Seminar GUI-Tools

WS 1994/95

8½ von Plan 9

Elmar Ludwig

8½ - Das Fenstersystem von Plan 9

Plan 9:

neues Betriebssystem der Bell-Labs von AT&T

– sieht in vielen Bereichen UNIX ähnlich aus
(z.B. Shell, viele Tools; aber kein *vi*)

– aber neue Konzepte

Ziele bei der Entwicklung:

- Verteiltes System für sehr große Netze
- hardware-unabhängig (viele Rechnertypen)
- sehr sicheres System:
  - kein Super-User mehr
  - Messagecodierung

Was gibt's Neues?

z.B.:

- Namensraum: Alle unter ‘‘/’’ sichtbaren Namen (z.B. für *ls*)

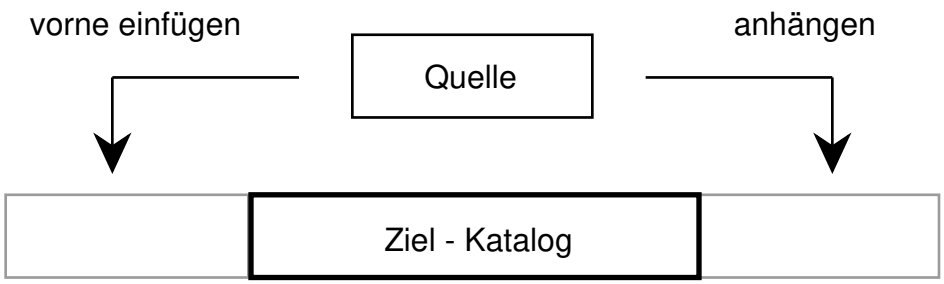
  – ist Prozeßeigenschaft und kann vererbt werden
  – ‘‘virtuelle’’ Dateien / Kataloge

  Beispiel: Namensraum im Kern

‘‘#b/...’’	Bit-Device	
‘‘#c/...’’	cons-Device	
‘‘#e/...’’	Environment	usw...

- Umbau im Namensraum mit *bind(1)*

  Quelle und Ziel sind existente Dateien / Kataloge
  Möglichkeiten: Quelle verdeckt Ziel
  Quelle vor Ziel einfügen
  Quelle an Ziel anhängen



- Erweiterung mit *mount(1)*

  Quelle ist ein File-Deskriptor,
  Zugriffe werden auf Messages abgebildet
  Möglichkeiten wie bei *bind*

  Beispiele: *8½(1)*, *ramfs(1)*, *ftpfs(1)*

- rfork (flags)

  erweitertes *fork(2)* mit Ressourcen-Kontrolle

  Flags erlauben:

  - neuen Prozeß erstellen / alten verändern
  - keine Waitmsg hinterlassen
  - neue Prozeßgruppe beginnen
  - Namensraum kopieren / löschen / teilen
  - Environment kopieren / löschen / teilen
  - Filedeskriptoren kopieren / löschen / teilen
  - Speicher gemeinsam nutzen

Grafik unter Plan 9:

Das Bit-Device ‘‘#b’’ verwaltet Bitmap-Terminal
Es übernimmt ungefähr die Aufgaben eines X-Servers:

- verwaltet das Display (Grafik, Maus)
- verwaltet Datenstrukturen (als IDs sichtbar)
  z.B. Bitmaps, Fonts, Subfonts
- liefert Maus-Events an den Clienten
- enthält den maschinenspezifischen Code

8½ verwendet 16-Bit Zeichensatz.

Repräsentierung als:

– Runes (16 Bit) oder
– UTF-String (1-3 Byte)
  (UTF = Universal Transformation Format)

  Vorteil: ASCII-Zeichen stellen sich selbst dar,
  damit Kompatibilität

Kommunikation:

Beim Anmelden werden ‘‘#b’’ und ‘‘#c’’ auf ‘‘/dev/’’ gebunden; anschließend Kommunikation durch Lese- und Schreibzugriffe auf die Dateien:

/dev/ { bitblt, mouse, mousectl } und /dev/ { cons, consctl }

- *write* in ‘‘/dev/bitblt’’ verschickt Request und löst damit Aktion aus.
- *read* von ‘‘/dev/bitblt’’ liefert (falls vorgesehen) ein Resultat.
- *read* von ‘‘/dev/mouse’’ wartet, bis Maus-Event verfügbar ist und liefert diesen zurück.

