩
var(
  ⟨ Variables 3 ⟩
  debug glog.Level = 1
)
func main(){
  glog.V(debug).Infoln("main")
  defer glog.V(debug).Infoln("main is done")
  ⟨ Parse command line arguments 12 ⟩
  ⟨ Try to open mailfs 17 ⟩
  ⟨ Subscribe on notifications of plumber 32 ⟩
  ⟨ Init root of mailfs 25 ⟩
  ⟨ Start a collector of message identifiers 124 ⟩
  if len(flag.Args())>0 {
    ⟨ Start a main message loop 35 ⟩
```

```
    for _, name := range flag.Args() {
      ⟨ Create box 36 ⟩
      ⟨ Start a message loop for box 60 ⟩
      ⟨ Inform box to create a window 74 ⟩
    }
  } else {
    ⟨ Create the main window 41 ⟩
    ⟨ Start a main message loop 35 ⟩
    go func(){⟨ Enumerating of mailboxes 26 ⟩}()
  }
  ⟨ Process events are specific for boxes 34 ⟩
}
```

3. Exiting.
At first we should quit correctly. So a channel *exit* is defined. All goroutines should wait for a message from *exit*.

⟨ Variables 3 ⟩ ≡
```
exit chan bool = make(chan bool)
```
See also sections 6, 10, 16, 20, 23, 40, 85, 91, 110, 120, 157, 221, 230, 234, and 245.
This code is used in section 2.

4.

⟨ On exit? 4 ⟩ ≡
```
case ← exit:
    glog.V(debug).Infoln("on exit!")
    return
```
This code is used in sections 32, 33, 34, 35, 49, 61, and 124.

5.

⟨ Exit! 5 ⟩ ≡
```
glog.V(debug).Infoln("exit!")
close(exit)
```
This code is used in sections 7, 26, and 45.

6. We have to quit when all window of mailboxes and main window are closed. *wcount* contains a count of mailboxes's windows. *wch* is a channel to manipulate of *wcount*. When the main window is closed, the program has to exit immediately.

⟨ Variables 3 ⟩ +≡
```
wch chan int = make(chan int, 100)
wcount int
```

7. When $wcount \equiv 0$, the program quits.

⟨ Processing of other common channels 7 ⟩ ≡
```
case i :=← wch:
    wcount += i
    if wcount ≡ 0 {
        ⟨ Exit! 5 ⟩
        return
    }
```
See also sections 88, 111, and 158.
This code is used in section 34.

8.

⟨ Increase the windows count 8 ⟩ ≡
```
glog.V(debug).Infoln("increase the windows count")
wch ← 1
```
This code is used in sections 41 and 75.

9.

⟨ Decrease the windows count 9 ⟩ ≡
```
glog.V(debug).Infoln("decrease the windows count")
wch ← (−1)
```
This code is used in sections 45 and 78.

10. Parsing command line arguments.

⟨ Variables 3 ⟩ +≡
```
shownew bool
showthreads bool
levelmark string
newmark string
skipboxes []string
```

11.

⟨ Imports 11 ⟩ ≡
```
"flag"
"fmt"
"os"
"strings"
"sort"
```
See also sections 13, 15, 28, 31, 39, 47, 93, 122, 214, 229, and 233.

This code is used in section 2.

12.

⟨ Parse command line arguments 12 ⟩ ≡
```
{
  glog.V(debug).Infoln("parsing␣command␣line␣arguments")
  var skip string
  flag.BoolVar(&shownew, "new", true, "show␣new␣messages␣only")
  flag.BoolVar(&showthreads, "threads", true, "show␣threads␣of␣messages")
  flag.StringVar(&skip, "skip", "", "boxes␣to␣be␣skiped,␣separated␣by␣comma")
  flag.StringVar(&levelmark, "levelmark", "+", "mark␣of␣level␣for␣threads")
  flag.StringVar(&newmark, "newmark", "(*)", "mark␣of␣new␣messages")
  flag.Usage = func(){
    fmt.Fprintf(os.Stderr, "Mail␣client␣for␣Acme␣programming␣environment\n")
    fmt.Fprintf(os.Stderr, "Usage:␣%s␣[options]␣[<mailbox␣1>]...[<mailbox␣N>]\n", os.Args[0])
    fmt.Fprintln(os.Stderr, "Options:")
    flag.PrintDefaults()
  }
  flag.Parse()
  ⟨ Check levelmark and newmark 14 ⟩
  if len(skip)>0 {
    skipboxes = strings.Split(skip, ",")
    for i, v := range skipboxes {
      skipboxes[i] = strings.TrimSpace(v)
    }
    sort.Strings(skipboxes)
    glog.V(debug).Infof("these␣mailboxes␣will␣be␣skipped:␣%v\n", skipboxes)
  }
}
```
This code is used in section 2.

13.

⟨ Imports 11 ⟩ +≡
```
"unicode"
"unicode/utf8"
```

14. *levelmark* shouldn't have ending digit and *newmark* shouldn't have leading digit, because the digits change a message id.

```
⟨ Check levelmark and newmark 14 ⟩ ≡
  glog.V(debug).Infoln("checking␣of␣levelmark␣and␣newmark")
  if r, _ := utf8.DecodeLastRuneInString(levelmark); unicode.IsDigit(r) {
    fmt.Fprintln(os.Stderr, "last␣symbol␣of␣level␣mark␣shouldn't␣be␣a␣digit")
    os.Exit(1)
  }
  if r, _ := utf8.DecodeRuneInString(newmark); unicode.IsDigit(r) {
    fmt.Fprintln(os.Stderr, "first␣symbol␣of␣new␣mark␣shouldn't␣be␣a␣digit")
    os.Exit(1)
  }
```

This code is used in section 12.

15. Mounting of the Acme filesystem.

```
⟨ Imports 11 ⟩ +≡
  "github.com/santucco/goplan9-clone/plan9/client"
  "github.com/golang/glog"
```

16.

```
⟨ Variables 3 ⟩ +≡
  fsys ∗ client.Fsys
  rfid ∗ client.Fid
  srv string = "mail"
```

17.

```
⟨ Try to open mailfs 17 ⟩ ≡
  {
    glog.V(debug).Infoln("try␣to␣open␣mailfs")
    var err error
    if fsys, err = client.MountService(srv); err ≠ nil {
      glog.Errorf("can't␣mount␣mailfs:␣%v\n", err)
      os.Exit(1)
    }
  }
```

This code is used in section 2.

18. Enumeration of mailboxes.

19. Let's make a structure of a mailbox.

```
⟨ Types 19 ⟩ ≡
  mailbox struct{
    name string
    ⟨ Rest of mailbox members 21 ⟩
  }
  mailboxes []*mailbox
  message struct{
    id int
    ⟨ Rest of message members 29 ⟩
  }
  messages []*message
```

See also sections 82, 83, 118, 119, 139, and 215.

This code is used in section 2.

20. *boxes* contains all message boxes are enumerated and sorted

```
⟨ Variables 3 ⟩ +≡
  boxes mailboxes
```

21. The *mailbox* structure has to be extended a bit:

- *all* is a list of all messages in the box;
- *unread* is a list of unread messages in the box;
- *mch* is a channel to manipulate of *all* and *unread*;
- *dch* is a channel to inform the box a message has been deleted.

```
⟨ Rest of mailbox members 21 ⟩ ≡
  all messages
  unread messages
  mch chan int
  dch chan int
```

See also sections 48, 72, 77, 107, 141, 159, 179, and 192.

This code is used in section 19.

22.

```
⟨ Rest of initialization of mailbox 22 ⟩ ≡
  mch: make(chan int, 100),
  dch: make(chan int, 100) ,
```

See also sections 73, 108, 142, and 193.

This code is used in section 36.

23. Four global channels for synchronious mails counting should be defined:

- *mch* is a channel receives info about a message from `plumber`;
- *dch* is a channel receives info about deleted message from `plumber`;
- *bch* is a channel receives info about new boxes;
- *rfch* is a channel receives info about a box should be refreshed in the main window.

```
⟨ Variables 3 ⟩ +≡
  mch = make(chan *struct{
    name string;
    id int
  }, 100)
  dch = make(chan *struct{
    name string;
    id int
  }, 100)
  bch = make(chan string, 10)
  rfch = make(chan *mailbox, 100)
```

24. A slice of enumerated mailboxes should be sorted. A few methods have to be implemented for *mailboxes* to have an ability to sort of them

```
  func (this mailboxes) Len() int{
    return len(this)
  }
  func (this mailboxes) Less(i, j int) bool{
    return this[i].name⟨this[j].name
  }
  func (this mailboxes) Swap(i, j int){
    t := this[i]
    this[i] = this[j]
    this[j] = t
  }
```

25. Here we open the root of mailfs

```
⟨ Init root of mailfs 25 ⟩ ≡
  glog.V(debug).Infoln("initialization␣of␣root␣of␣mailfs")
  var err error
  rfid, err = fsys.Walk(".")
  if err ≠ nil {
    glog.Errorf("can't␣open␣mailfs:␣%v\n", err)
    os.Exit(1)
  }
  defer rfid.Close()
```

This code is used in section 2.

26. Here we read all directory names.

```
⟨ Enumerating of mailboxes 26 ⟩ ≡
  {
    glog.V(debug).Infoln("enumerating␣of␣mailboxes")
    fi, err := rfid.Dirreadall()
    if err ≠ nil {
      glog.Errorf("can't␣read␣mailfs:␣%v\n", err)
      ⟨ Exit! 5 ⟩
      return
    }
    for _, f := range fi {
      if f.Mode & plan9.DMDIR ≡ plan9.DMDIR {
        name := f.Name
        ⟨ Add a mailbox with name 27 ⟩
      }
    }
    glog.V(debug).Infoln("enumerating␣of␣mailboxes␣is␣done")
  }
```

This code is used in section 2.

27. Names of directories are sent in *bch*

```
⟨ Add a mailbox with name 27 ⟩ ≡
  glog.V(debug).Infof("send␣a␣mailbox␣'%s'␣to␣put␣in␣the␣list\n", name)
  bch ← name
```

This code is used in sections 26 and 49.

28. *newMessage* is a method of *mailbox* to fill a message with *id*.

```
⟨ Imports 11 ⟩ +≡
  "io"
  "bufio"
```

29.

```
⟨ Rest of message members 29 ⟩ ≡
  unread bool
  box ∗ mailbox
```

See also sections 64, 94, 121, 194, 201, and 216.

This code is used in section 19.

30. *newMessage* creates *msg* and fills its fields from `"info"` file. `"flags"` are parsed to detect the message is new.

```
func (this ∗ mailbox) newMessage(id int) (msg ∗ message, unread bool, err error){
  glog.V(debug).Infof("newMessage: trying to open '%d/info'\n", id)
  f, err := this.fid.Walk(fmt.Sprintf("%d/info", id))
  if err ≡ nil {
    err = f.Open(plan9.OREAD)
  }
  if err ≠ nil {
    glog.Errorf("can't open to '%s/%d/info': %s\n", this.name, id, err)
    return
  }
  defer f.Close()
  msg = &message{id: id, box: this, ⟨Rest of initialization of message 217⟩}
  b := bufio.NewReader(f)
  unread = true
  glog.V(debug).Infof("newMessage: reading and parsing of a content of '%d/info'\n", id)
  for s, err := b.ReadString('\n'); err ≡ nil; s, err = b.ReadString('\n') {
    if s[len(s) − 1] ≡ '\n' {
      s = s[:len(s) − 1]
    }
    if strings.HasPrefix(s, "flags ") {
      if strings.Index(s, "seen") ≥ 0 {
        unread = false
      }
      continue
    }
    ⟨Read other fields of a message 95⟩
  }
  msg.unread = unread
  return
}
```

31. Subscription on notifications about new messages.

⟨ Imports 11 ⟩ +≡

```
"github.com/santucco/goplumb"
"github.com/santucco/goplan9-clone/plan9"
```

32. Here a subscription on `"seemail"` port of `plumber` is made. The messages is checked for *filetype* ≡ `"mail"` and `"mailtype"` are existing. In case a new mail message we send a name of a mailbox an an id of the message in *mch*, in case of a mail message is deleted - in *dch*.

```
⟨ Subscribe on notifications of plumber 32 ⟩ ≡
  {
    glog.V(debug).Infoln("trying to open 'seemail' plumbing port")
    if sm, err := goplumb.Open("seemail", plan9.OREAD); err ≠ nil {
      glog.Errorf("can't open plumb/seemail: %s\n", err)
    } else {
      sch, err := sm.MessageChannel(0)
      if err ≠ nil {
        glog.Errorf("can't get message channal for plumb/seemail: %s\n", err)
      } else {
        go func(){
          defer sm.Close()
          defer glog.V(debug).Infoln("plumbing goroutine is done")
          for{
            select{
              ⟨ On exit? 4 ⟩
              case m, ok :=← sch:
                if ¬ok {
                  glog.Warningln("it seems plumber has finished")
                  sch = nil
                  return
                }
                glog.V(debug).Infof("a plumbing message has been received: %v\n", m)
                if m.Attr["filetype"] ≠ "mail" {
                  glog.Warningln("attribute 'filetype' is not 'mail'")
                  continue
                }
                v, ok := m.Attr["mailtype"]
                if ¬ok {
                  glog.Warningln("can't find 'mailtype' attribute")
                  continue
                }
                s := strings.TrimLeft(string(m.Data), "Mail/")
                n := strings.LastIndex(s, "/")
                if n ≡ −1 {
                  glog.Warning("can't found a number of message in '%s'\n", s)
                  continue
                }
                num, err := strconv.Atoi(s[n + 1:])
                if err ≠ nil {
                  glog.Error(err)
                  continue
                }
                if v ≡ "new" {
                  glog.V(debug).Infof("'%d' is a new message in the '%s' mailbox\n", num, s[:n])
                  mch ← &struct{
                    name string;
                    id int
                  }{name: s[:n], id: num}
```

```
            } else if v ≡ "delete" {
              glog.V(debug).Infof("'%d' is a deleted message in the '%s' mailbox\n", num,
                  s[:n])
              dch ← &struct{
                name string;
                id int
              }{name: s[:n], id: num}
            }
          }
        }
      }()
    }
  }
}
```

See also section 33.

This code is used in section 2.

33. Here a subscription on `"sendmail"` port of `plumber` is made.

```
⟨ Subscribe on notifications of plumber 32 ⟩ +≡
  {
    glog.V(debug).Infoln("trying to open 'sendmail' plumbing port")
    if sm, err := goplumb.Open("sendmail", plan9.OREAD); err ≠ nil {
      glog.V(debug).Infof("can't open plumb/sendmail: %s\n", err)
    } else {
      sch, err := sm.MessageChannel(0)
      if err ≠ nil {
        glog.Errorf("can't get message channal for plumb/sendmail: %s\n", err)
      } else {
        go func(){
          defer sm.Close()
          defer glog.V(debug).Infoln("plumbing goroutine is done")
          for{
            select{
              ⟨ On exit? 4 ⟩
              case m, ok :=← sch:
                if ¬ok {
                  glog.Warningln("it seems plumber has finished")
                  sch = nil
                  return
                }
                glog.V(debug).Infof("a plumbing message has been received: %v\n", m)
                var msg *message
                ⟨ Create a new message window 238 ⟩
                name := fmt.Sprintf("Amail/New")
                ⟨ Print the name for window w 53 ⟩
                addr := fmt.Sprintf("To: %s\n\n", string(m.Data))
                w.Write([]byte(addr))
                ⟨ Append common signature 250 ⟩
            }
          }
        }()
      }
    }
  }
```

34. The main message loop.

Via *bch* names of new mailboxes are received, the mailboxes is created and processed. Via *mch* and *dch* messages about new and deleted messages are received, the corresponding mailboxes are found and the messages identifiers are send in the corresponding channels of the mailboxes.

```
⟨ Process events are specific for boxes 34 ⟩ ≡
  glog.V(debug).Infoln("process␣events␣are␣specific␣for␣the␣list␣of␣mailboxes")
  for{
    select{
      ⟨ On exit? 4 ⟩
      case name :=← bch:
        ⟨ Continue if the box name should be skiped 54 ⟩
        ⟨ Create box 36 ⟩
        ⟨ Send a signal to refresh all mailboxes 71 ⟩
        ⟨ Start a message loop for box 60 ⟩
      case d :=← mch:
        name := d.name
        ⟨ Looking for a name mailbox, storing an index of the mail box was found in i, continue if not
            found 38 ⟩
        glog.V(debug).Infof("sending␣'%d'␣to␣add␣in␣the␣'%s'␣mailbox\n", d.id, boxes[i].name)
        boxes[i].mch ← d.id
      case d :=← dch:
        name := d.name
        ⟨ Looking for a name mailbox, storing an index of the mail box was found in i, continue if not
            found 38 ⟩
        glog.V(debug).Infof("sending␣'%d'␣to␣delete␣from␣the␣'%s'␣mailbox\n", d.id, boxes[i].name)
        boxes[i].dch ← d.id
        ⟨ Processing of other common channels 7 ⟩
    }
  }
```

This code is used in section 2.

35. This is a message loop for main window. It reads and processes messages from different channels.

A pointer to a mailbox b is received from *rfch*. In case $b \equiv$ **nil** we should print a state of all mailboxes or state of b otherwise.

```
⟨ Start a main message loop 35 ⟩ ≡
  go func(){
    glog.V(debug).Infoln("start a main message loop")
    defer glog.V(debug).Infoln("main message loop is done")
    for{
      select{
        ⟨ On exit? 4 ⟩
        case b :=← rfch:
          if b ≡ nil {
            ⟨ Print all mailboxes 44 ⟩
          } else {
            ⟨ Refresh main window for a box b 46 ⟩
          }
          ⟨ Processing of other channels 45 ⟩
      }
    }
  }()
```

This code is used in section 2.

36.

