ToaruOS at 5 Years

A Closer Look at a Hobbyist Operating System

Outline

Outline

Background & Design Goals

"What happens when…" - deep walkthrough

Demos

Future

Questions?

Background

Background

とあるOS - Pronounced "Toe Ah Roo Oh Es"

Means something along the lines of "A Specific OS" or "Some Such OS"

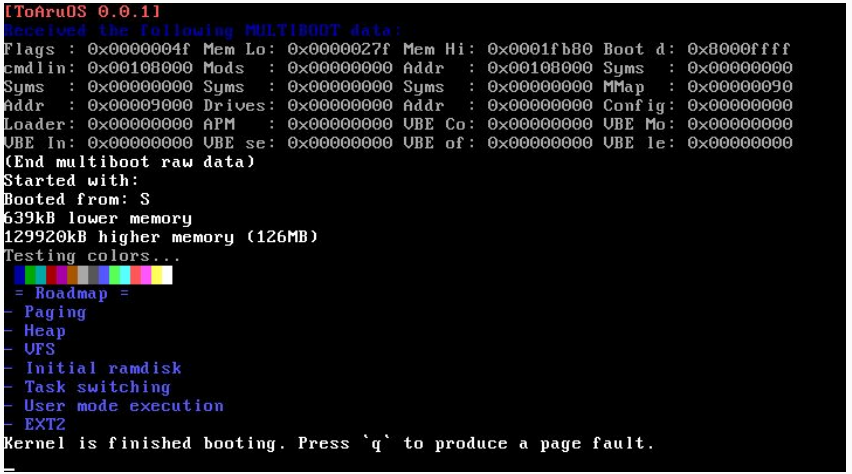
Parody of generic hobby OS names like "MyOS"

Background

Project started in December, 2010

First code commits on 2011-01-15

First screenshot from 2011-01-20



Design Goals - Then

POSIX

x86

C

Overall Design

Monolithic, modular

32-bit x86

Unix-like

Compositing window manager

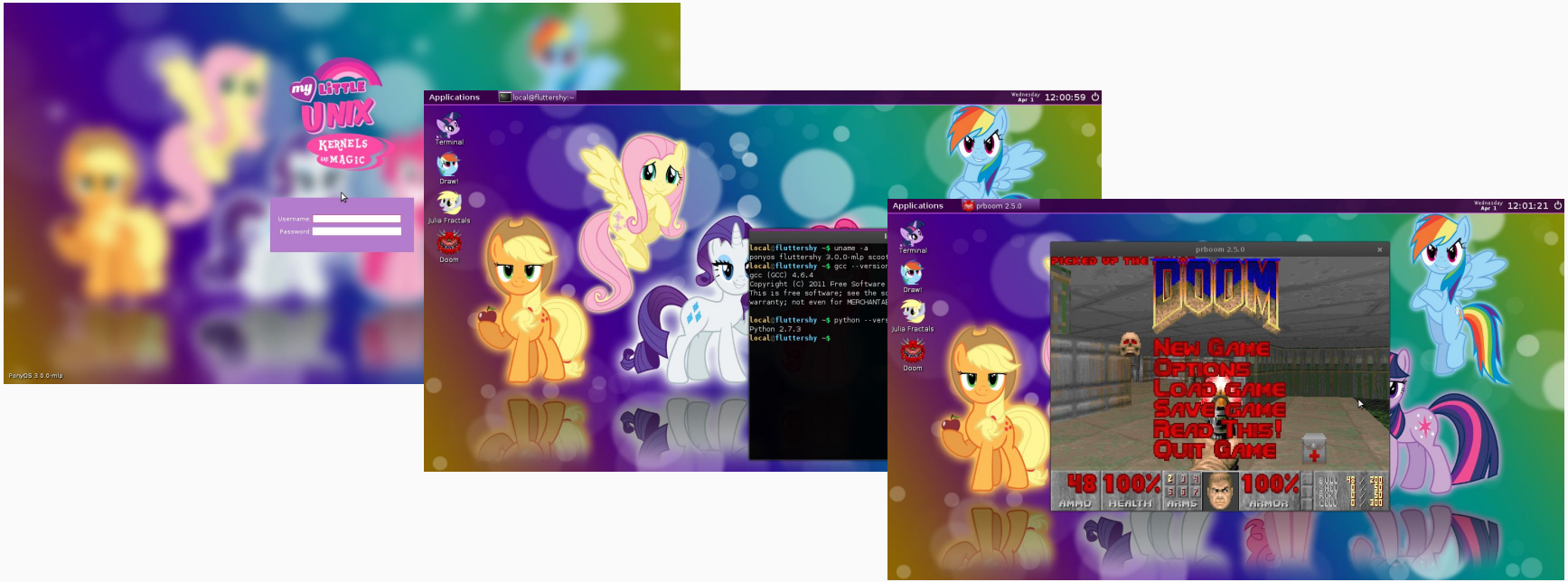
Publicity

github.com/showcases/open-source-operating-systems

phoronix.com/scan.php?page=news_item&px=MTgzMTk

osnews.com/story/28038/ToAruOS_hobby_kernel_and_userspace

April Fools: PonyOS



What happens when...

What happens when...

… you type "google.com" into your web browser and hit <enter>?

What happens when...

… you type ~~"google.com" into your web browser~~ and hit <enter>?

What happens when...

... you type `` `fetch http://toaruos.org/docs/talk.pdf` `` in a shell and hit <enter>?

You are standing in an open field...

First, let's talk about the initial state of our system.

We'll assume we've already opened a terminal, the shell is running, and it is waiting for our input.

But what does that look like in code?

First off, let's talk about the shell.

ToaruOS's shell implements complex line editing using a function called `rline`, which you can see to the right.

Since we're waiting for input, `rline` is blocked in the call to `fgetc` on line 48.

```
28 int rline(char * buffer, int buf_size, rline_callbacks_t * calls
29     /* Initialize context */
30     rline_context_t context = {
31         buffer,
32         callbacks,
33         0,
34         buf_size,
35         0,
36         0,
37         0,
38         0,
39     };
40
41     printf("\033[s");
42     fflush(stdout);
43
44     key_event_state_t kbd_state = {0};
45
46     /* Read keys */
47     while ((context.collected < context.requested) && (!context
48         uint32_t key_sym = kbd_key(&kbd_state, fgetc(stdin));
49         if (key_sym == KEY_NONE) continue;
50         switch (key_sym) {
51             case KEY_CTRL_C:
52                 printf("^C\n");
53                 context.buffer[0] = '\0';
54                 return 0;
55             case KEY_CTRL_R:
56                 if (callbacks->rev_search) {
57                     callbacks->rev_search(&context);
58                     return context.collected;
59                 }
60                 continue;
61             case KEY_ARROW_UP:
62             case KEY_CTRL_P:
63                 if (callbacks->key_up) {
64                     callbacks->key_up(&context);
```

Skipping over the details of the C library, this call to `fgetc` eventually turns into a system call, so we'll turn our attention to the system call handler and `sys_read` in the kernel.

