The Process Model (2)

L41 Lecture 4

Dr Robert N. M. Watson

15 November 2017

Reminder: last time

- The process model and its evolution
  - **Isolation** via **virtual addressing** and **rings**
  - **Controlled transition** to kernel via **traps**
  - **Controlled communication** to other processes intitiated via the kernel
- Brutal (re,pre)-introduction to **virtual memory**
- Where processes come from: the **process life cycle**, **ELF** and **run-time linking**

This time: the process model (2)

- More on **traps** and **system calls**
  - **Synchrony** and **asynchrony**
  - **Security** and **reliability**
  - Kernel work in system calls and traps
- **Virtual memory** support for the process model
- Readings for next time

System calls

- User processes request kernel services via **system calls**:
  - **Traps** that model **function-call semantics**; e.g.,
  - `open()` opens a file and returns a file descriptor
  - `fork()` creates a new process
- System calls appear to be library functions (e.g., `libc`)
  1. Function triggers trap to transfer control to the kernel
  2. System-call arguments copied into kernel
  3. Kernel implements service
  4. System-call return values copied out of kernel
  5. Kernel returns from trap to next user instruction
- 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

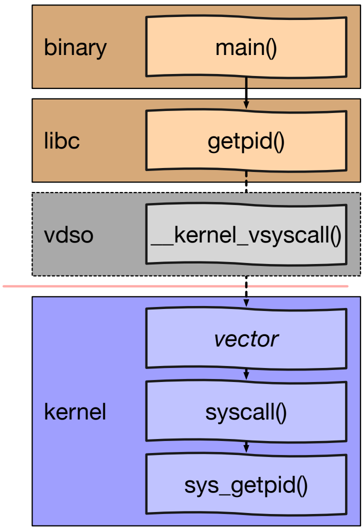
System-call synchrony

- Most syscalls behave like **synchronous** C functions
  - Calls with arguments (**by value** or **by reference**)
  - Return values (an integer/pointer or by reference)
  - Caller regains control when the work is complete; e.g.,
    - `getpid()` retrieves the **process ID** via a return value
    - `read()` reads data from a file: on return, data in buffer
- Except .. some syscalls manipulate **control flow** or **process thread/life cycle**; e.g.:
  - `_exit()` never returns
  - `fork()` returns … twice
  - `pthread_create()` creates a new thread
  - `setucontext()` rewrites thread register state

System-call asynchrony

- Synchronous calls can perform **asynchronous work**
  - Some work may not be complete on return; e.g.,
  - `write()` writes data to a file .. to disk .. eventually
  - Caller can re-use buffer immediately (**copy semantics**)
  - `mmap()` maps a file but doesn't load data
  - Caller traps on access, triggering I/O (**demand paging)**
  - Copy semantics mean that user program can be unaware of asynchrony (… sort of)
- Some syscalls have **asynchronous call semantics**
  - `aio_write()` requests an asynchronous write
  - `aio_return()/aio_error()` collect results later
  - Caller must wait to re-use buffer (**shared semantics**)

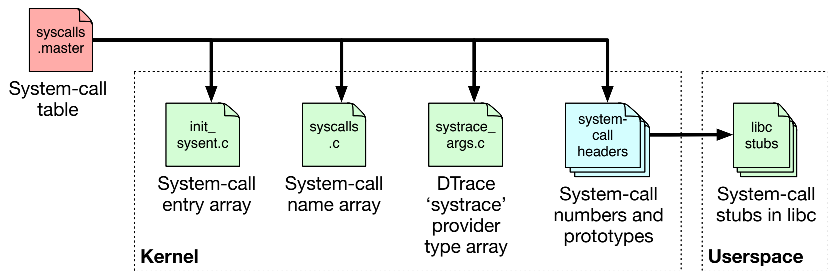
System-call invocation



- `libc` system-call stubs provide linkable symbols
- Inline system-call instructions or dynamic implementations
  - Linux `vdso`
  - Xen **hypercall page**
- **Machine-dependent trap vector**
- **Machine-independent** function `syscall()`
  - Prologue (e.g., breakpoints, tracing)
  - Actual service invoked
  - Epilogue (e.g., tracing, signal delivery)

