The Process Model (1)

L41 Lecture 3
Dr Robert N. M. Watson
2019-2020
Reminder: last time

• What is an operating system?
• Operating systems research
• About the module
• Lab reports

• DTrace
• The probe effect
• The kernel: Just a C program?
• A little on kernel dynamics: How work happens
This time: The process model

• The process model and its evolution
• Brutal (re, pre)-introduction to virtual memory
• Where do programs come from?
• Traps and system calls
• Reading for next time
The Process Model: 1970s foundations


- **Multics process model**
  - ‘Program in execution’
  - Process isolation bridged by controlled communication via supervisor (kernel)

- **Hardware foundations**
  - Supervisor mode
  - Memory segmentation
  - Trap mechanism

- Hardware protection rings (Schroeder and Saltzer, 1972)
The process model: today

- ‘Program in execution’
  - Process ≈ address space
  - Threads execute code
- Unit of resource accounting
  - Open files, memory, ...
- Kernel interaction via traps: system calls, page faults, ...
- Hardware foundations
  - Rings control MMU, I/O, etc.
  - Virtual addressing (MMU) to construct virtual address spaces
  - Trap mechanism
- Details vary little across {BSD, OS X, Linux, Windows, ...}
- Recently: OS-Application trust model inverted due to untrustworthy operating systems – e.g., Trustzone, SGX, ...
The UNIX process life cycle

- **fork()**
  - Child inherits address space and other properties
  - Program prepares process for new binary (e.g., stdio)
  - Copy-on-Write (COW)

- **execve()**
  - Kernel replaces address space, loads new binary, starts execution

- **exit()**
  - Process can terminate self (or be terminated)

- **wait4()** (et al)
  - Parent can await exit status

- **NB:** `posix_spawn()`?
Evolution of the process model

- **1980s**: Code, heap, and stack
- **1990s**: Dynamic linking, threading
- **2000s**: Scalable memory allocators implement multiple arenas (e.g., as in jemalloc)
- Co-evolution with virtual memory (VM) research
Process address space: dd(1)

- Inspect dd process address space with `procstat -v`

```
root@beaglebone:/data # procstat -v 734

<table>
<thead>
<tr>
<th>PID</th>
<th>START</th>
<th>END</th>
<th>PRT</th>
<th>RES</th>
<th>PRES</th>
<th>REF</th>
<th>SHD</th>
<th>FLAG</th>
<th>TP</th>
<th>PATH</th>
</tr>
</thead>
<tbody>
<tr>
<td>734</td>
<td>0x8000</td>
<td>0xd000</td>
<td>r-x</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>CN--</td>
<td>vn</td>
<td>/bin/dd</td>
</tr>
<tr>
<td>734</td>
<td>0x14000</td>
<td>0x16000</td>
<td>rw-</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>----</td>
<td>df</td>
<td></td>
</tr>
<tr>
<td>734</td>
<td>0x20014000</td>
<td>0x20031000</td>
<td>r-x</td>
<td>29</td>
<td>32</td>
<td>31</td>
<td>14</td>
<td>CN--</td>
<td>vn</td>
<td>/libexec/ld-elf.so.1</td>
</tr>
<tr>
<td>734</td>
<td>0x20038000</td>
<td>0x20039000</td>
<td>rw-</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>C---</td>
<td>vn</td>
<td>/libexec/ld-elf.so.1</td>
</tr>
<tr>
<td>734</td>
<td>0x20039000</td>
<td>0x20052000</td>
<td>rw-</td>
<td>16</td>
<td>16</td>
<td>1</td>
<td>0</td>
<td>----</td>
<td>df</td>
<td></td>
</tr>
<tr>
<td>734</td>
<td>0x20100000</td>
<td>0x2025f000</td>
<td>r-x</td>
<td>351</td>
<td>360</td>
<td>31</td>
<td>14</td>
<td>CN--</td>
<td>vn</td>
<td>/lib/libc.so.7</td>
</tr>
<tr>
<td>734</td>
<td>0x2025f000</td>
<td>0x20266000</td>
<td>---</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>----</td>
<td>df</td>
<td></td>
</tr>
<tr>
<td>734</td>
<td>0x20266000</td>
<td>0x2026e000</td>
<td>rw-</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>C---</td>
<td>vn</td>
<td>/lib/libc.so.7</td>
</tr>
<tr>
<td>734</td>
<td>0x2026e000</td>
<td>0x20285000</td>
<td>rw-</td>
<td>7</td>
<td>533</td>
<td>2</td>
<td>0</td>
<td>----</td>
<td>df</td>
<td></td>
</tr>
<tr>
<td>734</td>
<td>0x20400000</td>
<td>0x20c00000</td>
<td>rw-</td>
<td>526</td>
<td>533</td>
<td>2</td>
<td>0</td>
<td>--S-</td>
<td>df</td>
<td></td>
</tr>
<tr>
<td>734</td>
<td>0xbffe0000</td>
<td>0xc0000000</td>
<td>rwx</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>----D</td>
<td>df</td>
<td></td>
</tr>
</tbody>
</table>
```

