

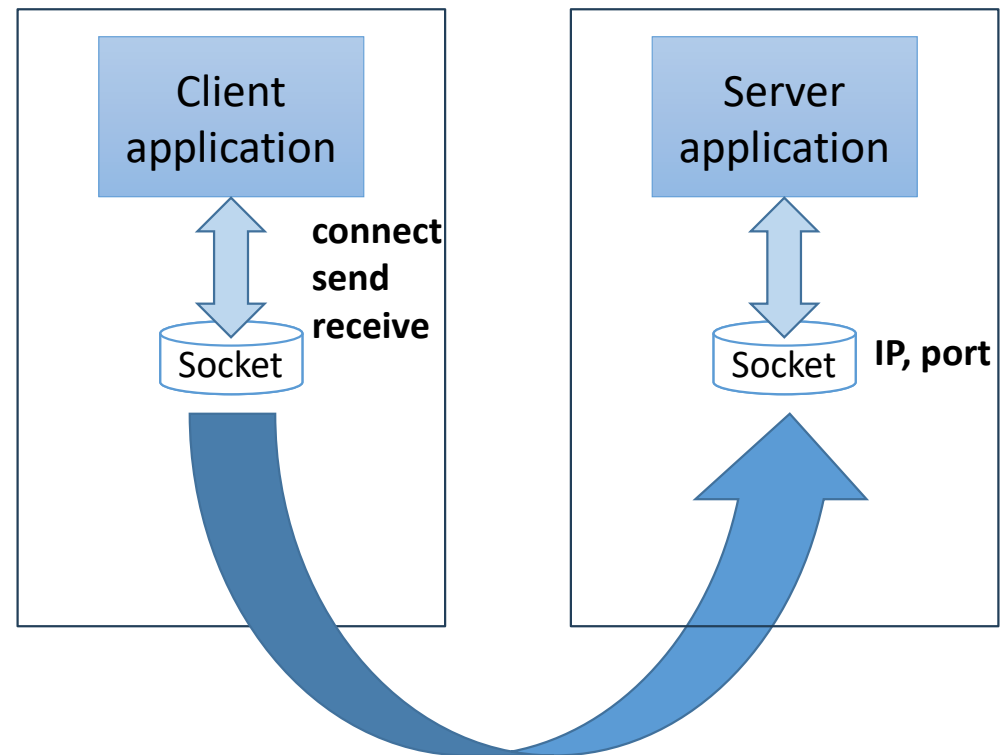
Network I/O subsystem in Linux

Mythili Vutukuru
CSE, IIT Bombay

(with help from Debojeet Das)

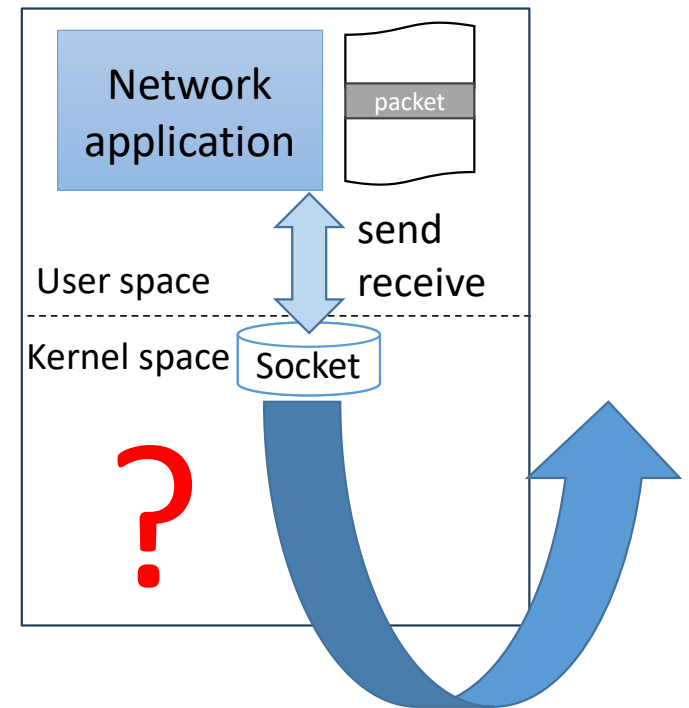
Networking applications

- Networking applications: web server, email client, browser etc..
- Exchange network packets via APIs like **sockets**
- Servers open sockets at well known IP address + port number
- Socket of web client **connects** to socket at web server, **send** and **receive** messages



What happens inside the kernel?

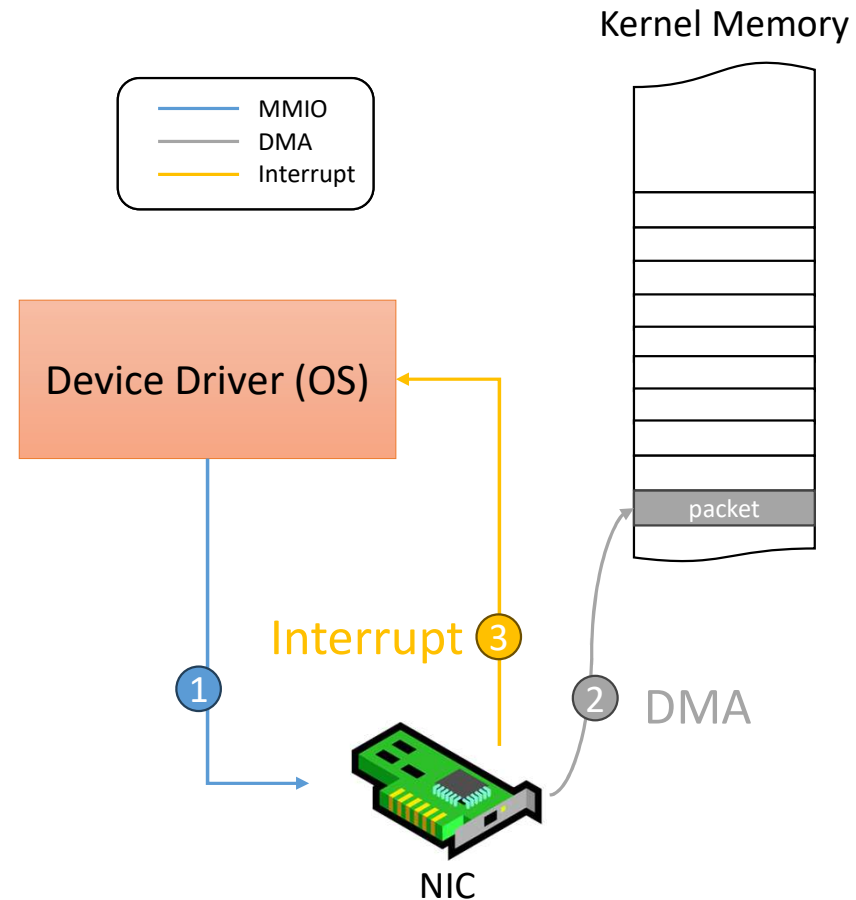
- What happens when you send and receive data through a socket?
- Packets are switched through the network from the sender host to receiver host.
- What happens at the sender and receiver **end host kernel network stack**?



