Programming like it's 2007

Varnish – a 2007 software design

Poul-Henning Kamp

phk@FreeBSD.org

Varnish Cheat-Sheet

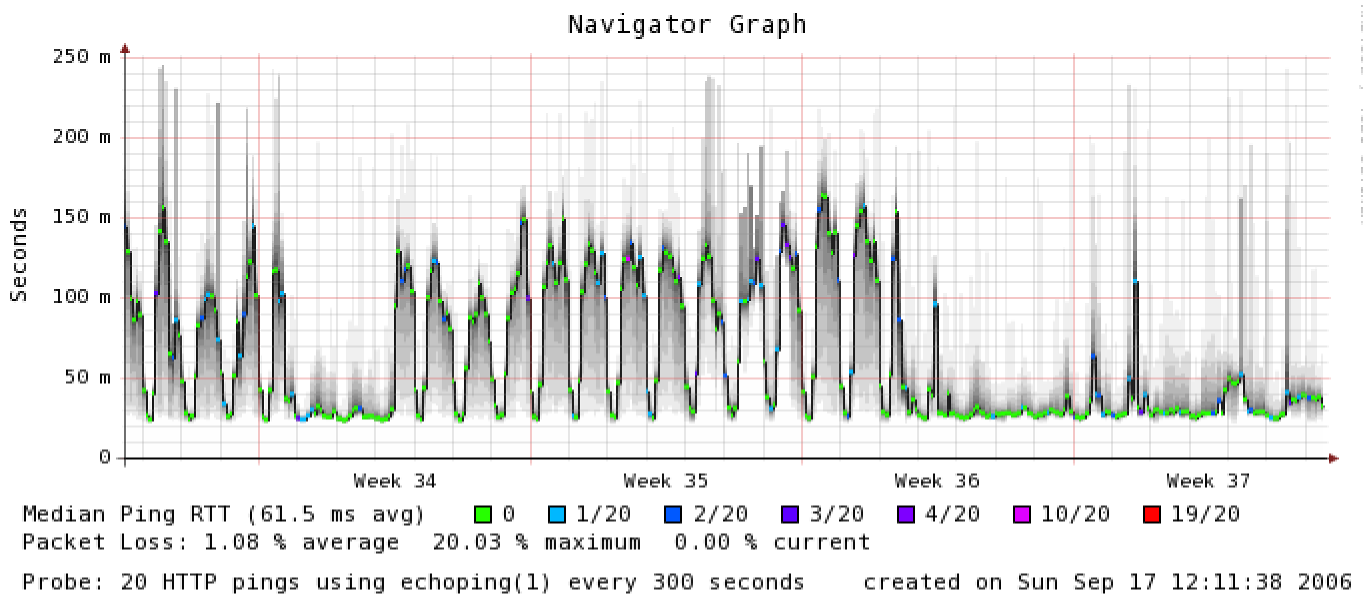
- Web-accelleration for slow CMS systems
- Narrow focus on server side speedup
  - No FTP etc.
  - Content provider features
- High Performance
  - 32 & 64bit, large RAM, sendfile, accept filters
  - SMP/Multicore friendly architecture
  - 2006 software design
  - 11 syscall + 7 locks per cache hit

Varnish is fast!

- 1000 cache hits @ 100 Mbit/sec:

Sample	Min	Max	Median	Average	Stddev
Full	12,9	3001	16,2	71,1	338,5
90% fractile	12,9	26	15,9	16,3	1,7

(all times are in microseconds)



var·nish (vär′nĭsh)

n.

1. **a.** A paint containing [...]

tr.v. **var·nished**, **var·nish·ing**, **var·nish·es**

1. To cover with varnish.

2. To give a smooth and glossy finish to.

3. To give a deceptively attractive appearance to; gloss over.

Varnish Coordinates

- **www.varnish-cache.org**
  - Yes, we need to work on the web-page.
- Coded by: Poul-Henning Kamp
  - phk@FreeBSD.org
- Runs on: Any resonable modern UNIX
  - FreeBSD, Linux etc.
- Open Source
  - BSD license