```
⟨ Create box 36 ⟩ ≡
  glog.V(debug).Infof("creating a '%s' mailbox\n", name)
  box := &mailbox{name: name, ⟨ Rest of initialization of mailbox 22 ⟩}
  boxes = append(boxes, box)
  sort.Sort(boxes)
```

This code is used in sections 2 and 34.

37. *mailboxes.Search* finds a mailbox with *name* and returns a position of the mailbox in the list and **true** or a position where the box can be inserted and **false**

```
  func (this mailboxes) Search(name string) (int, bool){
    pos := sort.Search(len(this),
      func(i int) bool{
        return this[i].name ≥ name
      });
    if pos ≠ len(this) ∧ this[pos].name ≡ name {
      return pos, true
    }
    return pos, false
  }
```

38.

⟨ Looking for a *name* mailbox, storing an index of the mail box was found in i, continue if not found 38 ⟩ ≡

```
glog.V(debug).Infof("looking␣for␣a␣'%s'␣mailbox\n", name)
i, ok := boxes.Search(name)
if ¬ok {
    ⟨ Continue if the box name should be skiped 54 ⟩
    glog.Warningf("can't␣find␣message␣box␣'%s'\n", name)
    continue
}
```

This code is used in sections 34 and 88.

39. The main window.

⟨ Imports 11 ⟩ +≡
```
"github.com/santucco/goacme"
```

40.

⟨ Variables 3 ⟩ +≡
```
mw ∗ goacme.Window
ech ← chan ∗ goacme.Event
```

41.

⟨ Create the main window 41 ⟩ ≡
```
glog.V(debug).Infoln("creating␣the␣main␣window")
defer goacme.DeleteAll()
var err error
if mw, err = goacme.New(); err ≠ nil {
    glog.Errorf("can't␣create␣a␣window:␣%v\n", err)
    os.Exit(1)
}
name := "Amail"
w := mw
⟨ Print the name for window w 53 ⟩
if ech, err = mw.EventChannel(0, goacme.Look | goacme.Execute); err ≠ nil {
    glog.Errorf("can't␣open␣an␣event␣channel␣of␣the␣window␣%v\n", err)
    os.Exit(1)
}
⟨ Write a tag of main window 172 ⟩
⟨ Increase the windows count 8 ⟩
```
This code is used in section 2.

42.

⟨ Constants 42 ⟩ ≡
```
const mailboxfmt = "%-30s\t%10d\t%10d\n"
const mailboxfmtprc = "%-30s\t%10d\t%10d\t%d%%\n"
const wholefile = "0,$"
```
See also sections 84, 140, 164, and 185.

This code is used in section 2.

43.

⟨ Quote name of mailbox if it is necessary 43 ⟩ ≡
```
name := b.name
if strings.IndexFunc(name, unicode.IsSpace) ≠ −1 {
    name = "'" + name + "'"
}
```
This code is used in sections 44 and 46.

44. Here we clean up the main window and print states of all mailboxes.

```
⟨ Print all mailboxes 44 ⟩ ≡
  if mw ≠ nil {
    glog.V(debug).Infoln("printing␣of␣the␣mailboxes")
    if err := mw.WriteAddr(wholefile); err ≠ nil {
      glog.Errorf("can't␣write␣'%s'␣to␣'addr'␣file:␣%s\n", wholefile, err)
    } else if data, err := mw.File("data"); err ≠ nil {
      glog.Errorf("can't␣open␣'data'␣file:␣%s\n", err)
    } else {
      for _, b := range boxes {
        ⟨ Quote name of mailbox if it is necessary 43 ⟩
        if b.total ≡ len(b.all) {
          data.Write([]byte(fmt.Sprintf(mailboxfmt, name, len(b.unread), len(b.all))))
        } else if b.total ≠ 0 ∧ len(b.all) * 100/b.total⟩0 {
          data.Write([]byte(fmt.Sprintf(mailboxfmtprc, name, len(b.unread), len(b.all),
              len(b.all) * 100/b.total)))
        } else {
          data.Write([]byte(fmt.Sprintf(mailboxfmt, name, 0, 0)))
        }
      }
    }
    w := mw
    ⟨ Set window w to clean state 50 ⟩
    ⟨ Go to top of window w 96 ⟩
  }
```

This code is used in section 35.

45. Let's add processing of evens from the main window. Here events from the main window are processed.

```
⟨ Processing of other channels 45 ⟩ ≡
  case ev, ok :=← ech:
    glog.V(debug).Infof("an␣event␣from␣main␣window␣has␣been␣received:␣%v\n", ev)
    if ¬ok {
      ech = nil
      ⟨ Decrease the windows count 9 ⟩
      return
    }
    if (ev.Type & goacme.Execute) ≡ goacme.Execute {
      switch ev.Text {
        case "ShowNew":
          shownew = true
        case "ShowAll":
          shownew = false
        case "ShowThreads":
          showthreads = true
        case "ShowPlain":
          showthreads = false
        case "Del":
          mw.UnreadEvent(ev)
          mw.Close()
          mw = nil
          ⟨ Exit! 5 ⟩
          return
        case "debug":
          debug = 0
          continue
        case "nodebug":
          debug = 1
          continue
        default:
          mw.UnreadEvent(ev)
          continue
      }
      ⟨ Write a tag of main window 172 ⟩
      continue
    } else if (ev.Type & goacme.Look) ≡ goacme.Look {
      name := ev.Text
          // a box name can contain spaces
      if len(ev.Arg)⟩0 {
        name += "␣" + ev.Arg
      }
      name = strings.TrimLeft(strings.TrimRight(strings.TrimSpace(name), "'"), "'")
      if i, ok := boxes.Search(name); ok {
        box := boxes[i]
        ⟨ Inform box to create a window 74 ⟩
        continue
      }
    }
    mw.UnreadEvent(ev)
```

This code is used in section 35.

46. If not all messages are counted, the refresh of state of mailbox in the main window will be processed every percent of counted messages.

```
⟨ Refresh main window for a box b 46 ⟩ ≡
  glog.V(debug).Infof("refreshing␣main␣window␣for␣the␣'%s'␣mailbox,␣len(all):␣%d,␣\
      total:␣%d\n", b.name, len(b.all), b.total)
  if mw ≠ nil {
    if len(b.all) ≠ b.total ∧ b.total/100 ≠ 0 ∧ len(b.all) % (b.total/100) ≠ 0 {
      continue
    }
    ⟨ Quote name of mailbox if it is necessary 43 ⟩
    if err := mw.WriteAddr("0/^%s.*\\n/", escape(name)); err ≠ nil {
      glog.V(debug).Infof("can't␣write␣to␣'addr'␣file:␣%s\n", err)
      continue
    }
    if data, err := mw.File("data"); err ≠ nil {
      glog.Errorf("can't␣open␣'data'␣file:␣%s\n", err)
    } else if len(b.all) ≡ b.total {
      if _, err := data.Write([]byte(fmt.Sprintf(mailboxfmt, name, len(b.unread), len(b.all)))); 
          err ≠ nil {
        glog.Errorf("can't␣write␣to␣'data'␣file:␣%s\n", err)
        continue
      }
      w := mw
      ⟨ Set window w to clean state 50 ⟩
      ⟨ Go to top of window w 96 ⟩
    } else if _, err := data.Write([]byte(fmt.Sprintf(mailboxfmtprc, name, len(b.unread), len(b.all),
          len(b.all) * 100/b.total))); err ≠ nil {
      glog.Errorf("can't␣write␣to␣'data'␣file:␣%s\n", err)
    }
  }
```

This code is used in section 35.

47.

```
⟨ Imports 11 ⟩ +≡
  "strconv"
```

48.

```
⟨ Rest of mailbox members 21 ⟩ +≡
  fid * client.Fid
  total int
```

49. Here messages of a mailbox are counted. If some directories are not numbers, they are supposed to be mailboxes and its names are sent to *bch* to start of counting of the new mailbox. New messages are counted here too. The enumeration of the messages is started from the end of the list, because new messages have bigger numbers. To avoid of unresponding main window the counting is made in **default** branch of **select** .

```
⟨ Count of messages in a box 49 ⟩ ≡
  {
    glog.V(debug).Infof("counting␣of␣messages␣in␣the␣'%s'␣mailbox\n", box.name)
    var err error
    box.fid, err = rfid.Walk(box.name)
    if err ≠ nil {
      glog.Errorf("can't␣walk␣to␣'%s':␣%v", box.name, err)
      return
    }
    defer box.fid.Close()
    fs, err := box.fid.Dirreadall()
    if err ≠ nil {
      glog.Errorf("can't␣read␣a␣mailbox␣'%s':␣%s", box.name, err)
      return
    }
    box.total = len(fs) − 2
    box.all = make(messages, 0, box.total)
    for i := len(fs) − 1; i ≥ 0; {
      select{
        ⟨ On exit? 4 ⟩
        ⟨ Processing of other box channels 62 ⟩
        default:
          d := fs[i]
          i−−
          if (d.Mode & plan9.DMDIR) ≠ plan9.DMDIR {
            continue
          }
          id, err := strconv.Atoi(d.Name)
          if err ≠ nil {      // it seems this is a mailbox
              // decrease a total number of messages
            box.total−−
            name := box.name + "/" + d.Name
            ⟨ Add a mailbox with name 27 ⟩
            continue
          }
          if msg, new, err := box.newMessage(id); err ≡ nil {
            if new {
              ⟨ Add msg to unread 65 ⟩
            }
            ⟨ Add msg to all 66 ⟩
          } else {
            glog.V(debug).Infof("can't␣create␣a␣new␣'%d'␣message␣in␣the␣'%s'␣mailbox:␣%v\n",
                id, box.name, err)
            box.total−−
            continue
          }
          ⟨ Send box to refresh the main window 70 ⟩
```

```
        ⟨ Inform box to print counting messages 162 ⟩
      }
    }
    ⟨ Inform box to print the rest of counting messages 163 ⟩
  }
```

This code is used in section 61.

50.

```
⟨ Set window w to clean state 50 ⟩ ≡
  if w ≠ nil {
    glog.V(debug).Infoln("setting the window to clean state")
    if err := w.WriteCtl("clean"); err ≠ nil {
      glog.Errorf("can't write to 'ctl' file: %s\n", err)
    }
  }
```

This code is used in sections 44, 46, 78, 198, and 204.

51.

```
⟨ Set window w to dirty state 51 ⟩ ≡
  if w ≠ nil {
    glog.V(debug).Infoln("setting the window to dirty state")
    if err := w.WriteCtl("dirty"); err ≠ nil {
      glog.Errorf("can't write to 'ctl' file: %s\n", err)
    }
  }
```

This code is used in section 198.

52. *escape* escapes the regex specific charactets

```
  func escape(s string) (res string){
    for _, v := range s {
      if strings.ContainsRune("\\/[].+?()*^$", v) {
        res += "\\"
      }
      res += string(v)
    }
    return res
  }
```

53. If *name* contains spaces, they will be replaced by underlines.

```
⟨ Print the name for window w 53 ⟩ ≡
  glog.V(debug).Infoln("printing a name for a window")
  if err := w.WriteCtl("name %s", strings.Replace(name, " ", "", -1)); err ≠ nil {
    glog.Errorf("can't write to 'ctl' file: %s\n", err)
  }
```

This code is used in sections 33, 41, 78, 180, 190, 204, and 237.

54.

⟨ Continue if the box *name* should be skiped 54 ⟩ ≡

```
glog.V(debug).Infoln("continue if the box should be skiped")
if i := sort.SearchStrings(skipboxes, name); i ≠ len(skipboxes) ∧ skipboxes[i] ≡ name {
  continue
}
```

This code is used in sections 34 and 38.

55. *messages.Search* finds a message with *id* and returns a position of the message in the list and **true** or a position where the message can be inserted and **false**

```
func (this messages) Search(id int) (int, bool){
  pos := sort.Search(len(this), func(i int) bool{
    return this[i].id ≤ id
  });
  if pos ≠ len(this) ∧ this[pos].id ≡ id {
    return pos, true
  }
  return pos, false
}
```

56. *messages.Insert* inserts a message *msg* in position *pos*

```
func (this ∗ messages) Insert(msg ∗ message, pos int){
  ∗this = append(∗this, nil)
  copy((∗this)[pos + 1:], (∗this)[pos:])
  (∗this)[pos] = msg
}
```

57. *messages.SearchInsert* inserts a message *msg* and returns a position of the message in the list and **true** or a position where the message already exists and **false**

```
func (this ∗ messages) SearchInsert(msg ∗ message) (int, bool){
  pos, ok := this.Search(msg.id)
  if ok {
    return pos, false
  }
  this.Insert(msg, pos)
  return pos, true
}
```

58. *messages.Delete* deletes a message at *pos* position and returns a pointer to the message is removed and **true** if the message is deleted, **false** otherwise

```
func (this ∗ messages) Delete(pos int) (∗message, bool){
  if pos⟨0 ∨ pos⟩len(∗this) − 1 {
    return nil, false
  }
  msg := (∗this)[pos]
  ∗this = append((∗this)[:pos], (∗this)[pos + 1:]...)
  return msg, true
}
```

59. *messages.DeleteById* deletes a message with *id* and returns a pointer to the message is removed and **true** if the message is deleted, **false** otherwise

```
func (this * messages) DeleteById(id int) (*message, bool){
  pos, ok := this.Search(id)
  if ¬ok {
    return nil, false
  }
  return this.Delete(pos)
}
```

60. The message loop for a mailbox.

```
⟨ Start a message loop for box 60 ⟩ ≡
  go box.loop()
```

This code is used in sections 2 and 34.

61.

```
func (box * mailbox) loop(){
  glog.V(debug).Infof("start␣a␣message␣loop␣for␣the␣'%s'␣mailbox\n", box.name)
  counted := false
  pcount := 0
  ontop := false
  ⟨ Count of messages in a box 49 ⟩
  counted = true
  ⟨ Write a tag of box window 181 ⟩
  if box.threadMode() {
    ⟨ Inform box to print messages 146 ⟩
  }
  defer glog.V(debug).Infof("a␣message␣loop␣of␣the␣'%s'␣mailbox␣is␣done\n", box.name)
  for{
    select{
      ⟨ On exit? 4 ⟩
      ⟨ Processing of other box channels 62 ⟩
    }
  }
}
```

62. Here new messages of *box* are processed. If *box* shows a particular thread the message should be printed only if it is in the thread. Also the message should be printed in other boxes with the thread. A new message can be in a thread so we have to send

```
⟨ Processing of other box channels 62 ⟩ ≡
  case id :=← box.mch:
    glog.V(debug).Infof("'%d' should be added to the '%s' mailbox\n", id, box.name)
    msg, new, err := box.newMessage(id)
    if err ≠ nil {
      continue
    }
    if new {
      ⟨ Add msg to unread 65 ⟩
    }
    box.total++
    ⟨ Add msg to all 66 ⟩
    ⟨ Print msg at exact positon 156 ⟩
    if ¬box.thread {
      if box.threadMode() {
        ⟨ Get root of msg 134 ⟩
        var msgs messages
        src := append(messages{}, root)
        ⟨ Make a full thread in msgs for root 147 ⟩
        ⟨ Inform box to print msgs 152 ⟩
      } else {
        ⟨ Inform box to print msg 151 ⟩
      }
    }
    ⟨ Send box to refresh the main window 70 ⟩
```

See also sections 63, 75, 78, 109, 143, and 195.

This code is used in sections 49 and 61.

63. Here deleted messages of *box* are processed.

```
⟨ Processing of other box channels 62 ⟩ +≡
  case id :=← box.dch:
    glog.V(debug).Infof("'%d' should be deleted from the '%s' mailbox\n", id, box.name)
    ⟨ Delete a message with id 67 ⟩
```

64. *deleted* points out the message is marked to delete.

```
⟨ Rest of message members 29 ⟩ +≡
  deleted bool
```

65.

```
⟨ Add msg to unread 65 ⟩ ≡
  {
    glog.V(debug).Infof("addition of the '%d' message to the list of unread messages\
        of the '%s' mailbox\n", msg.id, box.name)
    box.unread.SearchInsert(msg)
  }
```

This code is used in sections 49 and 62.

66.

```
⟨ Add msg to all 66 ⟩ ≡
  {
    glog.V(debug).Infof("addition␣of␣the␣'%d'␣message␣to␣the␣list␣of␣all␣messages␣of\
        ␣the␣'%s'␣mailbox\n", msg.id, box.name)
    box.all.SearchInsert(msg)
  }
```

This code is used in sections 49 and 62.

67.

```
⟨ Delete a message with id 67 ⟩ ≡
  if i, ok := box.all.Search(id); ok {
    msgs := append(messages{}, box.all[i])
    ⟨ Delete a message at position i 68 ⟩
    ⟨ Send deleted msgs 112 ⟩
  }
```

This code is used in section 63.

68. Here we delete a message from *box.all* and *box.unread*, decrease *total* and *deleted* counts, send the message id to clean a thread links and send a signal to refresh the main window.

```
⟨ Delete a message at position i 68 ⟩ ≡
  {
    if msg, ok := box.all.Delete(i); ok {
      glog.V(debug).Infof("deleting␣the␣'%d'␣message␣from␣the␣'%s'␣mailbox\n", msg.id, box.name)
      box.unread.DeleteById(msg.id)
      box.total--
      if msg.deleted {
        box.deleted--
      }
      ⟨ Clean up msg 137 ⟩
      ⟨ Send box to refresh the main window 70 ⟩
    }
  }
```

This code is used in sections 67 and 106.

69.

