Memory Management
A View for Software Engineers

Richard Braun

April 7 2014

Framework

- C / Assembly
- Unix (POSIX)
- System / network programming

Goals

- Performance
- Scalability
- Debugging
- Best practices

Hardware Layout

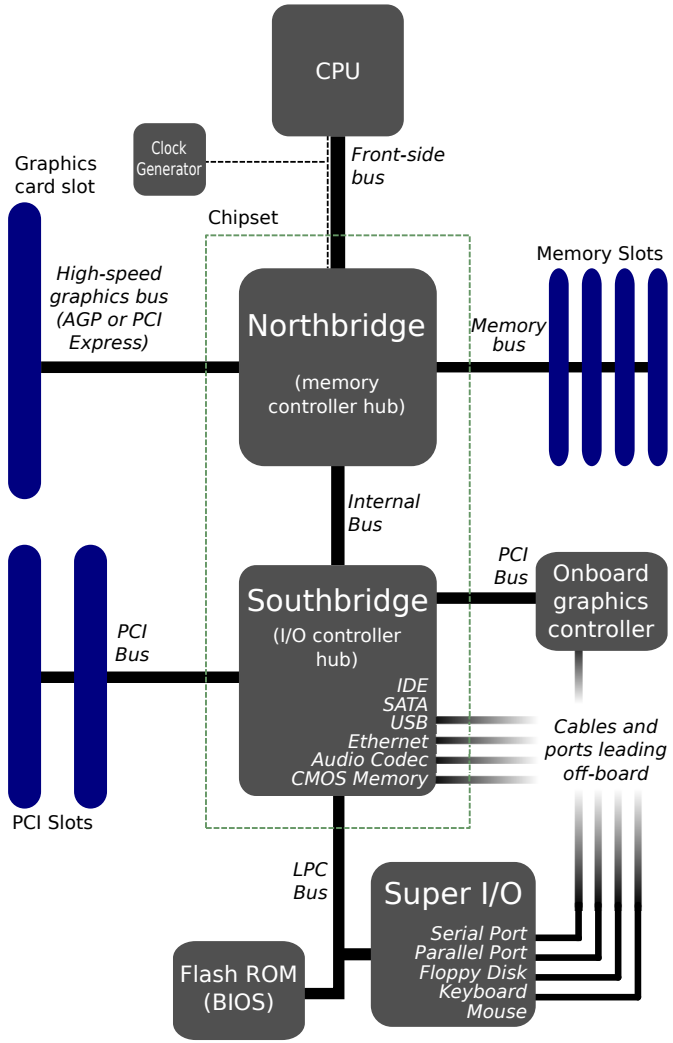


Figure: Common Intel-based hardware layout

CPU Cache Hierarchy

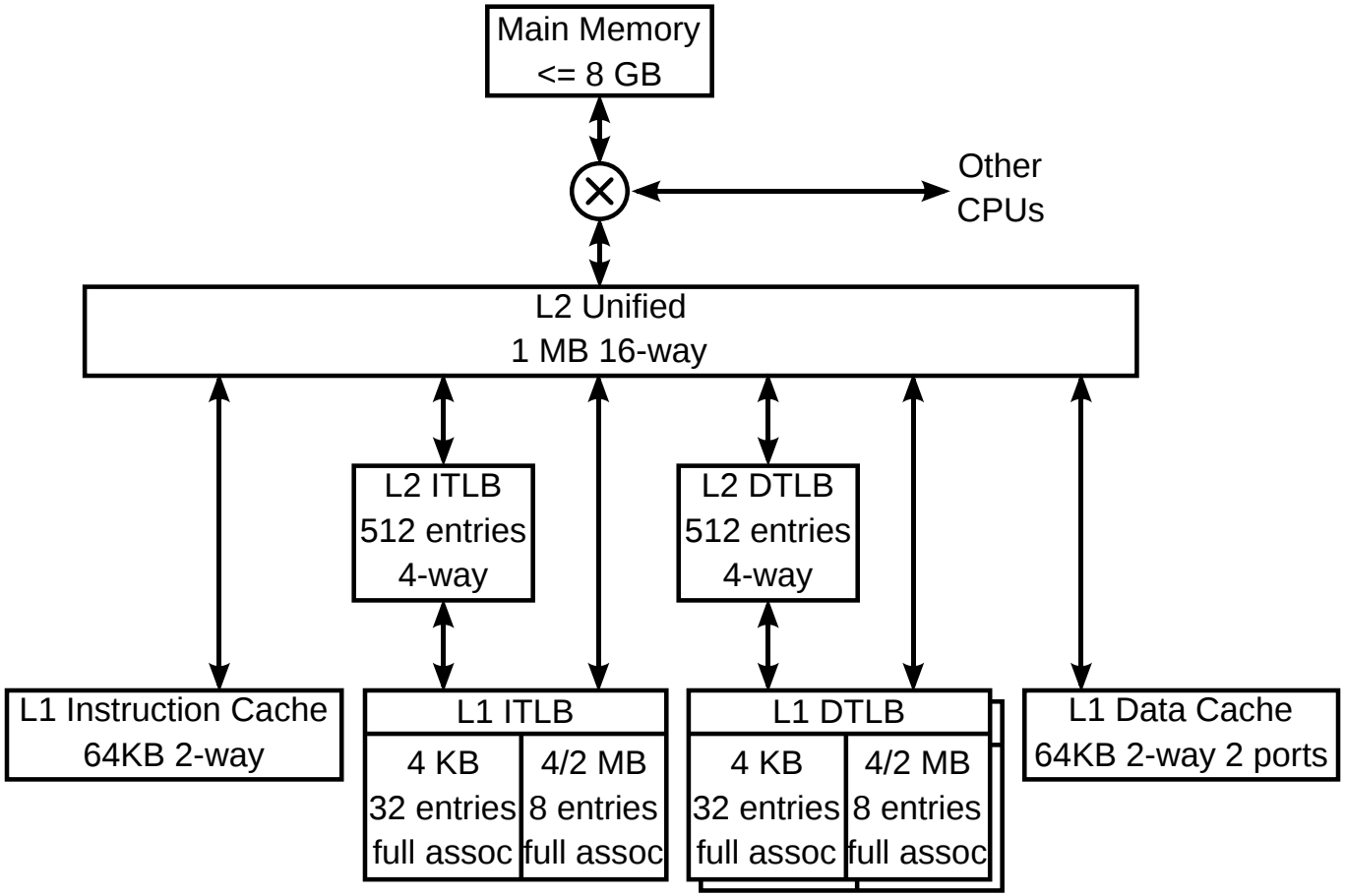
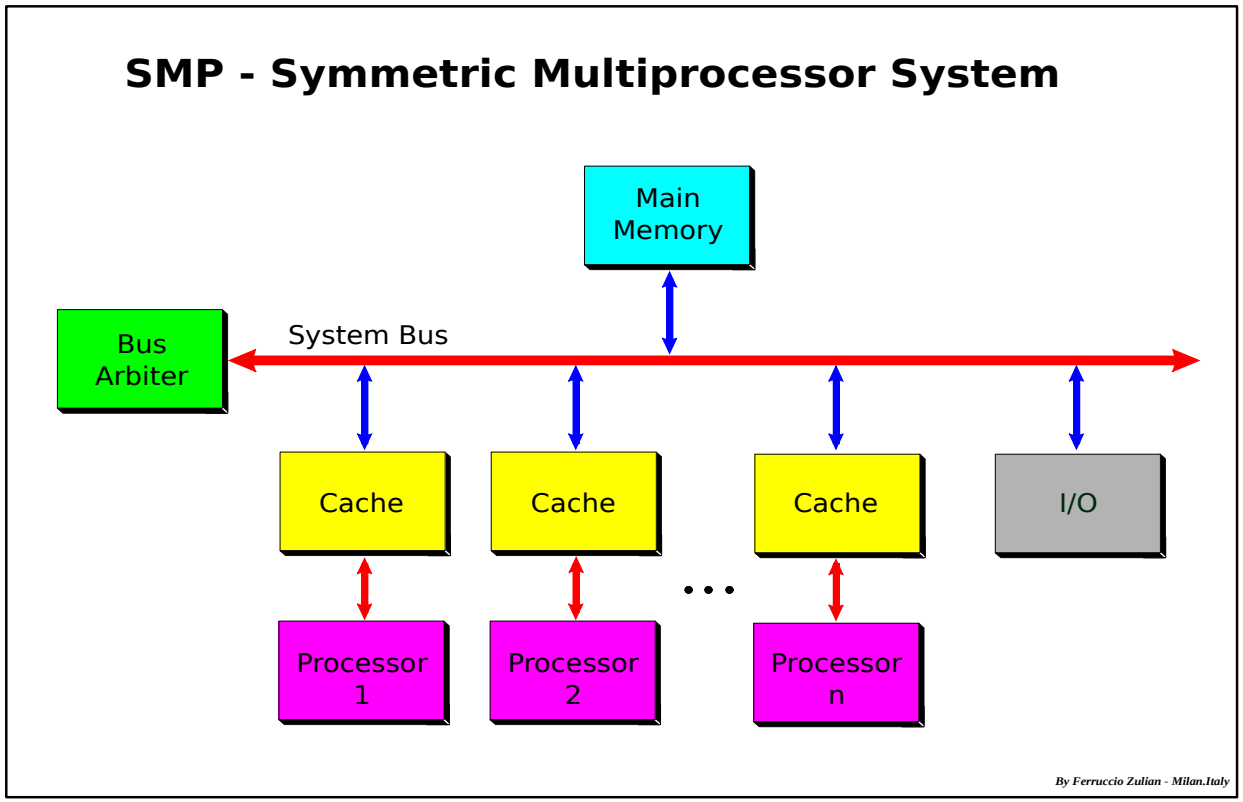