Outside end host: switching, routing, congestion control

Device drivers

- Device driver manages interaction between NIC (network interface card) and software
- Configures NIC via memory mapped I/O (**MMIO**)
- NIC performs Direct Memory Access (**DMA**) of network packets into kernel memory
- NIC raises **interrupt** to indicate reception of packets
- We will discuss only RX path here



Interrupt handling

- How are interrupts handled?
 - CPU is running process P and interrupt arrives
 - CPU saves context of P, runs OS code to handle interrupt in kernel mode
 - Restore context of P, resume P in user mode
- Interrupt handling code is part of OS device drivers
- Network device drivers handle interrupts from NICs

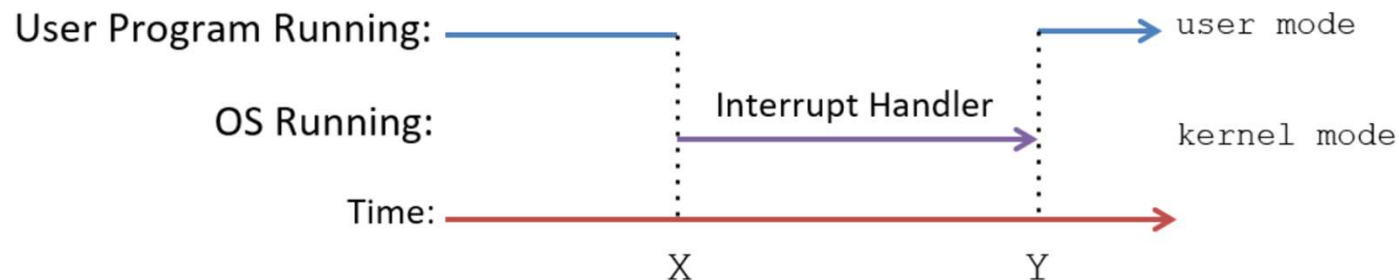


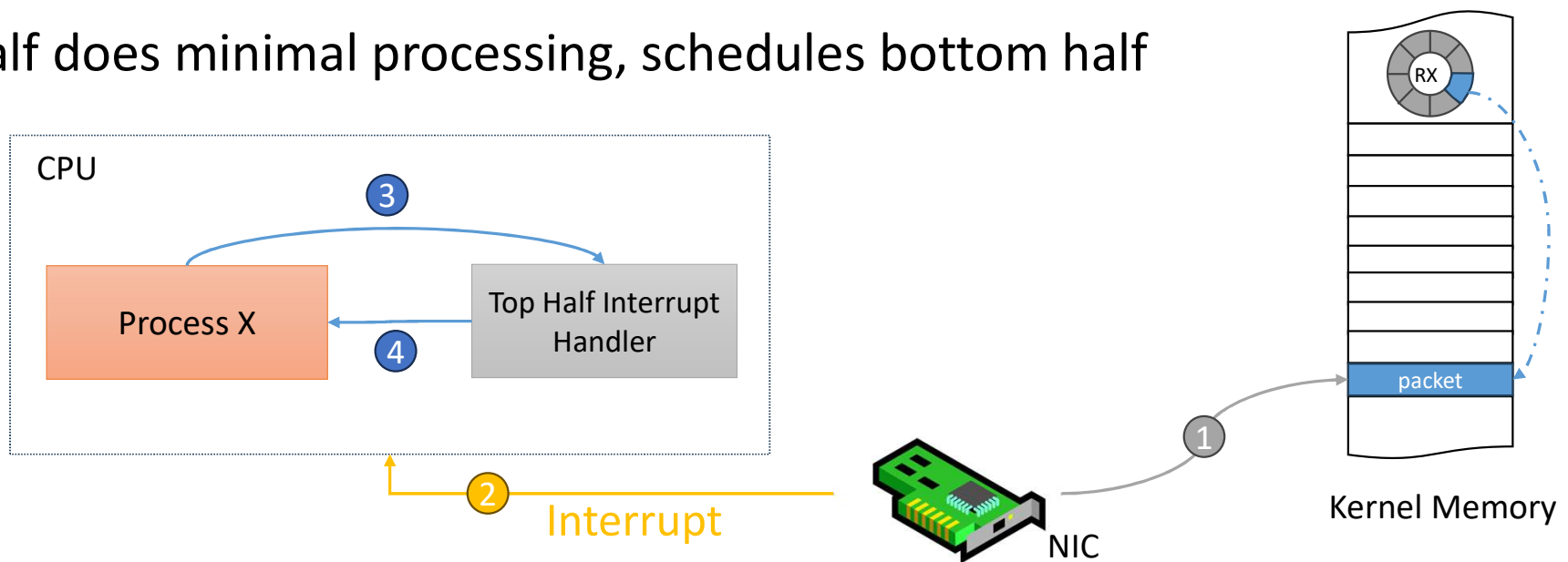
Image credit: Dive into Systems, by Mathews, Newhall, Webb

Network Interrupt handling

- Interrupt handling from NIC involves lot of work
 - Processing information about the network, congestion control, ...
- To avoid excessive disruption to interrupted process, NIC interrupt handling split into two parts
- **Top half** interrupt handler acknowledges interrupt, does minimal processing
- Top half schedules a kernel process for full interrupt handling, called **bottom half** interrupt handler
- Bottom half processes the packet and delivers the packet to userspace

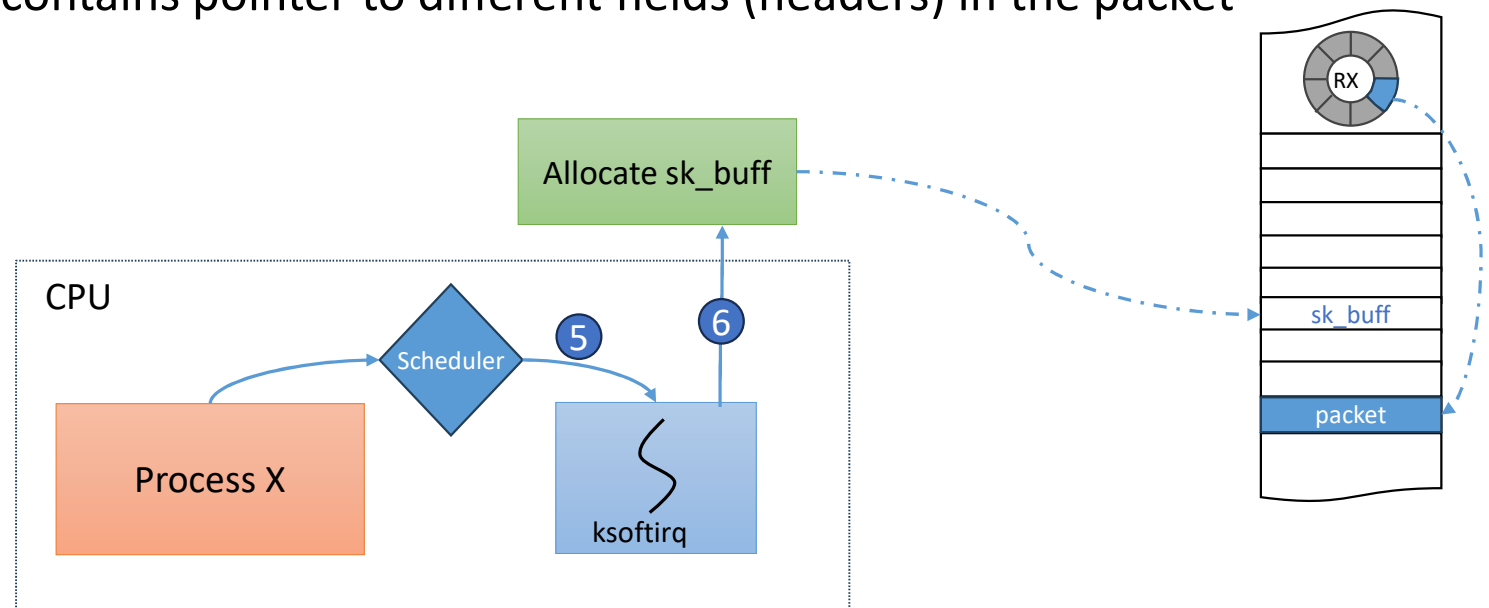
Top half interrupt handler

- NIC and kernel exchange information about packets via TX/RX “rings”
- RX ring: circular array containing pointers to received packets
- NIC does DMA, updates pointer in RX ring, interrupts
- Top half does minimal processing, schedules bottom half



