L41 - Lecture 4: The Process Model (2)

Dr Robert N. M. Watson

4 March 2015
Reminder: last time

1. The process model and its evolution
   - *Isolation*
   - *Controlled communication* to kernel and other processes
     - Kernel must initiate communication, but can continue after return

2. Brutal pre-introduction to virtual memory

3. Where do programs come from?

4. Traps and system calls

5. Reading for next time
This time: more about the process model

1. More on traps and system calls
2. Virtual memory support for the process model
3. Threads and the process model
4. Readings for next time
System calls

- **System calls** allow user processes to request kernel services
  - `read()` reads data from a file descriptor to user memory
  - `fork()` creates a new process

- Exposed to userspace as system-library functions (e.g., libc)
- Under the hood, a *hardware trap* transfers control to the kernel
- Once the work is done, the kernel returns control to userspace

- Mostly synchronous, like normal C functions, but not always:
  - `_exit()` never returns
  - `sigreturn()` returns ... but not to a caller
  - `fork()` returns ... twice

- Even if a call is synchronous, its work is often asynchronous
  - `send()` writes data to a socket .. to get somewhere eventually
  - `aio_write()` explicitly performs an asynchronous write; later calls to `aio_return()` / `aio_error()` collect results
System-call invocation from user to kernel

- libc system-call function stubs provide linkable symbols
- Stubs can execute system-call instructions directly, or use dynamic implementations
  - Linux vdso
  - Xen hypercall page
- Low-level vector calls syscall()
- System-call prologue runs (e.g., breakpoints, tracing)
- Actual kernel service invoked
- System-call epilogue runs (e.g., more tracing, signal delivery)
The system-call table: `syscalls.master`

```
33 AUE_ACCESS    STD { int access(char *path, int amode); }
34 AUE_CHFLAGS   STD { int chflags(const char *path, u_long flags); }
35 AUE_FCHFLAGS  STD { int fchflags(int fd, u_long flags); }
36 AUE_SYNC      STD { int sync(void); }
37 AUE_KILL      STD { int kill(int pid, int signum); }
38 AUE_STAT      COMPAT { int stat(char *path, struct ostat *ub); }
39 AUE_GETPPID   STD { pid_t getppid(void); }
```

NB: If this looks like RPC stub generation .. that’s because it is.
Security and reliability

- System calls perform work on behalf a user thread
  - Work authorised by the thread’s credential
  - Resources (e.g., CPU time, memory) billed to the thread
  - Debugging/profiling information exposed to the thread’s owner

- Kernel interface is key *Trusted Computing Base* (TCB) surface
  - *Isolation* goals: *integrity*, *confidentiality*, *availability*
  - Scope global effects except as specified for service
  - Enforce access-control policies on all operations
  - Provide mechanisms for accountability (e.g., event auditing)

- But the kernel cannot trust user thread
  - Handle failures gracefully: terminate process, not kernel
  - Avoid priority inversions, unbounded resource allocation, etc
  - Confidentiality is expensive; e.g., zero pages, structure padding
  - Be aware of *covert channels*, *side channels*

- User code is the adversary – may try to break isolation
  - System-call arguments and return values are data, not code
  - Access user addresses safely (e.g., `copyin()`, `copyout()`)
System calls

System-call entry – the guts: syscallEnter

cred_update_thread
sv_fetch_syscall_args
ktrsyscall
ptrace_stop
IN_CAPABILITY_MODE
syscall_thread_enter
sytrace_probe_func
AUDIT_SYSCALL_ENTER
sa->callp->sy_call
AUDIT_SYSCALL_EXIT
sytrace_probe_func
syscall_thread_exit
sv_set_syscall_retval