Figure: AMD Athlon 64 K8 core hierarchy

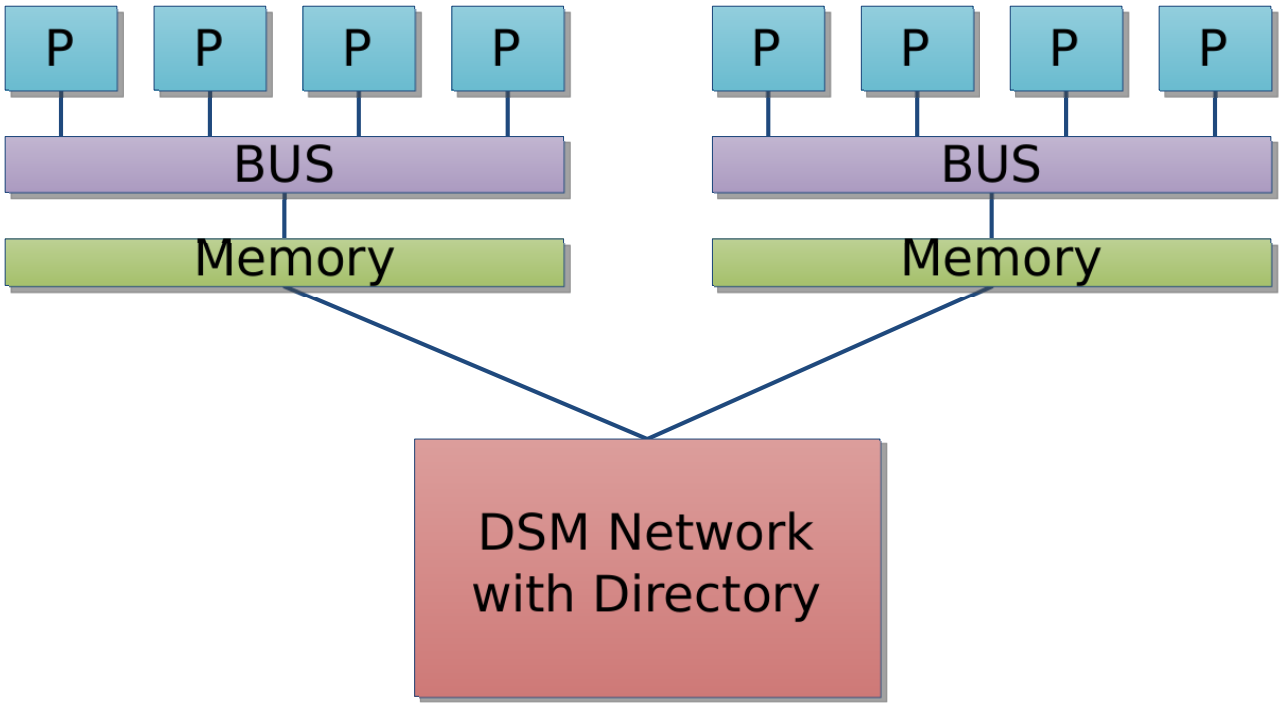
Cache Lines

- Processors transfer memory in units of cache lines
- Cache lines are stored in processor caches (L1/L2/L3)
- Data with close addresses are likely to share cache lines
- Current cache line sizes are usually 32 bytes (embedded) or 64 bytes (desktop/server)

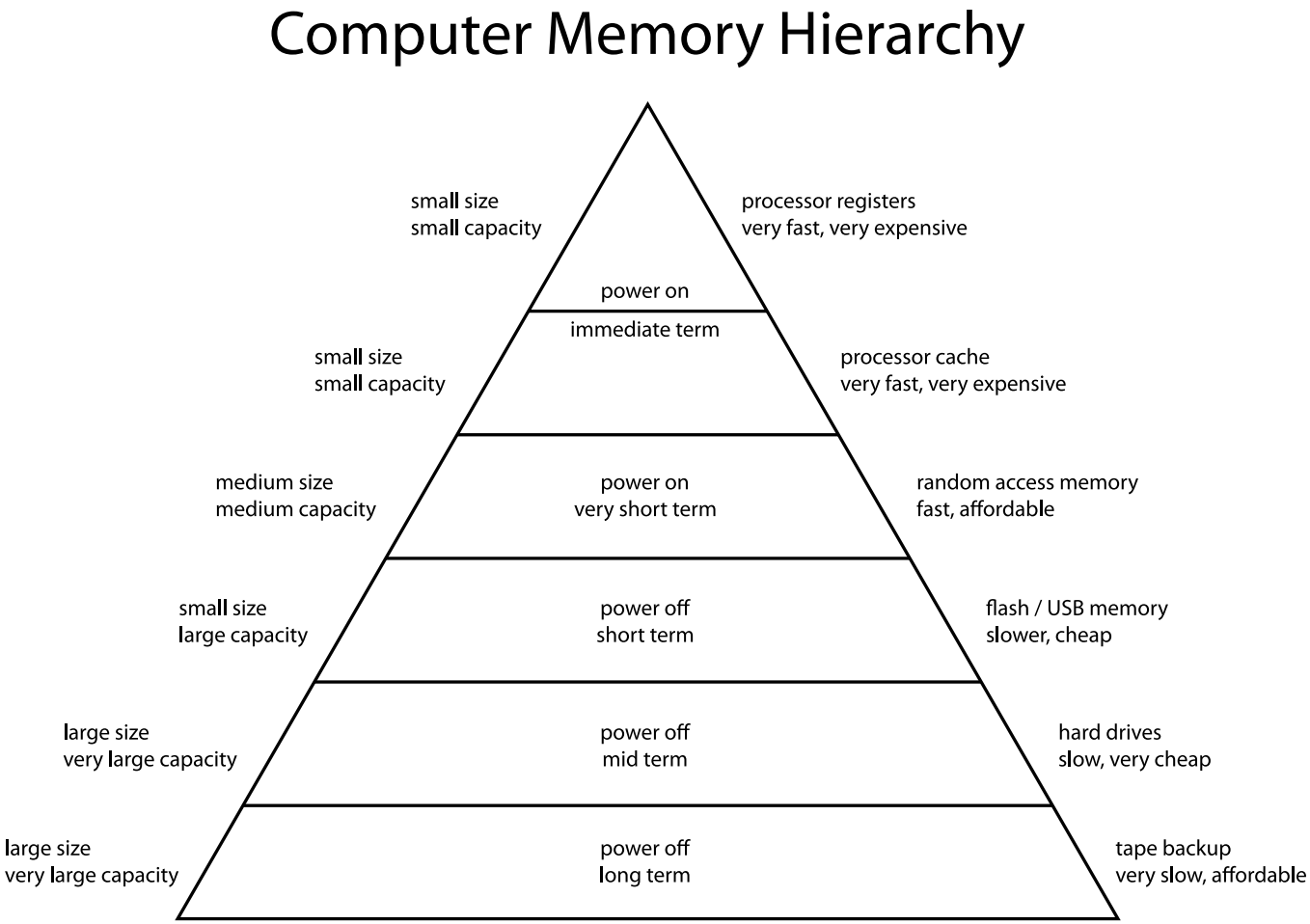
Symmetric Multiprocessor



Non-Uniform Memory Access



Memory Hierarchy



Direct Memory Access

- Transfer data without consuming processor cycles
- Transfers in burst / cycle stealing / transparent modes
- Address / data bus sharing
- Arbitration required, can cause latency indeterminacy
- Third-party DMA: dedicated DMA controllers (e.g. ISA)
- First-party DMA: bus mastering (e.g. PCI, AMBA AHB)
- Cache coherency issues