```
  func (box * mailbox) threadMode() bool{
    return box.thread ∨ box.showthreads ∧ ¬box.shownew
  }
```

70. Here we make a snapshot of *box* state and send it to *rfch*

```
⟨ Send box to refresh the main window 70 ⟩ ≡
  glog.V(debug).Infof("sending␣a␣snapshot␣of␣the␣'%s'␣mailbox␣to␣refresh␣the␣main␣window\n",
      box.name)
  b := *box
  rfch ← &b
```

This code is used in sections 49, 62, 68, 116, and 195.

71.

⟨ Send a signal to refresh all mailboxes 71 ⟩ ≡
```
glog.V(debug).Infoln("sending a signal to refresh all mailboxes")
rfch ← nil
```
This code is used in section 34.

72. Let's add some members to *mailbox*. *shownew* and *showthreads* is a copy of global corresponding flags.
ech is a channel of events from *box*'s window.
w is a *box*'s window.
cch is a channel receives the signal to create *box*'s window.

⟨ Rest of *mailbox* members 21 ⟩ +≡
```
shownew bool
showthreads bool
ech ← chan *goacme.Event
w *goacme.Window
cch chan bool
```

73.

⟨ Rest of initialization of *mailbox* 22 ⟩ +≡
```
shownew: shownew,
showthreads: showthreads,
cch: make(chan bool, 100) ,
```

74.

⟨ Inform *box* to create a window 74 ⟩ ≡
```
glog.V(debug).Infof("inform the '%s' mailbox to create a window\n", box.name)
box.cch ← true
```
This code is used in sections 2 and 45.

75. Here we are waiting for a signal to create *box*'s window and create it.

⟨ Processing of other *box* channels 62 ⟩ +≡
```
case ← box.cch:
    glog.V(debug).Infof("a signal to create the '%s' mailbox window has been received\n",
        box.name)
    if box.w ≡ nil {
        box.shownew = shownew
        box.showthreads = showthreads
        box.thread = false
        ⟨ Create a window for the box 76 ⟩
        ⟨ Inform box to print messages 146 ⟩
        ⟨ Increase the windows count 8 ⟩
    } else {
        glog.V(debug).Infof("a window of the '%s' mailbox already exists, just show it\n",
            box.name)
        box.w.WriteCtl("dot=addr\nshow")
    }
```

76.

⟨ Create a window for the box 76 ⟩ ≡

```
glog.V(debug).Infof("creation␣a␣window␣for␣the␣'%s'␣mailbox\n", box.name)
var err error
if box.w, err = goacme.New(); err ≠ nil {
    glog.Errorf("can't␣create␣a␣window:␣%v\n", err)
    os.Exit(1)
}
if box.ech, err = box.w.EventChannel(0, goacme.Look | goacme.Execute); err ≠ nil {
    glog.Errorf("can't␣open␣an␣event␣channel␣of␣the␣window␣%v\n", err)
    os.Exit(1)
}
⟨ Write a name of box window 180 ⟩
⟨ Write a tag of box window 181 ⟩
```

This code is used in section 75.

77. *thread* flag points out the *box*'s window shows a particular thread of messages.

⟨ Rest of *mailbox* members 21 ⟩ +≡

```
thread bool
```

78. Processing of events from the box's window

```
⟨ Processing of other box channels 62 ⟩ +≡
  case ev, ok :=← box.ech:
    glog.V(debug).Infof("an␣event␣has␣been␣received␣from␣the␣'%s'␣mailbox␣window:␣%v\n",
        box.name, ev)
    if ¬ok {
      box.ech = nil
      continue
    }
    if (ev.Type & goacme.Execute) ≡ goacme.Execute {
      switch ev.Text {
        case "Del":
          ⟨ Clean window-specific stuff 160 ⟩
          box.w.Del(true)
          box.w.Close()
          box.w = nil
          ⟨ Decrease the windows count 9 ⟩
          continue
        case "ShowNew":
          box.thread = false
          box.shownew = true
        case "ShowAll":
          if box.showthreads ∧ ¬counted {
            continue
          }
          box.thread = false
          box.shownew = false
        case "ShowThreads":
          if ¬counted {
            continue
          }
          box.showthreads = true
          box.thread = false
          if box.shownew ≡ true {
            ⟨ Write a tag of box window 181 ⟩
            continue
          }
        case "ShowPlain":
          box.showthreads = false
          box.thread = false
          if box.shownew ≡ true {
            ⟨ Write a tag of box window 181 ⟩
            continue
          }
        case "Thread":
          if ¬counted {
            continue
          }
          var msg *message
          if len(ev.Arg) ≡ 0 {
            ⟨ Get a pointer msg to current message 90 ⟩
          } else if num, err := strconv.Atoi(strings.TrimSpace(ev.Arg)); err ≠ nil {
```

```
            continue
          } else if p, ok := box.all.Search(num); ok {
            msg = box.all[p]
          }
          if msg ≠ nil {
            box.thread = true
            ⟨ Write a tag of box window 181 ⟩
            ⟨ Clean box window 183 ⟩
            ⟨ Clean window-specific stuff 160 ⟩
            ⟨ Inform box to print a full thread with msg 150 ⟩
          }
          continue
        case "Delmesg":
          ⟨ Mark to delete messages 98 ⟩
          continue
        case "UnDelmesg":
          ⟨ Unmark messages 99 ⟩
          continue
        case "Put":
          ⟨ Delete messages 106 ⟩
          continue
        case "Mail":
          var msg *message
          ⟨ Create a new message window 238 ⟩
          name := fmt.Sprintf("Amail/%s/New", box.name)
          ⟨ Print the name for window w 53 ⟩
          ⟨ Append box-specific signature 248 ⟩
          continue
        case "Seen":
          ⟨ Mark messages as seen 113 ⟩
          continue
        case "Search":
          glog.V(debug).Infof("search␣argument:␣'%s'\n", ev.Arg)
          ⟨ Search messages 190 ⟩
          continue
        default:
          box.w.UnreadEvent(ev)
          continue
      }
      ⟨ Write a name of box window 180 ⟩
      ⟨ Write a tag of box window 181 ⟩
      ⟨ Clean box window 183 ⟩
      ⟨ Set window w to clean state 50 ⟩
      ⟨ Clean window-specific stuff 160 ⟩
      ⟨ Inform box to print messages 146 ⟩
      continue
    } else if (ev.Type & goacme.Look) ≡ goacme.Look {
      ⟨ Create msgs 86 ⟩
      if (ev.Type & goacme.Tag) ≡ goacme.Tag {
        s := ev.Text
        ⟨ Read a message number 81 ⟩
      } else {
```

```
        ⟨ Open selected messages 79 ⟩
      }
      if len(msgs) ≠ 0 {
        ⟨ Send msgs for viewing 89 ⟩
        continue
      }
    }
    box.w.UnreadEvent(ev)
```

79. Several messages can be selected to open. The address in *ev* will be inspected instead of *ev.Text*, because a size of the selected messages can be more that 256 symbols. The address will send to `"addr"` file of the *box*'s window and then symbols will be read from `"xdata"` file.

```
⟨ Open selected messages 79 ⟩ ≡
  glog.V(debug).Infof("event:␣%v\n", ev)
  if err := box.w.WriteAddr("#%d,#%d", ev.Begin, ev.End);
  err ≠ nil { glog.Errorf("can't␣write␣to␣'addr':␣%s\n", err)
  } else ⟨ Read message numbers 80 ⟩
```

This code is used in section 78.

80.

```
⟨ Read message numbers 80 ⟩ ≡
  if xdata, err := box.w.File("xdata"); err ≠ nil {
    glog.Errorf("can't␣open␣'xdata'␣file:␣%s\n", err)
  } else {
    b := bufio.NewReader(xdata)
    for s, err := b.ReadString('\n'); err ≡ nil ∨ err ≡ io.EOF; s, err = b.ReadString('\n') {
      ⟨ Read a message number 81 ⟩
      if err ≡ io.EOF {
        break
      }
    }
  }
```

This code is used in sections 79 and 97.

81. A message path can contain not only a number but a mailbox name too. So we have to parse an input string to separate the name and the number. In any case the message will be opened via the main loop.

```
⟨ Read a message number 81 ⟩ ≡
  {
    glog.V(debug).Infof("looking␣a␣message␣number␣in␣'%s'\n", s)
    s = strings.TrimLeft(s, levelmark + deleted)
    f := strings.Split(s, "/")
    glog.V(debug).Infof("parts␣of␣message␣path:␣'%v'\n", f)
    id := 0
    for i, v := range f {
      var err error
      if id, err = strconv.Atoi(strings.TrimRight(v, newmark)); err ≡ nil {
        name := box.name
        if i⟩0 {
          name = strings.Join(f[:i], "/")
          glog.V(debug).Infof("the␣message␣number␣is␣'%d'␣in␣the␣'%s'␣mailbox\n", id, name)
        }
        ⟨ Add a id message to msgs 87 ⟩
        break
      }
    }
  }
```

This code is used in sections 78, 80, and 97.

82.

```
⟨ Types 19 ⟩ +≡
  msgmap map[string][]int
```

83. Let's do an universal approach to handle actions with groups of messages instead of making a separate channel for every action.

```
⟨ Types 19 ⟩ +≡
  action int
```

84.

```
⟨ Constants 42 ⟩ +≡
  const(
    view action = iota
    ⟨ Other kinds of actions 100 ⟩
  )
```

85. A channel to do actions with groups of messages.

```
⟨ Variables 3 ⟩ +≡
  ach = make(chan *struct{
    m msgmap;
    a action
  }, 100)
```

86.

```
⟨ Create msgs 86 ⟩ ≡
  msgs := make(msgmap)
```

This code is used in sections 78, 97, and 212.

87.

```
⟨ Add a id message to msgs 87 ⟩ ≡
  glog.V(debug).Infof("adding␣the␣'%d'␣of␣the␣'%s'␣mailbox\n", id, name)
  msgs[name] = append(msgs[name], id)
```

This code is used in sections 81 and 212.

88. Let's add a processing of *ach* to the main thread

```
⟨ Processing of other common channels 7 ⟩ +≡
  case d :=← ach:
    if d.m ≡ nil {
      continue
    }
    for name, ids := range d.m {
      ⟨ Looking for a name mailbox, storing an index of the mail box was found in i, continue if not
          found 38 ⟩
      boxes[i].ach ← &struct{
        ids []int;
        a action
      }{ids, d.a}
    }
```

89.

```
⟨ Send msgs for viewing 89 ⟩ ≡
  ach ← &struct{
    m msgmap;
    a action
  }{msgs, view}
```

This code is used in sections 78 and 212.

90.

⟨ Get a pointer *msg* to current message 90 ⟩ ≡

```
glog.V(debug).Infof("getting␣a␣pointer␣to␣current␣message␣in␣the␣window␣of␣the␣'\
    %s'␣mailbox\n", box.name)
num := 0
if err := box.w.WriteCtl("addr=dot"); err ≠ nil {
  glog.Errorf("can't␣write␣to␣'ctl':␣%s\n", err)
} else if err := box.w.WriteAddr("-/^/"); err ≠ nil {
  glog.V(debug).Infof("can't␣write␣to␣'addr':␣%v\n", err)
} else if err := box.w.WriteAddr("/[0-9]+(%s)?\\//", escape(newmark)); err ≠ nil {
  glog.V(debug).Infof("can't␣write␣to␣'addr':␣%s\n", err)
} else if data, err := box.w.File("data"); err ≠ nil {
  glog.Errorf("can't␣open␣'data'␣file:␣%s\n", err)
} else if str, err := bufio.NewReader(data).ReadString('/'); err ≠ nil {
  glog.Errorf("can't␣read␣from␣'data'␣file:␣%s\n", err)
} else if _, err := fmt.Sscanf(strings.TrimLeft(str, levelmark), "%d", &num); err ≡ nil {
  glog.V(debug).Infof("current␣message:␣%d(%s)\n", num, str)
  if p, ok := box.all.Search(num); ok {
    msg = box.all[p]
  }
} else {
  glog.V(debug).Infof("can't␣get␣a␣current␣message␣from:␣%s\n", str)
}
```

This code is used in section 78.

91.

⟨ Variables 3 ⟩ +≡

```
deleted = "(deleted)-"
```

92.

```
⟨ Compose a header of msg 92 ⟩ ≡
  glog.V(debug).Infof("compose␣a␣header␣of␣the␣'%d'␣message␣of␣the␣'%s'␣mailbox\n", msg.id,
      box.name)
  buf = append(buf, fmt.Sprintf("%s%s%d%s/\t%s\t%s\n\t%s\n",
    func() string{
      if msg.deleted {
        return deleted
      };
      return ""
    }(),
    func() string{
      if msg.box ≠ box {
        return fmt.Sprintf("%s/", msg.box.name)
      };
      return ""
    }(),
    msg.id,
    func() string{
      if msg.unread {
        return newmark
      };
      return ""
    }(),
    msg.from,
    msg.date,
    msg.subject)...)
```

This code is used in section 168.

93.

```
⟨ Imports 11 ⟩ +≡
  "time"
```

94.

```
⟨ Rest of message members 29 ⟩ +≡
  from string
  date time.Time
  subject string
```

95.

```
⟨ Read other fields of a message 95 ⟩ ≡
  if strings.HasPrefix(s, "from␣") {
    msg.from = s[len("from␣"):]
    msg.from = strings.Replace(msg.from, "''␣", "", −1)
    continue
  }
  var unixdate int64
  if _, err := fmt.Sscanf(s, "unixdate␣%d", &unixdate); err ≡ nil {
    msg.date = time.Unix(unixdate, 0)
    continue
  }
  if strings.HasPrefix(s, "subject␣") {
    msg.subject = s[len("subject␣"):]
    continue
  }
```

See also sections 123 and 202.

This code is used in section 30.

96.

```
⟨ Go to top of window w 96 ⟩ ≡
  glog.V(debug).Infoln("go␣to␣top␣of␣the␣window")
  w.WriteAddr("#0")
  w.WriteCtl("dot=addr\nshow")
```

This code is used in sections 44, 46, 167, 204, and 251.

97.

```
⟨ Get numbers of selected messages 97 ⟩ ≡
  ⟨ Create msgs 86 ⟩
  if (ev.Type & goacme.Tag) ≡ goacme.Tag ∧ len(ev.Arg)⟩0 { s := ev.Arg
  ⟨ Read a message number 81 ⟩
  } else if err := box.w.WriteCtl("addr=dot");
  err ≠ nil { glog.Errorf("can't␣write␣to␣'ctl':␣%s\n", err)
  } else ⟨ Read message numbers 80 ⟩
```

This code is used in sections 98, 99, and 113.

98.

```
⟨ Mark to delete messages 98 ⟩ ≡
  ⟨ Get numbers of selected messages 97 ⟩
  if len(msgs) ≠ 0 {
    ⟨ Send msgs to delete 101 ⟩
    continue
  }
```

This code is used in section 78.

99.

```
⟨ Unmark messages 99 ⟩ ≡
  ⟨ Get numbers of selected messages 97 ⟩
  if len(msgs) ≠ 0 {
    ⟨ Send msgs to undelete 102 ⟩
    continue
  }
```

This code is used in section 78.

100.

```
⟨ Other kinds of actions 100 ⟩ ≡
  del
  undel
```

See also section 114.

This code is used in section 84.

101.

```
⟨ Send msgs to delete 101 ⟩ ≡
  glog.V(debug).Infoln("sending␣messages␣to␣mark␣for␣deletion")
  ach ← &struct{
    m msgmap;
    a action
  }{msgs, del}
```

This code is used in section 98.

102.

```
⟨ Send msgs to undelete 102 ⟩ ≡
  glog.V(debug).Infoln("sending␣messages␣to␣unmark␣for␣deletion")
  ach ← &struct{
    m msgmap;
    a action
  }{msgs, undel}
```

This code is used in section 99.

103.

```
⟨ Handling of other types of action 103 ⟩ ≡
  case del:
    var msgs messages
  for _, id := range d.ids{⟨ Mark to delete id message 104 ⟩}
  ⟨ Refresh msgs 155 ⟩
  case undel:
    var msgs messages
  for _, id := range d.ids{⟨ Unmark to delete id message 105 ⟩}
  ⟨ Refresh msgs 155 ⟩
```

See also section 116.

This code is used in section 195.

104.

```
⟨ Mark to delete id message 104 ⟩ ≡
  if p, ok := box.all.Search(id); ok {
    if box.all[p].deleted {
      continue
    }
    box.all[p].deleted = true
    box.deleted++
    msgs = append(msgs, box.all[p])
    if box.all[p].w ≠ nil {
      msg := box.all[p]
      ⟨ Write a tag of message window 205 ⟩
    }
    glog.V(debug).Infof("the␣'%v'␣message␣is␣marked␣for␣deletion\n", id)
  }
```

This code is used in section 103.

105.

```
⟨ Unmark to delete id message 105 ⟩ ≡
  if p, ok := box.all.Search(id); ok {
    if ¬box.all[p].deleted {
      continue
    }
    box.all[p].deleted = false
    box.deleted--
    msgs = append(msgs, box.all[p])
    if box.all[p].w ≠ nil {
      msg := box.all[p]
      ⟨ Write a tag of message window 205 ⟩
    }
    glog.V(debug).Infof("the␣'%v'␣message␣is␣unmarked␣for␣deletion\n", id)
  }
```

This code is used in section 103.

106. Here is processing a final deletion of messages from `mailfs`. Any message could be printed in other mailboxes in threads, so we collect messages in *msgs* and send *msgs* to all mailboxes.

```
⟨ Delete messages 106 ⟩ ≡
  f, err := box.fid.Walk("ctl")
  if err ≡ nil {
    err = f.Open(plan9.OWRITE)
  }
  if err ≠ nil {
    glog.Errorf("can't␣open␣'ctl':␣%v\n", err)
    continue
  }
  var msgs messages
  for i := 0; i⟨len(box.all); {
    if ¬box.all[i].deleted ∨ box.all[i].w ≠ nil {
      i++
      continue
    }
    msgs = append(msgs, box.all[i])
    ⟨ Delete a message at position i 68 ⟩
  }
  dmsgs := msgs
  for{
    cmd := fmt.Sprintf("delete␣%s", box.name)
    c := len(dmsgs)
    if c ≡ 0 {
      break
    } else if c⟩50 {
      c = 50
    }
    for _, msg := range dmsgs[:c] {
      cmd = fmt.Sprintf("%s␣%d␣", cmd, msg.id)
    }
    glog.V(debug).Infof("command␣to␣delete␣messages:␣'%s'\n", cmd)
    if _, err := f.Write([]byte(cmd)); err ≠ nil {
      glog.Errorf("can't␣delete␣messages:␣%v\n", err)
    }
    dmsgs = dmsgs[c:]
  }
  ⟨ Send deleted msgs 112 ⟩
  f.Close()
```

This code is used in section 78.

107. *mdch* is a channel receives slices of messages to delete.

```
⟨ Rest of mailbox members 21 ⟩ +≡
  mdch chan messages
```

108.

```
⟨ Rest of initialization of mailbox 22 ⟩ +≡
  mdch: make(chan messages, 100) ,
```

109. All messages from a received slice m will be removed from *box*'s window. In case of the thread mode *children* is obtained and refreshed.

