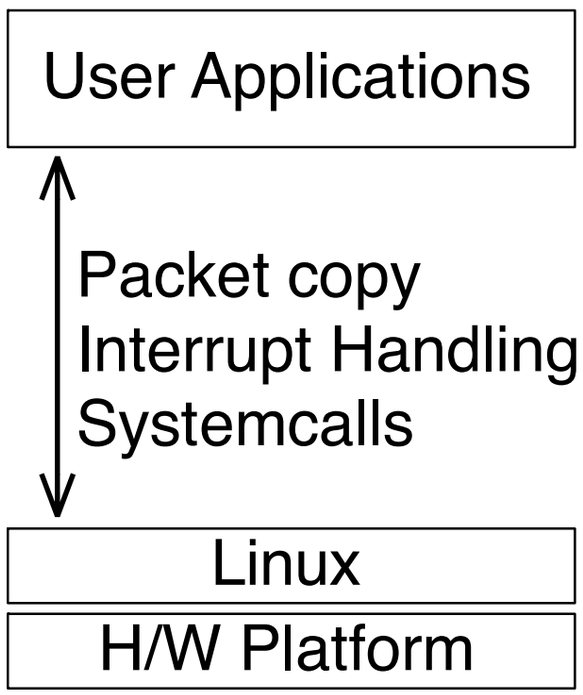
Software-Based Networking

Getting started with DPDK

Linux Packet Processing

Traditional networking:

- NIC uses DMA to copy data into kernel buffer
- Interrupt when packets arrive
- Copy packet data from kernel space to user space
- Use system call to send data from user space

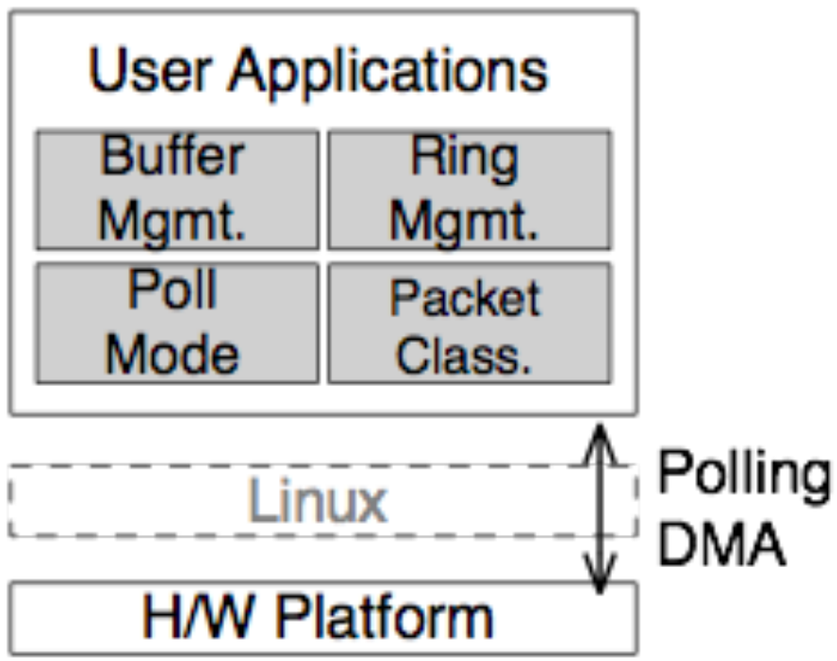


Can you handle being interrupted 60 million times per second?

User Space Packet Processing

Recent NICs and OS support allow user space apps to directly access packet data

- NIC uses DMA to copy data into ~~kernel~~ **user space** buffer
- ~~Interrupt~~ **use polling to find** when packets arrive
- ~~Copy packet data from kernel space to user space~~
- Use ~~system~~ **regular function** call to send data from user space



Data Plane Development Kit

High performance I/O library

Poll mode driver reads packets from NIC

Packets bypass the OS and are copied directly into user space memory

Low level library... does not provide:

- Support for multiple network functions
- SDN-based control
- Interrupt-driven NFs
- State management
- TCP stack

Data Plane Development Kit

Where to find it:

- http://dpdk.org/

What to use it for:

- Applications that need high speed access to low-level packet data

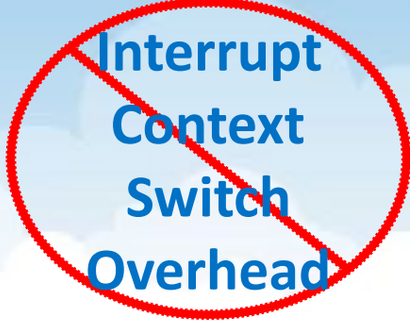
Why try it:

- One of the best documented open source projects I've ever seen

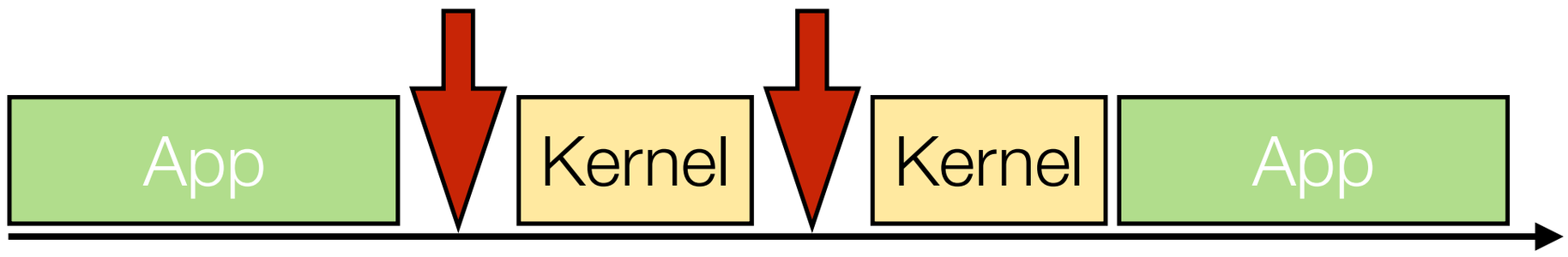
Alternatives:

- netmap
- PF_RING

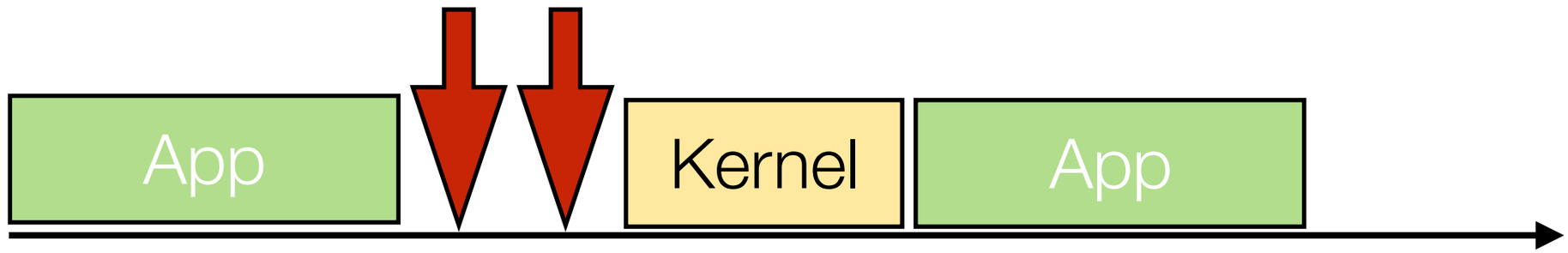
Network Interrupts



Very distracting! Have to stop doing useful work to handle incoming packets

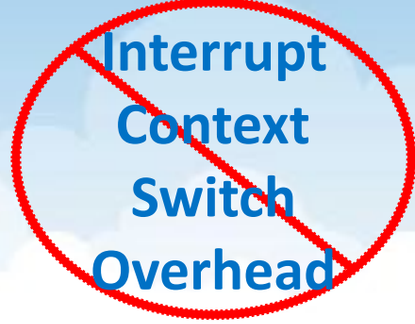


Coalescing interrupts helps, but still causes problems



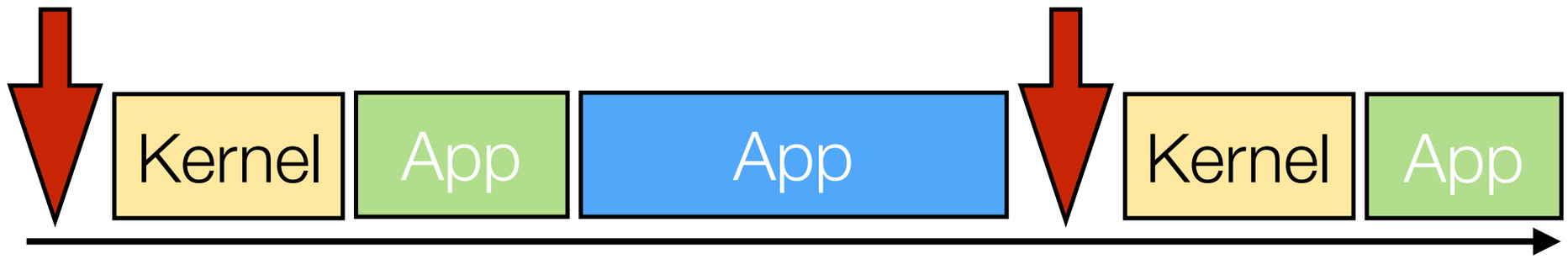
- Interrupts can arrive during critical sections!
- Interrupts can be delivered to the wrong CPU core!
- Still must pay context switch cost

Polling



Continuously loop looking for new packet arrivals

Trade-off?