- **r**: read
- **w**: write
- **x**: execute
- **C**: Copy-on-write
- **D**: Downward growth
- **S**: Superpage
ELF binaries

• UNIX: Executable and Linkable Format (ELF)
• Mac OS X/iOS: Mach-O; Windows: PE/COFF; same ideas
• Inspect dd ELF program header using objdump -p:

root@beaglebone:~ # objdump -p /bin/dd
/bin/dd: file format elf32-littlearm

Program Header:
0x70000001 off 0x00000469c vaddr 0x00000c69c paddr 0x00000c69c align 2**2
  filesz 0x00000158 memsz 0x00000158 flags r--
  PHDR off 0x00000034 vaddr 0x000008034 paddr 0x000008034 align 2**2
    filesz 0x000000e0 memsz 0x000000e0 flags r-x
  INTERP off 0x00000114 vaddr 0x000008114 paddr 0x000008114 align 2**0
    filesz 0x00000015 memsz 0x00000015 flags r--
  LOAD off 0x00000000 vaddr 0x000008000 paddr 0x000008000 align 2**15
    filesz 0x0000047f8 memsz 0x0000047f8 flags r-x
  LOAD off 0x0000047f8 vaddr 0x000147f8 paddr 0x000147f8 align 2**15
    filesz 0x000001b8 memsz 0x00001020 flags rw-
  DYNAMIC off 0x000004804 vaddr 0x00014804 paddr 0x00014804 align 2**2
    filesz 0x000000f0 memsz 0x000000f0 flags rw-
  NOTE off 0x0000012c vaddr 0x00000812c paddr 0x00000812c align 2**2
    filesz 0x0000004c memsz 0x0000004c flags r--
Virtual memory (quick but painful primer)
Virtual memory (quick but painful primer)

• **Memory Management Unit (MMU)**
  • Transforms *virtual addresses* into *physical addresses*
  • Memory is laid out in *virtual pages* (4K, 2M, 1G, ...)
  • Control available only to the supervisor (historically)
  • Software handles failures (e.g., store to read-only page) via *traps*

• **Page tables**
  • SW-managed *page tables* provide *virtual-physical mappings*
  • Access permissions, page attributes (e.g., caching), dirty bit
  • Various configurations + traps implement BSS, COW, sharing, ...

