Threads, Dispatch, and IPC

Advanced Operating Systems
(263-3800-00)

Timothy Roscoe

Thursday 4 November 2010

Outline

Assumptions

- You already know:
  - what a semaphore is.
  - what pipes and sockets are
  - threads and processes are in Unix, Windows, Oberon, etc.
  - how IPC is used in Unix or Windows
  - material in previous lectures this course :-)

Definitions

Scheduling: deciding which task to run (later this semester)

Dispatch: how the chosen task starts (or resumes) execution

Event: a notification to a task that something happened

Transport: conveying (non-trivial) data to a task

IPC: general inter-process communication

- usually combines dispatch, notification, transport

These are hard, if not impossible, to separate entirely.

The problem

- Threads are a programming language abstraction (different, possibly parallel activities)
  $\Rightarrow$ should be lightweight, in the language runtime
- Threads are a kernel abstraction (virtual or physical processors)
  $\Rightarrow$ way to manage "big" resources like CPUs
- Threads need to communicate (either within or between address spaces)
- Thread and IPC performance critical to applications

User-level threads

Older Unices, etc.:

- High performance (10x procedure call)
- Scalable
- Flexible (application-specific)
- Built over kernel-level processes
- Treat a process as a virtual processor
- But it isn't:
  - Page faults
  - I/O
  - Multiprocessors

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...

Scheduler activations

[Anderson et al., 1991]

- Kernel allocates (multiple) processors to address spaces
- Address space's ULS allocates threads to processors
- Kernel notifies address space when:
  - number of allocated processors changes
  - a user-level thread blocks or wakes up in the kernel
- Address space notifies kernel of requests for more or fewer processors
  - ...but not of most thread scheduling decisions
- Application programmer sees no difference in AP (except faster!)

Scheduler activations

[Anderson et al., 1991]

- Basic mechanism: **upcall** to the ULS from the kernel
- Context for this: a *scheduler activation*
- Structural 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

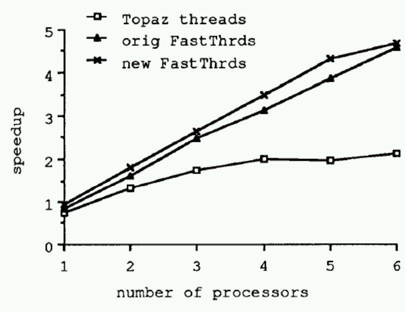


Figure 1: Speedup of N-Body Application vs. Number of Processors, 100% of Memory Available

Scheduler activations memory footprint

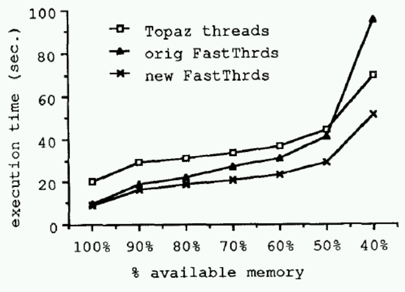


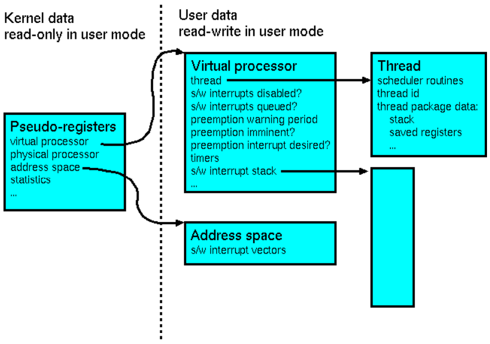
Figure 2: Execution Time of N-Body Application vs. Amount of Available Memory, 6 Processors

Psyche threads

[Marsh et al., 1991]

- 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 standardizes interface for blocking/unblocking threads

Psyche data structures



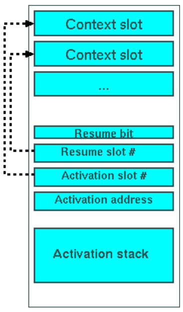
Interesting features of Psyche

- 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

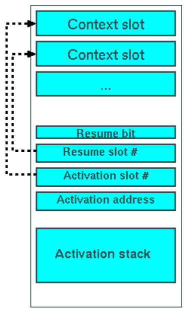
[Leslie et al., 1996]

- Present processor usefully to domain
- Minimize kernel policy & implementation
- Facilitate flexible user-level threads
- Per-domain data structures all user read/write!
- Almost identical in K42 [Appavoo et al., 2002], Barrelfish, ...



