Developing

Tilck
A Tiny Linux-Compatible Kernel

Vladislav K. Valtchev (2022)
What Tilck is

- A project consisting on:
  - A monolithic kernel written in C and assembly
  - A bootloader working both on UEFI and legacy BIOS systems
  - Several test suites and a powerful CMake-based build system
  - Buildroot-like scripts for downloading & building 3rd party software
- Partially compatible with Linux at binary level
- Uniprocessor, but fully preemptable
- Educational, with potential to be more than that (see testing etc.)
- Runs only on i686, at the moment (will be ported to ARM, RISC-V etc.)
- Open source, distributed under the BSD 2-clause license
What Tilck is NOT

- An attempt to replace Linux
- An attempt to be yet another desktop operating system
- An attempt to be a large-scale server operating system
- A real-time OS, but it might become one in the future
- A OS running on NOMMU machines, but (probably) will in the future
- Ready for production use: it still lacks features as storage, networking etc.

Vladislav K. Valtchev (2022)
Why the binary compatibility with Linux?

- It’s cool being able to test the same “bits” both on Linux and Tilck
- Robustness: Tilck can empirically show robustness and correctness by running 3rd party software never written for it
- Didn’t want to design a whole new syscall interface from scratch
- Didn’t want to implement a whole libc too
- Didn’t want to build a custom GCC toolchain. I wanted to use the pre-built toolchains from: https://toolchains.bootlin.com/
- Increase the likelihood the project to get more interest from the community?
- Porting pre-existing software to Tilck will require a little or no effort at all.
Core values & goals

► Minimal memory footprint
► Ultra low-latency
► Deterministic behavior
► Extra robustness
► Portability
► Simplicity
► Partial compatibility with Linux
► Must work on real (modern) hardware
► Exceptional developer experience: building & testing the project should be as easy as technologically possible

Vladislav K. Valtchev (2022)
Live demo

Because a demo is worth more than a thousand words
Funny stories & interesting challenges
My latest bug [1/6]

(and its 2-char fix)

- I have a test (fork_oom) that:
  1. Estimates the amount of *committed* memory that can be used
  2. Allocates and commits more than half of that
  3. Calls fork()
  4. In the child, tries to commit *all* of that memory
  5. Expects the child to be killed by the kernel
My latest bug [1/6]

(and its 2-char fix)

I have a test (fork_oom) that:

1. Estimates the amount of committed memory that can be used
2. Allocates and commits more than half of that
3. Calls fork()
4. In the child, tries to commit all of that memory
5. Expects the child to be killed by the kernel

I just found that it fails on real HW machines

Vladislav K. Valtchev (2022)
My latest bug [1/6]
(and its 2-char fix)

- I have a test (fork_oom) that:
  1. Estimates the amount of committed memory that can be used
  2. Allocates and commits more than half of that
  3. Calls fork()
  4. In the child, tries to commit all of that memory
  5. Expects the child to be killed by the kernel
- I just found that it fails on real HW machines
- Quickly, I discovered that it fails on VMs too but only when they have significantly more RAM. That’s weird. Mmm...
My latest bug [1/6]  
(and its 2-char fix)

- I have a test (fork_oom) that:
  1. Estimates the amount of *committed* memory that can be used
  2. Allocates and commits more than half of that
  3. Calls fork()
  4. In the child, tries to commit *all* of that memory
  5. Expects the child to be killed by the kernel
- I just found that it fails on real HW machines
- Quickly, I discovered that it fails on VMs too but only when they have significantly more RAM. That’s weird. Mmm...
- I had to debug that.
That’s fine…

How could we commit so much memory? 262 MB x 2 = 524 MB > 501 MB [usable] (ehm, we don’t have swap)

That means trying to free a page not allocated in the heap, during munmap().
So, I started debugging the CoW page-fault logic...

```c
bool handle_potential_cow(void *context) {
    /* ... */
    if (!(pt->pages[pt_index].avail & PAGE_COW_ORIG_RW))
        return false; /* Not a COW page */

    const u32 orig_page_paddr = (u32)
        pt->pages[pt_index].pageAddr << PAGE_SHIFT;

    if (pf_ref_court_get(orig_page_paddr) == 1) {
        /* This page is not shared anymore. No need for copying it. */
        pt->pages[pt_index].rw = true;
        pt->pages[pt_index].avail = 0;
        invalidate_page_hw(vaddr);
        return true;
    }
    /* ... */
}
```

After committing a few MBs in the child, we end up here!
My latest bug [4/6]

I realized I had ASSERTs disabled in that build! So, after turning them on...

Aha, gotcha! You’re really trying to free the zero page!
My latest bug [5/6]

Let’s look at this limit case...

```bash
root@t1ck:/# devshell -c fork_oom
/devshell] Executing built-in command 'fork_oom'
[  parent  ] Estimating usable memory...
[child] Pid: 37
[  parent  ] Child killed by signal 9
[  parent  ] Estimated usable memory: 487 MB
Alloc 255 MB...  # Highlighted line
Write to the buffer...
Done. Now, fork()..
Child [381]: writing to the whole CoW buffer...
parent: the child exited with signal 9, as expected.
root@t1ck:/#
```

Allocating 255 MB works...

Vladislav K. Valtchev (2022)
My latest bug [6/6]

That means only one thing...

```c
2    static  u16  *pageframes_refcount;
3    static  ulong  phys_mem_lim;
4
5    static  ALWAYS_INLINE  u32  pf_ref_count_get(u32  paddr)
6    {
7        if  (UNLIKELY(paddr  >=  phys_mem_lim))
8            return  0;
9
10       return  pageframes_refcount[paddr  >>  PAGE_SHIFT];
11    }
```
That means only one thing...