• **Translation Look-aside Buffer (TLB)**
  • Hardware cache of entries – avoid walking pagetables
  • Content Addressable Memory (CAM); 48? 1024? entries
  • TLB *tags*: entries *global* or for a specific *address-space ID (ASID)*
  • Software- vs. hardware-managed TLBs

• **Hypervisors and IOMMUs:**
  • I/O performs **direct memory access (DMA)** via virtual address space
Role of the run-time linker (rtld)

- **Static linking**: program, libraries linked into one binary
  - Process address space laid out (and fixed) at compile time

- **Dynamic linking**: program, libraries in separate binaries
  - Shared libraries avoid code duplication, conserving memory
  - Shared libraries allow different update cycles, ABI ownership
  - Program binaries contain a list of their library dependencies
  - The run-time linker (rtld) loads and links libraries
  - Also used for plug-ins via dlopen(), dlsym()

- Three separate but related activities:
  - **Load**: Load ELF segments at suitable virtual addresses
  - **Relocate**: Rewrite position-dependent code to load address
  - **Resolve symbols**: Rewrite inline/PLT addresses to other code
Role of the run-time linker (rtld)

• When the execve system call starts the new program:
  • ELF binaries name their **inter**preter in ELF metadata
  • Kernel maps rtld and the application binary into memory
  • Userspace starts execution in rtld
  • rtld loads and links dynamic libraries, runs constructors
  • rtld calls main()

• Optimisations:
  • **Lazy binding**: don’t resolve all function symbols at load time
  • **Prelinking**: relocate, link in advance of execution
  • Difference is invisible – but surprising to many programmers
Arguments and ELF auxiliary arguments

• C-program arguments are `argc`, `argv[]`, and `envv[]`:

```
root@beaglebone:/data # procstat -c 716
    PID   COMM     ARGS
    716   dd       dd if=/dev/zero of=/dev/null bs=1m
```

• The run-time linker also accepts arguments from the kernel:

```
root@beaglebone:/data # procstat -x 716
    PID   COMM     AUXV       VALUE
    716   dd       AT_PHDR    0x8034
    716   dd       AT_PHENT   32
    716   dd       AT_PHNUM   7
    716   dd       AT_PAGESZ  4096
    716   dd       AT_FLAGS   0
    716   dd       AT_ENTRY   0x8cc8
    716   dd       AT_BASE    0x20014000
    716   dd       AT_EXECPATH 0xbfffff4c
    716   dd       AT_OSRELDATE 1100062
    716   dd       AT_NCPUS   1
    716   dd       AT_PAGESIZES 0xbfffffff9c
    716   dd       AT_PAGESIZESLEN 8
...
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, handler $pc$ installed, control coprocessor (MMU, ...)
|              | Kernel address space becomes available for fetch/load/store
|              | Reserved registers in ABI ($k0, $k1) or banking ($pc, $sp, ...)
|              | Software must save other state (i.e., other registers) |
| CISC         | HW saves context to in-memory trap frame (variably sized?) |

• User context switch:
  • (1) trap to kernel, (2) save register context; (3) optionally change address space, (4) restore another register context; (5) trap return
Break
The Process Model (2)

L41 Lecture 4
Dr Robert N. M. Watson
2019-2020
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
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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>AUE_ACCESS</td>
<td>STD</td>
<td>{ int access(char *path, int amode); }</td>
</tr>
<tr>
<td>34</td>
<td>AUE_CHFLAGS</td>
<td>STD</td>
<td>{ int chflags(const char *path, u_long flags); }</td>
</tr>
<tr>
<td>35</td>
<td>AUE_FCHFLAGS</td>
<td>STD</td>
<td>{ int fchflags(int fd, u_long flags); }</td>
</tr>
<tr>
<td>36</td>
<td>AUE_SYNC</td>
<td>STD</td>
<td>{ int sync(void); }</td>
</tr>
<tr>
<td>37</td>
<td>AUE_KILL</td>
<td>STD</td>
<td>{ int kill(int pid, int signum); }</td>
</tr>
<tr>
<td>38</td>
<td>AUE_STAT</td>
<td>COMPAT</td>
<td>{ int stat(char *path, struct ostat *ub); }</td>
</tr>
</tbody>
</table>

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

---

**Diagram:**

![Diagram showing the system-call entry array, system-call name array, DTrace 'systrace' provider type array, system-call numbers and prototypes, and system-call stubs in libc.]
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 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 sv_fetch_syscall_args ktrsyscall ptracestop IN_CAPABILITY_MODE syscall_thread_enter systrace_probe_func AUDIT_SYSCALL_ENTER

sa->callp->sy_call AUDIT_SYSCALL_EXIT systrace_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

• That’s a lot of tracing hooks – why so many?
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_GETAUID);
    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 – syscall ret

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 probing to complete
System-time profiling charge
Scheduler adjusts priorities
... 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 of this nothing fast – e.g., via per-thread flags and globals that remain in the data cache
System calls in practice: dd (1)

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

cy Hispanic::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);
}
```

