Week 5: Exokernels and Self-paging
Advanced Operating Systems
(263-3800-00L)
Timothy Roscoe
Herbstsemester 2012
http://www.systems.ethz.ch/courses/fall2012/AOS
Milestone 4: Handling page faults

• Barrelfish does this \textit{in the faulting process}!
  – “Self-paging”

• Understanding this involves:
  – OS architecture, esp. Exokernels
  – Object-based virtual memory
  – Self-paging itself
OS Architecture and Exokernels
What is “Operating System Architecture?"

• Coarse-grained structure of the OS
• How the complexity is factored
• Mapping onto:
  – Programming language features
  – Execution environment presented to applications
  – Address spaces
  – Hardware protection features (rings, levels, etc.)
  – Execution patterns (subroutines, threads, coroutines)
  – Hardware execution (interrupts, traps, call gates)
Architectural models

• There are many, and they are models!
  – Idealized, extreme view of how system is structured
• Real systems always entail compromises
  – Hard to convey ⇒ it’s good to build a few
• Think of these as tools for thinking about OSes
  – Each has its reasons
  – Solve particular problems at particular times
Outline

• Monolithic or component-based systems
• Kernel-based systems
• Microkernels
• Kernel thread models
  – Per-thread kernel stack
  – Single kernel stack
• Exokernel systems
  – Nemesis and Exokernel
• Multikernels
• Barrelfish
• References
1. Monolithic / component-based systems

- **Examples:**
  - Cedar [Swinehart et al., 1986]
  - TinyOS [Hill et al., 2004]
  - Oberon
  - Singularity [Hunt and Larus, 2007]

- **Hardware provides time multiplexing**
  - Interrupts
  - threads (in Cedar’s case)

- **Language provides modularity & protection**
  - Module calls
  - Inter-thread communication
Protection-based component-based systems

• Examples:
  – KeyKOS [Bromberger et al., 1992]
  – Pebble [Bruna et al., 1999]

• Even simpler kernel than microkernels
  – Kernel only mediates protection domain switches
  – Scheduling, threads, etc. implemented in “user space”

• Aimed at:
  – High security (very small TCB)
  – Embedded systems (highly configurable)
2. Kernel-based systems

• Examples: Unix [Thompson, 1974], VMS, Windows NT/XP/Vista/7
• Hardware enforces user vs. kernel mode
• Machine in user space multiplexed into address spaces
• Kernel provides:
  – All shared services
  – All device abstraction
3. Microkernels

- Examples: L4, Mach, Amoeba, Chorus
- Kernel provides:
  - Threads
  - Address spaces
  - IPC
- All other functionality in server processes
  - Device drivers
  - File systems
  - Etc.
- Instead of syscalls, applications send IPC to servers
Kernel thread models

• Key design choices when implementing an OS:
  – Support for > 1 execution context in the kernel?
  – Where is the stack for executing kernel code?
  – Can kernel code block?
  – If so, how?

• Result: the **kernel thread model**.
Kernel thread models

2 basic alternatives:

1. Per-thread kernel stack:
   – Every thread has a matching kernel stack

2. Single kernel stack:
   – Only one stack is used in the kernel (per core).
Per-thread kernel stack

- Every user thread/process has its own kernel stack
- Thread’s kernel state implicitly stored in kernel activation stack
- A kernel thread blocks → switch to another kernel stack
- Resuming: simply switch back to original stack
- Preemption is easy
- No conceptual difference between kernel- and user-mode

```c
example(arg1, arg2) {
    P1(arg1, arg2);
    if (need_to_block) {
        thread_block();
        P2(arg2);
    } else {
        P3();
    }
    /* return to user */
    return SUCCESS;
}
```
Single kernel stack

• Challenges:
  – How can a single kernel stack support many application processes/threads?
  – How to handle system calls that block?