That’s the problem: a 16-bit ref-count

```c
2 static u16 *pageframes_refcount;
3 static ulong phys_mem_lim;
4
5 static ALWAYS_INLINE u32 pf_ref_count_get(u32 paddr)
6 {
7     if (UNLIKELY(paddr >= phys_mem_lim))
8         return 0;
9 
10     return pageframes_refcount[paddr >> PAGE_SHIFT];
11 }
```
My latest bug [6/6]

That means only one thing...

That's the problem: a 16-bit ref-count

```
2 static u16 *pageframes_refcount;
3 static ulong phys_mem_lim;

5 static ALWAYS_INLINE u32 pf_ref_count_get(u32 paddr)
6 {
7     if (UNLIKELY(paddr >= phys_mem_lim))
8         return 0;
9     return pageframes_refcount[paddr >> PAGE_SHIFT];
11 }
```

It wraps around after 65,535 pages, meaning that the kernel cannot support 256 MB or more of uncommitted memory!
Making the framebuffer console fast
Making the framebuffer console *fast*

- Premise: why implement a framebuffer console?
  - Text mode was almost completely dead even 5 years ago
  - Pure-UEFI machines don’t support text mode
  - Text mode is a x86 thing: Raspberry PI and other machines don’t support it
Making the framebuffer console *fast*

- Premise: why implement a framebuffer console?
  - Text mode was almost completely dead even 5 years ago
  - Pure-UEFI machines don’t support text mode
  - Text mode is a x86 thing: Raspberry PI and other machines don’t support it

- Why speed matters so much? Just mark the pages as WC and it will be reasonably fast.
Making the framebuffer console *fast*

- Premise: why implement a framebuffer console?
  - Text mode was almost completely dead even 5 years ago
  - Pure-UEFI machines don’t support text mode
  - Text mode is a x86 thing: Raspberry PI and other machines don’t support it

- Why speed matters so much? Just mark the pages as WC and it will be reasonably fast.
  - I didn’t know about WC (write-combining) at the time
Making the framebuffer console fast

- Premise: why implement a framebuffer console?
  - Text mode was almost completely dead even 5 years ago
  - Pure-UEFI machines don’t support text mode
  - Text mode is a x86 thing: Raspberry PI and other machines don’t support it

- Why speed matters so much? Just mark the pages as WC and it will be reasonably fast.
  - I didn’t know about WC (write-combining) at the time
  - Therefore, I implemented a series of optimizations before discovering WC
PSF fonts: a bitfield per each glyph
The simplest draw function (failsafe)

```c
4 static inline void fb_draw_pixel(u32 x, u32 y, u32 color)
5 {
6    if (fb_bpp == 32)
7        *(volatile u32 *)(fb_vaddr + (fb_pitch * y) + (x << 2)) = color;
8    else
9        // Assumption: bpp is 24
10       memcpy((void *)(fb_vaddr + (fb_pitch * y) + (x * 3)), &color, 3);
11 }
12
13 void fb_draw_char(u32 x, u32 y, u16 e)
14 {
15    u8 *data = font_glyph_data + font_bytes_per_glyph * vgaentry_get_char(e);
16    u32 arr[] = { vga_rgb_colors[vgaentry_get_fg(e)], vga_rgb_colors[vgaentry_get_bg(e)] };
17    for (u32 row = y; row < (y + font_h); row++, data += font_width_bytes) {
18        for (u32 b = 0; b < font_width_bytes; b++) {
19            for (u32 i = 0; i < 8; i++)
20                fb_draw_pixel(x + (b << 3) + (8 - i - 1), /* x */
21                                row, /* y */
22                                arr[!(data[b] & (1 << i))]); /* color */
23        }
24    }
25 }
26
Vladislav K. Valtchev (2022)
Performance? Too slow, in particular on the modern machine (left)

16x8 font, 800x600

Intel Core i7-7500U Kaby Lake
- 1,124,773 RDTSC cycles / char (avg.)
  \[\sim 385.7 \, \mu s\]

Intel Atom N270 Diamondville (32-bit)
- 297,287 RDTSC cycles / char (avg.)
  \[\sim 186.3 \, \mu s\]

32x16 font, 3200x1800

- 7,416,012 RDTSC cycles / char (avg.)
  \[\sim 2543.2 \, \mu s\]

Scrolling the whole screen takes several seconds!!
A naïve optimization: loop unrolling

#define draw_char_partial(b) \
    do { \
        fb_draw_pixel(x + (b << 3) + 7, row, arr[!(data[b] & (1 << 0))]); \
        fb_draw_pixel(x + (b << 3) + 6, row, arr[!(data[b] & (1 << 1))]); \
        fb_draw_pixel(x + (b << 3) + 5, row, arr[!(data[b] & (1 << 2))]); \
        fb_draw_pixel(x + (b << 3) + 4, row, arr[!(data[b] & (1 << 3))]); \
        fb_draw_pixel(x + (b << 3) + 3, row, arr[!(data[b] & (1 << 4))]); \
        fb_draw_pixel(x + (b << 3) + 2, row, arr[!(data[b] & (1 << 5))]); \
        fb_draw_pixel(x + (b << 3) + 1, row, arr[!(data[b] & (1 << 6))]); \
        fb_draw_pixel(x + (b << 3) + 0, row, arr[!(data[b] & (1 << 7))]); \
    } while (0)

void fb_draw_char(u32 x, u32 y, u16 e) 
{
    u8 *data = font_glyph_data + font_bytes_per_glyph * vgaentry_get_char(e);
    u32 arr[] = { vga_rgb_colors[vgaentry_get_fg(e)], vga_rgb_colors[vgaentry_get_bg(e)] };

    if (LIKELY(font_width_bytes == 1))
        for (u32 row = y; row < (y+font_h); row++, data += font_width_bytes)
            draw_char_partial(0);
    else if (font_width_bytes == 2)
        for (u32 row = y; row < (y+font_h); row++, data += font_width_bytes) {
            draw_char_partial(0);
            draw_char_partial(1);
        }
    else
        for (u32 row = y; row < (y+font_h); row++, data += font_width_bytes)
            for (u32 b = 0; b < font_width_bytes; b++)
                draw_char_partial(b);
Benefits? Nah.

<table>
<thead>
<tr>
<th></th>
<th>Intel Core i7-7500U Kaby Lake</th>
<th>Intel Atom N270 Diamondville (32-bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before (avg.)</td>
<td>385.72 μs / char</td>
<td>186.27 μs / char</td>
</tr>
<tr>
<td>After (avg.)</td>
<td>384.44 μs / char</td>
<td>175.30 μs / char</td>
</tr>
<tr>
<td>Speed up</td>
<td>0.3% faster</td>
<td>6.2% faster</td>
</tr>
</tbody>
</table>

Old school optimizations work better on old school machines!

Vladislav K. Valtchev (2022)
Intuition 1: rendering glyphs pixel by pixel is too slow
Solution 1: pre-rendering!

- But... is pre-rendering every glyph in the font even feasible?
Pre-rendering! (font 16x8)

16 x 8 x 4 x 256 x 16 x 16 =

<table>
<thead>
<tr>
<th>Height x Width</th>
<th># glyphs</th>
<th>FG colors</th>
<th>BG colors</th>
</tr>
</thead>
</table>

Bytes per pixel

32 MB: unfeasible!
Pre-rendering! (font 32x16)

$$32 \times 16 \times 4 \times 256 \times 16 \times 16 =$$

- Height x Width
- # glyphs
- FG colors
- BG colors
- Bytes per pixel

128 MB: pure madness!
A better idea: pre-render all the possible 8-bit “scanlines” (= glyph rows)

\[2^8 \times 4 \times 8 \times 16 \times 16 = 2 \text{ MB}\]

Still expensive, but affordable!
It works on 32x16 fonts too!
The pre-render code

