# L41 - Lecture 4: The Process Model (2)

Dr Robert N. M. Watson

3 November 2015

# Reminder: last time

1. The process model and its evolution
   - *Isolation* via *virtual addressing* and *rings*
   - *Controlled transition* to kernel via *traps*
   - *Controlled communication* to other processes via the kernel
2. Brutal introduction to virtual memory
3. Programs: *ELF* and *run-time linking*

# This time: more about the process model

1. More on traps and system calls
   - Synchrony and asynchrony
   - Security and reliability
   - Entry and return
2. Virtual memory support for the process model
3. ~~Threads and the process model~~
4. Readings for next time

# System calls

- User processes request kernel services via *system calls*; e.g.,
  - `open()` opens a file and returns a file descriptor
  - `fork()` creates a new process
- System calls exposed via library functions (e.g., in `libc`)
  - Function triggers *trap* to transfer control to the kernel
  - Arguments and return values copied in/out of kernel
  - Kernel returns control to userspace once done
- Some quirks relative to normal APIs; e.g.,
  - C return values via normal ABI calling convention...
  - ... but also per-thread `errno` to report error conditions
  - ... `EINTR`: for some calls, work got interrupted, try again

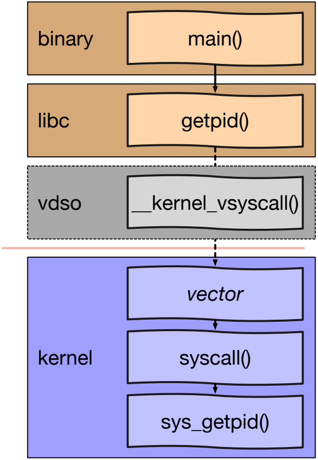
# System-call synchrony

- Some syscalls manipulate control flow or process/thread life cycle
  - `_exit()` never returns
  - `fork()` returns ... twice
  - `pthread_create()` creates a new thread
- But most syscalls behave like C functions and are *synchronous*
  - Called with arguments (by value, by reference)
  - Return values (an integer/pointer, or by reference)
  - When the caller regains control, the work is done
  - `getpid()` retrieves the *process ID* via return value
  - `read()` reads data from a file: on return, data is in buffer

# System-call asynchrony

- However, synchronous syscalls often perform *asynchronous* work
  - Some types of work may not be complete on system-call return
  - `write()` writes data to a file .. goes to disk eventually
  - Caller can re-use buffer immediately ('copy semantics')
  - `mmap()` maps a file but doesn't load data
  - Caller will trap on attempted access; trigger actual I/O
- Some syscalls are explicitly asynchronous
  - `aio_write()` explicitly requests an asynchronous write
  - Calls to `aio_return()/aio_error()` collect results later
  - Caller must wait to re-use buffer ('shared semantics')

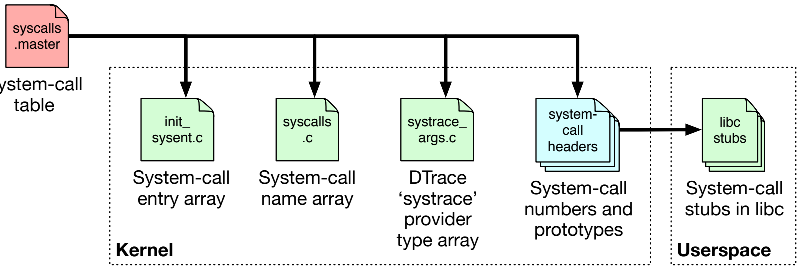
# System-call invocation from user to kernel