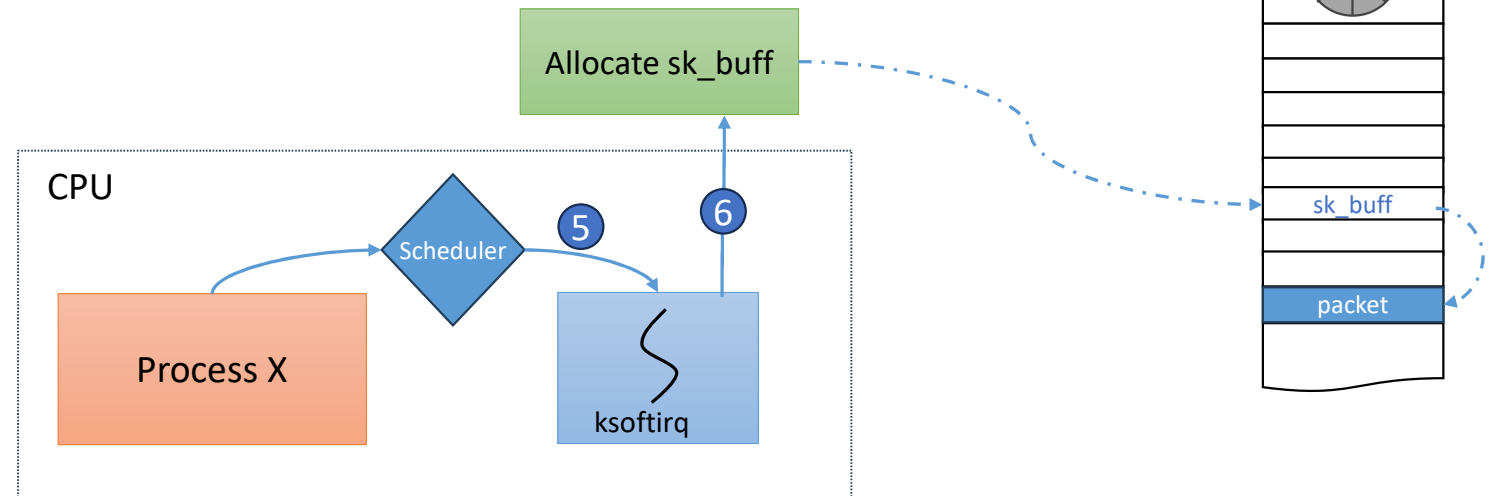
Bottom half interrupt handler

- Bottom half or ksoftirq process scheduled when CPU is free
- Processes all packets collected in the RX ring since the last round
 - Allocates socket buffer (sk_buff) structure for each packet
 - Socket buffer contains pointer to different fields (headers) in the packet



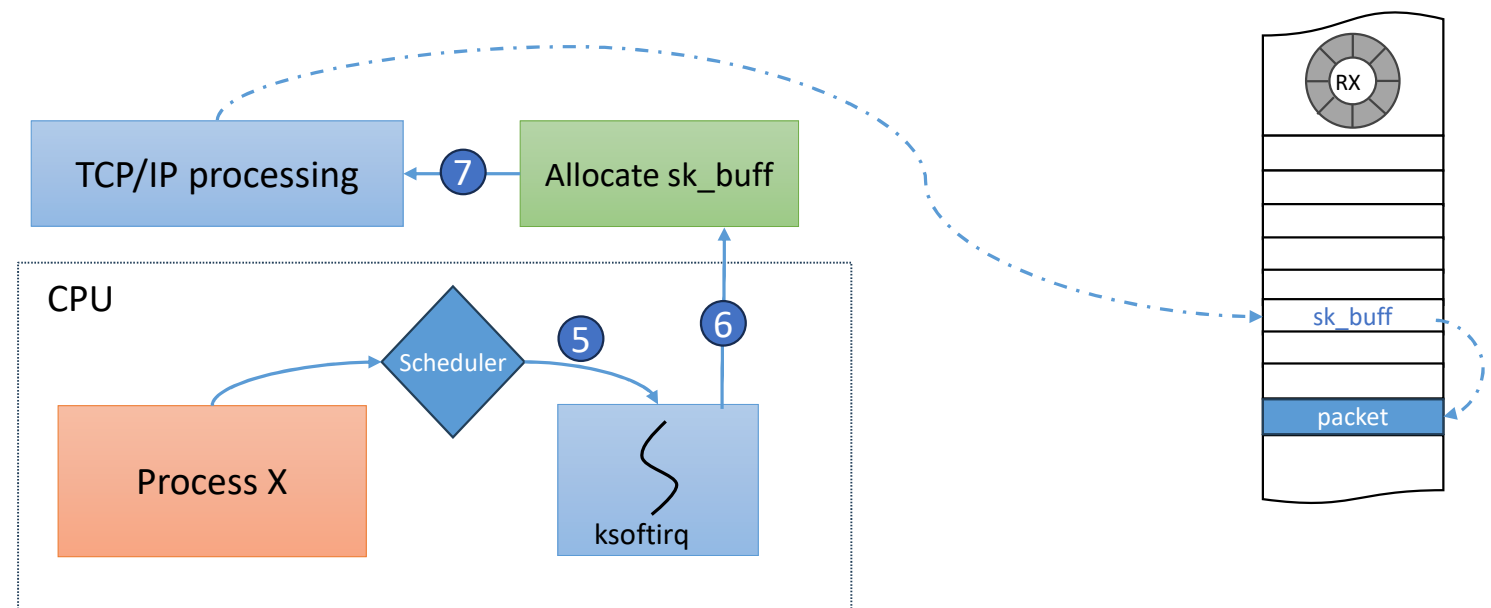
Bottom half interrupt handler

- Bottom half or ksoftirq process scheduled when CPU is free
- Processes all packets collected in the RX ring since the last round
 - Allocates socket buffer (sk_buff) structure for each packet
 - Socket buffer contains pointer to different fields (headers) in the packet
- Interrupt + ksoftirq can run on **multiple cores**