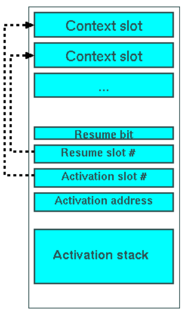
Deschedule/preemption

- Present processor usefully to domain
- Minimize kernel policy & implementation
- Facilitate flexible user-level threads
- Per-domain data structures all user read/write!
- Almost identical in K42 [Appavoo et al., 2002], Barrelfish, ...



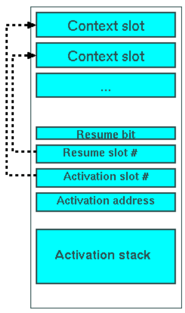
Deschedule/preemption

- 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
    ⇒ 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 (á la Nemesis)
- Multiple upcall entries (from K42):
  - Preemption / reschedule
  - Page fault
  - Exception
  - etc.
- User-level thread schedulers span address spaces

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

Lots of IPC mechanisms in Unix

- Pipes
- Signals
- Unix-domain sockets
- POSIX semaphores
- FIFOs (named pipes)
- Shared memory segments
- System V semaphore sets
- POSIX message queues
- System V message queues
- etc.

IPC is usually heavyweight

- IPC mechanisms in conventional systems tend to combine:
  - **Notification**: (telling the destination process that something has happened)
  - **Scheduling**: (changing the current runnable status of the destination, or source)
  - **Data transfer**: (actually conveying a message payload)
- Unix doesn't have a *lightweight IPC mechanism*

IPC is usually polled

- IPC mechanisms in Unix are generally polled:
  - Blocking `read()`/`recv()` or `select()`/`poll()`
- Signals are the nearest thing to upcalls, but...
  - Dedicated (small) stack
  - Limited number of syscalls available (e.g. semaphores)
  - Calling out with `longjmp()` problematic, to say the least
- Unix lacks a good upcall / event delivery mechanism

The problem

- How to perform has cross-domain invocations?
- Does the calling domain/process block?
- Is the scheduler involved?
- Is more than one thread involved?
- What happens across physical processors?

Lightweight RPC (LRPC)

[Bershad et al., 1990]

- Basic concepts:
  - Simple control transfer: client's thread executes in server's domain
  - Simple data transfer: shared argument stack, plus registers
  - Simple stubs: i.e. highly optimized marshalling
  - Design for concurrency: Avoids shared data structures

High overhead of previous cross-domain RPC:

- Stubs copy lots of data (not an issue for the network)
- Message buffers usually copied through the kernel (4 copies!)
- Access validation
- Message transfer (queueing/dequeuing of messages)
- Scheduling: programmer sees thread crossing domains, system actually rendezvous's two threads in different domains
- Context switch (x 2)
- Dispatch: find a receiver thread to interpret message, and either dispatch another thread, or leave another one waiting for more messages

Most messages are short

[Bershad et al., 1990]

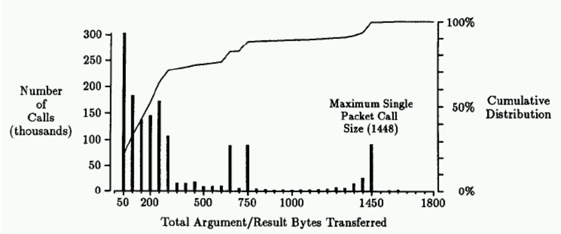


Fig. 1. RPC size distribution.

LRPC Binding

Binding: connection setup phase:

- *Procedure Descriptors* (PDs) registered with kernel for each procedure in the called interface
- For each PD, *argument stacks* (A-stacks) are preallocated and mapped read/write in both domains
- Kernel preallocates linkage records for return from A-stacks
- Returns A-stack list to client as (unforgeable) Binding Object

Calling Sequence

(all on client thread)

1. verify Binding Object, find correct PD
2. verify A-Stack, find corresponding linkage
3. ensure no other thread using that A-stack/linkage pair
4. put caller's return addr and stack pointer in linkage
5. pushes linkage on to thread control block's stack (for nested calls)
6. find an execution stack (E-stack) in server's domain
7. update thread's SP to run off E-stack
8. perform address space switch to server domain
9. upcall server's stub at address given in PD

LRPC Discussion

- Main kernel housekeeping task is allocating A-stacks and E-stacks
- Shared A-stacks reduce copying of data while still safe
- Stubs incorporated other optimizations (see paper)
- Address space switch is most of the overhead (no TLB tags)
- For multiprocessors:
  - Check for processor idling on server domain
  - If so, swap calling and idling threads
  - (note: thread migration was very cheap on the Firefly!)
  - Same trick applies on return path

L4 synchronous RPC

- L4 pushed this idea further (for uniprocessor case)
- No kernel-allocated A-stack: server must have waiting thread (no upcalls possible)
- RPC just exchanges register contents with calling thread
- *Synchronous RPC: calling thread blocks, waits for reply*
- Scheduler bypassed completely

- The infamous "null RPC" microbenchmark
  - Latency of a single call, nothing else happening
- Design couples notification, transfer, scheduling.