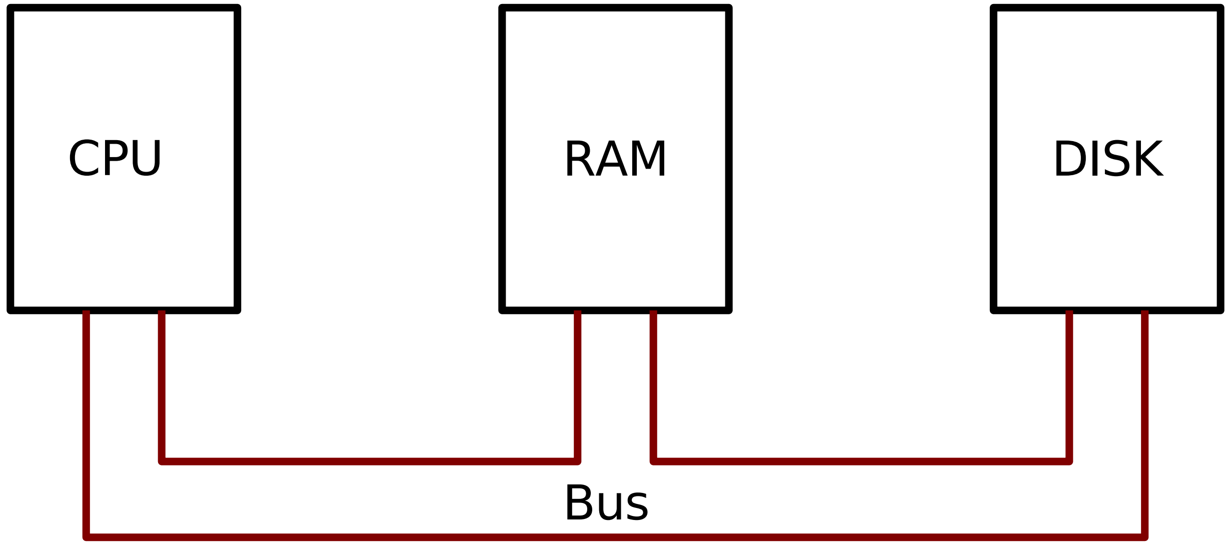
Why I wrote Varnish

- I was sceptical when VG approached me with the Varnish Project.
  - Not really my specialty
  - Not really my preferred area
- On the other hand...
  - I'm tired of people writing lousy programs and blaming it on ”my” kernel
  - A fine chance to educate by example
- Also, I could use the money.

