An introduction to Netmap

Giuseppe Lettieri

Dipartimento di Ingegneria dell'Informazione
Università di Pisa

SIGCOMM'17, Los Angeles, CA, 27 August 2017

What is netmap?

- A framework for high speed packet I/O
- Initially designed and developed by Luigi Rizzo at University of Pisa (http://info.iet.unipi.it/ luigi/netmap/)
- Part of FreeBSD, external module for Linux and Windows
- Code available on github: https://github.com/luigirizzo/netmap

What is netmap?

- A framework for high speed packet I/O
- Initially designed and developed by Luigi Rizzo at University of Pisa (http://info.iet.unipi.it/ luigi/netmap/)
- Part of FreeBSD, external module for Linux and Windows
- Code available on github: https://github.com/luigirizzo/netmap

What is netmap?

- A framework for high speed packet I/O
- Initially designed and developed by Luigi Rizzo at University of Pisa (http://info.iet.unipi.it/ luigi/netmap/)
- Part of FreeBSD, external module for Linux and Windows
- Code available on github: https://github.com/luigirizzo/netmap

What is netmap?

- A framework for high speed packet I/O
- Initially designed and developed by Luigi Rizzo at University of Pisa (http://info.iet.unipi.it/ luigi/netmap/)
- Part of FreeBSD, external module for Linux and Windows
- Code available on github: https://github.com/luigirizzo/netmap

netmap numbers

- line rate for 10G with minimum sized packets using a fraction of a core
- over 30 Mpps on 40G NICs (limited by the NIC's hardware)
- over 20 Mpps on VALE ports (software switch)
- over 100 Mpps on netmap pipes
- almost the same on bare metal or virtual machines

netmap numbers

- line rate for 10G with minimum sized packets using a fraction of a core
- over 30 Mpps on 40G NICs (limited by the NIC's hardware)
- over 20 Mpps on VALE ports (software switch)
- over 100 Mpps on netmap pipes
- almost the same on bare metal or virtual machines

netmap numbers

- line rate for 10G with minimum sized packets using a fraction of a core
- over 30 Mpps on 40G NICs (limited by the NIC's hardware)
- over 20 Mpps on VALE ports (software switch)
- over 100 Mpps on netmap pipes
- almost the same on bare metal or virtual machines

netmap numbers

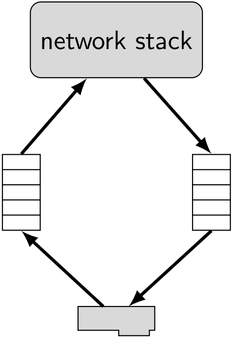
- line rate for 10G with minimum sized packets using a fraction of a core
- over 30 Mpps on 40G NICs (limited by the NIC's hardware)
- over 20 Mpps on VALE ports (software switch)
- over 100 Mpps on netmap pipes
- almost the same on bare metal or virtual machines

netmap numbers

- line rate for 10G with minimum sized packets using a fraction of a core
- over 30 Mpps on 40G NICs (limited by the NIC's hardware)
- over 20 Mpps on VALE ports (software switch)
- over 100 Mpps on netmap pipes
- almost the same on bare metal or virtual machines

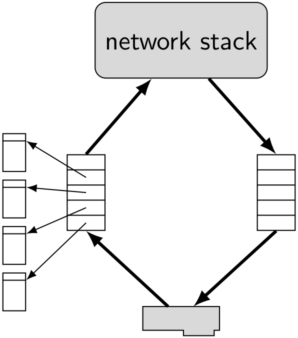
Background: RX in traditional OS

let us consider the RX path



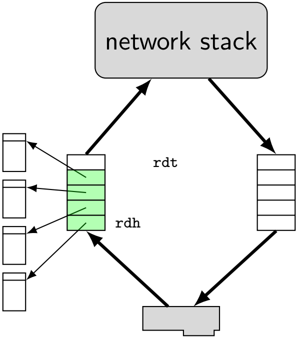
Background: RX in traditional OS

at open time, the driver fills the RX ring with empty skbufs



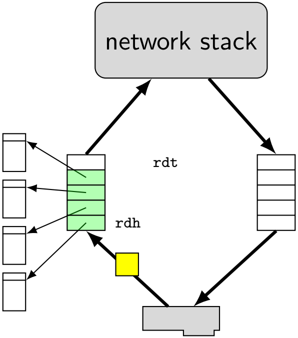
Background: RX in traditional OS

the ring slots that the NIC can use are in the $[\mathtt{RDH}, \mathtt{RDT})$ interval



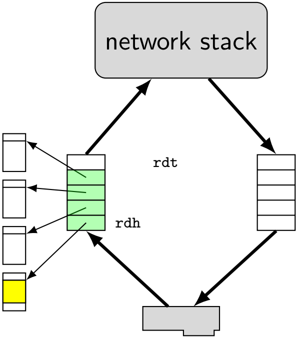
Background: RX in traditional OS

when a new message is received



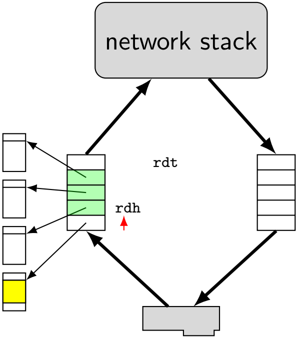
Background: RX in traditional OS

the NIC copies it into the first available skbuf



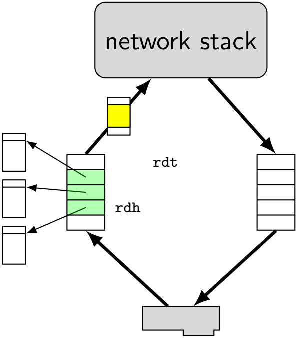
Background: RX in traditional OS

it updates the head pointer and possibily sends an interrupt to notify the driver



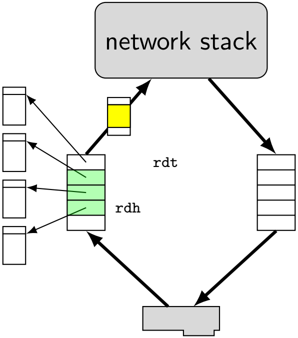
Background: RX in traditional OS

the driver eventually notices the new message and moves the skbuf up the stack



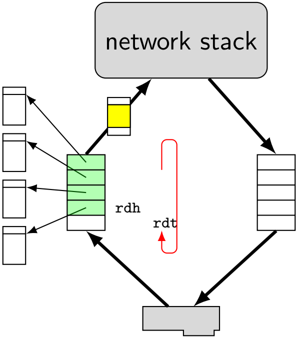
Background: RX in traditional OS

the driver allocates a new empty skbuf to fill the ring again

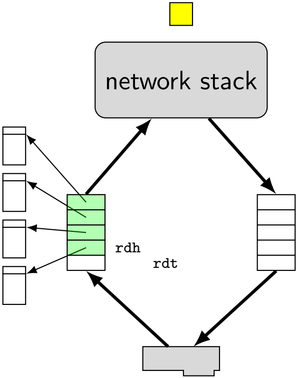


Background: RX in traditional OS

it then moves the tail to make the new skbuf available to the NIC

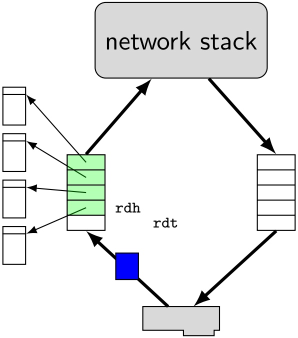


the message is eventually copied to userspace and the containing skbuf is discarded