Interrupts help share the CPU

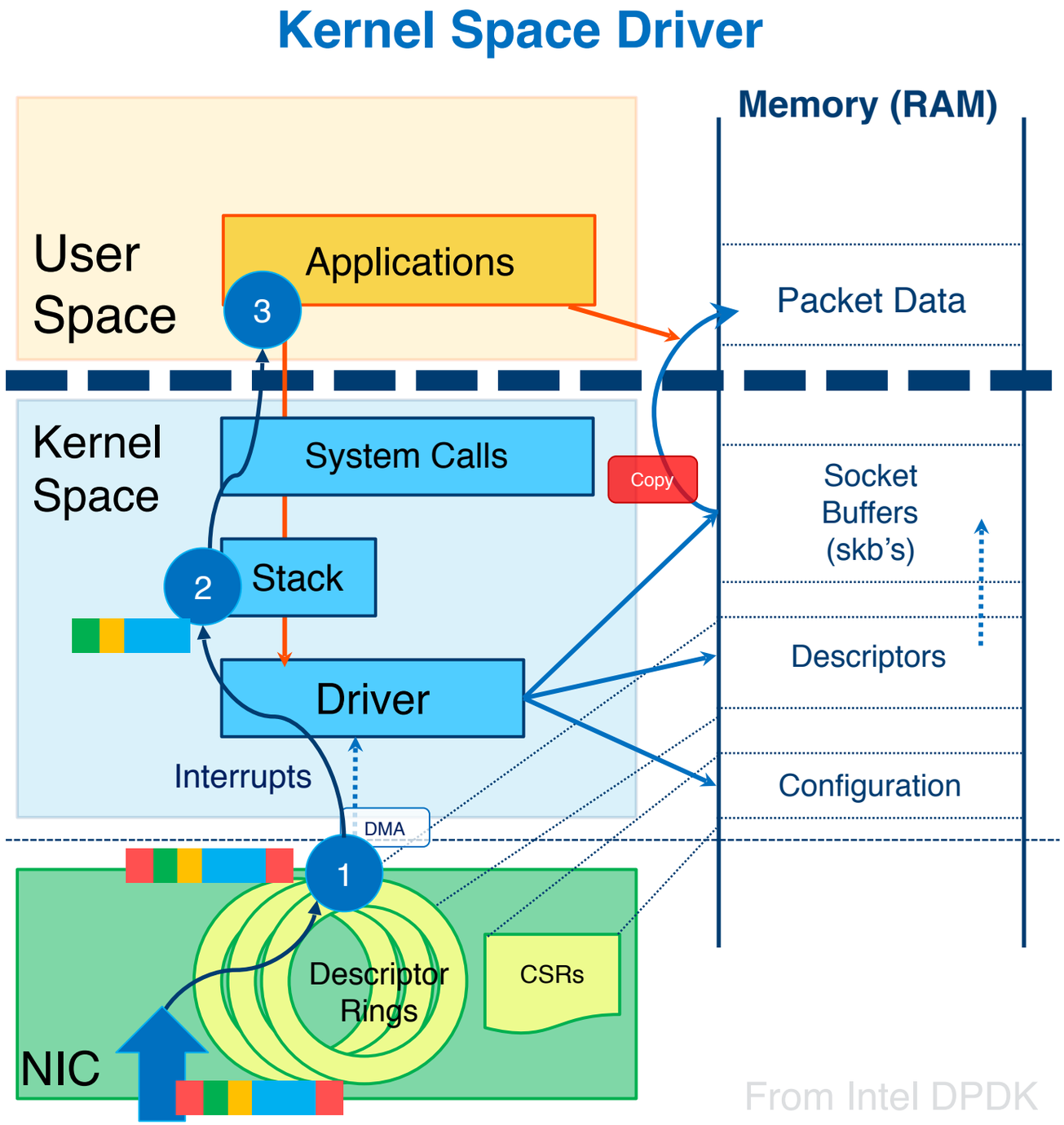
Polling can be wasteful

Kernel-User Overhead

NIC Driver operates in kernel mode

- Reads packets into kernel memory
- Stack pulls data out of packets
- Data is copied into user space for application
- Application uses system calls to interface with OS

Why is copying so bad?