```c
# define PSZ 4    /* pixel size = 32 bpp / 8 = 4 bytes */
# define SL_COUNT 256 /* all possible 8-pixel scanlines */
# define SL_SIZE 8 /* scanline size: 8 pixels */
# define FG_COLORS 16 /* #fg colors */
# define BG_COLORS 16 /* #bg colors */
# define TOT_CHAR_SCANLINES_SIZE (PSZ*SL_COUNT*FG_COLORS*BG_COLORS*SL_SIZE)

bool fb_pre_render_char_scanlines(void)
{
    fb_w8_char_scanlines = kmalloc(TOT_CHAR_SCANLINES_SIZE);

    if (!fb_w8_char_scanlines)
        return false;

    for (u32 fg = 0; fg < FG_COLORS; fg++) {
        for (u32 bg = 0; bg < BG_COLORS; bg++) {
            for (u32 sl = 0; sl < SL_COUNT; sl++) {
                for (u32 pix = 0; pix < SL_SIZE; pix++) {
                    fb_w8_char_scanlines[
                        fg * (BG_COLORS * SL_COUNT * SL_SIZE) +
                        bg * (SL_COUNT * SL_SIZE) +
                        sl * SL_SIZE +
                        (SL_SIZE - pix - 1)
                    ] = (sl & (1 << pix)) ? vga_rgb_colors[fg] : vga_rgb_colors[bg];
                }
            }
        }
    }

    return true;
}
```
Intuition 2: copying 4 bytes at a time is too slow!

- Pre-rendering the glyphs or the just the “scanlines” is not enough
- The x86 `rep movsl` instruction copies just 4 bytes (= 1 pixel) at a time
Solution 2: use the FPU

- Introduce something like `fpu_memcpy()`
- Write a whole row at a time during scrolling
- Only this way, we could offset the cost of saving/restoring the FPU registers
void fb_draw_row(u32 y, u16 *entries, u32 count, bool fpu)
{
    static const void *ops[] = {
        &width_1_fpu, &width_1_nofpu, &width_2_fpu, &width_2_nofpu, &width_2_nofpu
    };
    const u32 bgp_shift = 4 + (font_bytes_per_glyph == 64) * 2; // 4 or 6
    const u32 w4_shift = 5 + (font_w == 16); // 5 or 6
    const void *const op = ops[(font_w == 16) * 2 + fpu]; // ops[0..3]
    const ulong vaddr_base = fb_vaddr + (fb_pitch + y);
    for (u32 ei = 0; ei < count; ei++) {
        const u16 e = entries[ei];
        const u32 c_off = (u32)((vgaentry_get_fg(e) << 15) + (vgaentry_get_bg(e) << 11));
        void *vaddr = (void *)vaddr_base + (ei << w4_shift);
        const u8 *d = &font_glyph_data[vgaentry_get_char(e) << bgp_shift];
        u32 *scanslines = &fb_u8_char_scanslines[c_off];
        goto *op;
    
    width_1_fpu:
        for (u32 r = 0; r < font_h; r++, d++, vaddr += fb_pitch)
            fpucpy_single_256_mt(vaddr, &scanslines[d[0] << 3]);
            continue;
    
    width_1_nofpu:
        for (u32 r = 0; r < font_h; r++, d++, vaddr += fb_pitch)
            memcpy32(vaddr, &scanslines[d[0] << 3], SL_SIZE);
            continue;
    
    width_2_fpu:
        for (u32 r = 0; r < font_h; r++, d++, vaddr += fb_pitch) {
            fpucpy_single_256_mt(vaddr, &scanslines[d[0] << 3]);
            fpucpy_single_256_mt(vaddr + 32, &scanslines[d[1] << 3]);
        } continue;
    
    width_2_nofpu:
        for (u32 r = 0; r < font_h; r++, d++, vaddr += fb_pitch) {
            memcpy32(vaddr, &scanslines[d[0] << 3], SL_SIZE);
            memcpy32(vaddr + 32, &scanslines[d[1] << 3], SL_SIZE);
        } continue;
}
The FPU code [1/2]

static void *get_fpu_cpy_single_256_nt_func(void)
{
    if (!kopt_no_fpu_memcpy) {
        if (x86_cpu_features.can_use_avx2)
            return &fpu_cpy_single_256_nt_avx2;
        if (x86_cpu_features.can_use_sse2)
            return &fpu_cpy_single_256_nt_sse2;
        if (x86_cpu_features.can_use_sse)
            return &fpu_cpy_single_256_nt_sse;
    }
    return NULL;
}

void init_fpu_memcpy(void)
{
    void *func;
    if ((func = get_fpu_cpy_single_256_nt_func())
        simple_hot_patch(&__asm_fpu_cpy_single_256_nt, func, 128);
}
The FPU code [2/2]