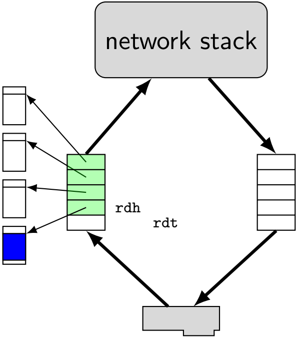
Background: RX in traditional OS

many messages may be received before the driver starts processing the queue



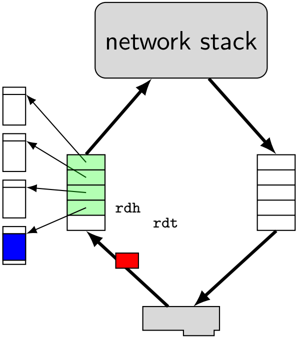
Background: RX in traditional OS

many messages may be received before the driver starts processing the queue



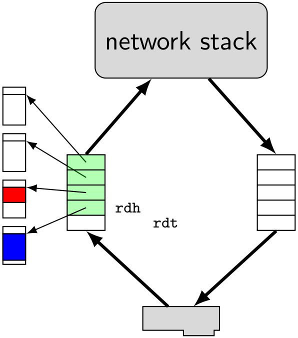
Background: RX in traditional OS

many messages may be received before the driver starts processing the queue



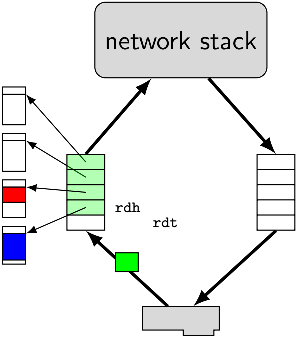
Background: RX in traditional OS

many messages may be received before the driver starts processing the queue



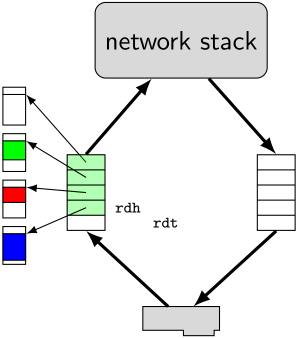
Background: RX in traditional OS

many messages may be received before the driver starts processing the queue



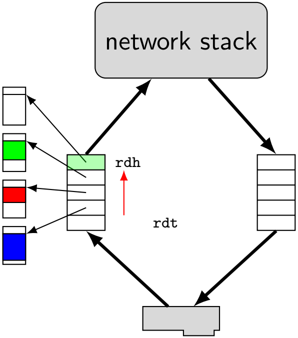
Background: RX in traditional OS

many messages may be received before the driver starts processing the queue



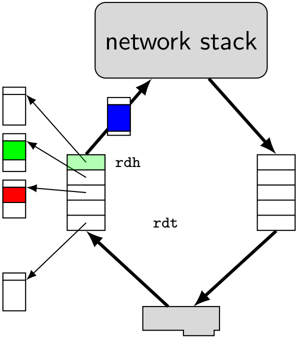
Background: RX in traditional OS

a single update of the head pointer will reveal all the new messages



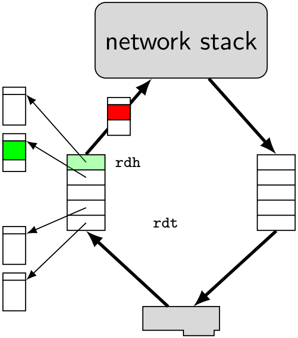
Background: RX in traditional OS

the driver will process all of them, possibly in a single run



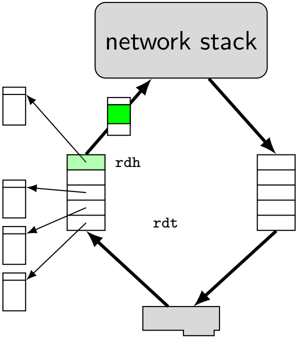
Background: RX in traditional OS

the driver will process all of them, possibly in a single run



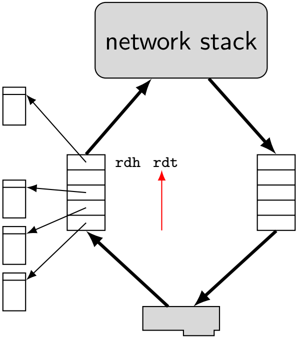
Background: RX in traditional OS

the driver will process all of them, possibly in a single run



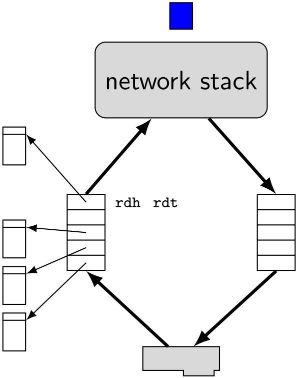
Background: RX in traditional OS

the tail pointer can be updated a single time



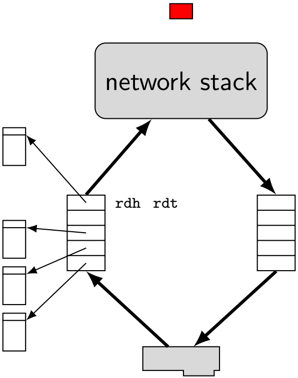
Background: RX in traditional OS

the user will still have to copy each one of the messages, possibly via several system calls



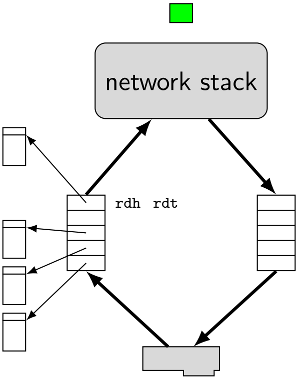
Background: RX in traditional OS

the user will still have to copy each one of the messages, possibly via several system calls



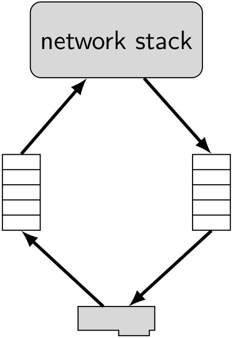
Background: RX in traditional OS

the user will still have to copy each one of the messages, possibly via several system calls



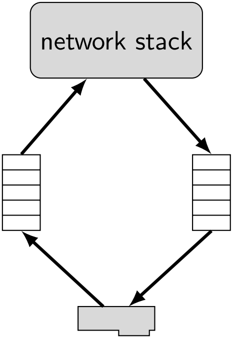
Background: TX in traditional OS

let us now consider the TX path



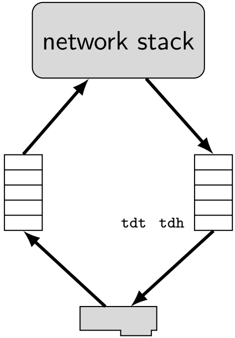
Background: TX in traditional OS

at open time, the TX ring is empty



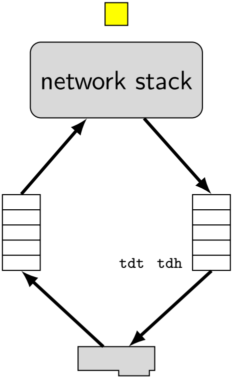
Background: TX in traditional OS

this is signaled by $\mathtt{TDT} = \mathtt{TDH}$



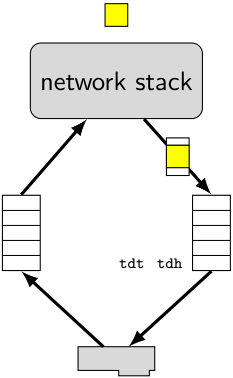
Background: TX in traditional OS

assume an application sends a new message through a socket



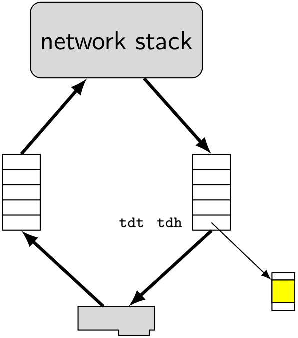
Background: TX in traditional OS

the kernel allocates an skbuf where it copies the message, then pushes it down the stack until it reaches the driver



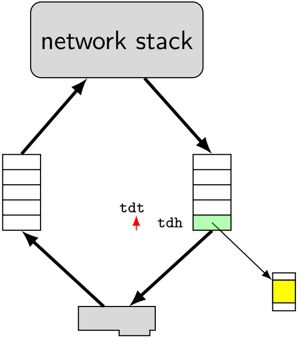
Background: TX in traditional OS

the driver links the skbuf in the TX ring



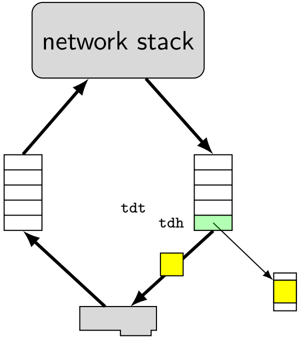
Background: TX in traditional OS

then it notifies the NIC by updating the ring tail



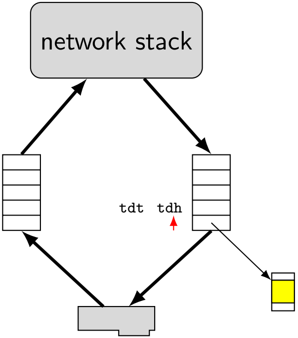
Background: TX in traditional OS

the NIC reads the message via DMA and sends it over the link



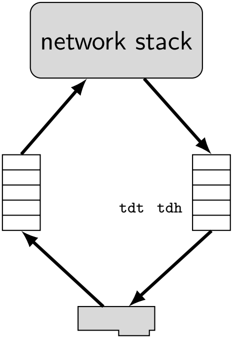
Background: TX in traditional OS

when the DMA is completed the NIC updates the ring head and possibily sends an interrupt