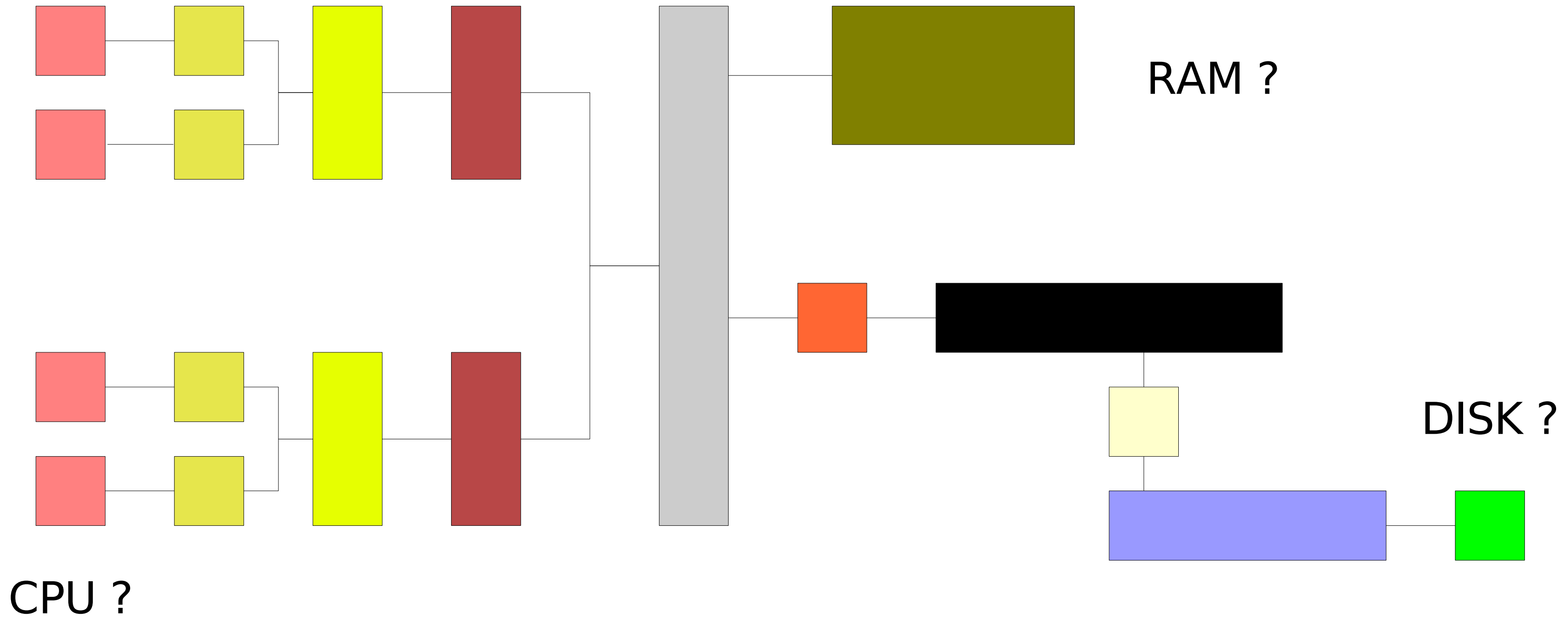
Performance Programming

- Understand and exploit the envelope of your requirements
- Avoid unecessary work
- Use few cheap operations
- Use even fewer expensive operations
- Don't fight the kernel
- Use the features the nice kernel guys have given you

”A computer consists of a CPU, internal storage and external storage”

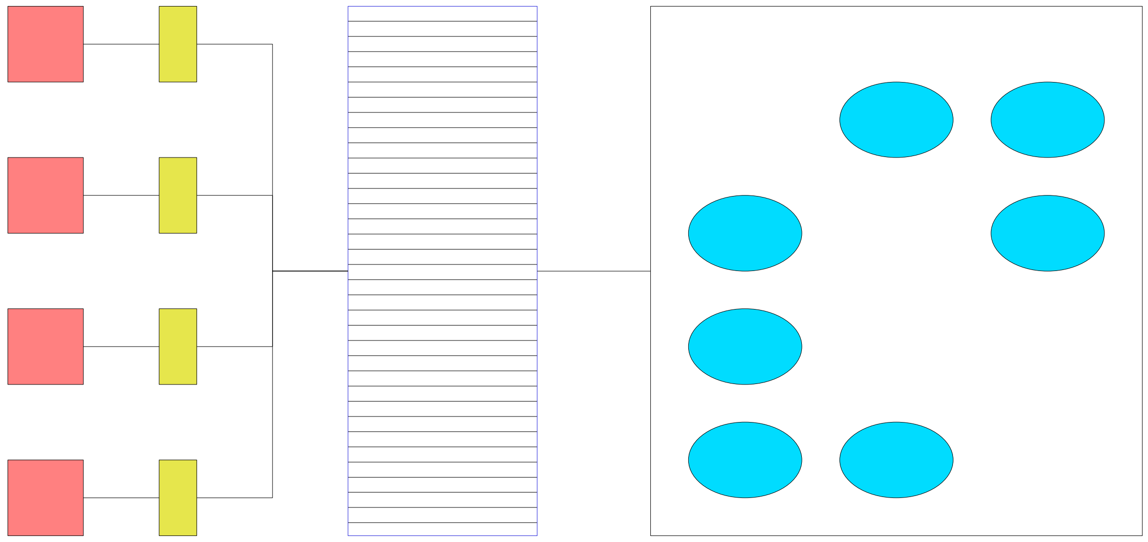


”A Computer consists of a number of processing units (”cores”) connected via a number of caches and busses to a virtualized storage.”