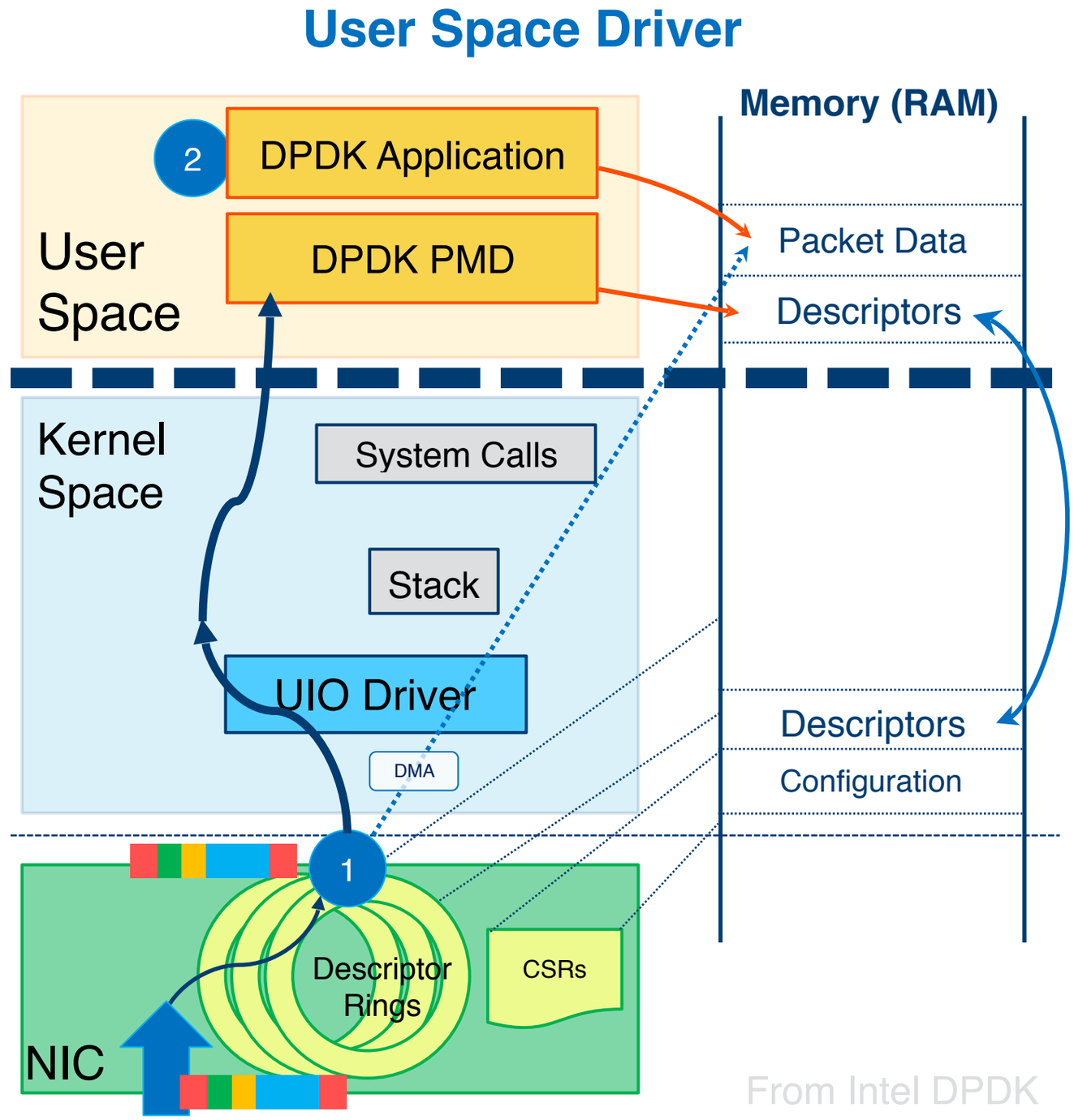
From Intel DPDK University Lecture

Kernel Bypass

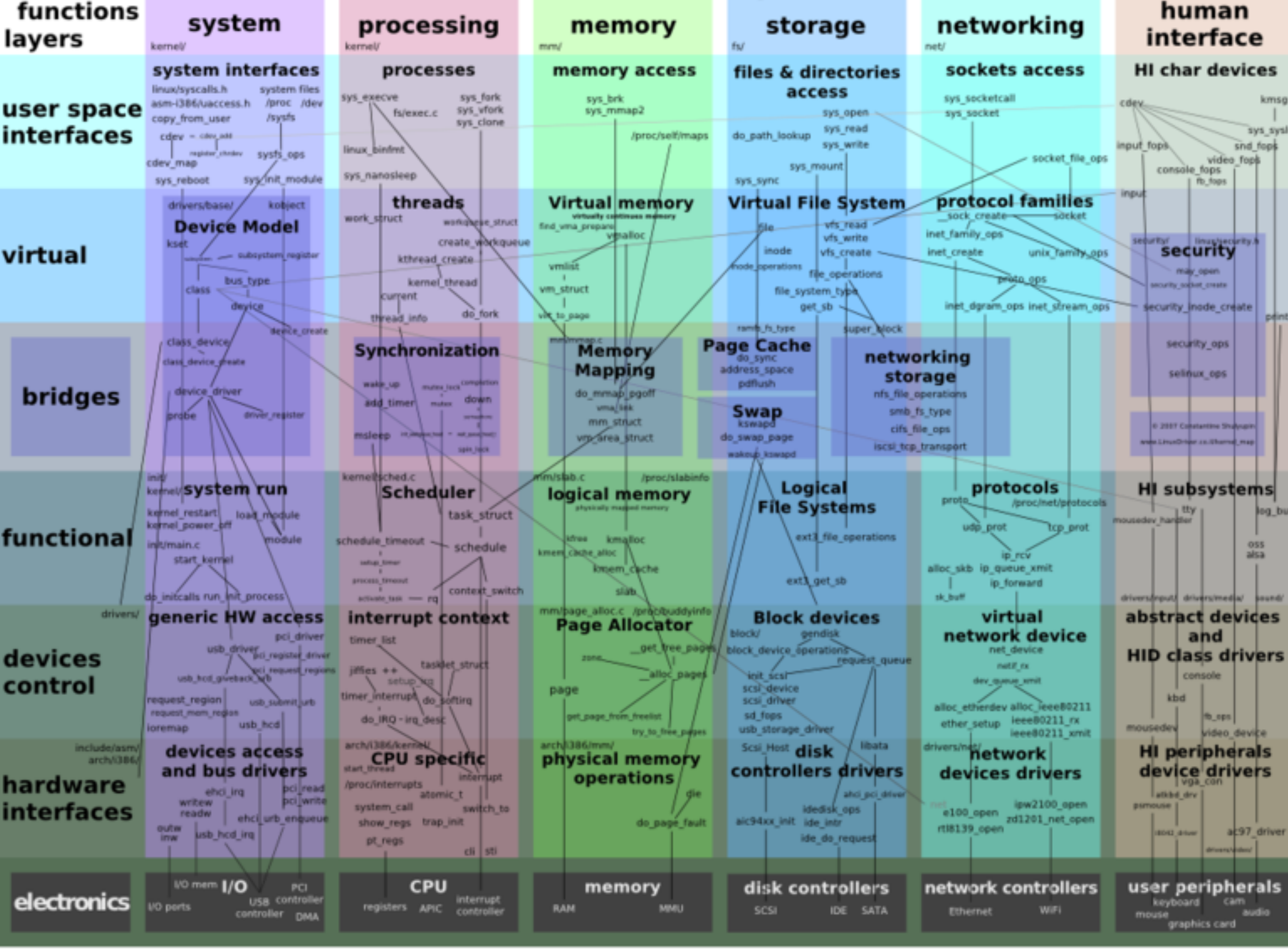
User-mode Driver

- Kernel only sets up basic access to NIC
- User-space driver tells NIC to DMA data directly into user-space memory

- No extra copies
- No in-kernel processing
- No context switching



From Intel DPDK University Lecture


functions
layers
system
kernel/
processing
kernel/
memory
mm/
storage
fs/
networking
net/
human interface
user space interfaces
system interfaces
linux/syscalls.h
asm-i386/uaccess.h
copy_from_user
system files
/proc
/dev
/sysfs
cdev
cdev_add
register_chrdev
cdev_map
sysfs_ops
sys_reboot
sys_init_module
processes
sys_execve
fs/exec.c
sys_fork
sys_vfork
sys_clone
linux_binfmt
sys_nanosleep
memory access
sys_brk
sys_mmap2
/proc/self/maps
files & directories access
sys_open
sys_read
sys_write
do_path_lookup
sys_mount
sys_sync
sockets access
sys_socketcall
sys_socket
socket_file_ops
HI char devices
cdev
kmsg
sys_sysl
input_fops
snd_fops
video_fops
console_fops
fb_fops
input
virtual
drivers/base/
kobject
Device Model
kset
subsystem
subsystem_register
bus_type
class
device
device_create
threads
work_struct
workqueue_struct
create_workqueue
kthread_create
kernel_thread
current
thread_info
do_fork
Virtual memory
virtually continuous memory
find_vma_prepare
vmalloc
vmlist
vm_struct
vir_to_page
mm/mmap.c
Virtual File System
file
inode
inode_operations
vfs_read
vfs_write
vfs_create
file_operations
file_system_type
get_sb
super_block
ramfs_fs_type
protocol families
__sock_create
socket
inet_family_ops
inet_create
unix_family_ops
proto_ops
inet_dgram_ops
inet_stream_ops
security/
linux/security.h
security
may_open
security_socket_create
security_inode_create
security_ops
selinux_ops
bridges
class_device
class_device_create
device_driver
probe
driver_register
Synchronization
wake_up
add_timer
msleep
mutex_lock
mutex
completion
down
spin_lock
Memory Mapping
do_mmap_pgoff
vma_link
mm_struct
vm_area_struct
Page Cache
do_sync
address_space
pdflush
Swap
kswapd
do_swap_page
wakeup_kswapd
networking storage
nfs_file_operations
smb_fs_type
cifs_file_ops
iscsi_tcp_transport
functional
init/
kernel/
system run
kernel_restart
kernel_power_off
load_module
module
init/main.c
start_kernel
do_initcalls
run_init_process
kernel/sched.c
Scheduler
task_struct
schedule_timeout
schedule
setup_timer
process_timeout
activate_task
rq
context_switch
mm/slab.c
/proc/slabinfo
logical memory
physically mapped memory
kfree
kmalloc
kmem_cache_alloc
kmem_cache
slab
Logical File Systems
ext3_file_operations
ext3_get_sb
protocols
proto
/proc/net/protocols
udp_prot
tcp_prot
ip_rcv
alloc_skb
ip_queue_xmit
ip_forward
sk_buff
HI subsystems
tty
mousedev_handler
oss
alsa
devices control
drivers/
generic HW access
pci_driver
usb_driver
pci_register_driver
pci_request_regions
usb_hcd_giveback_urb
request_region
request_mem_region
ioremap
usb_submit_urb
usb_hcd
interrupt context
timer_list
jiffies ++
setup_irq
timer_interrupt
do_softirq
do_IRQ
irq_desc
tasklet_struct
mm/page_alloc.c
/proc/buddyinfo
Page Allocator
__get_free_pages
zone
__alloc_pages
page
get_page_from_freelist
try_to_free_pages
Block devices
block/
gendisk
block_device_operations
request_queue
init_scsi
scsi_device
scsi_driver
sd_fops
usb_storage_driver
virtual network device
net_device
netif_rx
dev_queue_xmit
alloc_etherdev
alloc_ieee80211
ether_setup
ieee80211_rx
ieee80211_xmit
abstract devices and HID class drivers
console
kbd
mousedev
fb_sps
video_device
hardware interfaces
include/asm/
arch/i386/
devices access and bus drivers
ehci_irq
writew
readw
outw
inw
usb_hcd_irq
pci_read
pci_write
ehci_urb_enqueue
arch/i386/kernel/
CPU specific
start_thread
/proc/interrupts
atomic_t
system_call
show_regs
trap_init
pt_regs
interrupt
switch_to
cli
sti
arch/i386/mm/
physical memory operations
die
do_page_fault
disk controllers drivers
Scsi_Host
libata
aic94xx_init
idedisk_ops
ide_intr
ide_do_request
ahci_pci_driver
network devices drivers
drivers/net/
e100_open
rtl8139_open
ipw2100_open
zd1201_net_open
HI peripherals device drivers
vga_con
atkbd_drv
psmouse
ac97_driver
electronics
I/O
I/O mem
I/O ports
USB controller
PCI controller
DMA
CPU
registers
APIC
interrupt controller
memory
RAM
MMU
disk controllers
SCSI
IDE
SATA
network controllers
Ethernet
WIFI
user peripherals
keyboard
cam
mouse
audio
graphics card