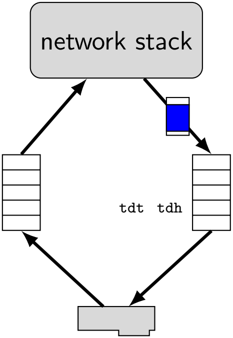
Background: TX in traditional OS

the driver eventually notices and frees the skb



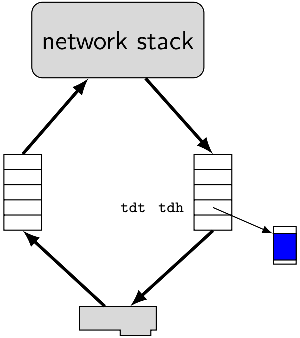
Background: TX in traditional OS

the driver may push several skb's in the queue



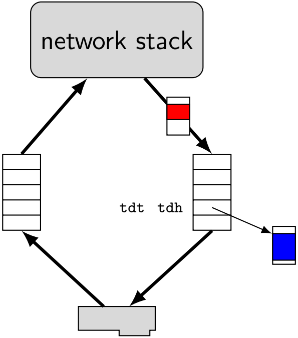
Background: TX in traditional OS

the driver may push several skb's in the queue



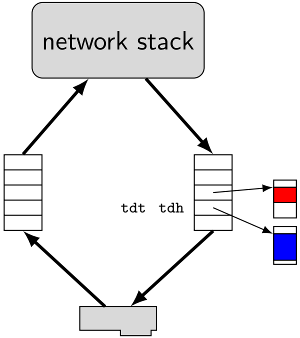
Background: TX in traditional OS

the driver may push several skb's in the queue



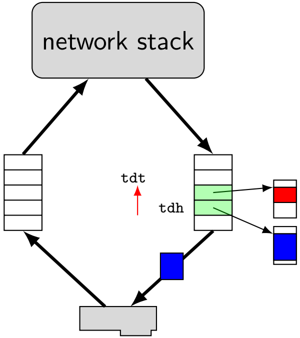
Background: TX in traditional OS

the driver may push several skb's in the queue



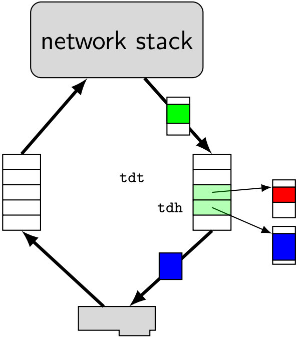
Background: TX in traditional OS

a single update of TDT may notify several messages



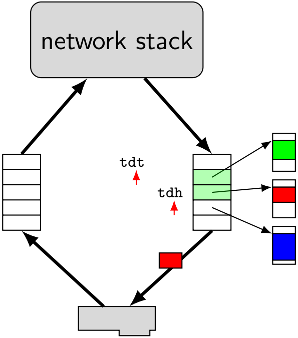
Background: TX in traditional OS

other messages may be enqued while the driver is sending the previous ones



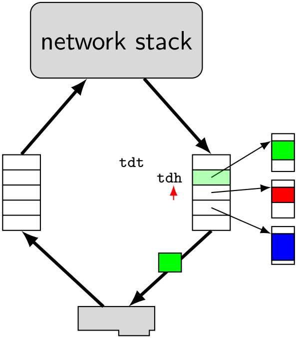
Background: TX in traditional OS

other messages may be enqued while the driver is sending the previous ones



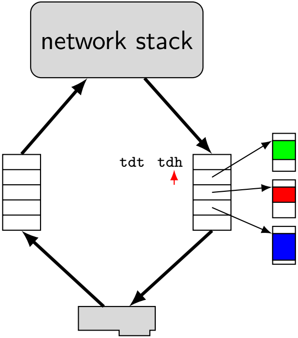
Background: TX in traditional OS

other messages may be enqued while the driver is sending the previous ones



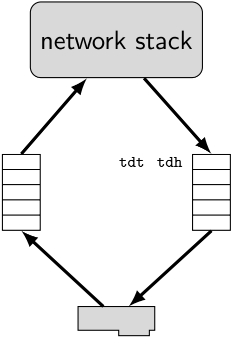
Background: TX in traditional OS

other messages may be enqued while the driver is sending the previous ones



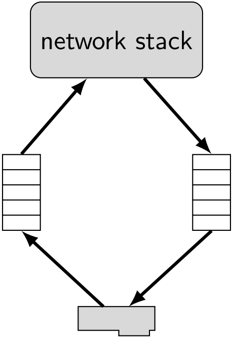
Background: TX in traditional OS

all completed skb's may be fred in a single run of the driver

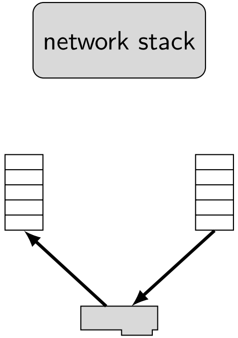


Netmap mode

let us now consider a NIC open in netmap mode

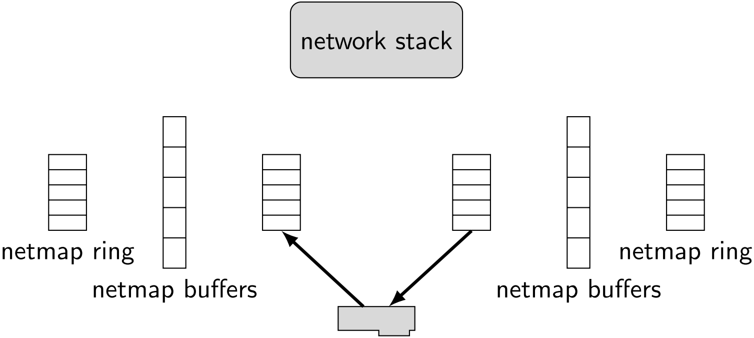


when we open a NIC in netmap mode, the NIC is disconnected from the host stack



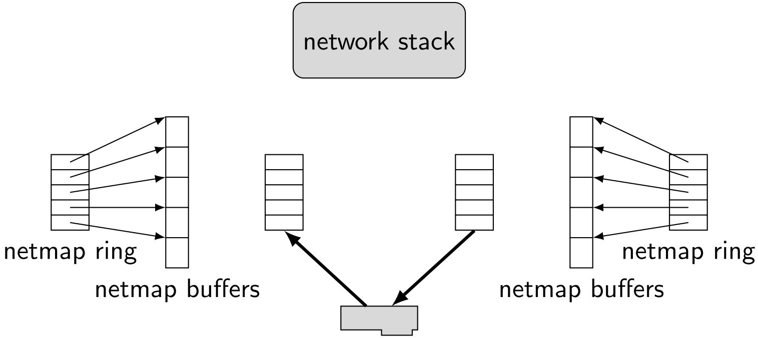
Netmap mode

netmap buffers and rings are allocated in shared memory



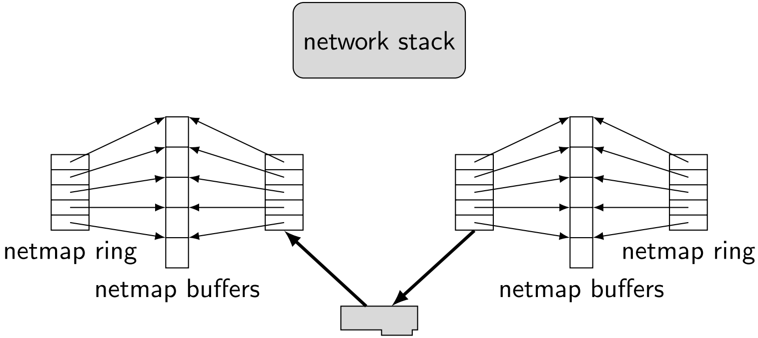
Netmap mode

the netmap rings are pre-filled with netmap buffers



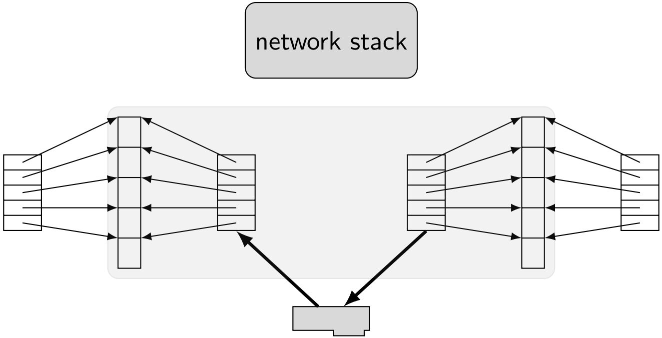
Netmap mode

and the NIC rings are made to share the same buffers



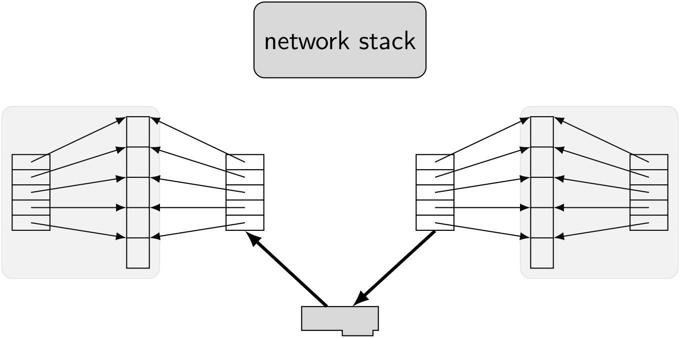
Netmap mode

the NIC only has access to its rings and the netmap buffers



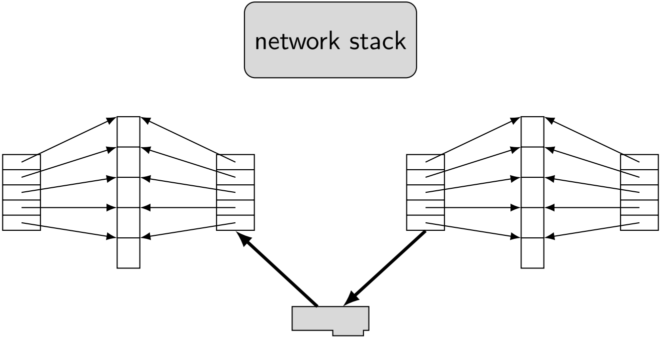
Netmap mode

the netmap application has access to the netmap rings and buffers



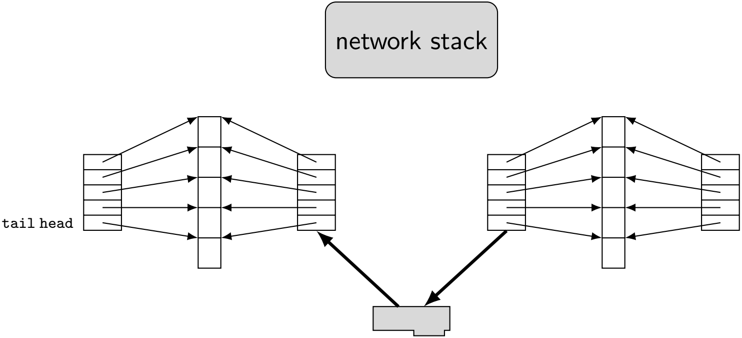
Netmap mode

the netmap rings also have *head* and *tail* pointers. Netmap applications may access the slots and buffers in [`head`, `tail`)



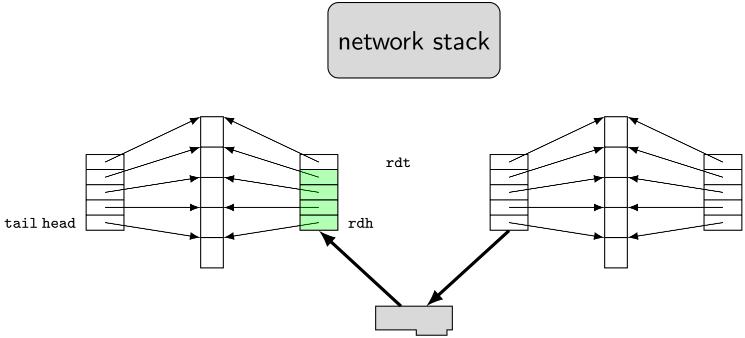
Netmap mode

For the RX ring, this interval contains *new packets*. Initially, it is empty. . .



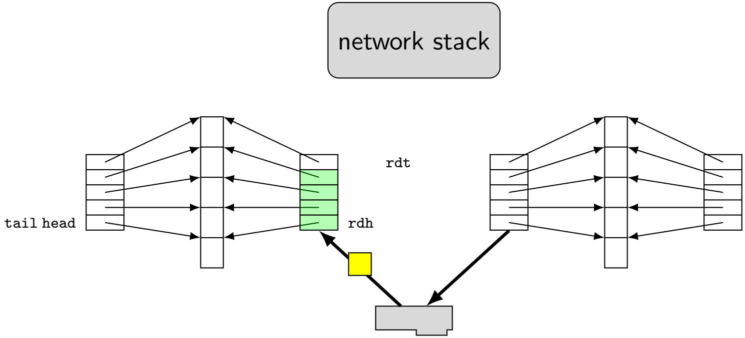
Netmap mode

. . . and all buffers belong to the NIC



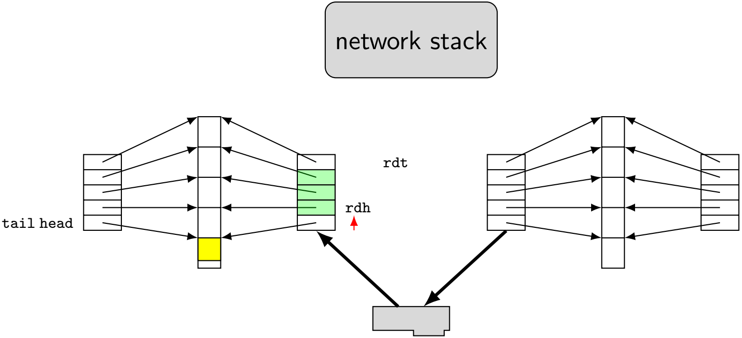
Netmap mode

Assume a packet is received by the NIC



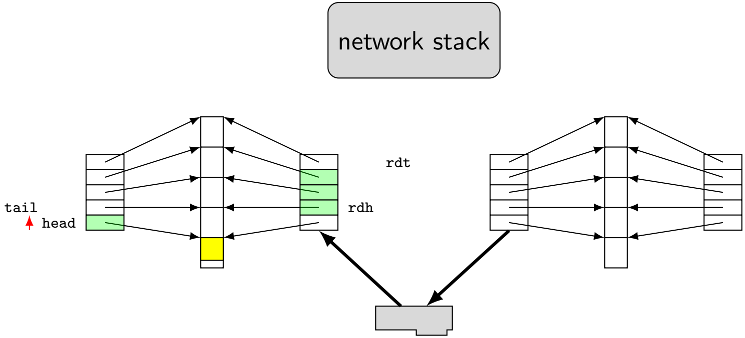
Netmap mode

The NIC copies it into the first available netmap buffer and notifies the *netmap application*



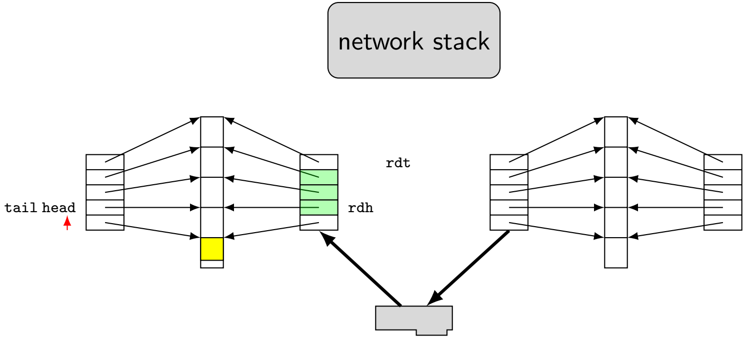
Netmap mode

When the netmap application orders a *ring sync*, the `tail` pointer is updated to reflect the new state