- `libc` system-call function stubs provide linkable symbols
- Stubs inline system-call instructions, or use dynamic implementations
  - Linux, FreeBSD `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); }
...
```




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

# Security and reliability

- Kernel interface is key *Trusted Computing Base* (TCB) surface
  - "Minimum software required for the system to be secure"
- Foundational security goal: *isolation*
  - *Integrity*, *confidentiality*, *availability*
  - Limit scope of effects of calls
  - Enforce access control on all operations (e.g., DAC)
  - Accountability mechanisms (e.g., event auditing)
- System calls perform work on behalf a user thread
  - Credential *unforgeably* tied to process/threads
  - Thread credential authorises work kernel performs
  - Resources (e.g., CPU, memory) billed to the thread
- Kernel must be robust to user-thread misbehaviour
  - Handle failures gracefully: terminate process, not kernel
  - Avoid priority inversions, unbounded resource allocation, etc.

# Security and reliability (cont)

- Confidentiality is both hard and expensive
  - Explicitly zero memory before re-use between security domains
  - Prevent kernel-user data leaks (e.g., in structure padding)
  - Be aware of *covert channels*, *side channels*
- User code is the adversary – may try to break isolation
  - Kernel must carefully enforce all access-control rules
  - System-call arguments and return values are data, not code
  - Extreme care with user-originated pointers
- User passes kernel pointer to system call
  - System-call arguments must be processed with rights of user code
  - E.g., prohibit `read()` from passing kernel pointer so that in-kernel credentials can be overwritten
  - Explicit `copyin()`, `copyout()` check pointer validity, copy data
- Kernel dereferences user pointer by accident
  - Kernel bugs could cause kernel to access user memory by mistake
  - Kernel `NULL`-pointer vulnerabilities
  - Intel Supervisor Mode Access Prevention (SMAP)

# System-call entry – the guts: `syscallenter`

| | |
|---|---|
| `cred_update_thread` | Update thread cred from process |
| `sv_fetch_syscall_args` | ABI-specific `copyin()` of arguments |
| `ktrsyscall` | `ktrace` syscall entry |
| `ptracestop` | `ptrace` syscall entry breakpoint |
| `IN_CAPABILITY_MODE` | Capsicum capability-mode check |
| `syscall_thread_enter` | Thread drain barrier (module unload) |
| `systrace_probe_func` | DTrace system-call entry probe |
| `AUDIT_SYSCALL_ENTER` | Security event auditing |
| `sa->callp->sy_call` | **System-call implementation! Woo!** |
| `AUDIT_SYSCALL_EXIT` | Security event auditing |
| `systrace_probe_func` | DTrace system-call return probe |
| `syscall_thread_exit` | Thread drain barrier (module unload) |
| `sv_set_syscall_retval` | ABI-specific return value |

- That's a lot of tracing hooks – why so many?

# getauid: return process audit ID

```
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-call return – the guts: `syscallret`

| | |
|---|---|
| `userret` | Complicated things like signals |
| → `KTRUSERRET` | `ktrace` syscall return |
| → `g_waitidle` | Wait for disk probe to settle |
| → `addupc_task` | System-time profiling charge |
| → `sched_userret` | Scheduler adjusts priority |
| | ... various debugging assertions ... |
| `p_throttled` | `racct` resource throttling |
| `ktrsysret` | Kernel tracing: syscall return |
| `ptracestop` | `ptrace` syscall return breakpoint |
| `thread_suspend_check` | Single-threading check |
| `P_PPWAIT` | `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 in practice: dd

```
# 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 in practice: dd (2)

```
# 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: ≈ 3.2ms total – but `time(1)` reports 396ms system time?

# Traps in practice: `dd` (1)

```
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);
}
```

```
               65
          2953756
             5185
        380762894
```

NB: 65 system calls at ≈3ms; 5185 traps at ≈381ms! But which traps?