`sys_read` takes three arguments: a file descriptor number, a pointer to a buffer, and a length of data to read. The kernel will use the current process's file descriptor table to find the right read method to run.

```
753 void syscall_handler(struct regs * r) {
754     if (r->eax >= num_syscalls) {
755         return;
756     }
757
758     uintptr_t location = (uintptr_t)syscalls[r->eax];
759     if (!location) {
760         return;
761     }
762
763     /* Update the syscall registers for this process */
764     current_process->syscall_registers = r;
765
766     /* Call the syscall function */
767     scall_func func = (scall_func)location;
768     uint32_t ret = func(r->ebx, r->ecx, r->edx, r->esi, r->edi);
769
770     if ((current_process->syscall_registers == r) ||
771             (location != (uintptr_t)&fork && location != (uintptr_t)&clone)) {
772         r->eax = ret;
773     }
774 }
```

```
 58 static int sys_read(int fd, char * ptr, int len) {
 59     if (FD_CHECK(fd)) {
 60         PTR_VALIDATE(ptr);
 61         fs_node_t * node = FD_ENTRY(fd);
 62         uint32_t out = read_fs(node, node->offset, len, (uint8_t *)ptr);
 63         node->offset += out;
 64         return (int)out;
 65     }
 66     return -1;
```

The file descriptor that our `fgetc` call is trying to read on is 0 - standard input, which for the shell points to a *pseudo terminal device*, so our next stop is the *tty* driver, where we find `read_pty_slave`. We'll skip over some details of *tty modes* and note that `read_pty_slave` has called `ring_buffer_read`. TTYs in ToaruOS use a *ring buffer* to store data that has been written on one end but not read from the other.

```
228 uint32_t  read_pty_slave(fs_node_t * node, uint32_t offset, uint32_t size, uint8_t *buffer) {
229     pty_t * pty = (pty_t *)node->device;
230
231     if (pty->tios.c_lflag & ICANON) {
232         return ring_buffer_read(pty->in, size, buffer);
233     } else {
234         if (pty->tios.c_cc[VMIN] == 0) {
235             return ring_buffer_read(pty->in, MIN(size, ring_buffer_unread(pty->in)), buffer);
236         } else {
237             return ring_buffer_read(pty->in, MIN(pty->tios.c_cc[VMIN], size), buffer);
238         }
239     }
240 }
```

So what's our ring buffer doing? We've asked to read from an empty ring buffer, so the ring buffer needs to wait until data is available to read. It calls `sleep_on` to wait until then.

What will wake it up? Writing to the other end of the TTY, which will be done by the *terminal emulator*.

```
51 size_t ring_buffer_read(ring_buffer_t * ring_buffer, size_t size, uint8_t * buffer) {
52     size_t collected = 0;
53     while (collected == 0) {
54         spin_lock(ring_buffer->lock);
55         while (ring_buffer_unread(ring_buffer) > 0 && collected < size) {
56             buffer[collected] = ring_buffer->buffer[ring_buffer->read_ptr];
57             ring_buffer_increment_read(ring_buffer);
58             collected++;
59         }
60         spin_unlock(ring_buffer->lock);
61         wakeup_queue(ring_buffer->wait_queue_writers);
62         if (collected == 0) {
63             if (sleep_on(ring_buffer->wait_queue_readers) && ring_buffer->internal_stop) {
64                 ring_buffer->internal_stop = 0;
65                 break;
66             }
67         }
68     }
69     wakeup_queue(ring_buffer->wait_queue_writers);
70     return collected;
71 }
```

As such, our next stop in the codebase is the terminal emulator - one of ToaruOS's earliest graphical applications.

The terminal is actually doing a lot of things, but we're particularly interested in how it will receive our key press when we first type `'f'` and for that we need to look at the `handle_incoming` method.

```
1121 void * handle_incoming(void * garbage) {
1122     while (!exit_application) {
1123         yutani_msg_t * m = yutani_poll(yctx);
1124         if (m) {
1125             switch (m->type) {
1126                 case YUTANI_MSG_KEY_EVENT:
1127                     {
1128                         struct yutani_msg_key_event * ke
1129                         int ret = (ke->event.action == KE
1130                         key_event(ret, &ke->event);
1131                     }
1132                     break;
1133                 case YUTANI_MSG_WINDOW_FOCUS_CHANGE:
1134                     {
1135                         struct yutani_msg_window_focus_ch
1136                         yutani_window_t * win = hashmap_g
1137                         if (win) {
1138                             win->focused = wf->focused;
1139                             render_decors();
1140                         }
1141                     }
1142                     break;
1143                 case YUTANI_MSG_SESSION_END:
1144                     {
1145                         kill(child_pid, SIGKILL);
1146                         exit_application = 1;
1147                     }
1148                     break;
1149                 case YUTANI_MSG_RESIZE_OFFER:
1150                     {
1151                         struct yutani_msg_window_resize *
1152                         resize_finish(wr->width, wr->heig
1153                     }
1154                     break;
1155                 case YUTANI_MSG_WINDOW_MOUSE_EVENT:
```

The terminal is waiting for events from the *window compositor*, which will send messages through a ToaruOS-specific messaging system called a *packet exchange*.

That means we need to look at another application to see how our key presses get to the shell, so next we'll look at the compositor.

```
1121 void * handle_incoming(void * garbage) {
1122     while (!exit_application) {
1123         yutani_msg_t * m = yutani_poll(yctx);
1124         if (m) {
1125             switch (m->type) {
1126                 case YUTANI_MSG_KEY_EVENT:
1127                     {
1128                         struct yutani_msg_key_event * ke
1129                         int ret = (ke->event.action == KE
1130                         key_event(ret, &ke->event);
1131                     }
1132                     break;
1133                 case YUTANI_MSG_WINDOW_FOCUS_CHANGE:
1134                     {
1135                         struct yutani_msg_window_focus_ch
1136                         yutani_window_t * win = hashmap_g
1137                         if (win) {
1138                             win->focused = wf->focused;
1139                             render_decors();
1140                         }
1141                     }
1142                     break;
1143                 case YUTANI_MSG_SESSION_END:
1144                     {
1145                         kill(child_pid, SIGKILL);
1146                         exit_application = 1;
1147                     }
1148                     break;
1149                 case YUTANI_MSG_RESIZE_OFFER:
1150                     {
1151                         struct yutani_msg_window_resize *
1152                         resize_finish(wr->width, wr->heig
1153                     }
1154                     break;
1155                 case YUTANI_MSG_WINDOW_MOUSE_EVENT:
```

The compositor performs many functions, but we'll focus on how it handles keyboard input for now.

The compositor runs a thread to read scancodes from the *keyboard device file* and turns them into messages which it sends back to another thread.

The `read` call on line 562 directly translates to a read system call that is very similar to the one from the shell.

```
546 /**
547  * Keyboard input thread
548  *
549  * Reads the kernel keyboard device and converts
550  * key presses into event objects to send to the
551  * core compositor.
552  */
553 void * keyboard_input(void * garbage) {
554     int kfd = open("/dev/kbd", O_RDONLY);
555
556     yutani_t * y = yutani_init();
557     key_event_t event;
558     key_event_state_t state = {0};
559
560     while (1) {
561         char buf[1];
562         int r = read(kfd, buf, 1);
563         if (r > 0) {
564             kbd_scancode(&state, buf[0], &event);
565             yutani_msg_t * m = yutani_msg_build_key_event(0, &event, &state);
566             int result = yutani_msg_send(y, m);
567             free(m);
568         }
569     }
570 }
```

Hurry up and wait...

So our shell, terminal, and compositor are all sitting around *waiting* for other things to happen.

Let's see how things unfold when we hit that first key: *f*

As soon as we hit the key, a *hardware interrupt* triggers on the CPU. Long before our shell ever started running, the *PS/2 keyboard driver* set up an *interrupt handler*, and so the CPU will call our `keyboard_handler` function.

