The Process Model (2)

Lecture 4, Part 1: Traps and System Calls

Prof. Robert N. M. Watson

2021-2022

The process model (2)

- **Traps** and **system calls**
  - Synchrony and asynchrony
  - System-call structure
  - Security and reliability
- System calls and traps in practice
- **Virtual memory** for the process model

Lecture 4, Part 1

Lecture 4, Part 2

Lecture 4, Part 3

Traps and system calls

- Asymmetric domain transition, **trap**, shifts control to kernel
  - **Asynchronous traps**: e.g., timer, peripheral interrupts, Inter-Processor Interrupts (IPIs)
  - **Synchronous traps**: e.g., system calls, divide-by-zero, page faults
- $pc to **interrupt vector**: dedicated OS code to handle trap
- Key challenge: kernel must gain control safely, securely

RISC	User $pc saved, kernel $pc installed, priv. state switched (MMU, …) Kernel address space becomes available for insn fetch/load/store Reserved registers in ABI ($k0, $k1 - MIPS) or banking ($pc, $sp, ...) Software must save other state (i.e., GPRs, FPRs, status words, ...)
CISC	HW saves context to in-memory trap frame (variably sized?)

- Thread/process context switch, with suitable atomicity:
  - (1) trap to kernel, (2) save register context; (3) update kernel-internal state, (4) optionally change address space, (5) restore register context; (6) trap return

UNIX 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 (usually) 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

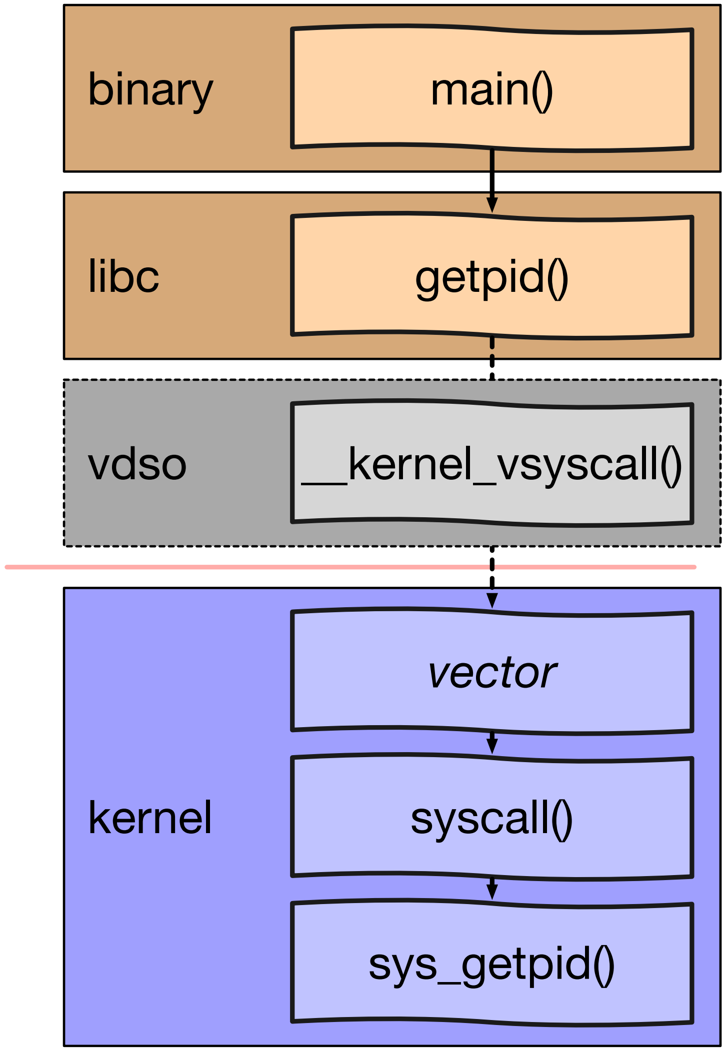
System-call synchrony

- 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 or socket .. eventually
    - Caller can re-use buffer immediately (**copy semantics**)
    - File writes are visible to other processes .. unless OS/HW fails
    - For IPC/socket writes, data may be enqueued but not yet sent
  - `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