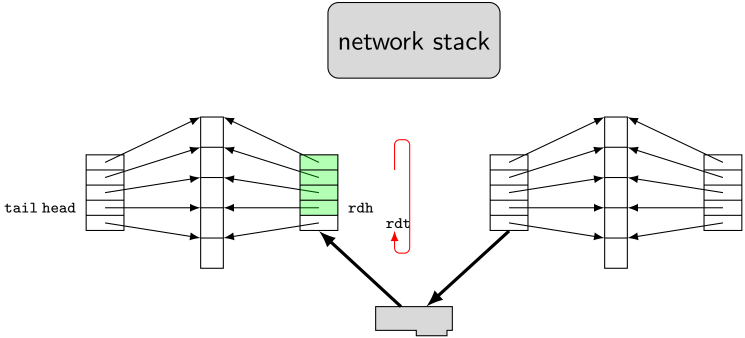
Netmap mode

Now the netmap application may read the new message. When it is done, it moves `head`



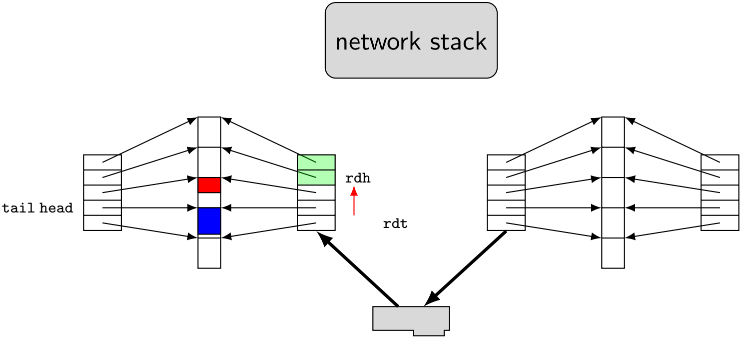
Netmap mode

The next time that it orders a ring sync, the NIC tail pointer is updated



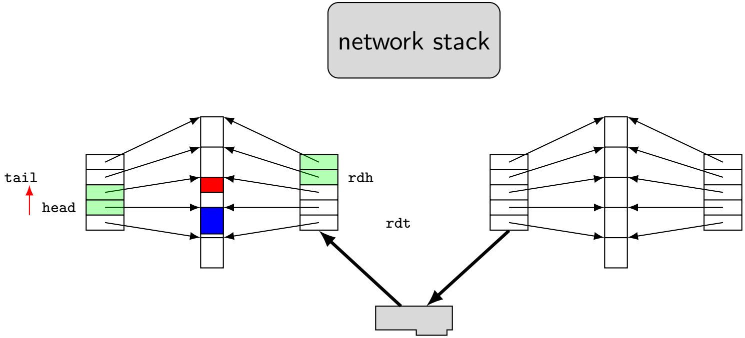
Netmap mode

Batching is possible: assume two new packets arrive



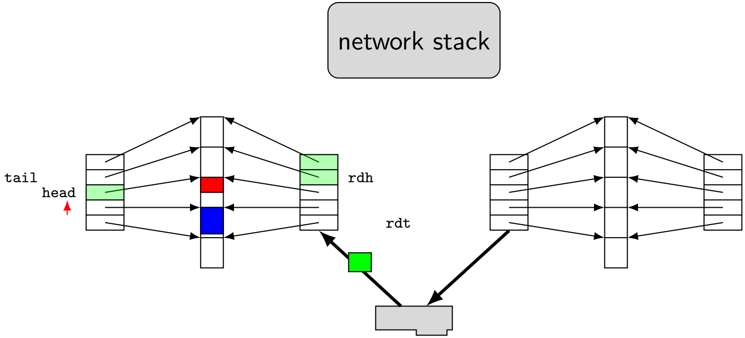
Netmap mode

The netmap application orders a sync and `tail` now reveals both packets



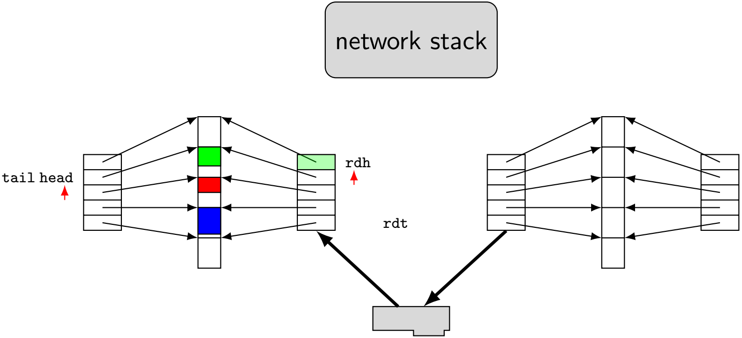
Netmap mode

While the application processes the new packets, other may arrive



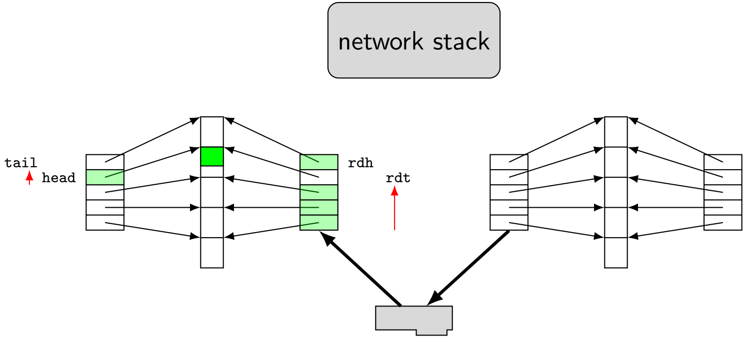
Netmap mode

While the application processes the new packets, other may arrive



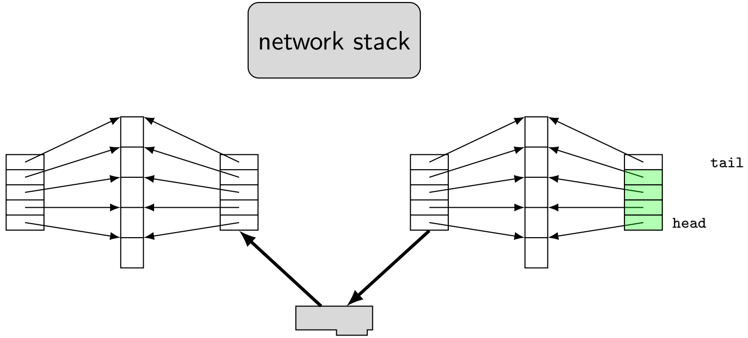
Netmap mode

When the application orders a sync again, both `tail` and `rdt` are updated



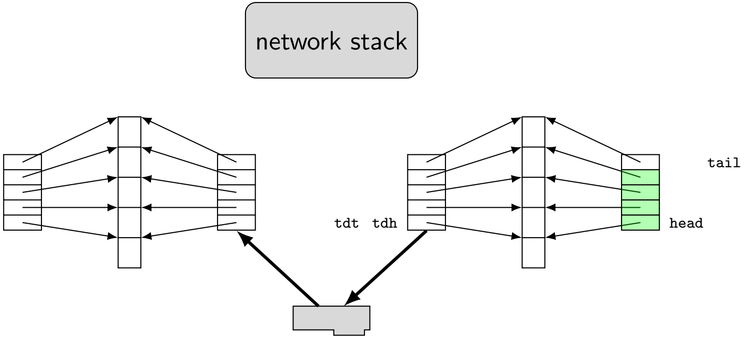
Netmap mode

For the TX ring, the [`head`, `tail`) interval contains *empty buffers*. Initally, it is full.



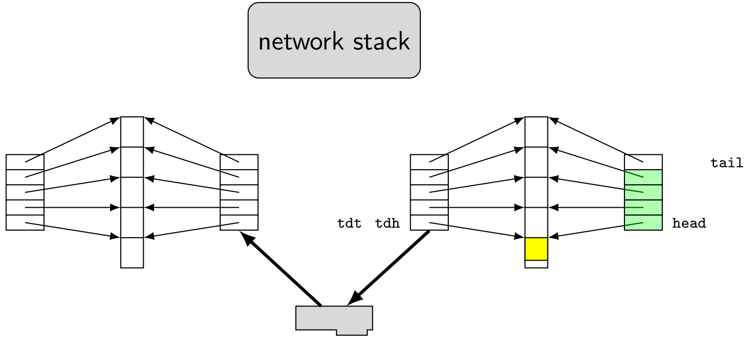
Netmap mode

The NIC has initially no buffers



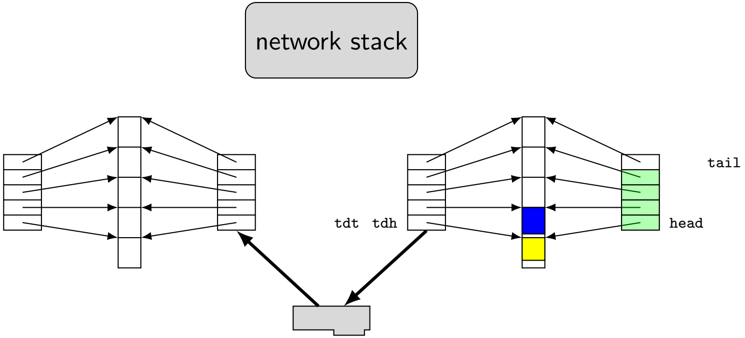
Netmap mode

The netmap application may prepare several packets in the empty buffers. . .

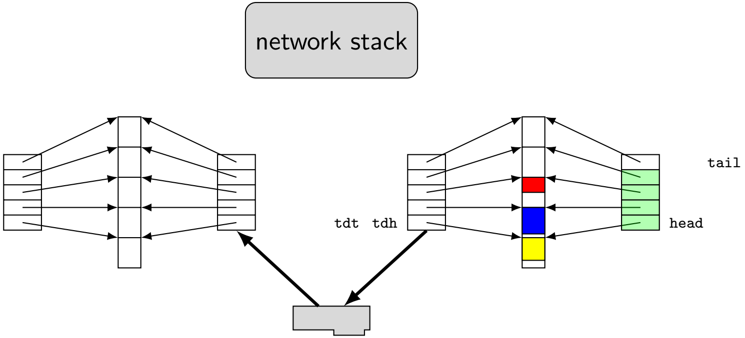


Netmap mode

The netmap application may prepare several packets in the empty buffers. . .

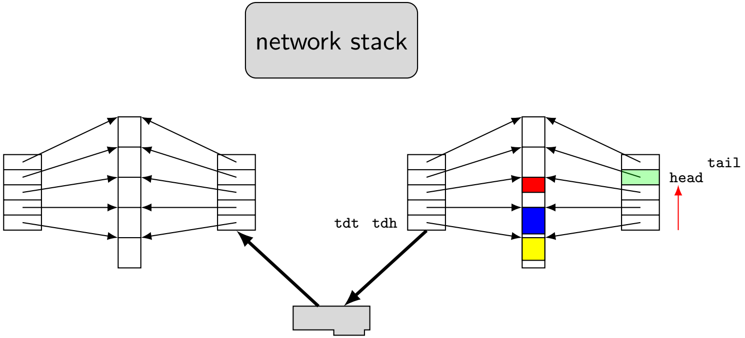


The netmap application may prepare several packets in the empty buffers. . .