Networking

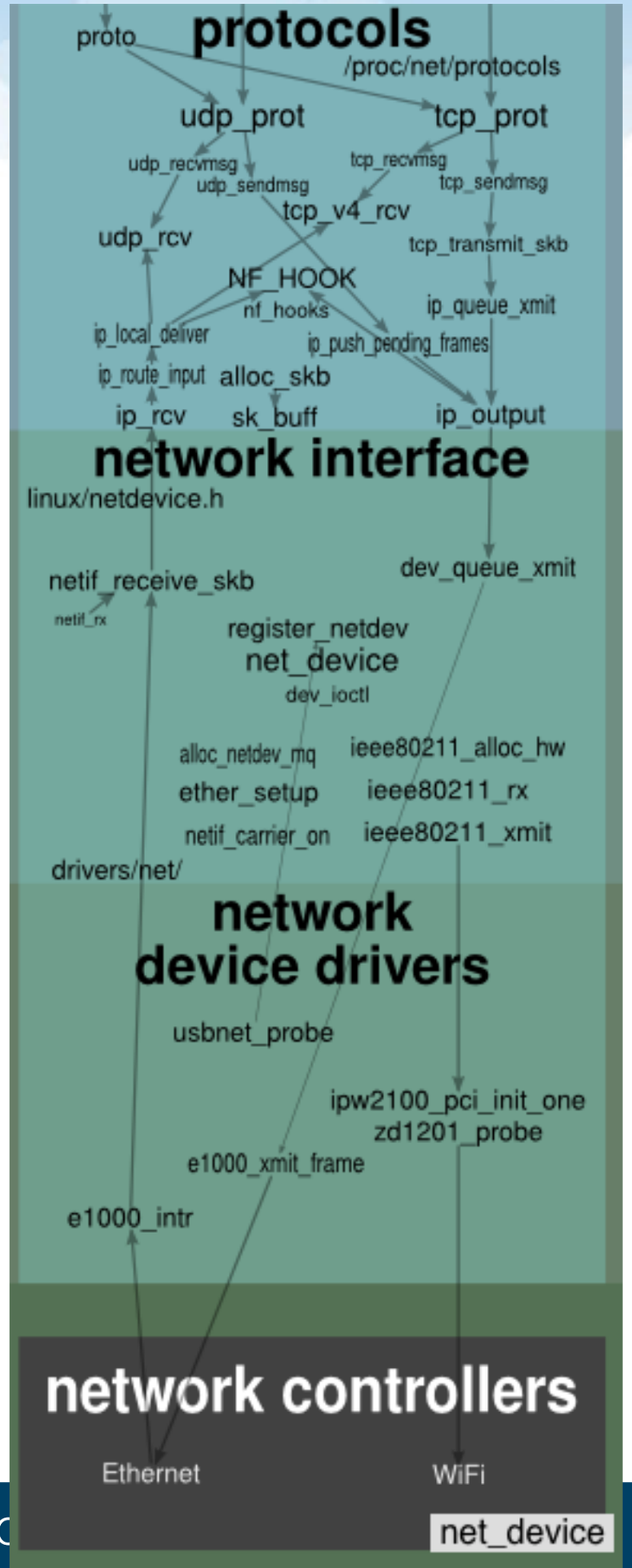
Linux networking stack has a lot of extra components