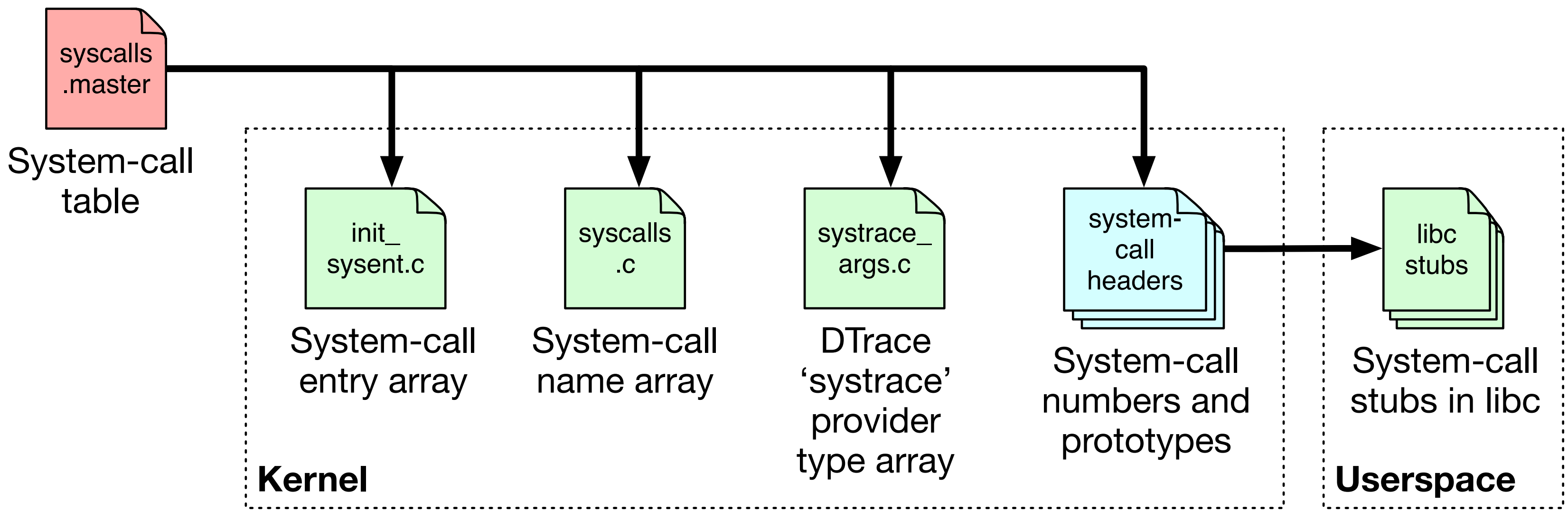


Note: This is something of a mashup of the system-call paths of different operating systems, to illustrate how the ideas compose

- `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)

FreeBSD 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); }
...
```



- If this looks like RPC stub generation .. that's because it is.
- In fact, if you read some of the original work on the BSD kernel, this design was intended to support system-call forwarding between hosts

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)

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.

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 process passes a kernel pointer as the buffer argument to the read() system call?
  - Without checks, the kernel might overwrite its own memory with data from a file or socket – e.g., the process's credential!
  - Goal: User-originated pointers are accessed with user privilege
    - Explicit `copyin()`, `copyout()` routines check pointer validity, copy data safely – and return errors, rather than faulting, on failure
- What if the kernel accidentally uses a user pointer?
  - Kernel bugs could cause the kernel to access user memory "by mistake", inappropriately trusting user code or data
    - E.g., the kernel accidentally calls a NULL kernel function pointer
    - Address 0 is in user-controlled memory, so the kernel runs whatever code is there in privileged mode!
  - Goal: Only permit intentional user memory access
    - Intel Supervisor Mode Access Prevent (SMAP), Supervisor Mode Execute Prevention (SMEP)
    - ARM Privileged eXecute Never (PXN)
- These are all examples of the **confused deputy problem**
  - Privileged code exercises its privilege in violation of a security policy
  - This vulnerability pattern exists in many other contexts