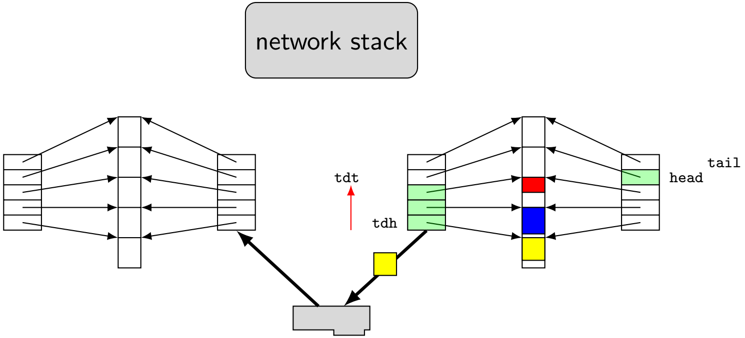
Netmap mode

The netmap application may prepare several packets in the empty buffers. . .



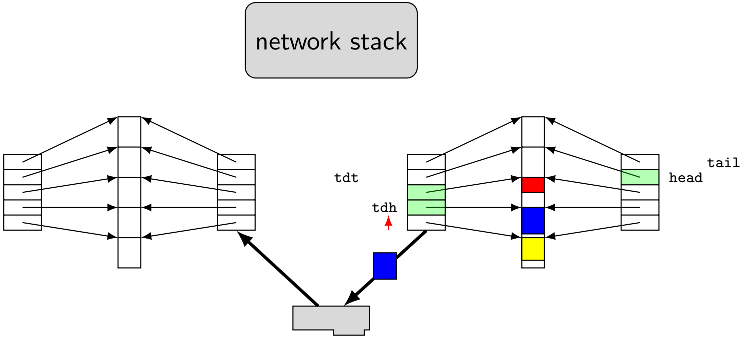
Netmap mode

. . . and then orders a sync. The kernel will update `tdt`, therefore notifying the NIC, which will start transmitting



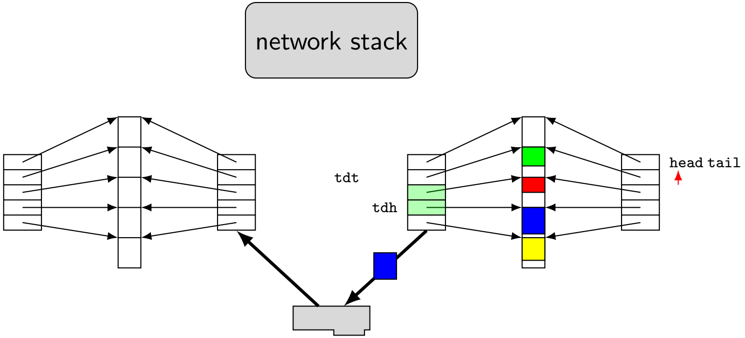
Netmap mode

. . . and then orders a sync. The kernel will update `tdt`, therefore notifying the NIC, which will start transmitting



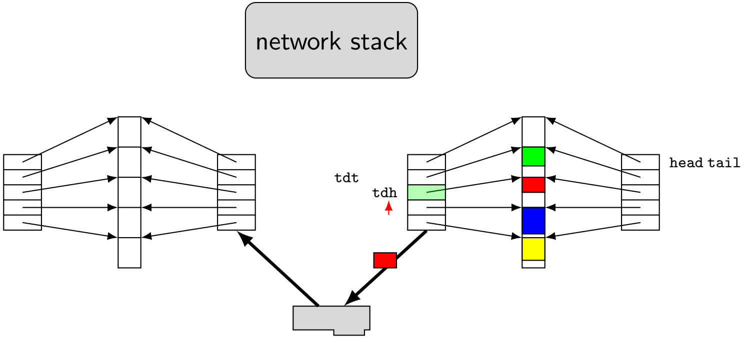
Netmap mode

meanwhile, the application may prepare new packets



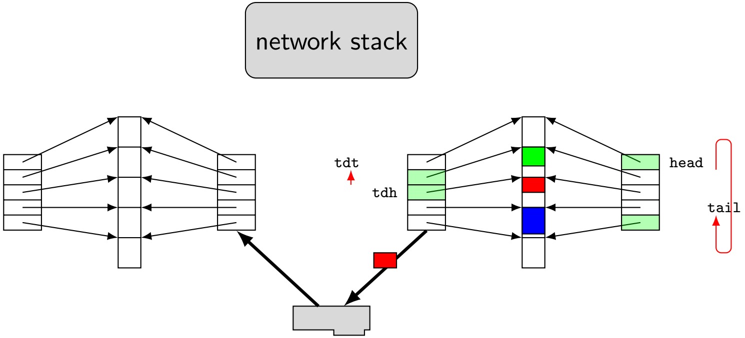
Netmap mode

Assume two packets have been transmitted when the app orders a sync again



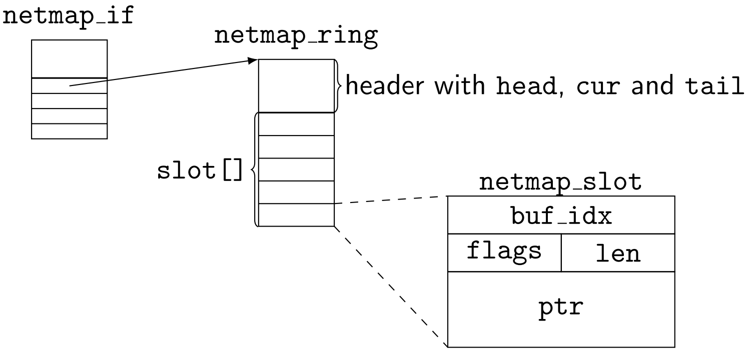
Netmap mode

Now, both `tail` and `tdt` are updated



The netmap user data-structures

Defined (and documented!) in `/usr/include/net/netmap.h`.



What is `cur`?

- Another user-controlled pointer in the ring
- It must lie in [`head`, `tail`]
- It moves past the slots that the application has seen, without returning them to netmap
- In RX rings:
  - hold packets that you have seen but not yet finished to process
- in TX rings:
  - the available slots are not sufficient and you need to wait for more (e.g., you have to send a multi-slot packet)
- in most cases, just let `cur` = `head`.

What is `cur`?

- Another user-controlled pointer in the ring
- It must lie in [`head`, `tail`]
- It moves past the slots that the application has seen, without returning them to netmap
- In RX rings:
  - hold packets that you have seen but not yet finished to process
- in TX rings:
  - the available slots are not sufficient and you need to wait for more (e.g., you have to send a multi-slot packet)
- in most cases, just let `cur` = `head`.

What is `cur`?

- Another user-controlled pointer in the ring
- It must lie in [`head`, `tail`]
- It moves past the slots that the application has seen, without returning them to netmap
- In RX rings:
  - hold packets that you have seen but not yet finished to process
- in TX rings:
  - the available slots are not sufficient and you need to wait for more (e.g., you have to send a multi-slot packet)
- in most cases, just let `cur` = `head`.

What is `cur`?

- Another user-controlled pointer in the ring
- It must lie in [`head`, `tail`]
- It moves past the slots that the application has seen, without returning them to netmap
- In RX rings:
  - hold packets that you have seen but not yet finished to process
- in TX rings:
  - the available slots are not sufficient and you need to wait for more (e.g., you have to send a multi-slot packet)
- in most cases, just let `cur` = `head`.

What is `cur`?

- Another user-controlled pointer in the ring
- It must lie in [`head`, `tail`]
- It moves past the slots that the application has seen, without returning them to netmap
- In RX rings:
  - hold packets that you have seen but not yet finished to process
- in TX rings:
  - the available slots are not sufficient and you need to wait for more (e.g., you have to send a multi-slot packet)
- in most cases, just let `cur` = `head`.

What is `cur`?

- Another user-controlled pointer in the ring
- It must lie in [`head`, `tail`]
- It moves past the slots that the application has seen, without returning them to netmap
- In RX rings:
  - hold packets that you have seen but not yet finished to process
- in TX rings:
  - the available slots are not sufficient and you need to wait for more (e.g., you have to send a multi-slot packet)
- in most cases, just let `cur` = `head`.

What is `cur`?

- Another user-controlled pointer in the ring
- It must lie in [`head`, `tail`]
- It moves past the slots that the application has seen, without returning them to netmap
- In RX rings:
  - hold packets that you have seen but not yet finished to process
- in TX rings:
  - the available slots are not sufficient and you need to wait for more (e.g., you have to send a multi-slot packet)
- in most cases, just let `cur` = `head`.

What is `cur`?

- Another user-controlled pointer in the ring
- It must lie in [`head`, `tail`]
- It moves past the slots that the application has seen, without returning them to netmap
- In RX rings:
  - hold packets that you have seen but not yet finished to process
- in TX rings:
  - the available slots are not sufficient and you need to wait for more (e.g., you have to send a multi-slot packet)
- in most cases, just let `cur` = `head`.

The simplified setup API

Include libs

```
#include <net/netmap.h>
#define NETMAP_WITH_LIBS
#include <net/netmap_user.h>
```

Open a port in netmap mode

```
struct nm_des *nmd =
  nm_open("netmap:eth0", NULL, 0, NULL);
```

Reach your netmap_if

```
  struct netmap_if *nifp = nmd->nifp;
```

The simplified setup API

Include libs

```
#include <net/netmap.h>
#define NETMAP_WITH_LIBS
#include <net/netmap_user.h>
```

Open a port in netmap mode

```
struct nm_des *nmd =
  nm_open("netmap:eth0", NULL, 0, NULL);
```

Reach your `netmap_if`

```
  struct netmap_if *nifp = nmd->nifp;
```

The simplified setup API

Include libs

```
#include <net/netmap.h>
#define NETMAP_WITH_LIBS
#include <net/netmap_user.h>
```

Open a port in netmap mode

```
struct nm_des *nmd =
  nm_open("netmap:eth0", NULL, 0, NULL);
```

Reach your `netmap_if`

```
  struct netmap_if *nifp = nmd->nifp;
```

A first look at nm_open()

```
struct nm_desc *
nm_open(const char *ifname, /* other args */);
```

- use NULL, 0 and NULL for other args
- if ifname is "netmap:*if*" it
  1. puts *if* in netmap mode, if necessary
  2. mmap()s the netmap user structures into the process address space
  3. returns an nm_desc with all the info
- on error it returns NULL and sets errno (errno = 0 means ifname not recognized)

A first look at nm_open()

```
struct nm_desc *
nm_open(const char *ifname, /* other args */);
```