Update thread cred from process
ABI-specific copyin() of arguments
ktrace syscall entry
ptrace syscall entry breakpoint
Capsicum capability-mode check
Thread drain barrier (module unload)
DTrace system-call entry probe
Security event auditing
System-call implementation! Woo!
Security event auditing
DTrace system-call return probe
Thread drain barrier (module unload)
ABI-specific return value
System calls

getauid: return process audit ID

```c
int
sys_getauid(struct thread *td, struct getauid_args *uap)
{
    int error;

    if (jailed(td->td_ucred))
        return (ENOSYS);
    error = priv_check(td, PRIV_AUDIT_GETAUDIT);
    if (error)
        return (error);
    return (copyout(&td->td_ucred->cr_audit.ai_auid, uap->auid,
                    sizeof(td->td_ucred->cr_audit.ai_auid)));
}
```

- Current thread, system-call argument structure
  - Security checks: lightweight virtualisation, privilege
  - Copy value to user address space – can’t write to it directly!
  - No synchronisation as all fields thread-local

- Does it matter how fresh the credential pointer is?
System calls

System-call return – the guts: syscallret

userret
→ KTRUSERRET
→ g_waitidle
→ addupc_task
→ sched_userret

p_throttled
ktrsysret
ptracestop
thread_suspend_check
P_PPWAIT

Complicated things like signals
ktrace syscall return
Wait for disk probe to settle
System-time profiling charge
Scheduler adjusts priority
... various debugging assertions ...
racct resource throttling
Kernel tracing: syscall return
ptrace syscall return breakpoint
Single-threading check
vfork wait

- That is a lot of stuff that largely never happens
- The trick is making all this nothing fast – e.g., via a small number of per-thread flags and globals that remain in the cache
System calls

System calls in practice: `dd`

```plaintext
# time dd if=/dev/zero of=/dev/null bs=10m count=1 status=none
0.000u 0.396s 0:00.39 100.0% 25+170k 0+0io 0pf+0w

syscall:::entry /execname == "dd"/ {
    self->start = timestamp;
    self->insyscall = 1;
}

syscall:::return /execname == "dd" && self->insyscall != 0/ {
    length = timestamp - self->start;
    @syscall_time[probefunc] = sum(length);
    @totaltime = sum(length);
    self->insyscall = 0;
}

END {
    printa(@syscall_time);
    printa(@totaltime);
}
```
System calls

System calls in practice: dd (2)

```bash
# time dd if=/dev/zero of=/dev/null bs=10m count=1 status=none
0.000u 0.396s 0:00.39 100.0% 25+170k 0+0io 0pf+0w
```

sysarch 7645
issetugid 8900
lseek 9571
sigaction 11122
clock_gettime 12142
ioctl 14116
write 29445
readlink 49062
access 50743
sigprocmask 83953
fstat 113850
munmap 154841
close 176638
lstat 453835
openat 562472
read 697051
mmap 770581

3205967

NB: ≈ 3ms total – but `time(1)` reports 396ms system time?
Traps in practice: `dd` (1)

```c
syscall:::entry /execname == "dd"/ {
    @syscalls = count();
    self->insyscall = 1;
    self->start = timestamp;
}

syscall:::return /execname == "dd" && self->insyscall != 0/ {
    length = timestamp - self->start; @syscall_time = sum(length);
    self->insyscall = 0;
}

fbt::trap:entry /execname == "dd" && self->insyscall == 0/ {
    @traps = count(); self->start = timestamp;
}

fbt::trap:return /execname == "dd" && self->insyscall == 0/ {
    length = timestamp - self->start; @trap_time = sum(length);
}

END {
    printa(@syscalls); printa(@syscall_time);
    printa(@traps); printa(@trap_time);
}
```

NB: 65 system calls at $\approx 3$ms; 5185 traps at $\approx 381$ms! But which traps?
Traps in practice: dd (1)

profile-997 /execname == "dd"/ { @traces[stack()] = count(); }

...  

kernel `PHYS_TO_VM_PAGE+0x1
kernel `trap+0x4ea
kernel `0xffffffff80e018e2
default_handler
ekerneldriver+0x1
kernel `trap+0x4ea
kernel `0xffffffff80e018e2
default_handler
ekerneldriver+0x1

...  

kernel `vm_map_lookup_done+0x1
kernel `trap+0x4ea
kernel `0xffffffff80e018e2
default_handler
ekerneldriver+0x1

...  

kernel `pagezero+0x10
kernel `trap+0x4ea
kernel `0xffffffff80e018e2
default_handler
ekerneldriver+0x1

A sizeable fraction of time is spent in pagezero: on-demand zeroing of previously untouched pages; but why \( \approx 5120 \) faults?