```
⟨ Processing of other box channels 62 ⟩ +≡
  case m :=← box.mdch:
    if box.w ≡ nil {
      continue
    }
    glog.V(debug).Infof("%d␣messages␣were␣received␣to␣be␣deleted␣from␣the␣'%s'␣mailbox\n",
        len(m), box.name)
    for _, msg := range m {
      ⟨ Erase the message 199 ⟩
      if box.threadMode() {
        ⟨ Get children for msg 129 ⟩
        ⟨ Refresh children 171 ⟩
      }
    }
    ⟨ Check for a clean state of the box's window 198 ⟩
```

110. One message can be presented in multiple boxes, so we have to delete messages from all boxes. *mdch* is a channel to receive signals to delete messages.

```
⟨ Variables 3 ⟩ +≡
  mdch chan messages = make(chan messages, 100)
```

111.

```
⟨ Processing of other common channels 7 ⟩ +≡
  case msgs :=← mdch:
    for i, _ := range boxes {
      glog.V(debug).Infof("sending␣%d␣messages␣to␣delete␣in␣the␣'%s'␣mailbox\n", len(msgs),
          boxes[i].name)
      boxes[i].mdch ← append(messages{}, msgs ...)
    }
```

112.

```
⟨ Send deleted msgs 112 ⟩ ≡
  mdch ← msgs
```

This code is used in sections 67 and 106.

113.

```
⟨ Mark messages as seen 113 ⟩ ≡
  ⟨ Get numbers of selected messages 97 ⟩
  if len(msgs) ≠ 0 {
    ⟨ Send msgs to mark them seen 115 ⟩
    continue
  }
```

This code is used in section 78.

114.

```
⟨ Other kinds of actions 100 ⟩ +≡
  seen
```

115.

⟨ Send *msgs* to mark them seen 115 ⟩ ≡

```
glog.V(debug).Infoln("sending␣messages␣to␣mark␣them␣seen")
ach ← &struct{
  m msgmap;
  a action
}{msgs, seen}
```

This code is used in section 113.

116. We open `"ctl"` file of the window and write the command `"read"` with *id* of messages. To avoid of reaching of limits we break the full list down into pieces by 50 messages.

```
⟨ Handling of other types of action 103 ⟩ +≡
  case seen:
    f, err := box.fid.Walk("ctl")
    if err ≡ nil {
      err = f.Open(plan9.OWRITE)
    }
    if err ≠ nil {
      glog.Errorf("can't open 'ctl' file of the '%s' messagebox: %v\n", box.name, err)
      continue
    }
    for{
      c := len(d.ids)
      if c ≡ 0 {
        break
      } else if c⟩50 {
        c = 50
      }
      ids := d.ids[0:c]
      d.ids = d.ids[c:]
      var ms messages
      for _, id := range ids {
        p, ok := box.all.Search(id)
        if ¬ok ∨ ¬box.all[p].unread {
          continue
        }
        ms = append(ms, box.all[p])
      }
      cmd := "read"
      for _, v := range ms {
        cmd += fmt.Sprintf(" %d", v.id)
      }
      if _, err := f.Write([]byte(cmd)); err ≠ nil {
        glog.Errorf("can't write to 'ctl' file of the '%s' messagebox: %v\n", box.name, err)
      }
      var msgs messages
      for _, msg := range ms {
        id := msg.id
        ⟨ Remove id message from unread 196 ⟩
        ⟨ Refresh the message's view 197 ⟩
      }
      ⟨ Send box to refresh the main window 70 ⟩
      ⟨ Refresh msgs 155 ⟩
    }
    f.Close()
```

117. Linking of threads.
Here we define global map of unique message identifiers on a pointer to messages, its parents and children. An unique id of every message will be stored in this map. It will be changed in a separated goroutine, so a corresponding channel *idch* is defined too. *idch* operates with a pair contains a message *msg* and *val* **interface**{}. We can use *val* with different types of data to do different operations with *idmap*. In case of an addition of a new message the *val* is set to **chan bool**. The channel is used to inform the sender the message has been added successfully. In case of a deletion of the message *val* is set to **nil**. To get children of the message we set *val* to a channel of *idmessages*. To get a parent of the message we set *val* to a channel of *parentmsg*. To get a root of a thread with the message we set *val* to a channel of *rootmsg*. To get a level of the message in the thread we set *val* to a channel of **int**.

118.

```
⟨ Types 19 ⟩ +≡
  idmessages []*message
  rootmsg *message
  parentmsg *message
```

119. *idlinks* struct contains *msgs* slice of messages with the same *messageid*, a pointer to *idlinks* of parent and *children* slice of pointers to *idlinks* of children.

```
⟨ Types 19 ⟩ +≡
  idlinks struct{
    msgs messages
    parent *idlinks
    children []*idlinks
  }
```

120.

```
⟨ Variables 3 ⟩ +≡
  idmap = make(map[string]*idlinks)
  idch = make(chan struct{
    msg *message;
    val interface{}
  }, 100)
```

121.

```
⟨ Rest of message members 29 ⟩ +≡
  inreplyto string
  messageid string
```

122.

```
⟨ Imports 11 ⟩ +≡
  "errors"
```

123. Here we read *inreplyto* and *messageid*. When *messageid* is obtained, we send *msg* to *idch*. We assume *inreplyto* is already filled, because it is placed above of *messageid* in the `"info"` file.

```
⟨ Read other fields of a message 95 ⟩ +≡
  {
    if _, err := fmt.Sscanf(s, "inreplyto␣%s", &msg.inreplyto); err ≡ nil {
      msg.inreplyto = strings.Trim(msg.inreplyto, "<>")
      continue
    }
    if _, err := fmt.Sscanf(s, "messageid␣%s", &msg.messageid); err ≡ nil {
      msg.messageid = strings.Trim(msg.messageid, "<>")
      ch := make(chan bool)
      idch ← struct{
        msg *message;
        val interface{}
      }{msg, ch}
      if ok :=← ch; ¬ok {
        return nil, false, errors.New(fmt.Sprintf("a␣message␣'%s'␣is␣duplicated", msg.messageid))
      }
      continue
    }
  }
```

124. Processing of *idch* in a separated goroutine..

```
⟨ Start a collector of message identifiers 124 ⟩ ≡
  go func(){
    for{
      select{
        ⟨ On exit? 4 ⟩
        case v :=← idch:
          if v.val ≡ nil {
            ⟨ Clean an entry with v.msg.messageid from idmap 126 ⟩
          } else if ch, ok := v.val.(chan bool); ok {
            ⟨ Append a message with v.msg.messageid to idmap 125 ⟩
          } else if ch, ok := v.val.(chan idmessages); ok {
            ⟨ Send children 128 ⟩
          } else if ch, ok := v.val.(chan parentmsg); ok {
            ⟨ Send parent 130 ⟩
          } else if ch, ok := v.val.(chan rootmsg); ok {
            ⟨ Send root 133 ⟩
          } else if ch, ok := v.val.(chan int); ok {
            ⟨ Send level 135 ⟩
          }
      }
    }
  }()
```

This code is used in section 2.

125. When *msg* is appended we should check if *v.id* already exists. It can exist if there are duplicated messages or there are children for this *v.id*. For the latter case an entry is added to *idmap* with empty *msgs*. Later, when a message with *v.id* is added, we just add the new *msg* in *msgs*.

If *msg* has *inreplyto* is filled, we add *val* pointer to *children* of *msg.inreplyto* message and set a parent for *val*.

```
⟨ Append a message with v.msg.messageid to idmap 125 ⟩ ≡
  {
    glog.V(debug).Infof("appending␣a␣'%s'␣('%s/%d')␣message␣to␣idmap\n", v.msg.messageid,
        v.msg.box.name, v.msg.id)
    val, ok := idmap[v.msg.messageid]
    if ¬ok {
      glog.V(debug).Infof("'%s'␣('%s/%d')␣message␣␣doesn't␣exist,␣creating\n",
          v.msg.messageid, v.msg.box.name, v.msg.id)
      val = new(idlinks)
      idmap[v.msg.messageid] = val
    }
    if len(val.msgs)⟩0 {
      glog.V(debug).Infof("%v(%v)␣is␣a␣duplicate␣of␣%v(%v)\n", v.msg.id, v.msg.messageid,
          val.msgs[0].id, val.msgs[0].messageid)
    }
    val.msgs = append(val.msgs, v.msg)
    if len(v.msg.inreplyto) ≠ 0 ∧ len(val.msgs) ≡ 1 {
      pval, ok := idmap[v.msg.inreplyto]
      if ¬ok {
        pval = new(idlinks)
        idmap[v.msg.inreplyto] = pval
      }
          // to avoid reverence to itself
      if val ≠ pval {
        pval.children = append(pval.children, val)
        val.parent = pval
      }
    }
    ch ← true
  }
```

This code is used in section 124.

126. When we are removing a message, we just remove *msg* from *msgs* slice. But if *msgs* slice is empty, we have to clean pointer to the entry in a parent and chindren entries. We leave a non-empty entry in *idmap* to store links of children.

```
⟨ Clean an entry with v.msg.messageid from idmap 126 ⟩ ≡
  {
    val, ok := idmap[v.msg.messageid]
    if ¬ok {
      continue
    }
    for i, _ := range val.msgs {
      if val.msgs[i] ≡ v.msg {
        val.msgs.Delete(i)
        break
      }
    }
    if len(val.msgs)⟩0 {
      continue
    }
    if val.parent ≠ nil {
      for i, _ := range val.parent.children {
        if val.parent.children[i] ≡ val {
          val.parent.children = append(val.parent.children[:i], val.parent.children[i + 1:] ...)
          break
        }
      }
    }
    for _, ch := range val.children {
      ch.parent = nil
    }
    if len(val.children) ≡ 0 {
      delete(idmap, v.msg.messageid)
    }
  }
```

This code is used in section 124.

127. A few methods have to be implemented for *idmessages* to have an ability to sort of them in order of increasing of date.

```
  func (this idmessages) Len() int{
    return len(this)
  }
  func (this idmessages) Less(i, j int) bool{
    return this[i].date.Unix()⟨this[j].date.Unix()
  }
  func (this idmessages) Swap(i, j int){
    t := this[i]
    this[i] = this[j]
    this[j] = t
  }
```

128. If there is *v.msg.messageid* in *idmap*, we fill *children* with correspinding children and sort them in order of increasing of date

```
⟨ Send children 128 ⟩ ≡
  {
    if m, ok := idmap[v.msg.messageid]; ok {
      var children idmessages
      for _, val := range m.children {
        ⟨ Get mgs with the same box like v.msg.box or the first one 131 ⟩
        if msg ≠ nil {
          children = append(children, msg)
        }
      }
      sort.Sort(children)
      glog.V(debug).Infof("sending␣%d␣children␣for␣'%s'\n", len(children), v.msg.messageid)
      ch ← children
    } else {
      glog.V(debug).Infof("'%s'␣is␣not␣found\n", v.msg.messageid)
      ch ← nil
    }
  }
```

This code is used in section 124.

129.

```
⟨ Get children for msg 129 ⟩ ≡
  var children idmessages
  {
    ch := make(chan idmessages)
    glog.V(debug).Infof("getting␣children␣for␣'%s'\n", msg.messageid)
    idch ← struct{
      msg *message;
      val interface{}
    }{msg, ch}
    children =← ch
  }
```

This code is used in sections 109, 149, 189, 207, 209, 210, and 212.

130.

```
⟨ Send parent 130 ⟩ ≡
  {
    if val, ok := idmap[v.msg.messageid]; ¬ok {
      glog.V(debug).Infof("'%s' is not found\n", v.msg.messageid)
      ch ← nil
    } else if val.parent ≡ nil ∨ len(val.parent.msgs) ≡ 0 {
      glog.V(debug).Infof("'%s' hasn't got a parent\n", v.msg.messageid)
      ch ← nil
    } else {
      val = val.parent
      ⟨ Get mgs with the same box like v.msg.box or the first one 131 ⟩
      if msg ≠ nil {
        glog.V(debug).Infof("sending parent '%s' for '%s'\n", msg.messageid, v.msg.messageid)
      }
      ch ← msg
    }
  }
```

This code is used in section 124.

131. We are looking for a message with the same *box*.

```
⟨ Get mgs with the same box like v.msg.box or the first one 131 ⟩ ≡
  var msg *message
  if val ≠ nil ∧ len(val.msgs)⟩0 {
    msg = val.msgs[0]
    for i, _ := range val.msgs {
      if val.msgs[i].box ≡ v.msg.box {
        msg = val.msgs[i]
        break
      }
    }
  }
```

This code is used in sections 128, 130, and 133.

132.

```
⟨ Get parent for msg 132 ⟩ ≡
  var parent *message
  {
    ch := make(chan parentmsg)
    glog.V(debug).Infof("getting parent for '%s'\n", msg.messageid)
    idch ← struct{
      msg *message;
      val interface{}
    }{msg, ch}
    parent =← ch
  }
```

This code is used in sections 189, 207, 209, 210, and 212.

133.

```
⟨ Send root 133 ⟩ ≡
  {
    if val, ok := idmap[v.msg.messageid]; ok {
      for val.parent ≠ nil ∧ len(val.parent.msgs)⟩0 {
        val = val.parent
      }
      ⟨ Get mgs with the same box like v.msg.box or the first one 131 ⟩
      if msg ≡ nil {
        msg = v.msg
      }
      ch ← rootmsg(msg)
    } else {
      glog.V(debug).Infof("'%s' is not found\n", v.msg.messageid)
      ch ← nil
    }
  }
```

This code is used in section 124.

134.

```
⟨ Get root of msg 134 ⟩ ≡
  root := msg
  {
    ch := make(chan rootmsg)
    glog.V(debug).Infof("getting root for '%s' ('%s/%d')\n", msg.messageid, msg.box.name, msg.id)
    idch ← struct{
      msg *message;
      val interface{}
    }{msg, ch}
    root =← ch
  }
```

This code is used in sections 62, 146, and 150.

135.

```
⟨ Send level 135 ⟩ ≡
  {
    if val, ok := idmap[v.msg.messageid]; ok {
      level := 0
      for val.parent ≠ nil ∧ len(val.parent.msgs)⟩0 {
        val = val.parent
        level++
      }
      glog.V(debug).Infof("sending level '%d' for '%s' ('%s/%d')\n", level, v.msg.messageid,
          v.msg.box.name, v.msg.id)
      ch ← level
    } else {
      glog.V(debug).Infof("'%s' is not found\n", v.msg.messageid)
      ch ← 0
    }
  }
```

This code is used in section 124.

136.

```
⟨ Get level for msg 136 ⟩ ≡
  var level int
  {
    ch := make(chan int)
    glog.V(debug).Infof("getting␣root␣for␣'%s'\n", msg.messageid)
    idch ← struct{
      msg *message;
      val interface{}
    }{msg, ch}
    level =← ch
  }
```

This code is used in section 170.

137. Here we send *msg* to *idch* with **nil** like a *val* to clean up a thread links.

```
⟨ Clean up msg 137 ⟩ ≡
  glog.V(debug).Infof("cleaning␣up␣the␣'%d'␣message\n", msg.id)
  if msg ≠ nil {
    idch ← struct{
      msg *message;
      val interface{}
    }{msg: msg}
  }
```

This code is used in section 68.

138. Printing of messages.
Printing of the messages is a kind of trick. To avoid of locks of *box*'s stuff the print is produced in the *box*'s message loop. *rfch* is a channel receives slice of messages have to be printed and flag to seek a position to start a print or to print in the end. A data from *rfch* is redirected to an internal channel *irfch*. A slice of messages is sent to *rfch*, the *box*'s message loop reads the slice and print at most 100 messages, then resend the rest to *rfch*. If we need to stop printing of messages, we drop the rest of a printing queue by recreation of *irfch*.

139. *refresh* holds flags point out how to print *msgs*

```
⟨ Types 19 ⟩ +≡
  refreshFlags int
  refresh struct{
    flags refreshFlags
    msgs messages
  }
```

140. *seek* means a position of the message should be determinated, *insert* means the message should be inserted if the position is not found, *exact* means the message should be inserted only if its exact position is found.

```
⟨ Constants 42 ⟩ +≡
  const(
    seek refreshFlags = 1 ≪ iota
    insert refreshFlags = 1 ≪ iota
    exact refreshFlags = 1 ≪ iota
  )
```

141.

```
⟨ Rest of mailbox members 21 ⟩ +≡
  rfch chan ∗refresh
  irfch chan ∗refresh
  reset bool
```

142.

```
⟨ Rest of initialization of mailbox 22 ⟩ +≡
  rfch: make(chan ∗refresh, 100),
  irfch: make(chan ∗refresh, 100) ,
```

143. *box.rfch* receives a slice of messages to be printed. In case of threaded messages should be printed, but linking of messages still hasn't finished, the slice is ignored. Actually *box.rfch* is an external channel, it resend a data into *box.irfch*. If we need to stop printing, we just recreate *box.irfch*.

```
⟨ Processing of other box channels 62 ⟩ +≡
  case v :=← box.rfch:
    box.irfch ← v
  case v :=← box.irfch:
    glog.V(debug).Infof("a␣signal␣to␣print␣message␣of␣the␣'%s'␣mailbox␣window␣has␣be\
        en␣received\n", box.name)
    if box.w ≡ nil {
      glog.V(debug).Infof("a␣window␣of␣the␣'%s'␣mailbox␣doesn't␣exist,␣ignore␣the␣signal\n",
          box.name)
      continue
    }
    if box.threadMode() ∧ ¬counted {
      glog.V(debug).Infof("counting␣of␣threads␣of␣the␣'%s'␣mailbox␣is␣not␣finished,␣ig\
          nore␣the␣signal\n", box.name)
      continue
    }
    ⟨ Print messages from v.msgs 165 ⟩
```

144.

```
⟨ Determine of src 144 ⟩ ≡
  var src messages
  if box.shownew {
    src = box.unread
  } else {
    src = box.all
  }
```

This code is used in sections 146, 162, 163, 182, and 186.

145. *messages.Check* checks for message with *messageid* is already exists in *this* If it doesn't exist it is added and **true** is returned, **false** otherwise.