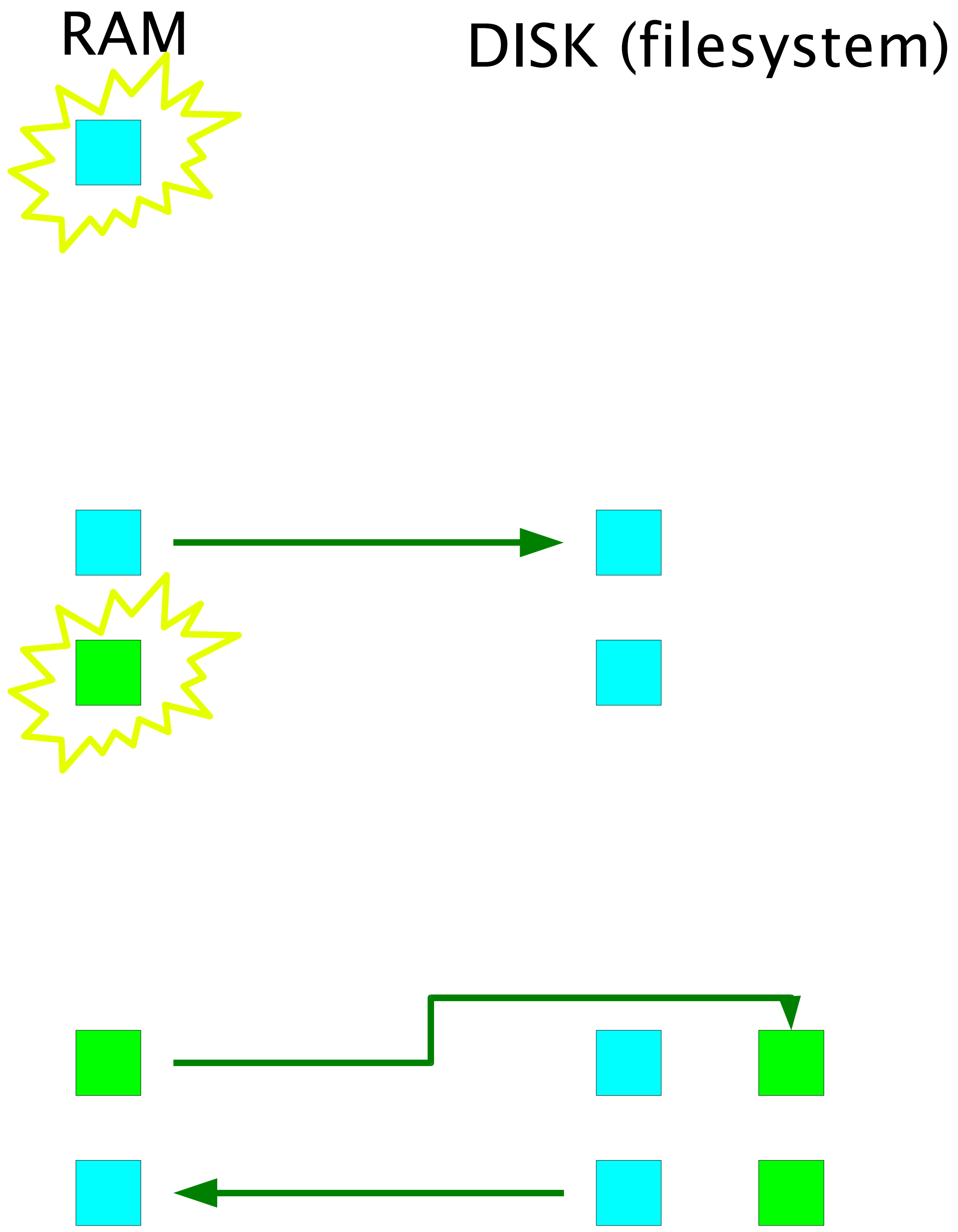
A usable conceptual model

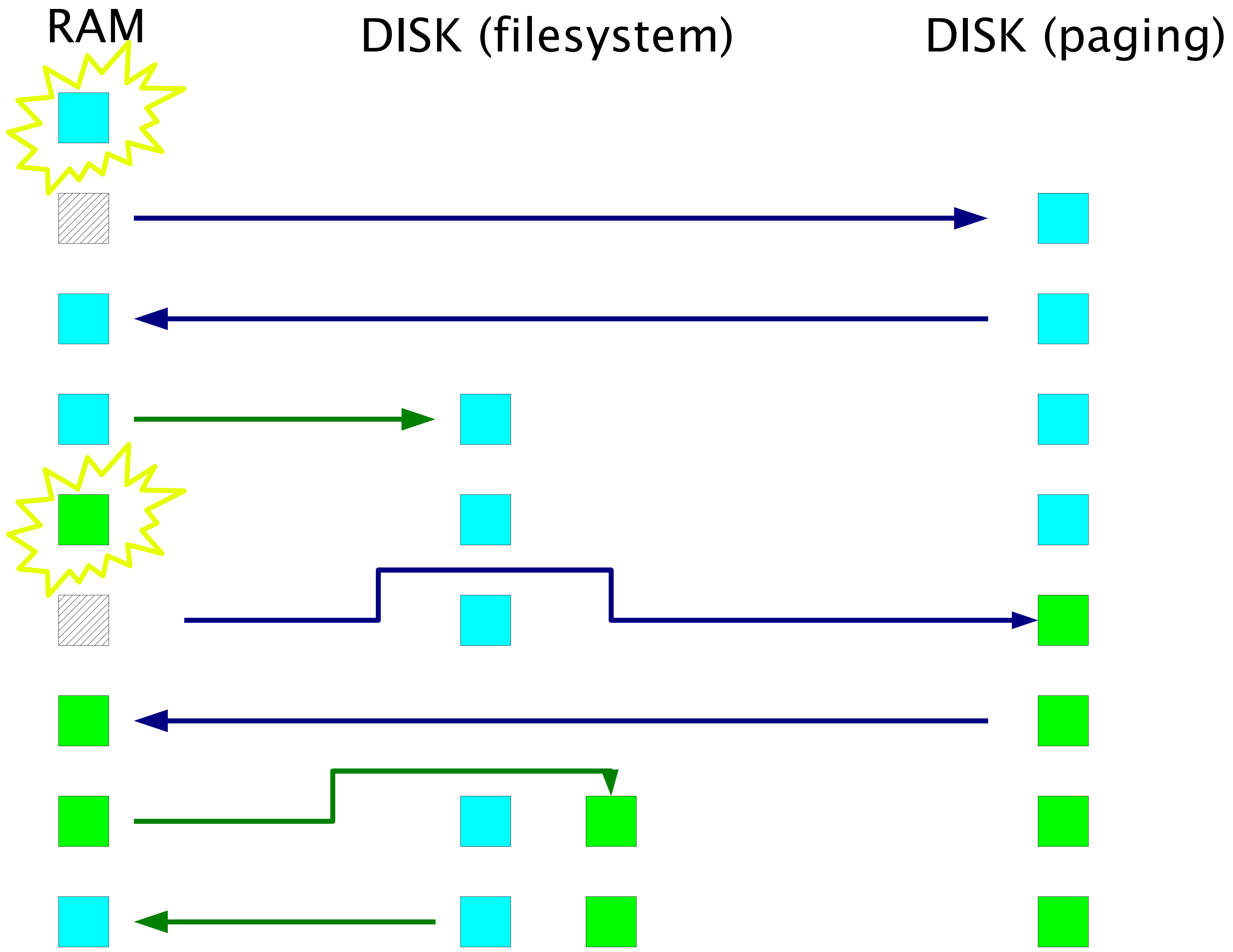
”A computer consists of a number of cores with private caches, a shared, page granularity, cache for external storage objects.”