- use NULL, 0 and NULL for other args
- if ifname is "netmap:*if*" it
    1. puts *if* in netmap mode, if necessary
    2. mmap()s the netmap user structures into the process address space
    3. returns an nm_desc with all the info
- on error it returns NULL and sets errno (errno = 0 means ifname not recognized)

A first look at `nm_open()`

```
struct nm_desc *
nm_open(const char *ifname, /* other args */);
```

- use `NULL`, `0` and `NULL` for other args
- if `ifname` is "`netmap:`*if*" it
  1. puts *if* in netmap mode, if necessary
  2. `mmap()`s the netmap user structures into the process address space
  3. returns an `nm_desc` with all the info
- on error it returns `NULL` and sets `errno` (`errno` = 0 means `ifname` not recognized)

A first look at `nm_open()`

```
struct nm_desc *
nm_open(const char *ifname, /* other args */);
```

- use `NULL`, `0` and `NULL` for other args
- if `ifname` is "`netmap:`*if*" it
  1. puts *if* in netmap mode, if necessary
  2. `mmap()`s the netmap user structures into the process address space
  3. returns an `nm_desc` with all the info
- on error it returns `NULL` and sets `errno` (`errno` = 0 means `ifname` not recognized)

A first look at nm_open()

```
struct nm_desc *
nm_open(const char *ifname, /* other args */);
```

- use NULL, 0 and NULL for other args
- if ifname is "netmap:*if*" it
  1. puts *if* in netmap mode, if necessary
  2. mmap()s the netmap user structures into the process address space
  3. returns an nm_desc with all the info
- on error it returns NULL and sets errno (errno = 0 means ifname not recognized)

A first look at nm_open()

```
struct nm_desc *
nm_open(const char *ifname, /* other args */);
```

- use NULL, 0 and NULL for other args
- if ifname is "netmap:*if*" it
  1. puts *if* in netmap mode, if necessary
  2. mmap()s the netmap user structures into the process address space
  3. returns an nm_desc with all the info
- on error it returns NULL and sets errno (errno = 0 means ifname not recognized)

Working with the rings

- Once you have the pointer to netmap_if you can reach the rings:

```
struct netmap_ring *rxring = NETMAP_RXRING(nifp, 0);
struct netmap_ring *rtring = NETMAP_TXRING(nifp, 0);
```

- There may be many rings! Use the {first,last}_{tx,rx}_ring fields in the nmd returned by nm_open()
- there are ring->num_slots slots in the ring->slots[] array
- to obtain the buffer pointed to by a slot of ring:

```
void *buf = NETMAP_BUF(ring, slot->buf_idx);
```

Working with the rings

- Once you have the pointer to `netmap_if` you can reach the rings:

```
struct netmap_ring *rxring = NETMAP_RXRING(nifp, 0);
struct netmap_ring *rtring = NETMAP_TXRING(nifp, 0);
```

- There may be many rings! Use the `{first,last}_{tx,rx}_ring` fields in the `nmd` returned by `nm_open()`
- there are `ring->num_slots` slots in the `ring->slots[]` array
- to obtain the buffer pointed to by a slot of `ring`:

```
void *buf = NETMAP_BUF(ring, slot->buf_idx);
```

Working with the rings

- Once you have the pointer to `netmap_if` you can reach the rings:

  ```
  struct netmap_ring *rxring = NETMAP_RXRING(nifp, 0);
  struct netmap_ring *rtring = NETMAP_TXRING(nifp, 0);
  ```

- There may be many rings! Use the `{first,last}_{tx,rx}_ring` fields in the `nmd` returned by `nm_open()`
- there are `ring->num_slots` slots in the `ring->slots[]` array
- to obtain the buffer pointed to by a slot of `ring`:

  ```
  void *buf = NETMAP_BUF(ring, slot->buf_idx);
  ```

Working with the rings

- Once you have the pointer to `netmap_if` you can reach the rings:

```
struct netmap_ring *rxring = NETMAP_RXRING(nifp, 0);
struct netmap_ring *rtring = NETMAP_TXRING(nifp, 0);
```

- There may be many rings! Use the `{first,last}_{tx,rx}_ring` fields in the `nmd` returned by `nm_open()`
- there are `ring->num_slots` slots in the `ring->slots[]` array
- to obtain the buffer pointed to by a slot of `ring`:

```
void *buf = NETMAP_BUF(ring, slot->buf_idx);
```

Sync-ing the rings: non-blocking

RX rings

```
ioctl(nmd->fd, NIOCRXSYNC);
```

Sync all registered RX rings;

TX rings

```
ioctl(nmd->fd, NIOCTXSYNC);
```

Sync all registered TX rings;

Can be used to implement busy-polling.

Sync-ing the rings: non-blocking

RX rings

```
ioctl(nmd->fd, NIOCRXSYNC);
```

Sync all registered RX rings;

TX rings

```
ioctl(nmd->fd, NIOCTXSYNC);
```

Sync all registered TX rings;

Can be used to implement busy-polling.

Sync-ing the rings: non-blocking

RX rings

```
ioctl(nmd->fd, NIOCRXSYNC);
```

Sync all registered RX rings;

TX rings

```
ioctl(nmd->fd, NIOCTXSYNC);
```

Sync all registered TX rings;

Can be used to implement busy-polling.

Other useful functions

- `nm_ring_next(struct netmap_ring *, uint32_t)`: increment ring pointers with wrap around
- `nm_ring_empty(struct netmap_ring *)`: true iff there are no new packets (RX ring) or slots available for sending (TX ring)
- `nm_ring_space(struct netmap_ring *)`: number of slots available for sending (TX ring) or number of received packets (RX ring)
- `nm_close(struct nm_desc *)`: pass the `nmd` returned by `nm_open()` to clean up and free it

First example: busy-polling sink

Write a netmap applications that

- accepts two arguments from the command line:
  - the name of a netmap port
  - an UDP port number
- it opens the netmap port and counts the received UDP packets with the given port number

Boilerplate code already available in

```
codelab/sink.c
```

Sync-ing the rings: blocking

using `poll()`

```
struct pollfd pfd = {
    .fd = nmd->fd,
    .events = POLLIN /* and/or POLLOUT */
};

int i = poll(pfd, 1, timeout);
```

- sync the registered empty RX rings if `POLLIN`
- sync on the registered TX rings if `POLLOUT` *OR* there are packets to send
- if all rings are empty (`cur` = `tail`), block
- Note: no need to sync again when `poll()` returns!

Sync-ing the rings: blocking

using poll()

```
struct pollfd pfd = {
    .fd = nmd->fd,
    .events = POLLIN /* and/or POLLOUT */
};

int i = poll(pfd, 1, timeout);
```

- sync the registered empty RX rings if `POLLIN`
- sync on the registered TX rings if `POLLOUT` *OR* there are packets to send
- if all rings are empty (`cur` = `tail`), block
- Note: no need to sync again when `poll()` returns!

Sync-ing the rings: blocking

using poll()

```
struct pollfd pfd = {
    .fd = nmd->fd,
    .events = POLLIN /* and/or POLLOUT */
};

int i = poll(pfd, 1, timeout);
```

- sync the registered empty RX rings if `POLLIN`
- sync on the registered TX rings if `POLLOUT` *OR* there are packets to send
- if all rings are empty (`cur` = `tail`), block
- Note: no need to sync again when `poll()` returns!

Sync-ing the rings: blocking

using poll()

```
struct pollfd pfd = {
    .fd = nmd->fd,
    .events = POLLIN /* and/or POLLOUT */
};

int i = poll(pfd, 1, timeout);
```

- sync the registered empty RX rings if `POLLIN`
- sync on the registered TX rings if `POLLOUT` *OR* there are packets to send
- if all rings are empty (`cur` = `tail`), block
- Note: no need to sync again when `poll()` returns!

Second example: blocking sink

Modify the first example to use `poll()` instead of `ioctl()`.

Polling on several ports and mixed directions

- You can `nm_open()` several ports and `poll()` all of them in a single call
- Don't `poll()` needlessly, or you fall back to busy wait
- TX is subtle:
  - you need to `POLLOUT` only if you are out of TX slots
  - but also to start TX as a side effect
  - in default mode, TX will start also if you `POLLIN`
  - (latter can be disabled by passing the `NETMAP_NO_TX_POLL` flag to `nm_open()`)

Polling on several ports and mixed directions

- You can `nm_open()` several ports and `poll()` all of them in a single call
- Don't `poll()` needlessly, or you fall back to busy wait
- TX is subtle:
  - you need to `POLLOUT` only if you are out of TX slots
  - but also to start TX as a side effect
  - in default mode, TX will start also if you `POLLIN`
  - (latter can be disabled by passing the `NETMAP_NO_TX_POLL` flag to `nm_open()`)

Polling on several ports and mixed directions

- You can `nm_open()` several ports and `poll()` all of them in a single call
- Don't `poll()` needlessly, or you fall back to busy wait
- TX is subtle:
  - you need to `POLLOUT` only if you are out of TX slots
  - but also to start TX as a side effect
  - in default mode, TX will start also if you `POLLIN`
  - (latter can be disabled by passing the `NETMAP_NO_TX_POLL` flag to `nm_open()`)

Polling on several ports and mixed directions

- You can `nm_open()` several ports and `poll()` all of them in a single call
- Don't `poll()` needlessly, or you fall back to busy wait
- TX is subtle:
  - you need to `POLLOUT` only if you are out of TX slots
  - but also to start TX as a side effect
  - in default mode, TX will start also if you `POLLIN`
  - (latter can be disabled by passing the `NETMAP_NO_TX_POLL` flag to `nm_open()`)

Polling on several ports and mixed directions

- You can `nm_open()` several ports and `poll()` all of them in a single call
- Don't `poll()` needlessly, or you fall back to busy wait
- TX is subtle:
  - you need to `POLLOUT` only if you are out of TX slots
  - but also to start TX as a side effect
  - in default mode, TX will start also if you `POLLIN`
  - (latter can be disabled by passing the `NETMAP_NO_TX_POLL` flag to `nm_open()`)

Polling on several ports and mixed directions

- You can `nm_open()` several ports and `poll()` all of them in a single call
- Don't `poll()` needlessly, or you fall back to busy wait
- TX is subtle:
    - you need to `POLLOUT` only if you are out of TX slots
    - but also to start TX as a side effect
    - in default mode, TX will start also if you `POLLIN`
    - (latter can be disabled by passing the `NETMAP_NO_TX_POLL` flag to `nm_open()`)

Polling on several ports and mixed directions

- You can `nm_open()` several ports and `poll()` all of them in a single call
- Don't `poll()` needlessly, or you fall back to busy wait
- TX is subtle:
  - you need to `POLLOUT` only if you are out of TX slots
  - but also to start TX as a side effect
  - in default mode, TX will start also if you `POLLIN`
  - (latter can be disabled by passing the `NETMAP_NO_TX_POLL` flag to `nm_open()`)

Third example: forward

Write a netmap application that receives two netmap port names and an UDP destination port number from the command line. Then the application

- forwards from the first to the second port all the UDP packets with the given destination port
- drops all other packets
- if the destination port is 0, it forwards all packets

Boilerplate in

```
codelab/forward.c
```

Zero-copy

- just swap the buffers between the slots of two rings
- whenever you change a `buf_idx` of a slot, remember to set the `NS_BUF_CHANGED` flag in the slot `flags`

Only possible if rings and buffers are in the same *netmap memory region*

Zero-copy

- just swap the buffers between the slots of two rings
- whenever you change a `buf_idx` of a slot, remember to set the `NS_BUF_CHANGED` flag in the slot `flags`

Only possible if rings and buffers are in the same *netmap memory region*

Zero-copy

- just swap the buffers between the slots of two rings
- whenever you change a `buf_idx` of a slot, remember to set the `NS_BUF_CHANGED` flag in the slot `flags`

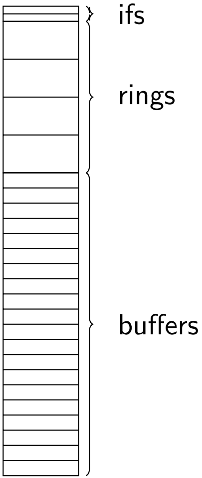
Only possible if rings and buffers are in the same *netmap memory region*

Zero-copy

- just swap the buffers between the slots of two rings
- whenever you change a `buf_idx` of a slot, remember to set the `NS_BUF_CHANGED` flag in the slot `flags`

Only possible if rings and buffers are in the same *netmap memory region*

What are netmap memory regions?