The keyboard driver is very simple and just fills a buffer with the incoming scancode, which will wake up the compositor.

```
37 static int keyboard_handler(struct regs *r) {
38     unsigned char scancode;
39     keyboard_wait();
40     scancode = inportb(KEY_DEVICE);
41     irq_ack(KEY_IRQ);
42
43     write_fs(keyboard_pipe, 0, 1, (uint8_t []){scancode});
44     return 1;
45 }
46
```

Back in the compositor, our `read` call has finished and we have a scancode. We turn that scancode into a key event message and send it to the compositor's main thread as a packet exchange message with `yutani_msg_send`.

```c
/**
 * Keyboard input thread
 *
 * Reads the kernel keyboard device and converts
 * key presses into event objects to send to the
 * core compositor.
 */
void * keyboard_input(void * garbage) {
    int kfd = open("/dev/kbd", O_RDONLY);

    yutani_t * y = yutani_init();
    key_event_t event;
    key_event_state_t state = {0};

    while (1) {
        char buf[1];
        int r = read(kfd, buf, 1);
        if (r > 0) {
            kbd_scancode(&state, buf[0], &event);
            yutani_msg_t * m = yutani_msg_build_key_event(0, &event, &state);
            int result = yutani_msg_send(y, m);
            free(m);
        }
    }
}
```

The compositor's main event handler thread is waiting to receive messages, both from the keyboard thread and from all of the client applications that are communicating with it.

It will see the key event and then pass it off to `handle_key_event`.

There, it will be checked against various window management keybindings and eventually passed off to the focused window's application.

```
2014
2015      while (1) {
2016          pex_packet_t * p = calloc(PACKET_SIZE, 1);
2017          pex_listen(server, p);
2018
2019          yutani_msg_t * m = (yutani_msg_t *)p->data;
          ...
2086              case YUTANI_MSG_KEY_EVENT:
2087                  {
2088                      /* XXX Verify this is from a valid device client */
2089                      struct yutani_msg_key_event * ke = (void *)m->data;
2090                      handle_key_event(yg, ke);
2091                  }
2092                  break;
          ...
```

```
1428 static void handle_key_event(yutani_globals_t * yg, struct yutani_msg_key_event * ke) {
1429     yutani_server_window_t * focused = get_focused(yg);
1430     memcpy(&yg->kbd_state, &ke->state, sizeof(key_event_state_t));
1431     if (focused) {
1432         if ((ke->event.action == KEY_ACTION_DOWN) &&
1433             (ke->event.modifiers & KEY_MOD_LEFT_CTRL) &&
1434             (ke->event.modifiers & KEY_MOD_LEFT_SHIFT) &&
1435             (ke->event.keycode == 'z')) {
1436             mark_window(yg,focused);
1437             focused->rotation -= 5;
1438             mark_window(yg,focused);
1439             return;
              ...
1583     /* Finally, send the key to the focused client. */
1584     if (focused) {
1585
1586         yutani_msg_t * response = yutani_msg_build_key_event(focused->wid, &ke->event, &ke->state);
1587         pex_send(yg->server, focused->owner, response->size, (char *)response);
1588         free(response);
1589
1590     }
1591 }
```

… which brings us back to the terminal. Our call to `yutani_poll` returns, giving us a `KEY_EVENT`, which we then pass to the appropriately-named `key_event` for further processing.

```
1121 void * handle_incoming(void * garbage) {
1122     while (!exit_application) {
1123         yutani_msg_t * m = yutani_poll(yctx);
1124         if (m) {
1125             switch (m->type) {
1126                 case YUTANI_MSG_KEY_EVENT:
1127                     {
1128                         struct yutani_msg_key_event * ke
1129                         int ret = (ke->event.action == KE
1130                         key_event(ret, &ke->event);
1131                     }
1132                     break;
1133                 case YUTANI_MSG_WINDOW_FOCUS_CHANGE:
1134                     {
1135                         struct yutani_msg_window_focus_ch
1136                         yutani_window_t * win = hashmap_g
1137                         if (win) {
1138                             win->focused = wf->focused;
1139                             render_decors();
1140                         }
1141                     }
1142                     break;
1143                 case YUTANI_MSG_SESSION_END:
1144                     {
1145                         kill(child_pid, SIGKILL);
1146                         exit_application = 1;
1147                     }
1148                     break;
1149                 case YUTANI_MSG_RESIZE_OFFER:
1150                     {
1151                         struct yutani_msg_window_resize *
1152                         resize_finish(wr->width, wr->heig
1153                     }
1154                     break;
1155                 case YUTANI_MSG_WINDOW_MOUSE_EVENT:
```

`key_event` checks if we need to do special processing before writing the key to the *TTY master*. Many keys, such as the *function keys and arrow keys*, need to be encoded as special *escape sequences* as they can't be represented as single bytes.

Our f key can just be sent as the character "f", though, so we call `handle_input('f')` which will write to our TTY.

```
832 void key_event(int ret, key_event_t * event) {
833     if (ret) {
834         if (event->modifiers & KEY_MOD_LEFT_ALT || event->modifiers & KEY_MOD_RIGHT_ALT) {
835             handle_input('\033');
836         }
837         if ((event->modifiers & KEY_MOD_LEFT_SHIFT || event->modifiers & KEY_MOD_RIGHT_SHIFT) &&
838             event->key == '\t') {
839             handle_input_s("\033[Z");
840             return;
841         }
842         handle_input(event->key);
843     } else {
844         if (event->action == KEY_ACTION_UP) return;
845         switch (event->keycode) {
846             case KEY_F1:
847                 handle_input_s("\033OP");
848                 break;
849             case KEY_F2:
850                 handle_input_s("\033OQ");
851                 break;
852             case KEY_F3:
853                 handle_input_s("\033OR");
```

As there is now data available from our TTY, we can go back to the shell, which will finally return from `fgetc`, receiving the letter f.

To support advanced line editing like cursor movements, the shell needs to figure out if the data it is receiving is raw characters or escape sequences, so it runs everything through a state machine. Luckily, our f needs no special processing, so we can continue.

```
int rline(char * buffer, int buf_size, rline_callbacks_t * callb
    /* Initialize context */
    rline_context_t context = {
        buffer,
        callbacks,
        0,
        buf_size,
        0,
        0,
        0,
        0,
    };

    printf("\033[s");
    fflush(stdout);

    key_event_state_t kbd_state = {0};

    /* Read keys */
    while ((context.collected < context.requested) && (!context
        uint32_t key_sym = kbd_key(&kbd_state, fgetc(stdin));
        if (key_sym == KEY_NONE) continue;
        switch (key_sym) {
            case KEY_CTRL_C:
                printf("^C\n");
                context.buffer[0] = '\0';
                return 0;
            case KEY_CTRL_R:
                if (callbacks->rev_search) {
                    callbacks->rev_search(&context);
                    return context.collected;
                }
                continue;
            case KEY_ARROW_UP:
            case KEY_CTRL_P:
                if (callbacks->key_up) {
                    callbacks->key_up(&context);
```

Something we neglected to mention earlier, as we glossed over TTY modes, is that our TTY is set to not automatically echo the characters we type as we type them. This is important in supporting line editing, and means we need to write back out the character we received so it shows up on the terminal. We also need to add the `f` to an internal buffer.

```
237         } else {
238             printf("%c", (char)key_sym);
239             if (context.collected < context.requested) {
240                 context.buffer[context.collected] = (char)key_sym;
241                 context.buffer[++context.collected] = '\0';
242                 context.offset++;
243             }
244             fflush(stdout);
245         }
```

Since we've printed a character, we now need to go back up the TTY to the terminal again! Since we've already seen some of the details of the TTY device, we'll skip all the way to the terminal itself.

A thread in the terminal is always reading from the TTY master, and will receive our `f` and run it through the *ANSI escape sequence processor*.

```
1417        unsigned char buf[1024];
1418        while (!exit_application) {
1419            int r = read(fd_master, buf, 1024);
1420            spin_lock(&display_lock);
1421            for (uint32_t i = 0; i < r; ++i) {
1422                ansi_put(ansi_state, buf[i]);
1423            }
1424            display_flip();
1425            spin_unlock(&display_lock);
1426        }
```

Terminals are complicated. In the early days of digital computer text, only basic black-and-white character cell displays were available, and they only supported a handful of characters.

These days, terminals support full-color Unicode text, hundreds of formatting options, cursor movement...

