LEGACY REUSE

BJÖRN DÖBEL
So far ...

- Basic concepts, resources, ...
- Device Drivers

Today:

- Operating System Personalities
- How to reuse existing infrastructure?
- How to make applications happy?
Getting applications the easy way:

- Virtualize OS + app
- All services provided legacy OS (e.g., Linux)
- Original implementation, good app compatibility
- Limited isolation
MULTI-VM SETUP

Multiple VMs:
- Improved isolation
- Communication via virtual network
- Run same OS multiple times
- Run different OSes concurrently

VM 1: App, Linux
VM 2: App, Linux
VM 3: App, Win

RE + Virt
Microkernel
Hardware
Virtualization:
- Reuses legacy OS + applications
- Applications run in their natural environment

Problem: Applications trapped in VMs
- Different resource pools, namespaces
- Cooperation is cumbersome (network, ...)
- Full legacy OS in VM adds overhead
- Management overhead, multiple desktops?
MAKING THE CUT

■ **Hardware level** (virtualization)
  ■ Legacy OS + applications on top of new OS

■ **Operating system level** (e.g., Wine)
  ■ Legacy OS’s interfaces reimplemented on top of new OS

■ **Application level**
  ■ Applications, Toolkits, frameworks, libraries ported to new OS
OPERATING SYSTEM PERSONALITIES
**OS PERSONALITY**

**Idea:** Adapt at OS / application boundary
- (Re-)Implement legacy APIs, not whole OS
- May need to recompile application

**Benefits:**
- Get desired application, established APIs
- More flexible, configurable, easier to achieve
- Better integration (namespaces, files, ...)
- Improved reliability, less overhead
Central server provides consistent view for both:

- **Servers**: client state (e.g., file tables)
- **Applications**: system resources (e.g., files)

Potential issues:

- Scalability
- High complexity
- Single point of failure
- Emulation library:
  - Linked into applications
  - Interacts with servers
  - Provides consistent view

- Each server keeps its own client state

**In real world:** emulation library hidden below libc
What to do:
- Determine what application needs
- Provide the needed APIs

Way to go:
- Do not reinvent the wheel!
- Reuse libraries, map to existing APIs
- Better compatibility, more complete
- Make everything modular
■ „Portable Operating System Interface“ is a family of standards (POSIX 1003.*)

■ POSIX makes UNIX variants source-code compatible (also introduced in Windows NT)

■ Defines interfaces and properties:
  ■ I/O: files, sockets, terminal, ...
  ■ Threads, synchronization: pthreads
  ■ System tools

■ Accessible through C library
WHAT IS THE LIBC?

- C library abstracts underlying OS
- Collection of common functionality
- Abstraction level varies:
  - low level: `memcpy()`, `strlen()`
  - medium level: `fopen()`, `fread()`
  - high level: `getpwent()`
- ... and so do dependencies:
  - none (freestanding): `memcpy()`, `strlen()`
  - small: `malloc()` depends on `mmap()`
  - strong: `getpwent()` needs file access, name service, ...
■ libC support on L4Re: µClibc
  ■ Compatible to GNU C library „glibc“
  ■ Works well with libstdc++
  ■ Small and portable
  ■ Designed for embedded Linux

■ Fiasco.OC + L4Re != Linux

■ How to port a low-level library?
MULTI-SERVER LIBC

Application
libc + System Call Bindings

memcpy()
fopen()

open(), read(), mmap()

L4Re::Env::mem_alloc()
L4::L4fs::open()

Monolithic Kernel
Ext2, VFAT

System Call Entry
VFS / MM

uClibc

VFS BE
mem BE
Rofs BE
L4fs BE
time BE

Application

L4Re::Env::mem_alloc()
L4::L4fs::open()