Direct Memory Access

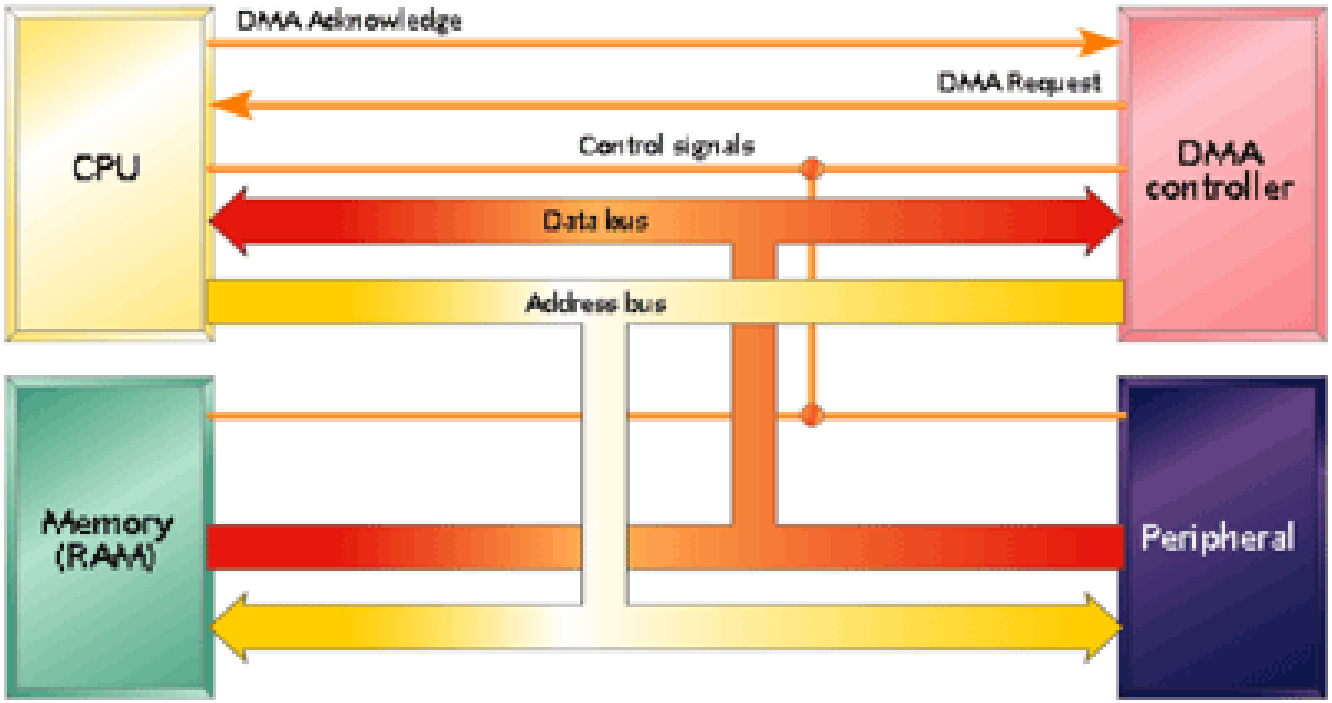


Figure: Third-party DMA

Data Storage Devices

- Magnetic storage: regular file systems (ext4, zfs, btrfs)
- Flash memory: CF/MMC/SD, USB keys, SSD (NAND based)
- Main flash types: NOR (slow, small) and NAND (faster, larger)
- Basic operation: block erasure
- Flash Translation Layer: block selection, wear leveling
- Without FTL, use specialized file systems (jffs2, ubifs)
- Reliability: use ECC memory to prevent bit flips

Virtual Memory

- Swapping: more memory than physically available
- Address spaces: isolate processes from each other
- Shared memory: reuse physical memory in many processes
- Access rights: fine-grained permissions on shared memory
- Paging: manage virtual memory in page units