```
  func (this ∗ messages) Check(msg ∗ message) bool{
    pos := sort.Search(len(∗this), func(i int) bool{
      return (∗this)[i].messageid ≤ msg.messageid
    });
    if pos ≠ len(∗this) ∧ (∗this)[pos].messageid ≡ msg.messageid {
      return false
    }
    ∗this = append(∗this, nil)
    copy((∗this)[pos + 1:], (∗this)[pos:])
    (∗this)[pos] = msg
    return true
  }
```

146. All enumerated messages should be printed according to the options. In case of the thread mode sequences of full threads should be made. To avoid of duplicating of threads roots of threads are accumulated and every new root is checked for an existance.

```
⟨ Inform box to print messages 146 ⟩ ≡
  {
    glog.V(debug).Infof("inform␣the␣'%s'␣mailbox␣to␣print␣messages\n", box.name)
    ⟨ Determine of src 144 ⟩
    msgs := append(messages{}, src...)
    if box.threadMode() {
      src = msgs
      msgs = make(messages, 0, len(src))
      var roots messages
      for len(src)⟩0 {
        msg := src[0]
        ⟨ Get root of msg 134 ⟩
        if root ≡ nil ∨ ¬roots.Check(root) {
          ⟨ Remove msg from src 148 ⟩
          continue
        }
        glog.V(debug).Infof("root␣of␣thread:␣'%s/%d'\n", root.box.name, root.id)
        ⟨ Make a full thread in msgs for root 147 ⟩
      }
    } else {
      ⟨ Set pos of box 161 ⟩
    }
    box.rfch ← &refresh{0, msgs}
  }
```

This code is used in sections 61, 75, and 78.

147. *msg* is added to the *msgs* list and all its children are processed. To avoid duplicates *msg* has to be removed from *src*.

```
⟨ Make a full thread in msgs for root 147 ⟩ ≡
  msgs = append(msgs, root)
  ⟨ Remove msg from src 148 ⟩
  msgs, src = getchildren(root, msgs, src)
```

This code is used in sections 62, 146, 150, and 171.

148.

```
⟨ Remove msg from src 148 ⟩ ≡
  if p, ok := src.Search(msg.id); ok {
    glog.V(debug).Infof("removing␣'%d'␣from␣src\n", src[p].id)
    src.Delete(p)
  }
```

This code is used in sections 146, 147, and 149.

149. *getchildren* gets children for *msg*, removes *msg* from *src* and does the same for all children recursively.

```
func getchildren(msg *message, dst messages, src messages) (messages, messages){
  ⟨ Get children for msg 129 ⟩
  for _, msg := range children {
    dst = append(dst, msg)
    ⟨ Remove msg from src 148 ⟩
    dst, src = getchildren(msg, dst, src)
  }
  return dst, src
}
```

150. A list with full thread of messages is made for *msg*

```
⟨ Inform box to print a full thread with msg 150 ⟩ ≡
  ⟨ Get root of msg 134 ⟩
  var msgs messages
  src := append(messages{}, root)
  ⟨ Make a full thread in msgs for root 147 ⟩
  box.rfch ← &refresh{0, msgs}
```

This code is used in section 78.

151. Only *msg* should be printed.

```
⟨ Inform box to print msg 151 ⟩ ≡
  {
    glog.V(debug).Infof("inform the '%s' mailbox to print a message '%d'\n", box.name, msg.id)
    box.rfch ← &refresh{seek | insert, append(messages{}, msg)}
  }
```

This code is used in section 62.

152.

```
⟨ Inform box to print msgs 152 ⟩ ≡
  {
    glog.V(debug).Infof("inform the '%s' mailbox to print messages\n", box.name)
    box.rfch ← &refresh{seek | insert, msgs}
  }
```

This code is used in section 62.

153. Only *msg* should be refreshed.

```
⟨ Refresh msg 153 ⟩ ≡
  {
    glog.V(debug).Infof("refresh a message '%d'\n", msg.id)
    mrfch ← &refresh{seek, append(messages{}, msg)}
  }
```

This code is used in section 212.

154. *msgs* will be refreshed in *box* window with setting a position for every message if is found.

```
⟨ Inform box to refresh msgs 154 ⟩ ≡
  {
    if len(msgs) ≠ 0 {
      glog.V(debug).Infof("inform␣the␣'%s'␣mailbox␣to␣refresh␣messages\n", box.name)
      box.rfch ← &refresh{seek, msgs}
    }
  }
```

This code is used in section 171.

155. *msgs* will be refresh with setting a position for every message if is found.

```
⟨ Refresh msgs 155 ⟩ ≡
  {
    if len(msgs) ≠ 0 {
      glog.V(debug).Infoln("refresh␣messages\n")
      mrfch ← &refresh{seek, msgs}
    }
  }
```

This code is used in sections 103, 116, and 195.

156. *msg* will be printed only if the exact position is found.

```
⟨ Print msg at exact positon 156 ⟩ ≡
  {
    glog.V(debug).Infof("print␣'%s/%d'␣at␣exact␣position\n", box.name, msg.id)
    mrfch ← &refresh{seek | insert | exact, append(messages{}, msg)}
  }
```

This code is used in section 62.

157. One message can be presented in multiple boxes, so we have to refresh messages in all boxes. *mrfch* is a channel to receive signals to refresh messages.

```
⟨ Variables 3 ⟩ +≡
  mrfch chan *refresh = make(chan *refresh)
```

158.

```
⟨ Processing of other common channels 7 ⟩ +≡
  case r :=← mrfch:
    for i, _ := range boxes {
      glog.V(debug).Infof("sending␣messages␣to␣refresh␣in␣the␣'%s'␣mailbox\n", boxes[i].name)
      boxes[i].rfch ← &refresh{r.flags, append(messages{}, r.msgs ...)}
    }
```

159. We need to store a current position of *src* to know a message will be started to print with.

```
⟨ Rest of mailbox members 21 ⟩ +≡
  pos int
```

160.

⟨ Clean window-specific stuff 160 ⟩ ≡
```
box.pos = 0
ontop = false
```
See also section 169.

This code is used in sections 78 and 190.

161. We set *pos* to len of determinated *src* to avoid of printing messages twice.

⟨ Set *pos* of *box* 161 ⟩ ≡
```
box.pos = len(src)
```
This code is used in sections 146, 162, and 163.

162. Printing during the counting process is made only for plain mode. We use *box.pos* like a position of a first message to print and print a number of messages is multiple of 500

⟨ Inform *box* to print counting messages 162 ⟩ ≡
```
if ¬box.threadMode() {
  ⟨ Determine of src 144 ⟩
  if len(src) ≠ 0 ∧ box.pos⟨len(src) ∧ len(src) % 500 ≡ 0 {
    glog.V(debug).Infof("inform␣the␣'%s'␣mailbox␣to␣print␣the␣last␣%d␣messages\n",
        box.name, len(src) − box.pos)
    msgs := append(messages{}, src[box.pos:len(src)]...)
    ⟨ Set pos of box 161 ⟩
    box.rfch ← &refresh{0, msgs}
  }
}
```
This code is used in section 49.

163. Here we print the rest of counted messages.

⟨ Inform *box* to print the rest of counting messages 163 ⟩ ≡
```
if ¬box.threadMode() {
  ⟨ Determine of src 144 ⟩
  if box.pos⟨len(src) {
    glog.V(debug).Infof("inform␣the␣'%s'␣mailbox␣to␣print␣the␣last␣%d␣messages\n",
        box.name, len(src) − box.pos)
    msgs := append(messages{}, src[box.pos:len(src)]...)
    ⟨ Set pos of box 161 ⟩
    box.rfch ← &refresh{0, msgs}
  }
}
```
This code is used in section 49.

164.

⟨ Constants 42 ⟩ +≡
```
const eof = "$"
```

165.

```
⟨ Print messages from v.msgs 165 ⟩ ≡
  {
    glog.V(debug).Infof("printing␣of␣messages␣of␣the␣'%s'␣mailbox␣from␣v.msgs,␣len(v\
        .msgs):␣%d,␣with␣flags:␣%v\n", box.name, len(v.msgs), v.flags)
    f, err := box.w.File("data")
    if err ≠ nil {
      glog.Errorf("can't␣open␣'data'␣file␣of␣the␣'%s'␣messagebox:␣%v\n", box.name, err)
      continue
    }
    if (v.flags & seek) ≡ seek {
      ⟨ Write a tag of box window 181 ⟩
      msg := v.msgs[0]
      ⟨ Trying to find a place for msg in the box window 186 ⟩
    } else if err := box.w.WriteAddr(eof); err ≠ nil {
      glog.Errorf("can't␣write␣'%s'␣to␣'addr'␣file:␣%s\n", eof, err)
      continue
    }
    w := box.w
    glog.V(debug).Infof("printing␣of␣messages␣of␣the␣'%s'␣mailbox\n", box.name)
    buf := make([]byte, 0, #8000)
    ⟨ Compose messages of the box 168 ⟩
    if _, err := f.Write(buf); err ≠ nil {
      glog.Errorf("can't␣write␣to␣'data'␣file␣of␣the␣'%s'␣messagebox:␣%v\n", box.name, err)
    }
    ⟨ Go to the top of window for first 100 messages 167 ⟩
    ⟨ Send a rest of msgs 166 ⟩
  }
```

This code is used in section 143.

166.

```
⟨ Send a rest of msgs 166 ⟩ ≡
  if len(v.msgs)⟩0 {
    box.rfch ← &refresh{v.flags, v.msgs}
  } else {
    ⟨ Check for a clean state of the box's window 198 ⟩
  }
```

This code is used in sections 165 and 187.

167. To stay on top of the box's window when printing we go to top for first 100 messages, I hope it is enough to print other messages in the background without scrolling.

```
⟨ Go to the top of window for first 100 messages 167 ⟩ ≡
  if ¬ontop {
    glog.V(debug).Infof("pcount:␣%v,␣ontop:␣%v\n", pcount, ontop)
    ⟨ Go to top of window w 96 ⟩
    if pcount ≥ 100 {
      ontop = true
    }
  }
```

This code is used in section 165.

168. Here the messages composing is produced. To avoid of overloading of events processing we print a lot of messages at a time. But if *seek* is set in *v.flags* messages will be printed one by one, because we have to set a position for every message..

```
⟨ Compose messages of the box 168 ⟩ ≡
  c := 0
  for len(v.msgs)⟩0 ∧ c⟨100 {
    msg := v.msgs[0]
    glog.V(debug).Infof("printing␣of␣'%s/%d'␣message␣with␣in␣the␣'%s'␣mailbox\n",
        msg.box.name, msg.id, box.name)
    if box.threadMode() {
      ⟨ Add the thread level marks 170 ⟩
    }
    c++
    ⟨ Compose a header of msg 92 ⟩
    v.msgs = v.msgs[1:]
    if (v.flags & seek) ≡ seek {
      break
    }
  }
  pcount += c
```

This code is used in section 165.

169.

```
⟨ Clean window-specific stuff 160 ⟩ +≡
  {
    glog.V(debug).Infof("clean␣window-specific␣stuff␣of␣the␣'%s'␣mailbox\n", box.name)
    close(box.irfch)
    box.irfch = make(chan ∗refresh, 100)
    pcount = 0
    ontop = false
  }
```

170.

```
⟨ Add the thread level marks 170 ⟩ ≡
  {
    ⟨ Get level for msg 136 ⟩
    for ; level⟩0; level−− {
      buf = append(buf, levelmark...)
    }
  }
```

This code is used in section 168.

171. In case deleted message has children we should refresh views of these children. So we compose a list of messages and send them to refresh. But if a child is not belonged to *box* we have to erase it instead of refreshing.

```
⟨ Refresh children 171 ⟩ ≡
  { if len(children) ≠ 0 { var msgs messages
  var src messages
  for _, msg := range children { if msg.box ≠ box{⟨ Erase the message 199 ⟩} else {
    root := msg
    ⟨ Make a full thread in msgs for root 147 ⟩
  }
  }
  ⟨ Inform box to refresh msgs 154 ⟩
  }
  }
```

This code is used in section 109.

172.

```
⟨ Write a tag of main window 172 ⟩ ≡
  glog.V(debug).Infoln("writing a tag of the main window")
  del := []string{"ShowNew", "ShowAll", "ShowThreads", "ShowPlain"}
  var add []string
  if shownew {
    add = append(add, "ShowAll")
  } else {
    add = append(add, "ShowNew")
  }
  if showthreads {
    add = append(add, "ShowPlain")
  } else {
    add = append(add, "ShowThreads")
  }
  if err := writeTag(mw, del, add) err ≠ nil {
    glog.Errorf("can't set a tag of the main window: %v", err)
  }
```

This code is used in sections 41 and 45.

173.

```
  func writeTag(w *goacme.Window, del []string, add []string) error{
    if w ≡ nil ∨ del ≡ nil ∧ add ≡ nil {
      return nil
    }
    ⟨ Read a tag of w into s 174 ⟩
    ⟨ Split the tag into tag fields after the pipe symbol 175 ⟩
    ⟨ Compose newtag 176 ⟩
    ⟨ Clear the tag and write newtag to the tag 178 ⟩
    return nil
  }
```

174.

```
⟨ Read a tag of w into s 174 ⟩ ≡
  f, err := w.File("tag")
  if err ≠ nil {
    return err
  }
  if _, err := f.Seek(0, 0); err ≠ nil {
    return err
  }
  var b [1000]byte
  n, err := f.Read(b[:])
  if err ≠ nil {
    return err
  }
  s := string(b[:n])
```

This code is used in section 173.

175.

```
⟨ Split the tag into tag fields after the pipe symbol 175 ⟩ ≡
  if n = strings.LastIndex(s, "|"); n ≡ −1 {
    n = 0
  } else {
    n++
  }
  s = s[n:]
  s = strings.TrimLeft(s, "␣")
  tag := strings.Split(s, "␣")
```

This code is used in section 173.

176.

```
⟨ Compose newtag 176 ⟩ ≡
  newtag := append([]string{}, "")
  ⟨ Every part is contained in del we remove from tag 177 ⟩
  newtag = append(newtag, add ...)
  newtag = append(newtag, tag ...)
```

This code is used in section 173.

177.

```
⟨ Every part is contained in del we remove from tag 177 ⟩ ≡
  for _, v := range del {
    for i := 0; i⟨len(tag); {
      if tag[i] ≠ v {
        i++
        continue
      }
      copy(tag[i:], tag[i + 1:])
      tag = tag[:len(tag) − 1]
    }
  }
```

This code is used in section 176.

178.

```
⟨ Clear the tag and write newtag to the tag 178 ⟩ ≡
  s = strings.Join(newtag, "␣")
  if err := w.WriteCtl("cleartag"); err ≠ nil {
    return err
  }
  if _, err := f.Write([]byte(s)); err ≠ nil {
    return err
  }
```

This code is used in section 173.

179. *deleted* contains a count of messages to delete.

```
⟨ Rest of mailbox members 21 ⟩ +≡
  deleted int
```

180.

```
⟨ Write a name of box window 180 ⟩ ≡
  name := "Amail/" + box.name
  w := box.w
  ⟨ Print the name for window w 53 ⟩
```

This code is used in sections 76 and 78.

181.

```
⟨ Write a tag of box window 181 ⟩ ≡
  box.writeTag(counted)
```

This code is used in sections 61, 76, 78, 165, and 198.

182.

```
func (box * mailbox) writeTag(counted bool){
  glog.V(debug).Infof("write␣a␣tag␣of␣the␣'%s'␣mailbox's␣window\n", box.name)
  ⟨ Determine of src 144 ⟩
  del := []string{"Put", "Mail", "ShowNew", "ShowAll", "ShowPlain", "ShowThreads", "Delmesg",
      "UnDelmesg", "Thread", "Seen", "Search"}
  var add []string
  if box.deleted⟩0 {
    add = append(add, "Put")
  }
  add = append(add, "Mail")
  if box.thread {
    add = append(add, "ShowNew", "ShowAll", "ShowPlain", "ShowThreads")
  } else {
    if box.shownew {
      add = append(add, "ShowAll")
    } else {
      add = append(add, "ShowNew")
    }
    if box.showthreads {
      add = append(add, "ShowPlain")
    } else if counted {
      add = append(add, "ShowThreads")
    }
  }
  if len(src)⟩0 ∧ box.deleted⟩0 {
    add = append(add, "UnDelmesg")
  }
  if len(src)⟩box.deleted {
    add = append(add, "Delmesg")
  }
  if ¬box.thread ∧ len(src)⟩0 ∧ counted ∧ (box.shownew ∨ ¬box.showthreads) {
    add = append(add, "Thread")
  }
  if len(src)⟩0 {
    add = append(add, "Seen")
  }
  if err := writeTag(box.w, del, add); err ≠ nil {
    glog.Errorf("can't␣set␣a␣tag␣of␣the␣'%s'␣box's␣window:␣%v\n", box.name, err)
  }
}
```

183.

```
⟨ Clean box window 183 ⟩ ≡
  glog.V(debug).Infof("clean␣the␣'%s'␣mailbox's␣window\n", box.name)
  clean(box.w)
```

This code is used in sections 78 and 190.

184.

```
func clean(w *goacme.Window){
  if err := w.WriteAddr(wholefile); err ≠ nil {
    glog.Errorf("can't␣write␣'%s'␣to␣'addr'␣file:␣%s\n", wholefile, err)
  } else if data, err := w.File("data"); err ≠ nil {
    glog.Errorf("can't␣open␣'data'␣file:␣%s\n", err)
  } else if _, err := data.Write([]byte("")); err ≠ nil {
    glog.Errorf("can't␣write␣to␣'data'␣file:␣%s\n", err)
  }
}
```

185.

⟨ Constants 42 ⟩ +≡

```
const bof = "#0-"
const eol = "+#0"
```

186. For the first we try to find the message itself. If the message is new and *insert* is set in *v.flags*, we should find its neighbours and set address according to the position. The message should be skipped in other mailboxes if it is not the thread mode.

