POCS Quiz 1

12.09.2025

Name : *John Q. Sample*

Question:

Suppose you are tasked with designing an exokernel called ExoNet, whose purpose is to support a modern network interface card (NIC). Your task is to sketch the design of ExoNet, following the principles of an exokernel. **Specifically, answer the following questions**:

1. Enumerate the exokernel principles that you apply in ExoNet.
2. For each principle, discuss your design of ExoNet that realizes it, and briefly justify the correctness of your design and how your design follows the principle.

(Write your answer inside the box on the next page. Anything outside that box will be ignored.)

Background:

A NIC provides software with rings, which are arrays in memory that hold task descriptors. Each descriptor tells the NIC what packet to send. A ring is managed with two indices (indices wrap around to zero after reaching the ring length):

- Tail: advanced by the producer (software) when it enqueues new descriptors
- Head: advanced by the consumer (the NIC) when a send is complete.

The typical workflow is:

1. Software places a task descriptor into a ring and advances the tail.
2. The NIC reads the descriptor and transmits the packet.
3. Once transmission finishes, the NIC updates the head value in memory, so the software knows the send is done.

Assume the NIC can support up to 64 rings, meaning it can process up to 64 independent streams of task descriptors in parallel. Using all 64 rings allows the NIC to reach its peak performance. Of course, the system may be running many more than 64 applications at the same time that want to use the NIC. To keep things simple, we will focus only on transmission (sending packets) and ignore packet reception.

For this question, you can ignore issues such as address translation or memory access control for the NIC (which would normally require an IOMMU). You may assume that the rings are just regions of host memory that the NIC can access. You may also assume ExoNet already implements all the mechanisms described in Aegis, and ExoNet can use them directly.

In ExoNet, each hardware TX ring in the NIC is a first-class resource made up of its descriptor array and head/tail indices shared between software and the NIC. A capability per ring authorizes a user-space process to map that ring's memory, write descriptors to it, and advance its tail. The NIC DMAs packets from process memory and updates the ring head. ExoNet never interprets or copies descriptors.

Three exokernel ideas inform ExoNet's design: secure multiplexing, secure binding, and visible revocation.

Secure multiplexing & binding: On allocation/bind, ExoNet mints a capability and maps the ring pages into the caller's address space; the caller (producer of packets) gets RW rights to the descriptors and tail, RO rights to the head. Once authorization completes, subsequent use is direct, without exokernel mediation: the producer publishes a packet by writing the descriptors and then updating the tail. The NIC transmits packets and advances the head accordingly. ExoNet provides the mechanism and enforces isolation through memory protection (via the cap table and software-managed TLB). ExoNet can optionally expose per-ring stats RO to everyone, to inform user-space policy.

Visible revocation: To avoid underutilization, ExoNet enables dynamic, fast, atomic transfer of ring ownership from one process to another via a `swap_owner(resource, other_proc)` *syscall*[1]*. When a ring is requested, but none is free, ExoNet issues a revoke request to all owners, with a deadline. The owners decide who should give up rings (this constitutes policy, so not ExoNet's business), and one of them calls* `swap_owner`*. If the deadline is not met, ExoNet randomly chooses a ring to transfer to the requestor. Apps decide how to use rings: a dedicated ring per flow, time-shared across connections, etc.*

With this same mechanism, it is possible to run in user space either a central broker or per-core muxes that time-share/multiplex many apps over fewer rings, while keeping scheduling/admission policy outside the exokernel[2]*.*

The footnotes below are just for your information, we did not expect that level of detail.

[1] We could avoid adding a new syscall to ExoNet by employing a user-space wrapper over generic exokernel operations, but it's simpler to have a single syscall, especially for avoiding races. This does not violate the exokernel design principles.

[2] This would be essential if, for instance, more than 64 apps wanted to use the NIC simultaneously. One design is to delegate ring management to a user-space multiplexer per core, as part of the libOS. The ownership swaps occur among muxes instead of apps. The apps interact (via the libOS) with the muxes, which in turn negotiate with each other how to implement the policies they want (e.g., fair sharing). ExoNet only supplies the capabilities, mapping, revocation, and rebind primitives.

Grading Rubric

Total (10 pts)

- (2 pts) Secure multiplexing
  - (1 pt) Lays out clearly the hardware resource(s) that ExoNet multiplexes and explains how
  - (1 pt) Securely multiplexes hardware resources (rings) and leaves management policy to applications/libOSes
- (2 pts) Secure binding
  - (1 pt) Uses secure bindings to decouple authorization from actual ring use.
  - (1 pt) Leverages exokernel mechanisms to allocate rings and transfer them between processes. Access checks are enforced via the TLB.
- (2 pts) Visible resource revocation and abort protocol
  - (1 pt) ExoNet does not directly revoke rings when resources are scarce, but uses the revocation protocol described in the paper. Applications choose which resources to relinquish.
  - (1 pt) Provides an abort path for when applications are not cooperative
- (1 pt) Overall design always ensures the separation of policy and mechanism
- (2 pts) Scale-up
  - (1 pt) Design can accommodate more than 64 applications using the NIC simultaneously
  - (1 pt) Design provides good overall performance for such cases, as well as ensures proper isolation
- (1 pt) The writing is well structured and clear