Virtual Address Space

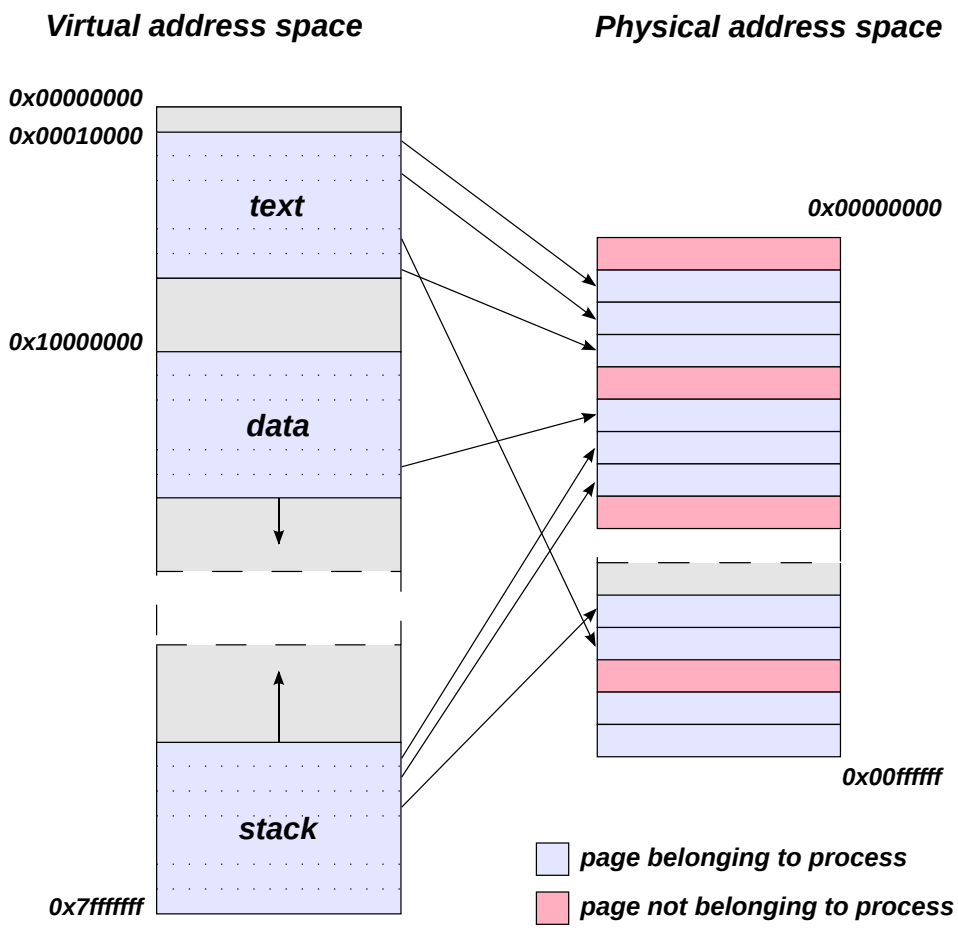


Figure: Virtual-to-physical address translations

Physical Translations

- MMU: Memory Management Unit
- Handles virtual-to-physical address translations
- Dedicated processor cache: TLB (Translation Lookaside Buffer)
- Translation unit: the page
- Modern processors can configure the page size, from 4 KiB up to 2 GiB (c.f. Linux hugetlbfs)
- Virtualization: nested page tables