Microkernel

VFS / MM
Mem IF
Rofs IF
L4fs IF
MOE
VPFS

Legacy Reuse
LIBC BE: TIME

Example 1: POSIX time API

```
time_t time(time_t *t)
{
    struct timespec a;
    libc_be_rt_clock_gettime(&a);
    if (t)
        *t = a.tv_sec;
    return a.tv_sec;
}
```

Replacement of POSIX function `time()`

```
uint64_t __libc_l4_rt_clock_offset;

int libc_be_rt_clock_gettime(struct timespec *tp)
{
    uint64_t clock = l4re_kip()->clock;
    clock += __libc_l4_rt_clock_offset;

    tp->tv_sec  = clock / 1000000;
    tp->tv_nsec = (clock % 1000000) * 1000;

    return 0;
}
```

L4Re-specific backend function
Example 2: Memory Management

- μClibc implements heap allocator
- requests memory pages via `mmap()`
- can be reused, if we provide `mmap()`
  - self_mem:
    - Minimalist, uses static pages from BSS
  - l4re_file:
    - Supports `mmap()`, `munmap()` for anon memory
    - Based on dataspaces and L4Re region manager
- `malloc()` calls `mmap()` with flags `MAP_PRIVATE|MAP_ANONYMOUS`
- Pages taken from large dataspace
- Attached via L4RM interface
- Reference counter tracks mapped regions

- `munmap()` detaches dataspace regions
  - `if (region_split) refs++; else refs--;`
  - Dataspace released on zero references
Example 3: POSIX signals

- Used for asynchronous event notification:
  - Timers: `setitimer()`
  - Exceptions: `SIGFPE, SIGSEGV, SIGCHLD, ...`
  - Issued by applications: `SIGUSR1, SIGUSR2`

Signals on Linux:
- Built-in kernel mechanism
- Delivered upon return from kernel

How to implement signals in L4Re?
Dedicated thread $E$ handles exceptions and timers

$E$ is exception handler of thread $T$

Exceptions in $T$ are reflected to $E$

If app configured signal handler:

$E$ sets up signal handler context

$E$ resets $T$’s program counter to start of signal handler

$T$ executes signal handler, returns

If possible, $E$ restarts $T$ where it had been interrupted
E: handles exceptions:

- Set up signal handler context:
  - Save T’s context
  - Push pointer to siginfo_t, signal number
  - Push address of return trap
- l4_utcb_exc_pc_set(ctx, handler)

T: execute signal handler, „returns“ to trap

E: resume thread after signal:

- Exception generated, reflected to E
- Detects return by looking at T’s exception PC
- Restore T’s context saved on stack, resume

Stack Frames (pre-exception)

| ucontext_t ctx |
| siginfo_t *siginfo |
| int signum |
| Return address |

void libc_be_sig_return_trap()
{
  /* trap, cause exception */
}
Basic mechanism for Fiasco.OC + L4Re:
- Exceptions are mapped to IPC messages
- E waits for exception IPCs in server loop

Timers implemented as IPC timeouts:
- \texttt{sigaction()} / \texttt{setitimer()} called by T
- T communicates time to wait to E
- E waits for IPC timeout
- E raises exception in T to deliver \texttt{SIGALRM}
What is needed for file I/O?

- `fprintf()` support: easy, just replace `write()`
- Minimalist backend can output text

```c
#include <unistd.h>
#include <errno.h>
#include <l4/sys/kdebug.h>

int write(int fd, const void *buf, size_t count) __THROW
{
    /* just accept write to stdout and stderr */
    if (((fd == STDOUT_FILENO) || (fd == STDERR_FILENO))
    {
        l4kdb_outnstring((const char*)buf, count);
        return count;
    }
    /* writes to other fds shall fail fast */
    errno = EBADF;
    return -1;
}
```
(1) Application calls `open(„rom/hello‟)"

(2) VFS traverses mount tree, finds Ro_fs mounted at „rom‟

(3) VFS asks Ro_fs to provide a file for name „hello‟, Ro_fs calls its `get_entry()` method