These formatting effects are made possible through complex escape sequences.

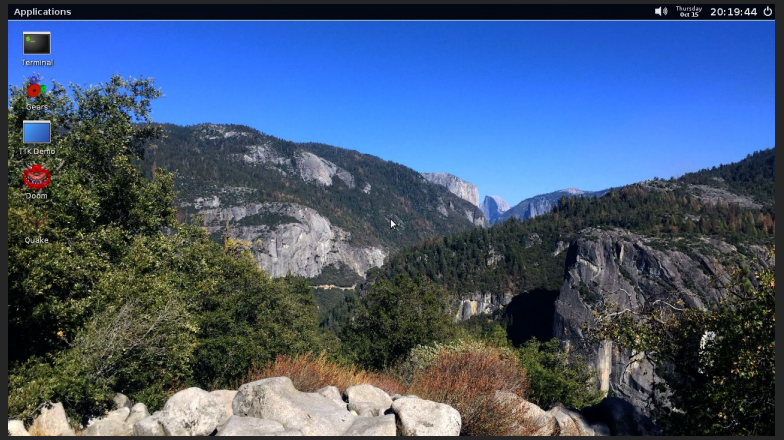
We're going to skip all of that.

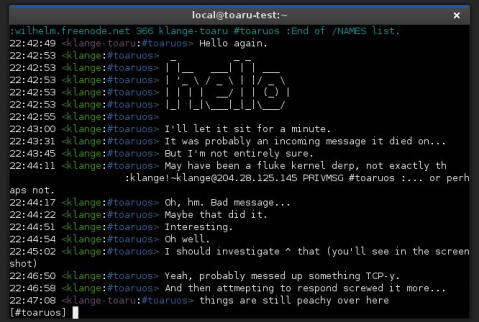
At the end of the day, the escape processor will eventually call two functions: `cell_set` and `cell_redraw`. The first of these stores the `f` and all of its formatting hints into a data structure representing the terminal display, and the latter actually draws the glyph for `f` into the window for the terminal.

```
395 static void cell_set(uint16_t x, uint16_t y, uint32_t c, uint32_t fg, uint32_t bg, uint8_t flags) {
396     if (x >= term_width || y >= term_height) return;
397     term_cell_t * cell = (term_cell_t *)((uintptr_t)term_buffer + (y * term_width + x) * sizeof(term_cell_t));
398     cell->c     = c;
399     cell->fg    = fg;
400     cell->bg    = bg;
401     cell->flags = flags;
402 }
403
404 static void cell_redraw(uint16_t x, uint16_t y) {
405     if (x >= term_width || y >= term_height) return;
406     term_cell_t * cell = (term_cell_t *)((uintptr_t)term_buffer + (y * term_width + x) * sizeof(term_cell_t));
407     if (((uint32_t *)cell)[0] == 0x00000000) {
408         term_write_char(' ', x * char_width, y * char_height, TERM_DEFAULT_FG, TERM_DEFAULT_BG, TERM_DEFAULT_FLAGS);
409     } else {
410         term_write_char(cell->c, x * char_width, y * char_height, cell->fg, cell->bg, cell->flags);
411     }
412 }
```

At this point, we need to take a quick break from looking at code and talk about how windows get displayed on the screen in ToaruOS.

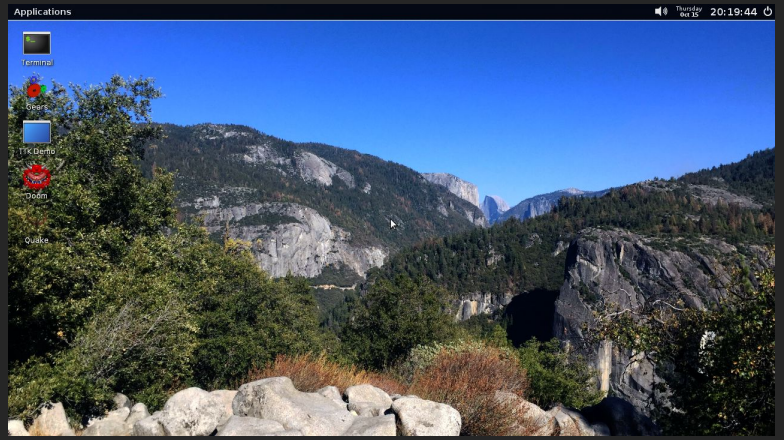
As we mentioned earlier, ToaruOS has a *window compositor*. The most important job the compositor has is to *composite* - to take all of the windows from all of the running applications and turn them into a final scene.

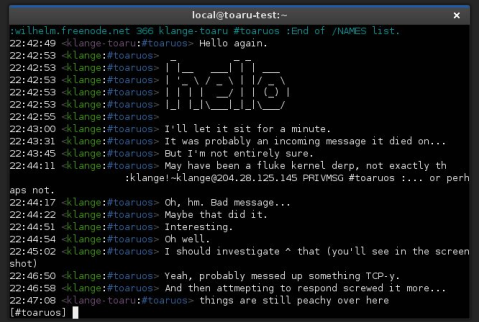




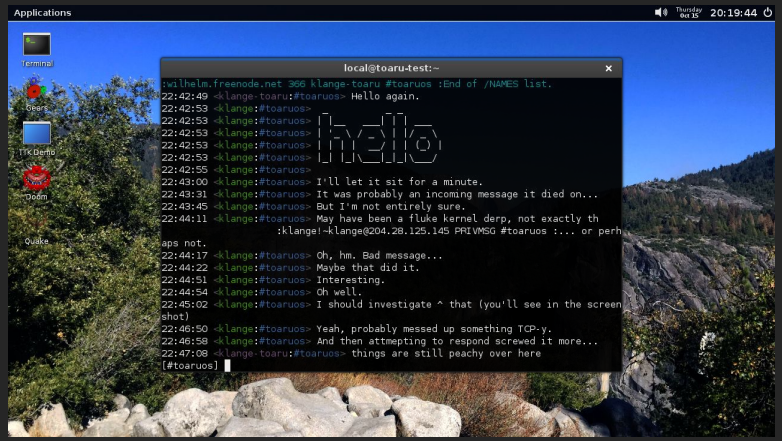
ToaruOS's window compositor is called *Yutani* (a reference to Wayland and *Alien*). Yutani represents windows as *canvases* of 32-bit RGBA pixels, which it stores in shared memory regions.

When an application wants to draw to a window, it modifies the pixels directly in memory and then informs the compositor that it should redraw the modified regions using a *flip* message.





The compositor redraws only the regions which have been modified since the last redraw, producing a final image which it then copies into video memory.



To actually draw text, ToaruOS makes use of a library called Freetype to read TrueType and OpenType fonts and draw glyphs.