```c
30 ALWAYS_INLINE FASTCALL void
31   fpu_cpy_single_256_nt_avx2(void *dest, const void *src)
32 {
33   asmVolatile("vmovdqa (%0), %xmm0\n\t"
34     "vmovntdq %xmm0, (%1)\n\t"
35     : /* no output */
36     : "r" (src), "r" (dest)
37     : "memory";
38 }
39
40
41 ALWAYS_INLINE FASTCALL void
42   fpu_cpy_single_256_nt_sse2(void *dest, const void *src)
43 {
44   asmVolatile("movdqa (%0), %xmm0\n\t"
45     "movdqa 16(%0), %xmm1\n\t"
46     "movntdq %xmm0, (%1)\n\t"
47     : /* no output */
48     : "r" (src), "r" (dest)
49     : "memory";
50 }
51 }
52 ALWAYS_INLINE FASTCALL void
53   fpu_cpy_single_256_nt_sse(void *dest, const void *src)
54 {
55   asmVolatile("movq (%0), %xmm0\n\t"
56     "movq 8(%0), %xmm1\n\t"
57     "movq 16(%0), %xmm2\n\t"
58     "movq 24(%0), %xmm3\n\t"
59     : /* no output */
60     : "r" (src), "r" (dest)
61     : "memory";
62 }
63
64
65
66
67

Vladislav K. Valtchev (2022)
### The moment of truth

**Font 16x8, resolution 800x600, default memory type*, not WC**

* Typically that means UC (uncacheable) set through MTRRs

<table>
<thead>
<tr>
<th>Core i7-7500U Kaby Lake, AVX 2, 256-bit regs</th>
<th>Atom N270 Diamondville (32-bit), SSSE 3, 128-bit regs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Before (avg.)</strong></td>
<td><strong>Before (avg.)</strong></td>
</tr>
<tr>
<td>385.72 μs / char</td>
<td>186.27 μs / char</td>
</tr>
<tr>
<td><strong>After (avg.)</strong></td>
<td><strong>After (avg.)</strong></td>
</tr>
<tr>
<td>67.42 μs / char</td>
<td>94.82 μs / char</td>
</tr>
<tr>
<td><strong>Speed up</strong></td>
<td><strong>Speed up</strong></td>
</tr>
<tr>
<td>5.72x faster</td>
<td>1.96x faster</td>
</tr>
</tbody>
</table>

Not bad at all!

Smaller impact, but smaller regs here
# The moment of truth

**Font 16x8, resolution 800x600, default memory type*, not WC**

*Typically that means UC (uncacheable) set through MTRRs

<table>
<thead>
<tr>
<th></th>
<th>Core i7-7500U Kaby Lake, AVX 2, 256-bit regs</th>
<th>Atom N270 Diamondville (32-bit), SSSE 3, 128-bit regs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before (avg.)</td>
<td>385.72 μs / char</td>
<td>186.27 μs / char</td>
</tr>
<tr>
<td>After (avg.)</td>
<td>67.42 μs / char</td>
<td>94.82 μs / char</td>
</tr>
<tr>
<td>Speed up</td>
<td>5.72x faster</td>
<td>1.96x faster</td>
</tr>
</tbody>
</table>

**Font 32x16, resolution 3200x1800, default memory type*, not WC**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Before (avg.)</td>
<td>2543.21 μs / char</td>
</tr>
<tr>
<td>After (avg.)</td>
<td>371.54 μs / char</td>
</tr>
<tr>
<td>Speed up</td>
<td>6.84x faster</td>
</tr>
</tbody>
</table>

*Typically that means UC (uncacheable) set through MTRRs

- Wow, that’s close to the max 8x improvement! (From 32 bit/write to 256 bit/write)
- Still not fast enough, though

Vladislav K. Valtchev (2022)
The writing combining memory type (WC)

- Allows data to be combined, temporarily stored in a buffer (WCB) and then released in burst mode.
- Cannot be used most of the time because it offers just weak ordering.
- Can be set using PAT or MTRRs.
- It’s perfect for frame buffers.
## Performance: the full picture [modern machine]

*Font 16x8, resolution 800x600, 32 bbp*

Intel Core i7-7500U Kaby Lake (AVX 2, 256-bit fpu regs)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Opt</th>
<th>Wc</th>
<th>FPU</th>
<th>Cycles/char</th>
<th>usec/char</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failsafe slow</td>
<td></td>
<td></td>
<td></td>
<td>1,124,773</td>
<td>385.72</td>
</tr>
<tr>
<td>Failsafe opt</td>
<td></td>
<td></td>
<td></td>
<td>1,121,034</td>
<td>384.44</td>
</tr>
<tr>
<td>Opt + fpu</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>196,584</td>
<td>67.42</td>
</tr>
<tr>
<td>Opt (no fpu)</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>1,177,902</td>
<td>403.94</td>
</tr>
<tr>
<td>Wc</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>34,055</td>
<td>11.68</td>
</tr>
<tr>
<td>Opt + Wc (no fpu)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>44,294</td>
<td>15.19</td>
</tr>
<tr>
<td>Opt + Wc + fpu</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>30,271</td>
<td>10.38</td>
</tr>
</tbody>
</table>

32.9x faster!

