The Need for Aynchronous, Zero-Copy Network I/O

Ulrich Drepper

Red Hat, Inc.

The Problem

Network hardware changed but the socket API stayed the same

- Transfer rates bigger (esp compared to bus and memory speed)
- New forms of NIC interfaces (RDMA, etc)

But:

- sockets provide a byte stream
- CPU speed kept (almost) up with NICs

The Way Forward

- Use asynchronous interfaces

To enable this:

- Efficient event handling

And to support modern hardware:

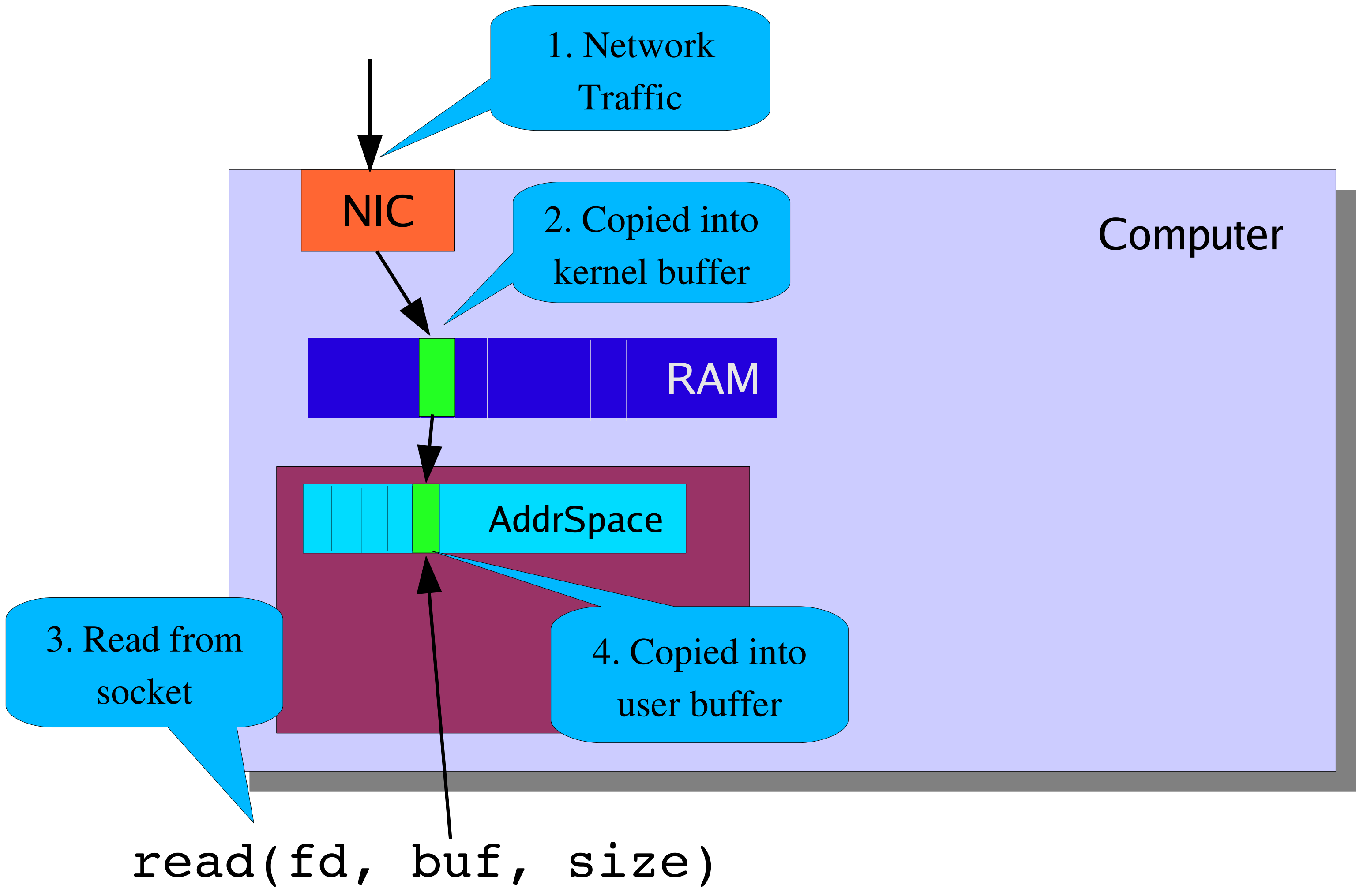
- Direct I/O from/to user buffers

State of the Art

- POSIX asynchronous I/O interfaces
  - Not really for network I/O
- O_ASYNC sockets