```
⟨ Trying to find a place for msg in the box window 186 ⟩ ≡
  ⟨ Determine of src 144 ⟩
  ⟨ Compose addr 188 ⟩
  glog.V(debug).Infof("composed␣message␣addr␣'%s'␣in␣the␣'%s'␣mailbox\n", addr, box.name)
  if err := box.w.WriteAddr(addr);
  err ≠ nil { glog.V(debug).Infof("the␣'%d'␣message␣is␣not␣found␣in␣the␣window␣of␣the␣'%s'␣mai\
      lbox\n", msg.id, box.name)
  if (v.flags & insert) ≡ 0 {
    ⟨ Skip current message 187 ⟩
  }
  if box.threadMode() {⟨ Set a position for a threaded message 189 ⟩} else if msg.box ≠ box{⟨ Skip current
        message 187 ⟩} else if p, ok := src.Search(msg.id); ¬ok {
    glog.V(debug).Infof("the␣'%d'␣message␣is␣not␣found␣in␣␣the␣'%s'␣mailbox's␣window\n",
        msg.id, box.name)
  } else if p ≡ 0 {
    if err := box.w.WriteAddr(bof); err ≠ nil {
      glog.Errorf("can't␣write␣'%s'␣to␣'addr'␣file␣of␣the␣'%s'␣mailbox's␣window:␣%s\n", bof,
          box.name, err)
    }
  } else if p ≡ len(src) − 1 {
    if err := box.w.WriteAddr(eof); err ≠ nil {
      glog.Errorf("can't␣write␣'%s'␣to␣'addr'␣file␣of␣the␣'%s'␣mailbox's␣window:␣%s\n", eof,
          box.name, err)
    }
  } else {
    msg := src[p − 1]
    ⟨ Compose addr 188 ⟩
    addr += eol
    if err := box.w.WriteAddr(addr); err ≠ nil {
      glog.V(debug).Infof("can't␣write␣'%s'␣to␣'addr'␣of␣the␣'%s'␣mailbox's␣window:␣%v\n",
          addr, box.name, err)
    }
  }
  }
```

This code is used in section 165.

187.

```
⟨ Skip current message 187 ⟩ ≡
  glog.V(debug).Infof("the␣'%d'␣message␣won't␣be␣inserted␣in␣the␣'%s'␣mailbox's␣window\n",
      v.msgs[0].id, box.name)
  v.msgs = v.msgs[1:]
  ⟨ Send a rest of msgs 166 ⟩
  continue
```

This code is used in sections 186 and 189.

188.

```
⟨ Compose addr 188 ⟩ ≡
  addr := fmt.Sprintf("0/^[%s]*(%s)?%s%d(%s)?\\/.*\\n.*\\n/",
    escape(levelmark),
    escape(deleted),
    func() string{
      if box ≠ msg.box {
        return escape(msg.box.name + "/")
      };
      return ""
    }(),
    msg.id,
    escape(newmark))
```

This code is used in sections 186, 189, and 200.

189. If *msg* has a parent, it should be printed after last child of the thread. In case of *msg* is only child of *msg.parent*, *msg* will be printed after *msg.parent*. If *msg* has no parent and *exact* is not set in *v.flags* it will be printed on top of the window.

```
⟨ Set a position for a threaded message 189 ⟩ ≡
  ⟨ Get parent for msg 132 ⟩
  if parent ≠ nil { glog.V(debug).Infof("msg␣'%d'␣has␣a␣parent,␣looking␣for␣a␣po\
      sition␣to␣print\n", msg.id)
  m := msg
  msg = parent
  found := false
  for ¬found {
    ⟨ Get children for msg 129 ⟩
    if len(children) ≡ 0 {
      break
    }
    for i, v := range children {
      if v ≡ m {
        if i ≡ 0 {
          found = true
        }
        break
      }
      msg = v
    }
  }
  glog.V(debug).Infof("the␣'%d'␣message␣will␣be␣printed␣after␣the␣'%d'␣message\n", m.id, msg.id)
  ⟨ Compose addr 188 ⟩
  addr += eol
  if err := box.w.WriteAddr(addr);
  err ≠ nil { glog.V(debug).Infof("can't␣write␣'%s'␣to␣'addr'␣of␣the␣'%s'␣mailbox's␣window:␣%v\
      \n", addr, box.name, err)
  if (v.flags & exact) ≡ exact{⟨ Skip current message 187 ⟩}
  }
  } else if (v.flags & exact) ≡ exact{⟨ Skip current message 187 ⟩} else if err := box.w.WriteAddr(bof);
    err ≠ nil {
    glog.Errorf("can't␣write␣'%s'␣to␣'addr'␣file␣of␣the␣'%s'␣mailbox's␣window:␣%v\n", bof,
        box.name, err)
  }
```

This code is used in section 186.

190.

```
⟨ Search messages 190 ⟩ ≡
  {
    msgs := box.search(ev.Arg)
    ⟨ Clean box window 183 ⟩
    ⟨ Clean window-specific stuff 160 ⟩
    name := fmt.Sprintf("Amail/%s/Search(%s)", box.name, strings.Replace(ev.Arg, "␣", "", −1))
    w := box.w
    box.thread = false
    box.shownew = true
    box.showthreads = false
    ⟨ Print the name for window w 53 ⟩
    glog.V(debug).Infof("len␣of␣msgs:␣%v\n", len(msgs))
    box.rfch ← &refresh{0, msgs}
  }
```

This code is used in section 78.

191.

```
func (box * mailbox) search(str string) (msgs messages){
  if len(str) ≡ 0 {
    return
  }
  f, err := box.fid.Walk("search")
  if err ≡ nil {
    err = f.Open(plan9.ORDWR)
  }
  if err ≠ nil {
    glog.Errorf("can't␣open␣'search'␣file:␣%s\n", err)
    return
  }
  defer f.Close()
  if _, err := f.Write([]byte(str)); err ≠ nil {
    glog.Errorf("can't␣write␣to␣'search'␣file:␣%s\n", err)
  }
  b := bufio.NewReader(f)
  for s, err := b.ReadString('␣'); err ≡ nil ∨ err ≡ io.EOF; s, err = b.ReadString('␣') {
    s = strings.TrimSpace(s)
    glog.V(debug).Infoln("search:␣", s)
    if num, err := strconv.Atoi(s); err ≡ nil {
      if p, ok := box.all.Search(num); ok {
        msgs.Insert(box.all[p], 0)
      }
    }
    if err ≡ io.EOF {
      break
    }
  }
  return
}
```

192. Viewing of a message.
At first let's extend *mailbox* by a *lch* channel

```
⟨ Rest of mailbox members 21 ⟩ +≡
  ach chan *struct{
    ids []int;
    a action
  }
```

193.

```
⟨ Rest of initialization of mailbox 22 ⟩ +≡
  ach: make(chan *struct{
    ids []int;
    a action
  }, 100) ,
```

194. We have to extend *message* too by *∗goacme.Window*

```
⟨ Rest of message members 29 ⟩ +≡
  w *goacme.Window
```

195. Here we will process requests to open messages. If the message is new, it should be removed from *box.unread* and its view in all windows should be changed. The count of unread messages on the main window should be refreshed too. We accumulate messages with changed status in *msgs* and refresh them after all messages are opened.

```
⟨ Processing of other box channels 62 ⟩ +≡
  case d :=← box.ach:
    switch d.a {
      case view:
        var msgs messages
        for _, id := range d.ids {
          glog.V(debug).Infof("opening␣a␣window␣with␣the␣'%d'␣message␣of␣the␣'%s'␣mailbox\n",
              id, box.name)
          p, ok := box.all.Search(id)
          if ¬ok {
            glog.V(debug).Infof("the␣'%d'␣message␣of␣the␣'%s'␣mailbox␣has␣not␣found\n", id,
                box.name)
            continue
          }
          msg := box.all[p]
          if msg.w ≡ nil {
            if msg.unread {
              ⟨ Remove id message from unread 196 ⟩
              ⟨ Refresh the message's view 197 ⟩
              ⟨ Send box to refresh the main window 70 ⟩
            }
            if err := msg.open(); err ≠ nil {
              continue
            }
          } else {
            glog.V(debug).Infof("a␣window␣of␣the␣'%d'␣message␣of␣the␣'%s'␣already␣exists,␣ju\
                st␣show␣it\n", id, box.name)
            msg.w.WriteCtl("dot=addr\nshow")
          }
        }
        ⟨ Refresh msgs 155 ⟩
        ⟨ Handling of other types of action 103 ⟩
    }
```

196.

```
⟨ Remove id message from unread 196 ⟩ ≡
  msg.unread = false
  box.unread.DeleteById(id)
```

This code is used in sections 116 and 195.

197. In case of viewing new messages only we have to remove the message from window. Also *msg* has to be added to *msgs* to refresh the message's view in other windows.

```
⟨ Refresh the message's view 197 ⟩ ≡
  if ¬box.thread ∧ box.shownew{⟨ Erase the message 199 ⟩⟨ Check for a clean state of the box's window 198 ⟩}
  msgs = append(msgs, msg)
```

This code is used in sections 116 and 195.

198.

```
⟨ Check for a clean state of the box's window 198 ⟩ ≡
  {
    glog.V(debug).Infof("box.deleted:%d\n", box.deleted)
    ⟨ Write a tag of box window 181 ⟩
    w := box.w
    if box.deleted ≡ 0 {
      ⟨ Set window w to clean state 50 ⟩
    } else {
      ⟨ Set window w to dirty state 51 ⟩
    }
  }
```

This code is used in sections 109, 166, and 197.

199. Here we remove a message *msg* from *box*'s window.

```
⟨ Erase the message 199 ⟩ ≡
  box.eraseMessage(msg)
```

This code is used in sections 109, 171, and 197.

200.

```
func (box ∗ mailbox) eraseMessage(msg ∗ message){
  if box.w ≡ nil {
    return
  }
  glog.V(debug).Infof("removing␣the␣'%d'␣message␣of␣the␣'%s'␣mailbox␣from␣the␣'%s'\
      ␣mailbox\n",
    msg.id, msg.box.name, box.name)
  ⟨ Compose addr 188 ⟩
  if err := box.w.WriteAddr(addr); err ≠ nil {
    glog.V(debug).Infof("can't␣write␣'%s'␣to␣'addr'␣of␣the␣'%s'␣mailbox's␣window:␣%v\n",
        addr, box.name, err)
  } else if data, err := box.w.File("data"); err ≠ nil {
    glog.Errorf("can't␣open␣'data'␣file␣of␣the␣box␣'%s':␣%s\n", box.name, err)
  } else if _, err := data.Write([]byte{}); err ≠ nil {
    glog.Errorf("can't␣write␣to␣'data'␣file␣of␣the␣box␣'%s':␣%s\n", box.name, err)
  }
}
```

201.

```
⟨ Rest of message members 29 ⟩ +≡
  to []string
  cc []string
```

202.

```
⟨ Read other fields of a message 95 ⟩ +≡
  if strings.HasPrefix(s, "to␣") {
    msg.to = split(s[len("to␣"):])
    continue
  }
  if strings.HasPrefix(s, "cc␣") {
    msg.cc = split(s[len("cc␣"):])
    continue
  }
```

203. *split* splits *s* to a []**string** of mail addresses that can contain a name and an address. If a name is just `''`, it is removed.

```
func split(s string) (strs []string){
  f := strings.Fields(s)
  m := ""
  for _, v := range f {
    if strings.Contains(v, "@") {
      m += v
      strs = append(strs, m)
      m = ""
    } else if v ≠ "''" {
      m += v + "␣"
    }
  }
  return
}
```

204. *open* opens a message in a separated window.

```
func (msg * message) open() (err error){
  glog.V(debug).Infof("open:␣trying␣to␣open␣'%d'␣directory\n", msg.id)
  bfid, err := msg.box.fid.Walk(fmt.Sprintf("%d", msg.id))
  if err ≠ nil {
    glog.Errorf("can't␣walk␣to␣'%s/%d':␣%v\n", msg.box.name, msg.id, err)
    return err
  }
  defer bfid.Close()
  isnew := msg.w ≡ nil
  if isnew {
    if msg.w, err = goacme.New(); err ≠ nil {
      glog.Errorf("can't␣create␣a␣window:␣%v\n", err)
      return err
    }
  } else {
    ⟨ Clean msg.w window 213 ⟩
  }
  buf := make([]byte, 0, #8000)
  ⟨ Compose a header of the message 211 ⟩
  ⟨ Compose a body of the message 218 ⟩
  w := msg.w
  name := fmt.Sprintf("Amail/%s/%d", msg.box.name, msg.id)
  ⟨ Print the name for window w 53 ⟩
  ⟨ Write a tag of message window 205 ⟩
  w.Write(buf)
  ⟨ Set window w to clean state 50 ⟩
  ⟨ Go to top of window w 96 ⟩
  if isnew {
    ⟨ Start a goroutine to process events from the message's window 212 ⟩
  }
  return
}
```

205.

⟨ Write a tag of message window 205 ⟩ ≡

```
msg.writeTag()
```

This code is used in sections 104, 105, 204, and 212.

206.

⟨ Get a previous *pmsg* 206 ⟩ ≡

```
pmsg := msg.prev()
```

This code is used in sections 210 and 212.

207.

```
func (this * message) prev() (pmsg * message){
  msg := this
  ⟨ Get parent for msg 132 ⟩
  if parent ≡ nil {
    return
  }
  msg = parent
  ⟨ Get children for msg 129 ⟩
  for _, v := range children {
    if v ≡ this {
      break
    }
    pmsg = v
  }
  return
}
```

208.

⟨ Get a next *nmsg* 208 ⟩ ≡

```
nmsg := msg.next()
```

This code is used in sections 210 and 212.

209.

```
func (this * message) next() (nmsg * message){
  msg := this
  ⟨ Get parent for msg 132 ⟩
  if parent ≡ nil {
    return
  }
  msg = parent
  ⟨ Get children for msg 129 ⟩
  for i := 0; i⟨len(children); i++ {
    if children[i] ≠ this {
      continue
    }
    i++
    if i⟨len(children) {
      nmsg = children[i]
    }
    break
  }
  return
}
```

210.

```
func (msg * message) writeTag(){
  glog.V(debug).Infof("writing a tag of the '%d' message's window\n", msg.id)
  del := []string{"Q", "Reply", "all", "Delmesg", "UnDelmesg", "Text", "Html", "Browser", "Up",
      "Down", "Prev", "Next", "Save"}
  add := append([]string{}, "Q", "Reply", "all")
  if msg.deleted {
    add = append(add, "UnDelmesg")
  } else {
    add = append(add, "Delmesg")
  }
  if msg.showhtml {
    add = append(add, "Text")
  } else {
    add = append(add, "Html")
  }
  if len(msg.html) ≠ 0 {
    add = append(add, "Browser")
  }
  ⟨ Get parent for msg 132 ⟩
  if parent ≠ nil {
    add = append(add, "Up")
  }
  ⟨ Get children for msg 129 ⟩
  if len(children) ≠ 0 {
    add = append(add, "Down")
  }
  ⟨ Get a previous pmsg 206 ⟩
  if pmsg ≠ nil {
    add = append(add, "Prev")
  }
  ⟨ Get a next nmsg 208 ⟩
  if nmsg ≠ nil {
    add = append(add, "Next")
  }
  add = append(add, "Save")
  if err := writeTag(msg.w, del, add); err ≠ nil {
    glog.Errorf("can't set a tag of the message window: %v", err)
  }
}
```

211.

⟨ Compose a header of the message 211 ⟩ ≡

```
{
  glog.V(debug).Infof("composing␣a␣header␣of␣the␣'%d'␣message\n", msg.id)
  buf = append(buf, fmt.Sprintf("From:␣%s\nDate:␣%s\nTo:␣%s\n%sSubject:␣%s\n\n\n",
    msg.from, msg.date, strings.Join(msg.to, ",␣"),
    func() string{
      if len(msg.cc) ≠ 0 {
        return fmt.Sprintf("CC:␣%s\n", strings.Join(msg.cc, ",␣"))
      };
      return ""
    }(),
    msg.subject)...)
}
```

This code is used in section 204.

212.

```
⟨ Start a goroutine to process events from the message's window 212 ⟩ ≡
  go func() { glog.V(debug).Infof("starting␣a␣goroutine␣to␣process␣events␣from␣the␣'%d'␣messag\
      e's␣window\n", msg.id)
  for ev, err := msg.w.ReadEvent();
  err ≡ nil;
  ev, err = msg.w.ReadEvent() { if ev.Origin ≠ goacme.Mouse {
    msg.w.UnreadEvent(ev)
    continue
  }
  quote := false
  replyall := false
  if (ev.Type & goacme.Execute) ≡ goacme.Execute { switch ev.Text {
    case "Del":
      msg.w.UnreadEvent(ev)
      msg.w.Close()
      msg.w = nil
      return
    case "Delmesg":
      if ¬msg.deleted {
        msg.deleted = true
        msg.box.deleted++
        msg.w.Del(true)
        msg.w.Close()
        msg.w = nil
        ⟨ Refresh msg 153 ⟩
        return
      }
      continue
    case "UnDelmesg":
      if msg.deleted {
        msg.deleted = false
        msg.box.deleted--
        ⟨ Write a tag of message window 205 ⟩
        ⟨ Refresh msg 153 ⟩
      }
      continue
    case "Text":
      if len(msg.text) ≠ 0 ∧ len(msg.html) ≠ 0 {
        msg.showhtml = false
        msg.open()
      }
      continue
    case "Html":
      if len(msg.text) ≠ 0 ∧ len(msg.html) ≠ 0 {
        msg.showhtml = true
        msg.open()
      }
      continue
    case "Browser":
      ⟨ Save stuff on disk and plumb a message to a web browser 226 ⟩
      continue
```

```
case "Save":
  ⟨ Save a message 236 ⟩
  continue
case "Q":
  quote = true
  if args := strings.Fields(ev.Arg); len(args)⟩0 {
    ev.Text = args[0]
    ev.Arg = strings.Join(args, "␣")
  }
  fallthrough
case "Reply", "Replyall":
  if ev.Text ≡ "Reply" {
    args := strings.Fields(ev.Arg)
    for _, v := range args {
      if v ≡ "all" {
        replyall = true
      }
    }
  } else if ev.Text ≡ "Replyall" {
    replyall = true
  }
  ⟨ Compose a message 237 ⟩
  continue
case "Up":
  ⟨ Get parent for msg 132 ⟩
  if parent ≠ nil {
    ⟨ Create msgs 86 ⟩
    name := parent.box.name
    id := parent.id
    ⟨ Add a id message to msgs 87 ⟩
    ⟨ Send msgs for viewing 89 ⟩
  }
  continue
case "Down":
  ⟨ Get children for msg 129 ⟩
  if len(children) ≠ 0 {
    ⟨ Create msgs 86 ⟩
    name := children[0].box.name
    id := children[0].id
    ⟨ Add a id message to msgs 87 ⟩
    ⟨ Send msgs for viewing 89 ⟩
  }
  continue
case "Prev":
  ⟨ Get a previous pmsg 206 ⟩
  if pmsg ≠ nil {
    ⟨ Create msgs 86 ⟩
    name := pmsg.box.name
    id := pmsg.id
    ⟨ Add a id message to msgs 87 ⟩
    ⟨ Send msgs for viewing 89 ⟩
  }
```

```
        continue
      case "Next":
        ⟨ Get a next nmsg 208 ⟩
        if nmsg ≠ nil {
          ⟨ Create msgs 86 ⟩
          name := nmsg.box.name
          id := nmsg.id
          ⟨ Add a id message to msgs 87 ⟩
          ⟨ Send msgs for viewing 89 ⟩
        }
        continue
    }
    } else if (ev.Type & goacme.Look) ≡ goacme.Look{}
    msg.w.UnreadEvent(ev)
    }
    } ( )
```

This code is used in section 204.

213.