# Traps in practice: `dd` (1)

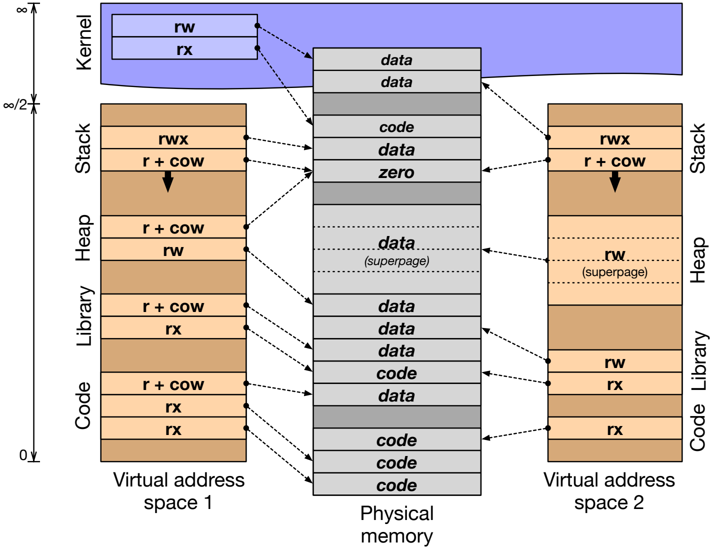
```
profile-997 /execname == "dd"/ { @traces[stack()] = count(); }

...
              kernel`PHYS_TO_VM_PAGE+0x1
              kernel`trap+0x4ea
              kernel`0xffffffff80e018e2
                5

              kernel`vm_map_lookup_done+0x1
              kernel`trap+0x4ea
              kernel`0xffffffff80e018e2
                5

              kernel`pagezero+0x10
              kernel`trap+0x4ea
              kernel`0xffffffff80e018e2
              346
```

- A sizeable fraction of time is spent in `pagezero`: on-demand zeroing of previously untouched pages
- Ironically, the kernel is filling pages with zeroes only to immediately `copyout()` zeros to it from `/dev/zero`

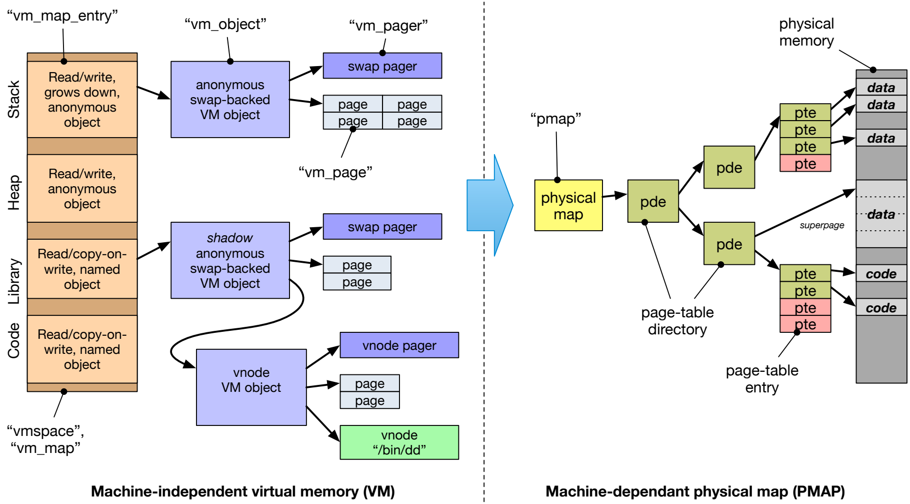
# So: back to virtual memory (VM)

- The process model's isolation guarantees incur real expenses
- But the virtual-memory subsystem works quite hard to avoid them
  - *Shared memory*, *copy-on-write*, *page flipping*
  - Background page zeroing
  - Superpages to improve TLB efficiency
- VM optimisation avoids work, but also manages 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
  - 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*

# Last time: virtual memory (quick but painful primer)



# A (kernel) programmer model for virtual memory



# 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:
  - To efficiently implement UNIX's `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 second lab: DTrace and I/O
- Begin to explore Inter-Process Communication (IPC) performance

- Ellard and Seltzer 2003

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