```
6.828 2014 Lecture 20: OSes and networking
==

Assigned reading: [IX: A Protected Dataplane Operating System for High Throughput and Low
Latency](https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay)


Plan:
 Lab 6
 Commodity OS networking
 IX

Lab 6
--
  [ see design picture in lab handout ]
  No interrupts
  Many IPCs
  Much copying of network packet content
  Any parallelism?
  Many kernel/User transitions?
  Where are packet queues?
  Many scheduling decisions?
  Could you have long delays to process a packet?
  Does this design achieve high throughput?

Commodity OS (simplified)
--
  [ draw picture ]
  NIC generates interrupt
  Interrupt handler runs in top-half
        runs in interrupt context
        performs miminimal work
  Kernel schedules Bottom-half runs
        performs TCP processing
        copies data into sockets
    may send some packets from outgoing socket
  Kernel schedules user-level process
  User-level process reads data from socket (system call)
  User-level process writes data to socket (system call)
    TCP processing as a side-effect of write
        Write to NIC

IX Paper
--
 Very recent paper (published two months ago)
   Builds on Dune paper from last week
   IX is another example use of Dune
 Brings a lot of 6.828 topics together
   Isolation
   Exokernel architecture
   Multicore scalability
   Virtualization
   Networking

IX
---

Goal: high performance
  High packets rates for short messages
    Data center apps involve many servers
  Low latency, predictable
    In data centers some messages are short
  Setting up/closing connection fast
```

```
Current OSes: fine-grained resource scheduling
  Multiple applications share a single core
  System call and interrupt latency > packet interarrival time
  --> interrupt coaeslecting, queing delay, intermediate buffering, CPU scheduling
  --> adds latency to packet processing
  --> buffering and synchronization increase memory and CPU overhead, reduce throughput
  API introduces sharing

Idea: separate control plane from data plane
  Control plane: responsible for coarse-grained resource allocation
  Data plane: network stack and app logic
  Use Dune for isolation of control plane, data plane, and app

IX Control plane
  Schedules resources among data planes
  CPU core is dedicated to data plane
  Memory in large pages are allocate to data plane
  NIC queues are assigned to dataplane cores
  Data plane has direct access to NIC queue

Data plane: a libos specialized for networking
  Run to completion with adaptive batching
    Allow use of polling, instead of interrupts
  Adaptive batching every stage of networking processing
    system call boundary, network API, NIC queue
        Only batch under high load
        Why a maximum batch size?
  Zero-copy API
    on receive: packets are mapped read-only into application
        on send: scatter/gather memory list
        flow control
  No interaction between cores
    RRS flow groups are assigned to a dedicated core, which does all processing
    Each core has its own memory pool
        API doesn't incur sharing
  Elastic threads dedicated to core
    Run networking processing and app processing code
    Time limit on app processign code
  Large pages
    Why is this important?

Isolation between data plane and control
  User-level stacks
    No protection between stack and application
    Stack may interfere with other stacks
  IX design
    Linux kernel (with Dune module) runs in root, ring 0
    IX runs as a library OS in non-root, ring 0
    Application (with libix) in non-root, ring 3
  Implementation: Dune
  Does app have direct access to NIC?
  Can app write pages after handing it to IX?
  Can app write messages buffers received from NIC?
  Who provides the *physical* addresses to NIC?
          Is this an isolation problem?

Libix: network API (see figure 1b and table 1)
  low-level API:
    load network commands into a shared buffer
    call run_io()
    ix process shared buffer
    posts results in shared buffer of event contitions
        poll NIC
  libevent
```

```
Aside: commutativity rule for multicore scalability
  Recall RadixVM paper:
    if map/unmap are on different regions, a scalable implementation exists
  General form:
    if two operations commute, then a scalable implementation must exist
  Example:
    map and unmap on different regions commute
        thus a scalable implementation exists (e.g., RadixVM)
  IX uses this rule for the design of its interface to ensure scalability
    no shared name space of file descriptors
        cookies instead of fds in table 1

IX Performance
  Comparing Linux, mTCP, and IX
  What is goodput?
  Is 5.7usec good?
    What is the limiting factor?
    Impact of non-root ring 0 and 3?
  Why much faster than Linux?
  Why 99% metric?
```