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 (1)

- User-kernel interface is a key **Trusted Computing Base (TCB)** surface
  - *Minimum software required for the system to be secure*
- Foundational security goal: **isolation**
  - Used to implement **integrity**, **confidentiality**, **availability**
  - Limit scope of system-call effects on global state
  - Enforce access control on all operations (e.g., MAC, DAC)
  - Accountability mechanisms (e.g., event auditing)

L41 Lecture 4 - The Process Model (2) 9

Security and reliability (2)

- System calls perform work on behalf of user code
  - **Kernel thread** operations implement system call/trap
- **Unforgeable credential** tied to each process/thread
  - Authorises use of kernel services and objects
  - Resources (e.g., CPU, memory) billed to the thread
  - Explicit checks in system-call implementation
  - Credentials may be cached to authorise asynchronous work (e.g., TCP sockets, NFS block I/O)
- Kernel must be robust to **user-thread misbehaviour**
  - Handle failures gracefully: terminate process, not kernel
  - Avoid priority inversions, unbounded resource allocation, etc.

L41 Lecture 4 - The Process Model (2) 10

Security and reliability (3)

- **Confidentiality** is both difficult and expensive
  - Explicitly zero memory before re-use between processes
  - Prevent kernel-user data leaks (e.g., in struct padding)
  - Correct implementation of process model via rings, VM
  - **Covert channels**, **side channels**
- User code is the adversary – may try to break access control or isolation
  - Kernel must carefully enforce all access-control rules
  - System-call arguments, return values are data, not code
  - Extreme care with user-originated pointers, operations

Security and reliability (4)

- What if a user process passes a kernel pointer to system call?
  - System-call arguments must be processed with rights of user code
  - E.g., prohibit `read()` from storing via kernel pointer, which might (e.g.,) overwrite in-kernel credentials
  - Explicit `copyin()`, `copyout()` routines check pointer validity, copy data safely
- Kernel dereferences user pointer by accident
  - Kernel bugs could cause kernel to access user memory “by mistake”, inappropriately trusting user code or data
  - Kernel `NULL`-pointer vulnerabilities
  - Intel Supervisor Mode Access Prevent (SMAP), Supervisor Mode Execute Prevention (SMEP)
  - ARM Privileged eXecute Never (PXN)

System-call entry – `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** pointer, system-call **argument structure**
  - Security: **lightweight virtualisation**, **privilege check**
  - Copy value to user address space – can't write to it directly!
  - No explicit synchronisation as fields are thread-local
- Does it matter how fresh the credential pointer is?

System-call return – `syscallret`

`userret`	Complicated things, like signals
➟ `KTRUSERRET`	`ktrace` syscall return
➟ `g_waitidle`	Wait for disk probing to complete
➟ `addupc_task`	System-time profiling charge
➟ `sched_userret`	Scheduler adjusts priorities
	… 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 of this nothing fast – e.g., via per-thread flags and globals that remain in the data cache

System calls in practice: dd (1)

```
# 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);
}
```

Traps 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
```

```
       65
  2953756
     5185
380762894
```

65 system calls

2.95ms in system calls

5,185 traps

380.76ms in traps

- 65 system calls at ≈3ms; 5,185 traps at ≈381ms!
- But which traps?

traps in practice: dd (3)

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

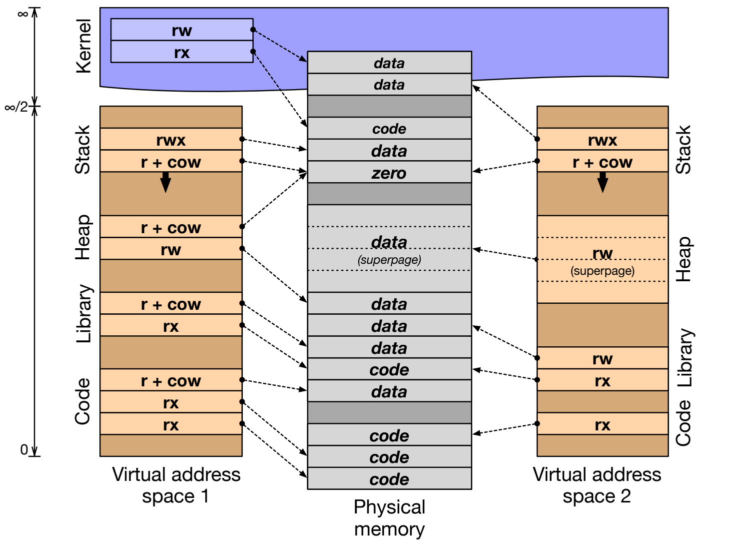
```
...
              kernel`PHYS_TO_VM_PAGE+0x1
              kernel`trap+0x4ea
              kernel`0xffffffff80e018e2
                5

