User Mode Linux

Linux inside Linux inside Linux inside...

Muli Ben-Yehuda

mulix@mulix.org

IBM Haifa Research Labs

what is UML?

- User Mode Linux (UML, hereafter) is a port of Linux (the kernel) to run as a program inside Linux (the system), creating a free software production quality Linux virtual machine. Instead of working directly with the hardware, UML uses the host's system call interface in place of the hardware. Surprisingly enough, it actually works.
- UML was developed primarily by Jeff Dike <jdike@karaya.com>. Many people (including yours truly) contributed patches and bug fixes.
- UML is part of the official Linux kernel distribution as of kernel 2.5, and there are patches available for 2.2 and 2.4.

TOC

- Introduction to UML (what is it good for?)
  - UML capabilities
  - UML limitations
- Overview of UML architecture
- Interlude - the ptrace API
- Tracing Thread (TT) mode
- Seperate Kernel Addressspace (SKAS) mode

what is it good for?

- testing and debugging kernel patches, without requiring a reboot and using a mature debugger (gdb)
- private servers on shared hosts, virtual machine hosting - people are selling hosting based on UML
- experimenting with system administration scenarios and new Linux distributions / services
- teaching operating systems ;-)
- UML clusters...
- (when UML-win32 comes of age) running Linux on windows machine - tapping into unused resources at night

building and running it

- make menuconfig ARCH=um
- make linux
- ./linux [gazillion optional options here]

simple, isn't it? Don't forget you also need the UML utilities and a root file systems. You can get everything (with instructions) from http://user-mode-linux.sf.net

UML capabilities

- run user space code unmodified (like vmware and as opposed to e.g. the Xen virtual machine)
- secure virtual machine in software only
- reasonable performance, dependant on the workload but generally speaking no worse than half than the host's performance
- access to host file systems via hostfs
- full networking support
- SMP, highmem support
- can run Linux on any other OS
- potential access to kernel primitives as a user space library

UML limitations

- performance...
- access to hardware requires special UML drivers
- cannot be used to debug architecture dependent kernel code
- only runs on Linux currently, and only on i386 (but porting to other OS's / architectures a Simple Matter of Programming)

UML design

UML is a port of the Linux kernel to a new “virtual” architecture - Linux’s system call interface. The Linux kernel is divided into an architecture independent part, which is the bulk of the code, and into an architecture dependent part (include/asm-* and arch/*). UML is a port of Linux to the “UML architecture”. Under arch/um, it implements the architecture specific interface the rest of the Linux kernel expects.

Example: setting up physical memory

When UML starts up, libc's startup code calls main(), which then calls linux_main().

linux_main() calls setup_physmem() to setup UML's imitation of physical memory:

```
void setup_physmem(unsigned long start, unsigned long reserve_end,
                   unsigned long len)
{
        unsigned long reserve = reserve_end - start;
        int pfn = PFN_UP(__pa(reserve_end));
        int delta = (len - reserve) >> PAGE_SHIFT;
        int err, offset, bootmap_size;

        physmem_fd = create_mem_file(len);

        offset = uml_reserved - uml_physmem;
        err = os_map_memory((void *) uml_reserved, physmem_fd, offset,
                            len - offset, 1, 1, 0);
        [snip]
}
```

create_mem_file

```
int create_mem_file(unsigned long len)
{
        int fd, err;
        char zero;

        fd = make_tempfile(TEMPNAME_TEMPLATE, NULL, 1);
        [snip error checking]
        err = os_mode_fd(fd, 0777);
        [snip various checks]
        err = os_set_exec_close(fd, 1);
        if(err < 0)
                os_print_error(err, "exec_close");
        return(fd);
}
```

os_close_on_exec - Linux implementation

```
int os_set_exec_close(int fd, int close_on_exec)
{
        int flag, err;

        if(close_on_exec) flag = FD_CLOEXEC;
        else flag = 0;

        do {
                err = fcntl(fd, F_SETFD, flag);
        } while((err < 0) && (errno == EINTR)) ;

        if(err < 0)
                return(-errno);
        return(err);
}
```

os_map_memory - Linux implementation

```
int os_map_memory(void *virt, int fd, unsigned long off, unsigned long le
                  int r, int w, int x)
{
        void *loc;
        int prot;

        prot = (r ? PROT_READ : 0) | (w ? PROT_WRITE : 0) |
                (x ? PROT_EXEC : 0);

        loc = mmap((void *) virt, len, prot, MAP_SHARED | MAP_FIXED,
                   fd, off);
        if(loc == MAP_FAILED)
                return(-errno);
        return(0);
}
```

interlude - the ptrace API

ptrace is a low level, architecture dependant POSIX API for one process to intimately manipulate another. It is the mechanism used by debuggers, for example.