z.B.: Punkt setzen ‘‘p id[2] pt[8] value[1] code[2]’’

```
void point (Bitmap *map, Point pt, int val, Fcode code)
{
    uchar *buf = bneed(14);

    buf[0] = 'p';
    BPSHORT(buf + 1, map->id);
    BPLONG(buf + 3, pt.x), BPLONG(buf + 7, pt.y);
    buf[11] = val;
    BPSHORT(buf + 12, code);
}
```

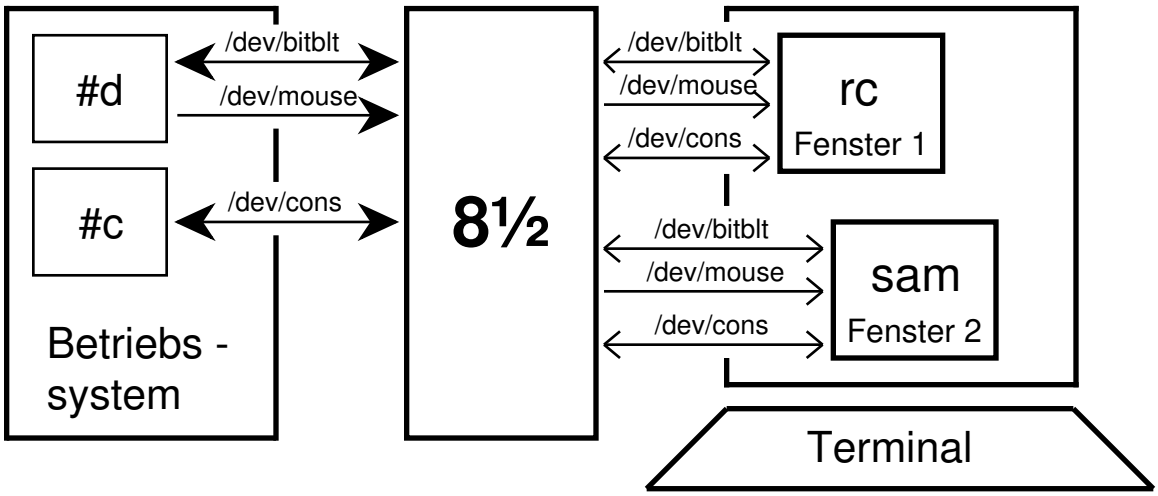
Linie zeichnen ‘‘l id[2] pt1[8] pt2[8] value[1] code[2]’’
Maus-Event (*read*) ‘‘m buttons[1] x[4] y[4] msec[4]’’

analog für das cons-Device:

- *read* von ‘‘/dev/cons’’ blockiert, bis Zeichen verfügbar sind.
- *write* in ‘‘/dev/cons’’ produziert Ausgabe auf der Console.
- *write* in ‘‘/dev/consctl’’ schaltet *raw-mode* an/aus

8½ - Das Fenstersystem

8½ verwaltet viele Fenster auf einem Display und übernimmt die Funktionen des Windowmanagers und eines einfachen Terminalemulators.
8½ selbst enthält keinen maschinenspezifischen Code mehr, die meisten Requests werden nur weitergereicht:



Über Menüpunkte kann man:

- neue Fenster öffnen
- existente verschieben / vergrößern / verkleinern
- Fenster löschen (hangup note an die Prozeßgruppe im Fenster)
- Fenster verstecken

Die Prozesse im Fenster haben ihr Fenster als lokales Display.
Voreinstellung ist ein einfaches Terminal,
wobei der Text im Fenster editierbar ist (!).

Fenster werden über IDs verwaltet:

Für jedes Fenster existiert ein Katalog ‘‘/dev/windows/[ID]’’ mit Kontrolldateien. → Hardcopy mit *lpr(1)* möglich

Außerdem sind im Namensraum der Prozesse im Fenster folgende Dateien sichtbar:

/dev/bitblt	lokale Version des Bit-Devices (geclipped)
/dev/mouse	analog, Erweiterung durch 8½: – Abschaltung des Editierens beim Öffnen – Reshape-Events – keine Events, wenn der Mauszeiger außerhalb des Fensters ist
/dev/nbmouse	aktueller Mauszustand
/dev/cons	lokale Version des cons-Devices
/dev/consctl	analog, 8½ ergänzt hier *hold-mode*
/dev/label	Fenstertitel (lesen/setzen mit *read*/*write*)
/dev/select	Selektion im Fenster (nur lesen)
/dev/snarf	Text im *snarf-buffer* (lesen/setzen)
/dev/text	gesamter Text im Fenster
/dev/winid	eindeutige ID des Fensters
/dev/window	Bitmap des Fensters

Eigene Programme mit Grafik

Notwendige Schritte:
(fertig in der Bibliothek)

```
open ("/dev/bitblt", ...)
open ("/dev/mouse", ...)
open ("/dev/cons", ...)
open ("/dev/consctl", ...)
write (consctl, "rawon", ...)
```

Anschließend Entritt in eine Eventschleife,
alles Weitere geschieht Ereignis-gesteuert.

