Week 3:
OS Architecture, Threads, and Dispatch
(263-3800-00L)
Timothy Roscoe
Fall 2014
https://www.systems.ethz.ch/courses/fall2014/aos
What today is about

• The structure of an OS
  – How threads can be implemented
  – How the kernel is organized
  – How the rest of the OS is arranged

• Preparation for next week's assignment:
  – Self-paging, and handling upcalls
OS Architecture
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
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
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

\( \triangleright = \) 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?

- Word processor
- Audio driver
- Video driver
- Audio server
- Network stack
- Disk driver
- Network driver
- Video player
- Window system
- Filing system

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

- Video player
  - Threads package
  - Window System
  - Filing system
  - Network stack
  - Disk driver
  - Video driver
  - Audio driver
  - Network driver

- Word processor
  - Coroutine package
  - Window System
  - Filing system
  - Network stack
  - Disk driver
  - Video driver
  - Network driver

- Compiler
  - Filing system
  - Network stack
  - Disk driver
  - Network driver

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 Barrelish 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
Kernel thread models: How the kernel provides threads
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, Barrelish
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.
User thread models:
Building user threads over kernel threads
The problem

- Threads are a programming language abstraction
  - (different, possibly parallel activities)
    ⇒ should be lightweight, in the language runtime

- Threads are a kernel abstraction
  - (virtual or physical processors)
  - way to manage “big” resources like CPUs

- Threads need to communicate (either within or between address spaces)
  - Thread and IPC performance critical to applications
What are the options?

1. Implement multiple threads over a single “virtual CPU” (kernel thread)
   – Perhaps many VCPUs per process
2. One user-level thread per “virtual CPU”
   – Multiple kernel threads in a process
3. Some combination of the above
   – Multiplex user threads over kernel threads
1. Many-to-one threads
1. Many-to-one threads

• Early “thread libraries”
  – Green threads (original Java VM)
  – GNU Portable Threads
  – Standard student exercise: implement them!

• Sometimes called “pure user-level threads”
  – No kernel support required
  – Also (confusingly) “Lightweight Processes”
Address space layout for user level threads

Stack

BSS
Data
Text

Thread 1 stack

Thread 3 stack
Thread 2 stack

BSS
Data
Text

Just allocate on the heap
User-level threads
Older Unices, etc.

• High performance
  – 10x procedure call
• Scalable
• Flexible (application-specific)
• Treat a process as a virtual processor
• But it isn’t:
  – Page faults
  – I/O
  – Multiprocessors
2. One-to-one threads

User

Kernel

CPU 0

CPU 1
2. One-to-one user threads

• Every user thread is/has a kernel thread.
• Equivalent to:
  – multiple processes sharing an address space
  – Except that “process” now refers to a group of threads
• Most modern OS threads packages:
  – Linux, Solaris, Windows XP, MacOSX, etc.
One-to-one user threads
Kernel threads
Linux, Vista, L4, etc.

• Excellent integration with the OS
• Slow
  – similar to process switch time
• Inflexible
  – kernel policy
  – Evidence: people implemented user-level threads over kernel threads anyway
⇒ same old problems...
Comparison

User-level threads
✓ Cheap to create and destroy
✓ Fast to context switch
✗ Can block entire process
  – Not just on system calls
  – Page faults!

One-to-one threads
✓ Easier to schedule
✓ Nicely handles blocking
✗ Memory usage (kernel stack)
✗ Slow to switch
  – Requires kernel crossing
3. Many-to-many threads

![Diagram of many-to-many threads]

- User
- Kernel
- CPU 0
- CPU 1
2. Many-to-many threads

• Multiplex user-level threads over several kernel-level threads
• Can “pin” user thread to kernel thread for performance/predictability
• Thread migration costs are “interesting”...
Issues with modern threads

• Hard for a runtime to tell:
  – How many user-level threads can run at a time
    (i.e. how many physical cores are allocated)
  – Which user-level threads can run
    (i.e. which physical cores are allocated)

• Severely limits flexibility of user-level scheduler
  – Can critically impact performance of parallel applications.
Scheduler Activations and Dispatch
Definitions

• Scheduling:
  – Deciding which task to run

• Dispatch:
  – How the chosen task starts (or resumes) execution
Scheduler activations

• Basic mechanism: upcall to the ULS from the kernel
  – Context for this: a scheduler activation
• Structurally like a kernel thread but . . .
  – created on-demand in response to events
    (blocking, preemption, etc.)
• User level threads package built on top
• Hardware: DEC SRC Firefly workstation (7-
  processor VAX)
Scheduler activations speedup

Figure 1: Speedup of N-Body Application vs. Number of Processors, 100% of Memory Available
Scheduler activations memory footprint

![Graph showing execution time vs. available memory]

Figure 2: Execution Time of N-Body Application vs. Amount of Available Memory, 6 Processors
Psyche threads

- Similar: aims to remove kernel from most thread scheduling decisions, reflect kernel-level events to user space
- Kernel and ULS share data structures (read/write, read-only)
- Kernel upcalls ULS (“software interrupts”) in a virtual processor for:
  - Timer expiration
  - Imminent preemption (err...)
  - Start of blocking system call
  - Unblocking of a system call
- Shared data structure standardises interface for blocking/unblocking threads
Psyche data structures

Kernel data
read-only in user mode

User data
read-write in user mode

Virtual processor
- thread
- s/w interrupts disabled?
- s/w interrupts queued?
- preemption warning period
- preemption imminent?
- preemption interrupt desired?
- timers
- s/w interrupt stack
- ...

Thread
- scheduler routines
- thread id
- thread package data:
  - stack
  - saved registers
  - ...

Pseudo-registers
- virtual processor
- physical processor
- address space
- statistics
- ...

Address space
- s/w interrupt vectors
Interesting features of Psyche

• Threads given warning of imminent preemption
  – Is there a problem here?
• Upcalls can be nested (stack)
  – Likewise?
• Upcalls can be disabled or queued
• Lots of user space data structures to be pinned
• Unlike Scheduler Activations, doesn’t handle (e.g.) page faults
Nemesis dispatch
(and K42, Barrellfish, others...)

• Avoid nested upcalls
  – Activation handler doesn’t need to be reentrant

• Per-domain data structures
  – all user read/write!
Deschedule / Preemption

- If resume bit == 0:
  - Processor state ← activation slot
- Else:
  - Processor state ← resume slot
- Enter the scheduler
Dispatch / Reschedule

- If resume bit == 0:
  - resume bit ← 1
  - Jump to activation addr on activation stack (small!)
- Else:
  - Processor state ← resume slot
- c.f. disabling interrupts
User-level schedulers in Nemesis

- Upcall handler gets activations on reschedule
- Resume always set on activation
  \[\Rightarrow\] no need for reentrant ULS
- Picks a context slot to run from
  - Slots are a cache for thread contexts
- Clears resume bit and resumes context
  - Implementation: Alpha PALmode call
    - (2 pipeline drains)
      - Must be atomic (or must it?)
- All implemented in user-level library
Dispatch in Barrelfish

- Activations: separate “dispatcher” per process per core
  - Avoid Psyche-like complexity
- No activation stack: disable mechanism (à la Nemesis)
- Multiple upcall entries (from K42):
  - Preemption/reschedule
  - Page fault
  - Exception
  - etc.
- User-level thread schedulers span address spaces across cores (dispatchers)
Summary of dispatch

• Plenty of ways to deliver processor to an application
• Expose underlying scheduler decisions
  ⇒ give more control to user-level thread scheduler
• On uniprocessor (e.g. Nemesis) gives flexibility
• On multiprocessor (e.g. Psyche) gives performance across cores