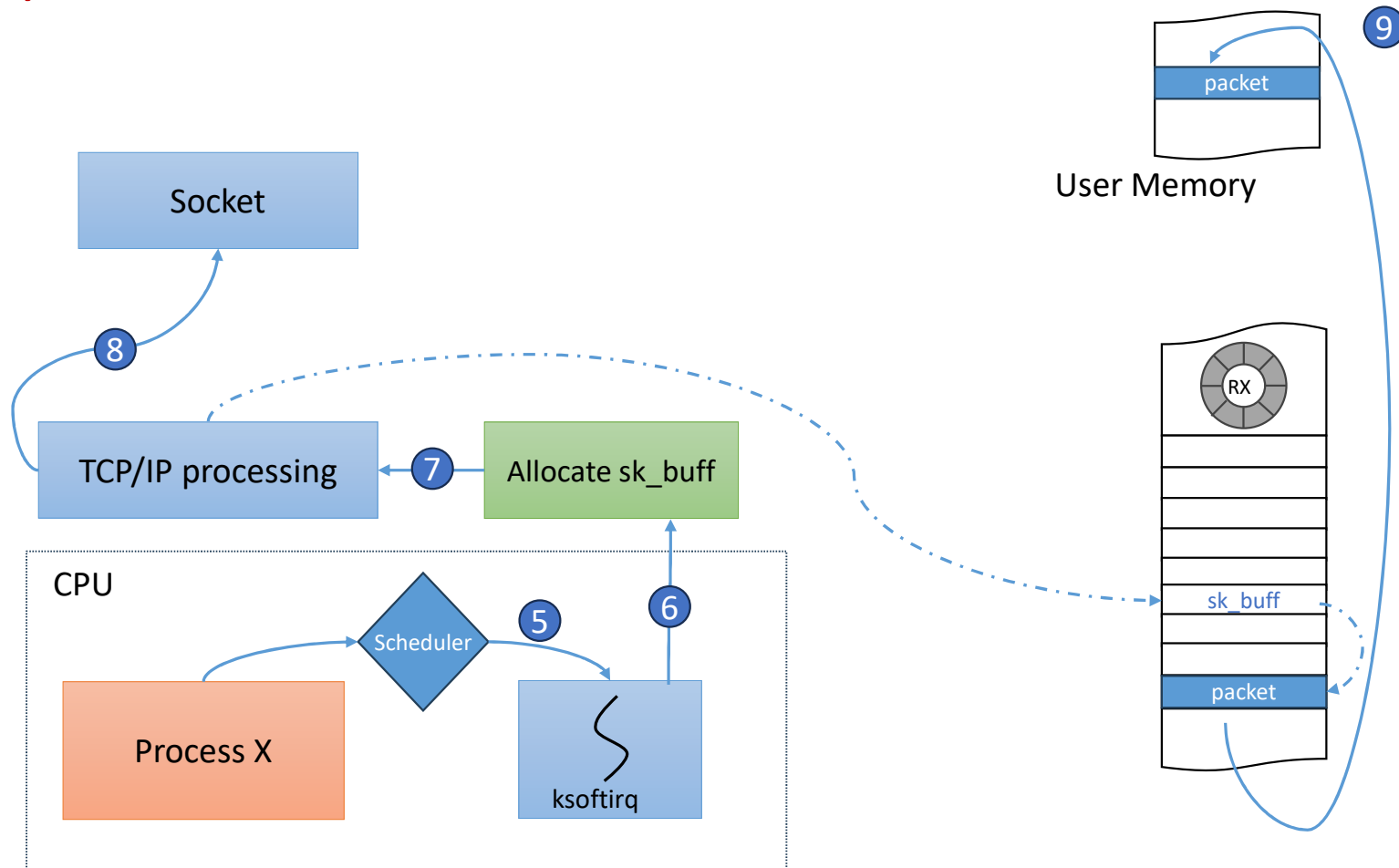
Network layer processing

- Bottom half interrupt handler performs all the network processing
 - Parsing and checking packet headers in sk_buff structure
 - IP routing, TCP reliability and congestion control algorithms



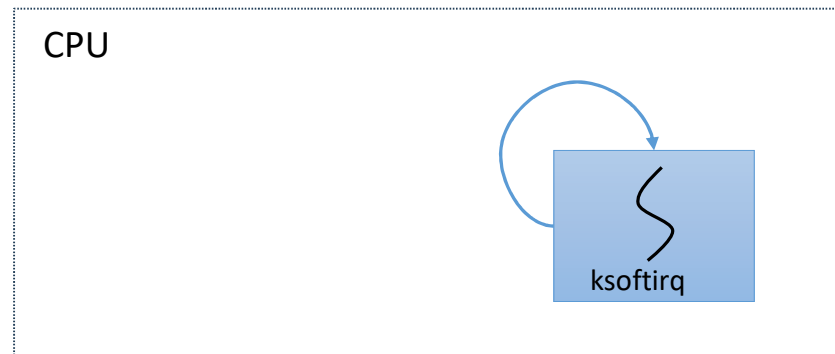
Packet copy to sockets

- Packet headers (port number) used to map received packet to socket
- On read from application, packet payload copied from kernel memory to user memory



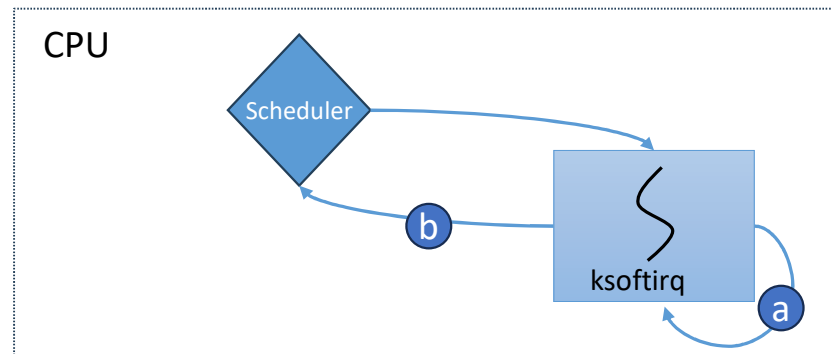
The problem of interrupts

- At high loads interrupts are generated continuously
- Each interrupt signals the same thing: there are packets to process
- Leads to excessive interrupt overhead and high CPU/system load
- Naïve solution: Run ksoftirq (bottom half) in busy loop – **busypolling** trades off CPU cycles for higher throughput and lower latency



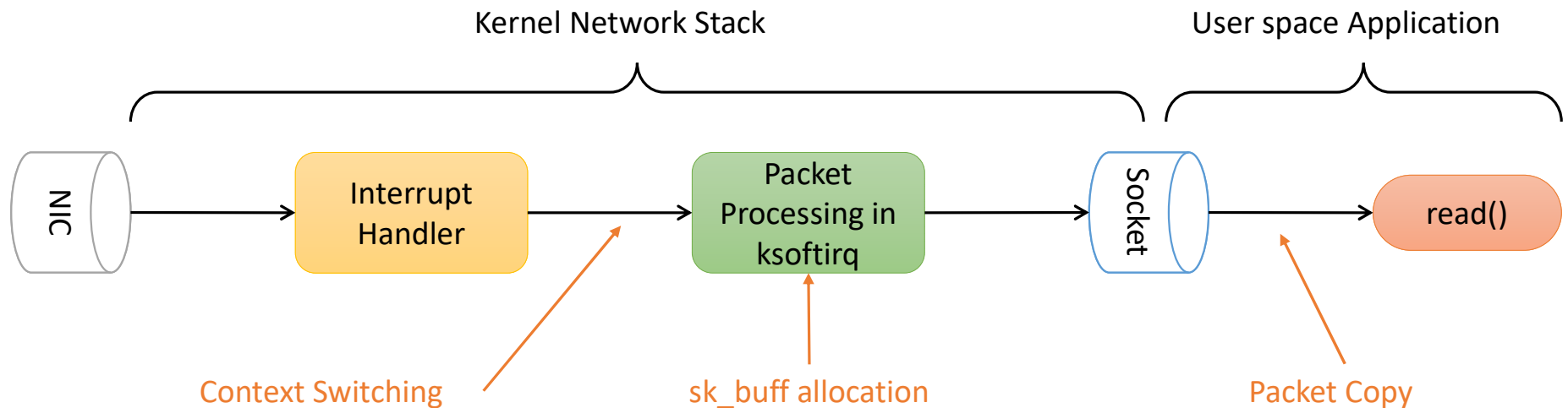
NAPI polling

- **Top half** interrupt handler acknowledges interrupt, schedules the **bottom half** and disables further interrupts
- The bottom half interrupt handler is scheduled with a **budget** – no of packets it can receive at a time
- If budget gets exhausted the bottom half is executed again, otherwise polling stops and interrupts are enabled again



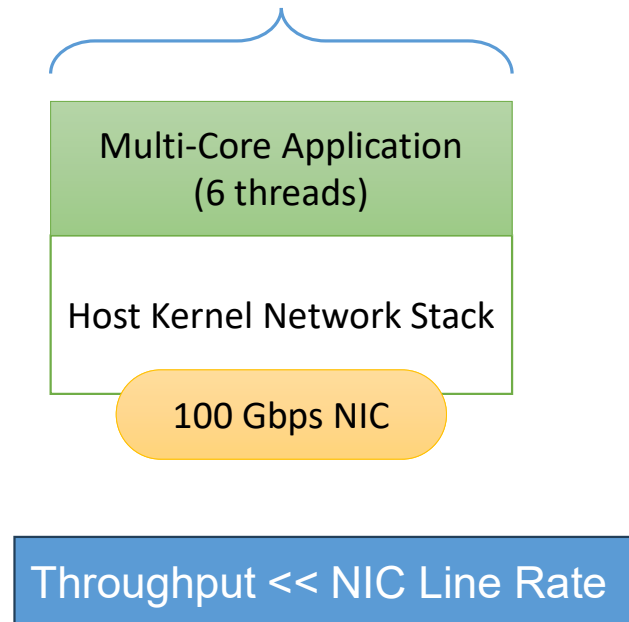
Overheads of the Linux network stack

- Interrupt handling, transition across user and kernel mode
- Context switching from application to ksoftirq
- Packet copy from kernel to user space