Programmierung in C:

Es gibt Bibliotheksfuntionen für:

- Initialisierung u.a.:
  binit, *bclose*, *bscreenrect*, *bflush* usw.
- Bitmaps:
  balloc, *bfree*, *readbitmapfile*, usw.
- Grafikausgabe:
  bitblt, *point*, *segment*, *circle*, *disc*, *arc*,
  ellipse, *texture*, *border*, *string* usw.
- Manipulation des Mauszeigers:
  cursorset, *cursorswitch*

– Events:

einit, *estart*	- Initialisierung
event, *emouse*, *ekbd*, *eread*	- Events lesen
ecanmouse, *ecankbd*, *ecanread*	- Test auf Verfügbarkeit

Die Eventstruktur in C:

```
typedef struct Event
{
        int kdbc;               /* gelesenes Zeichen (als Rune) */
        Mouse mouse;            /* enthält Buttons, Position, Zeit */
        int n;
        uchar data [...];       /* für asynchrone Leseoperationen */
} Event;
```

Events werden aus ‘‘dev/mouse’’ und ‘‘/dev/cons’’ gelesen.

Ausgabefunktionen schreiben gepuffert in ‘‘/dev/bitblt’’, explizites Schreiben des Puffers ist mit *bflush()* möglich.

```
Beispiel:  Events lesen

void main (void)
{
    Event event;
    ulong key;

    binit(NULL, NULL, "Event Tester");
    einit(Emouse|Ekeyboard);
    while (key = event(&event))
    {
        switch (key)
        {
            case Emouse    : ....     /* Maus-Event */
                             break;
            case Ekeyboard : ...      /* Keyboard-Event */
                             break;
        }
    }
    exits(NULL);
}
```

Eine Alternative: ALEF

Für viele Toplevel-Fenster werden viele Prozesse benötigt,
dafür ist C nicht gut geeignet.

ALEF (von Phil Winterbottom) ist eine Compilersprache,
die dieses Problem löst.

ALEF verwendet eine C ähnliche Syntax (mit Einschränkungen).

neu ist:
- Abstrakte Datentypen mit privaten Variablen
- Exception-Behandlung
- Multi-Thread Programmierung

  Es gibt Prozesse :
  - viele parallele Programmzähler
  - laufen unabhängig voneinander
  - nur Zugriff auf lokalen Speicher

  und Tasks :
  - Coroutine in einem Prozeß
  - nicht von anderen Tasks unterbrechbar
  - Zugriff auf gemeinsamen Speicher
- Kommunikation über Channels

  Informationsaustausch zwischen Tasks über gemeinsame Variablen oder Message-Übergabe; bei Prozessen immer über Message-Übergabe
- Synchronisation über Locks

Syntax

- Kommunikation über Channels:

```
chan (type) [buffers]opt name;
alloc name;
```

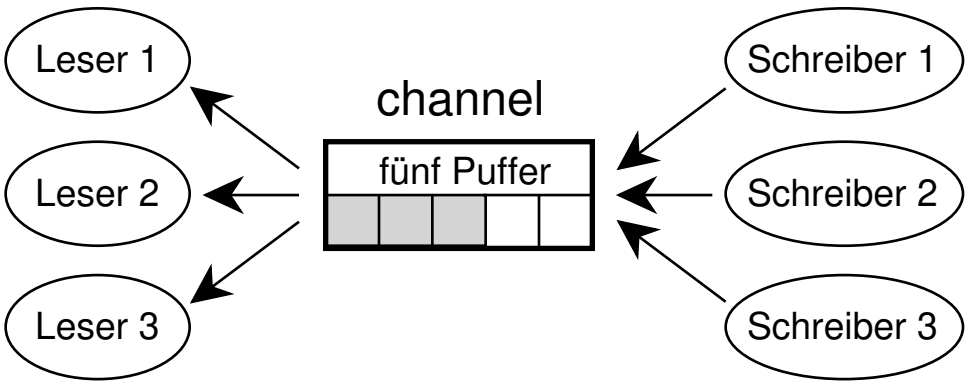
muß dynamisch mit *alloc* erzeugt werden (Freigabe mit *unalloc*)

Channel ohne Puffer: synchroner Kommunikationskanal

Der Leser / Schreiber wird blockiert, bis ein Partner da ist; dann wird der Wert übergeben.

