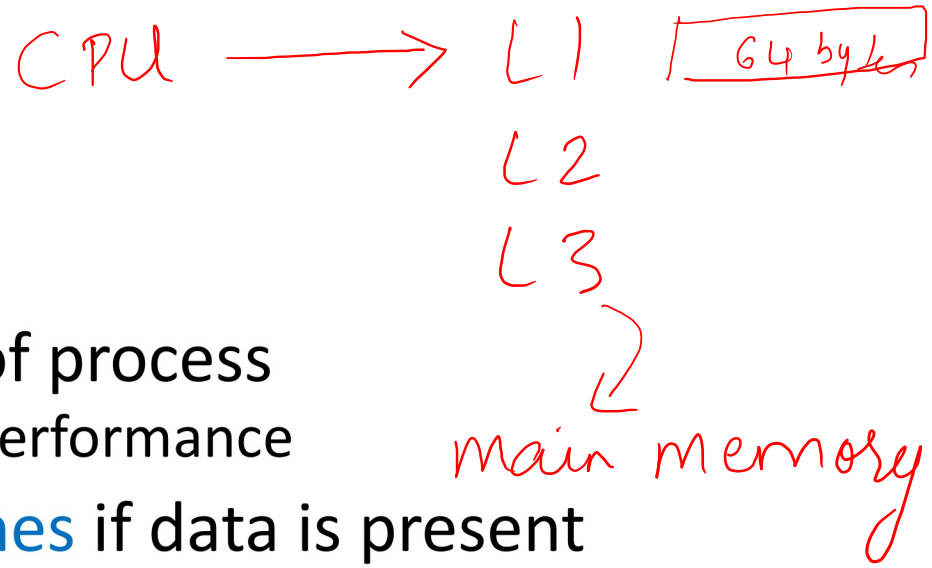
Design and Engineering of Computer Systems

Lecture 15: Optimizing memory access

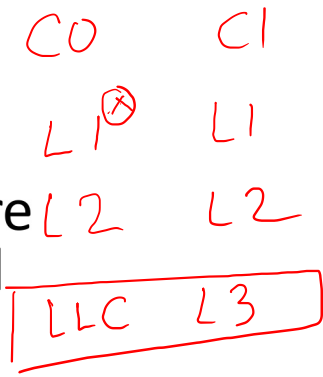
Mythili Vutukuru

IIT Bombay

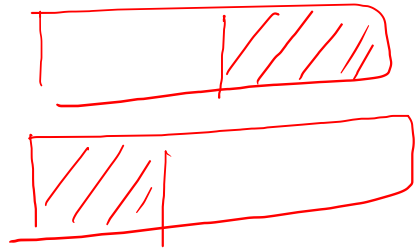
Memory access: CPU caches

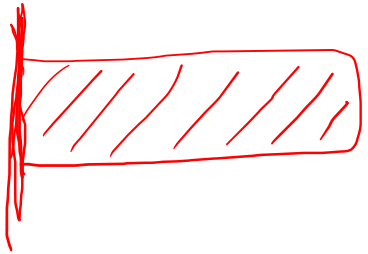


- CPU fetches instructions/data from memory of process
  - Faster memory access implies faster application performance
- First step in a memory access: check CPU caches if data is present
  - CPU caches store recently accessed memory in 64 byte cache lines
  - Uses locality of reference to avoid expensive main memory access
- Multiple levels of cache, some private, some common across cores
  - Memory location is cached in the private cache of one core C0, another core C1 also wishes to access the same memory contents → cache line is shared across cores via cache coherence mechanism
  - Cache coherence protocol ensures consistent view of memory across cores
  - But cache coherence mechanisms add overhead to memory access



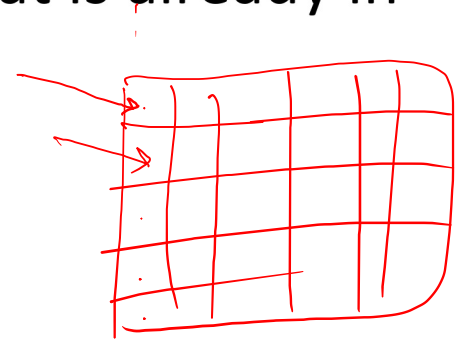
Optimizing cache usage (1)



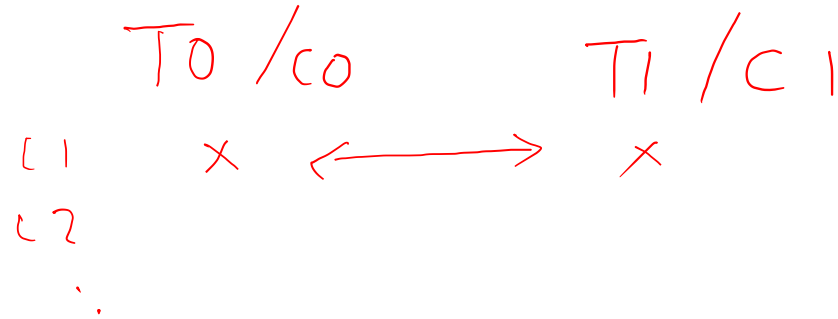


- Programmer can optimize code to maximize cache hit rates
- Align data structures to cache lines using language library primitives or compiler hints
- Store frequently accessed variables together in the same 64 byte cache line

