Legacy Containers and OS Personalities

Carsten Weinhold

Dresden, 2009-12-08
So far:

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

Today:

- Legacy Containers / OS Personalities
Adaptation to underlying system can happen at various levels:

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

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

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

- All (legacy) system services provided by a single server (e.g., L4Linux)
- No isolation among services
- Provides high level of compatibility, as original implementations are reused
- Ideally, applications do not need to be modified
• 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”
Hybrid applications (real time):

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

- Examples:
  - RT network, RT file system, RT window system / widget
Hybrid applications (security):

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

• Examples:
  – Cryptographic algorithms and key handling
Virtualized OSes: Recap

• Virtualization allows reuse of OS
• Multiple VMs
• Legacy applications run on top
• Applications are kind of trapped in VM:
  – Different resource pools
  – Different namespaces
  – Cooperation with apps running next to VM possible, though

• “... but I just want that application running”
A Case for OS Personalities

How is it different from virtualization?
- Adaptation above hardware / OS layer
- (Re-)implemented APIs instead of whole OS
- May require recompilation of the applications

Why?
- Reuse existing applications / APIs
- Sometimes easier to get it running
- Flexibility / Configurability
- Better integration (namespaces, files, ...)
- Improved reliability
- Lower resource consumption
Move from single server to multi server

Challenge: Application wants a consistent view in a distributed system
• Central server provides consistent view for both:
  – Applications
    • 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 legacy 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 on L4Re: uClibc

• L4Re uses uClibc:
  – Implementation of C library
  – Designed for Embedded Linux
  – Compatible to GNU C library “glibc”
  – Small and portable
  – Optional C++ support available (uClibc++)
  – Licensed under LGPL

• Fiasco.OC + L4Re ≠ Linux

• How to port it?
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
**Example 1:** POSIX time API

- Functions can easily be replaced on L4Re

```c
uint64_t __libc_l4_rt_clock_offset;

int libc_be_rt_clock_gettime(struct timespec *tp) {
    uint64_t clock;
    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 (called by `time()` and other POSIX functions)

Replacement of `time()` function

```c
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;
}
```
Example 2: memory management

- uClibc implements heap allocator
- Requests memory pages via mmap()
- Can be reused, just need to provide mmap()
- Multiple implementations possible, e.g.:
  - self_mem
    - Very simple, uses pages from BSS
    - Set up by allocating static array
  - l4re_mem
    - Full mmap(), munmap(), ...
    - Requests memory as dataspaces
Example 3: POSIX signals

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

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

- Signals on L4Re:
  - Implemented in user space
  - Based on exceptions and IPC timeouts
POSIX Signals on L4Re: Mechanics

- 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 signal handler
  - $T$ executes signal handler, returns
- If possible, $E$ restarts $T$ where it had been interrupted
• Basic mechanism:
  – Exceptions are mapped to IPC messages
  – E waits for exception IPCs in server loop

• Timers implemented as IPC timeouts:
  – `sigaction()`/`setitimer()`/... called in T
  – T communicates time to wait to E
  – E waits for IPC for timeout
  – E Raises exception in T to deliver SIGALRM
POSIX Signals on L4Env: kill()

Diagram:

- Application A
  - register_handler()
  - Signal backend
    - Signal Thread (a.03)
    - signal server
    - signal_threads
      - A a.03
      - B b.04
    - notify(SIGUSR1)
  - setup signal handlers

- Application B
  - register_handler()
  - Signal backend
    - Signal Thread (b.04)
    - setup signal handlers
What's 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;
}
```
Example 4: POSIX file API

- File system access provided by servers
- Multiple servers with their own namespaces
- Problem: consistent view at namespace
L4VFS: Distributed Name Service

- **Name server**
  - Maps symbolic names -> object ids: \((volume \ id) . (local \ object \ id)\)
  - Iteratively queries object servers to resolve names
  - Maps volume ids to object servers
  - Manages mount table

- **Object server**
  - Maintains locally unique object IDs (e.g., for files)
  - Maps names -> local object IDs
  - Keeps client state file pointer, access mode, ...
• Complex I/O backend: L4VFS
• Example:
  1) App calls `open("/dev/tty1")`
  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) Libc returns local file handle
Unnecessary backends can be omitted:
- Applications rely on less components
- Potentially better security properties
- Most appropriate backend can be chosen
Summary: POSIX Emulation on L4Re

- L4Re offers significant part of POSIX APIs:
  - C library: strcpy(), ...
  - Dynamic memory allocation
    - malloc(), free(), mmap()
    - Based on dataspaces
  - Pthread
  - Signal handling
  - I/O support: files, terminal, time, (network)

- POSIX is enabler: sqlite, Cairo, shell, …
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
    • Also ports for: L4Env, L4Re, Genode

• Qt/L4Env based on Qt3 for embedded Linux:
  – Has its own “Qt Windowing System” (QWS)
  – Relies on POSIX compliant OS
Qt Architecture

Application

Platform Independent

Qt API

File Access
Network
Threads / Sync
GUI
...

Platform Specific

POSIX / L4VFS
POSIX / L4VFS
L4Env thread, semaphore
QWS

Operating System
• **QWS is core part of Qt/L4Env:**
  - Displays and manages windows from multiple Qt applications
  - Operates on shared framebuffer
  - Events distributed via shared memory / sockets
DEMO
Qt/L4Env: 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 of them) can be developed and tested on Linux
• Rapid Application Development:
  – GUIs can be created with Qt Designer
  – Other tools (e.g., UI compiler) can also be usable
Coming up next ...

• **Later today:**
  • Compley lab

• **Next week:**
  • Lecture: “Security: Intro”
  • Practical excercise: “Capability Systems”
Example: Minix 3

Figure from [3]: “Reorganizing UNIX for Reliability”, J. N. Herder, H. Bos, B. Gras, P. Homburg, A. S. Tanenbaum
References