(4) `Ro_fs::get_entry()` creates new Ro_file object from read-only dataspace provided by MOE

(5) VFS registers file handle for Ro_file object

(6) Application calls `read()`: ends in `Ro_file::readv()`

(7) `Ro_file::readv()` attaches dataspace, copies requested data into read buffer
L4Re offers most important POSIX APIs

- C library: `strcpy()`, ...
- Dynamic memory allocation:
  - `malloc()`, `free()`, `mmap()`, ...
  - Based on L4Re dataspaces
- Threads, synchronization: `pthreads`
- Signal handling
- I/O support: files, terminal, time, (sockets)

POSIX is enabler: sqlite, Cairo, SDL, shell, ...
APPLICATION-LEVEL VIRTUALIZATION
■ POSIX is limited to basic OS abstractions
  ■ No graphics, GUI support
  ■ No audio support

■ Examples for more powerful APIs:
  ■ SDL “Simple Direct Media Layer“:
    ■ Multimedia applications and games
  ■ Qt toolkit:
    ■ Rich GUIs with tool support
    ■ Fairly complete OS abstractions
- **Qt** is a multi-platform toolkit library
- Compile & run same application on multiple platforms:
  - UNIX/X11, Embedded Linux
  - Windows, Mac OS X
  - Also ported to *L4Env, L4Re, Genode*
- L4 port based on Qt3 for Embedded Linux:
  - Brings its own windowing system
  - Relies on POSIX-like OS
QT/L4 PORT

Application

Qt API

File Access  Network  Threads / Sync  GUI  ...  
Platform independent

POSIX / L4 VFS  POSIX / L4 VFS+FLIPS  pthread or L4 Threads+ Semaphores  QWS  ...  
Platform specific

Operating System (POSIX compliant)
Thanks to POSIX APIs available on L4:

- Complete Qt GUI framework
- Threads, synchronization
- File access (L4 VFS backends / servers)
- Limited network access
- Tool support: GUI builder
Tools: Qt Designer, UI compiler (uic), moc

Test Applications (or parts of them) on Linux
LEGACY OPERATING SYSTEM AS A TOOLBOX
Applications are nice, but there’s more ...

Legacy OSes have lots of:
- Device drivers
- Protocol stacks
- File systems

Reuse drivers in natural environment
- Also see paper: "Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines", by LeVasseur, Uhlig, Stoess, Götz)

L4Linux:
- Hybrid applications: access legacy OS + L4Re
- In-kernel support: bridge Linux services to L4Re
GENERAL IDEA

L4Linux Kernel

Input Event IF

“Proxy” Driver

Interrupt

Mag

Application
- L4Linux has drivers
- L4Re has great infrastructure for servers:
  - IPC framework
  - Generic server loop
- **Problem:** C vs. C++, calling conventions
- **Bridge:** allow calls from Linux to L4Re
  - L4Re underneath L4Linux export C functions
  - L4Linux kernel module calls them
INPUT DRIVER

L4Linux Kernel

Input Event IF

Proxy Input Drv

Server Loop

L4Linux Container (ELF Loader)

C++

L4Re Kernel

C++

Application

Mag

Interrupt

Register Client, IRQ

C

Application
**Idea:** „enlightened“ applications
- Know that they run on L4Re
- Talk to L4Re servers via guest OS

**Proxy driver** in guest provides:
- Shared memory: Linux app + L4Re server
- Signaling: Interrupt objects
- Enables synchronous and asynchronous zero-copy communication (e.g., ring buffer)
Shared memory + Signaling:
- Trigger Linux IRQ, then unblock `read()` on chardev
- Call `write()` on chardev, then trigger L4 app's IRQ
Proxy driver suitable for different scenarios:

- Producer/consumer (either direction)

Split applications:
- Reuse application on either side
- Trusted / untrusted parts

Split services:
- Block device / file system / database / …
- Network stack
Next week:

- Lecture: „Virtualization“
