Legacy Containers and OS Personalities

Carsten Weinhold

Dresden, 2007-12-18
So far:

- Basics Concepts:
  - Tasks, Threads, IPC
  - Memory, Synchronization, ...
- Device Drivers
- Real-Time
- Naming
- Virtualization

Today:
- Legacy Containers / OS Personalities
Motivation

What are legacy containers / OS personalities?

• Emulation environment
• Allow to run legacy applications

How is it different from virtualization?

• Adaptation can be above hardware / OS layer
• Often requires recompilation of the application

Why?

• Reuse existing applications on new OS
• Reuse existing APIs for new applications
• Flexibility / Configurability
• Lower resource consumption
Adaptation to underlying system can happen at various levels:

• **Hardware level (virtualization):**
  - Legacy OS and applications are moved to new OS (full + para virtualization, e.g., L^4Linux)

• **OS level:**
  - Legacy OS's interfaces are reimplemented on top of new OS (e.g., Wine on Linux)

• **Application level:**
  - Toolkits, framework (e.g., Qt)
Single-server approach:

- All (legacy) system services provided by a single server (e.g., $L^4$Linux)
- No isolation among services
- Provides high level of compatibility, as original implementations are reused
- Ideally, applications do not need to be modified
Hybrid applications (real time):

- “Enlightened” applications:
  - Major parts of applications (e.g., GUIs) do not require real-time guarantees
  - Small parts may require real-time services running next to legacy container

- Examples:
  - RT network, RT file system, Window with RT widget
Hybrid applications (security):

- “Enlightened” applications:
  - Most parts are not security critical
  - Small parts require strong isolation from potentially insecure legacy container

- Examples:
  - Secure file system that encrypts all data

More in following lectures!
Multiple Single Servers

- Multiple instances of single-server OSes
- Useful for isolation of applications
- Not necessarily identical OS personalities (e.g., have UNIX and Windows servers)
- “Virtual machines”
Move from single server to multi server

Challenge: Application wants a consistent view in a distributed system
**Approach: Central Server**

- Central server provides consistent view for both:
  - Application
    - System resources (files, ...)
  - Servers
    - Client state (file tables, ...)

- Problem:
  - Performance bottleneck
  - Scalability
  - Complexity
Approach: Emulation Library

- Emulation library
  - Per application
  - Interacts with servers
  - Provides consistent view

- Each servers keeps its own client state

- In real world:
  - Applications use C library, emulation library hidden below
  - Emulation provides **POSIX API**
Legacy Environment: POSIX

- POSIX “Portable Operating System Interface” is a family of standards (POSIX 1003.*)
- POSIX makes various UNIX-like systems compatible (even Windows NT and newer)
- Defines interfaces and properties:
  - I/O: files, sockets, terminal, ...
  - Threads / synchronization: PThread
  - System tools
  - ...
- POSIX API accessible via C library
C Library: API to POSIX

- Collection of various common functions
- Abstraction layer above system calls
- Functions with different abstraction levels ...
  - low level: `memcpy()`, `strlen()`
  - medium: `fopen()`, `fread()`
  - high level: `getpwent()`

- ... and dependencies
  - none (freestanding): `memcpy()`, `strlen()`
  - small: `malloc()`: uses `mmap()` or `sbrk()`
  - strong: `getpwent()`: file access, “/etc/passwd”, name service, ...
Multi-Server Aware C Library

- Observations:
  - C library depends on underlying OS
  - Specific implementation may differ

![Diagram of Multi-Server Aware C Library](image)

- Observations:
  - C library depends on underlying OS
  - Specific implementation may differ

![Diagram of Multi-Server Aware C Library](image)
• What is uClibc?
  – Reimplementation of C library
  – Designed for Embedded Linux
  – Compatible to GNU C library “glibc”

• Why uClibc?
  – Small and portable
  – Optional C++ support available (uClibc++)
  – Licensed under LGPL

• uClibc + backends provide applications with a consistent view of L4-based multi-server OS
C Library Backends

- C library functions grouped in backends
- Simple example: `time`

```c
unsigned int offset;
void l4libc_init_time(void)
{
    int ret;

    l4_calibrate_tsc();
    ret = l4rtc_get_offset_to_realtime(&offset);
    if (ret != 0)
    {
        /* RTC server not found, */
        /* assuming 1/1/1970, 0:00 */
        offset = 0;
    }
}

extern unsigned int offset;
time_t time(time_t *t)
{
    time_t t_temp;
    l4_uint32_t s, ns;

    l4_tsc_to_s_and_ns(l4_rdtsc(), &s, &ns);
    /* get (cached) system time offset */
    t_temp = s + offset;
    if (t)
    {
        *t = t_temp;
        return t_temp;
    }
}
```

1. Initialization
   Call time server

2. Usage:
   Locally calculate time (no call to time server necessary)
Multiple implementations for some backends:

- Memory backends:
  - self_mem
  - sigma0_mem, sigma0_pool_mem
  - simple_mem
  - buddy_slab_mem,
  - TLSF (real-time memory allocator)