Once we've drawn the glyph for our 'f', we tell the compositor to flip our modified regions.

```
196 void drawChar(FT_Bitmap * bitmap, int x, int y, uint32_t fg, uint32_t bg) {
197     int i, j, p, q;
198     int x_max = x + bitmap->width;
199     int y_max = y + bitmap->rows;
200     for (j = y, q = 0; j < y_max; j++, q++) {
201         for ( i = x, p = 0; i < x_max; i++, p++) {
202             uint32_t a = _ALP(fg);
203             a = (a * bitmap->buffer[q * bitmap->width + p]) / 255;
204             uint32_t tmp = rgba(_RED(fg),_GRE(fg),_BLU(fg),a);
205             term_set_point(i,j, alpha_blend_rgba(premultiply(bg), premultiply(tmp)));
206         }
207     }
208 }
```

```
117 static void display_flip(void) {
118     if (l_x != INT32_MAX && l_y != INT32_MAX) {
119         yutani_flip_region(yctx, window, l_x, l_y, r_x - l_x, r_y - l_y);
120         l_x = INT32_MAX;
121         l_y = INT32_MAX;
122         r_x = -1;
123         r_y = -1;
124     }
125 }
```

The compositor collects all of the screen updates and then *blits* the windows together.

```
1049 static void redraw_windows(yutani_globals_t * yg) {
1050     /* Save the cairo contexts so we can apply clipping */
1051     save_cairo_states(yg);
1052     int has_updates = 0;
1053
1054     /* We keep our own temporary mouse coordinates as they ma
1055     int tmp_mouse_x = yg->mouse_x;
1056     int tmp_mouse_y = yg->mouse_y;
1057
1058     /* If the mouse has moved, that counts as two damage regi
1059     if ((yg->last_mouse_x != tmp_mouse_x) || (yg->last_mouse_
1060         has_updates = 2;
1061         yutani_add_clip(yg, yg->last_mouse_x / MOUSE_SCALE -
1062         yutani_add_clip(yg, tmp_mouse_x / MOUSE_SCALE - MOUSE
1063     }
1064
1065     yg->last_mouse_x = tmp_mouse_x;
1066     yg->last_mouse_y = tmp_mouse_y;
1067
1068     if (yg->bottom_z && yg->bottom_z->anim_mode) mark_window(
1069     if (yg->top_z && yg->top_z->anim_mode) mark_window(yg, yg
1070     foreach (node, yg->mid_zs) {
1071         yutani_server_window_t * w = node->value;
1072         if (w && w->anim_mode) mark_window(yg, w);
1073     }
1074
1075     /* Calculate damage regions from currently queued updates
1076     spin_lock(&yg->update_list_lock);
1077     while (yg->update_list->length) {
1078         node_t * win = list_dequeue(yg->update_list);
1079         yutani_damage_rect_t * rect = (void *)win->value;
1080
1081         /* We add a clip region for each window in the update
1082         has_updates = 1;
1083         yutani_add_clip(yg, rect->x, rect->y, rect->width, re
1084         free(rect);
1085         free(win);
1086     }
1087     spin_unlock(&yg->update_list_lock);
```

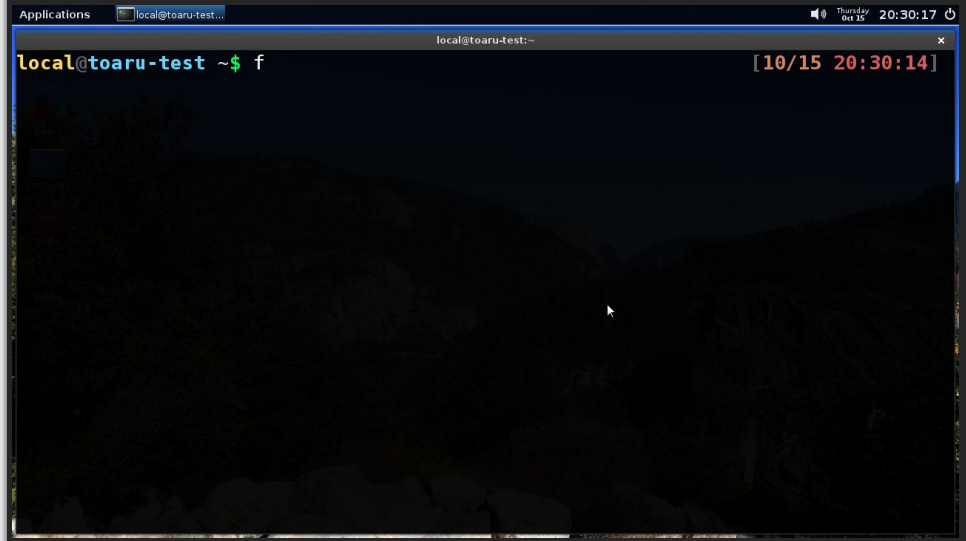
Using a library called *Cairo* to perform pixel pushing, the compositor can support transparent, rotated, and animated windows.

```
826 static int yutani_blit_window(yutani_globals_t * yg, cairo_t * ctx,
827
828     /* Obtain the previously initialized cairo contexts */
829     cairo_t * cr = ctx;
830
831     /* Window stride is always 4 bytes per pixel... */
832     int stride = window->width * 4;
833
834     /* Initialize a cairo surface object for this window */
835     cairo_surface_t * surf = cairo_image_surface_create_for_data(
836             window->buffer, CAIRO_FORMAT_ARGB32, window->width, winc
837
838     /* Save cairo context */
839     cairo_save(cr);
840
841     /*
842      * Offset the rendering context appropriately for the position c
843      * based on the modifier paramters
844      */
845     cairo_translate(cr, x, y);
846
847     /* Top and bottom windows can not be rotated. */
848     if (!window_is_top(yg, window) && !window_is_bottom(yg, window))
849         /* Calcuate radians from degrees */
850
851         /* XXX Window rotation is disabled until damage rects can ta
852         if (window->rotation != 0) {
853             double r = M_PI * (((double)window->rotation) / 180.0);
854
855             /* Rotate the render context about the center of the wir
856             cairo_translate(cr, (int)( window->width / 2), (int)( (i
857             cairo_rotate(cr, r);
858             cairo_translate(cr, (int)(-window->width / 2), (int)(-wi
859
860             /* Prefer faster filter when rendering rotated windows *
861             cairo_pattern_set_filter (cairo_get_source (cr), CAIRO_F
862         }
863     }
864     if (window->anim_mode) {
865         int frame = yutani_time_since(yg, window->anim_start);
866         if (frame >= yutani_animation_lengths[window->anim_mode]) {
867             /* XXX handle animation-end things like cleanup of closi
868             if (window->anim_mode == YUTANI_EFFECT_FADE_OUT) {
```

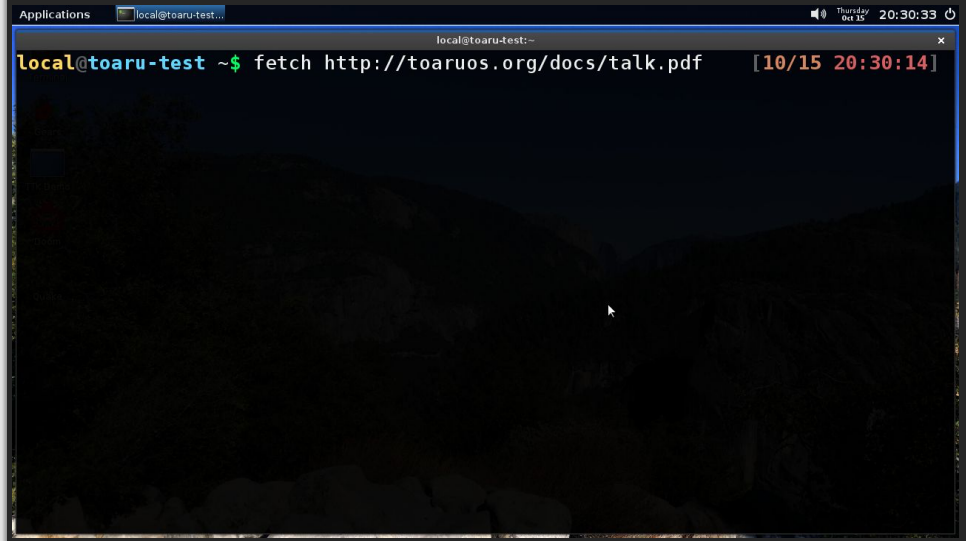
The last step in this process is to copy the final frame to video memory so it shows up on screen.

```
1151             * Flip the updated areas. This minimizes writes to video memory,
1152             * which is very important on real hardware where these writes are slow.
1153             */
1154            cairo_set_operator(yg->real_ctx, CAIRO_OPERATOR_SOURCE);
1155            cairo_translate(yg->real_ctx, 0, 0);
1156            cairo_set_source_surface(yg->real_ctx, yg->framebuffer_surface, 0, 0);
1157            cairo_paint(yg->real_ctx);
1158        }
1159        spin_unlock(&yg->redraw_lock);
```

Finally, we've typed the letter `f`!