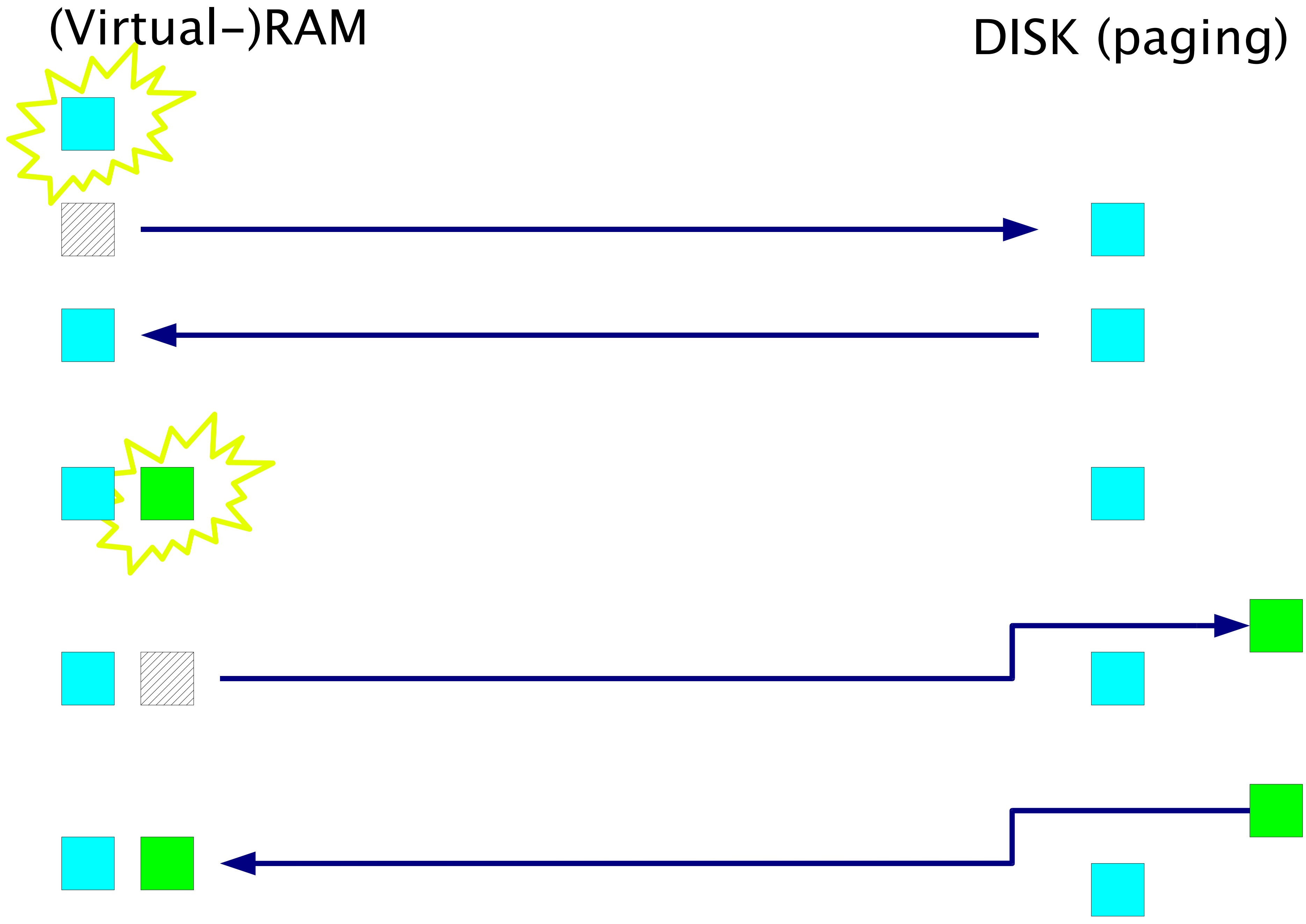


Squid is 1970 model

- Some objects ”in Memory”
- Other objects ”on disk”
- Explicit transfer between the two states:
  - Unused memory object gets written to disk
  - Disk object gets read to memory for transmission.


RAM
DISK (filesystem)


RAM
DISK (filesystem)
DISK (paging)


(Virtual-)RAM
DISK (paging)

Fighting the kernel from userland is plain stupid...

I know what you're thinking, proc:

'Did he send SIGHUP or SIGTERM ?'

Well, to tell you the truth, I've forgotten myself in all this excitement.

But being as this is SIGKILL, the most powerful signal in UNIX, and would blow your memory clean, you've got to ask yourself a question:

'Do I feel lucky?'