• Two basic approaches:
  1. Continuations [Draves et al., 1991]
  2. Stateless kernel [Ford et al., 1999]
Continuations

- State to resume blocked thread explicitly saved in TCB
  - Function pointer
  - Variables
- Stack can be discarded and reused for new thread
- Resuming involves discarding current stack and restoring the continuation

```c
example(arg1, arg2)
{
    P1(arg1, arg2);
    if (need_to_block) {
        save_context_in_TCB;
        thread_block(example_continue);
        panic("thread_block returned");
    } else {
        P3();
    }
    thread_syscall_return(SUCCESS);
}

example_continue()
{
    recover_context_from_TCB;
    P2(recovered_arg2);
    thread_syscall_return(SUCCESS);
}
```
Stateless kernel

• System calls simply do not block within kernel
• If a system call must block:
  – User must restart call when resources are available
  – Kernel stack content discarded
• Preemption within kernel difficult
  – Must (partially) roll back to a restart point
  – But may not be necessary with careful design
• Avoid page faults within kernel code
  – System call arguments in registers
  – Nested page fault is fatal
Kernel stack model summary

Per-thread kernel stack

✓ Simple, flexible
  – Kernel can always use threads
  – No special technique for saving state when
    – interrupted/blocked
  – No conceptual difference between kernel and user mode

✗ Larger cache and memory footprint

• Used by L4Ka::Pistachio, UNIX, Linux, etc.
Kernel stack model summary

Single kernel stack
✓ Lower cache & memory footprint (always the same stack)

Continuations:
✗ Complex to program
✗ Must save state conservatively (whatever might be needed)

• Used by Mach, NICTA::Pistachio
Kernel stack model summary

Single kernel stack
✓ Lower cache & memory footprint (always the same stack)

Stateless kernel:
✗ Also complex to program
   – Must request all resources prior to execution
   – Blocking system calls must be restartable
✗ Processor-provided stack mgmt. can get in the way
   – System calls need to be atomic

• Used by Fluke, Nemesis, Exokernel, Barrelfish
Why build a stateless kernel?

• It is the simplest model, if all kernel invocations are:
  – Atomic
  – Non-blocking
  – Bounded and short-running
  – Non-preemptable
  – Guaranteed not to page fault

• Restrictive, but quite appropriate for a uniprocessor μkernel with no blocking IPC.
4. Exokernels

• Kernel provides minimal multiplexing of h/w
  – All other functionality in userspace *libraries*
  – Unlike microkernels, where this in servers
  – “LibraryOS” concept

• Enables:
  – Strong isolation between applications
  – High degree of application-specific policies
Exokernel systems

Two different systems. Two different motivations:

1. Complexity, adaptability, performance
   ⇒ Aegis [Kaashoek et al., 1997]

2. QoS crosstalk
   ⇒ Nemesis [Leslie et al., 1996]

• Both approaches are similar:
  – Exterminate OS abstractions
  – Move all code possible into the application’s address space ⇒ library OSes
Aegis motivation
Exterminate all OS abstractions!

• A traditional OS or a microkernel such as L4:
  – Multiplexes physical resources
  – Shared, secure access to CPU, memory, network, etc.
  – Abstracts the same physical resources
    • Processes/threads, address spaces, virtual file system, network stack
• Multiplexing is required for security
  . . . but why should an OS abstract what it multiplexes?
Nemesis motivation
Eliminate QoS crosstalk

Consider a network stack:
• Layered protocol implementation
• Multiplex at each layer
• Conceptually, each layer is a process
  – C.f. early Comer books

= Multiplexing point
Consequences

• Pluses:
  – Conceptually simple, easy to code
  – Efficient resource usage

• Minuses:
  – Application of packet only known at top of stack
  – QoS - every multiplexing point must schedule
  – Disaster for multimedia / realtime mix
  – “QoS Crosstalk”
Layered multiplexing considered harmful

- Mux once, down low
- Rx:
  - Find target app first
  - Then execute protocol
- Tx:
  - Construct the wire format
  - Check it and mux last
- All packets scheduled with the application
- Works great with circuits!
Server-based monolithic OS

- Video player
- Audio Server
- X Server
- Compiler
- Monolithic Kernel

Multiplexing points!
Multiplexing points in operating system kernels

• Every server process is effectively multiplexing some resource
  – needs to schedule it.
  – needs to know system-wide scheduling policy
  – must be trusted to apply it
  – has to cope with contention and locks
  – is operating outside the control of any application
Microkernels?