This process repeats for the rest of our input until we hit enter.



At this point, the shell will parse our input, splitting it on spaces, expanding variables, etc.

```
526 int shell_exec(char * buffer, int buffer_size) {
527
528     /* Read previous history entries */
529     if (buffer[0] == '!') {
530         uint32_t x = atoi((char *)((uintptr_t)buffer + 1));
531         if (x <= shell_history_count) {
532             buffer = shell_history_get(x - 1);
533             buffer_size = strlen(buffer);
534         } else {
535             fprintf(stderr, "esh: !%d: event not found\n", x);
536             return 0;
537         }
538     }
539
540     char * history = malloc(strlen(buffer) + 1);
541     memcpy(history, buffer, strlen(buffer) + 1);
542
543     if (buffer[0] != ' ' && buffer[0] != '\n') {
544         shell_history_insert(history);
545     } else {
546         free(history);
547     }
548
549     char * argv[1024];
550     int tokenid = 0;
551
```

Once the shell has processed our input, it will `fork` a child process.

```
771         /* Now execute the last piece and wait on all of them */
772     } else {
773         shell_command_t func = shell_find(*arg_starts[0]);
774         if (func) {
775             return func(argcs[0], arg_starts[0]);
776         } else {
777             child_pid = fork();
778             if (!child_pid) {
779                 run_cmd(arg_starts[0]);
780             }
781         }
782     }
```

Forking is the traditional Unix method for creating new processes. Forking produces a nearly identical copy of the running process - it has the same virtual memory contents, the same instruction pointer, the same file descriptors.

```
/*
 * Fork.
 *
 * @return To the parent: PID of the child; to the child: 0
 */
uint32_t fork(void) {
    IRQ_OFF;

    uintptr_t esp, ebp;

    current_process->syscall_registers->eax = 0;

    /* Make a pointer to the parent process (us) on the stack */
    process_t * parent = (process_t *)current_process;
    assert(parent && "Forked from nothing??");
    /* Clone the current process' page directory */
    page_directory_t * directory = clone_directory(current_directory);
    assert(directory && "Could not allocate a new page directory!");
    /* Spawn a new process from this one */
    debug_print(INFO,"\033[1;32mALLOC {\033[0m");
    process_t * new_proc = spawn_process(current_process);
    debug_print(INFO,"\033[1;32m}\033[0m");
    assert(new_proc && "Could not allocate a new process!");
    /* Set the new process' page directory to clone */
    set_process_environment(new_proc, directory);

    struct regs r;
    memcpy(&r, current_process->syscall_registers, sizeof(struct regs));
    new_proc->syscall_registers = &r;

    esp = new_proc->image.stack;
    ebp = esp;

    new_proc->syscall_registers->eax = 0;

    PUSH(esp, struct regs, r);

    new_proc->thread.esp = esp;
    new_proc->thread.ebp = ebp;

    new_proc->is_tasklet = parent->is_tasklet;
```

But our new process will have a different return value for the call to `fork` than its parent, allowing it to take a different code path.

In the shell, that code path leads to a call to `execvp`. Within the C library, `execvp` is a complicated series of calls to `readdir` and `stat` to find a suitable executable within $PATH. We'll skip that and jump to what happens when we call the underlying `exec` system call.

```
771         /* Now execute the last piece and wait on all of them */
772     } else {
773         shell_command_t func = shell_find(*arg_starts[0]);
774         if (func) {
775             return func(argcs[0], arg_starts[0]);
776         } else {
777             child_pid = fork();
778             if (!child_pid) {
779                 run_cmd(arg_starts[0]);
780             }
781         }
782     }
```

```
508 void run_cmd(char ** args) {
509     int i = execvp(*args, args);
510     shell_command_t func = shell_find(*args);
511     if (func) {
512         int argc = 0;
513         while (args[argc]) {
514             argc++;
515         }
516         i = func(argc, args);
517     } else {
518         if (i != 0) {
519             fprintf(stderr, "%s: Command not found\n", *args);
520             i = 127;
521         }
522     }
523     exit(i);
524 }
```

exec needs to load and parse our binary.

ToaruOS uses a binary format called ELF, and our first task is to read that ELF file and make sure it is actually an ELF.

So let's look at how the file system works.

```
16 int exec_elf(char * path, fs_node_t * file, int argc, char ** argv, char ** env) {
17     Elf32_Header * header = (Elf32_Header *)malloc(file->length + 100);
18
19     debug_print(NOTICE, "---> Starting load.");
20     IRQ_RES;
21     read_fs(file, 0, file->length, (uint8_t *)header);
22     IRQ_OFF;
23     debug_print(NOTICE, "---> Finished load.");
24
25     current_process->name = malloc(strlen(path) + 1);
26     memcpy(current_process->name, path, strlen(path) + 1);
27
28     current_process->cmdline = argv;
29
30     /* Alright, we've read the binary, time to load the loadable sections */
31     /* Verify the magic */
32     if (    header->e_ident[0] != ELFMAG0 ||
33             header->e_ident[1] != ELFMAG1 ||
34             header->e_ident[2] != ELFMAG2 ||
35             header->e_ident[3] != ELFMAG3) {
36         /* What? This isn't an ELF... */
37         debug_print(ERROR, "Not a valid ELF executable.");
38         free(header);
39         close_fs(file);
40         return -1;
41     }
```

ToaruOS's primary on-disk filesystem is ext2, an older version of the most commonly used filesystem on Linux (ext4).

ext2 tracks files in a structure called an *inode*, and each *inode* references *blocks* that contain file data.

When we read part of a file, we need to find what blocks to look at and where they are on disk.

```
1123 static uint32_t read_ext2(fs_node_t *node, uint32_t offset, uint32_t size, uint8_t *buffer) {
1124     ext2_fs_t * this = (ext2_fs_t *)node->device;
1125     ext2_inodetable_t * inode = read_inode(this, node->inode);
1126     uint32_t end;
1127     if (inode->size == 0) return 0;
1128     if (offset + size > inode->size) {
1129         end = inode->size;
1130     } else {
1131         end = offset + size;
1132     }
1133     uint32_t start_block  = offset / this->block_size;
1134     uint32_t end_block    = end / this->block_size;
1135     uint32_t end_size     = end - end_block * this->block_size;
1136     uint32_t size_to_read = end - offset;
1137
1138     uint8_t * buf = malloc(this->block_size);
1139     if (start_block == end_block) {
1140         inode_read_block(this, inode, start_block, buf);
1141         memcpy(buffer, (uint8_t *)(((uint32_t)buf) + (offset % this->block_size)), size_to_read);
1142     } else {
1143         uint32_t block_offset;
1144         uint32_t blocks_read = 0;
1145         for (block_offset = start_block; block_offset < end_block; block_offset++, blocks_read++) {
1146             if (block_offset == start_block) {
1147                 inode_read_block(this, inode, block_offset, buf);
1148                 memcpy(buffer, (uint8_t *)(((uint32_t)buf) + (offset % this->block_size)), this->block_size - (offset % this->block_size));
1149             } else {
1150                 inode_read_block(this, inode, block_offset, buf);
1151                 memcpy(buffer + this->block_size * blocks_read - (offset % this->block_size), buf, this->block_size);
1152             }
1153         }
1154         if (end_size) {
1155             inode_read_block(this, inode, end_block, buf);
1156             memcpy(buffer + this->block_size * blocks_read - (offset % this->block_size), buf, end_size);
1157         }
1158     }
1159     free(inode);
1160     free(buf);
1161     return size_to_read;
1162 }
```