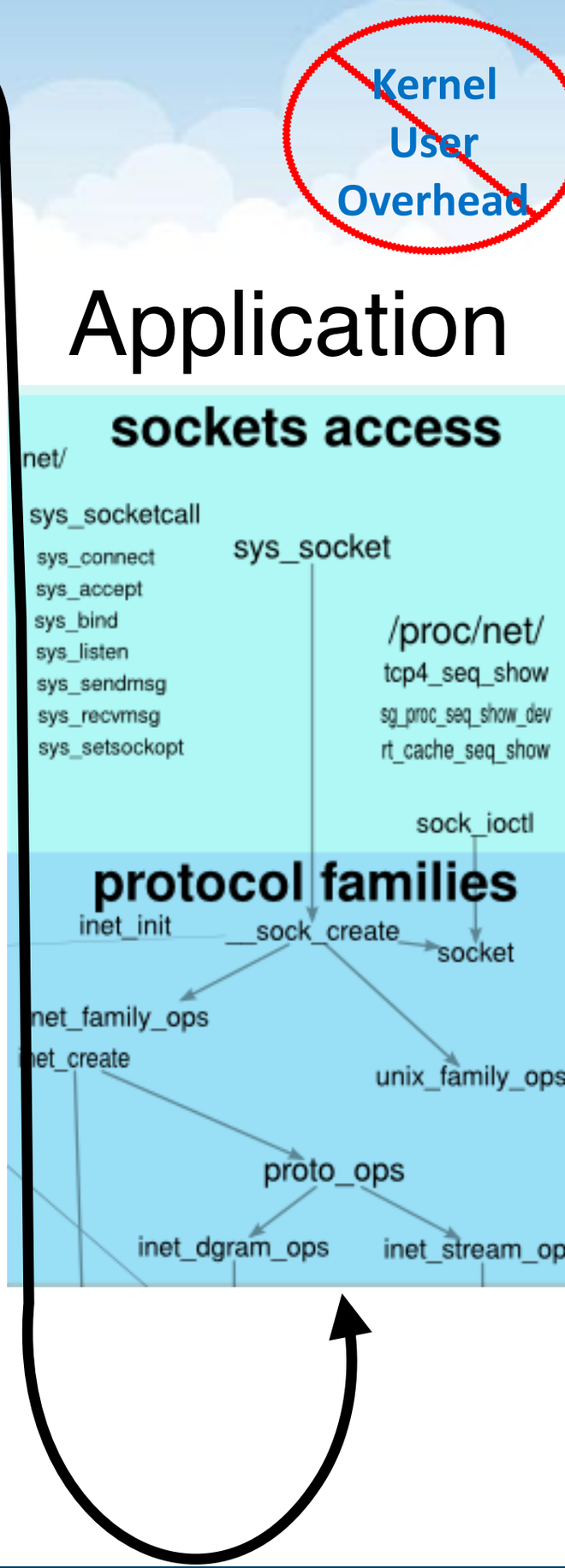
For NFV middlebox we don't use all of this:

- TCP, UDP, sockets

NFV middle boxes just need packet data

- Need it fast!