Complexity Nightmare!
Result: chaos and crosstalk

• System is full of shared servers
  – Each with resource contention
  – Each has no application knowledge
  – Multiple levels of dependency
  – No kernel scheduling algorithm can help
  – No IPC performance will help

• Relevant early ‘90s work:
  – “SVR4 scheduler unacceptable” paper
  – Processor Capacity Reserves (complexity!)
  – Resource Containers (ignore the problem!)
Nemesis

• Written for uniprocessor Alpha, 1992-95
• 64-bit single address space
  – Not a fundamental design motivation, as in Mungi
• “Multi-service operating system”
  – Mixture of soft real time, communication-oriented,
  – interactive, batch jobs
  – Designed for workstations
• Strong networking influence
  – Published in JSAC!
What is (in) an application?

- In an Exokernel, functionally, everything:
  - User code
  - Network stack
  - Filing system
  - Window system
  - Low-level I/O
  - Intra-application communication
Is this actually achievable?

• No.
• But very nearly...
• Kernel overhead
  – scheduling fudge factor
• Resource contention
  – move out of band (reservations/leases)
  – short, bounded atomic sections
## Nemesis Application domains

<table>
<thead>
<tr>
<th>Video player</th>
<th>Word processor</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads package</td>
<td>Coroutine package</td>
<td>Filing system</td>
</tr>
<tr>
<td>Window System</td>
<td>Window System</td>
<td>Network stack</td>
</tr>
<tr>
<td>Filing system</td>
<td>Filing system</td>
<td>Network stack</td>
</tr>
<tr>
<td>Network stack</td>
<td>Network stack</td>
<td>Disk driver</td>
</tr>
<tr>
<td>Disk driver</td>
<td>Disk driver</td>
<td>Video driver</td>
</tr>
<tr>
<td>Video driver</td>
<td>Video driver</td>
<td>Audio driver</td>
</tr>
<tr>
<td>Audio driver</td>
<td></td>
<td>Network driver</td>
</tr>
<tr>
<td>Network driver</td>
<td></td>
<td>Network driver</td>
</tr>
</tbody>
</table>

Nemesis Trusted Supervisor Code (NTSC)
Nemesis in action:
Programmability questions

• Isn’t it all rather complex to move functionality into the app?
  – No: libraries do what the kernel or servers used to do.

• Does the flexibility impact performance?
  – No: protection checks are mostly off the fast path
  – Each application can efficiently implement its policy

• What happens on a multiprocessor?
  – Unclear: a multiprocessor kernel requires plenty of embedded policy (e.g. locks)
  – Attempts to produce MP exokernels have not been as dramatically better at performance
  – See Barrellfish later...
Exokernel challenges

• Can you really expose all the hardware to the application and still stay sane?
• Can you multiplex the machine securely while removing (most) abstraction?
• Apparently, yes:
  – Threads and processes: see scheduler activations!
  – Networking: packet filtering
  – Disks (file systems): block or track-level protection, careful management of metadata
  – Window system: similar; blit tiles into protected windows
  – Self-paging
Self paging
Paging in a monolithic kernel

- Single page replacement policy in kernel
- Global allocation of page frames to processes
- Low overhead: single kernel entry
Lost flexibility?

• Claim: many applications are not well-served by traditional paged virtual memory abstractions
  – Garbage collected language runtimes
  – Database management systems
  – Multimedia (soft realtime) applications
  – Large out-of-core computations
External paging in a microkernel

- Page faults handled by separate process
  - Or processes...
- Policy can be moved into user space
- Different regions can have different pagers
Object-oriented virtual memory

- C.f. Mach, Chorus, L4, etc.