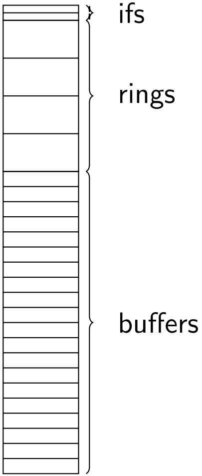


- system memory pre-allocated by netmap
- may be shared by many ports
- there may be more than one region

Beware

you always get access to *entire* regions, not just to the resources allocated by your request.

What are netmap memory regions?

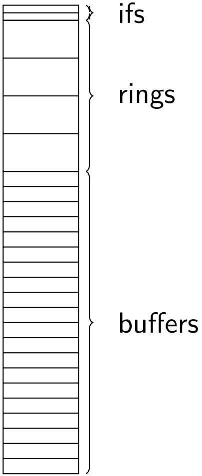


- system memory pre-allocated by netmap
- may be shared by many ports
- there may be more than one region

Beware

you always get access to *entire* regions, not just to the resources allocated by your request.

What are netmap memory regions?

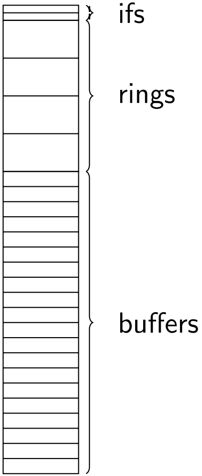


- system memory pre-allocated by netmap
- may be shared by many ports
- there may be more than one region

Beware

you always get access to *entire* regions, not just to the resources allocated by your request.

What are netmap memory regions?

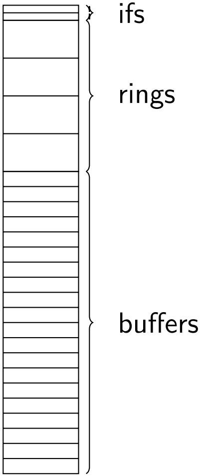


- system memory pre-allocated by netmap
- may be shared by many ports
- there may be more than one region

Beware

you always get access to *entire* regions, not just to the resources allocated by your request.

What are netmap memory regions?



- system memory pre-allocated by netmap
- may be shared by many ports
- there may be more than one region

Beware

you always get access to *entire* regions, not just to the resources allocated by your request.

Memory regions and ports

- when a port is put in netmap mode, netmap selects a memory region for the user data-structures
- legacy behaviour: by default all “hardware” ports use the same (global) region
- PROS of sharing: zero-copy
- CONS: no protection!
- the user may select a different region

Memory regions and ports

- when a port is put in netmap mode, netmap selects a memory region for the user data-structures
- legacy behaviour: by default all “hardware” ports use the same (global) region
- PROS of sharing: zero-copy
- CONS: no protection!
- the user may select a different region

Memory regions and ports

- when a port is put in netmap mode, netmap selects a memory region for the user data-structures
- legacy behaviour: by default all "hardware" ports use the same (global) region
- PROS of sharing: zero-copy
- CONS: no protection!
- the user may select a different region

Memory regions and ports

- when a port is put in netmap mode, netmap selects a memory region for the user data-structures
- legacy behaviour: by default all “hardware” ports use the same (global) region
- PROS of sharing: zero-copy
- CONS: no protection!
- the user may select a different region

Memory regions and ports

- when a port is put in netmap mode, netmap selects a memory region for the user data-structures
- legacy behaviour: by default all “hardware” ports use the same (global) region
- PROS of sharing: zero-copy
- CONS: no protection!
- the user may select a different region

nm_open() and memory regions

first port

```
nmd1 = nm_open("netmap:eth0", NULL, 0, NULL);
```

second port

```
nmd2 = nm_open("netmap:eth1", NULL,
        NM_OPEN_NO_MMAP, nmd1);
```

The two ports are in the same region iff

```
nmd1->mem == nmd2->mem
```

nm_open() and memory regions

first port

```
nmd1 = nm_open("netmap:eth0", NULL, 0, NULL);
```

second port

```
nmd2 = nm_open("netmap:eth1", NULL,
         NM_OPEN_NO_MMAP, nmd1);
```

The two ports are in the same region iff

```
nmd1->mem == nmd2->mem
```

nm_open() and memory regions

first port

```
nmd1 = nm_open("netmap:eth0", NULL, 0, NULL);
```

second port

```
nmd2 = nm_open("netmap:eth1", NULL,
        NM_OPEN_NO_MMAP, nmd1);
```

The two ports are in the same region iff

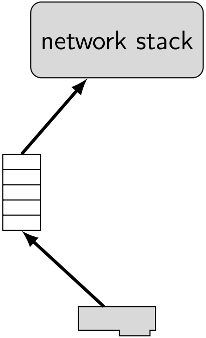
```
nmd1->mem == nmd2->mem
```

Fourth example: zero-copy forward

Modify the application from the previous example to support zero-copy if possibile, and fallback to copy otherwise.

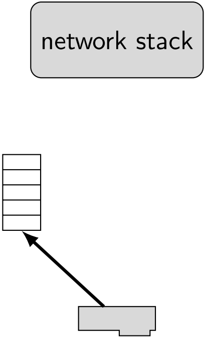
Netmap host rings (RX path)

let us focus on the RX path of the NIC



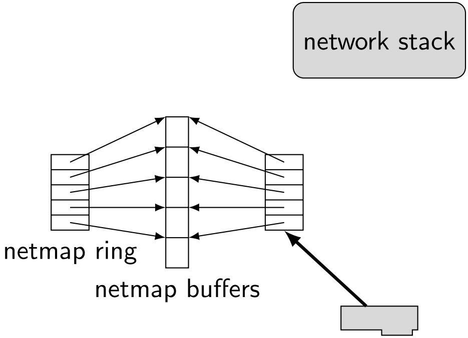
Netmap host rings (RX path)

we have seen that, when we open a NIC in netmap mode, the NIC is disconnected from the host stack



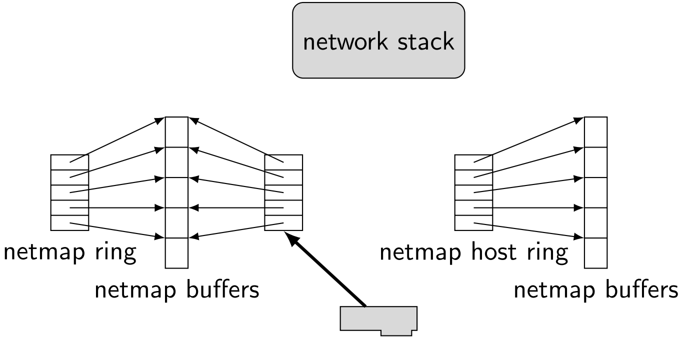
Netmap host rings (RX path)

and netmap rings are allocated and pre-filled with netmap buffers



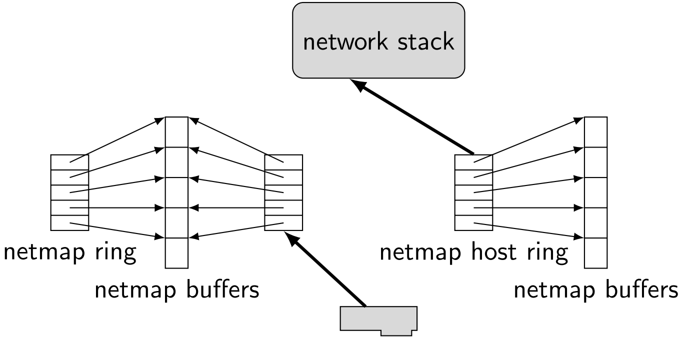
Netmap host rings (RX path)

an additional, software-only TX netmap ring is also allocated and pre-filled with netmap buffers



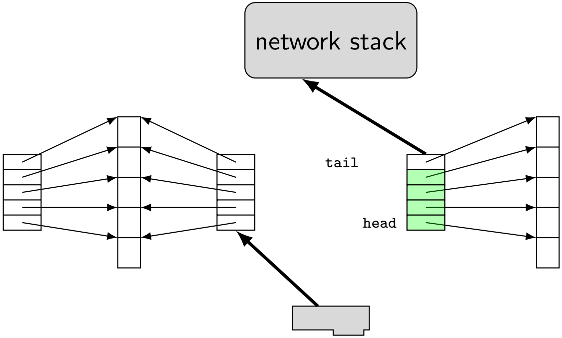
Netmap host rings (RX path)

this ring can be used to inject packets into the host stack, as if they were coming from the NIC



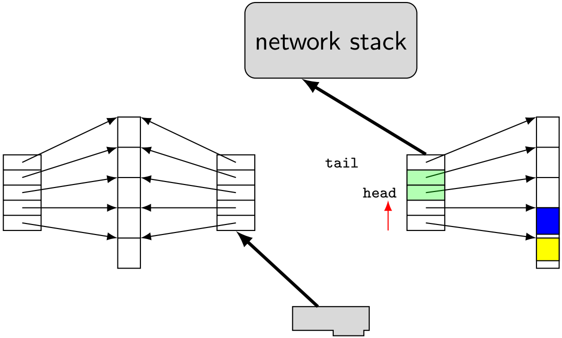
Netmap host rings (RX path)

it can be used as any other netmap TX ring



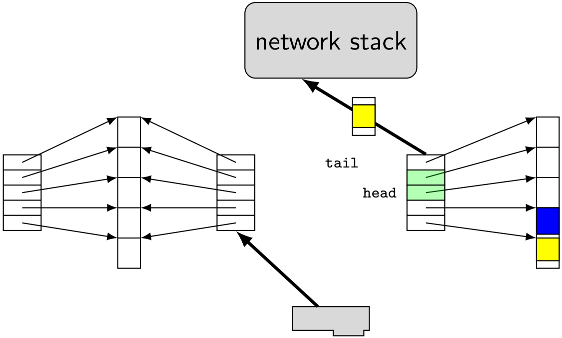
Netmap host rings (RX path)

it can be used as any other netmap TX ring



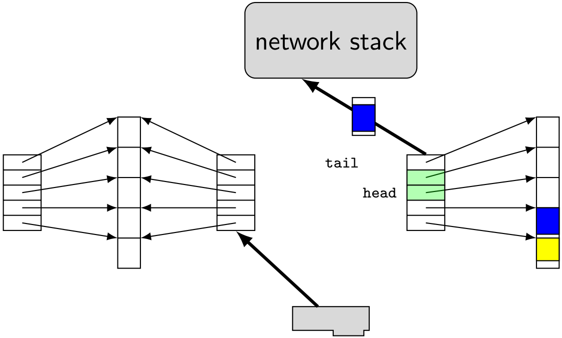
Netmap host rings (RX path)

during sync, netmap will copy each new message in a new skbuf and pushes it up the stack