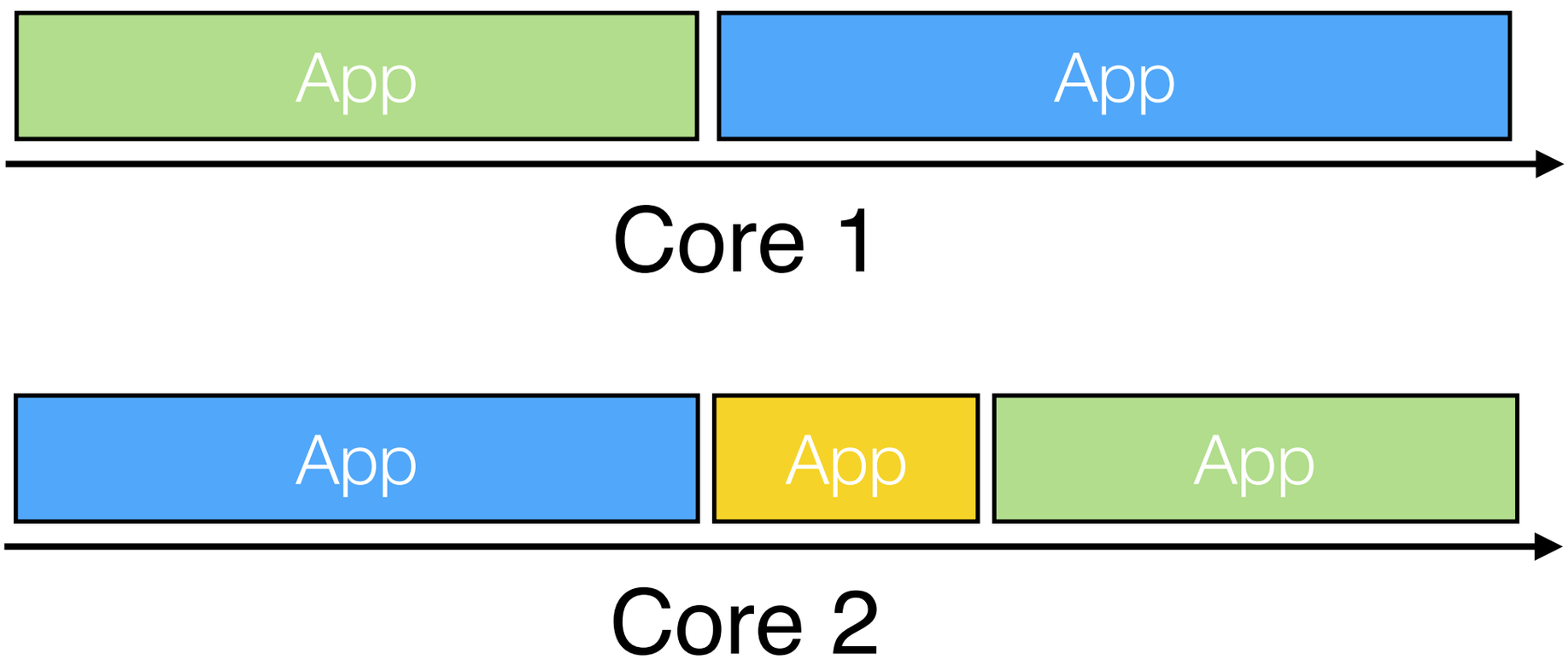




CPU Core Affinity

Linux Scheduler can move threads between cores

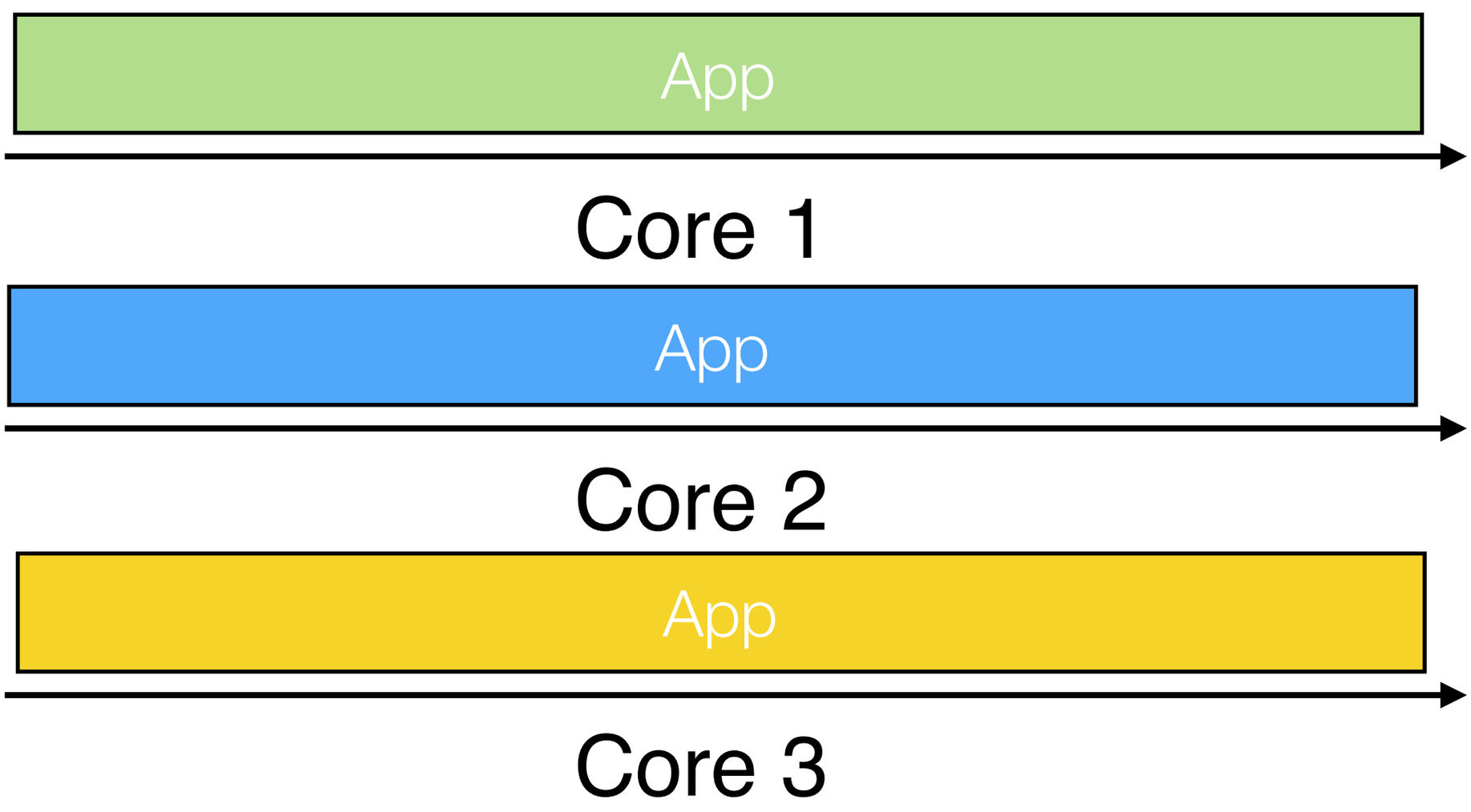
- Context switches :(
- Cache locality :(