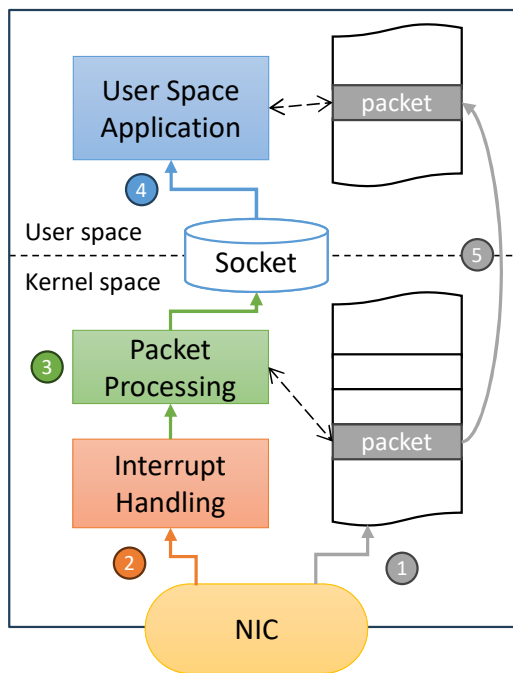
Need for alternate fast network I/O techniques

Max throughput possible is ~15 Gbps [1]

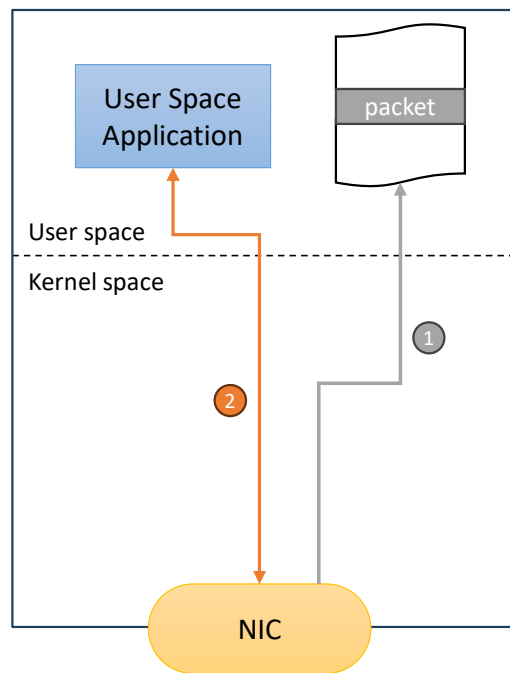


- Multi-threaded applications cannot easily achieve line rate in modern high-speed NICs, especially with small-sized packets
- Techniques to improve processing speed include kernel bypass techniques (directly DMA packets into user space) and using polling-mode device drivers
- Possible to process 100s of Gbps easily in software using such techniques

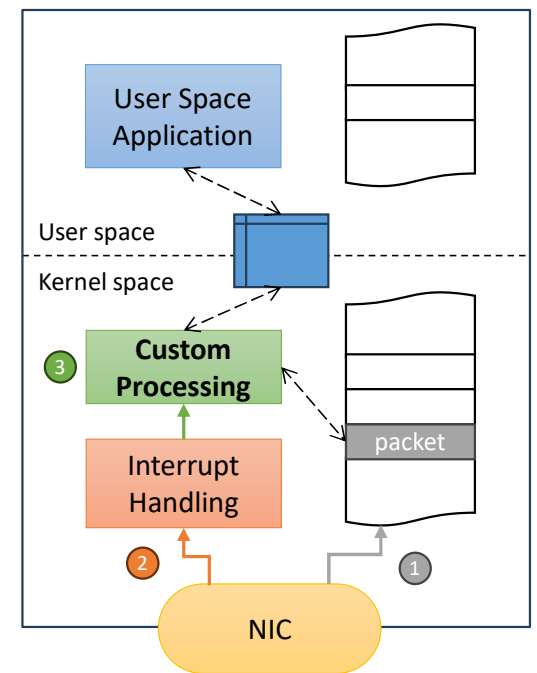
Fast I/O techniques



Generic Kernel Network Stack
High packet processing overheads



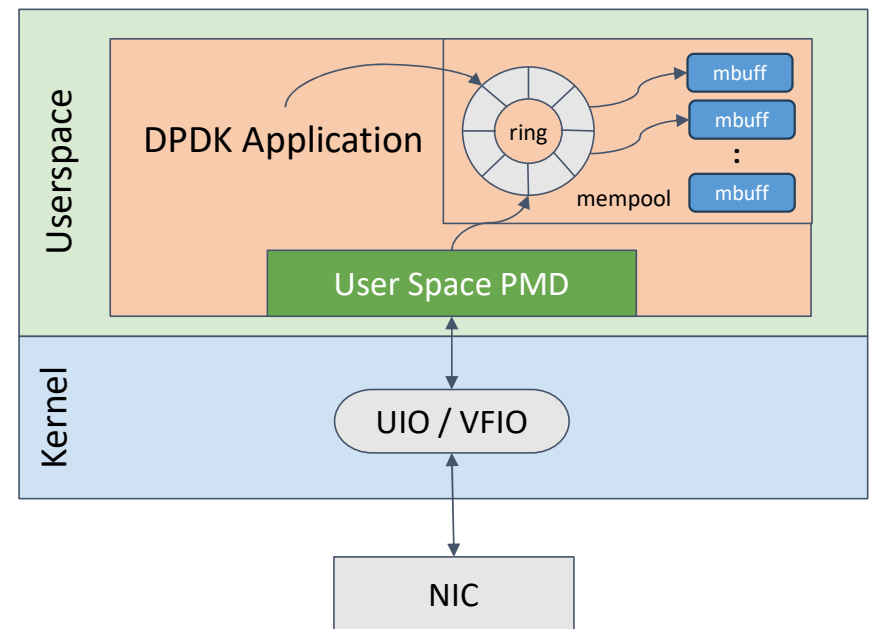
Kernel Bypass (DPDK)
Get packets directly into user space application



In-Kernel Program Offload (eBPF)
Push some extra application functionality into the kernel

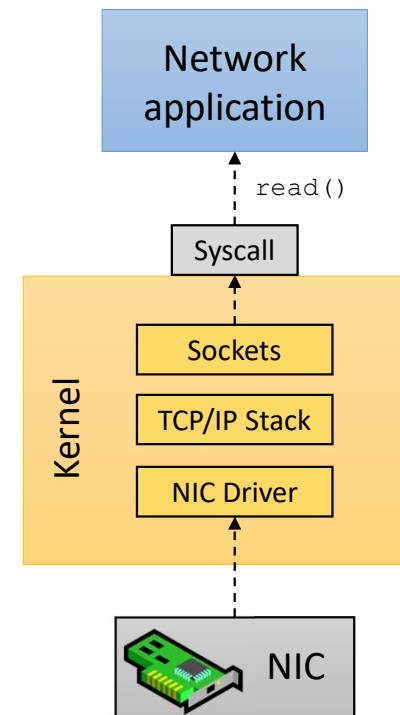
Kernel bypass with DPDK

- Another widely used kernel bypass mechanism
- Poll mode driver in userspace, kernel driver is just passthrough
 - Polls device, fetches and processes batches of packets
 - Packet buffers in user space pre-allocated buffers, huge pages
- Pros: Minimal involvement of kernel, high packet processing rates
- Cons: kernel tools don't work anymore, hard to co-exist with other apps