✓ Allows different:
  - Classes of paging region
  - Replacement policies
  - Pools of backing memory
  - Semantics of memory access

✗ Downsides:
  - Lots of kernel crossings
  - Lots of context switches
  - QoS crosstalk (shared pagers)
Self paging in Nemesis: goals

• Support guarantees to applications
  – Physical memory
  – Paging bandwidth to disk
• Support application-level policies
  – Page different areas differently
  – Custom page replacement
  – Allow full access to MMU facilities for other uses
• Approach:
  – Handle all page faults in the user program
  – Library provides Mach-like abstractions
Principles

• Control
  – Resource allocation between applications is guaranteed

• Power
  – Low-level (hence expressive) access to the MMU and physical memory

• Responsibility
  – All resource usage accounted to application
Self-paging in an exokernel

Application

Library OS

Message (upcall)

Page fault

Minimal exokernel

User space

Kernel space

Storage service

Application

Library OS

Application

Library OS

Application

Library OS
Architecture

• Virtual address allocator
  – Ranges of virtual addresses
  – Nemesis was a *single address space system*

• Physical frame allocator
  – Pages of RAM
  – Key resource: isolation between applications

• Association between the two entirely within the application
Architecture
Allocation

• Application has contract with frame allocator
  – $g$: Guaranteed frame limit
  – $x$: Optimistic frame limit
  – $n$: Number of frames allocated so far

• Allocation policy:
  – $n < g \Rightarrow$ allocation always succeeds
  – $g \leq n < x \Rightarrow$ optimistic allocation

• Frame allocator can revoke optimistic memory
Reclaiming memory without crashing the application

- Application maintains *frame stack*
  - List of PFNs ordered by importance
  - Each frame marked as *used* or *unused*

- Transparent revocation:
  - Frame allocator removes unused frames from top of stack (if it can...)

- Intrusive revocation:
  - Frame allocator asks application to make top k frames unused
Intrusive revocation of physical memory

1. Revocation notification

2. Application frees up top $k$ frames on the stack

3. Application replies that freeing done.

4. Frame allocator reclaims top $k$ frames from stack

Frame allocation

Frame Allocator

Memory manager

Application

Frame stack
Translation system

• High level:
  – Bootstrap and initialization
  – Creating and managing protection domains (collections of mappings)

• Low level: MMU access
  – Every (protection) domain can change mappings
  – Also protection information...
Fault handling

• General idea:
  – Page faults sent back to the application
  – Demultiplexed to the relevant stretch driver
    • Based on virtual address region
  – Stretch driver changes low-level mappings
    • Via low-level translation system
  – Faulting thread is made runnable again

• Many variations possible: application-specific
Fault handling in a domain

1. Thread takes a page fault.
2. Fault reflected back as upcall.
3. Upcall directed to fault handler.
4. Demultiplexed to appropriate stretch driver.
5. Stretch driver services fault immediately.
6. Faulting thread is resumed.

- Memory manager
- Upcall handler
- Fault handler
- Thread scheduler
- Event demux
- User space
- Kernel space
- Application threads
- Stretch driver

Thread scheduler

Faulting thread is resumed
Fault handling in a domain

5. Worker thread scheduled to service fault

6. Swap device accessed (read and/or write) to swap device

4. Fault demultiplexed to appropriate stretch driver

7. Worker thread services fault

Memory manager

Fault handler

Event demux

Thread scheduler

Upcall handler

Application threads

User space

Kernel space
Performance QoS

• Self-paging makes fault handling throughput dependent on application scheduling parameters
  – I.e. determined by the OS CPU scheduler

• A user-safe disk provides performance guarantees and isolation for demand paging
  – Linked by the OS to CPU guarantees

• Result:
  – real-time guarantees in the presence of paging?
Paging in: trace
Paging in: trace
Paging out: performance
Paging out: trace
Summary

• Microkernel-based OO virtual memory
  – Different kinds of memory
  – Complex control paths for faults and restarts
• Self-paging
  – Performance guarantees
  – Application-specific response to page faults
  – Same policy flexibility as Mach
  – Reduced complexity in control (though still more complex than Unix!)