This is ironic, as the kernel is presumably filling pages with zeroes only to immediately \texttt{copyout()} zeros to it from /dev/zero
Revisiting virtual memory

So: back to virtual memory (VM)

- The process model’s isolation guarantees incur real expenses
- But the virtual-memory subsystem works quite hard to avoid them
  - Memory sharing – and Copy-on-Write, ‘page flipping’
  - ‘Page flipping’: both process/kernel and between processes
  - Background page zeroing
  - Superpages to improve TLB efficiency
- VM optimisation avoids work, but also manage memory footprint
  - Memory as a cache of secondary storage (files, swap)
  - Demand paging vs. I/O clustering
  - LRU / Preemptive swapping/paging to maintain free page pool
  - Working-set modelling
  - Memory compression and deduplication
- These ideas were known before Mach, but ...
  - Acetta, et al turn them into an art form
  - Provide a model beyond V→P mappings in page tables
  - And ideas such as the message-passing–shared-memory duality
Revisiting virtual memory

Last time: virtual memory (quick but painful primer)

Virtual address space 1

Virtual address space 2

Physical memory

Heap

Stack

Library

Code

Kernel

Dr Robert N. M. Watson

L41 - Lecture 4: The Process Model (2)
A (kernel) programmer model for virtual memory

Machine-independent virtual memory (VM)

Stack
- Read/write, grows down, anonymous object
- "vm_map_entry"
- "vm_object"
- anonymous swap-backed VM object
- “vm_pager”
- swap pager
- page
- page
- page

Heap
- Read/write, anonymous object
- "vm_page"
- "vm_object"
- anonymous swap-backed VM object
- “vm_pager”
- swap pager
- page
- page

Library
- Read/copy-on-write, named object
- "vm_object"
- shadow anonymous swap-backed VM object
- “vm_pager”
- swap pager
- page
- page

Code
- Read/copy-on-write, named object
- "vm_object"
- vnode VM object
- "vm_pager"
- vnode pager
- page
- page

“vmipmap”, “vm_map”

Machine-dependant physical map (PMAP)

Dr Robert N. M. Watson L41 - Lecture 4: The Process Model (2) 4 March 2015 17 / 19
Mach VM in other operating systems

- In Mach, VM mappings, objects, pages, etc, were first-class objects exposed to userspace via system calls.
- In two directly derived systems, quite different stories:
  - **Mac OS X** Although XNU is not a microkernel, Mach’s VM/IPC APIs are visible to applications, and used frequently.
  - **FreeBSD** Mach VM is used as a foundation and are only available as a Kernel Programming Interface (KPI).

- In FreeBSD, Mach VM KPIs are used:
  - In efficiently implement UNIX APIs such as `fork()` and `execve()`.
  - For memory-management APIs such as `mmap()` and `mprotect()`.
  - By the filesystem to implement a merged VM-buffer cache.
  - By device drivers that manage memory in interesting ways (e.g., GPU drivers mapping pages into user processes).
  - By a set of VM worker threads, such as the page daemon, swapper, syncer, and page-zeroing thread.
For next time

- The first lab: DTrace and I/O
- Dig into processes, system calls, etc
- Gregg and Mauro, Chapter 1 (*Introduction to DTrace*) and Chapter 2 (*D Language*)
- Handout *L41: DTrace Quick Start*

If you are having trouble getting hold of the course texts: Please ask the department librarian or your college librarian to order copies.