```
⟨ Clean msg.w window 213 ⟩ ≡
  glog.V(debug).Infof("clean the '%s/%d' message's window\n", msg.box.name, msg.id)
  clean(msg.w)
```

This code is used in section 204.

214.

```
⟨ Imports 11 ⟩ +≡
  "os/exec"
```

215.

```
⟨ Types 19 ⟩ +≡
  file struct{
    name string
    mimetype string
    path string
  }
```

216. *text* contains a path in `mailfs` from the message root to a text variant of the message.
html contains a path in `mailfs` from the message root to a html variant of the message.
showmail is a flag to show the html variant of the message.
files contains ∗*file* with *path*s in `mailfs` from the message root to a file is attached in the message, *mimetype* and *name* of the file.
cids contains a map of `"Content-ID"` on ∗*file*.

```
⟨ Rest of message members 29 ⟩ +≡
  text string
  html string
  showhtml bool
  files []∗file
  cids map[string]∗file
```

217.

⟨ Rest of initialization of *message* 217 ⟩ ≡
cids: **make**(**map**[**string**] ∗ *file*) ,

This code is used in section 30.

218. If *text* and *html* is empty we should fill them by *bodyPath*, then we read corresponding variant of the message. In case of the html variant we print *buf* and start a pipe of external programs `"9p"` and `"htmlfmt"` to print the html message body to the window. Then we fill *buf* with command to obtain contents of *files*.

```
⟨ Compose a body of the message 218 ⟩ ≡
  {
    if len(msg.text) ≡ 0 ∧ len(msg.html) ≡ 0 {
      if err = msg.bodyPath(bfid, ""); err ≠ nil {
        glog.Errorf("can't ged a body path of '%d': %v\n", msg.id, err)
      }
      glog.V(debug).Infof("paths for bodies of the '%d' message have been found: text-\
          '%s', html-'%s'\n", msg.id, msg.text, msg.html)
    }
    if len(msg.text) ≠ 0 ∧ ¬msg.showhtml {
      glog.V(debug).Infof("using a path for a text body of the '%d' message: '%s'\n", msg.id,
          msg.text)
      if buf, err = readAll(bfid, msg.text, buf); err ≠ nil {
        glog.Errorf("can't read '%s': %v\n", msg.text, err)
        return
      }
    } else if len(msg.html) ≠ 0 {
      glog.V(debug).Infof("using a path for a html body of the '%d' message: '%s'\n", msg.id,
          msg.html)
      msg.w.Write(buf)
      buf = nil
      c1 := exec.Command("9p", "read", fmt.Sprintf("%s/%s/%d/%s", srv, msg.box.name, msg.id,
          msg.html))
      c2 := exec.Command("htmlfmt", "-cutf-8")
      c2.Stdout, _ = msg.w.File("body")
      c2.Stdin, err = c1.StdoutPipe()
      if err ≠ nil {
        glog.Errorf("can't get a stdout pipe: %v\n", err)
        return
      }
      if err = c2.Start(); err ≠ nil {
        glog.Errorf("can't start 'htmlfmt': %v\n", err)
        return
      }
      if err = c1.Run(); err ≠ nil {
        glog.Errorf("can't run '9p': %v\n", err)
        c2.Wait()
        return
      }
      if err = c2.Wait(); err ≠ nil {
        glog.Errorf("can't wait of completion 'htmlfmt': %v\n", err)
        return
      }
    }
    ⟨ Get home enviroment variable 222 ⟩
    for _, v := range msg.files {
      buf = append(buf, fmt.Sprintf("\n===> %s (%s)\n", v.path, v.mimetype)...)
      buf = append(buf, fmt.Sprintf("\t9p read %s/%s/%d/%sbody > '%s/%s'\n", srv, msg.box.name,
          msg.id, v.path, home, v.name)...)
```

```
    }
  }
```

This code is used in section 204.

219. *bodyPath* recursively looks for parts of the message to determine text and html variants of the message and attached files.

```
func (msg * message) bodyPath(bfid * client.Fid, path string) error{
  glog.V(debug).Infof("getting␣a␣path␣for␣a␣body␣of␣the␣'%d'␣message\n", msg.id)
  t, err := readString(bfid, path + "type")
  if err ≠ nil {
    return err
  }
  switch t {
    case
      "message/rfc822", "text", "text/plain", "text/richtext", "text/tab-separated-values":
      if len(msg.text) ≡ 0 {
        msg.text = path + "body"
        glog.V(debug).Infof("a␣path␣for␣a␣text␣body␣of␣the␣'%d'␣message:␣'%s'\n", msg.id, t)
        return nil
      }
    case "text/html":
      if len(msg.html) ≡ 0 {
        msg.html = path + "body"
        glog.V(debug).Infof("a␣path␣for␣a␣html␣body␣of␣the␣'%d'␣message:␣'%s'\n", msg.id, t)
        return nil
      }
    case "multipart/mixed", "multipart/alternative", "multipart/related",
         "multipart/signed", "multipart/report":
      for c := 1;; c++ {
        if err = msg.bodyPath(bfid, fmt.Sprintf("%s%d/", path, c)); err ≠ nil {
          break
        }
      }
      return nil
  }
  glog.V(debug).Infof("trying␣to␣read␣'%d/%sfilename'\n", msg.id, path)
  if n, err := readString(bfid, path + "filename"); err ≡ nil {
    f := &file{name: n, mimetype: t, path: path,}
    if len(n) ≡ 0 {
      f.name = "attachment"
    } else if cid, ok := msg.getCID(path); ok {
      msg.cids[cid] = f
    }
    msg.files = append(msg.files, f)
  }
  return nil
}
```

220. *getCID* parses `"mimeheader"` and takes `"Content-ID"` identifier for *path*

```
func (msg * message) getCID(path string) (string, bool){
  src := fmt.Sprintf("%d/%smimeheader", msg.id, path)
  glog.V(debug).Infof("getting␣of␣cids␣for␣path␣'%s'\n", src)
  fid, err := msg.box.fid.Walk(src)
  if err ≡ nil {
    err = fid.Open(plan9.OREAD)
  }
  if err ≠ nil {
    glog.Errorf("can't␣open␣'%s':␣%v\n", src, err)
    return "", false
  }
  defer fid.Close()
  fid.Seek(0, 0)
  b := bufio.NewReader(fid)
  for s, err := b.ReadString('\n'); err ≡ nil ∨ err ≡ io.EOF; s, err = b.ReadString('\n') {
    glog.V(debug).Infof("looking␣for␣a␣cid␣in␣'%s'\n", s)
    if strings.HasPrefix(s, "Content-ID:␣<") {
      s = s[len("Content-ID:␣<"):len(s) − 2]
      glog.V(debug).Infof("found␣a␣cid␣'%s'\n", s)
      return s, true
    }
    if err ≡ io.EOF {
      break
    }
  }
  return "", false
}
```

221. *home* environment variable.

```
⟨ Variables 3 ⟩ +≡
  home string
```

222.

```
⟨ Get home enviroment variable 222 ⟩ ≡
  ⟨ Get some things at once 235 ⟩
```

This code is used in sections 218 and 251.

223.

```
⟨ Get it at once 223 ⟩ ≡
  if home = os.Getenv("home"); len(home) ≡ 0 {
    if home = os.Getenv("HOME"); len(home) ≡ 0 {
      glog.Errorln("can't␣get␣a␣home␣directory␣from␣the␣environment,␣the␣home␣i\
          s␣assumed␣'/'")
      home = "/"
    }
  }
```

See also sections 232 and 247.

This code is used in section 235.

224. *readStrings* reads a full string from *name* file with *pfid* like a root.

```
func readString(pfid *client.Fid, name string) (str string, err error){
  glog.V(debug).Infof("readString: trying to open '%s'\n", name)
  f, err := pfid.Walk(name)
  if err ≡ nil {
    err = f.Open(plan9.OREAD)
  }
  if err ≠ nil {
    return
  }
  defer f.Close()
  str, err = bufio.NewReader(f).ReadString('\n')
  if err ≠ nil ∧ err ≠ io.EOF {
    return
  }
  return str, nil
}
```

225. *readAll* reads all content of *name* file with *pfid* like a root in *buf*

```
func readAll(pfid *client.Fid, name string, buf []byte) ([]byte, error){
  glog.V(debug).Infof("readAll: trying to open '%s'\n", name)
  f, err := pfid.Walk(name)
  if err ≡ nil {
    err = f.Open(plan9.OREAD)
  }
  if err ≠ nil {
    return buf, err
  }
  defer f.Close()
  b := bufio.NewReader(f)
  for s, err := b.ReadString('\n'); err ≡ nil ∨ err ≡ io.EOF; s, err = b.ReadString('\n') {
    if strings.HasSuffix(s, "\r\n") {
      s = strings.TrimRight(s, "\r\n")
      s += "\n"
    }
    buf = append(buf, s...)
    if err ≡ io.EOF {
      break
    }
  }
  return buf, nil
}
```

226. To view a message in a web brower we need to store a body of the message and all images of the message on disk and plumb a full pathname of the message to `"web"` rule. But in case of the images the body should be fixed to help a browser to find the images.

```
⟨ Save stuff on disk and plumb a message to a web browser 226 ⟩ ≡
  {
    ⟨ Get current user 231 ⟩
    dir := fmt.Sprintf("%s/amail-%s/%s/%d", os.TempDir(), cuser, msg.box.name, msg.id)
    if err := os.MkdirAll(dir, °700); err ≠ nil {
      glog.Errorf("can't␣create␣a␣directory␣'%s':␣%v\n", dir, err)
      continue
    }
    if len(msg.files) ≡ 0 {
      if err := saveFile(fmt.Sprintf("%s/%s/%d/%s", srv, msg.box.name, msg.id, msg.html),
          fmt.Sprintf("%s/%d.html", dir, msg.id)); err ≠ nil {
        continue
      }
    } else {
      if err := msg.fixFile(dir); err ≠ nil {
        continue
      }
      for _, v := range msg.files {
        saveFile(fmt.Sprintf("%s/%s/%d/%s/body", srv, msg.box.name, msg.id, v.path),
            fmt.Sprintf("%s/%s", dir, v.name))
      }
    }
    if p, err := goplumb.Open("send", plan9.OWRITE); err ≠ nil {
      glog.Errorf("can't␣open␣plumbing␣port␣'send':␣%v\n", err)
    } else if err := p.SendText("amail", "web", dir, fmt.Sprintf("file://%s/%d.html", dir, msg.id));
      err ≠ nil {
      glog.Errorf("can't␣plumb␣a␣message␣'%s':␣%v\n", fmt.Sprintf("file://%s/%d.html", dir,
          msg.id), err)
    }
  }
```

This code is used in section 212.

227. *saveFile* saves file on a disk by call `"9p"`

```
func saveFile(src, dst string) error{
  var err error
  c := exec.Command("9p", "read", src)
  f, err := os.OpenFile(dst, os.O_WRONLY | os.O_CREATE | os.O_TRUNC, °600)
  if err ≠ nil {
    glog.Errorf("can't␣create␣a␣file␣'%s':␣%v\n", dst, err)
    return err
  }
  defer f.Close()
  c.Stdout = f
  if err = c.Run(); err ≠ nil {
    glog.Errorf("can't␣run␣'9p':␣%v\n", err)
  }
  return err
}
```

228. *fixFile* reads the message body and replaces all `"cid"` on corresponding cids.

```
func (msg * message) fixFile(dir string) error{
  src := fmt.Sprintf("%d/%s", msg.id, msg.html)
  dst := fmt.Sprintf("%s/%d.html", dir, msg.id)
  df, err := os.OpenFile(dst, os.O_WRONLY | os.O_CREATE | os.O_TRUNC, °600)
  if err ≠ nil {
    glog.Errorf("can't␣create␣a␣file␣'%s':␣%v\n", dst, err)
    return err
  }
  defer df.Close()
  fid, err := msg.box.fid.Walk(src)
  if err ≡ nil {
    err = fid.Open(plan9.OREAD)
  }
  if err ≠ nil {
    glog.Errorf("can't␣open␣to␣'%s':␣%v\n", src, err)
    return err
  }
  defer fid.Close()
  b := bufio.NewReader(fid)
  for s, err := b.ReadString('\n'); err ≡ nil ∨ err ≡ io.EOF; s, err = b.ReadString('\n') {
    p := 0
    for b := strings.Index(s[p:], "\"cid:"); b ≠ −1; b = strings.Index(s[p:], "\"cid:") {
      b += p
      e := strings.Index(s[b + 1:], "\"")
      if e ≡ −1 {
        break
      }
      e++
      glog.V(debug).Infof("len(s):␣%v,␣p:␣%v,␣b:␣%v,␣e:␣%v\n", len(s), p, b, e)
      cid := s[b + 5:b + e]
      glog.V(debug).Infof("cid:␣%s\n", cid)
      if f, ok := msg.cids[cid]; ok {
        glog.V(debug).Infof("found␣a␣cid:␣%s,␣replace␣'%s'␣by␣'%s'\n", cid, s[b+1:b+e], f.name)
        s = strings.Replace(s, s[b + 1:b + e], f.name, 1)
      } else {
        p = b + e
      }
    }
    df.Write([]byte(s))
    if err ≡ io.EOF {
      break
    }
  }
  return err
}
```

229.

⟨ Imports 11 ⟩ +≡

```
"os/user"
```

230. current *user*

```
⟨ Variables 3 ⟩ +≡
  cuser string
```

231.

```
⟨ Get current user 231 ⟩ ≡
  ⟨ Get some things at once 235 ⟩
```

This code is used in section 226.

232.

```
⟨ Get it at once 223 ⟩ +≡
  if u, err := user.Current(); err ≠ nil {
    glog.Errorf("can't␣get␣a␣name␣of␣the␣current␣user:␣%v\n", err)
  } else {
    cuser = u.Username
  }
```

233.

```
⟨ Imports 11 ⟩ +≡
  "sync"
```

234.

```
⟨ Variables 3 ⟩ +≡
  once sync.Once
```

235.

```
⟨ Get some things at once 235 ⟩ ≡
  once.Do(func(){⟨ Get it at once 223 ⟩})
```

This code is used in sections 222, 231, and 246.

236.

```
⟨ Save a message 236 ⟩ ≡
  {
    if len(ev.Arg) ≡ 0 {
      continue
    }
    f, err := msg.box.fid.Walk("ctl")
    if err ≡ nil {
      err = f.Open(plan9.OWRITE)
    }
    if err ≠ nil {
      glog.Errorf("can't open 'ctl': %v\n", err)
      continue
    }
    bs := strings.Fields(ev.Arg)
    for _, v := range bs {
      s := fmt.Sprintf("save %s %d/", v, msg.id)
      if _, err := f.Write([]byte(s)); err ≠ nil {
        glog.Errorf("can't write '%s' to 'ctl': %v\n", s, err)
      }
    }
    f.Close()
  }
```

This code is used in section 212.

237. Composing a message.

```
⟨ Compose a message 237 ⟩ ≡
  {
    ⟨ Create a new message window 238 ⟩
    name := fmt.Sprintf("Amail/%s/%d/%sReply%s",
      msg.box.name,
      msg.id,
      func() string{
        if quote {
          return "Q"
        };
        return ""
      }(),
      func() string{
        if replyall {
          return "all"
        };
        return ""
      }())
    ⟨ Print the name for window w 53 ⟩
    buf := make([]byte, 0, #8000)
    buf = append(buf, fmt.Sprintf("To: %s\n", msg.from)...)
    if replyall {
      for _, v := range msg.to {
        buf = append(buf, fmt.Sprintf("To: %s\n", v)...)
      }
      for _, v := range msg.cc {
        buf = append(buf, fmt.Sprintf("CC: %s\n", v)...)
      }
    }
    buf = append(buf, fmt.Sprintf("Subject: %s%s\n",
      func() string{
        if ¬strings.Contains(msg.subject, "Re:") {
          return "Re: "
        }
        return ""
      }(),
      msg.subject)...)
    if quote {
      buf = append(buf, '\n')
      ⟨ Add quoted message 240 ⟩
    } else {
      buf = append(buf, fmt.Sprintf("Include: Mail/%s/%d/raw\n", msg.box.name, msg.id)...)
    }
    buf = append(buf, '\n')
    w.Write(buf)
    ⟨ Append msg.box-specific signature 249 ⟩
  }
```

This code is used in section 212.

238.

```
⟨ Create a new message window 238 ⟩ ≡
  w, err := goacme.New()
  if err ≠ nil {
    glog.Errorf("can't␣create␣a␣window:␣%v\n", err)
    continue
  }
  l := []string{"Look", "Post", "Undo"}
  if err := writeTag(w, l, l); err ≠ nil {
    glog.Errorf("can't␣write␣a␣tag␣for␣a␣new␣message␣window:␣%v\n", err)
  }
  ⟨ Start a goroutine to process events from a composed mail window 239 ⟩
```

This code is used in sections 33, 78, and 237.

239. If we are going to reply a message, we should specify *msg*.