Local RPC on Barrelfish

On a single core:

- IPC is asynchronous: one-way messaging only
  - RPC implemented at higher level in stubs
- Message is queued at destination, may cause an upcall
- L4-style fast path: thread can optionally wait for a message

- Unlike L4, can decouple notification & transfer
- Scheduler is always involved (but ...)
- More interesting techniques occur *between cores*

Other "interesting" kernel IPC mechanisms

- "Doors" in Spring from Sun Labs (threads and shuttles)
  - "The Spring Nucleus: A Microkernel for Objects", Graham Hamilton and Panos Kougiouris, Sun Microsystems Labs Technical Report TR-93-14, April 1993.
  - (middle ground between thread migration and thread transfer)
- The Synthesis kernel by Henry Massalin
  - "A Lock-Free Multiprocessor OS Kernel", Henry Massalin and Calton Pu, Technical Report CUCS-005-91, Computer Science Department, Columbia University, June 1991.
  - (on-the-fly mc68k machine code generation for, well, nearly everything).

User-level RPC (URPC)

[Bershad et al., 1991] - sound familiar?

- URPC is to Scheduler Activations what LRPC was to kernel threads
  - Kernel is *not* involved!
  - Unnecessary processor reallocation eliminated
  - Necessary rocessor reallocation amortized over several independent calls
  - Exploit inherent parallelism in message send/recv
- Key idea: decouple
  - notification (user-space)
  - scheduling (kernel)
  - data transfer (also user space).

Avoiding the kernel

- Use shared memory channels, mapped pairwise between domains
  - Queues with non-spinning test-and-set locks at each end
- Integrate with user-level thread management
  - Threads can block on channels without kernel involved.
- Messaging is fully *asynchronous* (below the thread abstraction)
- Domain can thread-switch rather than block on another address space
- Multiprocessor $\Rightarrow$ wins big if client and server threads run concurrently

URPC performance

All on a Firefly (4-processor CVAX):

- URPC cross-AS latency: 93μs
- URPC inter-processor overhead: 53μs
- LRPC latency: 157μs

- URPC thread fork: 43μs
- LRPC thread fork: > 1000μs

- Procedure call: 7μs
- Kernel trap: 20μs

Irony: LRPC, L4 seek performance by optimizing kernel path, URPC gains performance by bypassing kernel entirely

Barrelfish CC-UMP

- User-space message-passing based on URPC
- Performance:
  - Send a cache line in only 2 cache transactions
  - ~ 600 cycles on a modern machine
  - Still faster than same-core L4-style
- Relies heavily on second-guessing cache coherence
  - See later in the course for detailed discussion

References

- Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. (1991). **Scheduler activations: effective kernel support for the user-level management of parallelism**. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles, pages 95–109, New* York, NY, USA. ACM.
- Appavoo, J., Auslander, M., DaSilva, D., Edelsohn, D., Krieger, O., Ostrowski, M., Rosenburg, B.,Wisniewski, R.W., and Xenidis, J. (2002). **Scheduling in K42**. http://domino.research.ibm.com/comm/research_projects.nsf/pages/k42.index.html.
- Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. (1990). **Lightweight remote procedure call**. *ACM Trans. Comput. Syst., (1):37–55.*
- Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. (1991). **User-level interprocess communication for shared memory multiprocessors**. *ACM Trans. Comput. Syst., 9(2):175–198.*
- Leslie, I. M., McAuley, D., Black, R., Roscoe, T., Barham, P., Evers, D., Fairbairns, R., and Hyden, E. (1996). **The Design and Implementation of an Operating System to Support Distributed Multimedia Applications**. *IEEE Journal on Selected Areas in Communications, 14(7):1280–129.*
- Marsh, B. D., Scott, M. L., LeBlanc, T. J., and Markatos, E. P. (1991). **First-class user-level threads**. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles, pages 110–121, New* York, NY, USA. ACM.