- I/O backends:
  - minimal_log_io, io
Example: Minimalist I/O Backend

- Minimalist I/O backend:
  - Simple backend for writing to STDOUT/STDERR
  - Uses L4/Fiasco kernel debugger

```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;
}
```
Example: Complex I/O Backend

- Complex L4VFS I/O backend:
  1) App **test1**'s libc calls **open()**
  2) Resolve name at name server
     a) Backend calls name server
     b) Name server calls server **con_term**, gets back reference to **tty1**
  3) Backend opens **tty1**, server **con_term** updates client state
  4) Backend updates local file table
  5) Backend returns file handle
Example: POSIX signals on L4

• Signals used to deliver *asynchronous* event notifications
  – CPU exceptions (SIGFPE, …)
  – Kernel exceptions (SIGSEGV, SIGCHLD, …)
  – Issued by applications (SIGUSR1, SIGUSR2, SIGTERM, …)

• Signals on Linux
  – built-in kernel mechanism
  – delivered upon return from kernel

• Signals on L4
  – implement in user space
  – map signals to L4 IPC?
  – Problem: IPC is synchronous!
POSIX signals on L4: startup

- Application A
- Application B

**libl4sig**
- Signal thread
- setup signal handlers

**signal server**
- A -> A.3
- B -> B.7

**register_handler()**

**libl4sig**
- Signal thread
- setup signal handlers

**register_handler()**
POSIX signals on L4: sending a signal

Application A

signal thread
libl4sig

notify(SIGUSR1)

signal server

A -> A.3
B -> B.7

signal(A, SIGUSR1)

Application B

signal thread
libl4sig
POSIX signals on L4: dispatching

- Signal handler thread dispatches signal:
  - chose arbitrary application thread $\textbf{Ex}$ (except for L4Env threads)
  - store $\textbf{Ex}$'s thread state (EIP, ESP)
  - force $\textbf{Ex}$ to run configured signal handler using a dedicated stack
  - after $\textbf{Ex}$ returns from handler: restore original EIP and ESP

- Specialties:
  - SIGSTOP: set all threads to $\text{l4sleep()}$
  - SIGALRM: dedicated timer provided by the signal server
  - SIGKILL: signal server kills application at task server
Unnecessary backends can be omitted:
- Reduced TCB for applications
- Potentially better security properties
- Smaller memory footprint
Summary: POSIX Emulation on L4Env

• Provides a significant part of POSIX APIs:
  – C library: all platform independent parts
    • `strcpy()`, ...
  – Dynamic memory allocation
    • Various backends
  – I/O support (files, network, terminal, time):
    • L4VFS + L4VFS servers
    • RTC server
  – Signal handling:
    • `signal`
  – No PThread support
More Powerful Legacy Containers

- POSIX is limited to basic OS abstractions:
  - No graphics / GUI support
  - No audio support

- Examples for more powerful APIs:
  - SDL “Simple Direct Layer”
    (focuses on Multimedia)
  - Qt toolkit
    (focuses on rich GUIs and complete OS abstraction)
• What is Qt?
  – Multi-platform toolkit library
  – Provides abstraction of underlying OS
  – Available for:
    • UNIX/X11 (Linux, *BSD, ...)
    • Windows
    • Mac OS X
    • ... and L4 (Bastel port in progress)

• L4 port based on Qt's Linux/embedded version
  – Has its own “Qt Windowing System” (QWS)
  – Relies on POSIX compliant OS
Qt Architecture

Platform Independent

Application

Qt API

File Access  Network  Threads / Synch  GUI  ...

File Access: POSIX / L4VFS  POSIX / L4VFS  L4Env / thread, semaphore  QWS

Operating System

Platform Specific
QWS is core part of Qt/L4:
- Displays and manages windows from multiple Qt applications
- Operates on shared framebuffer
- Events distributed via shared memory / sockets
Qt/L4: Features

- Complete support of GUI framework
- Threads, Synchronization supported
- Support for file access
  - Depends on available L4VFS file servers
- Support for network access
  - Basic functionality with L4VFS server “FLIPS”
• Applications (or parts thereof) can be developed and tested on Linux
• Rapid Application Development:
  – GUIs can be created with Qt Designer
  – Other tools (e.g., UI compiler) can be reused
DEMO
References

• Carsten Weinhold: 'Portierung von Qt auf DROPS'
  Großer Beleg 2005

• Resources on POSIX standard:
  http://standards.ieee.org/regauth/posix/

• Christian Helmuth, Frank Mehnert, Lars Reuther, Martin Pohlack, Alexander Warg, Björn Döbel:
  MOS slides from previous years
Coming soon...

- **This week:**
  - Prepare for Christmas :-)

- **Next weeks:**
  - Security (08.01. + 15.01.)
  - Practical exercise “Bastei” (09.01.)