              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 demand filling pages with zeroes only to `copyout()` zeroes to it from `/dev/zero`

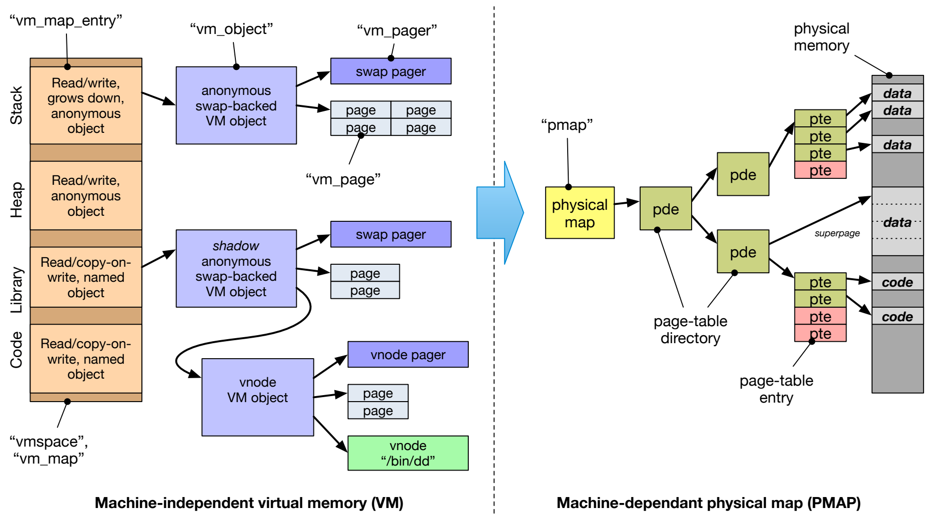
Last time: virtual memory (quick, painful)



So: back to Virtual Memory (VM)

- The process model's isolation guarantees incur real expense
- The VM subsystem works quite hard to avoid expense
  - **Shared memory**, **copy-on-write**, **page flipping**
  - **Background page zeroing**
  - **Superpages** to improve TLB efficiency
- VM 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 to maintain free-page pool
  - Recently: **memory compression** and **deduplication**
- These ideas were known before Mach, but…
  - Acetta, et al. impose principled design, 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**

Kernel programmer view of VM



Mach VM in other operating systems

- **Mach**: VM mappings, objects, pages, etc., are first-class kernel services exposed via system calls
- In two directly derived systems, quite different stories:

Mac OS X	Although not a microkernel, Mach's VM/IPC Application Programming Interfaces (APIs) are available to user programs, and widely used for IPC, debugging, …
FreeBSD	Mach VM is used as a foundation for UNIX APIs, but is available for use only as a Kernel Programming Interface (KPI)

- In FreeBSD, Mach is used:
  - To efficiently implement UNIX's `fork()` and `execve()`
  - For memory-management APIs – e.g., `mmap()` and `mprotect()`
  - By VM-optimised IPC – e.g., `pipe()` and `sendfile()`
  - 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

- Lab 2: DTrace and IPC
  - Explore Inter-Process Communication (IPC) performance
  - Leads into Lab 3: microarchitectural counters to explain IPC performance

- Ellard and Seltzer 2003