CPU Core Affinity

Pin threads and dedicate cores

- **Trade-offs?**



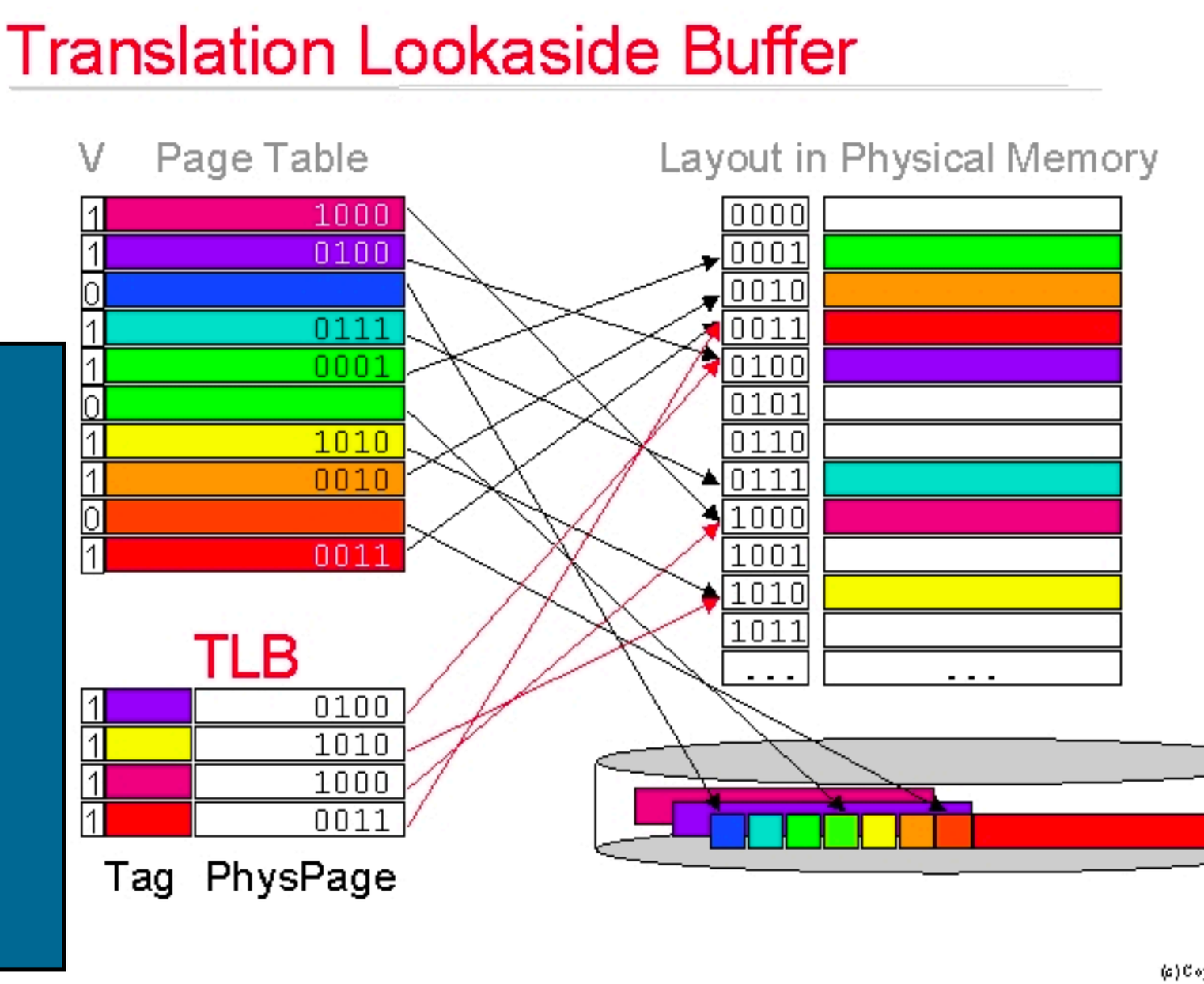
Paging Overhead

4KB Pages

- 4 packets per page
- 14 million pps
- 3.6 million page table entries every second

How big is the TLB?

Packet ~= 1KB



https://courses.cs.washington.edu/courses/cse378/00au/CSE378-00.Lec28/sld004.htm

Locks

Thread synchronization is expensive

- Tens of nanoseconds to take an uncontested lock
- 10Gbps -> 68ns per packet

Producer/Consumer architecture

- Gather packets from NIC (producer) and ask worker to process them (consumer)

Lock-free communication

- Ring-buffer based message queues

Bulk Operations

PCIe bus uses messaging protocols for CPU to interact with devices (NICs)

Each message incurs some overhead

Better to make larger bulk requests over PCIe

DPDK helps batch requests into bulk operations

- Retrieve a batch (32) of packet descriptors received by NIC
- Enqueue/dequeue beaches of packet descriptors onto rings

Trade-offs?

Using DPDK

Lots of examples! Great docs!

- But still fairly complex…
- http://dpdk.readthedocs.io/

Simple hands-on exercise… who will help?

Help me out!

I need volunteers to help run DPDK

- Find info at: **`https://github.com/sdnfv/onvm-tutorial`**
- Use nodes 2-9

Connect to your server in a terminal windows

- user=**`tutorial`** pw=**`sigcomm`**

```
# become root
sudo -s
# change to ONVM main directory
cd $ONVM_HOME
# configure DPDK to use NICs
./scripts/setup_nics.sh dpdk
```

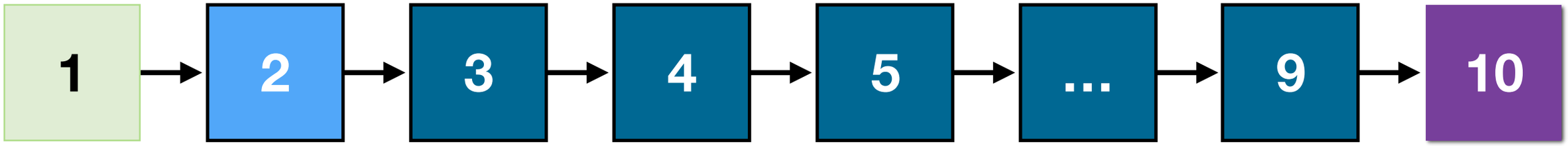
Legal stuff: *You may only use these servers for running code related to the tutorial!*

Testbed Setup

Servers are courtesy of Cloudlab.us NSF testbed

10 servers connected in a chain

Already have ONVM/DPDK installed and configured



Our goal:

- Send traffic from node 1 to node 10
- Forward all packets through the switches using DPDK

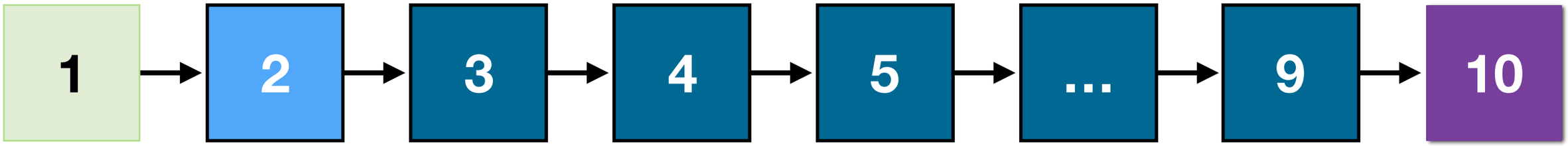
Run the basic forwarder

Chanage to the DPDK forwarding example directory

- **`cd $RTE_SDK/examples/skeleton`**

Run the go script to start the program

- **`./go.sh`**
- this is equivalent to: `./build/basicfwd -l 1 -n 4`



I will run a workload on nodes 1/10

DPDK Basic Forward

Basic forwarding with DPDK

- dpdk/examples/skeleton/basicfwd.c (docs)
- Create a memory pool to store packets
- Initialize NIC ports and configure with mem pool
- Initialize ring buffer for receiving packets from NIC
- Initialize ring buffer for transmitting packets to NIC
- For each port:
  - Dequeue a batch of packets
  - Transmit each packet out the opposite port

DPDK Limitations

Barebones I/O library

- No protocol processing (e.g., TCP stack)

Focused on running a single NF

- Need to specially design NFs to be able to run multiple functions at once
- NF monopolizes entire NIC port(s)

Extensive use of polling means high CPU usage

No management layer

- addressing, reliability, auto scaling, fault tolerance