Just 12.5% faster
## Performance: the full picture [older machine]

Font 16x8, resolution 800x600, 32 bbp

Intel Atom N270 Diamondville (32-bit, SSSE3, 128 bit fpu regs)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Opt</th>
<th>Wc</th>
<th>FPU</th>
<th>Cycles/char</th>
<th>usec/char</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failsafe slow</td>
<td></td>
<td></td>
<td></td>
<td>297,287</td>
<td>186.27</td>
</tr>
<tr>
<td>Failsafe opt</td>
<td></td>
<td></td>
<td></td>
<td>279,781</td>
<td>175.30</td>
</tr>
<tr>
<td>Opt + fpu</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>151,337</td>
<td>94.82</td>
</tr>
<tr>
<td>Opt (no fpu)</td>
<td>✓</td>
<td></td>
<td></td>
<td>279,453</td>
<td>175.10</td>
</tr>
<tr>
<td>Wc</td>
<td>✓</td>
<td></td>
<td></td>
<td>136,925</td>
<td>85.79</td>
</tr>
<tr>
<td>Opt + Wc (no fpu)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>136,914</td>
<td>85.79</td>
</tr>
<tr>
<td>Opt + Wc + fpu</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>136,906</td>
<td>85.78</td>
</tr>
</tbody>
</table>

2.04x faster

No difference at all!
Performance on native res [modern machine]
Font 32x16, resolution 3200x1800, 32 bbp

Intel Core i7-7500U Kaby Lake (AVX 2, 256-bit fpu regs)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Opt</th>
<th>WC</th>
<th>FPU</th>
<th>Cycles/char</th>
<th>usc/char</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failsafe slow</td>
<td></td>
<td></td>
<td></td>
<td>7,416,012</td>
<td>2543.21</td>
</tr>
<tr>
<td>Failsafe opt</td>
<td></td>
<td></td>
<td></td>
<td>7,408,637</td>
<td>2540.68</td>
</tr>
<tr>
<td>Opt + fpu</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>1,083,408</td>
<td>371.54</td>
</tr>
<tr>
<td>Opt (no fpu)</td>
<td>✓</td>
<td></td>
<td></td>
<td>7,815,696</td>
<td>2680.28</td>
</tr>
<tr>
<td>WC</td>
<td>✓</td>
<td></td>
<td></td>
<td>73,165</td>
<td>25.09</td>
</tr>
<tr>
<td>Opt + WC (no fpu)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>159,067</td>
<td>54.55</td>
</tr>
<tr>
<td>Opt + WC + fpu</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>27,841</td>
<td>9.55</td>
</tr>
</tbody>
</table>

6.84x faster
101.26x faster!
2.63x faster

Not bad!

Vladislav K. Valtchev (2022)
Performance vs Linux [modern machine]

Font 32x16, resolution 3200x1800, 32 bpp

Tilck
A Tiny Linux-Compatible Kernel

Commit a858f229, release build

Kernel 5.4.0 (Ubuntu 20.04.4 LTS)

Vladislav K. Valtchev (2022)
Performance vs Linux [modern machine]

Font 32x16, resolution 3200x1800, 32 bbp

Commit a858f229, release build

Kernel 5.4.0 (Ubuntu 20.04.4 LTS)

- 9.55 μs / char
Performance vs Linux [modern machine]

Font 32x16, resolution 3200x1800, 32 bpp

- 9.55 μs / char
- 56.40 μs / char

Commit a858f229, release build

Kernel 5.4.0 (Ubuntu 20.04.4 LTS)

Vladislav K. Valtchev (2022)
Performance vs Linux [modern machine]

Font 32x16, resolution 3200x1800, 32 bbp

Commit a858f229, release build

 Kernel 5.4.0 (Ubuntu 20.04 LTS)

- 56.40 μs / char

- 9.55 μs / char

5.9x faster!
Performance vs Linux [modern machine]

Font 32x16, resolution 3200x1800, 32 bbp

Commit a858f229, release build

Kernel 5.4.0 (Ubuntu 20.04 LTS)

56.40 µs - Linux 5.4.0

25.09 µs - Tilck failsafe + WC

9.55 µs - Tilck’s best OPT
The benchmark code

```c
1 void console_perf_test(void)
2 {
3     static const char letters[] =
4         "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
5         "abcdefghijklmnopqrstuvwxyz"
6
7     int iters = 3;
8     struct winsize w;
9     char *buf, tot_time_s[32], c_time_s[32];
10     size_t r, tot, written;
11     struct timespec ts_before, ts_after;
12     uint64_t start, end, c;
13     double tot_time_real, tot_time, time_c, cycles_per_sec;
14
15     if (ioctl(1, TIOCGWINSZ, &w) != 0) {
16         perror("ioctl() failed");
17         return;
18     }
19
20     tot = w.ws_row * w.ws_col;
21
22     if (!(buf = malloc(tot))) {
23         perror("malloc() failed\n");
24         return;
25     }
26
27     for (int i = 0; i < tot; i++) {
28         buf[i] = letters[i % sizeof(letters) - 1]);
29     }
30
31     printf("%s", CSI_ERASE_DISPLAY CSI_MOVE_CURSOR_TOP_LEFT);
32
33     retry:
34         clock_gettime(CLOCK_REALTIME, &ts_before);
35         start = RDTSRC();
36
37         for (int i = 0; i < iters; i++) {
38             for (r = 0, written = 0; written < tot; written += r) {
39                 if ((r = write(1, buf + written, tot - written)) < 0) {
40                     perror("write() failed");
41                         free(buf);
42                     return;
43                 }
44             }
45         }
46
47         end = RDTSRC();
48         clock_gettime(CLOCK_REALTIME, &ts_after);
49
50         c = (end - start) / iters;
51         tot_time_real = timespec_diff(&ts_after, &ts_before);
52         tot_time = tot_time_real / iters;
53         time_c = tot_time / (double)c;
54         cycles_per_sec = (end - start) / tot_time_real;
55
56         if (tot_time_real <= 0.1) {
57             iters *= 10;
58             goto retry;
59         }
60
61         timespec_to_human_str(tot_time_s, sizeof(tot_time_s), tot_time);
62         timespec_to_human_str(c_time_s, sizeof(c_time_s), time_c);
63         printf("Term size: %d rows x %d cols\n", w.ws_row, w.ws_col);
64         printf("Total iterations: %d\n", iters);
65         printf("Screen redraw: %llu cycles (%s)\n", c, tot_time_s);
66         printf("Avg. character cost: %llu cycles (%s)\n", c / tot, c_time_s);
67         printf("Cycles per sec: %12.0f cycles/sec\n", cycles_per_sec);
68         free(buf);
69     }
70 }
```
Making libmusl applications to work
Why libmusl?