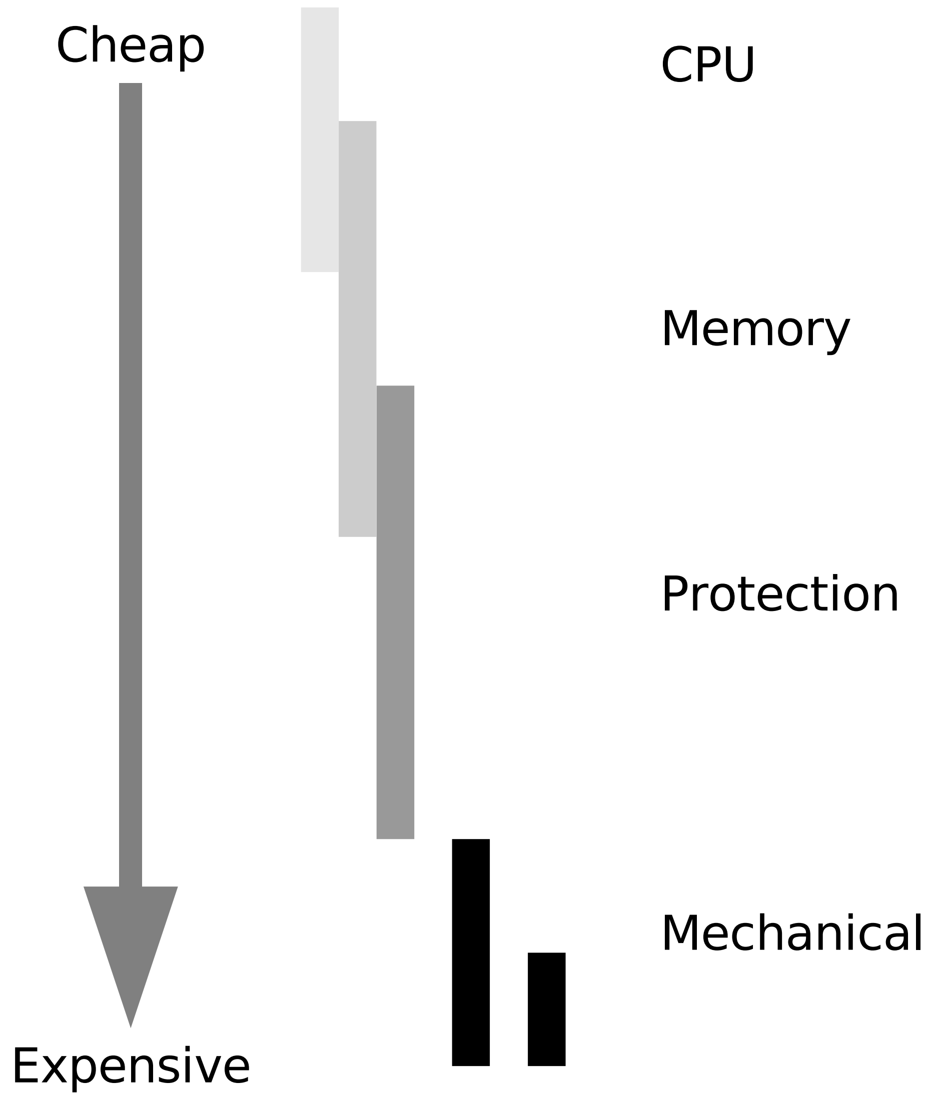
Well, do ya, proc?

Virtual memory shortage

- On 32 bit systems VM is limited to ~2GB
- If you have a webserver with a larger working set of content, you can afford to buy a new 64 bit CPU.

Performance price List

- char *p += 5;
- strlen(p);
- memcpy(p, q, l);
- Locking
- System Call
- Context Switch
- Disk Access
- Filesystem operation



How Cheap or Expensive ?

- CPU clock: 4 Ghz
- Multiple instructions per clock cycle
- Several layers of cache

- Conservative rule of thumb:

1 Billion instructions per second

Cost of a system call ?

- Context switch to kernel is expensive.
  - Thousands of clocks to save/restore registers
  - Argument validation
  - Page Table modifications

- 1 µsec = 0.000001s = 1,000 instructions

Cost of disk I/O

- Enter kernel
- Navigate disk layout
  - Filesystem, partitions etc.
- Via device driver to hardware
- Busses, handshaking etc.
- Mechanical movement.
- 1 msec= 0.001s = 1,000,000 instructions

Varnish logging

- Logging to shared memory
  - Practically Free (as in peanuts)
  - No slowdown for the real workload.
- Generate ”real” output in separate processes, which ”tail” the shmlog:
  - Apache format
  - Custom format
  - Realtime views

Logging

- Logging to a file is expensive:

```
FILE *flog;

flog = fopen("/var/log/mylog", "a");

[...]

fprintf(flog, "%s Something went wrong with %s\n",
    timestamp(), myrepresentation(object));
fflush(flog);
```

Filesystem operation

Once only

Disk I/O

Millons of calls

Cheap logging

- Shared memory logging is cheap:

```
char *logp, *loge;

fd = open(...);
logp = mmap(..., size);
loge = logp + size;

[...]
LOCK(&shmlog);
logp[1] = LOG_ERROR;
logp[2] = sprintf(logp + 3,
    "Something went bad with %s",
    myrepresentation(obj));
logp[3 + logp[2]] = LOG_END;
logp[0] = LOG_ENTRY;
logp += 3 + logp[2];
UNLOCK(&shmlog);
```

Filesystem ops

Once only

Memory and arithmetic

Millions of calls.