Physical Translations

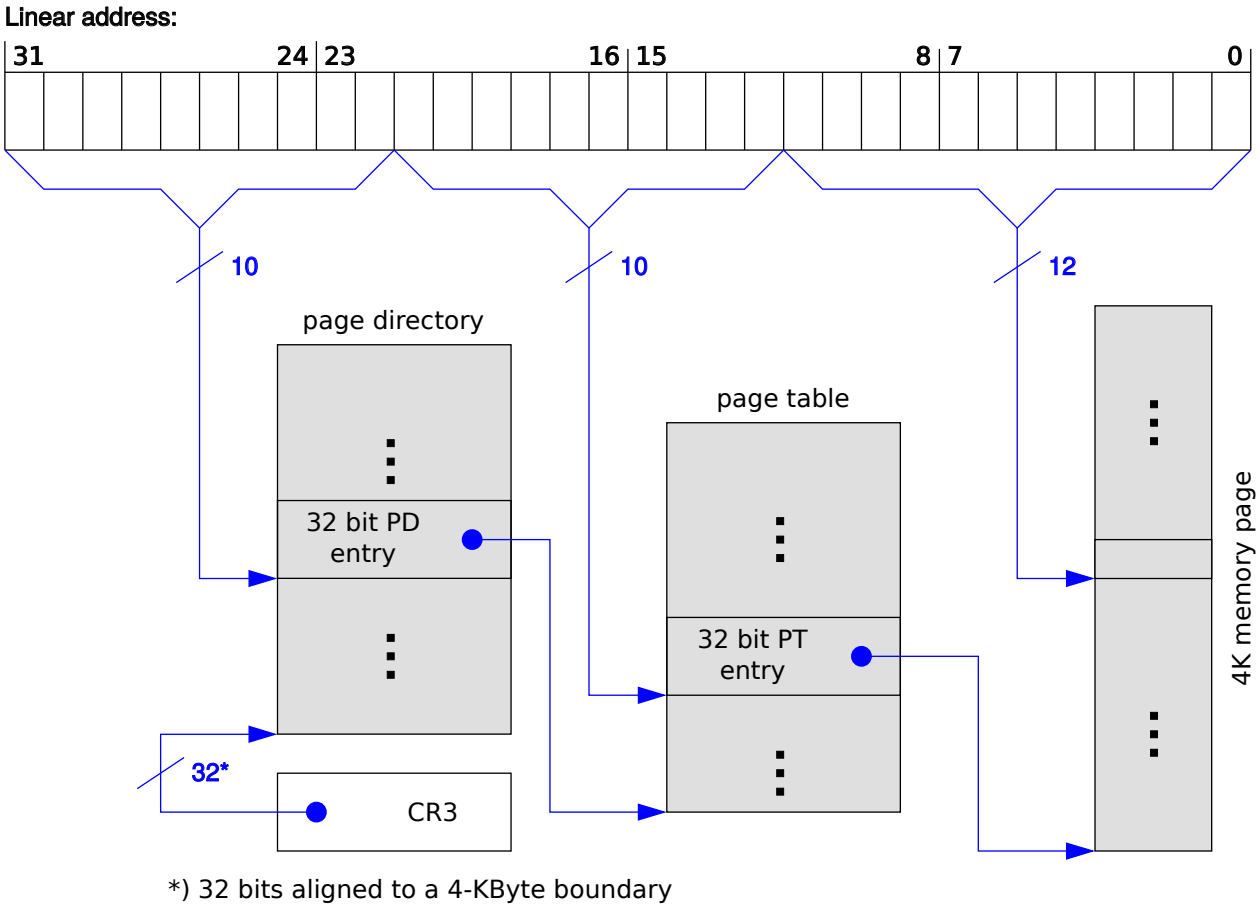


Figure: 2-level x86 page table structure

Logical Translations

- Address space: list of per process mappings
- Page: basic unit of memory, referenced in a memory object when resident (i.e. present in physical memory)
- Memory object: provides memory content (e.g. files)
- Mapping entry: associates a memory object with virtual addresses
- Page fault: exception used to implement on-demand paging
- Copy-on-write: make fork and data copies fast (zero-copy)
- Locked memory: underlying physical pages can't get evicted