# 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

L41 Lecture 2 - The Process Model
System calls in practice: dd (2)

```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
```

<table>
<thead>
<tr>
<th>Call</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>sysarch</td>
<td>7645</td>
</tr>
<tr>
<td>issetugid</td>
<td>8900</td>
</tr>
<tr>
<td>lseek</td>
<td>9571</td>
</tr>
<tr>
<td>sigaction</td>
<td>11122</td>
</tr>
<tr>
<td>clock_gettime</td>
<td>12142</td>
</tr>
<tr>
<td>ioctl</td>
<td>14116</td>
</tr>
<tr>
<td>write</td>
<td>29445</td>
</tr>
<tr>
<td>readlink</td>
<td>49062</td>
</tr>
<tr>
<td>access</td>
<td>50743</td>
</tr>
<tr>
<td>sigprocmask</td>
<td>83953</td>
</tr>
<tr>
<td>fstat</td>
<td>113850</td>
</tr>
<tr>
<td>munmap</td>
<td>154841</td>
</tr>
<tr>
<td>close</td>
<td>176638</td>
</tr>
<tr>
<td>lstat</td>
<td>453835</td>
</tr>
<tr>
<td>openat</td>
<td>562472</td>
</tr>
<tr>
<td>read</td>
<td>697051</td>
</tr>
<tr>
<td>mmap</td>
<td>770581</td>
</tr>
</tbody>
</table>

3205967

- NB: ≈3.2ms 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: trap() FBT probes are machine-dependent and these examples are from x86_64.
On ARMv7, use `fbt::abort_handler:{entry,return}`.
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 system calls at ≈3ms; 5,185 traps at ≈381ms!
- But which traps?
traps in practice: dd (3)

```c
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
Virtual memory (quick, painful)

Virtual address space 1

Kernel

Stack

Heap

Library

Code

Zero

Physical memory

Virtual address space 2

L41 Lecture 2 - The Process Model
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

Machine-independent virtual memory (VM)

Stack
- "vm_map_entry"
- Read/write, grows down, anonymous object

Heap
- "vm_object"
- anonymous swap-backed VM object
- "vm_page"
- swap pager

Library
- "vm_object"
- shadow anonymous swap-backed VM object
- swap pager

Code
- "vm_object"
- vnode VM object
- vnode pager

"vmspace", "vm_map"

Machine-dependant physical map (PMAP)

physical memory
- physical map
  - pde
  - pde
  - pte
  - pte
  - superpage
  - data
  - code

page-table directory
- page-table entry
- "pmap"
- "pte"
- "pte"
- "pte"
- "pte"

Lecture 2 - The Process Model
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:

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

• 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

• Review ideas from the first lab report
• Lab 2: DTrace and IPC
  • Explore Inter-Process Communication (IPC) performance
  • Leads into Lab 3: microarchitectural counters to explain IPC performance

• McKusick, et al: Chapter 6 (Memory Management)
• Optional: Anderson, et al, on Scheduler Activations
  • (Exercise: where can we find scheduler-activation-based concurrent programming models today?)
• Ellard and Seltzer 2003