```
⟨ Start a goroutine to process events from a composed mail window 239 ⟩ ≡
  go func(msg *message){
    glog.V(debug).Infoln("starting␣a␣goroutine␣to␣process␣events␣from␣a␣composed␣mail␣window")
    for ev, err := w.ReadEvent(); err ≡ nil; ev, err = w.ReadEvent() {
      if ev.Origin ≠ goacme.Mouse {
        w.UnreadEvent(ev)
        continue
      }
      if (ev.Type & goacme.Execute) ≡ goacme.Execute {
        switch ev.Text {
          case "Del":
            w.UnreadEvent(ev)
            w.Close()
            return
          case "Post":
            ⟨ Send the message 243 ⟩
        }
      }
      w.UnreadEvent(ev)
    }
  }(msg)
```

This code is used in section 238.

240.

```
⟨ Add quoted message 240 ⟩ ≡
  if len(msg.text) ≠ 0 {
    fn := fmt.Sprintf("%d/%s", msg.id, msg.text)
    f, err := msg.box.fid.Walk(fn)
    if err ≡ nil {
      err = f.Open(plan9.OREAD)
    }
    if err ≠ nil {
      glog.Errorf("can't␣open␣'%s/%s/%s':␣%v\n", srv, msg.box.name, fn, err)
      continue
    }
    ⟨ Quote a message 241 ⟩
    f.Close()
  } else if len(msg.html) ≠ 0 {
    ⟨ Quote a html message 242 ⟩
  }
```

This code is used in section 237.

241. To quote a message we read strings from f and add `">␣"` to the begin of every string.

```
⟨ Quote a message 241 ⟩ ≡
  {
    b := bufio.NewReader(f)
    for s, err := b.ReadString('\n'); err ≡ nil ∨ err ≡ io.EOF; s, err = b.ReadString('\n') {
      buf = append(buf, '>', '␣')
      if strings.HasSuffix(s, "\r\n") {
        s = strings.TrimRight(s, "\r\n")
        s += "\n"
      }
      buf = append(buf, s...)
      if err ≡ io.EOF {
        break
      }
    }
  }
```

This code is used in sections 240 and 242.

242. To quote the html message we start a pipe of external programs `"9p"` and `"htmlfmt"` and read an output of `"htmlfmt"`

```
⟨ Quote a html message 242 ⟩ ≡
  {
    c1 := exec.Command("9p", "read", fmt.Sprintf("%s/%s/%d/%s", srv, msg.box.name, msg.id,
        msg.html))
    c2 := exec.Command("htmlfmt", "-cutf-8")
    f, err := c2.StdoutPipe()
    if err ≠ nil {
      glog.Errorf("can't get a stdout pipe: %v\n", err)
    }
    c2.Stdin, err = c1.StdoutPipe()
    if err ≠ nil {
      glog.Errorf("can't get a stdout pipe: %v\n", err)
      f.(io.Closer).Close()
      continue
    }
    if err = c2.Start(); err ≠ nil {
      glog.Errorf("can't start 'htmlfmt': %v\n", err)
      f.(io.Closer).Close()
      continue
    }
    if err = c1.Start(); err ≠ nil {
      glog.Errorf("can't run '9p': %v\n", err)
      c2.Wait()
      f.(io.Closer).Close()
      continue
    }
    ⟨ Quote a message 241 ⟩
    c1.Wait()
    c2.Wait()
    f.(io.Closer).Close()
  }
```

This code is used in section 240.

243. To send a message we start an external program `"upas/marshal"` and send to its input recipient and body of the message. If $msg \neq$ **nil**, it will be added like a message is replied.

```
⟨ Send the message 243 ⟩ ≡
  {
    ⟨ Get plan9dir from enviroment variable 246 ⟩
    w.Seek(0, 0)
    w.WriteAddr(wholefile)
    ff, _ := w.File("xdata")
    b := bufio.NewReader(ff)
    var to, cc, bcc, attach, include []string
    var subject string
    for{
      s, err := b.ReadString('\n')
      if err ≠ nil {
        break
      }
      s = strings.TrimSpace(s)
      if len(s) ≡ 0 {
          // an empty line, the rest is a body of the message
        break
      }
      p := strings.Index(s, ":")
      if p ≠ −1 {
        f := strings.Split(s[p + 1:], ",")
        for i, _ := range f {
          f[i] = strings.TrimSpace(f[i])
        }
        switch strings.ToLower(s[:p]) {
          case "to":
            ⟨ Get last elements of addresses 244 ⟩
            to = append(to, f...)
          case "cc":
            ⟨ Get last elements of addresses 244 ⟩
            cc = append(cc, f...)
          case "bcc":
            ⟨ Get last elements of addresses 244 ⟩
            bcc = append(bcc, f...)
          case "attach":
            attach = append(attach, f...)
          case "include":
            include = append(include, f...)
          case "subject":
            subject = fmt.Sprintf("%s", strings.TrimSpace(s[p + 1:]))
        }
      } else {
          // recipient addresses can be written without "to:"
        f := strings.Split(s, ",")
        for i, _ := range f {
          f[i] = strings.TrimSpace(f[i])
        }
        ⟨ Get last elements of addresses 244 ⟩
        to = append(to, f...)
```

```
    }
  }
  args := append([]string{}, "-8")
  if msg ≠ nil {
    args = append(args, "-R", fmt.Sprintf("%s/%d", msg.box.name, msg.id))
  }
  if len(subject) ≠ 0 {
    args = append(args, "-s", subject)
  }
  for _, v := range include {
    args = append(args, "-A", v)
  }
  for _, v := range attach {
    args = append(args, "-a", v)
  }
  c := exec.Command(plan9dir + "/bin/upas/marshal", args...)
  p, err := c.StdinPipe()
  if err ≠ nil {
    glog.Errorf("can't get a stdin pipe: %v\n", err)
    continue
  }
  if err := c.Start(); err ≠ nil {
    glog.Errorf("can't start 'upas/marshal': %v\n", err)
    continue
  }
  if len(to) ≠ 0 {
    if _, err := fmt.Fprintln(p, "To:", strings.Join(to, ",")); err ≠ nil {
      glog.Errorf("can't write 'to' fields to 'upas/marshal': %v\n", err)
      continue
    }
  }
  glog.V(debug).Infoln("to is written")
  if len(cc) ≠ 0 {
    if _, err := fmt.Fprintln(p, "CC:", strings.Join(cc, ",")); err ≠ nil {
      glog.Errorf("can't write 'cc' fields to 'upas/marshal': %v\n", err)
      continue
    }
  }
  glog.V(debug).Infoln("cc is written")
  if len(bcc) ≠ 0 {
    if _, err := fmt.Fprintln(p, "BCC:", strings.Join(bcc, ",")); err ≠ nil {
      glog.Errorf("can't write 'bcc' fields to  'upas/marshal': %v\n", err)
      continue
    }
  }
  glog.V(debug).Infoln("bcc is written")
  for s, err := b.ReadString('\n'); err ≡ nil ∨ err ≡ io.EOF; s, err = b.ReadString('\n') {
    glog.V(debug).Infof("writing '%s':%v", s, err)
    p.Write([]byte(s))
    if err ≡ io.EOF {
      p.Write([]byte("\n"))
      break
```

```
        }
    }
    glog.V(debug).Infoln("body is written")
    p.Write([]byte("\n"))
    p.Close()
    c.Wait()
    w.Del(true)
    w.Close()
  }
```

This code is used in section 239.

244. An address can be preffered by name or alias of recipient. It has to be removed.

```
⟨ Get last elements of addresses 244 ⟩ ≡
  for i, _ := range f {
    f[i] = strings.TrimSpace(f[i])
    if sf := strings.Fields(f[i]); len(sf)>1 {
      f[i] = strings.TrimSpace(sf[len(sf) − 1])
    }
  }
```

This code is used in section 243.

245.

```
⟨ Variables 3 ⟩ +≡
  plan9dir string
```

246.

```
⟨ Get plan9dir from enviroment variable 246 ⟩ ≡
  ⟨ Get some things at once 235 ⟩
```

This code is used in section 243.

247.

```
⟨ Get it at once 223 ⟩ +≡
  if plan9dir = os.Getenv("PLAN9"); len(plan9dir) ≡ 0 {
    glog.Errorln("can't get PLAN9 directory from the environment, the plan9di\
        r is assumed '/usr/local/plan9'")
    plan9dir = "/usr/local/plan9"
  }
```

248.

```
⟨ Append box-specific signature 248 ⟩ ≡
  writeSignature(w, box)
```

This code is used in section 78.

249.

```
⟨ Append msg.box-specific signature 249 ⟩ ≡
  if msg ≠ nil {
    writeSignature(w, msg.box)
  } else {
    writeSignature(w, nil)
  }
```

This code is used in section 237.

250.

⟨ Append common signature 250 ⟩ ≡
```
writeSignature(w, nil)
```
This code is used in section 33.

251. At first we are looking for *box*-specific signature in `$HOME`/*mail* / ⟨ *mailbox* ⟩ . *signature* file. If the file doesn't exist, we are trying to open `$HOME`/*mail* / ⟨ *mailbox* ⟩ . *signature* file with common signature.

```
func writeSignature(w *goacme.Window, box *mailbox){
  ⟨ Get home enviroment variable 222 ⟩
  var f io.ReadCloser
  var err error
  if box ≠ nil {
    f, err = os.Open(fmt.Sprintf("%s/mail/%s.signature", home, box.name))
  }
  if err ≠ nil ∨ f ≡ nil {
    f, err = os.Open(fmt.Sprintf("%s/mail/signature", home))
  }
  if err ≡ nil {
    w.Write([]byte("\n"))
    b := bufio.NewReader(f)
    for buf, err := b.ReadBytes('\n'); err ≡ nil ∨ err ≡ io.EOF; buf, err = b.ReadBytes('\n') {
      w.Write(buf)
      if err ≡ io.EOF {
        break
      }
    }
    f.Close()
  }
  ⟨ Go to top of window w 96 ⟩
}
```

252. Index.

⟨ Get *level* for *msg* 136 ⟩ Used in section 170.
⟨ Get *mgs* with the same box like *v.msg.box* or the first one 131 ⟩ Used in sections 128, 130, and 133.
⟨ Get *parent* for *msg* 132 ⟩ Used in sections 189, 207, 209, 210, and 212.
⟨ Get *plan9dir* from enviroment variable 246 ⟩ Used in section 243.
⟨ Get *root* of *msg* 134 ⟩ Used in sections 62, 146, and 150.
⟨ Go to the top of window for first 100 messages 167 ⟩ Used in section 165.
⟨ Go to top of window *w* 96 ⟩ Used in sections 44, 46, 167, 204, and 251.
⟨ Handling of other types of action 103, 116 ⟩ Used in section 195.
⟨ Imports 11, 13, 15, 28, 31, 39, 47, 93, 122, 214, 229, 233 ⟩ Used in section 2.
⟨ Increase the windows count 8 ⟩ Used in sections 41 and 75.
⟨ Inform *box* to create a window 74 ⟩ Used in sections 2 and 45.
⟨ Inform *box* to print a full thread with *msg* 150 ⟩ Used in section 78.
⟨ Inform *box* to print counting messages 162 ⟩ Used in section 49.
⟨ Inform *box* to print messages 146 ⟩ Used in sections 61, 75, and 78.
⟨ Inform *box* to print the rest of counting messages 163 ⟩ Used in section 49.
⟨ Inform *box* to print *msgs* 152 ⟩ Used in section 62.
⟨ Inform *box* to print *msg* 151 ⟩ Used in section 62.
⟨ Inform *box* to refresh *msgs* 154 ⟩ Used in section 171.
⟨ Init root of `mailfs` 25 ⟩ Used in section 2.
⟨ Looking for a *name* mailbox, storing an index of the mail box was found in *i*, continue if not found 38 ⟩ Used in sections 34 and 88.
⟨ Make a full thread in *msgs* for *root* 147 ⟩ Used in sections 62, 146, 150, and 171.
⟨ Mark messages as seen 113 ⟩ Used in section 78.
⟨ Mark to delete messages 98 ⟩ Used in section 78.
⟨ Mark to delete *id* message 104 ⟩ Used in section 103.
⟨ On exit? 4 ⟩ Used in sections 32, 33, 34, 35, 49, 61, and 124.
⟨ Open selected messages 79 ⟩ Used in section 78.
⟨ Other kinds of actions 100, 114 ⟩ Used in section 84.
⟨ Parse command line arguments 12 ⟩ Used in section 2.
⟨ Print all mailboxes 44 ⟩ Used in section 35.
⟨ Print messages from *v.msgs* 165 ⟩ Used in section 143.
⟨ Print the *name* for window *w* 53 ⟩ Used in sections 33, 41, 78, 180, 190, 204, and 237.
⟨ Print *msg* at exact positon 156 ⟩ Used in section 62.
⟨ Process events are specific for *boxes* 34 ⟩ Used in section 2.
⟨ Processing of other channels 45 ⟩ Used in section 35.
⟨ Processing of other common channels 7, 88, 111, 158 ⟩ Used in section 34.
⟨ Processing of other *box* channels 62, 63, 75, 78, 109, 143, 195 ⟩ Used in sections 49 and 61.
⟨ Quote a html message 242 ⟩ Used in section 240.
⟨ Quote a message 241 ⟩ Used in sections 240 and 242.
⟨ Quote name of mailbox if it is necessary 43 ⟩ Used in sections 44 and 46.
⟨ Read a message number 81 ⟩ Used in sections 78, 80, and 97.
⟨ Read a tag of *w* into *s* 174 ⟩ Used in section 173.
⟨ Read message numbers 80 ⟩ Used in sections 79 and 97.
⟨ Read other fields of a message 95, 123, 202 ⟩ Used in section 30.
⟨ Refresh main window for a box *b* 46 ⟩ Used in section 35.
⟨ Refresh the message's view 197 ⟩ Used in sections 116 and 195.
⟨ Refresh *children* 171 ⟩ Used in section 109.
⟨ Refresh *msgs* 155 ⟩ Used in sections 103, 116, and 195.
⟨ Refresh *msg* 153 ⟩ Used in section 212.
⟨ Remove *id* message from *unread* 196 ⟩ Used in sections 116 and 195.
⟨ Remove *msg* from *src* 148 ⟩ Used in sections 146, 147, and 149.
⟨ Rest of initialization of *mailbox* 22, 73, 108, 142, 193 ⟩ Used in section 36.

⟨ Rest of initialization of *message* 217 ⟩ Used in section 30.
⟨ Rest of *mailbox* members 21, 48, 72, 77, 107, 141, 159, 179, 192 ⟩ Used in section 19.
⟨ Rest of *message* members 29, 64, 94, 121, 194, 201, 216 ⟩ Used in section 19.
⟨ Save a message 236 ⟩ Used in section 212.
⟨ Save stuff on disk and plumb a message to a web browser 226 ⟩ Used in section 212.
⟨ Search messages 190 ⟩ Used in section 78.
⟨ Send a rest of *msgs* 166 ⟩ Used in sections 165 and 187.
⟨ Send a signal to refresh all mailboxes 71 ⟩ Used in section 34.
⟨ Send deleted *msgs* 112 ⟩ Used in sections 67 and 106.
⟨ Send the message 243 ⟩ Used in section 239.
⟨ Send *box* to refresh the main window 70 ⟩ Used in sections 49, 62, 68, 116, and 195.
⟨ Send *children* 128 ⟩ Used in section 124.
⟨ Send *level* 135 ⟩ Used in section 124.
⟨ Send *msgs* for viewing 89 ⟩ Used in sections 78 and 212.
⟨ Send *msgs* to delete 101 ⟩ Used in section 98.
⟨ Send *msgs* to mark them seen 115 ⟩ Used in section 113.
⟨ Send *msgs* to undelete 102 ⟩ Used in section 99.
⟨ Send *parent* 130 ⟩ Used in section 124.
⟨ Send *root* 133 ⟩ Used in section 124.
⟨ Set a position for a threaded message 189 ⟩ Used in section 186.
⟨ Set window *w* to clean state 50 ⟩ Used in sections 44, 46, 78, 198, and 204.
⟨ Set window *w* to dirty state 51 ⟩ Used in section 198.
⟨ Set *pos* of *box* 161 ⟩ Used in sections 146, 162, and 163.
⟨ Skip current message 187 ⟩ Used in sections 186 and 189.
⟨ Split the tag into *tag* fields after the pipe symbol 175 ⟩ Used in section 173.
⟨ Start a collector of message identifiers 124 ⟩ Used in section 2.
⟨ Start a goroutine to process events from a composed mail window 239 ⟩ Used in section 238.
⟨ Start a goroutine to process events from the message's window 212 ⟩ Used in section 204.
⟨ Start a main message loop 35 ⟩ Used in section 2.
⟨ Start a message loop for *box* 60 ⟩ Used in sections 2 and 34.
⟨ Subscribe on notifications of `plumber` 32, 33 ⟩ Used in section 2.
⟨ Try to open `mailfs` 17 ⟩ Used in section 2.
⟨ Trying to find a place for *msg* in the *box* window 186 ⟩ Used in section 165.
⟨ Types 19, 82, 83, 118, 119, 139, 215 ⟩ Used in section 2.
⟨ Unmark messages 99 ⟩ Used in section 78.
⟨ Unmark to delete *id* message 105 ⟩ Used in section 103.
⟨ Variables 3, 6, 10, 16, 20, 23, 40, 85, 91, 110, 120, 157, 221, 230, 234, 245 ⟩ Used in section 2.
⟨ Write a name of *box* window 180 ⟩ Used in sections 76 and 78.
⟨ Write a tag of main window 172 ⟩ Used in sections 41 and 45.
⟨ Write a tag of message window 205 ⟩ Used in sections 104, 105, 204, and 212.
⟨ Write a tag of *box* window 181 ⟩ Used in sections 61, 76, 78, 165, and 198.

Amail - a mail client for Acme

(version 0.10.0)