Dynamic memory in practice

I/O and Network buffers

Herbert Bos
Vrije Universiteit Amsterdam
http://www.cs.vu.nl/~herbertb
The slow way to do it

- Rx data on NIC: Interrupt
  - Driver
    - allocates memory
    - Copies data to buffer
- Network stack process pkt
- TCP copies it to reassemble stream
- When app does read, copy to userland
- Lots of signalling
copy performance
different ways to transfer data
The Linux way
memory for data allocated via kmalloc
The Linux way

- skbuff: a complex, common pkt data structure
- all layers fill in appropriate info
- copying is fairly minimal
- skbufs are cached (see slab caches!) + reused
- quite large and hairy

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sk</td>
<td>pointer to owning socket</td>
</tr>
<tr>
<td>stamp</td>
<td>arrival time</td>
</tr>
<tr>
<td>dev</td>
<td>pointer to receiving/transmitting device</td>
</tr>
<tr>
<td>h</td>
<td>pointer to transport layer header</td>
</tr>
<tr>
<td>nh</td>
<td>pointer to network layer header</td>
</tr>
<tr>
<td>mac</td>
<td>pointer to link layer header</td>
</tr>
<tr>
<td>dst</td>
<td>pointer to dst_entry</td>
</tr>
<tr>
<td>cb</td>
<td>TCP per-packet control information</td>
</tr>
<tr>
<td>len</td>
<td>actual data length</td>
</tr>
<tr>
<td>csum</td>
<td>checksum</td>
</tr>
<tr>
<td>protocol</td>
<td>packet network protocol</td>
</tr>
<tr>
<td>truesize</td>
<td>buffer size</td>
</tr>
<tr>
<td>head</td>
<td>pointer to head of buffer</td>
</tr>
<tr>
<td>data</td>
<td>pointer to data head</td>
</tr>
<tr>
<td>tail</td>
<td>pointer to tail</td>
</tr>
<tr>
<td>end</td>
<td>pointer to end</td>
</tr>
<tr>
<td>destructor</td>
<td>pointer to destruct function</td>
</tr>
</tbody>
</table>
struct sk_buff {
    union {
        ethhdr *eth;  // Ethernet Header
        ipv6hdr *ipv6;  // IPv6 Header
        tcphdr *th;  // TCP Header
    } nh;
    head;  // Start of data
    data;  // Data segment
    tail;  // End of data
    end;  // End of buffer
}

union {
    uchar *raw;  // Raw data
} mac;  // MAC address

union {
    uchar *raw;  // Raw data
} h;  // Header
that is pretty good

• but...
that is pretty good

• but we need to copy across each protection domain boundary
• still a lot of dynamic allocation and de-allocation
• very complex
• sub-optimal!
Can we do better?

- Fbufs (SOSP'93) and IO-Lite (SOSP'99)
Fbufs

- in micro-kernel lots of boundaries (elsewhere also?)
- in practice: one side writes, intermediate read, one side consumes

Figure 1: Layers Distributed over Multiple Protection Domains
Fbufs

- IO by means of Fbufs
  - one or more contiguous VM pages
  - protection domain either allocates an Fbuf or receives it via IPC
  - ADT can be layered on top of Fbufs
Fbufs

- Let us make buffers immutable
- Share by (re-)mapping
- transfer works as follows
Fbuf transfer

1. Allocate an Aggregate Object (Originator)
   (a) Find and allocate a free virtual address range in the originator (per-fbuf)
   (b) Allocate physical memory pages and clear contents (per-page)
   (c) Update physical page tables (per-page)

2. Send Aggregate Object (Originator)
   (a) Generate a list of fbufs from the aggregate object (per-fbuf)
   (b) Raise protection in originator (read only or no access) (per-fbuf)
   (c) Update physical page tables, ensure TLB/cache consistency (per-page)
Fbuf transfer

3. Receive Aggregate Object (Receiver)
   (a) Find and reserve a free virtual address range in the receiver (per-fbuf)
   (b) Update physical page tables (per-page)
   (c) Construct an aggregate object from the list of fbufs (per-fbuf)

4. Free an Aggregate Object (Originator, Receiver)
   (a) Deallocate virtual address range (per-fbuf)
   (b) Update physical page table, ensure TLB/cache consistency (per-page)
   (c) Free physical memory pages if there are no more references (per-page)
Fbuf transfer

- substantial overhead!
Optimisations

• Restricted Dynamic Read Sharing
  – 2 restrictions:
    • only remap from a limited, globally shared range of virtual addresses
    • write access by receiver, or by sender when receiver holds a reference to the fbuf, is illegal → SEGV
Optimisations

• Restricted Dynamic Read Sharing
  – 2 restrictions:
    • only remap from a limited, globally shared range of virtual addresses
    • write access by receiver, or by sender when receiver holds a reference to the fbuf, is illegal → SEGV

3. Receive Aggregate Object (Receiver)

(a) Find and reserve a free virtual address range in the receiver (per fbuf)
(b) Update physical page tables (per-page)
(c) Construct an aggregate object from the list of fbufs (per-fbuf)
Optimisations