Logical Translations

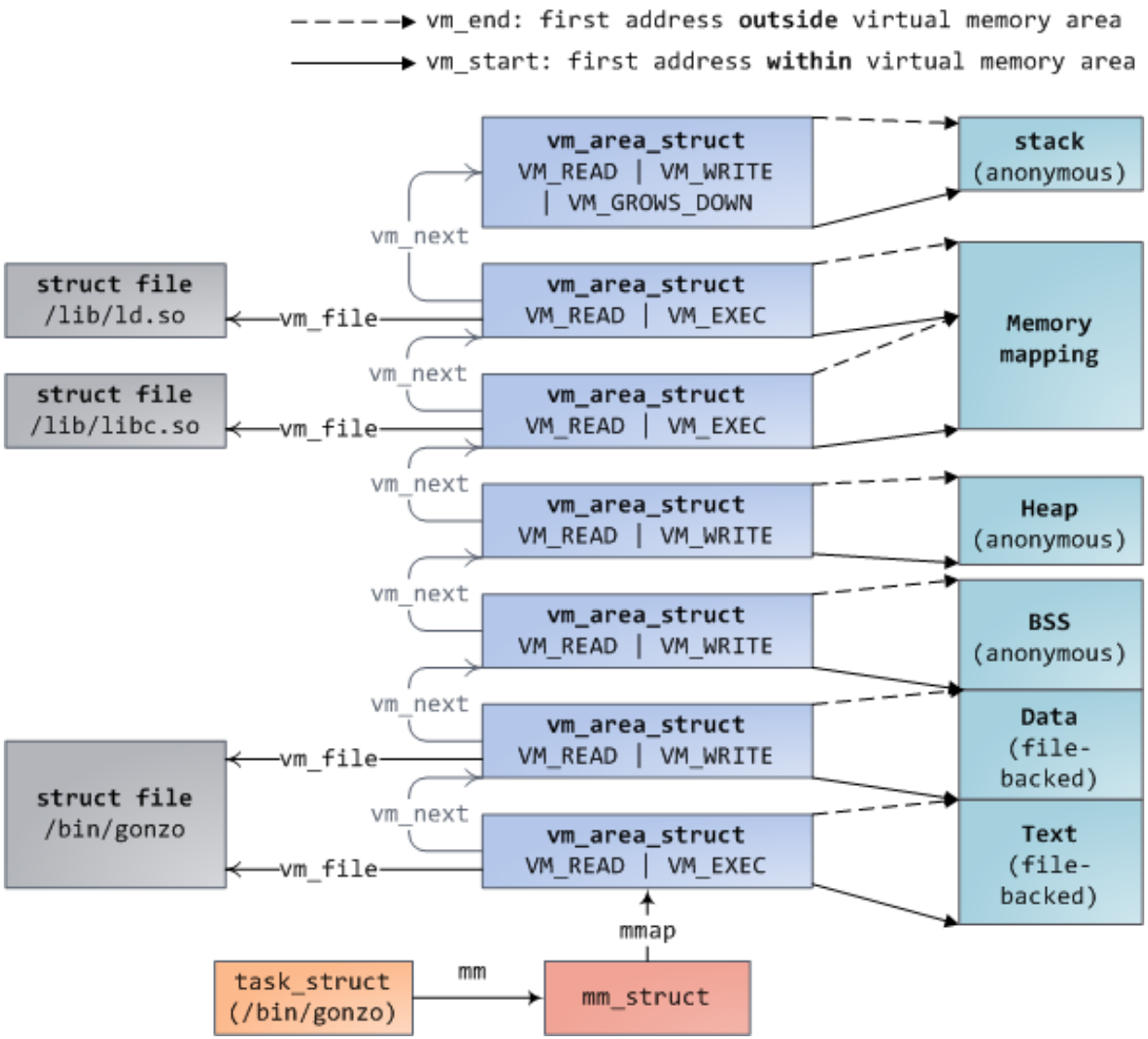


Figure: Virtual address space structure of a Linux process

Logical Translations

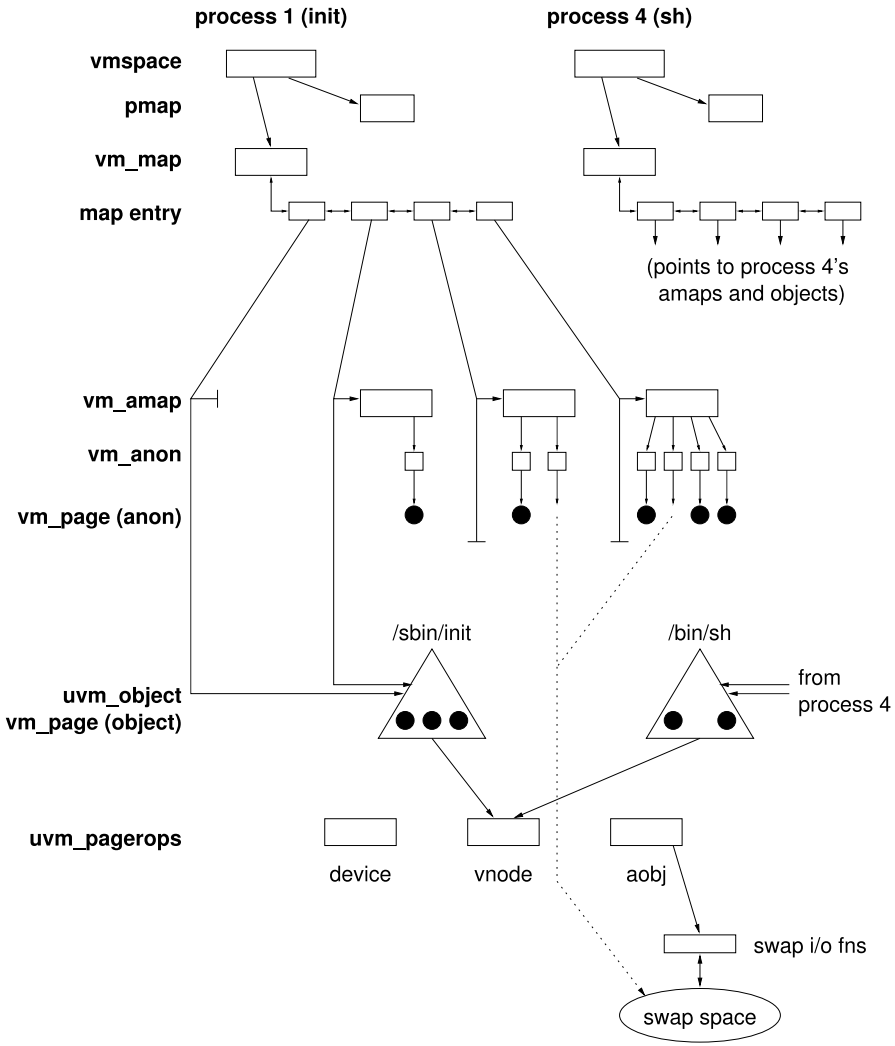


Figure: NetBSD UVM overview

The Page Cache

- "Free memory" should be thought of as "unused memory"
- Make use of that memory in the form of a data cache
- Data (file or disk) are cached in physical memory pages
- Writeback caching: flushes on eviction or after a timeout
- Eviction is triggered by memory pressure only
- LRU-like algorithms select evicted pages

Memory Mapping

- Access data (files, disks, devices) as memory
- Optimize data transfers with zero-copy techniques
- Mappings are either shared or private
- Share memory by mapping content in multiple address spaces
- 64-bits systems benefit the most
- POSIX: mmap, munmap, mprotect

Endianness

- Data order in a word
- Common orders: little endian and big endian
- Big endian: 1234 ("natural" order)
- Big endian is the standard "network byte order"
- Little endian: 4321 ("reversed" order)
- Little endian allows easy variable-size operations
- POSIX: htonl, htons, ntohl, ntohs convert between network and host byte orders

Endianness

```
struct iphdr
  {
#if __BYTE_ORDER == __LITTLE_ENDIAN
    unsigned int ihl:4;
    unsigned int version:4;
#elif __BYTE_ORDER == __BIG_ENDIAN
    unsigned int version:4;
    unsigned int ihl:4;
#else
# error "Please fix <bits/endian.h>"
#endif
    u_int8_t tos;
    u_int16_t tot_len;
    u_int16_t id;
    u_int16_t frag_off;
    u_int8_t ttl;
    u_int8_t protocol;
    u_int16_t check;
    u_int32_t saddr;
    u_int32_t daddr;
    /*The options start here. */
  };
```

Figure: Definition of the IP header in the GNU C library

Alignment

- Determine where data start in memory
- Power-of-two value
- Architecture can forbid unaligned memory accesses