- It made no sense to write a custom libc.
- Libmusl produces the smallest binaries (~13 KB for “hello world”)
- It’s actively maintained and widely used in the Embedded Linux world
- It’s supported by https://toolchains.bootlin.com/
- **Uclibc-ng** is more customizable but:
  - Typically produces larger binaries
  - Using a pre-built toolchain means no customization anyway
- **Dietlibc** is not well-maintained and has no pre-built toolchains

Vladislav K. Valtchev (2022)
Libmusl requires TLS support

- TLS requires `set_thread_area()`
Libmusl requires TLS support

- TLS requires set_thread_area()
- Can we cheat by returning -ENOSYS? 😊
int main();
weak void _init();
weak void _fini();
int __libc_start_main(int (*)(int, char **, void (*)(int, void (*)(void)), void(*)(void));

void _start_c(long *p)
{
    int argc = p[0];
    char **argv = (void *)(p+1);
    __libc_start_main(main, argc, argv, _init, _fini, 0);
typedef int lsm2_fn(int (*)(int,char **,char **), int, char **);
static lsm2_fn libc_start_main_stage2;

int __libc_start_main(int (**main)(int,char **,char **), int argc, char **argv)
{
    char **envp = argv+argc+1;

    /* External linkage, and explicit noinline attribute if available,
     * are used to prevent the stack frame used during init from
     * persisting for the entire process lifetime. */
    __init_libc(envp, argv[0]);
}

/* Barrier against hoisting application code or anything using ssp
 * or thread pointer prior to its initialization above. */
lsm2_fn *stage2 = libc_start_main_stage2;
__asm_ ( "" : "r"(stage2) : "memory" );
return stage2(main, argc, argv);
}

static int libc_start_main_stage2(int (**main)(int,char **,char **), int argc, char **argv)
{
    char **envp = argv+argc+1;

    /* Barrier against hoisting application code or anything using ssp
     * or thread pointer prior to its initialization above. */
lsm2_fn *stage2 = libc_start_main_stage2;
__asm_ ( "" : "r"(stage2) : "memory" );
return stage2(main, argc, argv);
}

(void) main

static int libc_start_main_stage2(int (**main)(int,char **,char **), int argc, char **argv)
{
    char **envp = argv+argc+1;

    /* Barrier against hoisting application code or anything using ssp
     * or thread pointer prior to its initialization above. */
lsm2_fn *stage2 = libc_start_main_stage2;
__asm_ ( "" : "r"(stage2) : "memory" );
return stage2(main, argc, argv);
}

(void) main

static int libc_start_main_stage2(int (**main)(int,char **,char **), int argc, char **argv)
{
    char **envp = argv+argc+1;

    /* Barrier against hoisting application code or anything using ssp
     * or thread pointer prior to its initialization above. */
lsm2_fn *stage2 = libc_start_main_stage2;
__asm_ ( "" : "r"(stage2) : "memory" );
return stage2(main, argc, argv);
}

(gdb) up
#2 0x8049270 in __init_libc (envp=0xbfffffff10, pn=<optimized out>) at src/env/__libc_start_main.c:39
#3 0x804937d in __libc_start_main (main=0x8049195 <main>, argc=1, argv=0xbfffffff08) at src/env/__libc_start_main.c:79
#4 0x804908f in _start_c (p=0xbfffffff04) at src/crt1.c:18
#5 0x804905b in _start ()
(gdb) down
#1 0x8049676 in static_init_tls (aux=0xbfffffff00) at src/env/__init_tls.c:149
#2 0x8049270 in __init_libc (envp=0xbfffffff10, pn=<optimized out>) at src/env/__libc_start_main.c:39
#3 0x804937d in __libc_start_main (main=0x8049195 <main>, argc=1, argv=0xbfffffff08) at src/env/__libc_start_main.c:79
#4 0x804908f in _start_c (p=0xbfffffff04) at src/crt1.c:18
(gdb) down
#3 0x804937d in __libc_start_main (main=0x8049195 <main>, argc=1, argv=0xbfffffff08) at src/env/__libc_start_main.c:79
for (i=0; aux[i]; i++) if (aux[i]<AUX_CNT) aux[aux[i]] = aux[i+1];
__hwcap = aux[AT_HWCAP];
if (aux[AT_SYSINFO]) __sysinfo = aux[AT_SYSINFO];
libc.page_size = aux[AT_PAGESZ];

if (!pn) pn = (void*)aux[AT_EXECFN];
if (!pn) pn = "");
__proname = __proname_full = pn;
for (i=0; pn[i]; i++) if (pn[i]=='/') __proname = pn+i+1;

__init_tls(aux);
__init_ssp(void *)aux[AT_RANDOM]);
if (aux[AT_UID]==aux[AT_EUID] && aux[AT_GID]==aux[AT_EGID]
    && !aux[AT_SECURE]) return;
struct pollfd pfd[3] = {{.fd=0}, {.fd=1}, {.fd=2}};
int r =
    #ifdef SYS_poll
    __syscall(SYS_poll, pfd, 3, 0);
#else
    __syscall(SYS_ppoll, pfd, 3, &(struct timespec){0}, 0, _NSIG/8);
}
0, libc.tls_size, PROT_READ|PROT_WRITE,
    MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);
/* -4095...-1 cast to void * will crash on dereference anyway,
    * so don't bloat the init code checking for error codes and
    * explicitly calling a_crash(). */
}