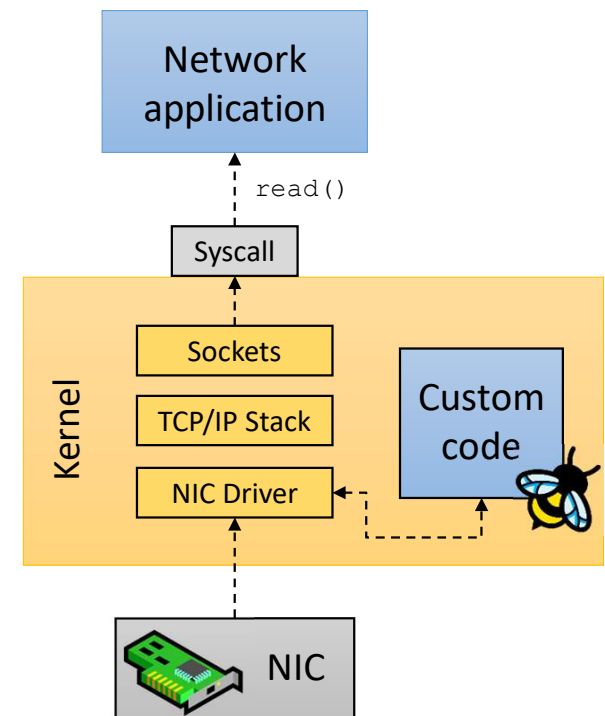
In-kernel packet processing with eBPF

- eBPF (extended Berkeley Packet Filter) is a way to embed custom packet processing code in the kernel safely at specific hook points
- Let's take example of XDP hook present in NIC driver to understand its working
- Runs custom code inside NIC driver for each packet when napi polling is done
- No `sk_buff` and copies required for XDP



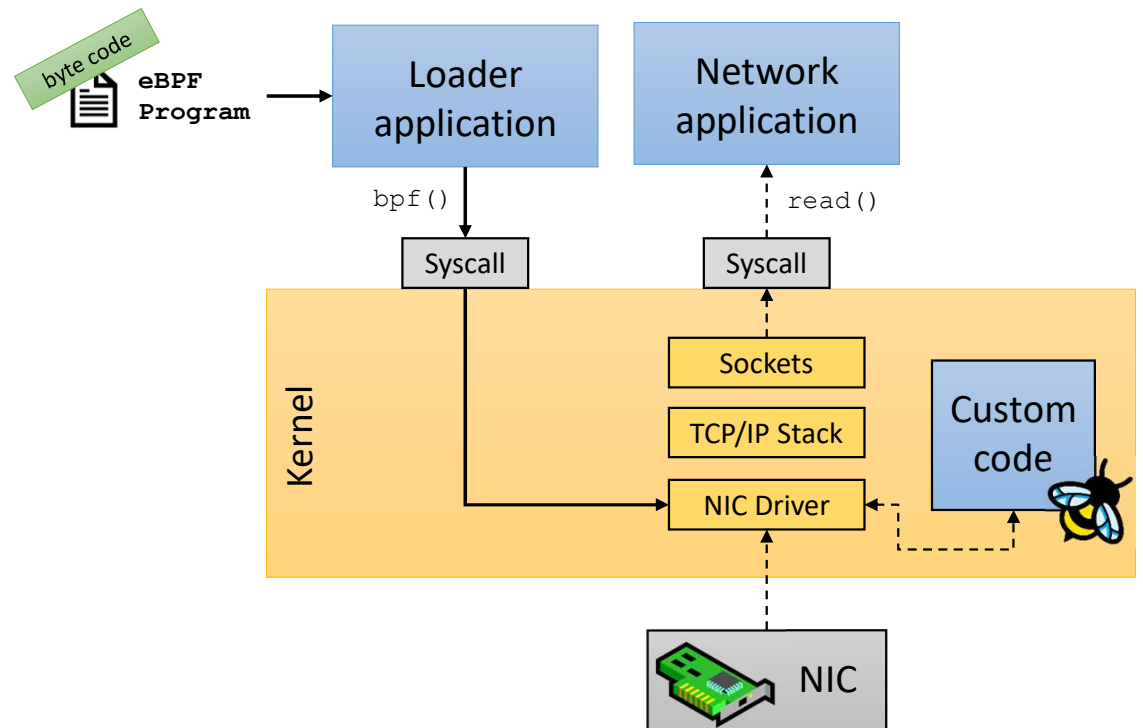
In-kernel packet processing with eBPF

- eBPF (extended Berkeley Packet Filter) is a way to embed custom packet processing code in the kernel safely at specific hook points
- Let's take example of XDP hook present in NIC driver to understand its working
- Runs custom code inside NIC driver for each packet when napi polling is done
- No `sk_buff` and copies required for XDP



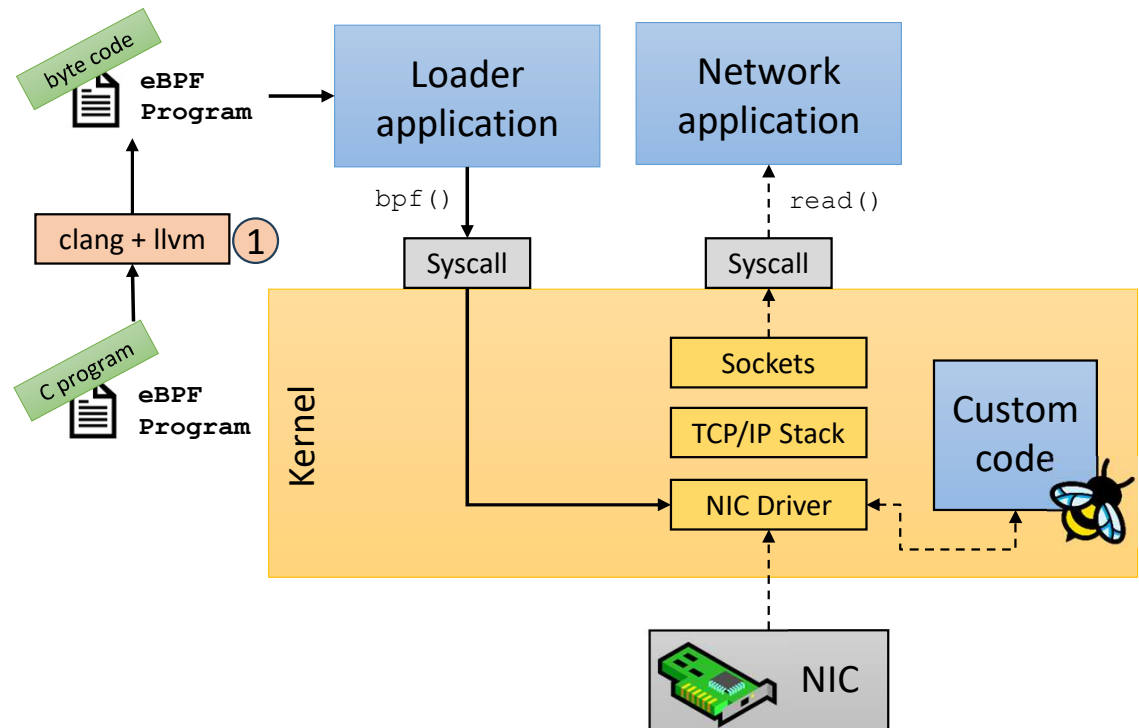
eBPF internals

- The program should be written using eBPF ISA
- bpf() system call is used to load the eBPF program
- Problems with this setup:
 1. Byte code is hard to write
 2. User program can be unsafe
 3. Byte code interpretation is slow