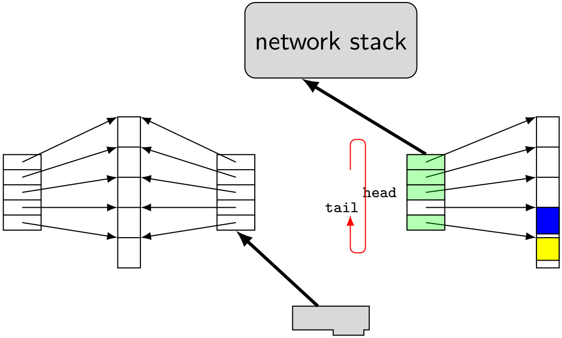
Netmap host rings (RX path)

during sync, netmap will copy each new message in a new skbuf and pushes it up the stack



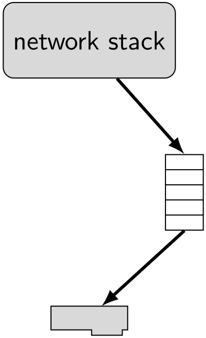
Netmap host rings (RX path)

during sync, netmap will copy each new message in a new skbuf and pushes it up the stack



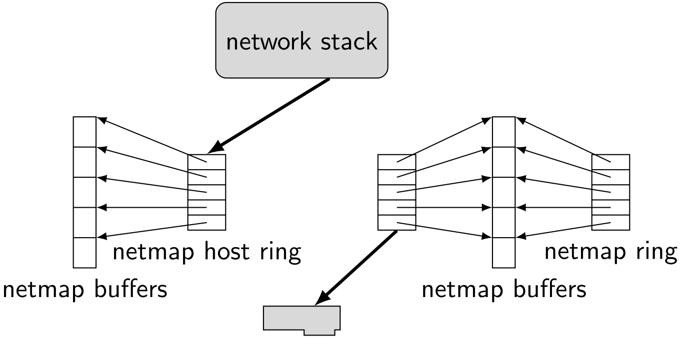
Netmap host rings (TX path)

now let us consider the TX path of the NIC



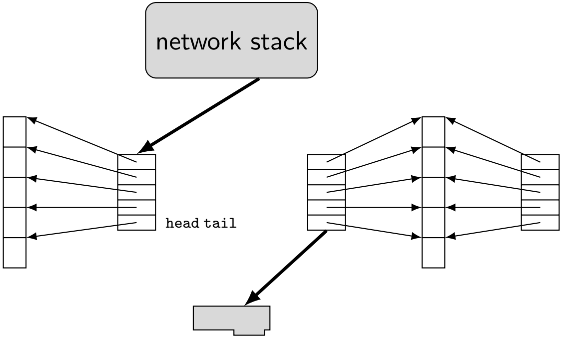
Netmap host rings (TX path)

when the NIC is open in netmap mode the stack TX is redirected to a software-only netmap RX ring



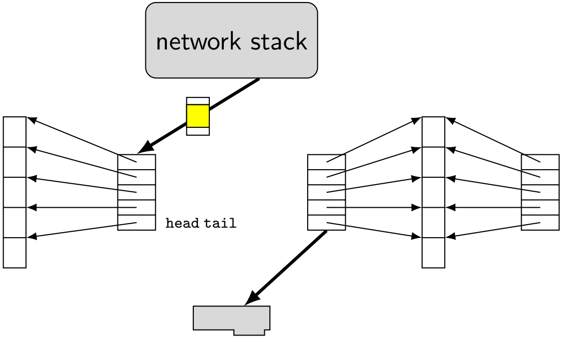
Netmap host rings (TX path)

for netmap applications, it is an RX ring



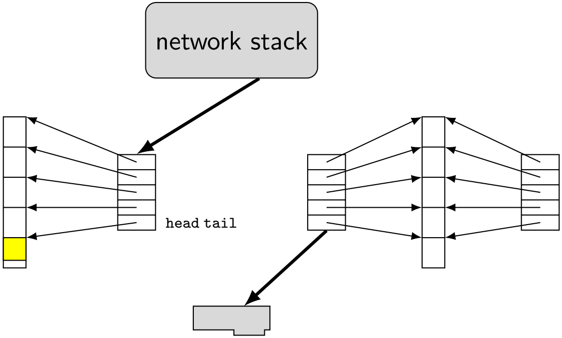
Netmap host rings (TX path)

when the stack wants to send packets through the NIC. . .



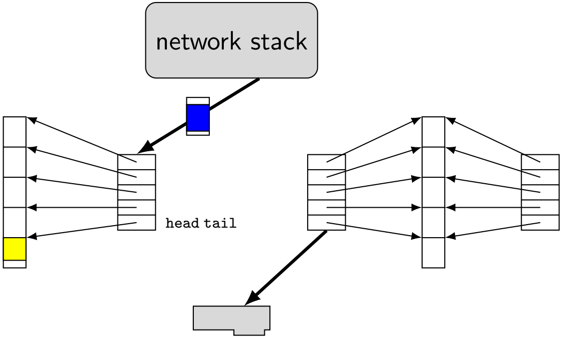
Netmap host rings (TX path)

. . . netmap intercepts the operations and copies the messages into the ring's netmap buffers



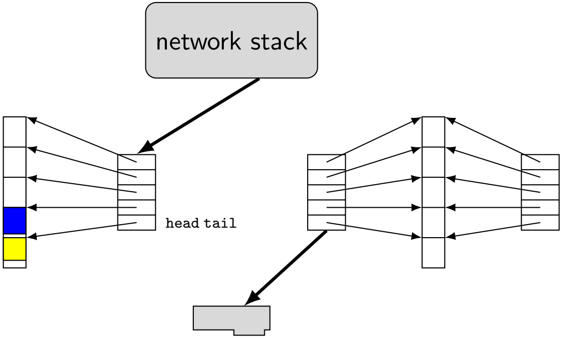
Netmap host rings (TX path)

. . . netmap intercepts the operations and copies the messages into the ring's netmap buffers



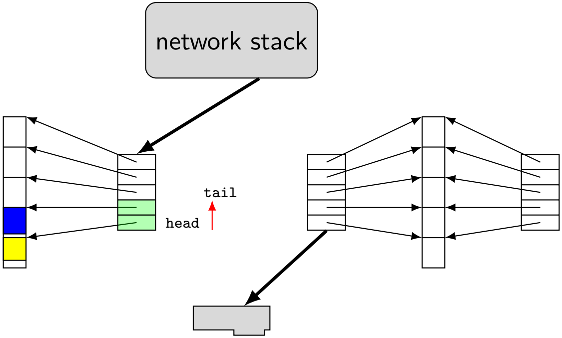
Netmap host rings (TX path)

...netmap intercepts the operations and copies the messages into the ring's netmap buffers



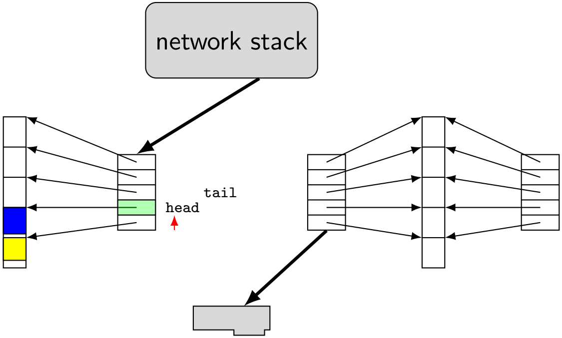
Netmap host rings (TX path)

after sync, the netmap application may then consume the messages



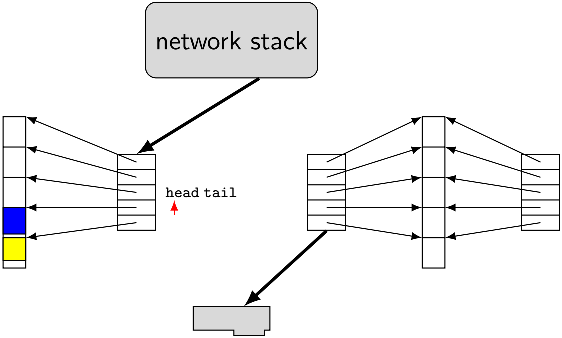
Netmap host rings (TX path)

after sync, the netmap application may then consume the messages



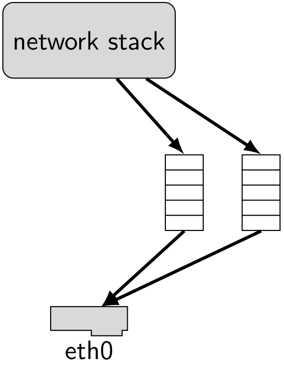
Netmap host rings (TX path)

after sync, the netmap application may then consume the messages



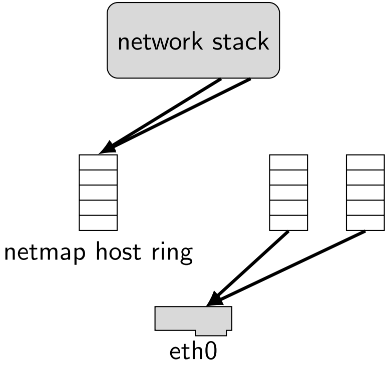
Cards with multiple rings (TX path)

Let us consider a NIC with multiple TX rings



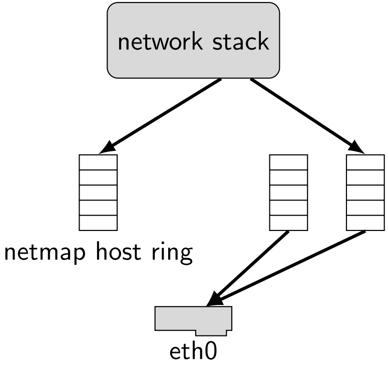
Cards with multiple rings (TX path)

A successful call to `nm_open("netmap:eth0",...)` will redirect everything to the netmap RX host ring



Cards with multiple rings (TX path)

Instead, `nm_open("netmap:eth0-0",...)` will redirect only TX ring 0



Cards with multiple rings (TX path)

Similarly for `nm_open("netmap:eth0-1",...)`