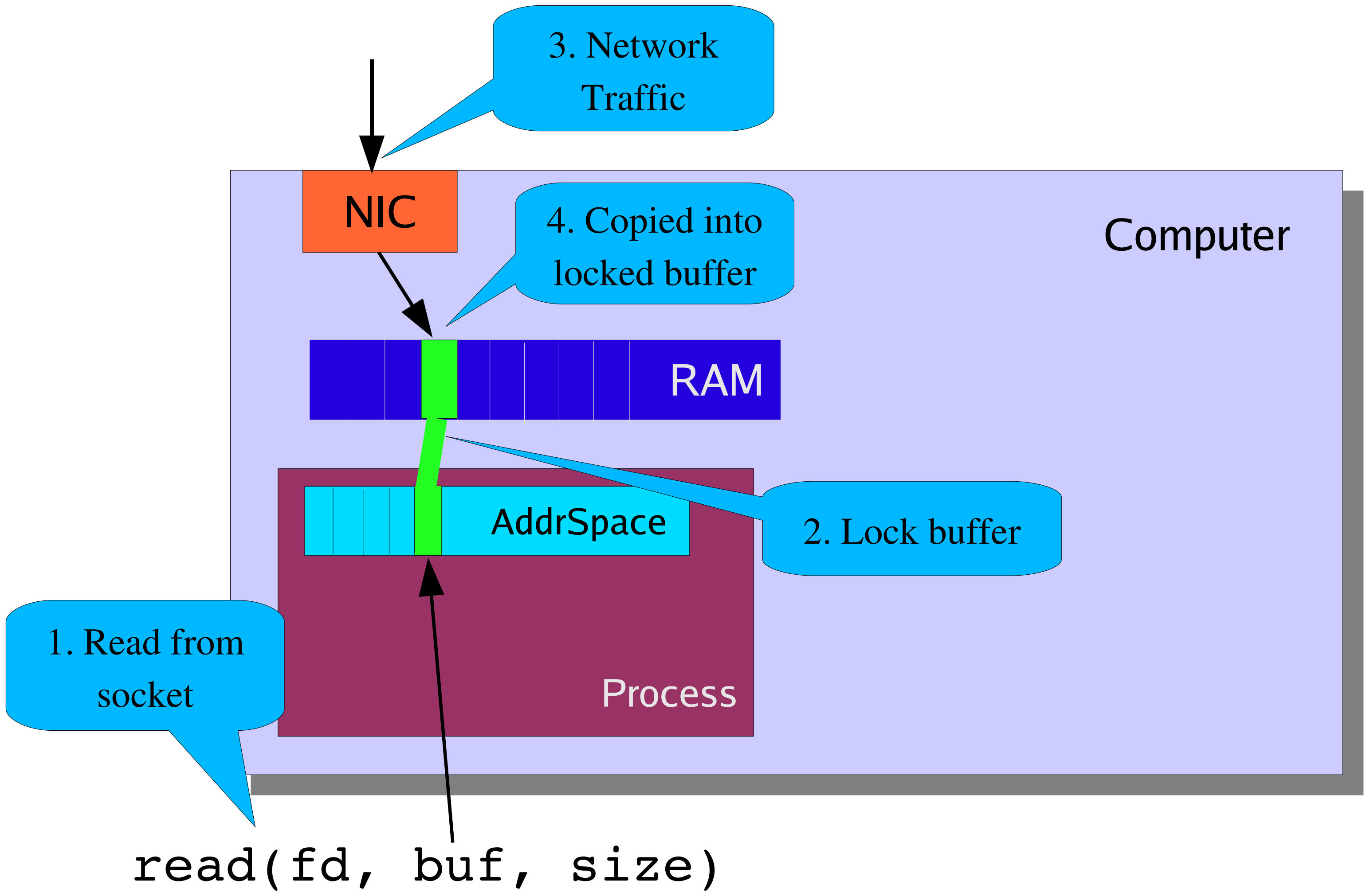
- Common problems:
  - Not really zero copy
  - Cumbersome and/or slow notification