Our ext2 driver keeps recently used blocks in a cache, but if we haven't yet read the blocks we need, we'll need to read them from the underlying *block device* - in this case, an ATA hard disk.

```
78 static uint32_t read_ata(fs_node_t *node, uint32_t offset, uint32_t size, uint8_t *
79
80     struct ata_device * dev = (struct ata_device *)node->device;
81
82     unsigned int start_block = offset / ATA_SECTOR_SIZE;
83     unsigned int end_block = (offset + size - 1) / ATA_SECTOR_SIZE;
84
85     unsigned int x_offset = 0;
86
87     if (offset > ata_max_offset(dev)) {
88         return 0;
89     }
90
91     if (offset + size > ata_max_offset(dev)) {
92         unsigned int i = ata_max_offset(dev) - offset;
93         size = i;
94     }
95
96     if (offset % ATA_SECTOR_SIZE) {
97         unsigned int prefix_size = (ATA_SECTOR_SIZE - (offset % ATA_SECTOR_SIZE));
98         char * tmp = malloc(ATA_SECTOR_SIZE);
99         ata_device_read_sector(dev, start_block, (uint8_t *)tmp);
100
101         memcpy(buffer, (void *)((uintptr_t)tmp + (offset % ATA_SECTOR_SIZE)), prefi
102
103         free(tmp);
104
105         x_offset += prefix_size;
106         start_block++;
107     }
108
109     if ((offset + size)  % ATA_SECTOR_SIZE && start_block < end_block) {
110         unsigned int postfix_size = (offset + size) % ATA_SECTOR_SIZE;
111         char * tmp = malloc(ATA_SECTOR_SIZE);
112         ata_device_read_sector(dev, end_block, (uint8_t *)tmp);
113
114         memcpy((void *)((uintptr_t)buffer + size - postfix_size), tmp, postfix_size
115
116         free(tmp);
117
118         end_block--;
119     }
120
121     while (start_block <= end_block) {
122         ata_device_read_sector(dev, start_block, (uint8_t *)((uintptr_t)buffer + x_
123         x_offset += ATA_SECTOR_SIZE;
124         start_block++;
125     }
126
```

Once we've read all of the blocks from disk, we can load our binary into memory by copying sections.

When we're done loading the binary, we jump to userspace and start running our newly loaded code.

```
48    /* Load the loadable segments from the binary */
49    for (uintptr_t x = 0; x < (uint32_t)header->e_shentsize * header->e_shnum; x += header->e_sh
50        /* read a section header */
51        Elf32_Shdr * shdr = (Elf32_Shdr *)((uintptr_t)header + (header->e_shoff + x));
52        if (shdr->sh_addr) {
53            /* If this is a loadable section, load it up. */
54            if (shdr->sh_addr == 0) continue; /* skip sections that try to load to 0 */
55            if (shdr->sh_addr < current_process->image.entry) {
56                /* If this is the lowest entry point, store it for memory reasons */
57                current_process->image.entry = shdr->sh_addr;
58            }
59            if (shdr->sh_addr + shdr->sh_size - current_process->image.entry > current_process->
60                /* We also store the total size of the memory region used by the application */
61                current_process->image.size = shdr->sh_addr + shdr->sh_size - current_process->i
62            }
63            for (uintptr_t i = 0; i < shdr->sh_size + 0x2000; i += 0x1000) {
64                /* This doesn't care if we already allocated this page */
65                alloc_frame(get_page(shdr->sh_addr + i, 1, current_directory), 0, 1);
66                invalidate_tables_at(shdr->sh_addr + i);
67            }
68            if (shdr->sh_type == SHT_NOBITS) {
69                /* This is the .bss, zero it */
70                memset((void *)(shdr->sh_addr), 0x0, shdr->sh_size);
71            } else {
72                /* Copy the section into memory */
73                memcpy((void *)(shdr->sh_addr), (void *)((uintptr_t)header + shdr->sh_offset), s
74            }
75        }
76    }
77
78    /* Store the entry point to the code segment */
```

```
155    /* Go go go */
156    enter_user_jmp(entry, argc, argv_, USER_STACK_TOP);
```

After all of that work, we're now running our `fetch` binary.

`fetch` makes use of a special feature in ToaruOS: The network filesystem. Instead of creating a socket, `fetch` opens the virtual file `/dev/net/toaruos.org:80`

```c
int main(int argc, char * argv[]) {

    int opt;

    while ((opt = getopt(argc, argv, "?c:ho:")) != -1) {
        switch (opt) {
            case '?':
                return usage(argv);
            case 'c':
                fetch_options.cookie = optarg;
                break;
            case 'h':
                fetch_options.show_headers = 1;
                break;
            case 'o':
                fetch_options.output_file = optarg;
                break;
        }
    }

    if (optind >= argc) {
        return usage(argv);
    }

    struct http_req my_req;
    parse_url(argv[optind], &my_req);

    char file[100];
    sprintf(file, "/dev/net/%s", my_req.domain);

    fetch_options.out = stdout;
    if (fetch_options.output_file) {
        fetch_options.out = fopen(fetch_options.output_file, "w");
    }

    FILE * f = fopen(file,"r+");

    if (!f) {
        fprintf(stderr, "Nope.\n");
        return 1;
    }

    if (fetch_options.cookie) {
        fprintf(f,
            "GET /%s HTTP/1.0\r\n"
            "User-Agent: curl/7.35.0\r\n"
            "Host: %s\r\n"
            "Accept: */*\r\n"
            "Cookie: %s\r\n"
            "\r\n", my_req.path, my_req.domain, fetch_options.cookie);

    } else {
        fprintf(f,
            "GET /%s HTTP/1.0\r\n"
            "User-Agent: curl/7.35.0\r\n"
            "Host: %s\r\n"
            "Accept: */*\r\n"
            "\r\n", my_req.path, my_req.domain);
    }
```

In the network filesystem, we lookup "toaruos.org" and create a TCP socket. We store this socket in a *virtual file node* that we'll add to the running process's file descriptor table.

```
291 static fs_node_t * finddir_netfs(fs_node_t * node, char * name) {
292     /* Should essentially find anything. */
293     debug_print(WARNING, "Need to look up domain or check if is IP: %s", nam
294     /* Block until lookup is complete */
295
296     int port = 80;
297     char * colon;
298     if ((colon = strstr(name, ":"))) {
299         /* Port numbers */
300         *colon = '\0';
301         colon++;
302         port = atoi(colon);
303     }
304
305     uint32_t ip = 0;
306     if (is_ip(name)) {
307         debug_print(WARNING, "   IP: %x", ip_aton(name));
308         ip = ip_aton(name);
309     } else {
310         if (hashmap_has(dns_cache, name)) {
311             ip = ip_aton(hashmap_get(dns_cache, name));
312             debug_print(WARNING, "   In Cache: %s → %x", name, ip);
313         } else {
314             debug_print(WARNING, "   Still needs look up.");
315             return NULL;
316         }
317     }
318
319     fs_node_t * fnode = malloc(sizeof(fs_node_t));
320     memset(fnode, 0x00, sizeof(fs_node_t));
321     fnode->inode = 0;
322     strcpy(fnode->name, name);
323     fnode->mask = 0555;
324     fnode->flags   = FS_CHARDEVICE;
325     fnode->read    = socket_read;
326     fnode->write   = socket_write;
327     fnode->device  = (void *)net_open(SOCK_STREAM);
328
329     net_connect((struct socket *)fnode->device, ip, port);
330
331     return fnode;
332 }
```