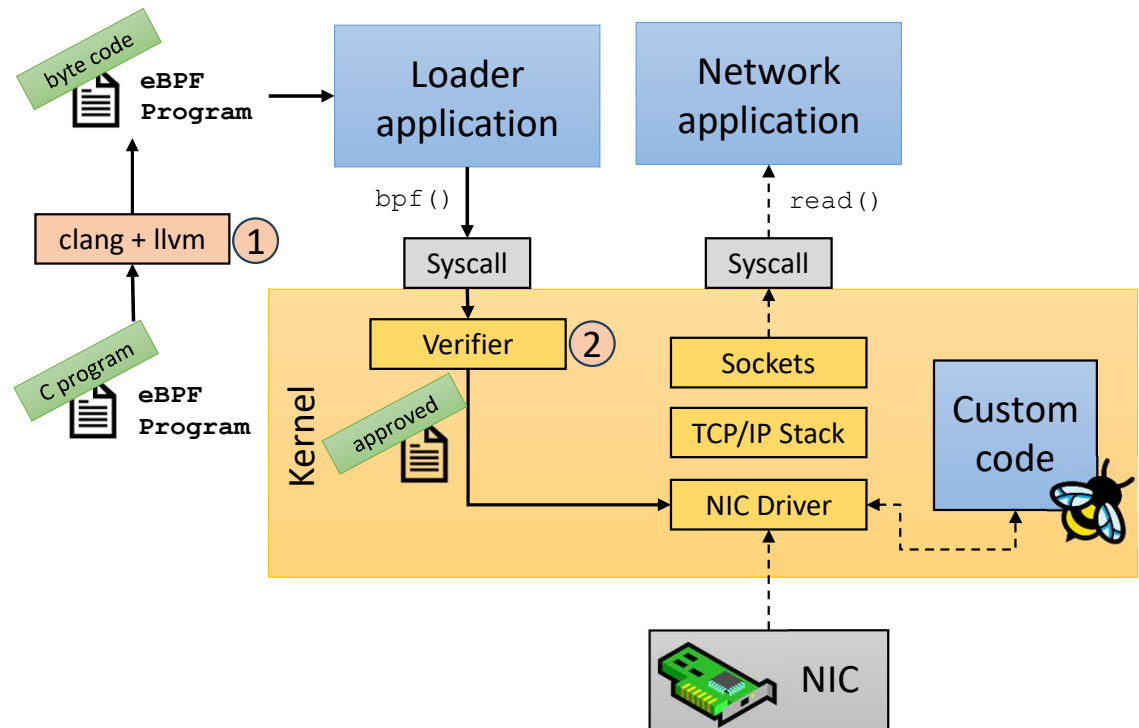
eBPF internals – clang + llvm

- The program should be written using eBPF ISA
- bpf() system call is used to load the eBPF program
- Problems with this setup:
 1. ~~Byte code is hard to write~~
 2. User program can be unsafe
 3. Byte code interpretation is slow



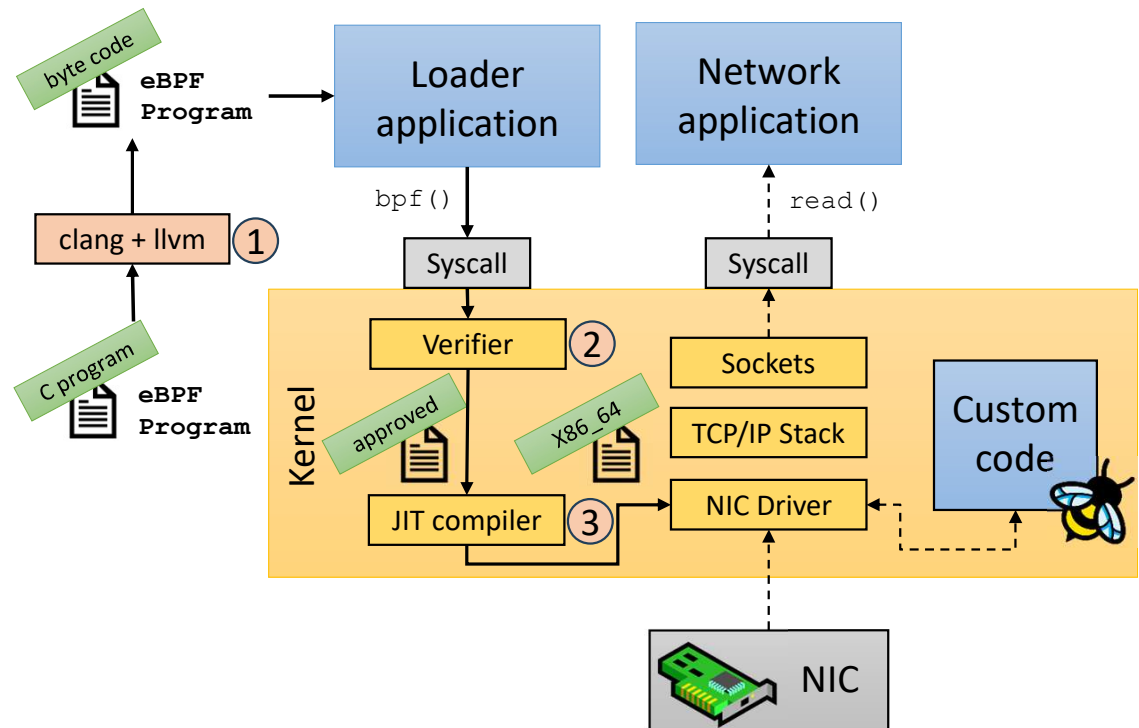
eBPF internals – the ebpf verifier

- The program should be written using eBPF ISA
- bpf() system call is used to load the eBPF program
- Problems with this setup:
 - ~~1. Byte code is hard to write~~
 - ~~2. User program can be unsafe~~
 3. Byte code interpretation is slow



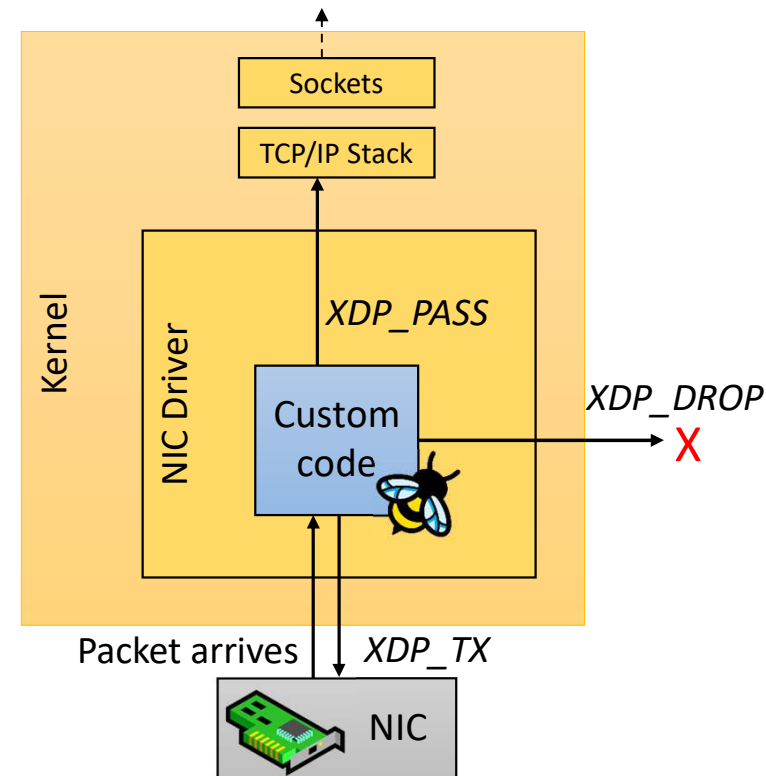
eBPF internals – JIT compilation

- The program should be written using eBPF ISA
- bpf() system call is used to load the eBPF program
- Problems with this setup:
 - ~~1. Byte code is hard to write~~
 - ~~2. User program can be unsafe~~
 - ~~3. Byte code interpretation is slow~~



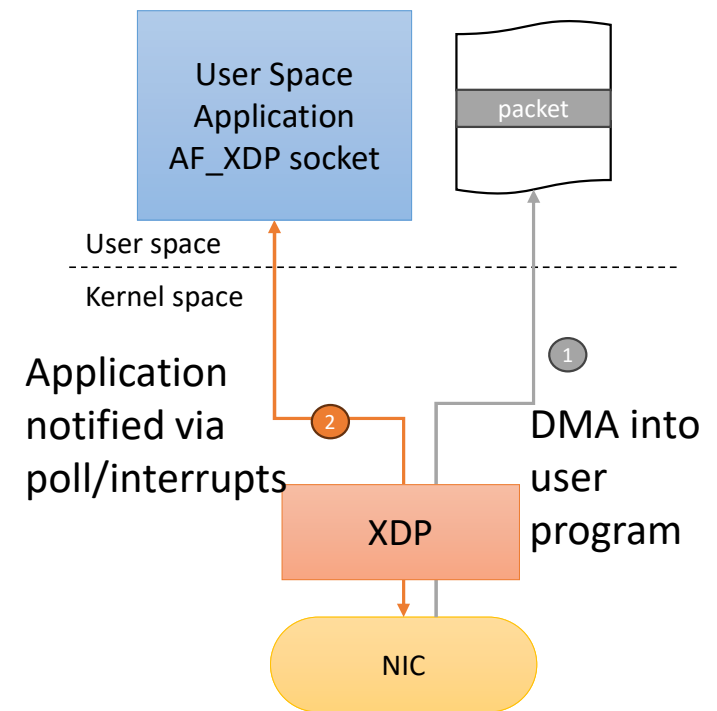
What can an eBPF program do at XDP?