- When allowed, they're slower (still faster than a cache miss)
- Locality of reference: pack data of the same working set close to one another so they're loaded in common cache lines
- Common practice: most frequently accessed members in a structure come first
- The alignment of a structure is the highest alignment of its members
- Usually requires implementation-specific compiler support for advanced control, e.g. __attribute__((aligned(value)))

Alignment

```
$ pahole x15
...
struct spinlock {
        unsigned int               locked;               /*     0     4 */

        /* size: 4, cachelines: 1, members: 1 */
        /* last cacheline: 4 bytes */
};
...
struct list {
        struct list *              prev;                 /*     0     8 */
        struct list *              next;                 /*     8     8 */

        /* size: 16, cachelines: 1, members: 2 */
        /* last cacheline: 16 bytes */
};
...
struct task {
        struct spinlock            lock;                 /*     0     4 */

        /* XXX 4 bytes hole, try to pack */

        struct list                node;                 /*     8    16 */
        struct list                threads;              /*    24    16 */
        struct vm_map *            map;                  /*    40     8 */
        char                       name[32];             /*    48    32 */
        /* --- cacheline 1 boundary (64 bytes) was 16 bytes ago --- */

        /* size: 80, cachelines: 2, members: 5 */
        /* sum members: 76, holes: 1, sum holes: 4 */
        /* last cacheline: 16 bytes */
};
...
```

Data Structures

- Parameters: complexity, locality of reference, overhead
- Common structures: arrays, linked lists, hash tables, balanced trees
- Modern structures: tries (radix trees, Judy arrays, etc...)