} else {
    mem = builtin_tls;
}

/* Failure to initialize thread pointer is always fatal. */
if (__init_tp(__copy_tls(mem)) < 0)
    a_crash();

weak_alias(static_init_tls, __init_tls);
```c
#include "pthread_impl.h"
#include "libc.h"
#include "atomic.h"
#include "syscall.h"

volatile int __thread_list_lock;

int __init_tp(void *p)
{
    pthread_t td = p;
    td->self = td;
    int r = __set_thread_area(TP_ADJ(p));
    if (r < 0) return -1;
    if (!r) libc.can_do_threads = 1;
    td->detach_state = DT_JOINABLE;
    td->tid = __syscall(SYS_set_tid_address, &__thread_list_lock);
    td->locale = &libc.global_locale;
    td->robust_list.head = &td->robust_list.head;
    td->sysinfo = __sysinfo;
    td->next = td->prev = td;
    return 0;
}
```
```c
#include "pthread_impl.h"
#include "libc.h"
#include "atomic.h"
#include "syscall.h"

volatile int __thread_list_lock;

int __init_tp(void *p)
{
    pthread_t td = p;
    td->self = td;
    int r = __set_thread_area(TP_ADJ(p));
    if (r < 0) return -1;
    if (!r) libc.can_do_threads = 1;
    td->detach_state = DT_JOINABLE;
    td->tid = __syscall(SYS_set_tid_address, &__thread_list_lock);
    td->locale = &libc.global.locale;
    td->robust_list.head = &td->robust_list.head;
    td->sysinfo = __sysinfo;
    td->next = td->prev = td;
    return 0;
}
```
```c
MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);

/* -4095...-1 cast to void * will crash on dereference anyway,
 * so don't bloat the init code checking for error codes and
 * explicitly calling a_crash(). */

} else {
    mem = builtin_tls;
}

/* Failure to initialize thread pointer is always fatal. */
if (__init_tp(__copy_tls(mem)) < 0)
    a_crash();

weak_alias(static_init_tls, __init_tls);
```
Sometimes cheating works
Sometimes cheating works

- Sometimes it doesn’t.
Sometimes cheating works

- Sometimes it *doesn’t*.
- Can we try returning 0 instead and see what happens?
struct user_desc {
    u32 entry_number;
    ulong base_addr;
    u32 limit;

    union {
        struct {
            u32 seg_32bit : 1;       /* Controls GDT_32BIT */
            u32 contents : 2;        /* Controls GDT_ACCESS_DC and GDT_ACCESS_EX */
            u32 read_exec_only : 1;  /* Controls GDT_ACCESS_RW */
            u32 limit_in_pages : 1;  /* Controls GDT_GRAN_4KB */
            u32 seg_not_present : 1; /* Controls GDT_ACCESS_PRESENT */
            u32 useable : 1;
            u32 ignored : 25;
        }
        u32 flags;
    }
};
```c
int __init_tp(void *p)
{
    pthread_t td = p;
    td->self = td;
    int r = __set_thread_area(TP_ADJ(p));
    if (r < 0) return -1;
    if (!r) libc.can_do_threads = 1;
    td->detach_state = DT_JOINABLE;
    td->tid = __syscall(SYS_set_tid_address, &__thread_list_lock);
    td->locale = &libc.global_locale;
}
```

In EDX we’re supposed to have now the entry number in the GDT. Clearly -1 is invalid.

So now we got an invalid selector now in EDX.
And, of course, here we get a GPF.
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
Load new symbol table from "/build/tlck_unstripped"? (y or n) y
Reading symbols from "/build/tlck_unstripped"
Error in re-setting breakpoint 2: No source file named /home/vlad/tlck/toolchain2/i386/musl/src/env/__libc_start_main.c.
Error in re-setting breakpoint 3: No source file named /home/vlad/tlck/toolchain2/i386/musl/src/env/__init_tls.c.
Error in re-setting breakpoint 4: No source file named /home/vlad/tlck/toolchain2/i386/musl/src/thread/i386/__set_thread_area.s.
(gdb)
What if we returned 0 and set a valid GDT entry number in \texttt{user\_desc}, without doing anything else?
Now EDX contains a valid GDT selector, 0x23, already used for userspace data.
We passed __init_tls(aux)!!

Remote Thread 1.1 In: __init_libc
(gdb) c
Continuing.

Breakpoint 5, 0x80494993 in __syscall1 (a1=134536688, n=258) at ./arch/i386/syscall_arch.h:25
(gdb) si
__init_tp (p=0x804c54c <builtin_tls+12>) at ./arch/i386/syscall_arch.h:26
0x80494967 in static_init_tls (aux=0xbfffffdd) at src/env/__init_tls.c:149
(gdb) si
(gdb) p $eax
$4 = 0
(gdb) si
__init_libc (envp=0xbfffff10, pn=optimized out) at src/env/__libc_start_main.c:40
(gdb)
We reached main()!!
Ehm... I don’t believe we’re going to pass that far indirect call...
Yep, page fault.
Vaddr is clearly just 0x10 because the GDT selector 0x23 has offset = 0 (flat segmentation)
Lesson learned

- Often, we cannot cheat.
Lesson learned

- Often, we cannot cheat.
- Even basic I/O functions use TLS variables.
Lesson learned

- Often, we cannot cheat.
- Even basic I/O functions use TLS variables.
- Had to provide a fully-functional implementation for `set_thread_area()`, in order run even single-threaded libmusl applications.
/* Handling the case where the user specified a GDT entry number */

int get_user_task_slot_for_gdt_entry(dc.entry_number);

if (!dc.entry_number || !gdt[dc.entry_number].access) {
    /* The entry is out-of-bounds or it's used by another task */
    rc = -EINVAL;
    goto out;
}

/* The entry is available, now find a slot */
slot = find_available_slot_in_user_task();

if (slot < 0) {
    /* Unable to find a free slot in this task */
    rc = -ESRCH;
    goto out;
}

gdt_set_slot(gdt�建构的, (u16)slot, (u16)dc.entry_number);

/* We're here because either we found a slot already containing this index
 * (therefore it must be valid) or the index is in-bounds and it is free.
 */

out: enable_preeemption();

if (rc)

/* Positive case: we get here with rc = SUCCESS, now flush back the
 * the struct user_desc (we might have changed its entry_number).
 */
rc = copy_to_user(ud, dc.entry_number, sizeof(struct user_desc));

if (rc < 0)
    rc = -EFAULT;

return rc;
That was quite some code, but it’s not enough. We need a **ref-count** for GDT entries as well.