Writing to the virtual file that the network filesystem gave us will send TCP packets.

```
469 static int net_send_tcp(struct socket *socket, uint16_t flags, uint8_t * payload, uint32_t payload_size) {
470     struct tcp_header *tcp = malloc(sizeof(struct tcp_header) + payload_size);
471 
472     tcp->source_port = htons(socket->port_recv);
473     tcp->destination_port = htons(socket->port_dest);
474     tcp->seq_number = htonl(socket->proto_sock.tcp_socket.seq_no);
475     tcp->ack_number = flags & (TCP_FLAGS_ACK) ? htonl(socket->proto_sock.tcp_socket.ack_no) : 0;
476     tcp->flags = htons(0x5000 ^ (flags & 0xFF));
477     tcp->window_size = htons(1800);
478     tcp->checksum = 0; // Fill in later
479     tcp->urgent = 0;
480 
481     if ((flags & 0xff) == TCP_FLAGS_SYN) {
482         // If only SYN set, expected ACK will be 1 despite no payload
483         socket->proto_sock.tcp_socket.seq_no += 1;
484     } else {
485         socket->proto_sock.tcp_socket.seq_no += payload_size;
486     }
487 
488     if (payload) {
489         memcpy(tcp->payload, payload, payload_size);
490     }
491 
492     return net_send_ip(socket, IPV4_PROT_TCP, tcp, sizeof(struct tcp_header) + payload_size);
493 }
```

Back in `fetch`, we write our HTTP requests headers...

```c
int main(int argc, char * argv[]) {

    int opt;

    while ((opt = getopt(argc, argv, "?c:ho:")) != -1) {
        switch (opt) {
            case '?':
                return usage(argv);
            case 'c':
                fetch_options.cookie = optarg;
                break;
            case 'h':
                fetch_options.show_headers = 1;
                break;
            case 'o':
                fetch_options.output_file = optarg;
                break;
        }
    }

    if (optind >= argc) {
        return usage(argv);
    }

    struct http_req my_req;
    parse_url(argv[optind], &my_req);

    char file[100];
    sprintf(file, "/dev/net/%s", my_req.domain);

    fetch_options.out = stdout;
    if (fetch_options.output_file) {
        fetch_options.out = fopen(fetch_options.output_file, "w");
    }

    FILE * f = fopen(file,"r+");

    if (!f) {
        fprintf(stderr, "Nope.\n");
        return 1;
    }

    if (fetch_options.cookie) {
        fprintf(f,
            "GET /%s HTTP/1.0\r\n"
            "User-Agent: curl/7.35.0\r\n"
            "Host: %s\r\n"
            "Accept: */*\r\n"
            "Cookie: %s\r\n"
            "\r\n", my_req.path, my_req.domain, fetch_options.cookie);

    } else {
        fprintf(f,
            "GET /%s HTTP/1.0\r\n"
            "User-Agent: curl/7.35.0\r\n"
            "Host: %s\r\n"
            "Accept: */*\r\n"
            "\r\n", my_req.path, my_req.domain);
    }
```

When the network card receives packets from the router, it passes them to the *network processing thread*. Each open socket has a corresponding queue of data that has yet to be read, and the network processing thread will add our newly received packet to the appropriate queue.

Reading from the virtual file will call `net_recv` to pull data out of the queue.

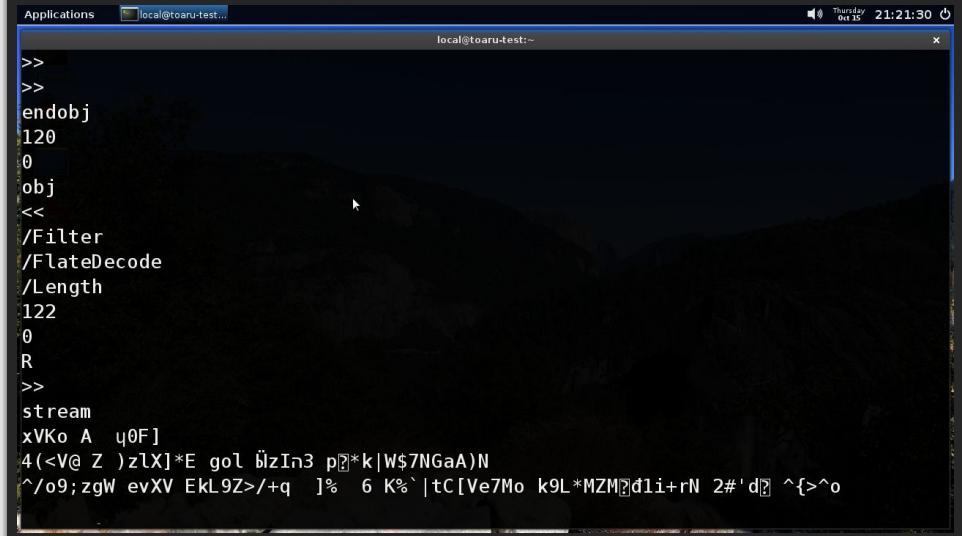
```
515 size_t net_recv(struct socket* socket, uint8_t* buffer, size_t len) {
516     tcpdata_t *tcpdata = NULL;
517     node_t *node = NULL;
518 
519     debug_print(WARNING, "0x%x [socket]", socket);
520 
521     size_t offset = 0;
522     size_t size_to_read = 0;
523 
524     do {
525 
526     if (socket->bytes_available) {
527         tcpdata = socket->current_packet;
528     } else {
529         spin_lock(socket->packet_queue_lock);
530         do {
531             if (socket->packet_queue->length > 0) {
532                 node = list_dequeue(socket->packet_queue);
533                 spin_unlock(socket->packet_queue_lock);
534                 break;
535             } else {
536                 if (socket->status == 1) {
537                     spin_unlock(socket->packet_queue_lock);
538                     debug_print(WARNING, "Socket closed, done reading.");
539                     return 0;
540                 }
541                 spin_unlock(socket->packet_queue_lock);
542                 sleep_on(socket->packet_wait);
543                 spin_lock(socket->packet_queue_lock);
544             }
545         } while (1);
546 
547         tcpdata = node->value;
548         socket->bytes_available = tcpdata->payload_size;
549         socket->bytes_read = 0;
550         free(node);
551     }
552 
553     size_to_read = MIN(len, offset + socket->bytes_available);
554 
555     if (tcpdata->payload != 0) {
556         memcpy(buffer + offset, tcpdata->payload + socket->bytes_read, size_to_read);
557     }
558 
559     offset += size_to_read;
560 
561     if (size_to_read < socket->bytes_available) {
562         socket->bytes_available = socket->bytes_available - size_to_read;
563         socket->bytes_read = size_to_read;
564         socket->current_packet = tcpdata;
```

The web server on the other end will respond, and we will read the result and print it to the terminal.

```
136     http_parser_settings settings;
137     memset(&settings, 0, sizeof(settings));
138     settings.on_header_field = callback_header_field;
139     settings.on_header_value = callback_header_value;
140     settings.on_body = callback_body;
141
142     http_parser parser;
143     http_parser_init(&parser, HTTP_RESPONSE);
144
145     while (!feof(f)) {
146         char buf[1024];
147         memset(buf, 0, sizeof(buf));
148         size_t r = fread(buf, 1, 1024, f);
149         http_parser_execute(&parser, &settings, buf, r);
150     }
151
152     fflush(fetch_options.out);
```

Turns out we just printed a PDF to our terminal, which looks like a bunch of garbage.

Oops.



Demos

Demos

Python 2.7

vim, gcc

Quake

Future

Design Goals - Now

Porting software

GUI

Clean code

Future

Now: "Misaka" x86-64 kernel project

Next year: Porting NetSurf; maybe Webkit?

Far future: glib, GTK...

Questions?

Thanks!

IRC
#toaruos on Freenode

Sites
toaruos.org
github.com/klange/toaruos

Historical Screenshots
y/toaruos-screens

Live CD
y/toaruos-iso