- Fbuf Caching
  - exploit locality in IPC: once a PDU has travelled along a certain path, it is likely that more will follow
  - do not de-allocate fbuf, do not remove mappings → just return to originator with W permission
Optimisations

• Fbuf Caching
  – exploit locality in IPC: once a PDU has travelled along a certain path, it is likely that more will follow
  – do not de-allocate fbuf, do not remove mappings → just return to originator with W permission

1. Allocate an Aggregate Object (Originator)
   (a) Find and allocate a free virtual address range in the originator (per-fbuf)
   (b) Allocate physical memory pages and clear contents (per-page)
   (c) Update physical page tables (per-page)
Optimisations

- Fbuf Caching
  - exploit locality in IPC: once a PDU has travelled along a certain path, it is likely that more will follow
  - do not de-allocate fbuf, do not remove mappings → just return to originator with W permission

3. Receive Aggregate Object (Receiver)

  (a) Find and reserve a free virtual address range in the receiver (per-fbuf)
  (b) Update physical page tables (per-page)
  (c) Construct an aggregate object from the list of fbufs (per-fbuf)
Optimisations

• Fbuf Caching
  - exploit locality in IPC: once a PDU has travelled along a certain path, it is likely that more will follow
  - do not de-allocate fbuf, do not remove mappings → just return to originator with W permission

4. Free an Aggregate Object (Originator, Receiver)
   (a) Deallocate virtual address range (per-fbuf)
   (b) Update physical page table, ensure TLB/cache consistency (per-page)
   (c) Free physical memory pages if there are no more references (per-page)
Optimisations

• integrated buffer management/caching
  – let us skip this one for now
Optimisations

• volatile fbufs
  – so far we removed, for safety reasons, W permission from originator, but:
    • not needed for trusted components
    • not needed for some user applications

2. Send Aggregate Object (Originator)

  (a) Generate a list of fbufs from the aggregate object (per-fbuf)

  (b) Raise protection in originator (read only or no access) (per-fbuf)

  (c) Update physical page tables, ensure TLB/cache consistency (per-page)
what remains is pretty fast

Figure 3: Throughput of a single domain boundary crossing
Fbufs and IO-Lite

- Fbufs were later generalised
  - include the page cache
  - Fbufs immutable but aggregate object are not
  - impressive work
problems

• when will it not provide benefits?
beltway buffers

- disadvantages of fbufs
  - fbufs and io-lite introduce a new, slightly more complex API
  - still some VM overhead at runtime
  - different fbufs travelling in same IO path not necessarily contiguous → why is this an issue? TLB!
  - no in-place modifications possible
let us try something new

- no runtime copy or vm overhead
- ring buffers: good for streaming
- a ring may contain many data blocks
- rings permanently mapped
why not a single ring?

- protection
- multicore sharing $\rightarrow$ cache conflicts (same is true for meta-data)
- embedded devices may have their own rings
- data transformation
separate data and metadata

- data buffers can be shared (Unix file permissions)
- Ibufs
  - provide private view
  - minimise copying
separate data and metadata

- mappings are permanent
- open, close, read, write on buffers
- we cover many protection domains
separate data and metadata

- Dbufs: can be large → good for TLB
- Ibufs: small, contiguous: good for caches
different protection domains

- zero-copy TCP reassembly
- classifier makes life easier and faster
zero-copy reassembly
optimisations

under the hood many optimisations possible

- e.g. different buffer types

![Diagram showing different buffer types: fixed-size slots buffer (p-DBuf), delimiter (v-DBuf), metadata in separate ring (d-DBuf), no delimiters (c-DBuf, handled by lBuf).]
optimisations

under the hood many optimisations possible

- e.g. different buffer types
optimisations

under the hood many optimisations possible

- e.g. adapt buffer size
optimisations

under the hood many optimisations possible

• e.g., peek in addition to read
optimisations

under the hood many optimisations possible

• splicing
optimisations

under the hood many optimisations possible

• splicing
even with a small-scale project like beltway

- performance increase over Linux is significant
  - pipes: 2x
  - splicing: 3x-30x
  - libpcap: 10x
question

- why do we have sockets?
pipesfs

- future: anycore
  - large monolithic programs perhaps not the way to go
- unify IO processing (UNIX-style)
  - use common programs to manipulate IO path
  - grep, sed, compress, cp, cat, etc.
- increase IO throughput
pipesfs

- virtual file system representing kernel IO paths
- pipes
- all data streams between kernel filters accessible from user space
- manipulated from user space
  - mkdir
  - ln -s ...
  - cat /pipes/.../http/get/all | compress > log.Z
another example: scrub http requests

#!/bin/sh
mkdir /pipes/httpclean
mv /pipes/[...]/http/get /pipes/httpclean/
cat /pipes/[...]/http/all | grep -v '..'
    > /pipes/httpclean/all
• do we *really* need sockets or do we have them for lack of something better?
Thank you!