- The custom code processes packets at returns a verdict:
 - XDP_PASS: Pass the packet to network stack
 - XDP_DROP: Drop the packet
 - XDP_TX: Transmit the packet ASAP
 - etc.
- Use cases:
High performance firewalls, load balancers, key-value stores, etc.
- Problems:
Difficult to write, no heap, limited program size, difficult to buffer packets, etc.



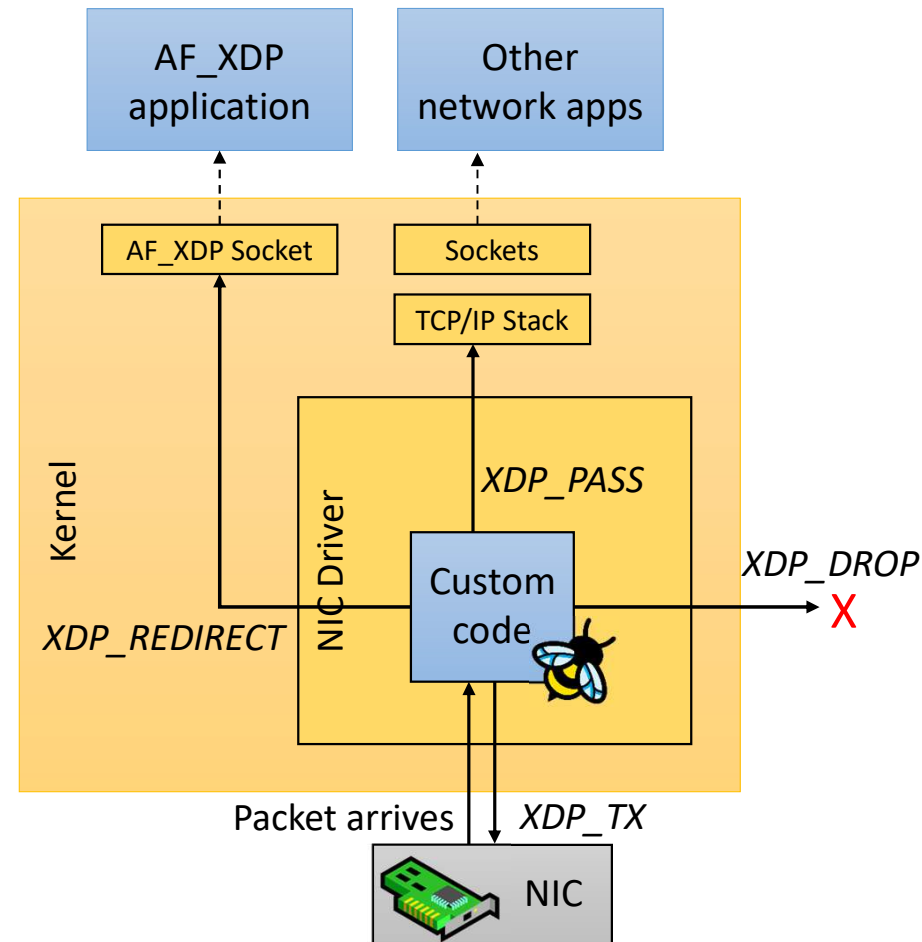
Kernel bypass with AF_XDP

- Regular sockets (AF_INET) receive packets after TCP/IP processing
- AF_XDP is special type of socket that can **receive packets directly from XDP hook**
 - Packet DMA directly into user space (with driver support), no extra copy, no kernel stack processing overhead
 - Program at XDP hook notifies user space app via poll/interrupt mechanisms
- Higher throughputs possible, while allowing kernel some control



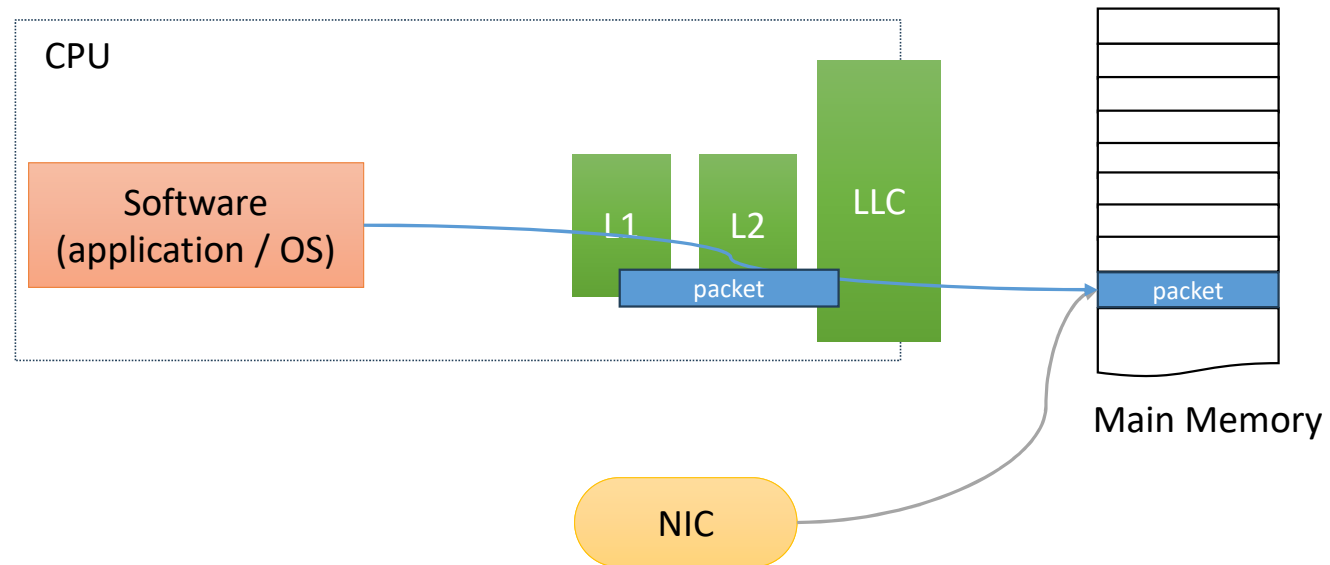
eBPF, XDP, AF_XDP – the connection

- eBPF programs can be safely embedded inside specific hook points to add custom functionality to kernel network stack
- eBPF program at XDP hook can also redirect packets to AF_XDP sockets for processing inside userspace application



Another problem: memory access bottleneck

- Memory wall: DRAM speeds have not increased as much as CPU or network hardware
- On high speed network links, only few nanoseconds budget per packet, but accessing main memory takes hundreds of nanosec



Direct Cache Access / DDIO

- Direct Cache Access (Intel's DDIO): NIC writes packet directly into CPU caches, and does not DMA into main memory
- User/kernel software can access packet quickly from cache
- Leads to much faster network packet processing