Channel mit Puffer: asynchroner Kommunikationskanal

Der Leser / Schreiber wird nur blockiert, wenn alle Puffer leer (beim Lesen) bzw. belegt (beim Schreiben) sind.



Senden: `channel <-= value`
(Test mit `channel?`, ob Schreiber blockiert wird)

Empfangen: `var = <- channel`
(Test mit `?channel`, ob ein Wert verfügbar ist)

Es geht auch parallel mit vielen Channels:

```
alt
{
        case expr1: ...        expr1...exprn müssen send oder
        case exprn: ...        receive Operation enthalten
}
```

Dabei können die Ausdrücke bei *case* mehrfach bewertet werden!

- Erzeugen von Prozessen / Tasks:

```
proc function (parameters);
task function (parameters);
```

Prozeß / Task führt die Funktion aus und endet dann.

```
par  { statements }
```

Für jedes Statement im Block wird ein neuer Prozeß erzeugt.
Der Prozeß nach *par {...}* ist derselbe wie vorher.

- Synchronisation:

`QLock lock;`	Lock anlegen
`lock.lock();`	blockiert, bis *lock* belegt werden kann
`lock.unlock();`	wieder freigeben

ALEF und Grafik

Es gibt wie in C fertige Funktionen für Initialisierung, Bitmaps und Grafikausgabe, aber nicht für die Event-Bearbeitung.

Idee:
- je einen Prozeß erzeugen, der Maus- bzw. Keyboard-Events liest und über einen Channel an die Eventschleife liefert
- Eventschleife mit *alt {...}*

```
Beispiel:  Events lesen

void keybd (chan (Mesg) req)
{
    Mesg msg;
    char buf[128], *ptr;
    int fd, fdc, n, w, kbdcnt;

    if ((fd = open("/dev/cons", OREAD)) < 0) error(...);
    if ((fdc = open("/dev/consctl", OWRITE)) < 0) error(...);
    write(fdc, "rawon", 5);

    ...
    for (;;)
    {
       if ((n = read(fd, ptr, sizeof buf - kbdcnt)) < 0) break;
       ...
       while (kbdcnt && fullrune(buf, kbdcnt))
       {
           ...
           req <-= msg;
       }
    }
}
```

```
void mouse (chan (Mesg) req)
{
    Mesg msg;
    int fd;
    char buf[128];

    if ((fd = open("/dev/mouse", OREAD)) < 0) error(...);
    ...
    for (;;)
    {
       if (read(fd, buf, sizeof buf) < 0) break;
       ...
       req <-= msg;
    }
}

void main (void)
{
    alloc mreq, kreq;

    rfork(RFNOTEG);           /* neue Prozessgruppe */
    proc mouse(mreq);         /* Prozesse erzeugen */
    proc keybd(kreq);
    binit(error, nil, "Event Tester");

    for (;;)
    {
       Mesg event;

       alt
       {
           case event = <-mreq : ...        /* Maus-Event */
                                 break;
           case event = <-kreq : ...        /* Keyboard-Event */
                                 break;
       }
    }
}
```

Grafikausgabe funktioniert genau wie in C.

Was ist mit Netzwerktransparenz?

cpu(1) stellt Verbindung zu anderem Rechner her, dabei ist der lokale Namensraum unter ‘‘/mnt/term’’ verfügbar.

Im profile wird ‘‘bind -b /mnt/term/mnt/8½ /dev’’ ausgeführt, damit führen die Dateien in ‘‘/dev’’ wieder zum lokalen Terminal.

In der Praxis funktioniert das leider nur bedingt.

Zusammenfassung:

+ viele neue Konzepte – UNIX Altlasten wurden eliminiert
– bis jetzt wenig benutzerfreundlich, wenig Software und sehr sparsame Dokumentation
+ netzwerktransparent
– es wird keine Anpassung durch den Benutzer unterstützt (z.B. Sprache, Ressourcedatenbank), daher fest codiert
± kein Tastatur-Mapping
  damit kann kein Client die Tastatur verstellen,
  aber: alle Tasten liefern hardware-spezifischen Code,
  daher keine Cursortasten, keine Zehnertastatur
– keine Möglichkeit, die Fensterposition oder -größe vom Programm aus zu ändern
+ sehr sicheres System