Full Zero Copy



```
read(fd, buf, size)
```

Full Zero Copy



Memory Locking

- Is a privileged operation
- Is expensive
- Only works with page size granularity

☹ Impractical to lock every I/O buffer individually

Proposed Memory Interfaces

- One possibility: add `MAP_DMA` flag
  - Not very flexible

```
int dma_alloc(dma_mem_t *handlep,
  size_t size, unsigned int flags);
int dma_free(dma_mem_t handle, size_t
  size);
```

Current Event Handling

- Pretty efficient interface with `epoll_wait`
  - Works only with file descriptors
  - Does not work with
    - Synchronization primitives
    - Message queues
    - Asynchronous I/O requests
    - Signals
- Ideally: one interfaces to rule them all

Event Handling Solutions

- `SOCK_SEQPACKET` protocol `PF_EVENT`
  - Uniform records of events (big union)
  - Kernel limits number of outstanding events
- Ring buffer in memory provided by program
  - Uniform records of events (again)
  - Size controlled by application
- Kernel can signal overflow out-of-band (signal, ...)
- Better yet...

Event Handling Solutions

Abstract out the user interface:

```
ec_t ec_create(unsigned flags);
int ec_destroy(ec_t ec);
int ec_next_event(ec_t ec, event_data_t *d);

int ec_to_fd(ec_t ec);
```

or

```
int ec_delay(ec_t ec, struct timespec *tout);
```

Using Event Channels

- Register file descriptors, message queue descriptors
  - No changes to existing interfaces
  - Descriptors can be used with multiple event channels and `poll`/`select` simultaneously
- Alternative: introduce separate interfaces specifying event channel to report to

Asynchronous Network I/O, Part 1

- Extend the POSIX asynchronous I/O interfaces
  - Add `msghdr` pointer to `aiocb`
  - Extend `sigevent`
    - Add event channel descriptor
    - Define `SIGEV_EC` to select event channel notification
- New interfaces like

```
int aio_send(struct aiocb *aiocbp, int flags);
```

Asynchronous Network I/O, Part 2

- POSIX AIO does not solve all problems
  - Not always zero copy
  - Memory locking privileges and expenses

Alternative Network AIO

- Directly associate DMA area with socket

```
int dma_assoc(int sock, dma_mem_t mem,size_t size,
  unsigned flags);

int dma_disassoc(int sock, dma_mem_t, size_t
  size);
```

- Get delivery and send data directly from that memory region

DMA Memory Handling

- DMA areas need administration
  - Do not overwrite buffer with received data until program is done with it
  - Do not write into buffer in preparation of sending when incoming data could also be written

```
int sio_reserve(dma_mem_t dma, void **memp off,
  size_t size);
int sio_release(dma_mem_t dma, void *mem, size_t
  size);
```

New Network AIO Interfaces

```
int sio_send(int sock, const void *buf, size_t size, int flags);

int sio_sendto(int sock, const void *buf, size_t size, int flags,

  const struct sockaddr *to, socklen_t tolen);

int sio_sendmsg(int sock, const void *buf, size_t size, int flags);

int sio_recv(int sock, void **buf, size_t size, int flags);

int sio_recvfrom(int sock, const void **buf, size_t size, int flags,

  struct sockaddr *to, socklen_t tolen);

int sio_recvmsg(int sock, const void **buf, size_t size, int flags);
```

Note: receive functions take pointer to a pointer !!!

Questions ?