show strace /bin/echo as an example of syscall tracing

show dumpmem /bin/echo as an example of memory dumping

interlude - the ptrace API interface

```
#include <sys/ptrace.h>

long ptrace(enum __ptrace_request request, pid_t pid, void
            *addr, void *data);

enum __ptrace_request
{
  /* Indicate that the process making this request should be traced.
     All signals received by this process can be intercepted by its
     parent, and its parent can use the other ‘ptrace’ requests.  */
  PTRACE_TRACEME = 0,
  /* Return the word in the process’s text space at address ADDR.  */
  PTRACE_PEEKTEXT = 1,
  /* Return the word in the process’s data space at address ADDR.  */
  PTRACE_PEEKDATA = 2,
```

interlude - the ptrace API interface - cont

```
 /* Return the word in the process's user area at offset ADDR.  */
PTRACE_PEEKUSER = 3,
/* Write DATA into the process's text space at address ADDR.*/
PTRACE_POKETEXT = 4,
/* Write DATA into the process's data space at address ADDR.*/
PTRACE_POKEDATA = 5,
/* Write DATA into the process's user area at offset ADDR.  */
PTRACE_POKEUSER = 6,
/* Continue the process.  */
PTRACE_CONT = 7,
/* Kill the process.  */
PTRACE_KILL = 8,
/* Single step the process. This is not supported on all machines.  */
PTRACE_SINGLESTEP = 9,
/* Get all general purpose registers used by a processes.
   This is not supported on all machines.  */
 PTRACE_GETREGS = 12,
```

interlude - the ptrace API interface - cont

```
    /* Set all general purpose registers used by a processes.
       This is not supported on all machines.  */
     PTRACE_SETREGS = 13,
    [snip]
    /* Attach to a process that is already running. */
    PTRACE_ATTACH = 16,
    /* Detach from a process attached to with PTRACE_ATTACH.  */
    PTRACE_DETACH = 17,
    [snip]
    /* Continue and stop at the next (return from) syscall.  */
    PTRACE_SYSCALL = 24
  };
```

Tracing Thread mode overview

- each UML process gets a process on the host
- the tracing thread system call tracing on the UML processes (via ptrace)
- the tracing thread nullified system calls, and caused the process to enter the UML kernel, which is mapped into the upper part of its address space

All user processes share the UML kernel's address space! without due care, they can write it and escapte the virtual machine. With due care, performance suffers.

guest process system calls

- A process running inside UML (guest process) executes a system call instruction (int 0x80)
- via ptrace, the tracing thread is woken up
- the tracing thread annuls the system call on behalf of the UML process, and then forces the kernel to execute the system call
- the system call is executed, and when it is done, the tracing thread is woken up again
- the tracing thread manipulates the UML process state to think it completed the system call
- the UML process continues running

SKAS mode

- the UML kernel runs in an entirely different host address space from its processes
- solves the security problem - UML kernel totally inaccessible to UML processes
- also solves the fingerprinting problem - guest processes address space now identical to what they would be on a non UML host
- major speedup by eliminating signal delivery(?)

SKAS mode - /proc/mm

Let’s look at what is required in order to support seperate address spaces...

go over the host-skas3.patch

SKAS mode - switch_mm

schedule(), the scheduler function, calls switch_mm() before switching tasks via switch_to(). switch_mm() is implemented in asm/um/mmu_context.h as:

```
static inline void switch_mm(struct mm_struct *prev,
                             struct mm_struct *next,
                             struct task_struct *tsk,
                             unsigned cpu)
{
        if(prev != next){
                clear_bit(cpu, &prev->cpu_vm_mask);
                set_bit(cpu, &next->cpu_vm_mask);
                if(next != &init_mm) {
                        int fd = next->context.skas.mm_fd;
                        CHOOSE_MODE((void) 0,
                                    switch_mm_skas(fd));
                }
        }
}
```

SKAS mode - switch_mm_skas

```
void switch_mm_skas(int mm_fd)
{
        int err;

        err = ptrace(PTRACE_SWITCH_MM, userspace_pid, 0, mm_fd);
        if(err)
                panic("switch_mm_skas - PTRACE_SWITCH_MM failed,"
                      "errno = %d\n", errno);
}
```

References

- The User Mode Linux homepage:
  http://user-mode-linux.sourceforge.net/
- The User Mode Linux SKAS page:
  http://user-mode-linux.sourceforge.net/skas.html
- The User Mode Linux Community Site:
  http://usermodelinux.org/