Why? Think about **fork()**. What happens if the parent dies before the child and we free the GDT slots?

```c
void arch_specific_new_proc_setup(struct process *pi, struct process *parent)
{
    arch_proc_members_t *arch = get_proc_arch_fields(pi);
    if (!parent) /* we're done */
        return;
    memcp(y(&pi->pi_arch, &parent->pi_arch, sizeof(pi->pi_arch));
    if (arch->ldt)
        gdt_entry_inc_ref_count(arch->ldt_index_in_gdt);
    for (int i = 0; i < ARRAY_SIZE(arch->gdt_entries); i++) {
        if (arch->gdt_entries[i])
            gdt_entry_inc_ref_count(arch->gdt_entries[i]);
    pi->set_child_tid = NULL;
}
```
ACPICA & AcpiOsWaitSemaphore()

- ACPICA requires the OSL to provide a counting semaphore implementation capable of waiting and signaling N units.
- That is weird requirement.
- It could be trivially implemented on the top of a regular counting semaphore, but that would be extremely inefficient.
- I implemented such a semaphore in Tilck.
Classic semaphore

```c
void ksem_wait(struct ksem *s)
{
    struct task *curr = get_curr_task();
    disable_preemption();

    if (--s->counter < 0) {
        task_set_wait_obj(curr, WOBJ_SEM, s, NO_EXTRA, &s->wait_list);
        enable_preemption_nosched();
        kernel_yield();
        return;
    }

    enable_preemption();
}
```

Vladislav K. Valtchev (2022)

New semaphore [1/2]

```c
int ksem_wait(struct ksem *s, int units, int timeout_ticks)
{
    int rc = -ETIME;
    ASSERT(units > 0);

    if (s->max != KSEM_NO_MAX && units > s->max)
        return -EINVAL;

    disable_preemption();

    if (timeout_ticks != KSEM_NO_WAIT) {
        u64 start_ticks, end_ticks;

        if (timeout_ticks > 0) {
            start_ticks = get_ticks();
            end_ticks = start_ticks + (u32)timeout_ticks;

            if (s->counter < units)
                task_set_wakeup_timer(get_curr_task(), (u32)timeout_ticks);
        }

        while (s->counter < units) {
            if (timeout_ticks > 0 && get_ticks() >= end_ticks)
                break;

            prepare_to_wait_on(WOBJ_SEM, s, (u32)units, &s->wait_list);
            enter_sleep_wait_state(); /* after that, preemption will be enabled */
            disable_preemption();
        }

        if (timeout_ticks > 0)
            task_cancel_wakeup_timer(get_curr_task());

        if (s->counter >= units) {
            s->counter -= units;
            rc = 0;
        }
    }

    enable_preemption();
    return rc;
}
```
void ksem_signal(struct ksem *s)
{
    disable_preemption();
    if (s->counter++ < 0) {
        ASSERT(!list_is_empty(&s->wait_list));
        struct wait_obj *task_wo =
            list_first_obj(&s->wait_list, struct wait_obj, wait_list_node);
        struct task *ti = CONTAINER_OF(task_wo, struct task, wobj);
        task_reset_wait_obj(ti);
    }
    enable_preemption();
}

int ksem_signal(struct ksem *s, int units)
{
    struct wait_obj *wo, *tmp;
    int rem_counter, rc = 0;
    ASSERT(units > 0);
    disable_preemption();
    if (s->max != KSEM_NO_MAX) {
        if (units > s->max) {
            rc = EINVAL;
            goto out;
        }
        if (s->counter > s->max - units) {
            rc = ENOQUOT;
            goto out;
        }
    }
    s->counter += units;
    rem_counter = s->counter;
    list_for_each(wo, tmp, &s->wait_list, wait_list_node) {
        if (rem_counter <= 0)
            break; /* not enough units to unblock anybody */
        int wait_units = (int)wo->extra;
        if (wait_units <= rem_counter) {
            struct task *ti = CONTAINER_OF(wo, struct task, wobj);
            rem_counter -= wait_units;
            wake_up(ti);
        }
    }
out:
    enable_preemption();
    return rc;
}
But.. how Linux did implement the counting semaphore to make ACPICA happy?
But.. how Linux did implement the counting semaphore to make ACPICA happy?

It didn’t 😊
But.. how Linux did implement the counting semaphore to make ACPICA happy?

```c
/*
 * TODO: Support for units > 1?
 *
 */
acpi_status acpi_os_wait_semaphore(acpi_handle handle, u32 units, u16 timeout)
{
    acpi_status status = AE_OK;
    struct semaphore *sem = (struct semaphore *)handle;
    long jiffies;
    int ret = 0;

    if (!acpi_os_initialized)
        return AE_OK;

    if (!sem || (units < 1))
        return AE_BAD_PARAMETER;

    if (units > 1)
        return AE_SUPPORT;

    ACPI_DEBUG_PRINT((ACPI_DB_MUTEX, "Waiting for semaphore[%p|%d|%d]\n",
                        handle, units, timeout));
```
But.. how Linux did implement the counting semaphore to make ACPIA happy?

Sometimes cheating works.
Thank you!

https://github.com/vvaltchev/tilck