But it can be even cheaper

- strlen(3) and ASCIIZ format is not cheap
- Keep track of both ends of strings
- Use memcpy(3) instead of strcpy(3)
  - memcpy(3) can move data in 32 or even 64 bit chunks.
  - strcpy(3) **has** to use 8 bit chunks.

But it can be **even** cheaper

- Add a shmlog buffer to worker threads
- Logging to this buffer is lockless
- Batch ”commit” local buffer to ”real” shmlog at synchronization points.
  - Cost: One lock + one memcpy().

- => Practically no mutex contention.

...and

- Coredumps contains the most recent log records
- We can afford to log much more detail, it does not take up disk-space
- Approx 30 records per request
  - =>100.000's records/sec

”varnishtop”

- Example of ad-hoc shmlog reader:
- ./varnishtop -i sessionclose
- Why do sessions close ?

```
132232.12 timeout
 39002.59 EOF
  5232.97 not HTTP/1.1
  3069.28 Connection: close
  1383.40 remote closed
   630.93 silent
    15.49 pipe
     9.15 Not in cache.
     6.36 Purged.
     5.41 no request
     0.98 Bad Request
```

Memory Strategy

- How long is a HTTP header ?
  - 80 char ? 256 char ? 2048 char ?
- Classic strategy:
  - Allocate small, extend with realloc(3)
- Realloc(3) to more memory is expensive
  - Needs memory copy 99% of the times.

Memory Strategy

Special offer!

Virtually **FREE** memory

offer good from 1980 onwards

Memory Strategy

Allocate enough memory for 99% of the cases up front

Unused & untouched memory is free

- as in ”peanuts”

Long lived data ? - trim back with realloc(3)

Memory management

- During the handling of a request we need various bits of memory
- Use a private pool
  - Keeps things close together (good caching)
  - Avoids locking against other CPUs
- Free everything with a single pointer assignment at the end
  - As cheap as can be

Memory/Lock interaction

- Do minimum work while holding lock
- Allocate memory before grabbing lock
- If you didn't need it anyway:

  - cache it until next time

Preallocation

Malloc(3)

May need to synchronize between CPUs with locks

May need to get memory from kernel with syscall

```
LOCK(storage_lock);
TAILQ_FOREACH(...) {
        if (...)
                break;  /* found */
}
if (ptr == NULL) {
        ptr = malloc(sizeof *wp->store);
        assert(wp->store != NULL);
        [...] // fill ptr->stuff
        TAILQ_INSERT_TAIL(...)
}
UNLOCK(storage_lock)
```

Preallocation

```
if (wp->store == NULL) {
        wp->store = malloc(sizeof *wp->store);
        assert(wp->store != NULL);
}

LOCK(storage_lock);
TAILQ_FOREACH(...) {
        if (...)
                break;  /* found */
}
if (ptr == NULL) {
        ptr = wp->store;
        wp->store = NULL
        [...] // fill ptr->stuff
        TAILQ_INSERT_TAIL(...)
}
UNLOCK(storage_lock)
```

malloc call moved out of locked code section.

Lock held shorter time.

Avoid unnecessary work

- Don't copy data when we can avoid it
  - Receive headers into one chunk of memory and leave them there
- Don't text-process all HTTP headers
  - We only care about few of them
- Transport headers by reference
  - Use scatter/gather I/O (see writev(2) syscall).

Avoid strlen(3)

- `find_hdr(sp->http, ”Host:”)`
  - find_hdr() must do strlen(3) on literal constant to find out how long it is.
- `find_hdr(sp->http, ”Host:”, 5)`
  - Better, but programmer has to get it right.

-
  ```
  #define HTTP_HOST ”\005Host:”
  find_hdr(sp->http, HTTP_HOST);
  ```
  - Even better, programmer must only get it right once.

Remember the caches

- Reuse most recently used resource.
  - It is more likely to be in cache.
- Process, Thread, Memory, File, Directory, Disk-block etc.
- Avoid round-robin scheduling.
  - Use ”Last-In-First-Out” (ie: stack)
  - Not ”First-In-First-Out” (ie: queue)

Measure & Benchmark

- Understand how often things happen
  - Use tcov(1) or add counters.
- Check your assumptions
  - Use ktrace(8) to see the system-calls.
- Study system statistics
  - Netstat(8), Vmstat(8), sysctl(8)
- Time things if in doubt
  - use clock_gettime(3) or RDTSC()