Data Structures

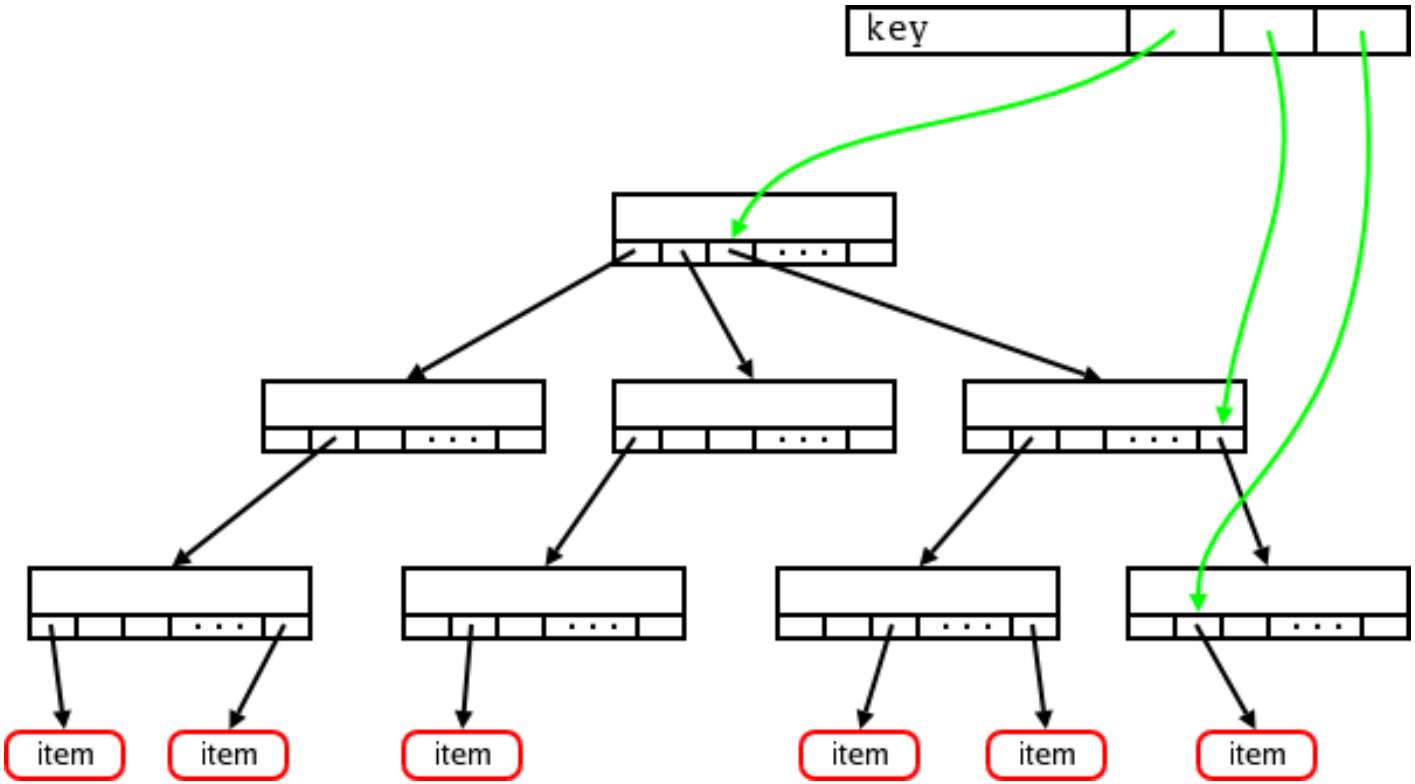


Figure: Internal structure of a radix tree

Type Punning

- Access the same memory through differently typed pointers
- Strict aliasing: two pointers of different type aren't allowed to refer to the same memory (introduced in C99)
- Cast: use only when you know what you're doing (may silence important warnings, breaks strict aliasing)
- Cast: mind the expected alignment of the underlying memory, e.g. casting `char[]` to `struct x *` may cause bus errors)
- Union pointer: the clean way (but implementation defined)

Access Control

- Bad memory access control is a major source of bugs
- Synchronize access to objects (e.g. with locks)
- Compiler barriers: prevent code reordering
- Memory barriers: prevent memory access reordering
- Release on no-user guarantee (e.g. reference counters)
- Garbage collection: automatic release once unreferenced (but mind tricky situations producing stale references such as circular references)
- Advanced techniques: lock-free and wait-free algorithms, batched reference counting (e.g. RCU)

Real Time

- Main constraints: no unexpected latency
- Reserve all resources prior to execution
- Disabling swap is not enough ! (doesn't prevent page faults)
- Lock memory to prevent fetching data from disks
- Bind to processor to prevent migration (and in turn, faults on another processor)
- POSIX: locking memory prevents pageins, not page faults, i.e. it may be required to manually access pages to fault them in

Device Memory

- Historic way: volatile pointers
- Problem: restricted by C specification
- Device memory should be mapped uncached
- Use specialized kernel accessors (for example `io_remap_page_range` on Linux)

C11

- New C Standard
- New keywords and functions
- Alignment: _Alignas, _Alignof, aligned_alloc
- Multithreading: _Thread_local, _Atomic
- Analyzability

Conclusion

Further reading :

- What Every Programmer Should Know About Memory by Ulrich Drepper
- Memory as a Programming Concept in C and C++, Frantisek Franek
- The Design and Implementation of the FreeBSD Operating System, Marshall Kirk McKusick and George V. Neville-Neil
- Memory Barriers: a Hardware View for Software Hackers, Paul E. Mckenney