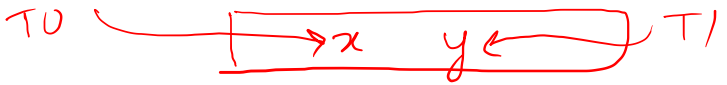
- Write code such that working set size (frequently accessed code sections or data structures) fit in CPU caches
- Write code to increase locality of reference (access data that is already in cache as far as possible)
  - Example: access matrix along rows rather than along columns
  - Example: merge two for-loops that loop over same array



Optimizing cache usage (2)



- When accessing data from multiple cores, avoid cross-core cache coherence traffic to make cache access faster
- Threads of program running on separate cores should access data in separate cache lines as far as possible
  - True sharing: two threads read same memory address from separate cores
  - False sharing: two threads read separate memory addresses, but both locations are on the same cache line
  - Both cause cache line to bounce across cores



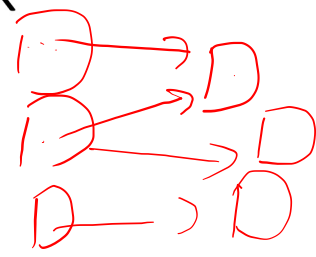
- Avoid shared data and lock contention between threads as far as possible
  - Shared lock variable accessed from multiple cores, cache line bounces across cores
  - Recent research on locks which avoid sharing lock variables across cores scalable locks
  - Lock-free data structures: data structure implementations that avoid locks using clever tricks

Memory access: MMU, TLB, page fault

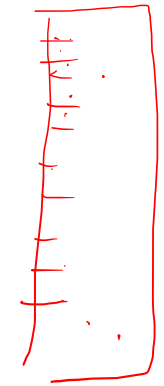
- If cache miss, CPU must fetch instructions/data from main memory
  - MMU checks TLB for virtual to physical address mapping
  - If TLB miss, MMU walks page table to translate address
  - Main memory is accessed using computed physical address
- TLB miss leads to extra memory accesses due to MMU page table walk
  - Optimizing TLB hit rate crucial, especially with multi-level page tables
- How to improve TLB hit rate?
  - Limit working set size in memory, use few memory pages at any point of time
  - Huge pages: can use larger page size in order to have fewer page table mappings
- If OS has not allocated memory to a page, MMU traps to OS for page fault
  - Servicing page faults may require multiple disk accesses to swap space
  - Too many page faults: thrashing, too much time wasted in swapping to/from disk
  - Avoid thrashing by limiting working set size, clearing up unnecessary memory

4KB

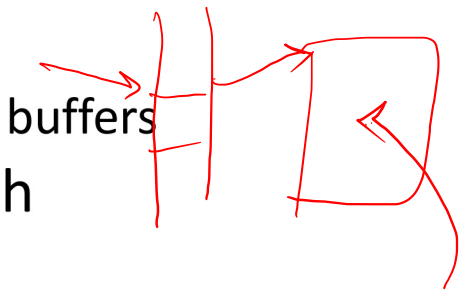
Zombie



Tips to optimize memory allocation/access

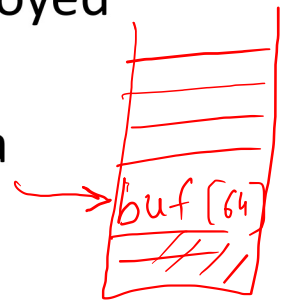


- DRAM allows random access of memory (jump to any address), but sequential access of memory is better for performance
  - CPU prefetcher predicts which memory will be accessed next (estimates stride length of access) and fetches it into cache
- Sequential access of disk data is better for traditional hard disks
  - Spinning magnetic disk has extra delays for random access
- Pre-allocation of memory is better than dynamic allocation via malloc
  - General purpose malloc that does variable sized allocation can be slow
- Custom memory allocators better than general purpose allocator in some cases
  - Slab allocators are better when dynamic memory allocation is in a few fixed sizes
  - Store data in memory-mapped anonymous pages instead of heap
- Avoid copying memory contents unnecessarily
  - Memory mapping a file avoids copying file data from kernel memory to user buffers
- Later in the course: how to measure performance and identify which optimizations are useful and which are not, via profiling code



Common memory-related bugs

- Memory leak = memory is allocated via “malloc” or “new”, but not explicitly freed by programmer via “free”/”delete”
  - Wastes memory space on heap
- Dangling pointers = pointers to memory chunks that have been freed up
  - Pointers to malloc memory after freeing it up, or pointers to stack variables after function returns
  - Accessing such pointers may lead to segmentation fault or incorrect behavior
- Avoid such errors with careful programming, or use language libraries that provide automatic garbage collection via reference counting (keeping track of pointers to allocated memory chunks)
  - Example: shared_ptr in C++ is automatically deallocated if all pointers are destroyed
- Buffer overflow = overwrite stack content and corrupt stack
  - Allocate buf[64] on stack, but read string longer than 64 bytes, overwriting data
- Other errors: misunderstanding pointers, not initializing memory, ..



Summary

- In this lecture:
  - How to make memory access faster
  - How to avoid bugs around memory accesses
- Programming exercise: explore smart pointers (shared_ptr) and other such reference counted pointers in C++